39th European Conference on Object-Oriented Programming

ECOOP 2025, June 30–July 2, 2025, Bergen, Norway

Edited by

Jonathan Aldrich

Alexandra Silva

Editors

Jonathan Aldrich
Carnegie Mellon University, Pittsburgh, PA, USA
jonathan.aldrich@cs.cmu.edu

Alexandra Silva
Cornell University, Ithaca, NY, USA
alexandra.silva@cornell.edu

ACM Classification 2012
Software and its engineering → General programming languages

ISBN 978-3-95977-373-7

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-373-7.

Publication date
June, 2025

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

License
This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0): https://creativecommons.org/licenses/by/4.0/legalcode.
In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:
- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.ECOOP.2025.0

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Christel Baier (TU Dresden, DE)
- Roberto Di Cosmo (Inria and Université Paris Cité, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Holger Hermanns (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)
- Daniel Král' (Leipzig University, DE and Max Planck Institute for Mathematics in the Sciences, Leipzig, DE)
- Sławomir Lasota (University of Warsaw, PL)
- Meena Mahajan (Institute of Mathematical Sciences, Chennai, IN – Chair)
- Chih-Hao Luke Ong (Nanyang Technological University, SG)
- Eva Rotenberg (Technical University of Denmark, Lyngby, DK)
- Pierre Senellart (ENS, Université PSL, Paris, France)
- Alexandra Silva (Cornell University, Ithaca, US)

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Preface
Jonathan Aldrich and Alexandra Silva 0:ix–0:x

Message from the Artifact Evaluation Chairs
Karine Even-Mendoza and Raphaël Monat 0:xi

Message from the AITO President
Davide Ancona 0:xiii

List of Authors
..... 0:xv–0:xviii

Regular Papers

A Theory of (Linear-Time) Timed Monitors
Mouloud Amara, Giovanni Bernardi, Mohammed Aristide Foughali, and Adrian Francalanza 1:1–1:30

The Algebra of Patterns
David Binder and Lean Ermantraut 2:1–2:28

Pydrofoil: Accelerating Sail-Based Instruction Set Simulators
Carl Friedrich Bolz-Tereick, Luke Panayi, Ferdia McKeogh, Tom Spink, and Martin Berger 3:1–3:31

Ensuring Convergence and Invariants Without Coordination
Dina Borrego, Nuno Preguiça, Elisa Gonzalez Boix, and Carla Ferreira 4:1–4:29

Compositional Bug Detection for Internally Unsafe Libraries: A Logical Approach to Type Unsoundness
Pedro Carrott, Sacha-Élie Ayoun, and Azalea Raad 5:1–5:28

Detecting Functionality-Specific Vulnerabilities via Retrieving Individual Functionality-Equivalent APIs in Open-Source Repositories
Tianyu Chen, Zeyu Wang, Lin Li, Ding Li, Zongyang Li, Xiaoning Chang, Pan Bian, Guangtai Liang, Qianxiang Wang, and Tao Xie 6:1–6:27

Monadic Type-And-Effect Soundness
Francesco Dagnino, Paola Giannini, and Elena Zucca 7:1–7:31

An Effectful Object Calculus
Francesco Dagnino, Paola Giannini, and Elena Zucca 8:1–8:30

Event Race Detection for Node.js Using Delay Injections
Andre Takeshi Endo and Anders Møller 9:1–9:28

Bottom-Up Synthesis of Memory Mutations with Separation Logic
Kasra Ferdowsi and Hila Peleg 10:1–10:32

Contract Usage and Evolution in Android Mobile Applications
David R. Ferreira, Alexandra Mendes, João F. Ferreira, and Carolina Carreira .. 11:1–11:30

39th European Conference on Object-Oriented Programming (ECOOP 2025).
Editors: Jonathan Aldrich and Alexandra Silva
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Automatic Goal Clone Detection in Rocq
Ali Ghanbari 12:1–12:19

FuzzFlesh: Randomised Testing of Decompilers via Control Flow Graph-Based Program Generation
Amber Gorzynski and Alastair F. Donaldson 13:1–13:26

IsaBIL: A Framework for Verifying (In)correctness of Binaries in Isabelle/HOL
Matt Griffin, Brijesh Dongol, and Azalea Raad 14:1–14:30

Spegion: Implicit and Non-Lexical Regions with Sized Allocations
Jack Hughes, Michael Vollmer, and Mark Batty 15:1–15:26

A Lightweight Method for Generating Multi-Tier JIT Compilation Virtual Machine in a Meta-Tracing Compiler Framework
Yusuke Izawa, Hidehiko Masuhara, and Carl Friedrich Bolz-Tereick 16:1–16:29

Contrasting Deadlock-Free Session Processes
Juan C. Jaramillo and Jorge A. Pérez 17:1–17:29

Practical Type-Based Taint Checking and Inference
Nima Karimipour, Kanak Das, Manu Sridharan, and Behnaz Hassanshahi 18:1–18:25

Profile-Guided Field Externalization in an Ahead-Of-Time Compiler
Sebastian Kloibhofer, Lukas Makor[1], Peter Hofer, David Leopoldseder, and Hanspeter Mössenböck 19:1–19:32

Incremental Computing by Differential Execution
Prashant Kumar, André Pacak, and Sebastian Erdweg 20:1–20:24

GSOHC: Global Synchronization Optimization in Heterogeneous Computing
Soumik Kumar Basu and Jyothi Vedurada 21:1–21:30

Quantifying Cache Side-Channel Leakage by Refining Set-Based Abstractions
Jacqueline L. Mitchell and Chao Wang 22:1–22:28

Wastrumentation: Portable WebAssembly Dynamic Analysis with Support for Intercession
Aäron Munsters, Angel Luis Scull Pupo, and Elisa Gonzalez Boix 23:1–23:29

Fair Termination of Asynchronous Binary Sessions
Luca Padovani and Gianluigi Zavattaro 24:1–24:29

Reusing Highly Optimized IR in Dynamic Compilation
Andrej Pečimúth, David Leopoldseder, and Petr Tůma 25:1–25:25

Lightweight Diagramming for Lightweight Formal Methods: A Grounded Language Design
Siddhartha Prasad, Ben Greenman, Tim Nelson, and Shriram Krishnamurthi 26:1–26:29

PoTo: A Hybrid Andersen's Points-To Analysis for Python
Ingkarat Rak-amnouykit, Ana Milanova, Guillaume Baudart, Martin Hirzel, and Julian Dolby 27:1–27:29

Reusing Caches and Invariants for Efficient and Sound Incremental Static Analysis
Mamy Razafintsialonina, David Bühler, Antoine Miné, Valentin Perrelle, and Julien Signoles 28:1–28:29

Declarative Dynamic Object Reclassification
Riccardo Sieve, Eduard Kamburjan, Ferruccio Damiani, and Einar Broch Johnsen 29:1–29:31

In-Memory Object Graph Stores
Aditya Thimmaiah, Zijian Yi, Joseph Kenis, Christopher J Rossbach, and Milos Gligoric 30:1–30:30

Multiparty Asynchronous Session Types: A Mechanised Proof of Subject Reduction
Dawit Tirore, Jesper Bengtson, and Marco Carbone 31:1–31:30

Compositional Static Value Analysis for Higher-Order Numerical Programs
Milla Valnet, Raphaël Monat, and Antoine Miné 32:1–32:29

Mono Types – First-Class Containers for Datalog
Runqing Xu, David Klopp, and Sebastian Erdweg 33:1–33:26

Taming and Dissecting Recursions Through Interprocedural Weak Topological Ordering
Jiawei Yang, Xiao Cheng, Bor-Yuh Evan Chang, Xiapu Luo, and Yulei Sui 34:1–34:31

Chain of Grounded Objectives: Concise Goal-Oriented Prompting for Code Generation
Sangyeop Yeo, Seung-Won Hwang, and Yu-Seung Ma 35:1–35:25

Efficient Neural Network Verification via Order Leading Exploration of Branch-and-Bound Trees
Guanqin Zhang, Kota Fukuda, Zhenya Zhang, H.M.N. Dilum Bandara, Shiping Chen, Jianjun Zhao, and Yulei Sui 36:1–36:29

Experience Papers

RacerF: Lightweight Static Data Race Detection for C Code
Tomáš Dacík and Tomáš Vojnar 37:1–37:19

Type-Safe and Portable Support for Packed Data
Arthur Jamet and Michael Vollmer 38:1–38:19

WebGlitch: A Randomised Testing Tool for the WebGPU API
Matthew K. L. Wong and Alastair F. Donaldson 39:1–39:26

Replication Paper

Scaling Up: Revisiting Mining Android Sandboxes at Scale for Malware Classification
Francisco Handrick Tomaz da Costa, Ismael Medeiros, Leandro Oliveira, João Calássio, Rodrigo Bonifácio, Krishna Narasimhan, Mira Mezini, and Márcio Ribeiro 40:1–40:26

Pearl/Brave New Ideas

Shouting at Memory: Where Did My Write Go?
Vasileios Klimis 41:1–41:26

Contract Systems Need Domain-Specific Notations
Cameron Moy, Ryan Jung, and Matthias Felleisen 42:1–42:24

Preface

Started in 1987, ECOOP is Europe's oldest programming conference, welcoming papers on all practical and theoretical investigations of programming languages, systems, and environments that provide innovative solutions to real problems as well as evaluations of existing solutions. Papers were submitted to one of four categories: *Research* for papers that advance the state of the art in programming; *Replication* for empirical evaluations that reconstruct a published experiment in a different context in order to validate the results of that earlier work; *Experience* for applications of known techniques in practice; and *Pearl/Brave New Idea* for papers that either explain a known idea in an elegant way or unconventional papers introducing ideas that may take some time to substantiate. The chairs thank the Program Committee for their dedication to ensuring a quality program and providing constructive feedback to authors: Aaron Bembenek, Albert Rubio, Carla Ferreira, Daniele Nantes-Sobrinho, Di Wang, Einar Broch Johnsen, Elisa Gonzalez Boix, Elvira Albert, Farzaneh Derakhshan, Federico Olmedo, Jingling Xue, João F. Ferreira, Jules Jacobs, Lingming Zhang, Luis Caires, Max Willsey, Michael Vollmer, Mira Mezini, Niki Vazou, Pascal Weisenburger, Peter Thiemann, Ping Hou, Robbert Krebbers, Sebastian Erdweg, Serena Elisa Ponta, Sophia Drossopoulou, Sung-Shik Jongmans, Tobias Wrigstad, Wenjia Ye, Werner Dietl. We are also grateful for the help from a number of external experts who helped us ensure every paper got a high-confidence review: Adam Barwell, Andreas Lööw, Andrew Lenharth, Ashish Mishra, Ben Titzer, Daniel Ramos, David Naumann, David Richter, Emilio Tuosto, Guillermo Henrique Botelho Guerra, Jérôme Feret, Jesus Correas, Joshua Sunshine, Justin Hsu, Marco Guarnieri, Mike Dodds, Pablo Gordillo, Paulo Torrens, Roberto Giacobazzi, Samir Genaim, Stefan Marr, Yu David Liu.

This year, we continued the review process introduced in 2024, in which all papers were rated on each of the following criteria:

- **Soundness:** How well the paper's contributions are supported by rigorous application of appropriate research methods;
- **Significance:** The extent to which the paper's contributions are novel, original, and important, with respect to the existing body of knowledge;
- **Presentation:** Whether the paper's quality of writing meets the high standards of ECOOP.

After the author response and reviewer discussions, papers were accepted if the PC decided that the paper meets our high bar for Soundness and Presentation, and if at least one reviewer judges the paper to meet the bar for Significance. The goal of this process is to ensure quality of writing and confidence in results, while assuming that if one reviewer finds the paper to be significant then there will be readers who do so as well.

Overall, 102 papers were submitted in the first and second rounds. In the end 42 papers were accepted, in some cases after a round of revisions between the two rounds or a final round of checking for papers that initially received a conditional accept rating. As is common with journals, the ability to resubmit improved versions of a paper allowed the conference to accept a larger percentage of papers overall than in prior editions of ECOOP, while maintaining a high quality thereshold.

39th European Conference on Object-Oriented Programming (ECOOP 2025).
Editors: Jonathan Aldrich and Alexandra Silva
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We hope that future chairs will continue to experiment with more, and perhaps, different innovations that will enrich the ECOOP community further.

Finally, we would like to thank the local organizers, in particular the ECOOP General Chair, Volker Stolz, and the Local Organization Chair, Violet Ka I Pun, for all the work they did to put together the conference, the AITO executive committee, and the Dagstuhl LIPIcs team. For those who have edited a volume supported by the team led by Michael Wagner will know how responsive and helpful they are, we are extremely grateful for the service they provide, as we recognize this is very valuable for our community and the mission of keeping science open to everyone.

Jonathan Aldrich
Program Committee Co-chair
Carnegie Mellon University

Alexandra Silva
Program Committee Co-chair
Cornell University

Message from the AEC chairs

ECOOP has a long-standing tradition of offering artifact evaluation dating back to 2013. This edition's artifact evaluation exceptionally focused on performing artifact evaluation of accepted papers only, contrary to the ECOOP tradition of evaluating artifacts at the same time as papers are reviewed.

In order to ensure a thorough and efficient artifact evaluation process, we recruited a large and diverse artifact evaluation committee that included a total of 55 artifact reviewers. The artifact submissions were due a few days after papers were accepted at ECOOP. Due to aforementioned change in the artifact evaluation process, we received a total of 21 submissions this year, down from 64 last year. Due to tight calendar constraints of this year's artifact evaluation, and to address potential hardware shortcomings, each artifact has been assigned four reviewers instead of the traditional three. The review phase started with a kick-the-tires review and author response, during which authors had the opportunity to clarify or address technical issues with their submissions. Full reviews have been provided a few weeks later.

We followed ACM's badging policy[1] since 2023. Details about the evaluation process and results can be found in the ECOOP 2025 DARTS proceedings.

The smooth and thorough artifact evaluation process would have not been possible without the members of the committee, who handled the artifact review workload with great dedication. We would like to thank them for their valuable work, feedback to authors and the inspiring discussions! We would also like to thank the ECOOP 2025 program committee chairs Jonathan Aldrich and Alexandra Silva for the pleasant and productive interactions over the coordination of the paper and artifact review processes, and the Dagstuhl Publishing team for their proactive and highly responsive assistance during the preparation of this DARTS volume.

Karine Even-Mendoza
King's College London

Raphaël Monat
Inria Lille & University of Lille

Yannick Zakowski
Inria Paris

[1] `https://www.acm.org/publications/policies/artifact-review-and-badging-current`

39th European Conference on Object-Oriented Programming (ECOOP 2025).
Editors: Jonathan Aldrich and Alexandra Silva
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Message from the AITO president

After 21 years, ECOOP returns to Norway, this time hosted by the Western Norway University of Applied Sciences in Bergen. I am delighted to welcome all ECOOP attendees to this vibrant and fascinating city.

Norway holds a special place in the history of ECOOP. One of the early editions of the conference was held here in 1988. More importantly, Norway is the home of Ole-Johan Dahl and Kristen Nygaard, the acknowledged co-inventors of Object-Oriented Programming, who were honored with the “ACM A. M. Turing Award" in 2001 for their groundbreaking contributions.

This year, for the first time, the “ACM SIGPLAN International Conference on Generative Programming: Concepts & Experiences" (GPCE) is co-located with ECOOP. I am confident this will establish a long-standing collaboration between the two conferences and their closely connected communities, which share many researchers and interests.

The ECOOP 2025 organizers, together with the GPCE team, have done an outstanding job putting together a rich and diverse program, featuring seven workshops and a doctoral symposium. Many thanks to the ECOOP General Chair, Volker Stolz; the Local Organization Chair, Violet Ka I Pun; the Program Committee Co-Chairs, Alexandra Silva and Jonathan Aldrich; and all other members of the organizing committee for their dedication and contributions.

I look forward to an exciting week of conferences and workshops, stimulating presentations, fruitful personal interactions, and inspiring keynotes—including the talks by the two 2025 Dahl-Nygaard Prize winners. Enjoy the conference and your time in Bergen!

Davide Ancona
AITO President

39th European Conference on Object-Oriented Programming (ECOOP 2025).
Editors: Jonathan Aldrich and Alexandra Silva
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

List of Authors

Mouloud Amara (1)
Université Paris Cité, CNRS, IRIF, F-75013 Paris, France

Sacha-Élie Ayoun (5)
Imperial College London, UK

H.M.N. Dilum Bandara (36)
CSIRO's Data61, Sydney, Australia; University of New South Wales, Sydney, Australia

Mark Batty (15)
School of Computing, University of Kent, Canterbury, UK

Guillaume Baudart (27)
Université Paris Cité, Inria, CNRS, IRIF, France

Jesper Bengtson (31)
IT University of Copenhagen, Denmark

Martin Berger (3)
University of Sussex, Brighton, UK; Montanarius Ltd, London, UK

Giovanni Bernardi (1)
Université Paris Cité, CNRS, IRIF, F-75013 Paris, France

Pan Bian (6)
Huawei Cloud Computing Technologies Co., Ltd., Guiyang, China

David Binder (2)
University of Kent, Canterbury, UK

Carl Friedrich Bolz-Tereick (3, 16)
Heinrich-Heine-Universität Düsseldorf, Germany

Rodrigo Bonifácio (40)
Computer Science Department, University of Brasília, Brazil

Dina Borrego (4)
NOVA School of Science and Technology, Caparica, Portugal; Vrije Universiteit Brussel, Belgium

David Bühler (28)
Université Paris-Saclay, CEA, List, Palaiseau, France

João Calássio (40)
Computer Science Department, University of Brasília, Brazil

Marco Carbone (31)
IT University of Copenhagen, Denmark

Carolina Carreira (11)
Carnegie Mellon University, Pittsburgh, PA, USA; INESC-ID & IST, University of Lisbon, Portugal

Pedro Carrott (5)
Imperial College London, UK

Bor-Yuh Evan Chang (34)
University of Colorado Boulder, Boulder, CO, USA; Amazon, Boulder, CO, USA

Xiaoning Chang (6)
Huawei Cloud Computing Technologies Co., Ltd., Guiyang, China

Shiping Chen (36)
CSIRO's Data61, Sydney, Australia; University of New South Wales, Sydney, Australia

Tianyu Chen (6)
Peking University, Beijing, China

Xiao Cheng (34)
University of New South Wales, Sydney, Australia

Tomáš Dacík (37)
Faculty of Information Technology, Brno University of Technology, Czech Republic

Francesco Dagnino (7, 8)
DIBRIS, Università di Genova, Italy

Ferruccio Damiani (29)
University of Turin, Italy

Kanak Das (18)
University of California, Riverside, CA, USA

Julian Dolby (27)
IBM Research, Yorktown Heights, NY, USA

Alastair F. Donaldson (13, 39)
Imperial College London, UK

Brijesh Dongol (14)
University of Surrey, Guildford, UK

Andre Takeshi Endo (9)
Federal University of São Carlos, Brazil

Sebastian Erdweg (20, 33)
JGU Mainz, Germany

39th European Conference on Object-Oriented Programming (ECOOP 2025).
Editors: Jonathan Aldrich and Alexandra Silva
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Lean Ermantraut (2)
Radboud University, Nijmegen, The Netherlands

Matthias Felleisen (42)
PLT @ Northeastern University, Boston, MA, USA

Kasra Ferdowsi (10)
University of California San Diego, CA, USA

Carla Ferreira (4)
NOVA School of Science and Technology, Caparica, Portugal

David R. Ferreira (11)
Faculty of Engineering, University of Porto, Portugal

João F. Ferreira (11)
INESC-ID & IST, University of Lisbon, Portugal

Mohammed Aristide Foughali (1)
Université Paris Cité, CNRS, IRIF, F-75013 Paris, France

Adrian Francalanza (1)
Dept. of Computer Science, ICT, University of Malta, Msida, Malta

Kota Fukuda (36)
Kyushu University, Fukuoka, Japan

Ali Ghanbari (12)
Auburn University, Al, USA

Paola Giannini (7, 8)
DiSSTE, Università del Piemonte Orientale, Vercelli, Italy

Milos Gligoric (30)
The University of Texas at Austin, TX, USA

Elisa Gonzalez Boix (4, 23)
Vrije Universiteit Brussel, Belgium

Amber Gorzynski (13)
Imperial College London, UK

Ben Greenman (26)
University of Utah, Salt Lake City, UT, USA

Matt Griffin (14)
University of Surrey, Guildford, UK; Imperial College London, UK

Francisco Handrick Tomaz da Costa (40)
Computer Science Department, University of Brasília, Brazil; The Environmental Sanitation Company of the Federal District, Brasília, Brazil

Behnaz Hassanshahi (18)
Oracle Labs, Brisbane, Australia

Martin Hirzel (27)
IBM Research, Yorktown Heights, NY, USA

Peter Hofer (19)
Oracle Labs, Linz, Austria

Jack Hughes (15)
School of Computing, University of Kent, Canterbury, UK

Seung-Won Hwang (35)
Department of Computer Science and Engineering, Seoul National University, South Korea

Yusuke Izawa (16)
Tokyo Metropolitan University, Japan

Arthur Jamet (38)
University of Kent, Canterbury, UK

Juan C. Jaramillo (17)
Unversity of Groningen, The Netherlands

Einar Broch Johnsen (29)
University of Oslo, Norway

Ryan Jung (42)
Northeastern University, Boston, MA, USA

Eduard Kamburjan (29)
IT University of Copenhagen, Denmark

Nima Karimipour (18)
University of California, Riverside, CA, USA

Joseph Kenis (30)
The University of Texas at Austin, TX, USA

Vasileios Klimis (41)
Queen Mary University of London, UK

Sebastian Kloibhofer (19)
Institute for System Software, Johannes Kepler University, Linz, Austria

David Klopp (33)
JGU Mainz, Germany

Shriram Krishnamurthi (26)
Brown University, Providence, RI, USA

Soumik Kumar Basu (21)
Department of Computer Science and Engineering, IIT Hyderabad, India

Prashant Kumar (20)
JGU Mainz, Germany

David Leopoldseder (19, 25)
Oracle Labs, Vienna, Austria

Ding Li (6)
Peking University, Beijing, China

Lin Li (6)
Huawei Cloud Computing Technologies Co., Ltd., Guiyang, China

Zongyang Li (6)
Peking University, Beijing, China

Guangtai Liang (6)
Huawei Cloud Computing Technologies Co., Ltd., Guiyang, China

Xiapu Luo (34)
The Hong Kong Polytechnic University, Hong Kong

Yu-Seung Ma (35)
Electronics and Telecommunications Research Institute, Daejeon, South Korea

Lukas Makor (19)
Institute for System Software, Johannes Kepler University, Linz, Austria

Hidehiko Masuhara (16)
Institute of Science Tokyo, Japan

Ferdia McKeogh (3)
University of St Andrews, UK

Ismael Medeiros (40)
Computer Science Department, University of Brasília, Brazil

Alexandra Mendes (11)
INESC TEC, Faculty of Engineering, University of Porto, Portugal

Mira Mezini (40)
TU Darmstadt, Germany; hessian.AI, Darmstadt, Germany; National Research Center for Applied Cybersecurity ATHENE, Darmstadt, Germany

Ana Milanova (27)
Rensselaer Polytechnic Institute, Troy, NY, USA

Antoine Miné (28, 32)
Sorbonne Université, CNRS, LIP6, Paris, France

Jacqueline L. Mitchell (22)
University of Southern California, Los Angeles, CA, USA

Raphaël Monat (32)
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Cameron Moy (42)
PLT @ Northeastern University, Boston, MA, USA

Aäron Munsters (23)
Vrije Universiteit Brussel, Brussels, Belgium

Hanspeter Mössenböck (19)
Institute for System Software, Johannes Kepler University, Linz, Austria

Anders Møller (9)
Aarhus University, Denmark

Krishna Narasimhan (40)
F1RE, Amsterdam, The Netherlands

Tim Nelson (26)
Brown University, Providence, RI, USA

Leandro Oliveira (40)
Computer Science Department, University of Brasília, Brazil

André Pacak (20)
JGU Mainz, Germany

Luca Padovani (24)
University of Bologna, Italy

Luke Panayi (3)
Imperial College London, UK

Hila Peleg (10)
Technion, Haifa, Israel

Valentin Perrelle (28)
Université Paris-Saclay, CEA, List, Palaiseau, France

Andrej Pečimúth (25)
Charles University, Prague, Czech Republic; Oracle Labs, Prague, Czech Republic

Siddhartha Prasad (26)
Brown University, Providence, RI, USA

Nuno Preguiça (4)
NOVA School of Science and Technology, Caparica, Portugal

Jorge A. Pérez (17)
Unversity of Groningen, The Netherlands

Azalea Raad (5, 14)
Imperial College London, UK

Ingkarat Rak-amnouykit (27)
Rensselaer Polytechnic Institute, Troy, NY, USA

Mamy Razafintsialonina (28)
Université Paris-Saclay, CEA, List, Palaiseau, France; Sorbonne Université, CNRS, LIP6, Paris, France

Márcio Ribeiro (40)
Federal University of Alagoas, Brazil

Christopher J Rossbach (30)
The University of Texas at Austin, TX, USA

Angel Luis Scull Pupo (23)
Vrije Universiteit Brussel, Brussels, Belgium

Riccardo Sieve (29)
University of Oslo, Norway

Julien Signoles (28)
Université Paris-Saclay, CEA, List, Palaiseau, France

Tom Spink (3)
University of St Andrews, UK

Manu Sridharan (18)
University of California, Riverside, CA, USA

Yulei Sui (34, 36)
University of New South Wales, Sydney, Australia

Aditya Thimmaiah (30)
The University of Texas at Austin, TX, USA

Dawit Tirore (31)
IT University of Copenhagen, Denmark

Petr Tůma (25)
Charles University, Prague, Czech Republic

Milla Valnet (32)
LIP6, Sorbonne Université, F-75005, Paris, France

Jyothi Vedurada (21)
Department of Computer Science and Engineering, IIT Hyderabad, India

Tomáš Vojnar (37)
Faculty of Informatics, Masaryk University, Brno, Czech Republic; Faculty of Information Technology, Brno University of Technology, Czech Republic

Michael Vollmer (15, 38)
School of Computing, University of Kent, Canterbury, UK

Chao Wang (22)
University of Southern California, Los Angeles, CA, USA

Qianxiang Wang (6)
Huawei Cloud Computing Technologies Co., Ltd., Guiyang, China

Zeyu Wang (6)
Huawei Cloud Computing Technologies Co., Ltd., Guiyang, China

Matthew K. L. Wong (39)
Department of Computing, Imperial College London, UK

Tao Xie (6)
Peking University, Beijing, China

Runqing Xu (33)
JGU Mainz, Germany

Jiawei Yang (34)
The Hong Kong Polytechnic University, Hong Kong

Sangyeop Yeo (35)
Electronics and Telecommunications Research Institute, Daejeon, South Korea

Zijian Yi (30)
The University of Texas at Austin, TX, USA

Gianluigi Zavattaro (24)
University of Bologna, Italy; INRIA OLAS team, Sophia Antipolis, France

Guanqin Zhang (36)
University of New South Wales, Sydney, Australia; CSIRO's Data61, Sydney, Australia

Zhenya Zhang (36)
Kyushu University, Fukuoka, Japan; National Institute of Informatics, Tokyo, Japan

Jianjun Zhao (36)
Kyushu University, Fukuoka, Japan

Elena Zucca (7, 8)
DIBRIS, Università di Genova, Italy

A Theory of (Linear-Time) Timed Monitors

Mouloud Amara
Université Paris Cité, CNRS, IRIF, F-75013 Paris, France

Giovanni Bernardi
Université Paris Cité, CNRS, IRIF, F-75013 Paris, France

Mohammed Aristide Foughali
Université Paris Cité, CNRS, IRIF, F-75013 Paris, France

Adrian Francalanza
Dept. of Computer Science, ICT, University of Malta, Msida, Malta

Abstract

Runtime Verification (RV) is gaining popularity due to its scalability and ability to analyse block-box systems. *Monitoring* is at the heart of RV; a logical formula ϕ, formalising some property of interest, is typically translated into a monitor that checks whether the system under scrutiny satisfies ϕ during its execution. A logical formula ϕ is violation (resp. satisfaction) *monitorable* iff there exists a monitor for ϕ that is both sound and complete w.r.t. its violation (resp. satisfaction). The *monitorability* problem is thus concerned with determining the largest subset of a logic L that is monitorable. Although this problem has been solved for expressive untimed logics, it remains open for timed logics, where formulae can express both the order of events and the quantity of time separating them. This paper solves the monitorability problem for T^{lin}, a new expressive (linear-time) timed μ-calculus that we propose. First, we show that T^{lin} is strictly more expressive than MTL, the de facto timed extension of LTL. Second, we identify MT^{lin}, the largest monitorable fragment of T^{lin}: we characterise its largest subsets of formulae that are violation monitorable, satisfaction monitorable, and *complete monitorable* (both satisfaction and violation monitorable). To wit, this is the first work that answers the monitorability question for such an expressive timed logic.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Timed logics, Runtime Verification, Monitorability

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.1

Related Version *Full Version*: `https://hal.science/hal-05043055/document`

Supplementary Material *Software*: `https://zenodo.org/records/15283069`
Software (ECOOP 2025 Artifact Evaluation approved artifact):
`https://doi.org/10.4230/DARTS.11.2.8`

Funding This work has received support under the program "Investissement d'Avenir" launched by the French Government and implemented by ANR, with the reference "ANR-18-IdEx-000" as part of its program "Émergence". Other fundings include the ANR-JST Grant CyPhAI as well as the ANR project MAVEriQ.

1 Introduction

Runtime Verification (RV) is a lightweight verification technique [30, 10], where the system at hand is *instrumented* with *monitors* [20, 21]. A monitor M is a software component that corresponds to a logical formula ϕ expressing some property of interest. At runtime, M *incrementally* analyses the execution of the system until it reaches a *verdict* w.r.t. the satisfaction or violation of ϕ [30, 2]. RV can mitigate two major drawbacks of exhaustive, offline techniques such as model checking. First, by trading exhaustiveness for scalability,

© Mouloud Amara, Giovanni Bernardi, Mohammed Aristide Foughali, and Adrian Francalanza;
licensed under Creative Commons License CC-BY 4.0

RV does not suffer from the state-space explosion problem. Second, provided that actions of interest are visible to monitors, RV is suitable for systems where no model is known beforehand [19], e.g., black-box and self-adaptive systems [33].

Due to the lack of exhaustiveness, the *monitorability* problem becomes a central issue [39, 10, 3]. Intuitively, a formula ϕ is monitorable iff its violation or satisfaction by the system can be detected by a monitor. Consider the following formula, expressed in the (action-based) Linear-time Temporal Logic (LTL) interpreted over infinite executions [38]:

$$\varphi \;=\; G\,(b \Rightarrow F\,c)$$

Formula φ expresses a classical "leads to" property, requiring that "every action b must be followed by an action c in future", with known algorithms to model-check it offline [31, 13, 47]. However, this formula is *not* monitorable, since any satisfaction verdict after a c may be contradicted by a future b not followed by a c (and dually, any violation verdict after a b may be contradicted by a future c). In more technical terms, a formula ϕ is said to be satisfaction (resp. violation) monitorable [2, 19] iff there exists a monitor for ϕ that is both sound and satisfaction (resp. violation) complete (alternative definitions of monitorability are discussed in Section 6). Formula ϕ is said *monitorable* iff it is satisfaction or violation monitorable, and *complete monitorable* if it is both satisfaction and violation monitorable. The *monitorability problem* is thus concerned with determining the largest monitorable subset of a given logic L.

Monitorability has been studied for the modal μ-calculus, also known as the Hennessy-Milner Logic with recursion (μ-HML) [28, 29]. Notably, the *maximally-expressive monitorable subset* of μ-HML, i.e., a subset within which any monitorable formula in μ-HML is expressible, has been identified for both linear-time and branching-time semantics [2, 19]. However, μ-HML is unable to express *timed properties*. Consider φ', a timed version of φ, written in the Metric Temporal Logic MTL [27] (the de facto timed extension of LTL):

$$\varphi' \;=\; G\,(b \Rightarrow F_{[0,5]}\,c)$$

Formula φ' expresses a *bounded-response* property, requiring that every occurrence of b must be followed by an occurrence of c *within 5 time units*. Timed properties are extremely important in real-time systems (see e.g., [35, 17]). Even though the body of research on RV for timed systems is extensive (more in Section 6), there is, to the best of our knowledge, no work that concretely tackles the monitorability problem in the timed setting.

Contributions. Our contribution is twofold. First, we propose T^{lin}, a new timed μ-calculus with a linear-time interpretation[1], and prove that T^{lin} is strictly more expressive than MTL (without past operators, Theorem 3.6). Second, we precisely identify VT^{lin}, ST^{lin} and CT^{lin}, the largest (i.e., maximally-expressive) fragments of T^{lin} that are violation monitorable, satisfaction monitorable and complete monitorable, respectively. Accordingly, $MT^{lin} = VT^{lin} \cup ST^{lin}$ is the largest monitorable fragment of T^{lin} (Theorem 5.43, Theorem 5.45, Theorem 5.44). For practical reasons, we also provide automatic translators (i) from MTL to T^{lin} and (ii) from MT^{lin} to monitors. To the best of our knowledge, this is the first work that solves the monitorability problem for an expressive timed logic, providing a full characterisation of its largest monitorable fragment.

[1] Note that a number of timed μ-calculi exist (see [14] for an overview), yet outside of this paper's scope due to their branching-time semantics.

Outline. The paper is organised as follows. After overviewing the preliminaries in Section 2, we discuss our core contributions in Section 3 through Section 5. Section 3 defines our new logic T^{lin} and proves that it is more expressive than MTL, providing along the way an automatic translation from the latter to the former. Section 4 lays out the theoretical foundations for timed monitors and instrumentation. We then tackle the monitorability problem in Section 5. After setting up the framework (Section 5.1), we provide a compiler from T^{lin} formulae to sound monitors (Section 5.2). We then identify CT^{lin}, VT^{lin} and ST^{lin} (both supersets of CT^{lin}), fragments of complete-, violation- and satisfaction-monitorable formulae in T^{lin}, respectively (Section 5.3, Section 5.4). In Section 5.5, we prove the maximality of CT^{lin}, VT^{lin} and ST^{lin}, and conclude accordingly that $MT^{lin} = VT^{lin} \cup ST^{lin}$ is the largest monitorable fragment of T^{lin}. Section 6 compares our contributions with related work and Section 7 concludes with future work directions.

Full version. An extended version of this paper with all proofs, more examples and further details, is available online (see Full Version above).

2 Preliminaries

In the remainder of this paper, let Σ denote a *non-empty, finite set* of *actions*.

2.1 Timed Labelled Transition Systems

The following definition is inspired from [1].

▶ **Definition 2.1.** *A Timed Labelled Transition System (TLTS) P over Σ is a tuple $\langle S_P, \text{LAB}, \longrightarrow_P \rangle$ where S_P is a set of states, $\text{LAB} = \Sigma \cup \mathbb{R}_{\geq 0} \cup \{\tau\}$ a set of labels, and $\longrightarrow_P \subseteq S_P \times \text{LAB} \times S_P$ the transition relation satisfying:*

1. *Time determinism: for every $p, p', p'' \in S_P$ and $d \in \mathbb{R}_{\geq 0}$, if $(p, d, p') \in \longrightarrow_P$ and $(p, d, p'') \in \longrightarrow_P$ then $p' = p''$,*
2. *Time additivity: for every $p, p'' \in S_P$ and $d_1, d_2 \in \mathbb{R}_{\geq 0}$, $(p, d_1 + d_2, p'') \in \longrightarrow_P$ iff $(p, d_1, p') \in \longrightarrow_P$ and $(p', d_2, p'') \in \longrightarrow_P$ from some $p' \in S_P$,*
3. *Zero delay: for every $p, p' \in S_P$, $(p, 0, p') \in \longrightarrow_P$ iff $p' = p$.*

A transition labelled $\alpha \in \Sigma$ (resp. $\alpha \in \mathbb{R}_{\geq 0}$) is a called an *action* transition (resp. delay transition). A τ transition is a non-observable transition labelled with the special symbol τ. We use a, a_i, b, c and d, d_i to denote, respectively, an element of Σ and $\mathbb{R}_{\geq 0}$. We write $p_1 \xrightarrow{\alpha_1} p_2 \xrightarrow{\alpha_2} p_3 \ldots$ to denote a *sequence of transitions* with $(p_1, \alpha_1, p_2), (p_2, \alpha_2, p_3) \ldots \in \longrightarrow_P$.

Runs. A *run* in the TLTS P is a sequence of labels (in $\text{LAB}^\omega \cup \text{LAB}^* = \text{LAB}^\infty$) corresponding to a sequence of transitions. If there exists an infinite transition sequence $p_1 \xrightarrow{\alpha_1} p_2 \xrightarrow{\alpha_2} p_3 \xrightarrow{\alpha_3} \ldots$, we say that run $\theta = \alpha_1 \alpha_2 \alpha_3 \ldots \in \text{LAB}^\omega$ is *infinite* from $p_1 \in S_P$.

Weak runs. Let $\mathcal{L} = \text{LAB} \setminus \{\tau\}$. To define weak transitions, we write $p \Longrightarrow p'$ instead of $p \, (\xrightarrow{\tau})^* \, p'$ which is the reflexive-transitive closure of the relation $\xrightarrow{\tau}$, and following [49], we write $p \xRightarrow{\alpha} p'$ for $\alpha \in \mathcal{L}$ in the following cases,

(a) $p \xRightarrow{a} p'$ if $p \Longrightarrow \cdot \xrightarrow{a} \cdot \Longrightarrow p'$,

(b) $p \xRightarrow{d} p'$ if there exists $n \geq 0$ s.t. $p \Longrightarrow \cdot \xrightarrow{d_0} \cdot \Longrightarrow \cdot \xrightarrow{d_1} \ldots \xrightarrow{d_n} \cdot \Longrightarrow p'$ where $d = \sum_{i \leq n} d_i$,

with the convention that if the set $d_1, \ldots, d_n$ is empty, then $\sum_{1 \leq i \leq n} d_i = 0$. Following [19], for $\alpha_1, \ldots, \alpha_n \in \mathcal{L}$ and $p, p' \in S_P$, we write sequences of transitions $p \xrightarrow{\alpha_1}\!\!\!\!\rightarrow \ldots \xrightarrow{\alpha_n}\!\!\!\!\rightarrow p'$ as $p \xrightarrow{\theta}\!\!\!\!\rightarrow p'$ where $\theta = \alpha_1 \ldots \alpha_n$. Such θ is called a weak run (in $\mathcal{L}^*$). We say that a weak finite run $\theta' \in \mathcal{L}^*$ *corresponds to* a finite run $\theta \in \text{LAB}^*$ iff removing τ transitions then summing consecutive delays in θ gives θ'. For instance, the weak finite run $5\,a\,3\,b\,2$ corresponds to e.g., finite runs $2\,\tau\,3\,a\,3\,b\,\tau\,2$ and $5\,a\,3\,b\,1\,\tau\,1$. The *empty weak run* is $\epsilon = 0$.

Reachability and inevitability. In TLTS P, a state $p' \in S_P$ is reachable from state $p \in S_P$ along a finite weak run $\theta \in \mathcal{L}^*$ iff $p \xrightarrow{\theta}\!\!\!\!\rightarrow p'$. We say that p' is *inevitably reachable* from p along θ, denoted $p \xrightarrow{\theta}\!\!\!\!\rightarrow_{inev} p'$, if p' is the only reachable state from p along θ, that is:

$$p \xrightarrow{\theta}\!\!\!\!\rightarrow p' \text{ and } \forall p'' \in S_P : p \xrightarrow{\theta}\!\!\!\!\rightarrow p'' \text{ implies } p' = p''$$

Similarly, we write $p \xrightarrow{\theta}\!\!\!\!\!\!\not\rightarrow p'$ if p' is not reachable from p along θ, $p \xrightarrow{\theta}\!\!\!\!\!\!\not\rightarrow$ if there exists a state p' reachable from p along θ, and $p \xrightarrow{\theta}\!\!\!\!\!\!\not\rightarrow$ if there is no reachable state from p along θ.

2.2 Timed Words

As in e.g., [12, 22], we interpret timed logics over infinite timed words.

▶ **Definition 2.2** (Timed words). *A timed word π over Σ is a possibly infinite sequence $(a_1, t_1)(a_2, t_2)...$ of timed events $(a_i, t_i) \in \Sigma \times \mathbb{R}_{\geq 0}$, where t_i are monotonically increasing timestamps, i.e., $\forall i \in \mathbb{N}$: $t_i \leq t_{i+1}$. An empty timed word contains no timed events.*

We denote $T\Sigma^\omega$ and $T\Sigma^*$ the set of infinite (omega) and finite timed words, respectively, and $T\Sigma^\infty$ the set of all timed words ($T\Sigma^\infty = T\Sigma^\omega \cup T\Sigma^*$). We denote by $|\pi|$ the *size* of a timed word $\pi \in T\Sigma^\infty$. The size of a finite timed word $\pi = (a_1, t_1)...(a_n, t_n) \in T\Sigma^*$ equals n. The size of an infinite timed word $\pi \in T\Sigma^\omega$ equals ∞. We use the notation $a_i(\pi)$ (resp. $t_i(\pi)$) for the action (resp. the timestamp) of the timed event at position i of $\pi \in T\Sigma^\infty$, with $i \in \mathbb{N}$ and $i \leq |\pi|$. For a timed event (a_i, t_i), we say that t_i is associated with a_i. The equality between two timed words $\pi, \pi' \in T\Sigma^\infty$ is defined in the usual way. For some $\pi \in T\Sigma^\infty$, we write π^i (resp. ${}^i\pi$), $i \leq |\pi|$, for the suffix of π starting at position i (resp. the prefix of π ending at i). Concatenation is defined through the partial function $Concat : T\Sigma^* \times T\Sigma^\infty \rightharpoonup T\Sigma^\infty$, defined only for couples $(\pi, \pi') \in T\Sigma^* \times T\Sigma^\infty$ such that $\pi' \in T\Sigma^\omega_{\vec{\pi}}$, where $T\Sigma^\omega_{\vec{\pi}}$ is defined as follows, w.r.t. to some finite timed word π: $T\Sigma^\omega_{\vec{\pi}} = \{\pi' \in T\Sigma^\omega \mid t_1(\pi') \geq t_{|\pi|}(\pi)\}$. We use the shorthand notation $\pi.\pi'$ instead of $Concat(\pi, \pi')$. For any timed word $\pi' \in T\Sigma^\infty$, we define the set of its prefixes $Pf(\pi') = \{\pi \in T\Sigma^* \mid \exists \pi'' \in T\Sigma^\omega_{\vec{\pi}} : \pi.\pi'' = \pi'\}$. Dually, we define for any finite timed word $\pi' \in T\Sigma^*$ its infinite *extensions* $Ext(\pi') = \{\pi \in T\Sigma^\omega \mid \pi' \in Pf(\pi)\}$.

Note that a timed word corresponds necessarily to a weak run in $\mathcal{L}^*$, with each timestamp recording the *absolute time* of occurrence of the action associated with it. For example, the finite timed word $(a, 0)(b, 3)(a, 3)(c, 7.2)$ results from the finite weak run $a\,3\,b\,a\,4.2\,c$. We will formalise the function to retrieve a finite weak run from a finite timed word in Section 5.1.

▶ Remark 2.3. Note that, in the timed systems literature, every execution in a TLTS corresponds to an infinite *time divergent* run/timed word (see e.g., [23, 8]), i.e., all runs/timed words diverge in time in the future. We do not make this assumption here since our results up to and including Section 4 do not depend on it. Instead, we formalise, justify and enforce it in Section 5.1 for monitorability purposes.

$$\phi, \psi \quad := \quad a \mid \neg a \mid \phi \wedge \psi \mid \phi \vee \psi \mid \bigcirc_I \phi \mid \phi \, \mathcal{U}_I \, \psi \mid G_I \, \phi$$

Figure 1 MTL syntax.

2.3 Intervals

We now define $\mathcal{I}$, the set of dense-time intervals with natural bounds. Hereafter, we refer to an element of $\mathcal{I}$ simply as "interval".

▶ **Definition 2.4** (Interval syntax). *The set $\mathcal{I}$ is given by the grammar $I \in \mathcal{I} ::= \sqsubset p, q \sqsupset$ where $\sqsubset \in \{[, (\}$, $\sqsupset \in \{],)\}$, $p \in \mathbb{N}$ and $q \in \mathbb{N} \cup \{\infty\}$. As per standard notation, if $q = \infty$ then $\sqsupset =)$. p (resp. q) is the left (resp. right) bound of I, we write accordingly $\mathcal{LB}(I) = p$ (resp. $\mathcal{RB}(I) = q$). I is said left closed if $\sqsubset = [$, right closed if $\sqsupset =]$, and bounded if $\mathcal{RB}(I) \neq \infty$.*

▶ **Definition 2.5** (Interval Semantics). *$[\![I]\!]$, the meaning of an interval $I = \sqsubset p, q \sqsupset \in \mathcal{I}$, is a subset of $\mathbb{R}_{\geq 0}$ defined as follows: $[\![I]\!] = \{x \in \mathbb{R}_{\geq 0} \mid p \prec_l x \prec_r q\}$ where*

- *$\prec_l = \leq$ if $\sqsubset = [$ and $\prec_l = <$ otherwise,*
- *$\prec_r = \leq$ if $\sqsupset =]$ and $\prec_r = <$ otherwise.*

Hereafter, we (i) consider only non-empty intervals, i.e., in the set $\{I \in \mathcal{I} \mid [\![I]\!] \neq \varnothing\}$ and (ii) use I to denote both the syntax and semantics of an interval I.

2.4 Metric Temporal Logic (MTL)

MTL [27] is a timed logic with linear-time semantics; it is the de facto timed extension of LTL. Note that we do not consider past operators.

Syntax and Semantics. The syntax of MTL is given in Figure 1. It includes the timed *Next* operator ($\bigcirc_I$) and the timed *Until* ($\mathcal{U}_I$) operator. As per standard notation, $\top = a \vee \neg a$, $\bot = a \wedge \neg a$, and $F_I \, \phi = \top \, \mathcal{U}_I \, \phi$ (denoting "Eventually ϕ within I"). As in [40], we add the timed *Globally* (G_I) operator to the syntax to guarantee that negations appear only before propositions, without loss of expressivity[2]. The *point-wise* semantics of MTL is given by the truth value of $\pi, i \models \phi$, denoting "$\pi \in T\Sigma^\omega$ satisfies formula ϕ at position i", with the convention that $\pi \models \phi$ iff $\pi, 1 \models \phi$ [6, 36, 45, 32]. Here, we define the semantics of a formula ϕ as the set $\{\pi \mid \pi \models \phi\}$ (Figure 2). In the remainder of this paper, we omit the subscript I of temporal operators if $I = [0, \infty)$. Accordingly, if all operators appearing in a formula ϕ are subscript free, then ϕ is an untimed MTL formula (i.e., $\phi \in$ LTL). For example, $F_{[0,\infty)} \, a$, with $a \in \Sigma$, is simply written as $F \, a$ and is therefore untimed.

▶ **Example 2.6.** Let $b, c \in \Sigma$. $[\![b \, \mathcal{U}_I \, c]\!]$ contains all infinite timed words where b is satisfied in every position from the start, until c becomes satisfied within interval I (relative to the first timestamp in π). In other words, an infinite timed word π is in $[\![b \, \mathcal{U}_I \, c]\!]$ iff $\pi \in [\![b \, \mathcal{U} \, c]\!]$ *and*

[2] That is, any formula expressed within the minimal syntax in e.g., [32] (excluding past operators), allowing negations of formulae and without G_I, can be written in our syntax in Figure 1 using the following standard equalities until negations are pushed to the level of propositions. For timed *Until*: $\neg(\phi \, \mathcal{U}_I \, \psi) = (G_I \, \neg\psi) \vee \neg\psi \, \mathcal{U}_I \, (\neg\psi \wedge \neg\phi)$, and for timed *Next* (see [25]): $\neg \bigcirc_I \phi = \bigvee_{a \in \Sigma} \bigcirc_{[0, \mathcal{LB}(I)\sqsupset} a \vee \bigcirc_I \neg\phi \vee \bigvee_{a \in \Sigma} \bigcirc_{\sqsubset \mathcal{RB}(I), \infty)} a$ where $\sqsupset =)$ if I is left closed and $\sqsupset =]$ otherwise, $\sqsubset = ($ if I is right closed and $\sqsubset = [$ otherwise.

$$
\begin{array}{lcl}
[\![a]\!] & = & \{\pi \mid a_1(\pi) = a\} \\
[\![\neg a]\!] & = & \{\pi \mid a_1(\pi) \neq a\} \\
[\![\phi \wedge \psi]\!] & = & [\![\phi]\!] \cap [\![\psi]\!] \\
[\![\phi \vee \psi]\!] & = & [\![\phi]\!] \cup [\![\psi]\!] \\
[\![\bigcirc_I \phi]\!] & = & \{\pi \mid (t_2(\pi) - t_1(\pi)) \in I \text{ and } \pi^2 \in [\![\phi]\!]\} \\
[\![\phi \,\mathcal{U}_I\, \psi]\!] & = & \{\pi \mid \exists i : (t_i(\pi) - t_1(\pi)) \in I \text{ and } \pi^i \in [\![\psi]\!] \text{ and } \forall 1 \leq j < i : \pi^j \in [\![\phi]\!]\} \\
[\![G_I\, \phi]\!] & = & \{\pi \mid \forall i \geq 1 : (t_i(\pi) - t_1(\pi)) \in I \implies \pi^i \in [\![\phi]\!]\}
\end{array}
$$

Figure 2 MTL semantics.

the first c occurs within I, i.e., at a position i of π satisfying $t_i(\pi) - t_1(\pi) \in I$. For instance, for $I = [2,5]$, $\pi = (b,3)(a,4)(c,7)\ldots$, $\pi' = (b,3)(b,4)(c,7)\ldots$ and $\pi'' = (b,2)(c,3)\ldots$, we have: $\pi \notin [\![b \,\mathcal{U}_I\, c]\!]$, $\pi' \in [\![b \,\mathcal{U}_I\, c]\!]$ and $\pi'' \notin [\![b \,\mathcal{U}_I\, c]\!]$. In particular, $\pi'' \notin [\![b \,\mathcal{U}_I\, c]\!]$ even though $\pi'' \in [\![b \,\mathcal{U}\, c]\!]$, because the first c occurs "too early" with timestamp 3 and $3 - 2 \notin [2,5]$.

3 The logic T^{lin}

We present T^{lin} (Section 3.1), our novel extension of (untimed) modal μ-calculus with linear-time semantics [15] [2, Sect.2] to the dense-time setting, and prove that T^{lin} is strictly more expressive than MTL (Section 3.2).

3.1 T^{lin} Syntax and Semantics

We first introduce a syntactic characterisation of timing constraints which we call *guards*. In the sequel, let $\mathcal{D}$ be an infinite set of variables taking values in $\mathbb{R}_{\geq 0}$, that we call *timestamp variables*, and let $\mathcal{V}$ be the set of valuation functions $v \in \mathcal{D} \to \mathbb{R}_{\geq 0}$.

▶ **Definition 3.1** (Guards). *The set $\mathcal{B}(\mathcal{D})$ of guards over $\mathcal{D}$ is given by the syntax*

$$g \;:=\; x \prec n \;\mid\; (x-y) \prec n \;\mid\; n \prec x \;\mid\; n \prec (x-y) \;\mid\; g \wedge g \;\mid\; \overline{g}$$

where $n \in \mathbb{N}$, $\prec \in \{<, \leq\}$, $x, y \in \mathcal{D}$ and $\overline{g}$ is a compact notation for $\neg g$.

A valuation v *satisfies* a guard g, denoted $v \models g$, iff replacing each variable x by $v(x)$ in g gives an expression that evaluates to true. For example, $v \models (x-y) < 2$ iff $v(x) - v(y) < 2$. We write $v \not\models g$ for $\neg(v \models g)$. If I is an interval, then we write $(x-y) \in I$ as an abbreviation for the guard $\mathcal{LB}(I) \prec_l (x-y) \wedge (x-y) \prec_r \mathcal{RB}(I)$ (where $\prec_l$ and $\prec_r$ are as defined in Definition 2.5), and $(x-y) \notin I$ for $\overline{((x-y) \in I)}$. We write $v[x := m]$, $x \in \mathcal{D}$ and $m \in \mathbb{R}_{\geq 0}$, to denote the valuation that maps x to m and every $y \in \mathcal{D} \setminus \{x\}$ to $v(y)$. We denote by $[x_1 = m_1, \ldots, x_n = m_n]$ the current valuations of some variables of interest $x_1, \ldots, x_n$ when valuation v is understood from the context.

Our T^{lin} consists of a set of *formulae* with linear-time timed semantics. Thus, before defining T^{lin}, we first define TT^{lin}, the set of *terms* that can be used to write a formula ($T^{lin} \subset TT^{lin}$). The set TT^{lin} is generated by the grammar in Figure 3, with $a \in \Sigma$, X an element of a countable set of recursion variables VAR, $x \in \mathcal{D}$, and $g \in \mathcal{B}(\mathcal{D})$. Syntactically $\mathsf{min}(X, \phi)$ and $\mathsf{max}(X, \phi)$ are binders of recursion variable X in ϕ, and $x \,\underline{\mathtt{in}}\, \phi$ binds timestamp variable $x \in \mathcal{D}$ in ϕ. The free recursion (resp. timestamp) variables of a term ϕ are given by the function $fv : TT^{lin} \longrightarrow \mathcal{P}(\text{VAR})$ (resp. $fv_{\text{TIME}} : TT^{lin} \longrightarrow \mathcal{P}(\mathcal{D})$), defined in the expected way. A term in which no variable is free is *closed*.

ϕ	$::=$	tt	*true*	$\mid$	ff	*false*
	$\mid$	X	*variable*	$\mid$	$x \,\underline{\mathtt{in}}\, \phi$	*timestamp*
	$\mid$	$\phi \wedge \psi$	*conjunction*	$\mid$	$\phi \vee \psi$	*disjunction*
	$\mid$	$\langle a \rangle \phi$	*diamond*	$\mid$	$[a]\phi$	*box*
	$\mid$	$\langle g \rangle \phi$	*diamond on time*	$\mid$	$[g]\phi$	*box on time*
	$\mid$	$\mathsf{min}(X, \phi)$	*min. fixpoint*	$\mid$	$\mathsf{max}(X, \phi)$	*max. fixpoint*

Figure 3 Syntax of TT^{lin}.

Compared to the untimed version of linear-time μ-calculi in [15] [2, Sect.2], TT^{lin} provides three new syntactic constructs to handle dense time: $x \,\underline{\mathtt{in}}\, \phi$, $[g]\phi$ and $\langle g \rangle \phi$. These are best discussed when introducing formulae later in this subsection.

$$
\begin{array}{lcllcl}
[\![\mathsf{tt}, \rho]\!]_v & = & T\Sigma^\omega & [\![\mathsf{ff}, \rho]\!]_v & = & \emptyset \\
[\![\phi \vee \psi, \rho]\!]_v & = & [\![\phi, \rho]\!]_v \cup [\![\psi, \rho]\!]_v & [\![\phi \wedge \psi, \rho]\!]_v & = & [\![\phi, \rho]\!]_v \cap [\![\psi, \rho]\!]_v \\
[\![\langle a \rangle \phi, \rho]\!]_v & = & \langle \cdot a \cdot \rangle ([\![\phi, \rho]\!]_v) & [\![[a]\phi, \rho]\!]_v & = & [\cdot a \cdot] ([\![\phi, \rho]\!]_v) \\
[\![x \,\underline{\mathtt{in}}\, \phi, \rho]\!]_v & = & \{\pi \mid \pi \in [\![\phi, \rho]\!]_{v[x := t_1(\pi)]}\} & [\![X, \rho]\!]_v & = & \rho(X) \\
[\![\mathsf{min}(X, \phi), \rho]\!]_v & = & \bigcap \{S \mid [\![\phi, \rho[X \to S]]\!]_v \subseteq S\} & [\![\langle g \rangle \phi, \rho]\!]_v & = & \langle \cdot g \cdot \rangle (v, [\![\phi, \rho]\!]_v) \\
[\![\mathsf{max}(X, \phi), \rho]\!]_v & = & \bigcup \{S \mid S \subseteq [\![\phi, \rho[X \to S]]\!]_v\} & [\![[g]\phi, \rho]\!]_v & = & [\cdot g \cdot] (v, [\![\phi, \rho]\!]_v)
\end{array}
$$

Figure 4 Linear time semantics of TT^{lin}.

The semantics of terms over infinite timed words is given in Figure 4, where we have used the auxiliary functions $\langle \cdot a \cdot \rangle(-)$, $[\cdot a \cdot](-)$, $\langle \cdot g \cdot \rangle(v, -)$, $[\cdot g \cdot](v, -) : \mathcal{P}(T\Sigma^\omega) \longrightarrow \mathcal{P}(T\Sigma^\omega)$ defined by the following equalities:

$$
\begin{array}{lll}
\langle \cdot a \cdot \rangle(Z) & = \{\pi \mid a_1(\pi) = a \wedge \pi^2 \in Z\} & [\cdot a \cdot](Z) = \{\pi \mid a_1(\pi) = a \Rightarrow \pi^2 \in Z\} \\
\langle \cdot g \cdot \rangle(v, Z) & = \{\pi \mid v \models g \wedge \pi \in Z\} & [\cdot g \cdot](v, Z) = \{\pi \mid v \models g \Rightarrow \pi \in Z\}
\end{array}
\tag{1}
$$

and where environment ρ is an element of Υ, the set of partial functions $\textsc{Var} \rightharpoonup \mathcal{P}(T\Sigma^\omega)$, and v a valuation in $\mathcal{V}$. The notation $[\![\phi, \rho]\!]_v$, with $fv(\phi) \subseteq dom(\rho)$, denotes the set of timed words that satisfy ϕ w.r.t. the environment ρ and the valuation v.

Properties of the semantics. Let $dom(f)$ denote the domain of definition of a partial function f. For any $\phi \in TT^{lin}$, $v \in \mathcal{V}$, $X \in \textsc{Var}$ and $\rho \in \Upsilon$ s.t. $fv(\phi) \subseteq dom(\rho) \cup \{X\}$ we let

$$
\begin{array}{lcl}
\wr \phi \int_{v,X,\rho} & : & \mathcal{P}(T\Sigma^\omega) \longrightarrow \mathcal{P}(T\Sigma^\omega) \\
\wr \phi \int_{v,X,\rho}(Z) & = & [\![\phi, \rho[X \mapsto Z]]\!]_v
\end{array}
\tag{2}
$$

In other words $\wr \phi \int_{v,X,\rho} = \lambda y. [\![\phi, \rho[X \mapsto y]]\!]_v$.

Since environments are functions from logical variables to complete lattice, we order them in the standard pointwise way to obtain a complete lattice $\langle \textsc{Var} \to \mathcal{P}(T\Sigma^\omega), \leq \rangle$, where infinitary least upper bounds and greatest lower bounds are defined as usual. We prove that the function given in Equation (2) is monotone w.r.t. the pointwise ordering of environments, therefore Knaster-Tarski's fixpoint theorem ensures that the largest and least fixpoints of every function $\wr \phi \int_{v,X,\rho}$ exist, and are the semantics of the terms defined via $\mathsf{min}(X, -)$

and $\mathsf{max}(X, -)$. As we further demonstrate that the semantic function is continuous w.r.t. directed sets over the CPO $\langle \mathcal{P}(T\Sigma^\omega), \subseteq \rangle$, we can reason on terms defined via $\mathsf{min}(X, -)$ (resp. $\mathsf{max}(X, -)$) using Kleene fixpoint theorem and its formal dual. These strong results are crucial to prove a number of statements in this paper, in particular Proposition 3.7.

Formulae. We say that a variable $X \in \textsc{Var}$ is *guarded* in a term ϕ if all occurrences of X in ϕ are in the scope of at least one modality (i.e., $\langle a \rangle -$ or $[a]-$, $a \in \Sigma$). We say that a term ϕ is *guarded* if for all its subterms $\mathsf{min}(X, \psi_1)$ and $\mathsf{max}(Y, \psi_2)$, X is guarded in ψ_1 and Y is guarded in ψ_2. For example, the term $\mathsf{min}(X, \langle a \rangle x \,\underline{\mathtt{in}}\, [x \leq 2] X \vee Y)$ is guarded, and the term $\mathsf{min}(X, \langle a \rangle x \,\underline{\mathtt{in}}\, [x \leq 2] X \vee X)$ is not, because the rightmost occurrence of X is not in the scope of a modality. The term $\mathsf{min}(X, x \,\underline{\mathtt{in}}\, [x < 3] X)$ is not guarded, because neither $x \,\underline{\mathtt{in}}$ nor $[x < 3]$ is a modality. We may now define T^{lin}, the set of formulae in TT^{lin}.

▶ **Definition 3.2.** *The set of formulae T^{lin} is defined as the set of closed and guarded terms in TT^{lin}: $T^{lin} = \{\phi \in TT^{lin} \mid \phi \text{ is guarded and } fv(\phi) = fv_{\mathit{TIME}}(\phi) = \emptyset\}$*

Since elements of T^{lin} are closed, their semantics does not depend on valuations and environments. As a consequence, we simply write $[\![\phi]\!]$ instead of $[\![\phi, \rho]\!]_v$ to denote the semantics of any $\phi \in T^{lin}$ in any environment $\rho \in \Upsilon$ and w.r.t. to any $v \in \mathcal{V}$.

▶ Remark 3.3. Note that we work up to α-conversion of bound timestamp and recursion variables. For clarity, we assume these to be distinct and pick fresh ones when manipulating formulae. We also assume each recursion variable to be bound at most once, e.g., the formula $\mathsf{min}(X, \langle a \rangle X \vee \langle b \rangle X) \vee \mathsf{max}(X, \langle c \rangle X)$ can be expressed as $\mathsf{min}(X, \langle a \rangle X \vee \langle b \rangle X) \vee \mathsf{max}(Y, \langle c \rangle Y)$.

Let us now give two examples of the semantics of a formula $\phi \in T^{lin}$, as to explain concretely the meaning of the new constructs $x \,\underline{\mathtt{in}}\, \phi$, $[g]\phi$ and $\langle g \rangle \phi$, relying on Figure 4. In the remainder of this paper, we write $\langle \Sigma \rangle \phi$ to denote a term $\bigvee_{a \in \Sigma} \langle a \rangle \phi$.

▶ **Example 3.4.** Let $\phi \in T^{lin}$ be the following formula:

$$\phi = x \,\underline{\mathtt{in}}\, [x \leq 5] \langle b \rangle \mathtt{tt}$$

The semantics of this formula equals the set of all infinite timed words π that satisfy the following: either $t_1(\pi) > 5$ or $t_1(\pi) \leq 5$ *and* $a_1(\pi) = b$. That is, ϕ expresses the property: "if the first timestamp is smaller than 5, then the action associated with it must be a b".

▶ **Example 3.5.** Let us discuss a more complex example. The following untimed formula, given in [4, Ex. 4.2], expresses the property "every other action is a b":

$$\phi_{even(b)} = \mathsf{max}(X, \langle \Sigma \rangle \langle b \rangle X)$$

Recalling that an untimed formula is implicitly timed with the interval $[0, \infty)$, $\phi_{even(b)}$ contains all timed words π satisfying the following: at every even position i, $a_i(\pi) = b$. We want a timed version of this formula, i.e., to express $\phi_{even(b),I}$, the generalisation of $\phi_{even(b)}$ to any interval $I \in \mathcal{I}$. More precisely, the semantics of $\phi_{even(b),I}$ must contain every infinite timed word π that satisfies the following: for every even position $i \in \mathbb{N}_{>1}$, i.e., $i \bmod 2 = 0$, if $t_i(\pi) - t_1(\pi)$ is within interval I, then $a_i(\pi)$ must be a b, and $a_i(\pi)$ can be any action otherwise. For this, we use the constructs $x \,\underline{\mathtt{in}}\, -$ and $\langle g \rangle -$ to build up the following formula, where y^a denotes a distinct timestamp variable when ranging over actions in Σ:

$$\phi_{even(b),I} = x_s \,\underline{\mathtt{in}}\, \mathsf{max}(X, \langle \Sigma \rangle (x_e \,\underline{\mathtt{in}}\, \langle x_e - x_s \in I \rangle \langle b \rangle X \vee \bigvee_{a \in \Sigma} y^a \,\underline{\mathtt{in}}\, \langle y^a - x_s \notin I \rangle \langle a \rangle X))$$

Let us explain through evaluating whether some infinite timed word π belongs to the semantics of $\phi_{even(b),I}$. The $x_s \,\underline{\texttt{in}}\, -$ is used outside of the maximal-fixpoint operator, allowing timestamp variable x_s (for x "start") to record the first timestamp in π. Then, each recursion step contains two consecutive actions (as in $\phi_{even(b)}$), but this time with two disjoint recursion "paths" (the terms $x_e \,\underline{\texttt{in}}\langle x_e - x_s \in I\rangle\langle b\rangle X$ and $\bigvee_{a\in\Sigma} y^a \,\underline{\texttt{in}}\langle y^a - x_s \notin I\rangle\langle a\rangle X$). The first path requires the second action to be a b occurring within I, and the second allows the second action to be any action in Σ occurring outside of I. To assess whether an action occurs e.g., in I, the first recursion path uses a timestamp variable x_e (for x "end"), updated with the timestamp of every other action in π, and requires the difference between x_e and x_s to belong to I ($x_e \,\underline{\texttt{in}}\langle x_e - x_s \in I\rangle$). Therefore, the only condition for π to be outside of the semantics of $\phi_{even(b),I}$ is the existence of some even position at which the action is not a b *and* occurs within I. For instance, for $I = [3,5]$ and $\pi = (a,3)(c,4)(a,7)(c,9)\ldots$, $\pi' = (a,3)(b,4)(b,7)(b,8)(c,10)\ldots$ and $\pi'' = (a,2)(b,5)(c,6)(c,7)\ldots$ we have:

- $\pi \in [\![\phi_{even(b),[3,5]}]\!]$ because there is no timed action at an even position that happens within $[3,5]$: at even position 2 we have $4-3 \notin [3,5]$, at the next even position 4 we have $9-3 \notin [3,5]$, and since at this position the right bound of the interval is already exceeded then actions at subsequent even positions will occur outside of I,
- $\pi' \in [\![\phi_{even(b),[3,5]}]\!]$ because the only even position with actions occurring within I is position 4 and we have $a_4(\pi') = b$,
- $\pi'' \notin [\![\phi_{even(b),[3,5]}]\!]$ because at position 4 an action occurs within I ($7-2 \in [3,5]$) and such action is not a b.

In general: $\pi \in [\![\phi_{even(b),I}]\!]$ iff $\forall i \in \mathbb{N}_{>1} : i \bmod 2 = 0 \wedge t_i(\pi) - t_1(\pi) \in I \implies a_i(\pi) = b$, which coincides with the generalisation of $\phi_{even(b)}$ to any interval $I \in \mathcal{I}$. Note that when $I = [0,\infty)$, we obtain exactly the untimed version $\phi_{even(b)}$ using the fact that $x_e - x_s \in [0,\infty)$ is always true and $y^a - x_s \notin [0,\infty)$, for every y^a, is always false.

3.2 T^{lin} is strictly more expressive than MTL

▶ **Theorem 3.6.**

1. $\forall \phi \in MTL\ \exists \psi \in T^{lin} : [\![\phi]\!] = [\![\psi]\!]$,
2. $\exists \psi \in T^{lin}\ \forall \phi \in MTL : [\![\psi]\!] \neq [\![\phi]\!]$.

Proof.

Proving Item 1 of Theorem 3.6. We proceed through proposing the function $\text{TO}\,T^{lin} : MTL \longrightarrow T^{lin}$ that transforms any MTL formula into an equivalent T^{lin} formula (Figure 5).

▶ **Proposition 3.7.** *For every $\phi \in MTL$ we have $[\![\phi]\!] = [\![\text{TO}\,T^{lin}(\phi)]\!]$.*

Proving Item 2 of Theorem 3.6. It suffices to find a formula $\psi \in T^{lin}$ that is not expressible in MTL. As a matter of fact, the formula $\phi_{even,I}$ that we used as an example for T^{lin} (Example 3.5) is not expressible in MTL, as states the following proposition[3]:

▶ **Proposition 3.8.** *For any $I \in \mathcal{I}$, the following formula is not expressible in MTL*

$$\phi_{even(b),I} = x_s \,\underline{\texttt{in}}\, \mathsf{max}(X, \langle\Sigma\rangle(x_e \,\underline{\texttt{in}}\langle x_e - x_s \in I\rangle\langle b\rangle X \vee \bigvee_{a\in\Sigma} y^a \,\underline{\texttt{in}}\langle y^a - x_s \notin I\rangle\langle a\rangle X))$$

[3] We recall that elements of $\mathcal{I}$ cannot be empty.

$$
\begin{array}{lcl}
\textsc{to}\, T^{lin}(a) & = & \langle a \rangle \mathsf{tt} \\
\textsc{to}\, T^{lin}(\neg a) & = & [a] \mathsf{ff} \\
\textsc{to}\, T^{lin}(\phi \wedge \psi) & = & \textsc{to}\, T^{lin}(\phi) \wedge \textsc{to}\, T^{lin}(\psi) \\
\textsc{to}\, T^{lin}(\phi \vee \psi) & = & \textsc{to}\, T^{lin}(\phi) \vee \textsc{to}\, T^{lin}(\psi) \\
\textsc{to}\, T^{lin}(\bigcirc_I \phi) & = & x_s \,\underline{\texttt{in}}\, \langle \Sigma \rangle (x_e \,\underline{\texttt{in}}\, \langle x_e - x_s \in I \rangle \textsc{to}\, T^{lin}(\phi)) \\
\textsc{to}\, T^{lin}(\phi \,\mathcal{U}_I\, \psi) & = & x_s \,\underline{\texttt{in}}\, \mathsf{min}(Y, (x_e \,\underline{\texttt{in}}\, \langle x_e - x_s \in I \rangle \textsc{to}\, T^{lin}(\psi)) \vee \\
 & & \qquad (\textsc{to}\, T^{lin}(\phi) \wedge \langle \Sigma \rangle Y)) \\
\textsc{to}\, T^{lin}(G_I\, \phi) & = & x_s \,\underline{\texttt{in}}\, \mathsf{max}(Y, (x_e \,\underline{\texttt{in}}\, [x_e - x_s \in I] \textsc{to}\, T^{lin}(\phi)) \wedge \langle \Sigma \rangle Y)
\end{array}
$$

Figure 5 From MTL to T^{lin}.

The proof of Proposition 3.8 follows a standard result on the lack of expressivity of temporal logics (even when timed) w.r.t. formulae that involve *counting* such as $\phi_{even,I}$ (see [48, 6]). This concludes the proof of Item 2 of Theorem 3.6, and therefore the proof of Theorem 3.6. ◀

Accordingly, T^{lin} is strictly more expressive than MTL. As we will see in Section 5, $\phi_{even(b),I}$ is monitorable for violations in the general case (and also for satisfactions whenever I is bounded), which means that T^{lin} allows for a larger set of monitorable formulae than MTL.

4 Monitoring a System

In this section, we formally define timed monitors (Section 4.1), as well as the instrumentation operator (Section 4.2). In the sequel, we refer to a timed monitor simply as monitor, and to a system that we want to monitor as a *system under monitoring* (SUM).

4.1 Monitors Syntax and Semantics

$$
\begin{array}{lcl}
M, N & ::= & E \mid a.M \mid g.M \mid M + N \mid \mathsf{rec}\ X.M \mid X \mid \textsc{let}\ x\ \textsc{in}\ M \\
E & ::= & \mathsf{END} \mid \mathsf{NO} \mid \mathsf{YES}
\end{array}
$$

Figure 6 Syntax of m-terms in TMon.

We first define TMon, the set of monitor terms, hereafter abbreviated m-terms (Figure 6, where $x \in \mathcal{D}$, $a \in \Sigma$, $g \in \mathcal{B}(\mathcal{D})$ and $X \in \textsc{Var}$). $\mathsf{YES}, \mathsf{NO}, \mathsf{END}$ are special m-terms called *verdicts*, corresponding respectively to acceptance, rejection and termination (i.e., an inconclusive outcome). By abuse of notation, we write E to denote $L(E)$, the set of m-terms generated by the grammar of Figure 6 starting from the nonterminal symbol E [26].

Notions of guardedness and free variables are defined similarly as for TT^{lin} (Section 3.1). In an m-term, $\mathsf{rec}\ X.M$ (resp. $\textsc{let}\ x\ \textsc{in}\ M$) is a binder of the recursion variable X (resp. timestamp variable x). The free recursion (resp. timestamp) variables of an m-term M are given by the function $mfv : \textsc{TMon} \longrightarrow \mathcal{P}(\textsc{Var})$ (resp. $mfv_{time} : \textsc{TMon} \longrightarrow \mathcal{P}(\mathcal{D})$), defined

in the expected way. A recursion variable X is guarded in an m-term M if every occurrence of X in M is in the scope of at least one modality $(a.)$, and $M \in \text{TMon}$ is guarded if for all its subterms $\mathsf{rec}\ X.N$, X is guarded in N. We may now define the set of monitors Mon.

▶ **Definition 4.1.** *The set of monitors* Mon *is defined as the guarded m-terms in* TMon *closed vis-à-vis recursion variables:* $\text{Mon} = \{M \in \text{TMon} \mid \ M \text{ is guarded and } mfv(M) = \emptyset\}$.

For example, $\mathsf{rec}\ X.(a.\text{let}\ x\ \text{in}\ g.X + Y) \notin \text{Mon}$ because it is not closed w.r.t. recursion variable Y, whereas $\mathsf{rec}\ X.\text{let}\ y\ \text{in}\ (x - y \leq 2).a.X \in \text{Mon}$.

▶ Remark 4.2. Note that, compared to that of formulae (Definition 3.2), Definition 4.1 is more relaxed as monitors are allowed to have free timestamp variables. The practical reason behind this is illustrated later in this subsection (Example 4.6).

▶ Remark 4.3. When manipulating monitors, we assume the same conventions as in Remark 3.3; in particular we assume fresh variables are picked at each recursion step.

The semantics of *all* monitors in Mon is given by the TLTS $\mathbb{M} = \langle S_{\mathbb{M}}, \mathcal{L}, \longrightarrow_{\mathbb{M}} \rangle$. A state $m \in S_{\mathbb{M}}$ is of the form (M, v_c, v_t) where M is a monitor (in Mon), $v_c \in \mathbb{R}_{\geq 0}$ the value of *monitor time* (to record the "global" time) and $v_t \in \mathcal{V}$ a valuation (to update timestamp variables and evaluate guards). The rules of the transition relation $\longrightarrow_{\mathbb{M}} \subseteq S_{\mathbb{M}} \times \mathcal{L} \times \mathbb{M}$ are given by[4] Figure 7, where $a \in \Sigma$ and $d \in \mathbb{R}_{\geq 0}$. The rule mVer states that a verdict state, i.e., a state of the form $(M \in E, -, -)$ can perform a transition with any visible action and reach a state preserving M, i.e., verdicts are *persistent*, a central property of monitors [2]. Rule mTime corresponds to a delay transition. The remaining rules are better explained through examples.

▶ Remark 4.4. In practice, in order to monitor a formula $\phi \in T^{lin}$, we start with a *closed* monitor M (i.e., $M \in \text{Mon}$ *and* $mfv_{time}(M) = \emptyset$) at initial state $m_0 = (M, 0, \mathbf{0})$. As the results in this subsection are more generic and do not rely on these assumptions, the latter will be introduced, justified and enforced in Section 5. Without loss of generality, we still use initial states of the form $(M, 0, \mathbf{0})$ for illustration purposes.

▶ **Example 4.5.** Let $\Sigma = \{b, c\}$, $M = \text{let}\ x\ \text{in}\ ((x > 5).\mathsf{YES} + (x \leq 5).(b.\mathsf{YES} + c.\mathsf{NO}))$, and let $m_0 = (M, 0, \mathbf{0})$ be a state of $\mathbb{M}$ (the TLTS of all monitors, see above). Say M performs, from m_0, the finite run $\theta = 6\,c$. This will give the following (unique) trace: $(M, 0, \mathbf{0}) \xrightarrow{6} (M, 6, \mathbf{0}) \xrightarrow{c} (\mathsf{YES}, 6, [x = 6])$, where, from left to right:

- the delay transition $d = 6$ is allowed by rule mTime,
- the action transition c is allowed by rule mLet. This is because the state $((x > 5).\mathsf{YES} + (x \leq 5).(a.\mathsf{YES} + b.\mathsf{NO}), 6, [x = 6])$ can perform any action in Σ and reach $(\mathsf{YES}, 6, [x = 6])$ following, respectively, rules mSelL, mGuard and MVer (Figure 7).

▶ **Example 4.6.** Let $\Sigma = \{b, c\}$ and the monitor

$$M = \text{let}\ x\ \text{in}\ (\mathsf{rec}\ X.(\text{let}\ y\ \text{in}\ ((y - x \leq 5).(b.X + c.\mathsf{NO}) + (y - x > 5).\mathsf{YES})))$$

and let

$$N = \mathsf{rec}\ X.(\text{let}\ y\ \text{in}\ ((y - x \leq 5).(b.X + c.\mathsf{NO}) + (y - x > 5).\mathsf{YES}))$$

i.e., N is the subterm of M without the binder of timestamp variable x. Intuitively, the syntax of M indicates that it accepts a run (becomes YES) iff every action seen within 5 time units, relative to the global time at which the first action occurs, is a b (otherwise it

[4] We formally prove that these rules are well defined over $S_{\mathbb{M}} \times \mathcal{L} \times S_{\mathbb{M}}$.

$$\textsc{mAct}\ \frac{}{(a.M, v_c, v_t) \xrightarrow{a} (M, v_c, v_t)} \qquad \textsc{mVer}\ \frac{M \in E}{(M, v_c, v_t) \xrightarrow{a} (M, v_c, v_t)}$$

$$\textsc{mLet}\ \frac{(M, v_c, v_t[x := v_c]) \xrightarrow{a} (M', v_c, v_t')}{(\text{LET } x \text{ IN } M, v_c, v_t) \xrightarrow{a} (M', v_c, v_t')} \qquad \textsc{mTime}\ \frac{}{(M, v_c, v_t) \xrightarrow{d} (M, v_c + d, v_t)}$$

$$\textsc{mSelL}\ \frac{(M, v_c, v_t) \xrightarrow{a} (M', v_c, v_t')}{(M + N, v_c, v_t) \xrightarrow{a} (M', v_c, v_t')} \qquad \textsc{mRec}\ \frac{(M[^{\text{rec } X.M}/_X], v_c, v_t) \xrightarrow{a} (M', v_c, v_t')}{(\text{rec } X.M, v_c, v_t) \xrightarrow{a} (M', v_c, v_t')}$$

$$\textsc{mSelR}\ \frac{(N, v_c, v_t) \xrightarrow{a} (N', v_c, v_t')}{(M + N, v_c, v_t) \xrightarrow{a} (N', v_c, v_t')} \qquad \textsc{mGuard}\ \frac{(M, v_c, v_t) \xrightarrow{a} (M', v_c, v_t') \wedge v_t \models g}{(g.M, v_c, v_t) \xrightarrow{a} (M', v_c, v_t')}$$

Figure 7 SOS rules for monitors.

rejects). Let us apply the rules of $\longrightarrow_{\mathbb{M}}$, starting at $(M, 0, \mathbf{0})$, to see whether M accepts the run $\theta = 2\,b\,6\,c\,\ldots$ as it should: $(M, 0, \mathbf{0}) \xrightarrow{2} (M, 2, \mathbf{0}) \xrightarrow{b} (M', 2, [x = 2, y = 2]) \xrightarrow{6} (M', 8, [x = 2, y = 2]) \xrightarrow{c} (\mathsf{YES}, 8, [x = 2, y = 8]) \ldots$ where

$$M' = \text{LET } y \text{ IN } ((y - x \leq 5).(b.N + c.\mathsf{NO}) + (y - x > 5).\mathsf{YES})$$

applying, respectively (from left to right) rule mTime, mLet, mTime and mLet. Notice now that, in M', there is no longer the binder of timestamp variable x, and therefore x is free in M'. This is because LET x IN in M, outside of the recursion, is intended to evaluate constantly to the value of global time corresponding to the first action. This simplification is handy to deal with monitors with both recursion and guards, and justifies the relaxed nature of Definition 4.1, allowing monitors to have free timestamp variables.

Parallel monitors. In practice, a formula is often monitored through composing monitors in parallel [2]. The syntax of m-terms is extended accordingly with two parallel composition operators: $\otimes$ for *parallel conjunction* and $\oplus$ for *parallel disjunction*, adapted from [2] to the (linear-time) timed setting. The extended syntax thereof is given in Figure 8, with $\odot \in \{\otimes, \oplus\}$.

M, N	$::= E$	$\mid a.M$	$\mid g.M$	$\mid M + N$
		$\mid \text{rec } X.M$	$\mid X$	$\mid \text{LET } x \text{ IN } M$
		$\mid$	$M \otimes N$	(par. conj.)
		$\mid$	$M \oplus N$	(par. disj.)
E	$::= \mathsf{VYES}$	$\mid \mathsf{VNO}$	$\mid \mathsf{VEND}$	
VYES	$::= \mathsf{YES}$	$\mid \mathsf{VYES} \oplus M$	$\mid \mathsf{VYES} \otimes \mathsf{VYES}$	
VNO	$::= \mathsf{NO}$	$\mid \mathsf{VNO} \otimes M$	$\mid \mathsf{VNO} \oplus \mathsf{VNO}$	
VEND	$::= \mathsf{END}$	$\mid \mathsf{VEND} \odot \mathsf{VEND}$		

Figure 8 Syntax of TMon (extended with parallel composition).

Accordingly, we extend the set of verdicts by grouping acceptance, rejection and termination verdicts in the set VYES, VNO and VEND respectively. By abuse of notation, we write $E \in \{\mathsf{VYES}, \mathsf{VNO}, \mathsf{VEND}\}$ to denote $L(E)$, the set of m-terms generated by the grammar of Figure 8 starting from the nonterminal symbol E. Each of the latter is defined recursively using the corresponding verdict from a *regular* (i.e., non-parallel) m-term in the expected way. For example, an acceptance verdict VYES is reached in a parallel disjunction of two regular m-terms whenever a YES verdict is reached by one of them. The set of monitors Mon is therefore extended accordingly (by applying Definition 4.1 to TMon, as extended to parallel composition). Thanks to lifting verdict composition rules to the syntax level, the extension of the semantics of monitors in Figure 7 to parallel monitors becomes rather simple; we only need one additional rule, stating that monitors in a composition perform actions together:

$$\textsc{mPar}\ \frac{M \odot N \notin E \quad (M, v_c, v_t) \xrightarrow{a} (M', v_c, v_t') \quad (N, v_c, v_t) \xrightarrow{a} (N', v_c, v_t'')}{(M \odot N, v_c, v_t) \xrightarrow{a} (M' \odot N', v_c, v_t' \overset{v_t}{\sqcup} v_t'')}$$

where

$$v_t' \overset{v_t}{\sqcup} v_t''(x) = \begin{cases} v_t'(x) \text{ if } v_t'(x) \neq v_t(x) \\ v_t''(x) \text{ if } v_t''(x) \neq v_t(x) \\ v_t(x) \text{ otherwise} \end{cases}$$

Recall that bound timestamp variables in M and N are disjoint, and therefore the first two cases are disjoint. $\mathbb{M} = \langle S_{\mathbb{M}}, \mathcal{L}, \longrightarrow_{\mathbb{M}} \rangle$, the TLTS describing the semantics of all monitors in Mon is therefore extended to parallel monitors, with the transition relation $\longrightarrow_{\mathbb{M}}$ defined using the rules in Figure 7 and the rule mPar above.

In the sequel, we use the shorthand notation $N + A.M$ to denote the selection between monitor N and $a.M$ for all actions in $A \subseteq \Sigma$, with the convention $N + A.M = N$ if A is empty. Moreover, we write $\overline{A}$, $A \subseteq \Sigma$, to denote the set $\Sigma \setminus A$ and similarly $\overline{a}$, $a \in \Sigma$ to denote $\Sigma \setminus \{a\}$. Using these compact notations, we can write e.g., for $\Sigma = \{a_1, a_2, a_3, a_4\}$, $a_2.M + \overline{a_2}.\mathsf{NO}$ instead of $a_2.M + a_1.\mathsf{NO} + a_3.\mathsf{NO} + a_4.\mathsf{NO}$.

Verdict persistence. As mentioned earlier, all monitors in Mon are verdict persistent.

▶ **Lemma 4.7** (Verdict presistence). *For all $(M, v_c, v_t), (N, v_c', v_t') \in S_{\mathbb{M}}$ and finite weak run $\theta \in \mathcal{L}^*$, if $M \in E$ then: $(M, v_c, v_t) \xrightarrow{\theta}\!\!\!\!\rightarrow (N, v_c', v_t')$ implies $N = M$.*

Verdict states. Thanks to verdict persistence, once a monitor reaches a *verdict state*, i.e., a state whose first component is a verdict, the information within such state become irrelevant regardless of what the monitor will perform in the future, at the exception of the verdict itself (which will remain unchanged). Therefore, we may denote by yes, no, end any verdict state in $(M, v_c, v_t) \in S_{\mathbb{M}}$ such that $M \in \mathsf{VYES}$, $M \in \mathsf{VNO}$, $M \in \mathsf{VEND}$, respectively. Strictly speaking, yes, no, end may be viewed as equivalent classes grouping states with acceptance, rejection and termination verdicts, respectively. In order to simplify the writing, we abuse notation by defining an equality instead. For all $(M, v_c, v_t), (N, v_c', v_t') \in S_{\mathbb{M}}$:

$$(M, v_c, v_t) = (N, v_c', v_t') = \mathsf{yes} \text{ iff } M \in \mathsf{VYES} \text{ and } N \in \mathsf{VYES}$$

$$(M, v_c, v_t) = (N, v_c', v_t') = \mathsf{no} \text{ iff } M \in \mathsf{VNO} \text{ and } N \in \mathsf{VNO}$$

$$(M, v_c, v_t) = (N, v_c', v_t') = \mathsf{end} \text{ iff } M \in \mathsf{VEND} \text{ and } N \in \mathsf{VEND}$$

For example, we write $m \xrightarrow{\theta}\!\!\!\!\rightarrow \mathsf{yes}$ if there exists a state (N, v_c, v_t) with $N \in \mathsf{VYES}$ reachable from m along θ, and similarly $m \xrightarrow{\theta}\!\!\!\!\rightarrow_{inev} \mathsf{yes}$ if all states (N, v_c, v_t) reachable along θ from m satisfy $N \in \mathsf{VYES}$.

4.2 Instrumentation

The notion of a monitor "performing an action" is in fact misleading, since a monitor is rather a *passive entity* that *follows* an SUM. We need an operator to compose a monitor with an SUM accordingly; this is known as *instrumentation*. We define next the rules of the instrumentation operator $\lhd$, adapted to the timed settings from [2]. Hereafter, recall $\mathbb{M} = \langle S_{\mathbb{M}}, \mathcal{L}, \longrightarrow_{\mathbb{M}} \rangle$ the TLTS describing the behaviour of all monitors, and let $P = \langle S_P, \text{LAB}, \longrightarrow_P \rangle$ be an arbitrary TLTS describing the behaviour of an SUM. That is, we do not assume any particular form of states in S_P or rules in $\longrightarrow_P$, since the SUM can be e.g., a blackbox system.

▶ **Definition 4.8** (Instrumentation). *We define the* monitored system *using the instrumentation operator $\lhd$ as follows: $\mathbb{M} \lhd P = \langle S_{\mathbb{M}} \times S_P, \text{LAB}, \longrightarrow_{\lhd} \rangle$. The transition relation $\longrightarrow_{\lhd}$ is defined in Figure 9, where $m, m' \in S_{\mathbb{M}}$, $p, p' \in S_P$, $\alpha \in \mathcal{L}$ and $a \in \Sigma$. To save notations, we use $m \lhd p$ for $(m, p) \in S_{\mathbb{M}} \times S_P$.*

$$\frac{p \xrightarrow{\alpha}_P p' \quad m \xrightarrow{\alpha}_{\mathbb{M}} m'}{m \lhd p \xrightarrow{\alpha}_{\lhd} m' \lhd p'}\ \textsc{iMon} \qquad \frac{p \xrightarrow{\tau}_P p'}{m \lhd p \xrightarrow{\tau}_{\lhd} m \lhd p'}\ \textsc{iAsy}$$

$$\frac{p \xrightarrow{a}_P p' \quad m \xrightarrow{a}\!\!\!\!\!\!\rightarrow_{\mathbb{M}} \quad m = (N, v_c, v_t)}{m \lhd p \xrightarrow{a}_{\lhd} (\mathsf{END}, v_c, v_t) \lhd p'}\ \textsc{iTer}$$

Figure 9 SOS rules for Instrumentation.

Rule iMon states that the monitor must follow any action or delay transition performed by the SUM, whereas rule iAsy corresponds to the SUM performing a τ transition. Finally, if the SUM may perform an action a and the monitor may not, rule iTer allows the SUM to perform a while forcing the monitor to reach a verdict state end.

▶ Remark 4.9. Note that iTer is defined only for actions and not for time delays. This is because a monitor can always perform time delays (rule mTime, Figure 7) and therefore it can always follow the SUM if it performs a delay.

Preserving the SUM behaviours. We prove, through strong timed bisimulation [1], that the instrumentation operator *preserves* the behaviours of the SUM, i.e., instrumenting any SUM P with any monitor M preserves *exactly* all the behaviours of P. Accordingly, given a finite run $\theta \in \text{LAB}^*$, performed by the SUM, we say that a monitor *consumes* the weak run $\theta' \in \mathcal{L}^*$ corresponding to θ (Section 2.1).

▶ **Example 4.10.** Let us consider the monitor M, starting at state $m_0 = (M, 0, \mathbf{0})$ (Example 4.5), and an SUM performing a finite run $\theta = 2\,\tau\,4\,\tau\,c$ from state p to state p'. We have then $m_0 \lhd p \xrightarrow{\theta} \mathsf{yes} \lhd p'$, which corresponds to M consuming $\theta' = 6\,c$, the weak finite run corresponding to θ. Informally, the monitor M checks the following property at runtime: "if the first consumed action occurs within 5 time units, then such action must be a b".

5 Monitorability of T^{lin}

In this section, we present the monitorability results on T^{lin}. In particular, we characterise its largest violation-, satisfaction- and complete-monitorable fragments.

5.1 Setting the framework

5.1.1 Abstracting Away the Instrumentation Operator

To alleviate notations, we abstract away the instrumentation operator through considering only *reactive* monitors, i.e., monitors that can follow any run from any state of any SUM (i.e., consume any run in $\mathcal{L}^*$). This choice should introduce no loss of generality: in Section 5.2, we show how it is rather straightforward to synthesise a reactive monitor that intuitively "corresponds" to (verifies online) any formula in T^{lin}.

▶ **Definition 5.1** (Monitor Reactivity (extended from [2])). *A monitor $M \in \textsc{Mon}$ is reactive iff* $\forall \theta \in \mathcal{L}^*, v_c \in \mathbb{R}_{\geq 0}, v_t \in \mathcal{V} : (M, v_c, v_t) \xrightarrow{\theta}\!\!\!\!\rightarrow$

Since we are in a linear-time setting, i.e., the set of behaviours of an SUM equals the set of its infinite runs, all behaviours of all SUMs simply collapse into all infinite runs in $\textsc{Lab}^\omega$. We write accordingly $m \xrightarrow{\theta}\!\!\!\!\rightarrow m'$ to denote a reactive monitor consuming weak run θ. This way, we can simply abstract away SUMs and the instrumentation operator, the former replaced with the sum of all infinite runs in $\textsc{Lab}^\omega$ and the latter collapsing into simple consumption of weak runs that is always possible thanks to the reactivity assumption.

5.1.2 Monitor Initial State

Intuitively, in order to assess whether M can monitor ϕ, monitoring should start at state $m_0 = (M, 0, \mathbf{0})$. That is, monitor M is *initialised* at state m_0, at which its time (monitor time) starts. Since monitor time may only increase (through consuming delays), it refers to the relative time since the monitoring of ϕ started. Examples of a monitor M starting at state m_0 have been already discussed within the previous section, e.g., in Example 4.5.

5.1.3 Timed Words, Weak Runs and Time Divergence

First, we make an important, yet natural assumption on infinite timed words, namely *time divergence*. To explain this, let us consider the following infinite timed word: $\pi = (a_1, \frac{1}{2}), (a_2, \frac{3}{4}), (a_3, \frac{7}{8}), (a_4, \frac{15}{16}), \ldots$ One can notice that π corresponds to a weak run with an initial delay of $\frac{1}{2}$, and each subsequent delay equalling half of the previous delay. This results in the timestamps of π increasing but never beyond absolute time 1 (since $\sum_{k=1}^{\infty}(\frac{1}{2^k}) = 1$). The infinite timed word π is *time convergent*, and it is clear to see that there is no "real" system that can produce such timed word; for the simple reason that time always diverges in reality (see e.g., [23, 8][18, Sect.2]). An infinite timed word π is time divergent iff its timestamps are unbounded, formally [23, Sect. 4], [7, Def. 1]:

$$\forall x \in \mathbb{R}, \exists i \in \mathbb{N} : t_i(\pi) > x \tag{3}$$

We restrict ourselves accordingly to the set $T\Sigma^\omega_{div} = \{\pi \in T\Sigma^\omega \mid \pi \text{ satisfies (3) }\}$ henceforth.

Second, as we will see later on, central notions on monitorability, such as monitor soundness and completeness, gravitate around the ability to reach a verdict, w.r.t. whether an infinite timed word $\pi \in T\Sigma^\omega_{div}$ is in the semantics of the monitored formula, based on

consuming a finite prefix of π. We need therefore to define the notion of consuming a finite timed word. Since (i) we already defined the notion of consuming a weak run (Section 4), and (ii) a finite timed word necessarily corresponds to a finite weak run in $\mathcal{L}^*$ (Section 2.2), we simply need to define a function to retrieve the latter from the former.

▶ **Definition 5.2** (From finite timed words to finite weak runs)**.**

$$\textsc{Trn} : T\Sigma^* \longrightarrow \mathcal{L}^*$$

where $\textsc{Trn}(\pi) = d_1 a_1 d_2 a_2 \ldots$ *with* $d_i = t_i - t_{i-1}$ *and* $t_0 = 0$*, and the convention that* $\textsc{Trn}(\pi)$ *gives the empty weak run* ϵ *if* π *is an empty timed word.*

In the remainder of this paper, by abuse of terminology, we often say a monitor "consumes (the finite timed word) π" instead of "consumes (the finite weak run) $\textsc{Trn}(\pi)$"

▶ **Example 5.3.** Consider the finite timed word $\pi = (a, 1.7)(b, 1.7)(b, 5.7)$. The corresponding weak run is obtained using Definition 5.2: $\textsc{Trn}(\pi) = 1.7\, a\, 0\, b\, 4\, b$.

▶ Remark 5.4. Note that time divergence does not pose any particular restriction on finite timed words, i.e., $\{Pf(\pi) \mid \pi \in T\Sigma^\omega_{div}\} = T\Sigma^*$. In words, any finite timed word (in $T\Sigma^*$) can be retrieved as a prefix of a time-divergent infinite timed word, see [8].

5.1.4 Monitor Soundness and Completeness

We now define the central monitor properties of *soundness* and *completeness* extended from [2, Definition 4.1] to our timed setting.

▶ **Definition 5.5** (Monitor Soundness)**.** *A reactive monitor* M *is sound for a formula* $\phi \in T^{lin}$ *if, for every finite timed word* $\pi \in T\Sigma^*$*:*

- $m_0 \overset{\textsc{Trn}(\pi)}{\longrightarrow\!\!\!\!\!\rightarrow}$ yes *implies* $(\forall \pi' \in Ext(\pi) : \pi' \in [\![\phi]\!])$
- $m_0 \overset{\textsc{Trn}(\pi)}{\longrightarrow\!\!\!\!\!\rightarrow}$ no *implies* $(\forall \pi' \in Ext(\pi) : \pi' \notin [\![\phi]\!])$

Sound monitors have the important property of *consistency*.

▶ **Definition 5.6** (Consistent monitors)**.** *A reactive monitor* M *is consistent iff* $\forall \pi \in T\Sigma^*$:

$m_0 \overset{\textsc{Trn}(\pi)}{\longrightarrow\!\!\!\!\!\rightarrow}$ yes *implies* $m_0 \overset{\textsc{Trn}(\pi)}{\not\longrightarrow\!\!\!\!\!\rightarrow}$ no *and* $m_0 \overset{\textsc{Trn}(\pi)}{\longrightarrow\!\!\!\!\!\rightarrow}$ no *implies* $m_0 \overset{\textsc{Trn}(\pi)}{\not\longrightarrow\!\!\!\!\!\rightarrow}$ yes

▶ **Lemma 5.7.** *If* M *is sound for some* $\phi \in T^{lin}$ *then* M *is consistent.*

▶ **Definition 5.8** (Monitor Completenness)**.** *Let* $M \in \textsc{Mon}$ *be a reactive monitor and* $\phi \in T^{lin}$ *a formula.* M *is:*

- *violation complete for* ϕ *iff for all* $\pi \in T\Sigma^\omega_{div}$*,* $\pi \notin [\![\phi]\!]$ *implies* $\exists \pi' \in Pf(\pi) : m_0 \overset{\textsc{Trn}(\pi')}{\longrightarrow\!\!\!\!\!\rightarrow}$ no
- *satisfaction complete for* ϕ *iff for all* $\pi \in T\Sigma^\omega_{div}$*,* $\pi \in [\![\phi]\!]$ *implies* $\exists \pi' \in Pf(\pi) : m_0 \overset{\textsc{Trn}(\pi')}{\longrightarrow\!\!\!\!\!\rightarrow}$ yes
- *complete for* ϕ *if it is both violation and satisfaction complete for* ϕ*.*

Strong completeness. In practice, it is desirable to have *reliable* monitors, i.e., monitors exhibiting a stronger property than violation/satisfaction completeness. Let us explain this through an example.

▶ **Example 5.9.** Let $\Sigma = \{b, c\}$ and consider the monitor

$$M' = \text{LET } x \text{ IN } ((x > 5).\mathsf{YES} + (x \leq 5).(b.\mathsf{YES} + c.\mathsf{NO} + b.\mathsf{END} + c.\mathsf{END}))$$

a slight modification of the monitor M in Example 4.5. M' is reactive, and it is rather easy to see that M' is sound and complete for the following T^{lin} formula:

$$\phi = x \,\underline{\mathtt{in}}[x \leq 5]\langle b\rangle \mathsf{tt}$$

stating that if the first timestamp is smaller than 5, then the action must be a b. For soundness, if verdict state no is reached, then the first timed event the monitor saw was (c, t) with $t \leq 5$, and it is guaranteed that any infinite extension of the finite timed word (c, t) will be outside the semantics of ϕ. (The case where yes is reached is analogous). For violation completeness, any infinite timed word $\pi \notin [\![\phi]\!]$ starts necessarily with a timed event (c, t) with $t \leq 5$, and given any prefix π' of π of the form (c, t), $t \leq 5$, no is reachable from $(M', 0, \mathbf{0})$ along π'. (The reasoning for satisfaction completeness is analogous). In practice, however, M' is not a reliable monitor for ϕ, as it may reach end instead of no after consuming e.g., $(c, 2)$ (and dually, to reach end instead of yes after consuming e.g., $(b, 3)$). We propose a stronger definition of completeness next.

▶ **Definition 5.10** (Strong Completenness)**.** *Let $M \in \text{MON}$ be a reactive monitor and $\phi \in T^{lin}$ a formula. M is:*

- *strongly violation complete for ϕ iff for all $\pi \in T\Sigma^{\omega}_{div}$, $\pi \notin [\![\phi]\!]$ implies $\exists \pi' \in Pf(\pi)$: $m_0 \overset{\text{TRN}(\pi')}{\longrightarrow\!\!\!\!\!\rightarrow}_{inev} \mathsf{no}$*
- *strongly satisfaction complete for ϕ iff for all $\pi \in T\Sigma^{\omega}_{div}$, $\pi \in [\![\phi]\!]$ implies $\exists \pi' \in Pf(\pi)$: $m_0 \overset{\text{TRN}(\pi')}{\longrightarrow\!\!\!\!\!\rightarrow}_{inev} \mathsf{yes}$*

M is strongly complete for ϕ if it is both strongly violation and strongly satisfaction complete for ϕ.

In brief, the key element in the above definition is the inevitability of verdicts. That is, if a strongly satisfaction- (resp. violation-) complete monitor for some $\phi \in T^{lin}$ is given an infinite timed word π that is in (resp. outside of) the semantics of ϕ, then it will inevitably reach verdict yes (resp. no) after consuming a finite prefix of π. Therefore, such monitor has the important feature of reliability in practice as discussed in Example 5.9 above.

▶ **Example 5.11.** In Example 5.9, the monitor M' is neither strongly satisfaction nor strongly violation complete for ϕ.

▶ Remark 5.12. In this paper, the notion of violation- or satisfaction-monitorable formulae will still rely on the existence of a monitor that obeys soundness (Definition 5.5) and the weaker version of completeness (Definition 5.8). Strong (satisfaction/violation) completeness (Definition 5.10) will be a property of *our monitors*, which we synthesise automatically from T^{lin}, for formulae within the satisfaction/violation/complete monitorable fragments.

5.2 Sound monitor synthesis

In this subsection, we define a compiler from T^{lin} to monitors and prove that, for every formula $\phi \in T^{lin}$, our compiler outputs a monitor that is sound for ϕ (Proposition 5.15).

The compiler $(\!|-|\!) : T^{lin} \longrightarrow \text{MON}$ is defined in Figure 10 (note that monitors generated by $(\!|-|\!)$ are necessarily closed since $(\!|-|\!)$ is applied only to formulae, i.e., elements of T^{lin}). The rules of $(\!|-|\!)$ are rather simple and straightforward, operating on the syntax of the formula

$(\!\lvert \mathtt{tt} \rvert\!)$	$=$	YES	$(\!\lvert \mathtt{ff} \rvert\!)$	$=$	NO
$(\!\lvert x \,\underline{\mathtt{in}}\, \phi \rvert\!)$	$=$	LET x IN $(\!\lvert \phi \rvert\!)$	$(\!\lvert X \rvert\!)$	$=$	X
$(\!\lvert [a]\phi \rvert\!)$	$=$	$a.(\!\lvert \phi \rvert\!) + \overline{a}.\mathsf{YES}$	$(\!\lvert \langle a \rangle \phi \rvert\!)$	$=$	$a.(\!\lvert \phi \rvert\!) + \overline{a}.\mathsf{NO}$
$(\!\lvert [g]\phi \rvert\!)$	$=$	$g.(\!\lvert \phi \rvert\!) + \overline{g}.\mathsf{YES}$	$(\!\lvert \langle g \rangle \phi \rvert\!)$	$=$	$g.(\!\lvert \phi \rvert\!) + \overline{g}.\mathsf{NO}$
$(\!\lvert \phi \wedge \psi \rvert\!)$	$=$	$(\!\lvert \phi \rvert\!) \otimes (\!\lvert \psi \rvert\!)$	$(\!\lvert \phi \vee \psi \rvert\!)$	$=$	$(\!\lvert \phi \rvert\!) \oplus (\!\lvert \psi \rvert\!)$
$(\!\lvert \mathsf{max}(X, \phi) \rvert\!)$	$=$	rec $X.(\!\lvert \phi \rvert\!)$	$(\!\lvert \mathsf{min}(X, \phi) \rvert\!)$	$=$	rec $X.(\!\lvert \phi \rvert\!)$

Figure 10 $(\!\lvert - \rvert\!)$ rules from T^{lin} to monitors, defined recursively on subterms of formulae.

provided as an input. For example, if the formula is $[a]\langle b\rangle\mathtt{tt}$, whose semantics is "if the first action is a, then the second must be a b" (Section 3.1), then we want a monitor that consumes the first action, and becomes verdict YES if such action is not a, and monitors that the second action is a b otherwise. We obtain thus the monitor $(\!\lvert [a]\langle b\rangle\mathtt{tt} \rvert\!) = a.(b.\mathsf{YES} + \overline{b}.\mathsf{no}) + \overline{a}.\mathsf{YES}$, which corresponds to the description above (under the rules defined in Section 4.1).

▶ **Example 5.13.** The monitor M in Example 4.5 is compiled from the formula: $x\,\underline{\mathtt{in}}[x \leq 5]\langle b\rangle\mathtt{tt}$ over $\Sigma = \{b, c\}$. More complex examples will follow within the next subsections.

▶ **Lemma 5.14.** *For all $\phi \in T^{lin}$, $(\!\lvert \phi \rvert\!)$ is reactive.*

▶ **Proposition 5.15.** *For all $\phi \in T^{lin}$, $(\!\lvert \phi \rvert\!)$ is sound for ϕ.*

5.3 Complete Monitorable Fragment

In this subsection, we identify CT^{lin}, a subset of T^{lin}, and prove its complete monitorability (Theorem 5.26).

▶ **Definition 5.16** (Complete-monitorable formulae)**.** *A formula $\phi \in T^{lin}$ is complete monitorable iff there exists a reactive monitor M that is sound and complete for it. A fragment $\Gamma \subseteq T^{lin}$ is complete monitrorable iff each $\phi \in \Gamma$ is complete monitorable.*

It has been shown in [2] that the recursion-free fragment of untimed linear-time μ-calculus (μ-HML) is its largest complete-monitorable fragment. We show that this result does not hold in the timed setting, i.e., the maximal set of complete-monitorable formulae in T^{lin} is strictly larger than that of μ-HML. To explain this, let us first prove a handy lemma. Recall that $\mathcal{D}$ is the set of timestamp variables and $\mathcal{V}$ all valuation functions over $\mathcal{D}$ (Section 3.1).

▶ **Lemma 5.17.** *Let $y \in \mathcal{D}$ and $n \in \mathbb{N}$. For all $\pi \in T\Sigma^{\omega}_{div}$, $x \in \mathcal{D} \setminus \{y\}$, $v \in \mathcal{V}$, $g \in \mathcal{B}(\mathcal{D})$:*
1. *$\exists i \in \mathbb{N}_{>0}$ such that $v[x := t_i(\pi)] \not\models (x - y \prec n) \wedge g$,*
2. *$\forall i \in \mathbb{N}_{>0}$: $v[x := t_i(\pi)] \not\models (x - y \prec n) \wedge g$ implies $\forall j \geq i$: $v[x := t_i(\pi)] \not\models (x - y \prec n) \wedge g$.*

Let us depict the consequences of the above lemma through a few examples. To ease the explanation, we sometimes use MTL formulae as a support.

▶ **Example 5.18.** Consider the formula $\phi = F\,b$ (Section 2.4), meaning "action b will be seen in the future" (recall that this is the special case of $F_I\,b$ with $I = [0, \infty)$). Clearly, there is no violation-complete monitor for ϕ, because there is no way for a monitor to say no

based on a finite timed word not containing b (since a b can occur later on). In contrast, ϕ is monitorable for satisfactions: it suffices for the monitor to see a b to say yes. It follows that $\phi' = \text{TO}\,T^{lin}(\phi) = \mathsf{min}(Y, \langle b\rangle \mathtt{tt} \vee \langle\Sigma\rangle Y)$ (recalling that $F\,b$ is syntactic sugar for $\top\,\mathcal{U}_I\,b$, then using the function $\text{TO}\,T^{lin}$, Section 3.2) is satisfaction monitorable, but not violation monitorable. Now, in a (time) *constrained* setting, i.e., whenever we use a bounded interval I (with $\mathcal{RB}(I) \in \mathbb{N}$, Definition 2.4), ϕ becomes both satisfaction and violation monitorable, and therefore complete monitorable. Consider the example $\phi_c = F_I\,b$ with $I = [5, 8]$. Its compilation into T^{lin} gives the following formula:

$$\phi'_c = x_s \,\underline{\mathtt{in}}\, \mathsf{min}(Y, (x_e \,\underline{\mathtt{in}} \langle x_e - x_s \in [5,8]\rangle \langle b\rangle \mathtt{tt}) \vee \langle\Sigma\rangle Y)$$

which means "eventually b will occur within [5,8]". The guard $x_e - x_s \in [5, 8]$ is of the form $x - y \prec n \wedge g$ where $y = x_s$ (constant), $n = 8$ (constant), $x = x_e$, $\prec = \leq$, $g = x_e - x_s \geq 5$, and x_e diverges after a finite number of recursions (because time diverges and Y is guarded by $\langle\Sigma\rangle$). Therefore, by Lemma 5.17, $x_e - x_s \in [5, 8]$ will become false after a finite number of recursions, and will remain false indefinitely afterwards. It follows that the monitor can safely say no when the guard becomes false (which means that the consumed prefix until then did not contain any action b between 5 and 8).

▶ **Example 5.19.** Let us now consider the more complex *leads to* formula $\psi = G\,(b \Rightarrow F\,c)$ stipulating that "whenever an action b occurs, an action c follows in the future". As explained in Section 1, this formula cannot be monitored (neither for satisfactions nor for violations). Now consider the *constrained bounded response* $\psi_c = G_I\,(b \Rightarrow F_{I'}\,c)$ with $I = [1, 3]$ and $I' = [2, 5]$; i.e., in which we constrain both temporal operators with bounded intervals. The compilation of ψ_c into T^{lin} (Figure 5) gives the following formula:

$$\psi'_c = x_s \,\underline{\mathtt{in}}\, \mathsf{max}(Y, (x_e \,\underline{\mathtt{in}} [x_e - x_s \in [1,3]]([b]\mathtt{ff} \vee \zeta)) \wedge (\langle\Sigma\rangle Y))$$

with:

$$\zeta = x'_s \,\underline{\mathtt{in}}\, \mathsf{min}(X, (x'_e \,\underline{\mathtt{in}} \langle x'_e - x'_s \in [2,5]\rangle \langle c\rangle \mathtt{tt}) \vee (\langle\Sigma\rangle X))$$

The constrained bounded response formula ψ'_c is also complete monitorable. Indeed, if no b occurs between 1 and 3, the monitor can safely say yes (when the right bound of $[1, 3]$ is exceeded). Otherwise, for every b occurring between 1 and 3, the monitor checks whether a c occurs between 2 and 5 (relative to the occurrence of such b), and says no if it is not the case. Since $[2, 5]$ is bounded, the monitor can decide as soon as it consumes an action more than 5 time units since the last occurrence of b between 1 and 3.

▶ **Example 5.20.** Let us now consider the formulae given in Example 3.5, not expressible in MTL. The formula

$$\phi_{even(b)} = \mathsf{max}(X, \langle\Sigma\rangle\langle b\rangle X)$$

is violation monitorable (a monitor can say no as soon as it sees an action at an even position different than b), but is not satisfaction monitorable. Now, following the same reasoning as in the previous examples (applying Lemma 5.17), the formula

$$\phi_{even(b),[3,5]} = x_s \,\underline{\mathtt{in}}\, \mathsf{max}(X, \langle\Sigma\rangle(x_e \,\underline{\mathtt{in}} \langle x_e - x_s \in [3,5]\rangle \langle b\rangle X \vee \bigvee_{a\in\Sigma} y^a \,\underline{\mathtt{in}} \langle y^a - x_s \notin I\rangle \langle a\rangle X))$$

is complete monitorable.

Syntactic characterisation. In order to characterise a sufficiently expressive syntax for complete-monitorable formulae, we remark the following. Formulae ϕ'_c (Example 5.18) is equivalent to the following formulae:

$$x_s \,\underline{\mathtt{in}}\, \mathsf{min}(Y, (x_e \,\underline{\mathtt{in}}\langle x_e - x_s \in [5,8]\rangle\langle b\rangle\mathtt{tt}) \vee \bigvee_{a\in\Sigma} y^a \,\underline{\mathtt{in}}\langle y^a - x_s \leq 8\rangle\langle a\rangle Y)$$

because when the monitor time exceeds the right bound of $[5,8]$, then there is no need to check if $\langle b\rangle\mathtt{tt}$ is satisfied, and therefore recursion can be prevented (and the monitor can say no at this stage). Formula ψ'_c in Example 5.19 can be transformed similarly to the following equivalent formula:

$$x_s \,\underline{\mathtt{in}}\, \mathsf{max}(Y, (x_e \,\underline{\mathtt{in}}[x_e - x_s \in [1,3]]([b]\mathtt{ff} \vee \zeta')) \wedge \bigvee_{a\in\Sigma} (y^a \,\underline{\mathtt{in}}[y^a - x_s \leq 3]\langle a\rangle Y))$$

where:

$$\zeta' = x'_s \,\underline{\mathtt{in}}\, \mathsf{min}(X, (x'_e \,\underline{\mathtt{in}}\langle x'_e - x'_s \in [2,5]\rangle\langle c\rangle\mathtt{tt}) \vee \bigvee_{a\in\Sigma} (z^a \,\underline{\mathtt{in}}\langle z^a - x'_s \leq 5\rangle\langle a\rangle X))$$

As for formula $\phi_{even(b),[3,5]}$ in Example 5.20, it can be rewritten similarly as follows:

$$x_s \,\underline{\mathtt{in}}\, \mathsf{max}(X, \langle\Sigma\rangle(x_e \,\underline{\mathtt{in}}\langle x_e - x_s \in [3,5]\rangle\langle b\rangle X \vee \bigvee_{a\in\Sigma} y^a \,\underline{\mathtt{in}}(\langle y^a - x_s < 3\rangle\langle a\rangle X \vee \langle y^a - x_s > 5\rangle\langle a\rangle\mathtt{tt})))$$

since there is no need to recurse after the right bound of $[3,5]$ is exceeded (in such case, the timed word is in the semantics of the formula).

More generally, in a complete-monitorable formula ϕ, every subterm $\mathsf{min}(X,\phi)$ or $\mathsf{max}(X,\psi)$ is *constrained*. That is, in every subterm of ϕ, every occurrence of recursion variable X is preceded by $x \,\underline{\mathtt{in}}\langle g\rangle\langle a\rangle$ or $x \,\underline{\mathtt{in}}[g]\langle a\rangle$ such that g becomes eventually false, and $a \in \Sigma$. To simplify the proofs, we use a straightforward syntactic manipulation in order to require recursion variables to be immediately preceded by $x \,\underline{\mathtt{in}}\langle g\rangle$ or $x \,\underline{\mathtt{in}}[g]$: every $x \,\underline{\mathtt{in}}\langle g\rangle\langle a\rangle$ preceding a recursion variable becomes $x \,\underline{\mathtt{in}}\langle g\rangle\langle a\rangle x' \,\underline{\mathtt{in}}\langle g'\rangle$ where in g' the timestamp variable x is replaced with x'. For example, the formula

$$\zeta' = x'_s \,\underline{\mathtt{in}}\, \mathsf{min}(X, (x'_e \,\underline{\mathtt{in}}\langle x'_e - x'_s \in [2,5]\rangle\langle c\rangle\mathtt{tt}) \vee \bigvee_{a\in\Sigma} (z^a \,\underline{\mathtt{in}}\langle z^a - x'_s \leq 5\rangle\langle a\rangle X))$$

above is semantically equivalent to

$$\zeta'' = x'_s \,\underline{\mathtt{in}}\, \mathsf{min}(X, (x'_e \,\underline{\mathtt{in}}\langle x'_e - x'_s \in [2,5]\rangle\langle c\rangle\mathtt{tt}) \vee \bigvee_{a\in\Sigma} (z^a \,\underline{\mathtt{in}}\langle z^a - x'_s \leq 5\rangle\langle a\rangle y^a \,\underline{\mathtt{in}}\langle y^a - x'_s \leq 5\rangle X))$$

In order to syntactically specify that some guard becomes unsatisfied after a finite number of recursions, we introduce cLVar, an infinite subset of logical variables Var, such that a variable that belongs to cLVar must appear in a particular form within the scope of a fixpoint operator.

▶ **Definition 5.21** (Complete-monitorable fragment). *A formula $\phi \in T^{lin}$ is in the set $CT^{lin} \subset T^{lin}$ of complete-monitorable formulae iff ϕ is generated by the grammar in Figure 11 (with $n \in \mathbb{N}$, $g \in \mathcal{B}(\mathcal{D})$, $x, y \in \mathcal{D}$ and $X \in$ cLVar)* and *for all $\mathsf{min}(X,\psi)$, $\mathsf{max}(X,\psi)$ subterms of ϕ, y is free in ψ[5]. A subterm $\mathsf{max}(X,\phi)$ (resp. $\mathsf{min}(X,\phi)$) of a formula in CT^{lin} is called a constrained max (resp. constrained min).*

$$
\begin{array}{lcllllll}
\phi, \psi & ::= & \mathsf{tt} & \mid \mathsf{ff} & \mid \phi \vee \psi & \mid \phi \wedge \psi & \mid x \,\underline{\mathtt{in}}\, \phi \\
 & & \mid \langle a \rangle \phi & \mid [a]\phi & \mid \langle g \rangle \phi & \mid [g]\phi & \\
 & & \mid \mathsf{max}(X, \phi) & \mid \mathsf{min}(X, \phi) & \mid x \,\underline{\mathtt{in}}\, [g_x] X & \mid x \,\underline{\mathtt{in}}\, \langle g_x \rangle X & \\
g_x & ::= & x - y \prec n & \mid g_x \wedge g &
\end{array}
$$

Figure 11 Grammar to generate formulae (closed and guarded terms) of CT^{lin}.

To prove that CT^{lin} is complete monitorable, we show that $(\!|\phi|\!)$ is sound and *strongly* complete for every $\phi \in CT^{lin}$. Strong completeness allows us to both guarantee completeness *and* show that our monitors (synthesised by $(\!|-|\!)$) are reliable for CT^{lin}.

▶ **Proposition 5.22.** *For all $\phi \in CT^{lin}$, $(\!|\phi|\!)$ is sound for ϕ.*

Proof. Since $CT^{lin} \subset T^{lin}$, we conclude by Proposition 5.15. ◀

▶ **Proposition 5.23.** *For all $\phi \in CT^{lin}$, $(\!|\phi|\!)$ is strongly satisfaction complete for ϕ.*

▶ **Proposition 5.24.** *For all $\phi \in CT^{lin}$, $(\!|\phi|\!)$ is strongly violation complete for ϕ.*

▶ **Example 5.25.** Consider the following formula:

$$\phi = x_s \,\underline{\mathtt{in}}\, \mathsf{min}(X, (x \,\underline{\mathtt{in}}\, \langle x - x_s \in (3,7] \rangle \langle b \rangle y \,\underline{\mathtt{in}}\, \langle y - x \in [0,2) \rangle \langle c \rangle \mathsf{tt}) \vee \bigvee_{a \in \Sigma} z^a \,\underline{\mathtt{in}}\, \langle z^a - x_s \leq 7 \rangle \langle a \rangle X)$$

easily expressible in CT^{lin} (see above), which expresses the following property "there exists an occurrence of b within interval $(3,7]$ that is immediately followed by a c within the interval $[0,2)$"; where "immediately followed by" means that there are no other actions occurring in between. In other words, the semantics of ϕ contains all infinite timed words π such that there exists a position i with $a_i(\pi) = b$ *and* $t_i(\pi) - t_1(\pi) \in (3,7]$ *and* $a_{i+1}(\pi) = c$ *and* $t_{i+1}(\pi) - t_i(\pi) \in [0,2)$. The compilation of ϕ into a monitor gives:

$$M = (\!|\phi|\!) = \textsc{let}\ x_s\ \textsc{in}\ \mathsf{rec}\ X.((\textsc{let}\ x\ \textsc{in}\ (((x - x_s \in (3,7]).(b.N + \overline{b}.\mathsf{NO})) +$$
$$(x - x_s \notin (3,7]).\mathsf{NO})) \oplus (\oplus_{a \in \Sigma}\ \textsc{let}\ z^a\ \textsc{in}\ (((z^a - x_s \leq 7).(a.X + \overline{a}.\mathsf{NO})) + (z^a - x_s > 7).\mathsf{NO})))$$

where

$$N = \textsc{let}\ y\ \textsc{in}\ (((y - x \in [0,2)).(c.\mathsf{YES} + \overline{c}.\mathsf{NO})) + (y - x \notin [0,2)).\mathsf{NO})$$

and M is sound and strongly complete for ϕ. For e.g., strong completeness, for any $\pi \in T\Sigma^{\omega}_{div}$ in the semantics of ϕ (resp. not in the semantics of ϕ), the monitor will inevitably reach a verdict state yes (resp. no) after consuming a finite prefix of π. For example, for any infinite timed word $\pi_1.\pi$ with $\pi_1 = (a,1)(b,8)(c,9)$ and $\pi \in T\Sigma^{\omega}_{\vec{\pi_1}}$ (resp. $\pi_2.\pi$ with $\pi_2 = (a,2)(b,6)(c,8)$ and $\pi \in T\Sigma^{\omega}_{\vec{\pi_2}}$) we have $m_0 \xrightarrow{\textsc{Trn}(\pi_1)}{}\!\!\!\!\rightarrow_{inev} \mathsf{yes}$ (resp. $m_0 \xrightarrow{\textsc{Trn}(\pi_2)}{}\!\!\!\!\rightarrow_{inev} \mathsf{no}$).

▶ **Theorem 5.26.** *CT^{lin} is complete monitorable.*

Proof. For every $\phi \in CT^{lin}$, $(\!|\phi|\!)$ is sound (Proposition 5.22), satisfaction complete (implied by Proposition 5.23) and violation complete (implied by Proposition 5.24) for ϕ. It follows that CT^{lin} is complete monitorable (Definition 5.16). ◀

[5] This is to guarantee that, within the scope of $\mathsf{max}(Y, \psi)$ or $\mathsf{min}(X, \psi)$, the valuation of y remains constant and so Lemma 5.17 can be applied.

5.4 Monitorabile Fragment

In this subsection, we identify MT^{lin}, a fragment of T^{lin}, and prove its monitorability (Theorem 5.38).

▶ **Definition 5.27** (Monitorable formulae)**.**

- *A formula $\phi \in T^{lin}$ is* violation *(resp.* satisfaction*)* monitorable *iff there exists a reactive monitor M that is sound and violation (resp. satisfaction) complete for ϕ. A fragment $\Gamma \subseteq T^{lin}$ is violation (resp. satisfaction) monitorable iff each $\phi \in \Gamma$ is violation (resp. satisfaction) monitorable.*
- *A formula $\phi \in T^{lin}$ is* monitorable *iff it is violation* or *satisfaction monitorable. A fragment $\Gamma \subseteq T^{lin}$ is monitorable iff each $\phi \in \Gamma$ is monitorable.*

In order to identify a monitorable fragment, let us first proceed with examples.

▶ **Example 5.28.** Consider the following T^{lin} formula:

$$\phi = \mathsf{max}(Y, ([b]\mathsf{ff} \vee \psi) \wedge (\langle\Sigma\rangle Y))$$

with:

$$\psi = x_s \,\underline{\mathtt{in}}\, \mathsf{min}(X, (x_e \,\underline{\mathtt{in}} \langle x_e - x_s \in I\rangle\langle c\rangle\mathsf{tt}) \vee (\langle\Sigma\rangle X))$$

with $I \in \mathcal{I}$ bounded ($\mathcal{RB}(I) \in \mathbb{N}$). Formula ϕ expresses the important bounded response property (in MTL: $G\,(b \Rightarrow F_I\, c)$, Section 1). Notice how every min subformula of ϕ can be rewritten as a constrained min (Section 5.3), whereas the max subformulae are clearly unconstrained. Formula ϕ is violation monitorable: after every occurrence of b, a monitor can check whether a c will follow within I (relative to such occurrence). Since I is bounded, such check always terminates (by Lemma 5.17), and the monitor can say no if no c occurred. However, this formula is not satisfaction monitorable.

▶ Remark 5.29. Bounded response is not *bounded-memory monitorable* in the general case [35] (it is when I is left-closed at 0, see e.g., [17]). Bounded-memory monitorability is out of this paper's scope and is left for future work (Section 7).

▶ **Example 5.30.** Consider now the following T^{lin} formula:

$$\eta = \mathsf{min}(Y, (\langle b\rangle\zeta \vee \langle\Sigma\rangle Y))$$

with:

$$\zeta = x_s \,\underline{\mathtt{in}}\, \mathsf{max}(X, (x_e \,\underline{\mathtt{in}} [x_e - x_s \in I]\langle c\rangle\mathsf{tt}) \wedge (\langle\Sigma\rangle X))$$

with $I \in \mathcal{I}$ bounded. Notice how every max subformula of η can be rewritten as a constrained max (Section 5.3), whereas the min subformulae are clearly not constrained. Formula η is satisfaction monitorable: if a b occurs, a monitor can safely say yes if every action seen within I after such occurrence is a c (this process terminates, again thanks to Lemma 5.17). This formula is however not violation monitorable: any no verdict may be falsified by a future b after which every action within I is a c.

$$\begin{array}{rcllllll} \phi, \psi & ::= & \mathsf{tt} & \mid \mathsf{ff} & \mid \phi \vee \psi & \mid \phi \wedge \psi & \mid x \,\underline{\mathtt{in}}\, \phi \\ & & \mid \langle a \rangle \phi & \mid [a]\phi & \mid \langle g \rangle \phi & \mid [g]\phi & \\ & & \mid \mathsf{min}(X, \phi) & \mid X & & & \end{array}$$

Figure 12 Grammar to generate formulae (closed and guarded terms) of ST^{lin}.

$$\begin{array}{rcllllll} \phi, \psi & ::= & \mathsf{tt} & \mid \mathsf{ff} & \mid \phi \vee \psi & \mid \phi \wedge \psi & \mid x \,\underline{\mathtt{in}}\, \phi \\ & & \mid \langle a \rangle \phi & \mid [a]\phi & \mid \langle g \rangle \phi & \mid [g]\phi & \\ & & \mid \mathsf{max}(Y, \phi) & \mid Y & & & \end{array}$$

Figure 13 Grammar to generate formulae (closed and guarded terms) of VT^{lin}.

Syntactic characterisation. The intuition is that both violation- and satisfaction-monitorable formulae will share the syntax of CT^{lin} and differ for (unconstrained) minimal and maximal fixpoint operators. To simplify the syntax, we prove that the semantics of constrained min and constrained max coincide. That is, for every constrained formula $\mathsf{min}(X, \phi)$, we have the equality $[\![\mathsf{min}(X, \phi)]\!] = [\![\mathsf{max}(X, \phi)]\!]$ (and dually for constrained max formulae). Accordingly, for violation-monitorable formulae, we do not need constrained min in their syntax (Figure 13). This is because constrained min formulae can be rewritten as constrained max formulae, and a constrained max is a special case of max. For example, in the bounded response property (Example 5.28), and after transforming ψ into a constrained min (Section 5.3), the minimal fixpoint operator can be replaced with the maximal one without changing the semantics. Dually, we do not need constrained max in the syntax of satisfaction-monitorable formula (Figure 12).

▶ **Definition 5.31** (Monitorable fragment). *The set of monitorable formulae $MT^{lin} \subset T^{lin}$ is the union of the set of violation-monitorable formulae (Figure 13) and satisfaction-monitorable formulae (Figure 12), i.e., $MT^{lin} = VT^{lin} \cup ST^{lin}$.*

▶ **Proposition 5.32.** *For all $\phi \in MT^{lin}$, $(\!|\phi|\!)$ is sound for ϕ.*

Proof. Since $MT^{lin} \subset T^{lin}$, we conclude by Proposition 5.15. ◀

▶ **Proposition 5.33.** *For all $\phi \in ST^{lin}$, $(\!|\phi|\!)$ is strongly satisfaction complete for ϕ.*

▶ **Theorem 5.34.** *ST^{lin} is satisfaction monitorable.*

Proof. Direct from Proposition 5.32, Proposition 5.33 and Definition 5.27. ◀

▶ **Proposition 5.35.** *For all $\phi \in VT^{lin}$, $(\!|\phi|\!)$ is strongly violation complete for ϕ.*

▶ **Example 5.36.** The formula $\phi_{even(b)} = \mathsf{max}(X, \langle\Sigma\rangle\langle b\rangle X)$ (Example 3.5) is violation monitorable. Its compilation by $(\!|-|\!)$ gives the monitor

$$M = (\!|\phi_{even(b)}|\!) = \mathsf{rec}\ X.(\oplus_{a \in \Sigma}\ a.(b.X + \overline{b}.\mathsf{NO}) + \overline{a}.\mathsf{NO})$$

and it is rather straightforward to see that M is sound and inevitably reaches verdict no (from m_0) when given any infinite timed word outside of the semantics of $\phi_{even(b)}$. Indeed, for any infinite timed word π, we have $\pi \notin [\![\phi_{even(b)}]\!]$ iff there exists a position $i \in \mathbb{N}_{>1}$, $i \bmod 2 = 0$ such that $a_i(\pi) \neq b$. For any such π and the smallest such i, we have $m_0 \overset{\text{TRN}(^i\pi)}{\longrightarrow\!\!\!\!\!\rightarrow}_{inev} \mathsf{no}$.

$\|M \in \mathsf{VYES}\|$	$=$	tt		$\|M \in \mathsf{VNO}\| = \|M \in \mathsf{VEND}\|$	$=$	ff	
$\|\textsc{let}\ x\ \textsc{in}\ M\|$	$=$	$x\ \underline{\mathtt{in}}\ \|M\|$		$\|M + N\|$	$=$	$\|M\| \vee \|N\|$	
$\|M \oplus N\|$	$=$	$\|M\| \vee \|N\|$		$\|M \otimes N\|$	$=$	$\|M\| \wedge \|N\|$	
$\|\alpha.M\|$	$=$	$\langle\alpha\rangle\|M\|$		$\|g.M\|$	$=$	$\langle g\rangle\|M\|$	
$\|X\|$	$=$	X		$\|\mathsf{rec}\ X.M\|$	$=$	$\mathsf{min}(X, \|M\|)$	

Figure 14 Compiler from reactive monitors to ST^{lin}, defined recursively on subterms of monitors.

$\langle\langle M \in \mathsf{VYES}\rangle\rangle = \langle\langle M \in \mathsf{VEND}\rangle\rangle$	$=$	tt		$\langle\langle M \in \mathsf{VNO}\rangle\rangle$	$=$	ff	
$\langle\langle\textsc{let}\ x\ \textsc{in}\ M\rangle\rangle$	$=$	$x\ \underline{\mathtt{in}}\langle\langle M\rangle\rangle$		$\langle\langle M + N\rangle\rangle$	$=$	$\langle\langle M\rangle\rangle \wedge \langle\langle N\rangle\rangle$	
$\langle\langle M \oplus N\rangle\rangle$	$=$	$\langle\langle M\rangle\rangle \vee \langle\langle N\rangle\rangle$		$\langle\langle M \otimes N\rangle\rangle$	$=$	$\langle\langle M\rangle\rangle \wedge \langle\langle N\rangle\rangle$	
$\langle\langle\alpha.M\rangle\rangle$	$=$	$[\alpha]\langle\langle M\rangle\rangle$		$\langle\langle g.M\rangle\rangle$	$=$	$[g]\langle\langle M\rangle\rangle$	
$\langle\langle X\rangle\rangle$	$=$	X		$\langle\langle\mathsf{rec}\ X.M\rangle\rangle$	$=$	$\mathsf{max}(X, \langle\langle M\rangle\rangle)$	

Figure 15 Compiler from reactive monitors to VT^{lin}, defined recursively on subterms of monitors.

▶ **Theorem 5.37.** *VT^{lin} is violation monitorable.*

Proof. Direct from Proposition 5.32, Proposition 5.35 and Definition 5.27. ◀

▶ **Theorem 5.38.** *MT^{lin} is monitorable.*

Proof. Direct from Theorem 5.34, Theorem 5.37, Definition 5.31 and Definition 5.27. ◀

5.5 Maximality

▶ **Definition 5.39** (Largest monitorable fragment)**.** *A fragment $\Gamma \subset T^{lin}$ is said:*

- *The largest satisfaction- (resp. violation-, complete-) monitorable fragment of T^{lin} iff (i) Γ is satisfaction (resp. violation, complete) monitorable and (ii) for every satisfaction- (resp. violation-, complete-) monitorable formula $\psi \in T^{lin}$, there exists $\phi \in \Gamma$ such that $[\![\phi]\!] = [\![\psi]\!]$,*
- *The largest monitorable fragment of T^{lin} iff (i) Γ is monitorable and (ii) for every monitorable formula $\psi \in T^{lin}$, there exists $\phi \in \Gamma$ such that $[\![\phi]\!] = [\![\psi]\!]$.*

In order to prove that MT^{lin} is the largest monitorable fragment of T^{lin}, we follow an approach inspired from [2]. First, we prove that for every reactive monitor M, if M is satisfaction (resp. violation) complete for two formulae in T^{lin}, then such formulae are semantically equivalent (Lemma 5.40). Second, we define two functions: $\|-\|$ (Figure 14), that maps any reactive monitor M to a formula in ST^{lin}, and $\langle\langle-\rangle\rangle$ (Figure 15) that maps any reactive monitor M to a formula in VT^{lin}; then prove that any reactive M is sound and satisfaction complete for its image by $\|-\|$ (i.e., $\|M\|$), and sound and violation complete for $\langle\langle M\rangle\rangle$ (Lemma 5.41). We conclude therefore that any formula $\phi \in T^{lin}$ that is satisfaction

(resp. violation) monitorable, through the existence of a monitor M that is satisfaction (resp. violation) complete for ϕ, is necessarily equivalent to $\|M\| \in ST^{lin}$ (resp. $\langle\langle M \rangle\rangle \in VT^{lin}$, Proposition 5.42). Accordingly, ST^{lin} is the largest satisfaction-monitorable fragment of T^{lin}, VT^{lin} the largest violation-monitorable fragment of T^{lin}, and $MT^{lin} = VT^{lin} \cup ST^{lin}$ the largest monitorable fragment of T^{lin} (Theorem 5.43, Theorem 5.44). We also prove that CT^{lin} is the largest complete-monitorable fragment of T^{lin} (Theorem 5.45).

▶ **Lemma 5.40.** *For every reactive monitor M and formulae $\phi, \psi \in T^{lin}$:*

1. *M is sound and satisfaction complete for ϕ and ψ implies $[\![\phi]\!] = [\![\psi]\!]$,*
2. *M is sound and violation complete for ϕ and ψ implies $[\![\phi]\!] = [\![\psi]\!]$.*

▶ **Lemma 5.41.** *For every reactive monitor M, if M is consistent then:*

1. *M is sound and satisfaction complete for $\|M\|$*
2. *M is sound and violation complete for $\langle\langle M \rangle\rangle$*

▶ **Proposition 5.42.** *For every formula $\phi \in T^{lin}$:*

1. *if ϕ is satisfaction monitorable, then there exists $\psi \in ST^{lin}$ such that $[\![\phi]\!] = [\![\psi]\!]$,*
2. *if ϕ is violation monitorable, then there exists $\psi \in VT^{lin}$ such that $[\![\phi]\!] = [\![\psi]\!]$.*

Proof.

Item 1. Assume $\phi \in T^{lin}$ is satisfaction monitorable. By definition, there exists a reactive monitor M that is sound and satisfaction complete for ϕ. By Lemma 5.7, we have M is consistent. Now, by Lemma 5.41 (Item 1), we deduce M is sound and satisfaction complete for $\|M\| \in ST^{lin}$, and finally we conclude $[\![\phi]\!] = [\![\|M\|]\!]$ by Lemma 5.40 (Item 1).

Item 2. Analogous. ◀

▶ **Theorem 5.43.**

1. *ST^{lin} is the largest satisfaction-monitorable fragment of T^{lin},*
2. *VT^{lin} is the largest violation-monitorable fragment of T^{lin}.*

Proof.

Item 1. Direct from Definition 5.39, Theorem 5.34, and Proposition 5.42 (Item 1).

Item 2. Direct from Definition 5.39, Theorem 5.37, and Proposition 5.42 (Item 2). ◀

▶ **Theorem 5.44.** *MT^{lin} is the largest monitorable fragment of T^{lin}.*

Proof. Direct from Definition 5.39, Theorem 5.43 and Definition 5.31. ◀

▶ **Theorem 5.45.** *CT^{lin} is the largest complete-monitorable fragment of T^{lin}.*

▶ Remark 5.46. An important remark here is the fact that MT^{lin} reflects precisely the effect of bounded intervals on monitorability. At a first glance, it may seem that the existence of a bounded interval in the syntax of a formula ϕ is sufficient for ϕ to be monitorable; however, this is generally false. Take for example the MTL formula $\psi = G\,(c \Rightarrow F_{[2,5]}\,(b \wedge F\,c))$. Although $[2, 5]$ is bounded, ψ is not monitorable. For satisfactions, the reasoning is similar to bounded response. For violations, take any infinite timed word π starting with $(c, 1)(b, 4)(b, 8)$ and satisfying $\forall i \in \mathbb{N}_{>3} : a_i(\pi) = b$, then we have $\pi \notin [\![\psi]\!]$. Yet, there is no way for a monitor to say no based on a finite prefix of π. Without a surprise, the compilation of ψ into T^{lin} falls outside of MT^{lin} as it uses both unconstrained min and max subformulae.

6 Related Work

The state of the art on RV of timed properties mainly centres around the study of LTL extensions such as MTL [27] and MITL, the fragment of MTL excluding singular intervals [5]. The use of a declarative logic to describe system behaviour creates a clear delineation between what can be specified and how this can then be verified algorithmically. This is in line with our approach and is more conducive to the study of monitorability. Baldor *et al.* [9] were the first to propose a transition based approach to monitoring of MITL formulae and establish complexity results for their proposed synthesis. Ho *et al.* [24] split unbounded and bounded parts of a dense-time MTL formula to permit a simpler construction for the bounded parts to obtain space-bounded monitors. Neither of these works consider monitorability issues.

The influential work by Bauer *et al.* [12] introduces the idea of an n-valued semantics for a timed version of LTL called TLTL, defined over *trace prefixes* by incorporating inconclusive outcome, typically denoted by "?". This is used as a definition of the expected monitor behaviour when analysing a trace prefix online, which then guides a method of obtaining executable monitors from timed properties. The main drawback of the approach is that monitor synthesis disregards monitorability aspects; it generates monitors for *any* timed property, without information on whether such property is monitorable or not. The recent work on monitoring MITL by Grosen *et al.* [22] also suffers from this drawback as a result of adopting a three-valued semantics. Despite this, they propose new symbolic method of synthesising monitors for MITL properties via a translation into timed Büchi automata. Pinisetty *et al.* [37] consider the monitoring of properties expressed as timed ω-automata. However, they ascribe an n-valued semantics to prefixes of infinite runs as discussed above. The work focuses on how prior knowledge on the system (also expressed as a timed automaton) can be used for sound predictive monitoring.

All of the above works take into account in one sense or another infinite executions; the infinitary nature of traces poses the main complication when considering the monitorability of linear-time properties [2]. Popular work on synthesising monitors for timed LTL properties with finitary semantics, such as that of Thati and Rosu [45] and of Basin *et al.* [11] sidestep any monitorability issues since every formula becomes monitorable. Basin *et al.* consider a first-order version of MTL called MFOTL to describe properties involving data ranging over an infinite domain. Ulus *et al.* [46] describe a monitor algorithm for Timed Regular Expressions based on the union of two-dimensional zones for finite words. Raszyk *et al.* [41] extend this work to an augmented monitoring setup that uses multiple analysis points (i.e., multi-head monitors) that relates to our parallel monitoring setup. Lima *et al.* [32] augment monitoring verdicts of MFOTL with explanations of why the monitored execution led to violations/satisfactions. Roussanaly *et al.* [43] study a decentralised monitor synthesis procedure for timed regular expressions that is similar to our parallel monitors. Note that a number of works study *quantitative monitoring* of STL [34], a version of MITL interpreted over continuous signals, proposing thereby a robustness measure instead of yes/no verdicts. Such works sidestep monitorability problems either through *approximate* measures under infinitary semantics (see e.g., [16]) or more precise measures under finitary semantics [42].

In summary, our approach is complementary to the above efforts. On the one hand, our work is the first that proposes through monitorability results for an expressive timed logic. As shown in Section 3.2, MTL is less expressive than our T^{lin} which both in general and w.r.t. to monitorability; we have shown that there exists monitorable formulae in T^{lin} that are not expressible in MTL (Section 5, Section 5.4). On the other hand, works such as [24] and [22] focus on efficiency of monitoring algorithms rather than monitorability issues.

Studies on monitorability generally consider one of two definitions; the one due to Pnueli and Zaks [39] or that due to Schneider [44]. Aceto *et al.* [4] have shown that the latter definition is strictly stronger and have systematised the various variants in terms of soundness and completeness criteria similar to those we used. In particular, the definition of monitorability assumed in our work is the stronger variant due to Schneider, in part because it allowed us to establish maximality results for our logical fragments; at the time of writing, the methods for establishing maximal monitorable fragments for Pnueli and Zaks monitorability are not well-established.

7 Conclusion

In this paper, we solved the monitorability problem for an expressive timed logic. We proposed T^{lin}, a modal timed μ-calculus with linear-time semantics and proved that it is strictly more expressive than the well-known and popular MTL. Then, we precisely identified the maximal fragments of violation-monitorable, satisfaction-monitorable, complete-monitorable, and monitorable formulae of T^{lin}. To the best of our knowledge, this is the first work that solves the monitorability problem for such expressive timed logic.

A natural direction of future work is to synthesise monitors as timed automata (TA). Such monitors should be simpler than Mon and can be efficiently implemented in TA-based tools. Another direction is the study of bounded-memory monitorability. At least two research axes in this direction are worth investigating. First, the identification of the fragments of bounded-memory monitorable formulae of T^{lin} under the same setting as in this paper (linear time, time-divergent infinite words). Second, analogous to the first axis, the characterisation of the weakest conditions on infinite timed words to enable bounded-memory monitorability for some properties that are not monitorable in bounded memory in the general case.

References

1 L. Aceto, A. Ingólfsdóttir, K. G. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007.

2 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. Adventures in monitorability: from branching to linear time and back again. *Proc. ACM Program. Lang.*, 3(POPL):52:1–52:29, 2019. doi:10.1145/3290365.

3 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. An operational guide to monitorability. In *International Conference on Software Engineering and Formal Methods (SEFM)*, pages 433–453. Springer, 2019. doi:10.1007/978-3-030-30446-1_23.

4 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. An operational guide to monitorability with applications to regular properties. *Software and Systems Modeling*, 20:335–361, 2021. doi:10.1007/s10270-020-00860-z.

5 Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43(1):116–146, 1996. doi:10.1145/227595.227602.

6 Rajeev Alur and Thomas A Henzinger. Real-time logics: complexity and expressiveness. *Information and Computation*, 104(1):35–77, 1993. doi:10.1006/inco.1993.1025.

7 Rajeev Alur and Thomas A Henzinger. A really temporal logic. *Journal of the ACM (JACM)*, 41(1):181–203, 1994. doi:10.1145/174644.174651.

8 Rajeev Alur and Thomas A Henzinger. Modularity for timed and hybrid systems. In *International Conference on Concurrency Theory (CONCUR)*, pages 74–88. Springer, 1997. doi:10.1007/3-540-63141-0_6.

9 Kevin Baldor and Jianwei Niu. Monitoring dense-time, continuous-semantics, metric temporal logic. In *International Conference on Runtime Verification (RV)*, pages 245–259. Springer, 2012. doi:10.1007/978-3-642-35632-2_24.

10 Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. *Lectures on Runtime Verification: Introductory and Advanced Topics*, pages 1–33, 2018. doi:10.1007/978-3-319-75632-5_1.

11 David A. Basin, Felix Klaedtke, Samuel Müller, and Eugen Zalinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2):15:1–15:45, 2015. doi:10.1145/2699444.

12 Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, 2011. doi:10.1145/2000799.2000800.

13 Edmund Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another look at LTL model checking. In *International Conference on Computer Aided Verification (CAV)*, pages 415–427. Springer, 1994. doi:10.1007/3-540-58179-0_72.

14 Rance Cleaveland, Jeroen JA Keiren, and Peter Fontana. An expressive timed modal mu-calculus for timed automata. In *International Conference on Quantitative Evaluation of Systems and Formal Modeling and Analysis of Timed Systems (QEST+FORMATS)*, pages 160–178. Springer, 2024. doi:10.1007/978-3-031-68416-6_10.

15 Christian Dax, Martin Hofmann, and Martin Lange. A proof system for the linear time μ-calculus. In S. Arun-Kumar and Naveen Garg, editors, *International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 273–284. Springer, 2006. doi:10.1007/11944836_26.

16 Alexandre Donzé and Oded Maler. Robust satisfaction of temporal logic over real-valued signals. In *International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, pages 92–106. Springer, 2010. doi:10.1007/978-3-642-15297-9_9.

17 Mohammed Foughali, Saddek Bensalem, Jacques Combaz, and Félix Ingrand. Runtime verification of timed properties in autonomous robots. In *International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 1–12. IEEE, 2020. doi:10.1109/MEMOCODE51338.2020.9315156.

18 Mohammed Foughali, Pierre-Emmanuel Hladik, and Alexander Zuepke. Compositional verification of embedded real-time systems. *J. Syst. Archit.*, 142:102928, 2023. doi:10.1016/j.sysarc.2023.102928.

19 A. Francalanza, L. Aceto, and A. Ingólfsdóttir. Monitorability for the Hennessy-Milner logic with recursion. *Formal Methods Syst. Des.*, 51(1), 2017. doi:10.1007/s10703-017-0273-z.

20 Adrian Francalanza. A theory of monitors. *Inf. Comput.*, 281:104704, 2021. doi:10.1016/j.ic.2021.104704.

21 Adrian Francalanza, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. A foundation for runtime monitoring. In *RV*, volume 10548 of *Lecture Notes in Computer Science*, pages 8–29. Springer, 2017. doi:10.1007/978-3-319-67531-2_2.

22 Thomas Møller Grosen, Sean Kauffman, Kim Guldstrand Larsen, and Martin Zimmermann. Monitoring timed properties (revisited). In *International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, pages 43–62. Springer, 2022. doi:10.1007/978-3-031-15839-1_3.

23 Thomas A Henzinger. Sooner is safer than later. *Information Processing Letters*, 43(3):135–141, 1992. doi:10.1016/0020-0190(92)90005-G.

24 Hsi-Ming Ho, Joël Ouaknine, and James Worrell. Online monitoring of metric temporal logic. In *International Conference on Runtime Verification (RV)*, pages 178–192. Springer, 2014. doi:10.1007/978-3-319-11164-3_15.

25 Ullrich Hustadt, Ana Ozaki, and Clare Dixon. Theorem proving for pointwise metric temporal logic over the naturals via translations. *Journal of Automated Reasoning*, 64(8):1553–1610, 2020. doi:10.1007/s10817-020-09541-4.

26 Donald Ervin Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2:127–145, 1968. doi:10.1007/BF01692511.

27 Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990. doi:10.1007/BF01995674.

28 Dexter Kozen. Results on the propositional μ-calculus. *Theoretical computer science*, 27(3):333–354, 1983. doi:10.1016/0304-3975(82)90125-6.

29 Kim G Larsen. Proof systems for satisfiability in Hennessy-Milner logic with recursion. *Theoretical Computer Science*, 72(2-3):265–288, 1990. doi:10.1016/0304-3975(90)90038-J.

30 Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The journal of logic and algebraic programming*, 78(5):293–303, 2009. doi:10.1016/j.jlap.2008.08.004.

31 Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Symposium on Principles of Programming Languages (POPL)*, pages 97–107, 1985. doi:10.1145/318593.318622.

32 Leonardo Lima, Andrei Herasimau, Martin Raszyk, Dmitriy Traytel, and Simon Yuan. Explainable online monitoring of metric temporal logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 473–491. Springer, 2023. doi:10.1007/978-3-031-30820-8_28.

33 Frank D Macías-Escrivá, Rodolfo Haber, Raul Del Toro, and Vicente Hernandez. Self-adaptive systems: A survey of current approaches, research challenges and applications. *Expert Systems with Applications*, 40(18):7267–7279, 2013. doi:10.1016/j.eswa.2013.07.033.

34 Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, volume 3253, pages 152–166. Springer, 2004. doi:10.1007/978-3-540-30206-3_12.

35 Oded Maler, Dejan Nickovic, and Amir Pnueli. On synthesizing controllers from bounded-response properties. In *International Conference on Computer Aided Verification (CAV)*, pages 95–107. Springer, 2007. doi:10.1007/978-3-540-73368-3_12.

36 Joël Ouaknine and James Worrell. On metric temporal logic and faulty turing machines. In *International Conference on Foundations of Software Science and Computation Structures*, pages 217–230. Springer, 2006. doi:10.1007/11690634_15.

37 Srinivas Pinisetty, Thierry Jéron, Stavros Tripakis, Yliès Falcone, Hervé Marchand, and Viorel Preoteasa. Predictive runtime verification of timed properties. *Journal of Systems and Software*, 132:353–365, 2017. doi:10.1016/j.jss.2017.06.060.

38 Amir Pnueli. The temporal logic of programs. In *Annual Symposium on Foundations of Computer Science (SFCS)*, pages 46–57. ieee, 1977. doi:10.1109/SFCS.1977.32.

39 Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. In *International Symposium on Formal Methods (FM)*, pages 573–586. Springer, 2006. doi:10.1007/11813040_38.

40 Vasumathi Raman, Mehdi Maasoumy, and Alexandre Donzé. Model predictive control from signal temporal logic specifications: A case study. In *International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems*, pages 52–55, 2014. doi:10.1145/2593458.2593472.

41 Martin Raszyk, David Basin, Srđan Krstić, and Dmitriy Traytel. Multi-head monitoring of metric temporal logic. In *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 151–170. Springer, 2019. doi:10.1007/978-3-030-31784-3_9.

42 Neha Rino, Mohammed Foughali, and Eugene Asarin. Efficiently computable distance-based robustness for a practical fragment of STL. In *International Conference on Quantitative Evaluation of Systems and Formal Modeling and Analysis of Timed Systems (QEST+FORMATS)*, pages 179–195. Springer, 2024. doi:10.1007/978-3-031-68416-6_11.

43 Victor Roussanaly and Yliès Falcone. Decentralised runtime verification of timed regular expressions. In *International Symposium on Temporal Representation and Reasoning (TIME)*, pages 6:1–6:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.TIME.2022.6.

44 Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000. doi:10.1145/353323.353382.

45 Prasanna Thati and Grigore Roşu. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science*, 113:145–162, 2005. doi:10.1016/j.entcs.2004.01.029.

46 Dogan Ulus, Thomas Ferrère, Eugene Asarin, and Oded Maler. Online timed pattern matching using derivatives. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 736–751. Springer, 2016. doi:10.1007/978-3-662-49674-9_47.

47 Moshe Y Vardi. An automata-theoretic approach to linear temporal logic. *Logics for concurrency: structure versus automata*, pages 238–266, 2005.

48 Pierre Wolper. Temporal logic can be more expressive. In *Annual Symposium on Foundations of Computer Science (SFCS)*, pages 340–348. IEEE Computer Society, 1981. doi:10.1109/SFCS.1981.44.

49 W. Yi. Real-time behaviour of asynchronous agents. In *International Conference on Concurrency Theory (CONCUR)*, 1990. doi:10.1007/BFb0039080.

The Algebra of Patterns

David Binder
University of Kent, Canterbury, UK

Lean Ermantraut
Radboud University, Nijmegen, The Netherlands

Abstract

Pattern matching is a popular feature in functional, imperative and object-oriented programming languages. Language designers should therefore invest effort in a good design for pattern matching. Most languages choose a *first-match semantics* for pattern matching; that is, clauses are tried in the order in which they appear in the program until the first one matches. As a consequence, the order in which the clauses appear cannot be arbitrarily changed, which results in a less declarative programming model. The declarative alternative to this is an *order-independent semantics* for pattern matching, which is not implemented in most programming languages since it requires more verbose patterns. The reason for this verbosity is that the syntax of patterns is usually not expressive enough to express the *complement* of a pattern. In this paper, we show a principled way to make order-independent pattern matching practical. Our solution consists of two parts: First, we introduce a *boolean algebra of patterns* which can express the complement of a pattern. Second, we introduce *default clauses* to pattern matches. These default clauses capture the essential idea of a fallthrough case without sacrificing the property of order-independence.

2012 ACM Subject Classification Theory of computation → Lambda calculus; Theory of computation → Type theory

Keywords and phrases functional programming, pattern matching, algebraic data types, equational reasoning

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.2

Related Version *Extended Version*: `https://arxiv.org/abs/2504.18920` [2]

Supplementary Material *Software (ECOOP 2025 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.11.2.10`

Funding *David Binder*: supported by ARIA programme for Safeguarded AI (grant MSAI-PR01-P11).

1 Introduction

Pattern matching is a programming language feature that combines two operations: Testing whether a value matches a given pattern and, if the test is successful, extracting subexpressions of the value and binding them to variables. A pattern matching expression consists of the *scrutinee* which is analyzed by the pattern match and a list of *clauses*. Each clause consists of a *pattern*, hence the name, and an expression called the *right-hand side*. These elements can all be seen in the following example of the length function, which computes the length of a list:

$$\text{length}(xs) \coloneqq \textbf{case}\ \underbrace{xs}_{\text{scrutinee}}\ \textbf{of}\ \{\ \underbrace{\texttt{Nil} \Rightarrow 0}_{\text{clause}},\ \underbrace{\texttt{Cons}(_, zs)}_{\text{pattern}} \Rightarrow \underbrace{1 + \text{length}(zs)}_{\text{right-hand side}}\ \}$$

Programming with pattern matching is popular. For a long time, only programmers who use functional languages could enjoy this feature, but today pattern matching has arrived in the mainstream of object-oriented and imperative programming languages. For example, Python, Java, Dart, and C# added pattern matching as an extension to the language, and some newer languages such as Rust and Swift, which are not primarily functional languages, supported pattern matching from the beginning.

© David Binder and Lean Ermantraut;
licensed under Creative Commons License CC-BY 4.0

Pattern matching has become one of the central constructs that is used to structure the control and data flow of a program. How pattern matches are checked and evaluated therefore fundamentally determines how we reason about the meaning of our programs; both informally when we read and write code in our daily lives, and formally in proof assistants, static analyzers and other verification tools. It is therefore important to choose a semantics for pattern matching which makes reasoning as simple as possible. One choice that language designers have to make is whether the clauses are tried in the order in which they appear in the program, or whether the order of the clauses does not matter. The following definition captures this difference:

▶ **Definition 1** (First-Match and Order-Independent Semantics). *A language uses* first-match semantics *if the clauses of a pattern match are tried in the order in which they occur until the first pattern matches the scrutinee. A language uses* order-independent semantics *if any of the matching clauses in a pattern match can be used to non-deterministically reduce a pattern matching expression.*

Overlapping clauses, i.e. two clauses with patterns that can match the same scrutinee, are a problem for order-independent semantics since a pattern-matching expression with overlapping clauses can non-deterministically reduce to two different results. The obvious solution is to enforce that two clauses must never overlap. But if we adopt this solution, then programmers usually have to write much more verbose patterns. For example, instead of the following function which checks whether the argument is the color `Red`

$$\text{isRed}(c) \coloneqq \textbf{case}\ c\ \textbf{of}\ \{\ \texttt{Red} \Rightarrow \texttt{True},\ _ \Rightarrow \texttt{False}\ \},$$

the programmer has to write the following more verbose expression

$$\text{isRed}(c) \coloneqq \textbf{case}\ c\ \textbf{of}\ \{\ \texttt{Red} \Rightarrow \texttt{True},\ \texttt{Green} \Rightarrow \texttt{False},\ \texttt{Blue} \Rightarrow \texttt{False}\ \}.$$

The source of this verbosity is the poor expressiveness of patterns, which usually do not allow to express the *complement* of a pattern without exhaustively enumerating all the alternatives. In this example, we had to explicitly mention both `Blue` and `Green` to express the complement of the pattern `Red`. In languages that support or-patterns we can combine the last two clauses in the single clause $(\texttt{Green} \parallel \texttt{Blue}) \Rightarrow \texttt{False}$, but we still have to mention all constructors explicitly. It is unreasonable to ask programmers to always list all constructors of a data type, and they would rightfully reject a programming language which enforces this. Every proposal to use order-independent pattern matching in a realistic programming language therefore has to address this verbosity problem.

We propose two complementary solutions to the verbosity problem. In our system, the function can be written in the following ways:

$$\text{isRed}(c) \coloneqq \textbf{case}\ c\ \textbf{of}\ \{\ \texttt{Red} \Rightarrow \texttt{True},\ \neg\texttt{Red} \Rightarrow \texttt{False}\ \} \qquad \text{(Negation Pattern)}$$

$$\text{isRed}(c) \coloneqq \textbf{case}\ c\ \textbf{of}\ \{\ \texttt{Red} \Rightarrow \texttt{True},\ \textbf{default} \Rightarrow \texttt{False}\ \} \qquad \text{(Default Clause)}$$

To write the first variant we enrich the syntax of patterns with *negation patterns* which match whenever the negated pattern does not match. The two patterns are no longer overlapping, and at the same time, the pattern matching expression is not more verbose than our first version. Such negation patterns were already proposed under the name of "anti-patterns" [16, 17, 6] and are also mentioned by some other authors [19, 11]; the core contribution of this paper, however, is to consider them in the context of a complete *boolean algebra of patterns* which allows us to apply algebraic reasoning to patterns.

To write the second variant we enrich the syntax of pattern-matching expressions with *default clauses*[1]. A pattern matching expression can contain at most one such default clause (where **default** is a keyword instead of a pattern) which matches whenever none of the normal clauses matches the scrutinee. At first glance, the meaning of a default clause looks identical to the meaning of a clause with a wildcard pattern, but this is only the case for first-match semantics. If we use order-independent semantics, then we see that the meaning of a default clause does not correspond to a clause with a wildcard pattern, but to a clause that is headed by the negation of the disjunction of all other patterns in the pattern matching expression. In the compilation algorithm, however, default clauses are treated natively, so that we don't have to expand a default clause to a large pattern in the case of pattern matches with many clauses.

We are particularly interested in the kind of reasoning which is preserved when we add or remove clauses from a pattern match, or when we add or remove constructors from a data type. We will argue that this is a significant software engineering challenge that most pattern-based languages do not adequately address yet.

1.1 Algebraic Reasoning and Variable Binding

The main contribution of this work is to support a rich catalog of equivalences for patterns. In fact, we strive to treat patterns as a boolean algebra which consists of negation patterns, and-patterns, or-patterns, wildcard-patterns and absurd-patterns, featuring all the expected equivalences. This is challenging from the perspective of variable binding. For instance, we want the law of double negation, i.e. $[\![\neg\neg p]\!] \equiv [\![p]\!]$, to hold, but this entails that we cannot ignore variables that occur inside patterns in a negated context. We will demonstrate that we can achieve the desired equivalences by a variable binding definition that tracks whether variables occur under an even or odd number of negations. A related problem is to have well-defined variable binding in the context of or-patterns $p \parallel p'$, which requires a careful definition of linearity to make sure that the same variables are bound in all cases of an or-pattern. We also describe the conditions under which we can replace a pattern $p_1 \parallel p_2$ by the pattern $p_2 \parallel p_1$; this transformation is not semantic preserving in most systems which implement or-patterns.

1.2 Overview and Contributions

The rest of this article is structured as follows:

- In Section 2 we motivate why the first-match semantics that most programming languages use for pattern matching fails to address several important reasoning and software engineering problems. We show how our contributions can improve upon this status quo.
- In Section 3 we present our first main contribution: the algebra of patterns. We describe the syntax of patterns, the rules for non-deterministically matching a pattern against a value, and the restrictions that characterize the subset of linear and deterministic patterns. We also prove the algebraic properties that characterize the semantic equivalence relation on patterns.

[1] Some programming languages support default clauses with similar semantics to what we propose. Ada, for example, enforces that clauses in a case statement are not overlapping, but allows an optional "other" clause with the same semantics as our default clause. The clauses that Ada allows are however much more restricted, since they only work for enums instead of algebraic data types.

- Section 4 introduces our second contribution, a small term language that contains pattern-matching expressions with default clauses. This section also introduces exemplary typing rules to the untyped patterns of Section 3 and the constructs of our term language; the compilation of patterns described in later sections, however, does not depend on patterns being typed.
- Not all extensions to pattern matching can be compiled to efficient code, but ours can. In Section 5 we show how to compile patterns by translating them to a variant of a disjunctive normal form. We use these normalized patterns in the compilation of pattern-matching expressions to decision trees, which we describe in Section 6.
- We discuss future work in Section 7, related work in Section 8 and conclude in Section 9.

The results in sections Section 3 have been formally verified in the proof assistant Rocq, and are made available as complementary material. Theorems and definitions which have been formalized are marked with the symbol .

2 Motivation

Most programming languages that implement pattern matching use a first-match semantics for which the order of clauses matters. Changing this state of affairs is our main motivation for introducing a more expressive set of algebraic patterns. Let us therefore explain why we think that first-match semantics is not a good default choice for declarative languages.

2.1 First-Match Semantics Weakens Equational Reasoning

Equational reasoning is one of the most important methods functional programmers use to reason about code. Equational reasoning allows us to show that two expressions are equivalent if we can obtain one from the other by a series of rewriting steps. Each rewriting step is justified by some equation between terms, and functions defined by pattern matching are one of the main sources of such equations. Take, for example, the definitions of the functions “id” and “map”:

$$\mathrm{id}(x) \coloneqq x \tag{1}$$
$$\mathrm{map}(f, []) \coloneqq [] \tag{2}$$
$$\mathrm{map}(f, x :: xs) \coloneqq f(x) :: \mathrm{map}(f, xs) \tag{3}$$

We can show that mapping the identity function over a list returns the original list, i.e. that $\mathrm{map}(\mathrm{id}, xs) = xs$ holds, by using the individual clauses of the definition of “map”:

$$\begin{aligned} \mathrm{map}(\mathrm{id}, []) &=_{(2)} [] \\ \mathrm{map}(\mathrm{id}, x :: xs) =_{(3)} \mathrm{id}(x) :: \mathrm{map}(\mathrm{id}, xs) &=_{(1)} x :: \mathrm{map}(\mathrm{id}, xs) =_{(IH)} x :: xs \end{aligned}$$

We have annotated each rewriting step with the equation that justifies it, and in the last step, we have used the induction hypothesis (IH). In this example, we have used the fact that every clause of the pattern match is a valid equation that we can use for rewriting. But this was only valid because the equations in the definition of “map” do not overlap. To see why this is essential, consider the example from the introduction again:

$$\mathrm{isRed}(\texttt{Red}) \coloneqq \texttt{True} \tag{4}$$
$$\mathrm{isRed}(_) \coloneqq \texttt{False} \tag{5}$$

If we are using these clauses as equations we can now show that True is equal to False, using the proof $\texttt{True} =_{(4)} \text{isRed}(\texttt{Red}) =_{(5)} \texttt{False}$. A human is of course very unlikely to make such a silly mistake, but if we cannot guarantee that every clause holds as a valid equality between expressions, then we also limit what automatic tools can do for us.

2.2 First-Match Semantics Complicates Reasoning About Change

Software engineering has been described as the problem of integrating programming over time [27]. This means that we not only have to understand the meaning of a program at a fixed point in time, but we also have to understand how the meaning of a program changes as it is developed and maintained. It should therefore be as simple as possible to reason about how the meaning of a program changes when a programmer adds a clause to a pattern match, removes a clause from a pattern match, adds a constructor to a data type, or removes a constructor from a data type. Let us see why first-match semantics complicates reasoning about these kinds of changes.

If we want to understand the consequences of adding or removing a clause from a pattern match, then we have to consider all clauses occurring both above and below the clause we are changing. Having to reason about the entire context of the clause makes it much harder to spot bugs in code reviews, since it is not uncommon that pattern matching expressions can span multiple pages of code. Order-independent semantics guarantees that it doesn't matter if we add, delete or modify a clause at the beginning, middle or end of a list of clauses.

The other kind of change that frequently occurs is that we add or remove a constructor from an existing data type. Whether we have to adjust existing pattern matches depends on how those have been written. Even if we enforce that clauses in a pattern matching expression must not overlap, we have multiple possibilities to write exhaustive pattern matches. Consider a data type "Group" which consists of administrators, registered users and guests. We want to guarantee that only administrators have write access, but we have two possibilities to write the function:

$\text{hasWriteAccess} : \text{Group} \to \text{Bool}$	$\text{hasWriteAccess} : \text{Group} \to \text{Bool}$
$\text{hasWriteAccess}(\texttt{Admin}) \coloneqq \texttt{True}$	$\text{hasWriteAccess}(\texttt{Admin}) \coloneqq \texttt{True}$
$\text{hasWriteAccess}(\texttt{RegisteredUser} \parallel \texttt{Guest}) \coloneqq \texttt{False}$	$\text{hasWriteAccess}(\neg\texttt{Admin}) \coloneqq \texttt{False}$

On the left side we have pattern matched exhaustively on the complement of `Admin`, whereas on the right side we have used a negation pattern. These two programs behave the same, but we can observe a difference when we add moderators as a fourth type of user. We have to revisit the function on the left, whereas the function on the right continues to compile.

Some programming languages have already started to add support for enforcing these kind of considerations. For example, Rust supports the `#[non_exhaustive]` attribute on type declarations[2]. This attribute restricts pattern matches and ensures that they continue to compile after a new constructor has been added to the type; annotating the Group type with such an attribute would disallow the definition on the left. Dually, OCaml warns against writing "fragile pattern matches". A fragile pattern can hide newly added constructors, which might have unintended consequences. The algebraic patterns presented in this paper provide the necessary tools to program in situations where the programmer wants to enforce either one of these dual restrictions. There is also some empirical evidence that programmers feel uneasy about using wildcard patterns in evolving codebases. For example, the authors

[2] Cp. doc.rust-lang.org/stable/reference/attributes/type_system.html.

of a recent study about the programming practices of functional programmers [21] write: "Participants felt similarly about the use of wildcards in pattern matching, which silently assign pre-existing behavior to new variants of an enumeration. P7 mentioned that, in their main codebase, wildcards are completely disallowed for this reason, even though they make programming more convenient."

2.3 Compositional Pattern Matching and Extensible Data Types

Our main motivation in this paper is to obtain a more declarative programming model for standard nominal data types and ordinary pattern matching. The compilation algorithm that we are presenting, however, also works for polymorphic variants [10]. Polymorphic variants are used in the programming language OCaml, for example, to handle open sets of possible error conditions without having to declare a data type which lists all possible errors. Our algorithm does not presuppose a data type declaration with a fixed set of constructors which is necessary to check for exhaustiveness; we do not discuss exhaustiveness of patterns in the main part of this paper, but outline the necessary modifications to check for it in the extended version [2]. We think that the ideas and the algebraic patterns that we present in this paper are also useful for some proposed solutions of the expression problem. In the compositional programming approach [25, 29, 28], for example, we can create pattern matches whose clauses are distributed among multiple different modules. In that scenario it is especially important to guarantee that the contributing clauses which come from different modules do not overlap in their left-hand side.

3 The Algebra of Patterns

As our first contribution, we introduce an *algebra of patterns*. The syntax of patterns and values is specified in Figure 1. We assume that we have an infinite set CtorNames of constructor names such as `True`, `False` or `Nil`, and a set Variables which contains variables such as x, y or z. Each constructor $\mathcal{C}$ comes with an implicit *arity* n, and we sometimes write $\mathcal{C}^n$ to make this explicit. For example, the constructor `Cons` has arity 2 since it needs to be applied to two arguments. Patterns p consist of variable patterns x, constructor patterns $\mathcal{C}^n(p_1, \ldots, p_n)$, and-patterns $p \,\&\, p$, or-patterns $p \parallel p$, wildcard-patterns _, absurd-patterns #, and negation patterns $\neg p$. Values consist of constructors applied to other values; the two-element list `Cons`(`True`, `Cons`(`False`, `Nil`)) is an example of a value. If a pattern p matches a value v we obtain a substitution σ. Such substitutions are formalized as lists of mappings $x \mapsto v$, and we write $\sigma_1 +\!\!+ \sigma_2$ for the concatenation of two such lists. Not every list of mappings is a proper map since some lists contain multiple mappings for the same variable, but we will later prove that patterns from the subset of linear patterns produce substitutions that map a variable to at most one value. There are two functions that compute the free variables contained in a pattern. The function $\mathrm{FV}^e(p)$ computes the free variables that occur under an *even* number of negations, whereas $\mathrm{FV}^o(p)$ computes the variables under an *odd* number of negations.

Variable, constructor and wildcard patterns are standard, so we do not introduce them explicitly and turn directly to the more interesting absurd-patterns, or-patterns, negation-patterns, and and-patterns.

The absurd-pattern # is the dual of the wildcard-pattern _. Whereas the wildcard-pattern matches any value, the absurd-pattern never matches a value. We include the absurd-pattern in the syntax of patterns since we need a canonical bottom element of the boolean algebra. But there are also more practical reasons to include an absurd pattern. In

$$
\begin{array}{lcll}
 & & \mathcal{C}, \mathcal{C}^n \in \textsc{CtorNames} \quad x, y, z \in \textsc{Variables} & \\
p & ::= & x \mid \mathcal{C}^n(p_1, \ldots, p_n) \mid p \,\&\, p \mid p \parallel p \mid _ \mid \# \mid \neg p & \textit{Patterns} \\
v & ::= & \mathcal{C}^n(v_1, \ldots, v_n) & \textit{Values} \\
m & ::= & x \mapsto v & \textit{Mapping} \\
\sigma & ::= & [] \mid m :: \sigma & \textit{Substitution}
\end{array}
$$

$$
\begin{array}{rlrl}
\mathrm{FV}^e(x) & := \{x\} & \mathrm{FV}^o(x) & := \emptyset \\
\mathrm{FV}^e(_) & := \emptyset & \mathrm{FV}^o(_) & := \emptyset \\
\mathrm{FV}^e(\#) & := \emptyset & \mathrm{FV}^o(\#) & := \emptyset \\
\mathrm{FV}^e(\neg p) & := \mathrm{FV}^o(p) & \mathrm{FV}^o(\neg p) & := \mathrm{FV}^e(p) \\
\mathrm{FV}^e(p_1 \parallel p_2) & := \mathrm{FV}^e(p_1) \cup \mathrm{FV}^e(p_2) & \mathrm{FV}^o(p_1 \parallel p_2) & := \mathrm{FV}^o(p_1) \cup \mathrm{FV}^o(p_2) \\
\mathrm{FV}^e(p_1 \,\&\, p_2) & := \mathrm{FV}^e(p_1) \cup \mathrm{FV}^e(p_2) & \mathrm{FV}^o(p_1 \,\&\, p_2) & := \mathrm{FV}^o(p_1) \cup \mathrm{FV}^o(p_2) \\
\mathrm{FV}^e(\mathcal{C}(p_1, \ldots, p_n)) & := \cup_{i=1}^n \mathrm{FV}^e(p_i) & \mathrm{FV}^o(\mathcal{C}(p_1, \ldots, p_n)) & := \cup_{i=1}^n \mathrm{FV}^o(p_i)
\end{array}
$$

Figure 1 The syntax of values, patterns and substitutions. We track whether variables occur under an even or odd number of negations.

the lazy programming language Haskell, for example, we sometimes have to write a pattern that never matches but forces the evaluation of an argument. This is often achieved by adding a guard clause that evaluates to false, but we can also use an absurd pattern for the same purpose.

An or-pattern $p_1 \parallel p_2$ matches a value if either p_1 or p_2 (or both) matches the value. Or-patterns introduce a complication for the compiler: we have to ensure that if values are bound to variables in p_1 during matching, then p_2 has to bind values of the same type to the same variables. The following example on the left shows how or-patterns without variables can be used to combine multiple clauses into one.

$$
\begin{array}{ll}
\textbf{data}\ \mathrm{Day} \ = \texttt{Mo} \mid \texttt{Tu} \mid \texttt{We} \mid \texttt{Th} \mid \texttt{Fr} \mid \texttt{Sa} \mid \texttt{Su} & \\
\mathrm{isWeekend} : \mathrm{Day} \to \mathrm{Bool} & \mathrm{isWeekend} : \mathrm{Day} \to \mathrm{Bool} \\
\mathrm{isWeekend}(\texttt{Sa} \parallel \texttt{Su}) := \texttt{True} & \mathrm{isWeekend}(\texttt{Sa} \parallel \texttt{Su}) := \texttt{True} \\
\mathrm{isWeekend}(\texttt{Mo} \parallel \texttt{Tu} \parallel \texttt{We} \parallel \texttt{Th} \parallel \texttt{Fr}) := \texttt{False} & \mathrm{isWeekend}(\neg(\texttt{Sa} \parallel \texttt{Su})) := \texttt{False}
\end{array}
$$

We could also have written the second clause of the previous example using a wildcard pattern, but then the two patterns would overlap. We can get rid of the verbose second clause by using a negation pattern, which is illustrated on the right. A negation pattern matches if, and only if, the negated pattern does not match.

The last type of patterns we introduce are and-patterns $p_1 \,\&\, p_2$ which succeed if both p_1 and p_2 match the scrutinee. While or-patterns $p_1 \parallel p_2$ require that p_1 and p_2 bind the same set of variables, and-patterns $p_1 \,\&\, p_2$ require p_1 and p_2 to bind disjoint sets of variables. Many languages already have a special case of and-patterns: An @-pattern $x@p$ (some languages also use the keyword "as") is an and-pattern where the left side must be a variable. If we want to rewrite the previous example to return a string instead of a boolean value, then we could use the following and-patterns:

$$
\begin{array}{l}
\mathrm{isWeekend} : \mathrm{Day} \to \mathrm{String} \\
\mathrm{isWeekend}(x \,\&\, (\texttt{Sa} \parallel \texttt{Su})) := \mathrm{show}(x) \mathbin{+\!\!+} \text{“ is on the weekend”} \\
\mathrm{isWeekend}(x \,\&\, \neg(\texttt{Sa} \parallel \texttt{Su})) := \mathrm{show}(x) \mathbin{+\!\!+} \text{“ is not on the weekend”}
\end{array}
$$

Here we use the show function to print a weekday and ++ for string concatenation. We will use this snippet as a running example in Section 5 and Section 6 to show how patterns are normalized and compiled to efficient code.

3.1 Matching Semantics of Patterns

We will now present the formal rules which specify what happens when we match a pattern against a value; these rules can be found in the upper half of Figure 2. We use two mutually recursive judgments to formalize the semantics of matching: The judgment $p \triangleright v \rightsquigarrow \sigma$ expresses that the pattern p successfully matches the value v and also binds values to variables of the pattern; those mappings are recorded in the substitution σ. The judgment $p \not\triangleright v \rightsquigarrow \sigma$ expresses that the pattern p does *not* match the value v, and also produces a substitution. That the judgment $p \not\triangleright v \rightsquigarrow \sigma$ also produces a substitution might be surprising: How can there be a substitution if the value does not match the pattern? We can think of these as "temporarily deactivated" substitutions which will become available again if we enclose the pattern in an additional negation. In fact, such substitutions are necessary if we want to guarantee that the pattern p is semantically equivalent to $\neg\neg p$ in the case where p contains variables, for example in Equation (5) below.

▶ **Example 2.** These examples show some well-behaved instances of the matching judgments:

$$\mathrm{Cons}(x, xs) \triangleright \mathrm{Cons}(2, \mathrm{Cons}(3, \mathrm{Nil})) \rightsquigarrow [x \mapsto 2, xs \mapsto \mathrm{Cons}(3, \mathrm{Nil})] \tag{1}$$

$$\mathrm{True} \parallel \mathrm{False} \triangleright \mathrm{True} \rightsquigarrow [] \tag{2}$$

$$\mathrm{True} \not\triangleright \mathrm{False} \rightsquigarrow [] \tag{3}$$

$$\neg x \not\triangleright \mathrm{True} \rightsquigarrow [x \mapsto \mathrm{True}] \tag{4}$$

$$\neg\neg x \triangleright \mathrm{True} \rightsquigarrow [x \mapsto \mathrm{True}] \tag{5}$$

Equation (1) illustrates how the rules Var and Ctor can be used to match a constructor pattern against a list while binding components of the list to variables x and xs. Equation (2) shows how the rule Or_1 (together with the rule Ctor) allows to successfully match the left side of an or-pattern against a value. Equation (3) illustrates the rule Ctor_2 and Equation (4) the combination of rules Var and Neg_2. Equation (5) shows why we need two mutually recursive judgments: in order for the pattern $\neg\neg x$ to match against the value True we need to use the rules Neg_1, Neg_2 and Var to ensure that the variable x is bound to the value True.

Using the rules for $p \triangleright v \rightsquigarrow \sigma$ and $p \not\triangleright v \rightsquigarrow \sigma$ we can already prove that they are, in a certain sense, sound and complete:

▶ **Theorem 3** (Matching rules are sound, Agda). *The rules from Figure 2 are sound. There is no pattern p, value v and substitutions σ_1, σ_2 such that both $p \triangleright v \rightsquigarrow \sigma_1$ and $p \not\triangleright v \rightsquigarrow \sigma_2$ hold.*

▶ **Theorem 4** (Matching rules are complete, Agda). *The rules from Figure 2 are complete, i.e. for any pattern p and value v there is some σ such that $p \triangleright v \rightsquigarrow \sigma$ or $p \not\triangleright v \rightsquigarrow \sigma$ holds.*

Since we are interested in algebraic reasoning, we have to say when we consider two patterns to be semantically equivalent. We will define semantic equivalence in Definition 6 using the two judgments introduced above. But first, we have to define when two substitutions (which we formalized as lists of mappings) are semantically equivalent:

Pattern matches: $p \triangleright v \rightsquigarrow \sigma$

$$\frac{}{x \triangleright v \rightsquigarrow [x \mapsto v]}\ \textsc{Var} \qquad \frac{}{_ \triangleright v \rightsquigarrow []}\ \textsc{Wild} \qquad \frac{p \not\triangleright v \rightsquigarrow \sigma}{\neg p \triangleright v \rightsquigarrow \sigma}\ \textsc{Neg}_1$$

$$\frac{p_1 \triangleright v \rightsquigarrow \sigma}{p_1 \parallel p_2 \triangleright v \rightsquigarrow \sigma}\ \textsc{Or}_1 \qquad \frac{p_2 \triangleright v \rightsquigarrow \sigma}{p_1 \parallel p_2 \triangleright v \rightsquigarrow \sigma}\ \textsc{Or}_2 \qquad \frac{p_1 \triangleright v \rightsquigarrow \sigma_1 \qquad p_2 \triangleright v \rightsquigarrow \sigma_2}{p_1 \,\&\, p_2 \triangleright v \rightsquigarrow \sigma_1 +\!\!+ \sigma_2}\ \textsc{And}$$

$$\frac{p_1 \triangleright v_1 \rightsquigarrow \sigma_1 \qquad \dots \qquad p_n \triangleright v_n \rightsquigarrow \sigma_n}{\mathcal{C}^n(p_1, \dots, p_n) \triangleright \mathcal{C}^n(v_1, \dots, v_n) \rightsquigarrow \sigma_1 +\!\!+ \dots +\!\!+ \sigma_n}\ \textsc{Ctor}$$

Pattern doesn't match: $p \not\triangleright v \rightsquigarrow \sigma$

$$\frac{}{\# \not\triangleright v \rightsquigarrow []}\ \textsc{Absurd} \qquad \frac{\exists i : p_i \not\triangleright v_i \rightsquigarrow \sigma}{\mathcal{C}^n(p_1, \dots, p_n) \not\triangleright \mathcal{C}^n(v_1, \dots, v_n) \rightsquigarrow \sigma}\ \textsc{Ctor}_1$$

$$\frac{p \triangleright v \rightsquigarrow \sigma}{\neg p \not\triangleright v \rightsquigarrow \sigma}\ \textsc{Neg}_2 \qquad \frac{\mathcal{C}^n \neq \mathcal{C}'^m}{\mathcal{C}^n(p_1, \dots, p_n) \not\triangleright \mathcal{C}'^m(v_1, \dots, v_m) \rightsquigarrow []}\ \textsc{Ctor}_2$$

$$\frac{p_1 \not\triangleright v \rightsquigarrow \sigma}{p_1 \,\&\, p_2 \not\triangleright v \rightsquigarrow \sigma}\ \textsc{And}_1 \qquad \frac{p_2 \not\triangleright v \rightsquigarrow \sigma}{p_1 \,\&\, p_2 \not\triangleright v \rightsquigarrow \sigma}\ \textsc{And}_2 \qquad \frac{p_1 \not\triangleright v \rightsquigarrow \sigma_1 \qquad p_2 \not\triangleright v \rightsquigarrow \sigma_2}{p_1 \parallel p_2 \not\triangleright v \rightsquigarrow \sigma_1 +\!\!+ \sigma_2}\ \textsc{Or}$$

Pattern is linear: $p\ \mathbf{lin}^+$ and $p\ \mathbf{lin}^-$

$$\frac{}{x\ \mathbf{lin}^\pm}\ \text{L-}\textsc{Var}^\pm \qquad \frac{}{\#\ \mathbf{lin}^\pm}\ \text{L-}\textsc{Absurd}^\pm \qquad \frac{}{_\ \mathbf{lin}^\pm}\ \text{L-}\textsc{Wild}^\pm$$

$$\frac{p_{1,2}\ \mathbf{lin}^+ \quad \mathrm{FV}^e(p_1) = \mathrm{FV}^e(p_2)}{p_1 \parallel p_2\ \mathbf{lin}^+}\ \text{L-}\textsc{Or}^+ \qquad \frac{p_{1,2}\ \mathbf{lin}^- \quad \mathrm{FV}^o(p_1) \cap \mathrm{FV}^o(p_2) = \emptyset}{p_1 \parallel p_2\ \mathbf{lin}^-}\ \text{L-}\textsc{Or}^-$$

$$\frac{p_{1,2}\ \mathbf{lin}^+ \quad \mathrm{FV}^e(p_1) \cap \mathrm{FV}^e(p_2) = \emptyset}{p_1 \,\&\, p_2\ \mathbf{lin}^+}\ \text{L-}\textsc{And}^+ \qquad \frac{p_{1,2}\ \mathbf{lin}^- \quad \mathrm{FV}^o(p_1) = \mathrm{FV}^o(p_2)}{p_1 \,\&\, p_2\ \mathbf{lin}^-}\ \text{L-}\textsc{And}^-$$

$$\frac{p\ \mathbf{lin}^-}{\neg p\ \mathbf{lin}^+}\ \text{L-}\textsc{Neg}^+ \qquad \frac{p\ \mathbf{lin}^+}{\neg p\ \mathbf{lin}^-}\ \text{L-}\textsc{Neg}^-$$

$$\frac{p_i\ \mathbf{lin}^+ \quad \bigcap_{i=1}^n \mathrm{FV}^e(p_i) = \emptyset}{C^n(p_1, \dots, p_n)\ \mathbf{lin}^+}\ \text{L-}\textsc{Ctor}^+ \qquad \frac{p_i\ \mathbf{lin}^- \quad \bigcup_{i=1}^n \mathrm{FV}^o(p_i) = \emptyset}{C^n(p_1, \dots, p_n)\ \mathbf{lin}^-}\ \text{L-}\textsc{Ctor}^-$$

Pattern is deterministic: $p\ \mathbf{det}$

$$\frac{}{x\ \mathbf{det}}\ \text{D-}\textsc{Var} \qquad \frac{}{_\ \mathbf{det}}\ \text{D-}\textsc{Wild} \qquad \frac{p_1\ \mathbf{det}}{\neg p_1\ \mathbf{det}}\ \text{D-}\textsc{Neg} \qquad \frac{}{\#\ \mathbf{det}}\ \text{D-}\textsc{Absurd}$$

$$\frac{p_{1,2}\ \mathbf{det} \quad p_1 \not\perp p_2}{p_1 \parallel p_2\ \mathbf{det}}\ \text{D-}\textsc{Or}_1 \qquad \frac{p_{1,2}\ \mathbf{det} \quad \mathrm{FV}^e(p_1) = \mathrm{FV}^e(p_2) = \emptyset}{p_1 \parallel p_2\ \mathbf{det}}\ \text{D-}\textsc{Or}_2$$

$$\frac{p_{1,2}\ \mathbf{det} \quad \neg p_1 \not\perp \neg p_2}{p_1 \,\&\, p_2\ \mathbf{det}}\ \text{D-}\textsc{And}_1 \qquad \frac{p_{1,2}\ \mathbf{det} \quad \mathrm{FV}^o(p_1) = \mathrm{FV}^o(p_2) = \emptyset}{p_1 \,\&\, p_2\ \mathbf{det}}\ \text{D-}\textsc{And}_2$$

$$\frac{p_1\ \mathbf{det} \cdots p_n\ \mathbf{det}}{C^n(p_1, \dots, p_n)\ \mathbf{det}}\ \text{D-}\textsc{Ctor}$$

Figure 2 Rules for matching patterns against values, and checking for linearity and determinism.

▶ **Definition 5** (Semantic equivalence of substitutions,). *Two substitutions σ and σ' are semantically equivalent if they contain the same mappings:*

$$[\![\,\sigma\,]\!] \equiv [\![\,\sigma'\,]\!] \coloneqq \forall m, m \in \sigma \Leftrightarrow m \in \sigma'$$

We can now state the definition for semantic equivalence of patterns.

▶ **Definition 6** (Semantic equivalence of patterns,). *Two patterns p and q are semantically equivalent if they match the same values with equivalent substitutions, and if they also do not match the same values with equivalent substitutions:*

$$\begin{aligned}[\![\,p\,]\!] \equiv [\![\,q\,]\!] \coloneqq \forall v, \forall \sigma,\ & p \triangleright v \rightsquigarrow \sigma \Rightarrow \exists \sigma', q \triangleright v \rightsquigarrow \sigma' \wedge [\![\,\sigma\,]\!] \equiv [\![\,\sigma'\,]\!] \\ \wedge\ & q \triangleright v \rightsquigarrow \sigma \Rightarrow \exists \sigma', p \triangleright v \rightsquigarrow \sigma' \wedge [\![\,\sigma\,]\!] \equiv [\![\,\sigma'\,]\!] \\ \wedge\ & p \not\triangleright v \rightsquigarrow \sigma \Rightarrow \exists \sigma', q \not\triangleright v \rightsquigarrow \sigma' \wedge [\![\,\sigma\,]\!] \equiv [\![\,\sigma'\,]\!] \\ \wedge\ & q \not\triangleright v \rightsquigarrow \sigma \Rightarrow \exists \sigma', p \not\triangleright v \rightsquigarrow \sigma' \wedge [\![\,\sigma\,]\!] \equiv [\![\,\sigma'\,]\!]\end{aligned}$$

We have to include both judgment forms if we want semantic equivalence to be a congruence relation. To see why this is the case, suppose that we omit the second half of Definition 6. It would then follow that the two patterns $\#$ and $\neg x$ are semantically equivalent since they match against the same set of values (i.e. no values at all). But if we apply a negation-pattern to both patterns, then we can easily show that the resulting patterns are no longer semantically equivalent: $[\![\,\neg\#\,]\!] \equiv [\![\,_\,]\!] \not\equiv [\![\,x\,]\!] \equiv [\![\,\neg\neg x\,]\!]$. But since Definition 6 requires both patterns to agree on the values they match *and* don't match against, it actually defines a congruence relation on patterns:

▶ **Theorem 7** (Congruence,). *If $[\![\,p_1\,]\!] \equiv [\![\,p_1'\,]\!]$ to $[\![\,p_n\,]\!] \equiv [\![\,p_n'\,]\!]$ hold, then we also have $[\![\,\neg p_1\,]\!] \equiv [\![\,\neg p_1'\,]\!]$, $[\![\,p_1 \,\&\, p_2\,]\!] \equiv [\![\,p_1' \,\&\, p_2'\,]\!]$, $[\![\,p_1 \parallel p_2\,]\!] \equiv [\![\,p_1' \parallel p_2'\,]\!]$ and $[\![\,\mathcal{C}(p_1, \ldots, p_n)\,]\!] \equiv [\![\,\mathcal{C}(p_1', \ldots, p_n')\,]\!]$.*

We can use the definition of semantic equivalence to prove that the following algebraic laws hold for patterns:

▶ **Theorem 8** (Algebraic Equivalences of Patterns, I,). *For all patterns p, q, r, the following equivalences hold:*

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}l}
[\![\,p \,\&\, q\,]\!] & \equiv & [\![\,q \,\&\, p\,]\!] & [\![\,p \parallel q\,]\!] & \equiv & [\![\,q \parallel p\,]\!] & \text{Commutativity} \\
[\![\,p \,\&\, (q \,\&\, r)\,]\!] & \equiv & [\![\,(p \,\&\, q) \,\&\, r\,]\!] & [\![\,p \parallel (q \parallel r)\,]\!] & \equiv & [\![\,(p \parallel q) \parallel r\,]\!] & \text{Associativity} \\
[\![\,p \,\&\, _\,]\!] & \equiv & [\![\,p\,]\!] & [\![\,p \parallel \#\,]\!] & \equiv & [\![\,p\,]\!] & \text{Neutral Elements} \\
[\![\,\neg_\,]\!] & \equiv & [\![\,\#\,]\!] & [\![\,\neg\#\,]\!] & \equiv & [\![\,_\,]\!] & \text{Duality} \\
[\![\,\neg(p \parallel q)\,]\!] & \equiv & [\![\,(\neg p) \,\&\, (\neg q)\,]\!] & [\![\,\neg(p \,\&\, q)\,]\!] & \equiv & [\![\,(\neg p) \parallel (\neg q)\,]\!] & \text{De Morgan} \\
\multicolumn{6}{c}{[\![\,\neg\neg p\,]\!] \equiv [\![\,p\,]\!]} & \text{Double Negation}
\end{array}$$

In addition to these boolean laws we can also prove the following equivalences which involve constructor patterns:

▶ **Theorem 9** (Equivalences of Constructor Patterns, I,). *For all patterns p_1 to p_n and p_1' to p_n', the following equivalences hold:*

$$\begin{aligned}[\![\,\mathcal{C}(p_1, \ldots, p_n) \,\&\, \mathcal{C}(p_1', \ldots, p_n')\,]\!] &\equiv [\![\,\mathcal{C}(p_1 \,\&\, p_1', \ldots, p_n \,\&\, p_n')\,]\!] \\ [\![\,\mathcal{C}(p_1, \ldots, p_i \parallel p_i', \ldots, p_n)\,]\!] &\equiv [\![\,\mathcal{C}(p_1, \ldots, p_i, \ldots, p_n) \parallel \mathcal{C}(p_1, \ldots, p_i', \ldots, p_n)\,]\!]\end{aligned}$$

There are still some equivalences that we expect to hold but which are missing from Theorem 8 and Theorem 9. These additional laws, like the distributive law for and- and or-patterns, are not universally valid and require additional restrictions on patterns. We will motivate and introduce these additional constraints in the next subsection.

3.2 Linear Patterns

The first restriction we impose on patterns is *linearity*. The linearity restriction is easy to state if the syntax of patterns does not include and-patterns, or-patterns and negation-patterns: A pattern is linear if every variable occurs at most once. To see why this restriction is necessary consider the nonlinear pattern $\texttt{Cons}(x, x)$, for which we can derive the following judgment:

$$\texttt{Cons}(x, x) \triangleright \texttt{Cons}(2, \texttt{Nil}) \rightsquigarrow [x \mapsto 2, x \mapsto \texttt{Nil}] \tag{6}$$

The problem here is that multiple values are bound to the same variable, i.e. the result of the pattern match is not a proper substitution.

We have a more expressive language of algebraic patterns, and the simple definition of linearity described above is no longer sufficient. To see why this is the case, consider the pattern $(x \,\&\, \texttt{True}) \parallel \texttt{False}$ that is linear according to the simple definition, but which shows the following pathological behavior:

$$(x \,\&\, \texttt{True}) \parallel \texttt{False} \triangleright \texttt{True} \rightsquigarrow [x \mapsto \texttt{True}] \tag{7}$$

$$(x \,\&\, \texttt{True}) \parallel \texttt{False} \triangleright \texttt{False} \rightsquigarrow [] \tag{8}$$

Equations (7) and (8) show that matching against this pattern can produce substitutions that do not have the same domain.

The two examples above tell us what the correct notion of linearity should guarantee: First, if a linear pattern matches against a value, then the resulting substitution should be a proper substitution which contains at most one mapping for a variable. Second, if we use a pattern in a clause we should be able to read off the variables that will be in the domain of the substitution of a successful match.

In order to ensure that these properties hold we have to use two mutually recursive judgments $p\ \mathbf{lin}^+$ and $p\ \mathbf{lin}^-$, whose rules are given in Figure 2. We use $p\ \mathbf{lin}^\pm$ for patterns which satisfy both $p\ \mathbf{lin}^+$ and $p\ \mathbf{lin}^-$. Theorems 10 and 11 show that linear patterns do have the properties described in the previous paragraph.

▶ **Theorem 10** (Linear patterns produce covering substitutions, Agda). *For any pattern p, value v and substitution σ:*

1. *If $p\ \mathbf{lin}^+$ and $p \triangleright v \rightsquigarrow \sigma$, then the domain of σ is equal to $\mathrm{FV}^e(p)$.*
2. *If $p\ \mathbf{lin}^-$ and $p \not\triangleright v \rightsquigarrow \sigma$, then the domain of σ is equal to $\mathrm{FV}^o(p)$.*

▶ **Theorem 11** (Linear patterns produce proper substitutions, Agda). *For any pattern p, value v and substitution σ:*

1. *If $p\ \mathbf{lin}^+$ and $p \triangleright v \rightsquigarrow \sigma$, then σ is a proper substitution.*
2. *If $p\ \mathbf{lin}^-$ and $p \not\triangleright v \rightsquigarrow \sigma$, then σ is a proper substitution.*

A substitution σ is proper if any variable occurs at most once in the domain of σ.

We will now motivate why the rules in Figure 2 have the specific form that they do. Theorem 10 tells us that we should think about the variables in $\mathrm{FV}^e(p)$ as the actual pattern variables which we can use in the right-hand side of a clause, and that we should think of the variables in $\mathrm{FV}^o(p)$ as the "temporarily deactivated" pattern variables "reactivated" by negation.

Let us first look at the restrictions on the variables in $\mathrm{FV}^e(p)$, since those are more intuitive. In the or-pattern $p_1 \parallel p_2$ we require that $\mathrm{FV}^e(p_1)$ is equal to $\mathrm{FV}^e(p_2)$ since we do not know which of the two patterns will match, and we have to ensure that the same variables appear in the substitution. In the and-pattern $p_1 \,\&\, p_2$ we require that $\mathrm{FV}^e(p_1)$ is

disjoint from $\mathrm{FV}^e(p_2)$ since we do not want the same variable to be mapped to two values in the substitution. The reasoning is very similar for constructor patterns $\mathcal{C}(p_1, \ldots, p_i)$ where we require all the $\mathrm{FV}^e(p_i)$ to be disjoint.

Next, let us look at the restrictions on $\mathrm{FV}^o(e)$, i.e. the variables which occur under an odd number of negations. The rules for and-patterns and or-patterns are motivated by duality. We have seen that the De Morgan rules are valid for patterns. We therefore want, for example, that if the pattern $\neg(p_1 \parallel p_2)$ is linear, the pattern $\neg p_1 \mathbin{\&} \neg p_2$ should be linear as well. For this to be true, the restrictions on $\mathrm{FV}^o(p)$ for or-patterns have to mirror the restrictions on $\mathrm{FV}^e(p)$ for and-patterns, and vice-versa. We still have to explain why we require for a constructor pattern $\mathcal{C}(p_1, \ldots, p_n)$ that all $\mathrm{FV}^o(p_i)$ have to be the empty set. The reason for this restriction lies in the following semantic equivalence, which is motivated by its importance for rewriting patterns into a normal form (see Section 5.1) and which we will prove below.

$$[\![\, \neg\mathcal{C}(p_1, \ldots, p_n) \,]\!] \equiv [\![\, \neg\mathcal{C}(_, \ldots, _) \parallel \mathcal{C}(\neg p_1, \ldots, _) \parallel \ldots \parallel \mathcal{C}(_, \ldots, \neg p_n) \,]\!]$$

The pattern on the right consists of several patterns that are joined by or-patterns. The rules for or-patterns require that each disjunct contains the same sets $\mathrm{FV}^e(-)$. But since the first pattern $\neg\mathcal{C}(_, \ldots, _)$ doesn't contain any variables, the other disjuncts must not contain any variables under an even number of negations. And since the subpatterns p_i occur under a negation, we can deduce that the patterns p_i must not contain any variables under an odd number of negations.

Using linearity we can now prove the missing equivalences which were not included in Theorems 8 and 9.

▶ **Theorem 12** (Algebraic equivalences of Patterns, II, 🏴‍☠️). *For all patterns p, q and r, the following equivalences hold if the patterns on both sides are linear (i.e. $\mathbf{lin}^{\pm}$).*

$[\![\, p \mathbin{\&} (q \parallel r) \,]\!] \equiv [\![\, (p \mathbin{\&} q) \parallel (p \mathbin{\&} r) \,]\!]$	$[\![\, p \parallel (q \mathbin{\&} r) \,]\!] \equiv [\![\, (p \parallel q) \mathbin{\&} (p \parallel r) \,]\!]$	Distributivity
$[\![\, p \mathbin{\&} p \,]\!] \equiv [\![\, p \,]\!]$	$[\![\, p \parallel p \,]\!] \equiv [\![\, p \,]\!]$	Idempotence
$[\![\, p \mathbin{\&} \# \,]\!] \equiv [\![\, \# \,]\!]$	$[\![\, p \parallel _ \,]\!] \equiv [\![\, _ \,]\!]$	Zeros

Theorem 8 and Theorem 12 taken together show that patterns form a boolean algebra, with the caveat that the conjunction or disjunction of two linear patterns does not necessarily result in a linear pattern. We can also prove some additional equivalences involving constructor patterns.

▶ **Theorem 13** (Equivalences of Constructor Patterns, II, 🏴‍☠️). *For all patterns p_1 to p_n and p'_1 to p'_n and constructors $\mathcal{C} \neq \mathcal{C}'$, the following equivalences hold if the patterns on both sides are linear (i.e. $\mathbf{lin}^{\pm}$).*

$$\begin{aligned}
[\![\, \mathcal{C}(p_1, \ldots, \#, \ldots, p_n) \,]\!] &\equiv [\![\, \# \,]\!] \\
[\![\, \mathcal{C}(p_1, \ldots, p_n) \mathbin{\&} \mathcal{C}'(p'_1, \ldots, p'_m) \,]\!] &\equiv [\![\, \# \,]\!] \\
[\![\, \mathcal{C}(p_1, \ldots, p_n) \mathbin{\&} \neg\mathcal{C}'(p'_1, \ldots, p'_m) \,]\!] &\equiv [\![\, \mathcal{C}(p_1, \ldots, p_n) \,]\!] \\
[\![\, \neg\mathcal{C}(p_1, \ldots, p_n) \,]\!] &\equiv [\![\, \neg\mathcal{C}(_, \ldots, _) \parallel \mathcal{C}(\neg p_1, \ldots, p_n) \parallel \ldots \parallel \mathcal{C}(p_1, \ldots, \neg p_n) \,]\!]
\end{aligned}$$

The equivalences of Theorem 13 are essential for proving the correctness of the normalization algorithm specified in Section 5.

3.3 Deterministic Patterns

We also have to introduce a notion of deterministic patterns. To see why this is necessary, consider the pathological pattern $\texttt{Pair}(x, _) \parallel \texttt{Pair}(_, x)$. This pattern is linear, but when we match against the value $\texttt{Pair}(2, 4)$ we can either bind the value 2 or the value 4 to the variable x. Deterministic patterns guarantee that when there are multiple derivations which show that a pattern matches against a value, then these alternatives must yield equivalent substitutions[3].

To define deterministic patterns we first have to introduce overlapping and disjoint patterns. In the extended version [2] we give a procedure for deciding whether two patterns in a certain normalized form – which we will introduce in Section 5 – overlap.

▶ **Definition 14** (Overlapping and Disjoint Patterns). *Two patterns p and q are overlapping, written $p \between q$, if there exists a value v and substitutions σ_1 and σ_2 such that $p \triangleright v \rightsquigarrow \sigma_1$ and $q \triangleright v \rightsquigarrow \sigma_2$. Two patterns p and q are disjoint, written $p \not\between q$ if such a value and such substitutions do not exist.*

We write p **det** for patterns which satisfy the determinism restriction specified in Figure 2. Note that there are two rules for or-patterns: The rule D-Or$_1$ says that an or-pattern is deterministic if the subpatterns are disjoint, whereas the rule D-Or$_2$ allows arbitrary deterministic subpatterns as long as they do not contain free variables. As for the linearity restriction, these two rules for or-patterns give rise to corresponding rules for and-patterns via De Morgan duality: Given that the pattern $\neg(p1 \parallel p2)$ is deterministic, we want the semantically equivalent pattern $\neg p_1 \,\&\, \neg p_2$ to be deterministic as well.

▶ **Theorem 15** (Matching for wellformed patterns is deterministic,). *For any pattern p, value v and substitutions σ_1, σ_2:*

1. *If p $\mathbf{lin}^+$, p $\mathbf{det}$ and $p \triangleright v \rightsquigarrow \sigma_1$ and $p \triangleright v \rightsquigarrow \sigma_2$, then $\sigma_1 \equiv \sigma_2$.*
2. *If p $\mathbf{lin}^-$, p $\mathbf{det}$ and $p \not\triangleright v \rightsquigarrow \sigma_1$, and $p \not\triangleright v \rightsquigarrow \sigma_2$, then $\sigma_1 \equiv \sigma_2$.*

This property is also necessary to show that the evaluation of terms is deterministic, since we use the relation $p \triangleright v \rightsquigarrow \sigma$ when we define the evaluation of pattern matches in the next subsection.

4 Terms and Types

In this section we present the second part of our solution: default clauses. We present a small language, specify its operational semantics and introduce the typing rules for patterns and expressions. The language contains only data types and pattern matching and omits higher-order functions, but these could be added in the standard way.

4.1 The Syntax of Expressions

Expressions e consist of variables x, the application of a constructor to arguments $\mathcal{C}(e_1, \ldots, e_n)$ and case expressions. Every case expression consists of a scrutinee, a list of normal clauses c which all have the form $p \Rightarrow e$, and a default clause $\mathbf{default} \Rightarrow e$.

$$\begin{array}{lcll} e & ::= & x \mid \mathcal{C}(e_1, \ldots, e_n) \mid \mathbf{case}\ e\ \mathbf{of}\ \{\ c_1, \ldots, c_n\ ;\ \mathbf{default} \Rightarrow e\ \} & \textit{Expression} \\ c & ::= & p \Rightarrow e & \textit{Clause} \end{array}$$

[3] Most languages which implement or-patterns do not check whether patterns are deterministic and instead evaluate or-patterns left-to-right, but this invalidates algebraic laws such as the commutativity of or-patterns. If we need such left-biased or-patterns $p \parallel^{\leftarrow} q$ then we can easily define them as $p \parallel (\neg p \,\&\, q)$.

Every case expression contains exactly one default clause, but a realistic language would also allow to omit the default clause if all other clauses are exhaustive. As we already mentioned, **default** does not belong to the syntax of patterns, but to the syntax of case expressions. In particular, the meaning of a default clause is not equal to the meaning of a clause $_ \Rightarrow e_d$ which uses a wildcard pattern, but equal to a clause $\neg p_1 \,\&\, \cdots \,\&\, \neg p_n \Rightarrow e_d$, where the p_i are all the patterns from the other clauses of the case expression. For this reason it is not possible to define the meaning of a default clause in isolation, unlike for normal clauses, and a default clause also does not immediately give rise to a valid equation for term rewriting.

4.2 Operational Semantics

We formalize a call-by-value semantics in which the only values v are constructors applied to other values. We use evaluation contexts E and a congruence rule E-Cong to specify evaluation within a subterm, following standard practice [9].

$$E \quad ::= \quad \Box \mid \mathcal{C}(v_1, \ldots, v_n, E, e_1, \ldots, e_m) \mid \mathbf{case}\ E\ \mathbf{of}\ \{\ c_1, \ldots, c_n\ ;\ \mathbf{default} \Rightarrow e\ \}$$

We write $e \to^r e'$ for the single-step reduction of the expression e to e' which does not use the congruence rule, and $e \to e'$ for the single step reduction within an evaluation context. The evaluation of case expressions is governed by the two rules E-Match and E-Default. The rule E-Match allows to reduce a case expression using any of the normal clauses that match the scrutinee; in particular, it is not necessary to try the clauses in the order in which they occur. On the right-hand side of that rule we write $e\,\sigma$ for the result of applying the substitution σ to the expression e. The rule E-Default allows to reduce a case expression to the expression of the default clause if none of the other clauses match against the scrutinee.

Single-step evaluation: $e \to e$

$$\frac{p \triangleright v \rightsquigarrow \sigma}{\mathbf{case}\ v\ \mathbf{of}\ \{c_1, \ldots, c_n, p \Rightarrow e, c_{n+2}, \ldots, c_m\ ;\ \mathbf{default} \Rightarrow e_d\} \to^r e\sigma}\ \text{E-Match}$$

$$\frac{p_1 \not\triangleright v \rightsquigarrow \sigma_1 \quad \cdots \quad p_n \not\triangleright v \rightsquigarrow \sigma_n}{\mathbf{case}\ v\ \mathbf{of}\ \{\ p_1 \Rightarrow e_1, \ldots, p_n \Rightarrow e_n\ ;\ \mathbf{default} \Rightarrow e_d\ \} \to^r e_d}\ \text{E-Default}$$

$$\frac{e \to^r e'}{E[e] \to E[e']}\ \text{E-Cong}$$

Nothing in these rules guarantees that evaluation is deterministic. For example, we can write a case expression with overlapping patterns. If we want evaluation to be deterministic, then we have to check that expressions are wellformed, written as $e\ \checkmark$. For this we have to check that all patterns which occur in case expressions are deterministic, positively linear, and that no two clauses in a case expression have overlapping patterns.

Wellformed expressions: $e\ \checkmark$

$$\frac{}{x\ \checkmark}\ \text{Wf-Var} \qquad \frac{e_1\ \checkmark \cdots e_n\ \checkmark}{\mathcal{C}(e_1, \ldots, e_n)\ \checkmark}\ \text{Wf-Ctor}$$

$$\frac{\{e, e_d, e_1, \ldots, e_n\}\ \checkmark \qquad p_i\ \mathbf{det} \quad p_i\ \mathbf{lin}^+ \quad \text{if } i \neq j \text{ then } p_i \not\between p_j}{\mathbf{case}\ e\ \mathbf{of}\ \{\ p_1 \Rightarrow e_1, \ldots, p_n \Rightarrow e_n\ ;\ \mathbf{default} \Rightarrow e_d\ \}\ \checkmark}\ \text{Wf-Case}$$

The evaluation of wellformed expressions is deterministic. Furthermore, to check whether an expression is wellformed does not depend on any type information.

▶ **Theorem 16** (Evaluation of wellformed expressions is deterministic). *For all expressions e which are wellformed (that is e ✓ holds), evaluation is deterministic: If $e \to e_1$ and $e \to e_2$, then $e_1 = e_2$.*

Multi-step evaluation $\to^*$ is defined as the reflexive-transitive closure of single-step evaluation and allows us to define when two closed expressions are semantically equivalent:

▶ **Definition 17** (Semantic Equivalence of Expressions). *Two closed expressions e and e' are semantically equivalent, written $[\![e]\!] \equiv [\![e']\!]$, if either both of them evaluate to the same value v, i.e. $e \to^* v$ and $e' \to^* v$, or if both expressions diverge.*

▶ **Theorem 18** (Permutation of clauses does not change semantics). *The meaning of a wellformed pattern match does not change when we permute clauses. If $\overline{c'}$ is a permutation of the list of clauses $\overline{c}$, then $[\![\mathbf{case}\ v\ \mathbf{of}\ \{\overline{c}\ ; \mathbf{default} \Rightarrow e_d\}]\!] \equiv [\![\mathbf{case}\ v\ \mathbf{of}\ \{\overline{c'}\ ; \mathbf{default} \Rightarrow e_d\}]\!]$.*

This theorem makes formal what we call pure no-overlap semantics for pattern matching. The next theorem guarantees that we can use algebraic principles when reasoning about patterns in clauses.

▶ **Theorem 19** (Substitution of equivalent patterns preserves semantics). *If $[\![p_1]\!] \equiv [\![p_2]\!]$, then*

$$[\![\mathbf{case}\ v\ \mathbf{of}\ \{p_1 \Rightarrow e, \overline{c}\ ; \mathbf{default} \Rightarrow e_d\}]\!] \equiv [\![\mathbf{case}\ v\ \mathbf{of}\ \{p_2 \Rightarrow e, \overline{c}\ ; \mathbf{default} \Rightarrow e_d\}]\!].$$

4.3 Typing Rules

In this section, we are going to introduce types, typing contexts and (pattern) typing rules. In order to illustrate the typing rules we include three simple data types: the boolean type $\mathbb{B}$ with constructors `True` and `False`, the type of pairs $\tau \otimes \tau$ with the constructor `Pair`, and the type of binary sums $\tau \oplus \tau$ with constructors ι_1 and ι_2.

$$\begin{array}{lcll} \tau & ::= & \mathbb{B} \mid \tau \otimes \tau \mid \tau \oplus \tau & \textit{Types} \\ \Gamma, \Delta & ::= & \cdot \mid \Gamma, x : \tau & \textit{Typing contexts} \end{array}$$

We use the judgment $\Gamma; \Delta \Rightarrow p : \tau$ to express that the pattern p matches against a value of type τ and binds variables whose types are recorded in the context Γ on a successful match, and in context Δ in the case of a non-successful match. In other words, the typing context Γ is used for typing the pattern variables that occur under an even number of negations, and the typing context Δ is used for typing the variables under an odd number of negations. We write $\Gamma_1 +\!\!+ \Gamma_2$ for the naive concatenation of two typing contexts.

Pattern typing: $\Gamma; \Delta \Rightarrow p : \tau$

$$\frac{}{x : \tau; \cdot \Rightarrow x : \tau}\ \text{P-Var} \qquad \frac{}{\cdot; \cdot \Rightarrow _ : \tau}\ \text{P-Wildcard} \qquad \frac{}{\cdot; \cdot \Rightarrow \# : \tau}\ \text{P-Absurd}$$

$$\frac{\Gamma_1; \Delta \Rightarrow p_1 : \tau \quad \Gamma_2; \Delta \Rightarrow p_2 : \tau}{\Gamma_1 +\!\!+ \Gamma_2; \Delta \Rightarrow p_1 \,\&\, p_2 : \tau}\ \text{P-And} \qquad \frac{\Gamma; \Delta_1 \Rightarrow p_1 : \tau \quad \Gamma; \Delta_2 \Rightarrow p_2 : \tau}{\Gamma; \Delta_1 +\!\!+ \Delta_2 \Rightarrow p_1 \parallel p_2 : \tau}\ \text{P-Or}$$

$$\frac{\Gamma; \Delta \Rightarrow p : \tau}{\Delta; \Gamma \Rightarrow \neg p : \tau}\ \text{P-Neg} \qquad \frac{}{\cdot; \cdot \Rightarrow \texttt{True} : \mathbb{B}}\ \text{P-True} \qquad \frac{}{\cdot; \cdot \Rightarrow \texttt{False} : \mathbb{B}}\ \text{P-False}$$

$$\frac{\Gamma; \Delta \Rightarrow p : \tau_1}{\Gamma; \Delta \Rightarrow \iota_1(p) : \tau_1 \oplus \tau_2}\ \text{P-Inl} \qquad \frac{\Gamma; \Delta \Rightarrow p : \tau_2}{\Gamma; \Delta \Rightarrow \iota_2(p) : \tau_1 \oplus \tau_2}\ \text{P-Inr}$$

$$\frac{\Gamma_1; \Delta_1 \Rightarrow p_1 : \tau_1 \qquad \Gamma_2; \Delta_2 \Rightarrow p_2 : \tau_2}{\Gamma_1 +\!\!+ \Gamma_2; \Delta_1 +\!\!+ \Delta_2 \Rightarrow \texttt{Pair}(p_1, p_2) : \tau_1 \otimes \tau_2} \text{ P-Pair}$$

These pattern typing rules do not make sense on their own, but only in conjunction with the rules for linearity. For example, the typing rules allow to derive the typing judgment $x : \tau_1, x : \tau_2; \cdot \Rightarrow \texttt{Pair}(x, x) : \tau_1 \otimes \tau_2$, but the linearity rules disallow this pattern[4]. Lastly, we present the rules for typing expressions:

Expression typing: $\Gamma \vdash e : \tau$

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash e_d : \tau \qquad \forall i : \ \Gamma_i; \Delta \Rightarrow p_i : \sigma \text{ and } \Gamma +\!\!+ \Gamma_i \vdash e_i : \tau}{\Gamma \vdash \textbf{case } e \textbf{ of } \{\ p_1 \Rightarrow e_1, \ldots, p_n \Rightarrow e_n\ ; \textbf{ default} \Rightarrow e_d\ \} : \tau} \text{ T-Match}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \text{ T-Var} \qquad \frac{}{\Gamma \vdash \texttt{True} : \mathbb{B}} \text{ T-True} \qquad \frac{}{\Gamma \vdash \texttt{False} : \mathbb{B}} \text{ T-False}$$

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{Pair}(e_1, e_2) : \sigma \otimes \tau} \text{ T-Pair} \qquad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \iota_1(e) : \sigma \oplus \tau} \text{ T-Inl} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \iota_2(e) : \sigma \oplus \tau} \text{ T-Inr}$$

In the rule T-Match we use the pattern typing rule for the pattern in each clause, and add the resulting typing context Γ_i which contains the types of all pattern variables under an even number of negations to the outer typing context Γ in the respective clause.

5 Normalizing Algebraic Patterns

So far, we have considered algebraic patterns in their unrestricted form which is written by the programmer and checked by the compiler. In this section, we consider normal forms for patterns which simplify the compilation algorithm in Section 6. We end up with a normal form that is a variant of disjunctive normal forms, but which takes the use of constructors and variables into account. We describe a normalization procedure which proceeds in three steps: The first step, described in Section 5.1, pushes negation-patterns inward so that they only occur applied to constructors without subpatterns, or to variables. The second step, described in Section 5.2, pushes or-patterns outwards, so that we end up with a disjunctive normal form, that is a disjunction of elementary conjuncts. In the third step, we further simplify these elementary conjuncts; this is described in Section 5.3. The default clause is not affected by the normalization procedure described in this section; the compilation of case expressions to decision trees described in the next section can handle default clauses natively.

5.1 Computing the Negation Normal Form

Consider the pattern $\neg\texttt{Pair}(\texttt{True}, \texttt{False})$ which matches anything except a tuple consisting of the boolean values `True` and `False`. Another way to express the behavior of this pattern is to say that it matches anything which isn't a tuple, or it matches a tuple if either the left element isn't the value `True` or the right element isn't the value `False`. We can therefore also write the pattern as $\neg\texttt{Pair}(_, _) \parallel \texttt{Pair}(\neg\texttt{True}, _) \parallel \texttt{Pair}(_, \neg\texttt{False})$. We can generalize this observation and use it to push all negation-patterns inside, by repeatedly rewriting the pattern using the last equation of Theorem 13. After we have normalized patterns in this way, negation-patterns only occur applied to variables or to constructors where all subpatterns of the constructor are wildcard patterns. We abbreviate these negated constructor patterns $\neg\mathcal{C}^n(_, \ldots, _)$ as $\neg\mathcal{C}^n$.

[4] An implementation might choose to check linearity and typing of patterns simultaneously, but we prefer to present them separately to illustrate that we can also use algebraic patterns in the untyped setting.

▶ **Definition 20** (Negation Normal Forms). *The syntax of negation normal forms is:*

$$N \quad ::= \quad x \mid \neg x \mid \neg \mathcal{C}^n \mid \mathcal{C}^n(N_1, \ldots, N_n) \mid N \,\&\, N \mid N \parallel N \mid _ \mid \#$$

$$\begin{array}{rlrl}
\mathtt{nnf}^-(p_1 \,\&\, p_2) & := \mathtt{nnf}^-(p_1) \parallel \mathtt{nnf}^-(p_2) & \mathtt{nnf}^+(p_1 \,\&\, p_2) & := \mathtt{nnf}^+(p_1) \,\&\, \mathtt{nnf}^+(p_2) \\
\mathtt{nnf}^-(p_1 \parallel p_2) & := \mathtt{nnf}^-(p_1) \,\&\, \mathtt{nnf}^-(p_2) & \mathtt{nnf}^+(p_1 \parallel p_2) & := \mathtt{nnf}^+(p_1) \parallel \mathtt{nnf}^+(p_2) \\
\mathtt{nnf}^-(\neg p) & := \mathtt{nnf}^+(p) & \mathtt{nnf}^+(\neg p) & := \mathtt{nnf}^-(p) \\
\mathtt{nnf}^-(\#) & := _ & \mathtt{nnf}^+(\#) & := \# \\
\mathtt{nnf}^-(_) & := \# & \mathtt{nnf}^+(_) & := _ \\
\mathtt{nnf}^-(x) & := \neg x & \mathtt{nnf}^+(x) & := x
\end{array}$$

$$\begin{array}{ll}
\mathtt{nnf}^-(\mathcal{C}^n(p_1, \ldots, p_n)) & := \neg\mathcal{C}^n \parallel \mathcal{C}^n(\mathtt{nnf}^-(p_1), \ldots, _) \parallel \ldots \parallel \mathcal{C}^n(_, \ldots, \mathtt{nnf}^-(p_n)) \\
\mathtt{nnf}^+(\mathcal{C}^n(p_1, \ldots, p_n)) & := \mathcal{C}^n(\mathtt{nnf}^+(p_1), \ldots, \mathtt{nnf}^+(p_n))
\end{array}$$

The normalization procedure $\mathtt{nnf}(-)$ is split into the two functions $\mathtt{nnf}^-(-)$ and $\mathtt{nnf}^+(-)$ which track if we are currently normalizing a negated or a non-negated pattern. At the top-level, the function $\mathtt{nnf}(-)$ therefore just invokes $\mathtt{nnf}^+(-)$.

▶ **Example 21.** We compute the negation normal form of our running example.

$$\mathtt{nnf}^+(x \,\&\, (\mathtt{Sa} \parallel \mathtt{Su})) = x \,\&\, (\mathtt{Sa} \parallel \mathtt{Su}) \qquad \mathtt{nnf}^+(x \,\&\, \neg(\mathtt{Sa} \parallel \mathtt{Su})) = x \,\&\, (\neg\mathtt{Sa}) \,\&\, (\neg\mathtt{Su})$$

We treat the negation normal forms N as a subset of patterns p, and hence all functions and properties of patterns can also be applied to normal forms. If we translate a pattern into negation normal form then the pattern is still wellformed, can be typed using the same context and type, and has the same semantics as before. This is witnessed by the following lemma.

▶ **Lemma 22** (Negation Normalization Preserves Linearity, Typing and Semantics). *For all patterns p, contexts Γ, Δ and types τ, we have:*

1. *Linearity is preserved: If p $\mathbf{lin}^+$, then $\mathtt{nnf}(p)$ $\mathbf{lin}^+$.*
2. *Typing is preserved: If $\Gamma; \Delta \Rightarrow p : \tau$, then $\Gamma; \Delta \Rightarrow \mathtt{nnf}(p) : \tau$.*
3. *The patterns are semantically equivalent: If p $\mathbf{lin}^\pm$, then $[\![p]\!] \equiv [\![\mathtt{nnf}(p)]\!]$.*

Proof. Available in the appendix of the extended version of this paper [2]. ◀

5.2 Computing the Disjunctive Normal Form

The next step is to bring the negation normal form into a disjunctive normal form. Patterns in disjunctive normal form consist of an outer disjunction of so-called "elementary conjuncts" which do not contain or-patterns. The syntax of the disjunctive normal forms D and elementary conjuncts K is given below. We can normalize a pattern in negation normal form to the disjunctive normal form by the function $\mathtt{dnf}(-)$ which takes a pattern in negation normal form and returns a set of elementary conjuncts. At the top-level we have to wrap the result in a $\parallel \{\ldots\}$ node.

▶ **Definition 23** (Disjunctive Normal Form).

$$\begin{array}{rcll}
D & ::= & \parallel \{K_1, \ldots, K_n\} & \text{Disjunctive Normal Form} \\
K & ::= & x \mid \neg x \mid \neg\mathcal{C}^n \mid \mathcal{C}^n(K_1, \ldots, K_n) \mid K \,\&\, K \mid _ \mid \# & \text{Elementary Conjunct}
\end{array}$$

$$\mathtt{dnf}(x) := \{x\} \quad \mathtt{dnf}(\neg x) := \{\neg x\} \quad \mathtt{dnf}(\neg\mathcal{C}^n) := \{\neg\mathcal{C}^n\} \quad \mathtt{dnf}(_) := \{_\} \quad \mathtt{dnf}(\#) := \{\#\}$$

$$\begin{array}{rcl}
\mathtt{dnf}(\mathcal{C}^n(N_1, \ldots, N_n)) & := & \{\mathcal{C}(k_1, \ldots, k_n) \mid k_1 \in \mathtt{dnf}(N_1), \ldots, k_n \in \mathtt{dnf}(N_n)\} \\
\mathtt{dnf}(N_1 \,\&\, N_2) & := & \{k_1 \,\&\, k_2 \mid k_1 \in \mathtt{dnf}(N_1), k_2 \in \mathtt{dnf}(N_2)\} \\
\mathtt{dnf}(N_1 \parallel N_2) & := & \mathtt{dnf}(N_1) \cup \mathtt{dnf}(N_2)
\end{array}$$

▶ **Example 24.** We compute the disjunctive normal form for our running example.

$$\texttt{dnf}(x \,\&\, (\texttt{Sa} \parallel \texttt{Su})) = \parallel \{x \,\&\, \texttt{Sa}, x \,\&\, \texttt{Su}\} \qquad \texttt{dnf}(x \,\&\, (\neg\texttt{Sa}) \,\&\, (\neg\texttt{Su})) = \parallel \{x \,\&\, (\neg\texttt{Sa}) \,\&\, (\neg\texttt{Su})\}$$

We can embed the disjunctive normal form $\parallel \{K_1, \ldots, K_n\}$ into patterns as $K_1 \parallel \ldots \parallel K_n$. This set of conjuncts is by design never empty ($n > 0$), however, if we do admit this case ($n = 0$) it translates to the absurd-pattern $\#$ and is treated as such in any use case (see Section 6).

5.3 Normalizing Elementary Conjuncts

As a last step, we further simplify the elementary conjuncts of Section 5.2. We write $\overline{D}$ and $\overline{K}$ for these normalized disjunctive normal forms and normalized elementary conjuncts:

▶ **Definition 25** (Normalized Disjunctive Normal Form)**.**

$$\begin{array}{lcl} \overline{D} & ::= & \parallel \{\overline{K}_1, \ldots, \overline{K}_n\} \\ \overline{K} & ::= & \{x_1, \ldots, x_n\} \,\&\, \mathcal{C}(\overline{K}_1, \ldots, \overline{K}_m) \mid \{x_1, \ldots, x_n\} \,\&\, \neg\{\mathcal{C}_1, \ldots, \mathcal{C}_m\} \mid \{x_1, \ldots, x_n\} \,\&\, \# \end{array}$$

These normalized conjuncts follow from the observation that there are essentially only three different kinds of conjuncts: positive, negative and unsatisfiable. Each of these three conjuncts can bind a set of variables $\{x_1, \ldots, x_n\}$. A *positive conjunct* $\{x_1, \ldots, x_n\} \,\&\, \mathcal{C}(\overline{K}_1, \ldots, \overline{K}_m)$ matches against any value headed by the constructor $\mathcal{C}$, and binds that value against all the variables in the set. A *negative conjunct* $\{x_1, \ldots, x_n\} \,\&\, \neg\{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$ matches against any value which is headed by a constructor which is *not* in the list $\{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$. The negative conjunct $\{x\} \& \neg\{\}$ can be used to represent a single pattern variable x, and similarly $\{\} \& \neg\{\}$ can be used to represent a wildcard pattern. An *unsatisfiable conjunct* $\{x_1, \ldots, x_n\} \,\&\, \#$ never matches against a value.

Computing these normalized conjuncts consists in collecting all the variables, as well as all negated and non-negated head constructors of an elementary conjunct. Intuitively, the head constructors are constructors that occur in a pattern without being nested in a subpattern of another constructor. An elementary conjunct is unsatisfiable if two different non-negated constructors appear in the conjunct, or if a head constructor appears both negated and non-negated. An elementary conjunct can be simplified to a negative conjunct if only negated head constructors appear in the original conjunct. Lastly, an elementary conjunct can be simplified to a positive conjunct if neither of the two cases above apply. We write $\texttt{norm}(-)$ for the function which computes these normalized conjuncts; the formal definition of this function is given in the appendix of the extended version of this paper [2]. We also remove negated variables in this step and treat them like absurd patterns; this is, strictly speaking, not semantics preserving, but negated variables cannot be used in the right-hand side of a clause anyway, so this has no effect on the compilation of programs.

▶ **Example 26.** We compute the normalized disjunctive normal form for our running example.

$$\begin{array}{rl} \texttt{norm}(\parallel \{x \,\&\, \texttt{Sa}, x \,\&\, \texttt{Su}\}) & = \parallel \{\{x\} \,\&\, \texttt{Sa}, \{x\} \,\&\, \texttt{Su}\} \\ \texttt{norm}(\parallel \{x \,\&\, (\neg\texttt{Sa}) \,\&\, (\neg\texttt{Su})\}) & = \parallel \{\{x\} \,\&\, \neg\{\texttt{Sa}, \texttt{Su}\}\} \end{array}$$

Like in Section 5.2, we treat the normal forms $\overline{D}$ and $\overline{K}$ as subsets of patterns p. We use patterns in normalized disjunctive normal form in the next section where we define a pattern matching compilation algorithm for nested patterns.

6 Compiling Algebraic Patterns

We have shown that algebraic patterns are more expressive because they explicitly allow us to speak about the complement of a pattern. But programmers won't use them if they have to pay for the gain in expressivity with slower programs. In order to compile programs using these patterns to efficient code we therefore have to use and modify one of the available pattern matching compilation algorithms.

These algorithms can be divided into two general classes: Algorithms which use backtracking [1] and algorithms which compile to decision trees [23]. The algorithms which use backtracking guarantee that the size of the generated code is linear in the size of the input, but the generated code might perform redundant tests. Algorithms which compile to decision trees never examine scrutinees more than once but the generated code might be exponential in the size of the input. Both classes of algorithms are competitive when they are optimized [20, 23], so it is not possible to say which alternative is preferable in general.

In this article we modify and extend the compilation scheme laid out by Maranget [23] which is based on decision trees. His algorithm already includes a treatment of or-patterns, and we will extend it to the normalized patterns that we introduced in Section 5. We introduce the general idea of the compilation to decision trees in Section 6.1, present the central branching step in Section 6.2 in detail and illustrate the algorithm with an example in Section 6.3. We conclude with a statement on the correctness of our extended compilation scheme in Section 6.4.

6.1 General Idea of the Algorithm

The algorithm operates on a generalization of the case expression **case** v **of** $\{\ p_1 \Rightarrow e_1, \ldots, p_n \Rightarrow e_n; \textbf{default} \Rightarrow e\ \}$ that we introduced in Section 4.1. Instead of only one scrutinee v and one pattern p in each clause we now allow a vector $v_1, \ldots, v_n$ of scrutinees and a matrix of clauses with patterns in nDNF (see Section 5.3).

$$\textbf{case } v_1, \ldots, v_n \textbf{ of } \begin{bmatrix} \overline{D}_1^1 & \ldots & \overline{D}_n^1 & \Rightarrow e^1 \\ & \vdots & & \\ \overline{D}_1^m & \ldots & \overline{D}_n^m & \Rightarrow e^m \\ & \textbf{default} & & \Rightarrow e_d \end{bmatrix} \tag{Input}$$

In such multi-column case expressions, the i-th pattern column tests the i-th scrutinee v_i. The main idea of the algorithm is that we take such a multi-column case expression as input and examine all scrutinees – and subscrutinees – one by one, producing a nesting of single-column case expressions with *simple* patterns as output. These simple patterns consist only of constructors applied to variables. We start the compilation by embedding an ordinary case expression into this generalized form on which we call the function *compile*. This function can take one of three steps. It either terminates with a **Default** or **Simple** step, or it examines one of the scrutinees in a **Branch** step.

Default If **default** $\Rightarrow e_d$ is the only clause then we terminate with the expression e_d.

Simple If the first row consists only of variable patterns of the form $\{x_1, \ldots x_k\}$ & $\neg\{\}$ then compilation terminates with the right-hand side of that clause together with appropriate substitutions. This rule also applies if $n = 0$.

Branch If neither of the above steps apply we pick a column i (with $1 \leq i \leq n$) and examine its scrutinee v_i in a single-column case against simple constructor patterns.

All the complexity of the algorithm lies within the **Branch** step, so we will discuss its details in the next subsection.

6.2 The Branching Step

The first step is to choose on which index i we want to perform the case split. This choice does not affect the correctness of the algorithm; selecting a column is a therefore a matter of optimization which other authors [26, 23] have discussed in detail. We only enforce that the selected pattern column must contain outermost constructors. Such a column must necessarily exist if neither the **Default** nor the **Simple** step apply.

Next, we have to gather the constructors against which we can test the scrutinee v_i. We collect these so-called head constructors into a set $\mathcal{H}(\overline{D}_i)$ defined below.

$$\mathcal{H}(\overline{D}_i) := \bigcup_{j=1,\ldots,m} \mathtt{head}(\overline{D}_i^j) \;\; \textit{with} \quad \begin{array}{lcl} \mathtt{head}(\| \{\overline{K}_1, \ldots, \overline{K}_t\}) & := & \bigcup\limits_{l=1,\ldots,t} \mathtt{head}(\overline{K}_l) \\ \mathtt{head}(\{x_1, \ldots, x_s\} \,\&\, \mathcal{C}(\overline{K}_1, \ldots, \overline{K}_t)) & := & \{\mathcal{C}\} \\ \mathtt{head}(\{x_1, \ldots, x_s\} \,\&\, \neg\{\mathcal{C}_1, \ldots, \mathcal{C}_t\}) & := & \{\mathcal{C}_1, \ldots, \mathcal{C}_t\} \\ \mathtt{head}(\{x_1, \ldots, x_s\} \,\&\, \#) & := & \emptyset \end{array}$$

Once we have found these head constructors $\{\mathcal{C}^{n_1}, \ldots, \mathcal{C}^{n_z}\}$ we generate fresh pattern variables x_1^k to $x_{n_k}^k$ for each constructor $\mathcal{C}^{n_k}$ and a simple pattern match. For each clause we have gained information about the scrutinee v_i: Within the right-hand side of a constructor clause, we know that our scrutinee must have conformed to the shape of our constructor. Likewise, if we arrive at the right-hand side of the default clause, we know that our scrutinee must have been different to all the constructors $\{\mathcal{C}^{n_1}, \ldots, \mathcal{C}^{n_z}\}$ before. We use this knowledge by computing clause-specific *subproblems* of our initial Input and we do so via the function $\mathcal{S}$ for constructor clauses and function $\mathcal{D}$ for the default clause.

$$\textbf{case } v_i \textbf{ of } \begin{bmatrix} \mathcal{C}^{n_1}(x_1^1, \ldots, x_{n_1}^1) & \Rightarrow & \mathit{compile}(\mathcal{S}(i, \mathcal{C}^{n_1}(x_1^1, \ldots, x_{n_1}^1), \mathrm{Input})) \\ \vdots & & \vdots \\ \mathcal{C}^{n_z}(x_1^z, \ldots, x_{n_z}^z) & \Rightarrow & \mathit{compile}(\mathcal{S}(i, \mathcal{C}^{n_z}(x_1^z, \ldots, x_{n_z}^z), \mathrm{Input})) \\ \textbf{default} & \Rightarrow & \mathit{compile}(\mathcal{D}(i, \{\mathcal{C}^{n_1}, \ldots, \mathcal{C}^{n_z}\}, \mathrm{Input})) \end{bmatrix} \qquad \text{(Output)}$$

All that is left is to compute these subproblems of *specialization* and *default* which are the arguments of recursive invocations of the compile function.

6.2.1 Computing Specialization

We now show how to compute $\mathcal{S}(i, \mathcal{C}^k(x_1, \ldots, x_k), \mathrm{Input})$ which specializes Input into a constructor-specific subproblem that utilizes *the assumption that the scrutinee v_i has already matched the constructor pattern $\mathcal{C}^k(x_1, \ldots, x_k)$*. We therefore remove the old scrutinee v_i and replace it with the bound variables $x_1, \ldots, x_k$ as new scrutinees.

$$\mathcal{S}(i, \mathcal{C}^k(x_1, \ldots, x_k), \mathrm{Input}) \;=\; \textbf{case } x_1, \ldots, x_k, v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n \textbf{ of } \left[S\right]$$

We now have to build a new clause matrix $\left[S\right]$ that matches against the new list of scrutinees and do so by iterating over the previous clauses. Each clause of the old Input contributes to $\left[S\right]$ only if its i-th pattern could have matched the old scrutinee v_i of shape $\mathcal{C}^k$ by *assumption*. This is, of course, the case for the default clause which we keep unchanged. Otherwise, these i-th patterns are in normalized DNF, and we have to check whether one of its elementary conjuncts could have matched. The following list gives all inclusion rules.

1. Row $\left[\textbf{default} \Rightarrow e_d\right]$ is added to $\left[S\right]$ unchanged.
2. Row $\left[\overline{D}_1^j \;\ldots\; \overline{D}_{i-1}^j \;\; \overline{\mathbf{D}}_\mathbf{i}^\mathbf{j} \;\; \overline{D}_{i+1}^j \;\ldots\; \overline{D}_n^j \;\Rightarrow\; e^j\right]$ only contributes according to $\overline{\mathbf{D}}_\mathbf{i}^\mathbf{j}$:
   a. If $\overline{\mathbf{D}}_\mathbf{i}^\mathbf{j} = \{y_1, \ldots y_s\} \,\&\, \mathcal{C}^k(\overline{K}_1, \ldots, \overline{K}_k)$,
   then add the row $\left[\overline{K}_1 \ldots \overline{K}_k \;\; \overline{D}_1^j \ldots \overline{D}_{i-1}^j \overline{D}_{i+1}^j \ldots \; \overline{D}_n^j \;\Rightarrow e^j[\{y_1, \ldots y_s\} \mapsto v_i]\right]$.
   b. If $\overline{\mathbf{D}}_\mathbf{i}^\mathbf{j} = \{y_1, \ldots y_s\} \,\&\, \neg\{\mathcal{C}_1, \ldots, \mathcal{C}_t\}$ and $\mathcal{C}^k \notin \{\mathcal{C}_1, \ldots, \mathcal{C}_t\}$,
   then add the row $\left[_ \;\ldots\; _ \;\; \overline{D}_1^j \ldots \overline{D}_{i-1}^j \overline{D}_{i+1}^j \ldots \; \overline{D}_n^j \;\Rightarrow e^j[\{y_1, \ldots y_s\} \mapsto v_i]\right]$.
   (Note that we use $_$ simply as a shorthand for the equivalent nDNF $\{\} \,\&\, \neg\{\}$)
   c. If $\overline{\mathbf{D}}_\mathbf{i}^\mathbf{j} = \|\{\overline{K}_1, \ldots, \overline{K}_t\}$, then
   recursively iterate the rows of $\begin{bmatrix} \overline{D}_1^j & \ldots & \overline{D}_{i-1}^j & \overline{\mathbf{K}}_1 & \overline{D}_{i+1}^j & \ldots & \overline{D}_n^j & \Rightarrow & e^j \\ & & & \vdots & & & & & \\ \overline{D}_1^j & \ldots & \overline{D}_{i-1}^j & \overline{\mathbf{K}}_t & \overline{D}_{i+1}^j & \ldots & \overline{D}_n^j & \Rightarrow & e^j \end{bmatrix}$.

We give an example for case 2b which handles negation. Suppose that want to specialize the following case expression into a case expression *subproblem* which assumes that v_1 was matched against $\texttt{Cons}(x, y)$.

$$\mathit{compile}\left(\begin{array}{l}\textbf{case } v_1, v_2 \textbf{ of} \\ \begin{bmatrix} \{\} \,\&\, \neg\{\texttt{Cons}\} & \overline{D}_2^1 & \Rightarrow e^1 \\ \{\} \,\&\, \neg\{\texttt{Nil}\} & \overline{D}_2^2 & \Rightarrow e^2 \\ \textbf{default} & & \Rightarrow e_d \end{bmatrix}\end{array}\right) \quad = \quad \begin{array}{l}\textbf{case } v_1 \textbf{ of} \\ \begin{bmatrix} \texttt{Cons}(x, y) & \Rightarrow \mathit{compile}(\mathit{subproblem}) \\ & \vdots \end{bmatrix}\end{array}$$

The first clause is discarded, because the pattern $\{\} \,\&\, \neg\{\texttt{Cons}\}$ is incompatible with the assumption that v_i matches $\texttt{Cons}$. The second clause using $\{\} \,\&\, \neg\{\texttt{Nil}\}$ is compatible with the assumption. So, how can we transform this clause into a new one testing x, y and v_2? We see that pattern $\{\} \,\&\, \neg\{\texttt{Nil}\}$ does not match on the subvalues x and y of v_1, but for v_2 we still have to match against the pattern $\overline{D}_2^2$. We keep the default clause by rule 1 unchanged.

$$\mathit{subproblem} = \textbf{case } x, y, v_2 \textbf{ of } \begin{bmatrix} _ & _ & \overline{D}_2^2 & \Rightarrow e^2 \\ \textbf{default} & & & \Rightarrow e_d \end{bmatrix}$$

6.2.2 Computing Default

Next we show how to compute $\mathcal{D}(i, \{\mathcal{C}^{n_1}, \ldots, \mathcal{C}^{n_z}\}, \text{Input})$ whose output reflects *the assumption that the scrutinee v_i did not match any of the head constructors $\mathcal{C}^{n_1}$ to $\mathcal{C}^{n_z}$*. This time we have no new scrutinees and only match the leftover list against a new pattern matrix $\left[D\right]$:

$$\mathcal{D}(i, \{\mathcal{C}^{n_1}, \ldots, \mathcal{C}^{n_z}\}, \text{Input}) \;=\; \textbf{case } \; v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n \; \textbf{ of } \; \left[D\right]$$

As before, we build a new matrix $\left[D\right]$ by row-wise iteration over the clauses of the Input. A clause only contributes if its i-th pattern entry allows for a match with scrutinee v_i, knowing that it differs from all the head constructors $C^{n_1}, \ldots, C^{n_z}$.

1. Row $\left[\textbf{default} \Rightarrow e_d\right]$ is added to $\left[D\right]$ unchanged.
2. Row $\left[\overline{D}_1^j \;\ldots\; \overline{D}_{i-1}^j \;\; \overline{\mathbf{D}}_\mathbf{i}^\mathbf{j} \;\; \overline{D}_{i+1}^j \;\ldots\; \overline{D}_n^j \;\Rightarrow\; e^j\right]$, only contributes according to $\overline{\mathbf{D}}_\mathbf{i}^\mathbf{j}$:
   a. If $\overline{\mathbf{D}}_\mathbf{i}^\mathbf{j} = \{y_1, \ldots, y_s\} \,\&\, \neg\{\mathcal{C}_1, \ldots, \mathcal{C}_t\}$,
   then add the row $\left[\overline{D}_1^j \ldots \overline{D}_{i-1}^j \overline{D}_{i+1}^j \ldots \; \overline{D}_n^j \;\Rightarrow e^j[\{y_1, \ldots y_s\} \mapsto v_i]\right]$.
   Note that v_i's constructor differs from all head constructors which include $\{\mathcal{C}_1, \ldots, \mathcal{C}_t\}$

b. If $\overline{\mathbf{D}}^{\mathbf{j}}_{\mathbf{i}} = \|\{\overline{K}_1, \ldots, \overline{K}_t\}$, then

recursively iterate the rows of $\begin{bmatrix} \overline{D}^j_1 & \ldots & \overline{D}^j_{i-1} & \overline{\mathbf{K}}_1 & \overline{D}^j_{i+1} & \ldots & \overline{D}^j_n & \Rightarrow & e^j \\ & & & \vdots & & & & & \\ \overline{D}^j_1 & \ldots & \overline{D}^j_{i-1} & \overline{\mathbf{K}}_t & \overline{D}^j_{i+1} & \ldots & \overline{D}^j_n & \Rightarrow & e^j \end{bmatrix}$.

Let us illustrate rule 2a with the case expression below, where we perform a case split on v_1. We want to construct the default *subproblem* which assumes that v_1 did not match the constructors `Red` or `Green`.

$$\mathit{compile}\left(\begin{array}{l}\textbf{case } v_1, v_2 \textbf{ of} \\ \begin{bmatrix} \{\} \,\&\, \texttt{Red} & \overline{D}^1_2 \Rightarrow e^1 \\ \{\} \,\&\, \neg\{\texttt{Green}\} & \overline{D}^2_2 \Rightarrow e^2 \\ \textbf{default} & \Rightarrow e_d \end{bmatrix}\end{array}\right) = \begin{array}{l}\textbf{case } v_1 \textbf{ of} \\ \begin{bmatrix} \texttt{Red} & \Rightarrow & \ldots \\ \texttt{Green} & \Rightarrow & \ldots \\ \textbf{default} & \Rightarrow & \mathit{compile}(\mathit{subproblem}) \end{bmatrix}\end{array}$$

In *subproblem*, we have no new scrutinees and only need to investigate the remaining scrutinee v_2 $\mathit{subproblem} = \textbf{case } v_2 \textbf{ of } \left[D\right]$. We ask for each clause if it is consistent with v_1 not looking like either `Red` or `Green`. The first clause could not match $\{\} \,\&\, \texttt{Red}$, so we discard it. For the second row, however, the test $\{\} \,\&\, \neg\{\texttt{Green}\}$ does match any v_1 that differs from both `Red` and `Green`, so we keep it. The default clause is also kept unchanged.

$$\mathit{subproblem} = \textbf{case } v_2 \textbf{ of } \begin{bmatrix} \overline{D}^2_2 & \Rightarrow e^2 \\ \textbf{default} & \Rightarrow e_d \end{bmatrix}$$

6.3 Compiling an Example

Let us illustrate the compilation process with a variant of the example introduced in Section 3.

$$\begin{array}{lll} \textbf{case } x \textbf{ of}\{ & y \,\&\, (\texttt{Sa} \parallel \texttt{Su}) & \Rightarrow \text{“Today is weekend!”}, \\ & y \,\&\, \neg(\texttt{Fr} \parallel \texttt{Sa} \parallel \texttt{Su}) & \Rightarrow \text{“Today is ”} +\!\!+ \ \mathrm{show}(y), \\ & \textbf{default} & \Rightarrow \text{“Tomorrow is weekend...”} \quad \} \end{array}$$

First, we rewrite this case expression into the multi-column notation, normalize all patterns (see Section 5), and abbreviate the right-hand sides by $e_1(y)$, $e_2(y)$, and d.

$$\mathit{Input} := \textbf{case } x \textbf{ of } \begin{bmatrix} \|\{\{y\} \,\&\, \texttt{Sa}, \{y\} \,\&\, \texttt{Su}\} & \Rightarrow & e_1(y) \\ \|\{\{y\} \,\&\, \neg\{\texttt{Fr}, \texttt{Sa}, \texttt{Su}\}\} & \Rightarrow & e_2(y) \\ \textbf{default} & \Rightarrow & d \end{bmatrix}$$

We branch on the only available scrutinee x by matching it in a single-column case expression against the head constructors $\{\texttt{Fr}, \texttt{Sa}, \texttt{Su}\}$. We then generate a new case expression with four clauses, one for each constructor. All that is left to do is to compute and compile the clause-specific *subproblems* of specialization and default.

$$\mathit{compile}\,(\mathit{Input}) = \textbf{case } x \textbf{ of } \begin{bmatrix} \texttt{Fr} & \Rightarrow \mathit{compile}\,(\mathcal{S}\,(1, \texttt{Fr}, \mathit{Input})) \\ \texttt{Sa} & \Rightarrow \mathit{compile}\,(\mathcal{S}\,(1, \texttt{Sa}, \mathit{Input})) \\ \texttt{Su} & \Rightarrow \mathit{compile}\,(\mathcal{S}\,(1, \texttt{Su}, \mathit{Input})) \\ \textbf{default} & \Rightarrow \mathit{compile}\,(\mathcal{D}\,(1, \{\texttt{Fr}, \texttt{Sa}, \texttt{Su}\}, \mathit{Input})) \end{bmatrix}$$

Let us consider the subproblem $\mathcal{S}\,(1, \texttt{Sa}, \mathit{Input})$. We know that scrutinee x has matched `Sa`, and we have no left-over scrutinees nor new subscrutinees to consider ($n = 0$), giving $\mathcal{S}\,(1, \texttt{Sa}, \mathit{Input}) = \textbf{case} \quad \textbf{of } \left[S\right]$. We get the new clause matrix $\left[S\right]$ by iterating the old one to see how each row contributes.

- For row $\left[\|\{\{y\}\ \&\ \texttt{Sa}, \{y\}\ \&\ \texttt{Su}\} \ \Rightarrow\ e_1(y)\right]$, by rule (c), iterate $\begin{bmatrix}\{y\}\ \&\ \texttt{Sa} & \Rightarrow & e_1(y)\\ \{y\}\ \&\ \texttt{Su} & \Rightarrow & e_1(y)\end{bmatrix}$.
  - For the first one, by (a) we add the row $\left[\quad \Rightarrow\ e_1(x)\right]$.
  - For the second one, no contribution is made.
- For row $\left[\|\{\{y\}\ \&\ \neg\{\texttt{Fr}, \texttt{Sa}, \texttt{Su}\}\quad \Rightarrow\quad e_2(y)\right]$ no contribution is made.
- For row $\left[\textbf{default}\quad \Rightarrow\quad d\right]$, by (1), we keep it unchanged.

This gives the specialization matrix $\left[S\right] = \begin{bmatrix} & \Rightarrow & e_1(x)\\ \textbf{default} & \Rightarrow & d\end{bmatrix}$.

The other specialization subproblems are computed similarly, and we obtain the following intermediate result.

$$\mathit{compile}\,(\mathit{Input}) = \textbf{case}\ x\ \textbf{of}\ \begin{bmatrix} \texttt{Fr} & \Rightarrow \mathit{compile}\left(\textbf{case}\quad \textbf{of}\left[\textbf{default}\quad \Rightarrow\quad d\right]\right), \\ \texttt{Sa} & \Rightarrow \mathit{compile}\left(\textbf{case}\quad \textbf{of}\begin{bmatrix} & \Rightarrow & e_1(x)\\ \textbf{default} & \Rightarrow & d\end{bmatrix}\right) \\ \texttt{Su} & \Rightarrow \mathit{compile}\left(\textbf{case}\quad \textbf{of}\begin{bmatrix} & \Rightarrow & e_1(x)\\ \textbf{default} & \Rightarrow & d\end{bmatrix}\right) \\ \textbf{default} & \Rightarrow \mathit{compile}\left(\textbf{case}\quad \textbf{of}\begin{bmatrix} & \Rightarrow & e_2(x)\\ \textbf{default} & \Rightarrow & d\end{bmatrix}\right) \end{bmatrix}$$

For each of the subproblems we can apply either **Default** or **Simple** and conclude the computation.

$$\mathit{compile}\,(\mathit{Input}) = \textbf{case}\ x\ \textbf{of}\ \begin{bmatrix} \texttt{Fr} & \Rightarrow d \\ \texttt{Sa} & \Rightarrow e_1(x) \\ \texttt{Su} & \Rightarrow e_1(x) \\ \textbf{default} & \Rightarrow e_2(x) \end{bmatrix}$$

Note how the right-hand expression of the previous default clause d is utilized in a constructor-clause, whereas the right-hand expression of the new default-clause originates from a previously non-default clause.

6.4 Correctness of Compilation

Correctness of compilation states that the *compile* function introduced in Section 6 preserves semantics. Because the compilation algorithm operates on multi-column pattern matches, we extend wellformedness of expressions, the operational semantics and denotations to multi-column pattern matches. We can then state the correctness for the compilation algorithm:

▶ **Theorem 27.** *On a wellformed multi-column pattern match of the normalized shape Input, the compilation algorithm preserves semantics:* $[\![\, \mathit{Input} \,]\!] \equiv [\![\, \mathit{compile}(\mathit{Input}) \,]\!]$.

We provide the proof and the extended definitions in the appendix of the extended version of this paper [2].

7 Future Work

There are two main directions that we plan to investigate in future work.

Allowing Overlapping Patterns

In this article we enforce that no two patterns in a pattern match can overlap in order to guarantee confluence of the system, but [7, 8] showed that this restriction can be relaxed. They allow overlapping patterns under the condition that the right-hand sides of the clauses must become judgmentally equal under a substitution computed from the overlapping patterns. Let us look at an example that is allowed in their system, but not ours:

$$\begin{aligned}\mathrm{or}(x) \coloneqq \textbf{case } x \textbf{ of } \{\ &\texttt{Pair}(\texttt{True}, _) \Rightarrow \texttt{True}, && \texttt{Pair}(_, \texttt{True}) \Rightarrow \texttt{True},\\ &\texttt{Pair}(\texttt{False}, x) \Rightarrow x, && \texttt{Pair}(x, \texttt{False}) \Rightarrow x\ \}\end{aligned}$$

In this example, the first and the last (as well as the second and first, second and third, and the last two) patterns are overlapping. But the definition of this function is still confluent, since if we unify the overlapping patterns $\texttt{Pair}(\mathrm{True}, _)$ and $\texttt{Pair}(x, \mathrm{False})$, and apply the substitution to the right-hand sides True and x, the terms become judgmentally equal. Such overlapping patterns can be useful in a proof assistant, since they yield additional judgmental equalities that can be used to simplify a term during normalization. We think that their approach to allow overlapping patterns and our algebraic patterns can complement each other, and we therefore plan to study their interaction.

Exhaustiveness Checking

In this article, we check whether a set of patterns is overlapping, but we do not check whether a set of patterns is exhaustive. This is in spite of the fact that we are compiling these patterns using decision trees, which is a method that can easily accommodate checking the exhaustiveness of sets of patterns [22]. Checking that a given set of patterns is exhaustive is essential for real programming languages, so we plan to develop and verify an algorithm for this purpose, building on the work of [22, 15, 12]. We sketch a basic outline for such an algorithm in the appendix of the extended version of this paper [2].

8 Related Work

Order-Independent Pattern Matching

As discussed in Section 7, [7] and [8] study the issue of order-independent pattern matching in proof assistants. They want to ensure that every clause in a pattern-matching expression can be used as a judgmental equality for normalizing terms during type checking. Their system lacks advanced patterns such as or-patterns and negation patterns, but they instead relax the non-overlap requirement.

Substructural logics

And-patterns and wildcard patterns are strongly related to the rules of contraction and weakening in the sequent calculus. This relationship between structural rules and certain forms of patterns has been observed by [4], who present a pattern matching calculus for the sequent calculus which includes both and-patterns and wildcard-patterns. This relationship can also be observed in programming languages with substructural type systems, such as Rust, where the expression `match v {x@y => (x, y)}` does not type check if the type of v does not support contraction, i.e. does not implement the `Copy` trait.

Logic Programming and Datalog

The problems with the binding structure of patterns involving negations that we discussed in this paper appears in a similar form for Datalog programs. In this article we are only interested in the binding occurrences of variables in a pattern, i.e. those variables which occur under an even number of negations, and which can be used in the right-hand side of a clause. Occurrences of logic variables in a Datalog program can be distinguished into binding occurrences and bound occurrences, where binding occurrences assign a value to a logic variable and bound occurrences can access the value the variable is bound to. The authors of [18] describe a type system which tracks binding and bound occurences of variables and describe the rules for disjunctions and negations in their Section 4.3. Their rule for negations is:

$$\frac{\Gamma_1 \vdash^{\#\cdot -} a\ \texttt{ok} \dashv \Gamma_2}{\Gamma_1 \vdash^{\#} \texttt{not}\ a\ \texttt{ok} \dashv \Gamma_2}\ \text{A-Not}$$

Here # stands for a polarity, either + or −, which controls which variables in the input context Γ_1 are already bound, and the multiplication with − correspondingly switches this interpretation. This rule therefore operates very similarly to our pattern typing rule P-Neg in Section 4.3 and guarantees that negation is involutive with respect to the typing relation.

Dynamic First Class Patterns

We use "dynamic first class patterns" for systems in which patterns can be computed and passed at runtime. Such systems, whose theory was studied by [14] and [13], can also express our algebraic patterns. The technical report [11] describes an extension of the object-oriented language Newspeak (cf. [3]) by such a system of dynamic pattern matching. They explicitly discuss various *pattern combinators*, i.e. combinators which combine patterns to yield new ones. They discuss disjunction, conjunction and negation pattern combinators which correspond to our or-patterns, and-patterns and negation patterns. In this article, we focus exclusively on statically-known patterns, since these can be analyzed and compiled more easily.

Negation Patterns

Negation patterns also appear in [19], where the author mentions that "Having false- and or-patterns allows us to define the complement of a pattern – for any pattern at a type A, we can define another pattern that matches exactly the values of type A the original does not". [19] presents a translation that eliminates negation patterns, but this translation makes a closed-world assumption about all the available constructors for each data type. For example, he translates the negated pair pattern $\neg\texttt{Pair}(p_1, p_2)$ by the pattern $\texttt{Pair}(\neg p_1, _) \parallel \texttt{Pair}(_, \neg p_2)$, whereas we (in Section 5.1) translate it by the pattern $\neg\texttt{Pair}(_, _) \parallel \texttt{Pair}(\neg p_1, _) \parallel \texttt{Pair}(_, \neg p_2)$. He can omit the first disjunct of the or-pattern because he knows that the pair type only has one constructor. If we extend his translation scheme to the list type, then we would translate the negation of the constructor-pattern Nil to the pattern Cons(_, _), since we know that the list type only has two constructors. Because we encode negative information directly in patterns, we never have to rely on type information during our compilation process. Negation patterns have also been introduced under the name of "anti-patterns" and "anti-pattern matching" by [16, 17, 6]. The formalism they present differs from ours in the following points: They only allow constructor-patterns, variable patterns and negation patterns, which means that they cannot give a recursive

translation for patterns involving negations, since that also requires or-patterns. Their formalism is also more rooted in the theory of term rewriting systems instead of programming languages, and they do not present a compilation algorithm to simple patterns as we do.

Pattern matching compilation algorithms

A simple way to execute pattern matching expressions with nested patterns is to try one clause after another, until the first one matches. This, however, is extremely inefficient if we have multiple clauses which are headed by the same outer constructor. In that case, we are performing the same tests multiple times. As an introduction to this problem, we can still recommend the book by [24, Chapter 5]. There are many algorithms that compile nested pattern matches to efficient code, but most of them fall into one of two categories. Nested pattern matching can be either compiled using backtracking automata [1, 20] or by compiling to decision trees [23]. The main difference between these two approaches is a classical time vs. space trade-off. Compiling to decision trees guarantees that every test is executed at most once, but can potentially increase the size of the generated code; backtracking automata do not increase the size of the code, but may perform some tests multiple times. In Section 6 we present a version of the algorithm by [23] which compiles to decision trees. Recently, Cheng and Parreaux [5] also proved the correctness of a pattern matching compilation algorithm for their very expressive pattern matching syntax; they, however, do not care about order-independence in the same way that we do.

9 Conclusion

Pattern matching is an extremely popular declarative programming construct. We argued that if we want to make pattern matching even more declarative, then the order of the clauses in a program should not matter. But if we require that two patterns must not overlap, and if we want to avoid overly verbose pattern matching expressions, then we have to make the language of patterns and pattern matching more expressive. We introduced two complementary features, a boolean algebra of patterns and default clauses, which together solve the verbosity problem. We have provided the operational and static semantics of these constructs, and have shown that they can be compiled to efficient code.

Data-Availability Statement

The theorems in Section 3 have been formalized and checked in the proof assistant Rocq. These proofs are available as related material.

References

1 Lennart Augustsson. Compiling pattern matching. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 368–381, Berlin, Heidelberg, 1985. Springer. doi:10.1007/3-540-15975-4_48.

2 David Binder and Lean Ermantraut. The algebra of patterns (extended version). *arXiv*, 2025.

3 Gilad Bracha. Newspeak programming language draft specification version 0.104, 2022. Technical report.

4 Serenella Cerrito and Delia Kesner. Pattern matching as cut elimination. *Theoretical Computer Science*, 323(1):71–127, 2004. doi:10.1016/j.tcs.2004.03.032.

5 Luyu Cheng and Lionel Parreaux. The ultimate conditional syntax. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024. doi:10.1145/3689746.

6 Horatiu Cirstea, Claude Kirchner, Radu Kopetz, and Pierre-Etienne Moreau. Anti-patterns for rule-based languages. *Journal of Symbolic Computation*, 45(5):523–550, 2010. Symbolic Computation in Software Science. doi:10.1016/j.jsc.2010.01.007.

7 Jesper Cockx. Overlapping and order-independent patterns in type theory. Master's thesis, KU Leuven, 2013. URL: https://lirias.kuleuven.be/retrieve/262314/.

8 Jesper Cockx, Frank Piessens, and Dominique Devriese. Overlapping and order-independent patterns. In Zhong Shao, editor, *Programming Languages and Systems*, pages 87–106, Berlin, Heidelberg, 2014. Springer. doi:10.1007/978-3-642-54833-8_6.

9 Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992. doi:10.1016/0304-3975(92)90014-7.

10 Jacques Garrigue. Programming with polymorphic variants. In *ML workshop*. Baltimore, 1998. URL: https://caml.inria.fr/pub/papers/garrigue-polymorphic_variants-ml98.pdf.

11 Felix Geller, Robert Hirschfeld, and Gilad Bracha. *Pattern Matching for an Object-Oriented and Dynamically Typed Programming Language*. Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam. Universitätsverlag Potsdam, 2010.

12 Sebastian Graf, Simon Peyton Jones, and Ryan G. Scott. Lower your guards: A compositional pattern-match coverage checker. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020. doi:10.1145/3408989.

13 Barry Jay. *Pattern Calculus: Computing with Functions and Structures*. Springer, 2009. doi:10.1007/978-3-540-89185-7.

14 Barry Jay and Delia Kesner. First-class patterns. *Journal of Functional Programming*, 19(2):191–225, 2009. doi:10.1017/S0956796808007144.

15 Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon Peyton Jones. Gadts meet their match: Pattern-matching warnings that account for gadts, guards, and laziness. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 424–436, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2784731.2784748.

16 Claude Kirchner, Radu Kopetz, and Pierre-Etienne Moreau. Anti-pattern matching. In Rocco De Nicola, editor, *Programming Languages and Systems*, pages 110–124, Berlin, Heidelberg, 2007. Springer. doi:10.1007/978-3-540-71316-6_9.

17 Claude Kirchner, Radu Kopetz, and Pierre-Etienne Moreau. Anti-pattern matching modulo. In Carlos Martín-Vide, Friedrich Otto, and Henning Fernau, editors, *Language and Automata Theory and Applications*, pages 275–286, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-88282-4_26.

18 David Klopp, Sebastian Erdweg, and André Pacak. A typed multi-level datalog ir and its compiler framework. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024. doi:10.1145/3689767.

19 Neelakantan R. Krishnaswami. Focusing on pattern matching. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 366–378, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1480881.1480927.

20 Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 26–37, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/507635.507641.

21 Justin Lubin and Sarah E. Chasins. How statically-typed functional programmers write code. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi:10.1145/3485532.

22 Luc Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17(3):387–421, 2007. doi:10.1017/S0956796807006223.

23 Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 35–46, 2008. doi:10.1145/1411304.1411311.

24 Simon Loftus Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International Series in Computer Science. Prentice Hall, 1987.

25 Nick Rioux, Xuejing Huang, Bruno C. d. S. Oliveira, and Steve Zdancewic. A bowtie for a beast: Overloading, eta expansion, and extensible data types in f. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571211.

26 Kevin Scott and Norman Ramsey. When do match-compilation heuristics matter. *University of Virginia, Charlottesville, VA*, 2000. URL: https://www.cs.tufts.edu/~nr/cs257/archive/norman-ramsey/match.pdf.

27 Titus Winters, Tom Manshreck, and Hyrum Wright. *Software Engineering at Google – Lessons Learned from Programming Over Time*. O'Reilly Media, Inc., 2020.

28 Weixin Zhang and Bruno C. d. S. Oliveira. Pattern matching in an open world. *SIGPLAN Not.*, 53(9):134–146, April 2020. doi:10.1145/3393934.3278124.

29 Weixin Zhang, Yaozhu Sun, and Bruno C. D. S. Oliveira. Compositional programming. *ACM Trans. Program. Lang. Syst.*, 43(3), September 2021. doi:10.1145/3460228.

Pydrofoil: Accelerating Sail-Based Instruction Set Simulators

Carl Friedrich Bolz-Tereick
Heinrich-Heine-Universität Düsseldorf, Germany

Luke Panayi
Imperial College London, UK

Ferdia McKeogh
University of St Andrews, UK

Tom Spink
University of St Andrews, UK

Martin Berger
University of Sussex, Brighton, UK
Montanarius Ltd, London, UK

Abstract

We present *Pydrofoil*, a multi-stage compiler that generates instruction set simulators (ISSs) from processor instruction set architectures (ISAs) expressed in the high-level, verification-oriented ISA specification language Sail. Pydrofoil achieves a $> 230\times$ speedup over the C-based ISS generated by Sail on our benchmarks, thanks to the following insights. (i) An ISS is effectively an interpreter loop, and tracing just-in-time (JIT) compilers have proven effective at accelerating those, albeit mostly for dynamically typed languages. (ii) ISS workloads are highly atypical, dominated by intensive bit manipulation operations. Conventional compiler optimisations for general-purpose programming languages have limited impact for speeding up such workloads. We develop suitable domain-specific optimisations. (iii) Neither tracing JIT compilers, nor ahead-of-time (AOT) compilation alone, even with domain-specific optimisations, suffice for the generation of performant ISSs. Pydrofoil therefore implements a hybrid approach, pairing an AOT compiler with a tracing JIT built on the meta-tracing PyPy framework. AOT and JIT use domain-specific optimisations. Our benchmarks demonstrate that combining AOT and JIT compilers provides significantly greater performance gains than using either compiler alone.

2012 ACM Subject Classification Software and its engineering → Retargetable compilers; Software and its engineering → Just-in-time compilers; Software and its engineering → Source code generation; Software and its engineering → Runtime environments; Software and its engineering → Domain specific languages; Hardware → Simulation and emulation; Software and its engineering → Interpreters

Keywords and phrases Instruction set architecture, processor, domain-specific language, just-in-time compilation, meta-tracing

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.3

Acknowledgements We thank Alasdair Armstrong, Max Bernstein, Tobias Grosser, Tim Hutt, Alex Jeffery, Yutaka Nagashima, Amir Naseredini, Zdeněk Přikryl, Alastair Reid, Niyas Sait, Laurie Tratt, Mojtaba Valizadeh and the anonymous reviewers for their helpful comments. We thank Nico Rittinghaus for implementing Tristate numbers in the RPython JIT.

1 Introduction

A processor's functionality is defined by its *instruction set architecture* (ISA), which specifies the machine language interpreted by the hardware. Accurate and comprehensive ISA specifications are essential for processor design, compiler development, and verification.

© Carl Friedrich Bolz-Tereick, Luke Panayi, Ferdia McKeogh, Tom Spink, and Martin Berger;
licensed under Creative Commons License CC-BY 4.0

Instruction set simulators (ISSs), also called functional models, provide software-based representations of processor behaviour. The efficiency of ISS execution directly impacts processor development speed and associated costs. Throughout this paper, unless otherwise noted, ISS refers to full system simulation, encompassing the entire hardware and software stack, not user-mode simulation, which models only the application layer and sends system calls to the host's operating system layer [47, 50].

Traditionally, ISSs were manually implemented, a process that becomes increasingly time-consuming and error-prone as ISA complexity grows. To address this scalability challenge, *architecture definition languages* (ADLs) emerged, allowing for automated ISS generation from formal ISA descriptions [41]. These ADLs aim to streamline the processor development process. Performance overhead associated with generated ISSs is often mitigated through just-in-time (JIT) compilation or partial evaluation. While ISSs are crucial for functional verification, computer architects also require cycle-accurate models and hardware synthesis capabilities. Consequently, many widely used ADLs, such as SystemC [32], extend beyond pure ISA specification to incorporate microarchitectural details, including pipelining information. This integration, while providing comprehensive modelling capabilities, typically uses low-level language constructs, such as C++ classes and macros, which introduce complexity into the specification and verification process.

Sail [6] is an alternative to low-level ADLs and eschews all hardware implementation detail in favour of ISA-only modelling. Sail streamlines ISA specification with high-level language features such as arbitrary-precision arithmetic, pattern matching, and type-based bitwidth-generics. Sail supports formal verification by also compiling to logic, targeting theorem provers like Isabelle/HOL, Lean, and Z3. Notably, Sail has been adopted as the official ISA-specification language for RISC-V [33]. However, the Sail-generated ISS suffers from performance limitations. This leads to our research agenda: *how can a high-level ISA-specification language like Sail be compiled into performant ISSs?* The unique combination of Sail's high-level abstraction and focus on verification challenges the applicability of conventional ISS compilation techniques, including existing JIT-based approaches. Concretely, our paper seeks to answer the following research questions:

- **RQ1.** What are the bottlenecks causing the Sail-generated ISSs to be slow?
- **RQ2.** Can tracing JIT compilation be used to speed up Sail-generated ISSs significantly?
- **RQ3.** How close can we get to the performance of an industrial strength ISS, hand-tuned to RISC-V?

Contributions. To address the research questions, we present *Pydrofoil*, an open-source[1] tracing JIT compiler for Sail, implemented using the RPython/PyPy meta-tracing framework[2]. We developed and evaluated a suite of domain-specific optimisations tailored to the unique characteristics of ISS workloads, which differ from the typical Python applications targeted by PyPy. Our evaluation methodology employs established computer architecture benchmarks, primarily Linux-boot and SPEC workloads, alongside detailed ablation studies to quantify the impact of individual optimisations. In evaluations using the RISC-V Sail model, Pydrofoil exhibited a >230× speedup over the Sail-generated ISS, while exhibiting a 26.7× slowdown compared to handwritten and optimised QEMU. This achievement shows that whilst we make significant improvements over the Sail interpreter, there is still performance to be gained. We discuss methods of achieving this in Section 5.

[1] `https://github.com/pydrofoil/pydrofoil/`
[2] `https://pypy.org/`

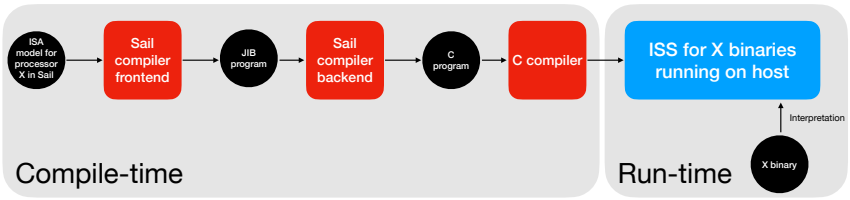


Figure 1 Sail's multi-stage compilation pipeline from Sail to an ISS running as a static binary.

2 Sail and the RISC-V Sail model

2.1 Sail

Sail is a domain-specific language (DSL) within the ML family [27, 40], designed to facilitate processor specification and verification. Sail hybridises first class support for stateful programming with a rich typing system, including algebraic-, singleton-, dependent- and higher-kinded types, as well as domain-specific arbitrary-precision and bitvector types, and a register construct. The compilation pipeline of Sail is depicted in Figure 1. Beyond its primary objective of enabling processor specification and verification through theorem proving, Sail's most noteworthy changes from other ML-family languages are:

- Sail restricts function definitions to first-order, eliminating higher-order functions.
- Sail supports *liquid types* [56, 80].

Liquid types, a form of dependent types, strike a balance between expressive power and decidable type inference. They are useful for addressing the bitwidth parameterisation inherent in modern processor architectures. For instance, the RISC-V ISA supports both 32-bit (RV32) and 64-bit (RV64) variants, where the majority of the ISA specification remains parametric with respect to the bitwidth, represented by a type declaration `xlen`. Traditional ADLs often lack strong typing system support for bitwidth-generic data structures, instead resorting to compile-time meta-programming techniques such as the C preprocessor or C++ templates. Meta-programming simplifies compiler optimisations, but necessitates redundant verification across different bitwidth configurations [12]. We will argue in Section 5 that the current Sail compiler may not fully exploit the optimisation potential afforded by liquid type usage within the RISC-V specification.

This paper primarily focuses on domain-specific compiler design rather than programming language ergonomics. Consequently, a detailed exposition of Sail's language features is beyond the scope of this work. However, it is noteworthy that Sail's capacity to seamlessly support both simulation and logic-based verification contributed to its adoption as the official specification language for RISC-V [33].

2.2 The RISC-V instruction set architecture

RISC-V [8, 75] is an open-source ISA designed to be modular. Unlike proprietary ISAs such as x86, Arm, and MIPS, RISC-V is vendor-neutral. This open nature facilitates academic research and industrial adoption. RISC-V "*is structured as a small base ISA with a variety of optional extensions. The base ISA is simple. [...] The optional extensions form a more powerful ISA for general purpose and high-performance computing.*" [75]. This modularity, combined with the availability of CHERI-RISC-V [78] and its 32-bit and 64-bit variants, results in a significant number of RISC-V ISA configurations, rendering manual ISS

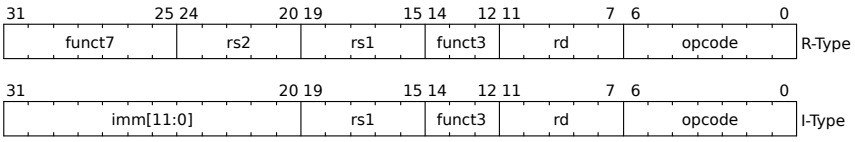


Figure 2 Two of the RISC-V instruction formats [76].

development impractical. Furthermore, the existence of RV32 and RV64 suggests the use of bitwidth-generic data structures, particularly bitvectors. While bitwidth-generics offer flexibility, they introduce compilation complexities and potential performance overhead in ISS implementations, as discussed in subsequent sections.

2.3 The RISC-V model in Sail

We shall now look at a concrete ISA specification in Sail, in order to illustrate how Sail supports the definition of ISA families, using the official RISC-V formal specification [34] as running example.

Abstractly, the toplevel function of the RISC-V Sail model is a big loop, called *fetch-decode-execute loop* in computer architecture. In each round, the loop fetches the next instruction from the external main memory (which an ISS simulates) as a bit pattern, then decodes that pattern into a concrete instruction, and finally calls an `execute` function that executes the decoded instruction. As a simplification, the following pseudo-code serves as a useful mental model:

```
initialise_processor()
pc = 0
while true:
   cmd = memory[pc]
   pc += 4
   match decode(cmd):
      case addi(rd, rs1, imm):
         execute_addi(rd, rs1, imm)
      case xori(rd, rs1, imm):
         execute_xori(rd, rs1, imm)
      ...
      case _:
         ... # Illegal instruction
```

Let us focus our attention on the addition instruction, which, in RISC-V, has the mnemonic `addi rd, rs1, imm`. This instruction adds the 12-bit integer `imm` to the content of a source register `rs1`. The result is stored in the destination register `rd`. Like most ISAs, RISC-V uses instruction formats, which can be seen as rough classifications of instructions that aid decoding bit patterns into instructions. Figure 2 shows the structure of two RISC-V instruction formats. Our example `addi rd, rs1, imm` is an instance of the `ITYPE` format. Like all `ITYPE` instructions,

- `ITYPE` has opcode `0b0010011`, found in bits 0-6.
- The destination register of the instruction is identified in bits 7-11, while the source register is given by bits 15-19.
- The actual function to be computed is in the `funct3` field (bits 12 - 14): `addi` uses `0b000` here, while, e.g., `0b100` identifies `xori`.
- The 12-bit constant to be added to the source register is in bits 20-31.

Let us look at a simplified form of the Sail implementation of `addi`.[3]

```
1  scattered union ast
2
3  val execute : ast -> Retired
4  scattered function execute
5
6  val encdec : ast <-> bits(32)
7  scattered mapping encdec
8
9  union clause ast = ILLEGAL : word
```

First some type declarations: `ast`, short for abstract syntax tree, is the disjoint union type, representing all decoded instructions. In Sail, `ast` is a *scattered* union, which means that the definitions of the variants are allowed to be in different files. This is useful when processor ISAs evolve: we can add new instruction formats in new files *without* needing to modify existing code. The function `execute` (also scattered) specifies the actual semantics of RISC-V instructions, meaning the changes to processor state induced by executing `addi`. The `execute` function takes an `ast` as argument and returns `Retired`, which is an `enum` specifying whether the instruction successfully executed or not (execution can fail, for example due to interrupts). The RISC-V model is imperative: `execute` can and does modify processor state, in particular registers, when invoked. Such processor state is global.

To translate between the encoding of instructions in bits, and their more structured representation as `ast`, the RISC-V model uses the (scattered) mapping `encdec`, short for "encoding-decoding". The keyword `mapping` is for *bidirectional* function definitions [25]. This means `encdec` can either take an `ast` and encode it into a 32-bit bitvector, or take a bitvector and decode it into an `ast`. This reflects, and the typing system enforces, the (almost) one-to-one relationship between ASTs and bitvectors – almost, because not all 32-bit patterns correspond to valid instructions. The model has a fall-through clause `union clause ast = ILLEGAL : word`.

We'll now look at the clauses of `ast`, `encdec` and `execute` for the `ITYPE` format.

```
1  enum iop = {RISCV_ADDI, RISCV_SLTI, RISCV_SLTIU, RISCV_XORI, RISCV_ORI, RISCV_ANDI}
2  ...
3  union clause ast = ITYPE : (bits(12), regidx, regidx, iop)
```

Line 1 enumerates all `ITYPE` instructions and names this type `iop`. Line 3, containing `union clause ast = ITYPE : ...`, defines the variant of the `ast` type that holds `ITYPE` instructions. This clause carries a tuple of information: `(bits(12), regidx, regidx, iop)` The first component holds 12 bits using the type `bits(12)`. This is the constant to be added to the source register. The next two components identify the source and target registers, where `regidx` abbreviates the type `bits(5)`, the type of bitvectors of length 5. (RISC-V has 32 general purpose registers, so 5 bits suffice.) We also need to translate between the 3-bit opcode field in the `ITYPE` instruction format and the enumeration `iop`. This is done by the following bidirectional `mapping`:

```
1  mapping encdec_iop : iop <-> bits(3) = {
2    RISCV_ADDI  <-> 0b000,
3    RISCV_SLTI  <-> 0b010,
4    RISCV_SLTIU <-> 0b011,
5    RISCV_ANDI  <-> 0b111,
6    RISCV_ORI   <-> 0b110,
7    RISCV_XORI  <-> 0b100
8  }
9
10 mapping clause encdec = ITYPE(imm, rs1, rd, op)
11   <-> imm @ rs1 @ encdec_iop(op) @ rd @ 0b0010011
```

[3] See `https://github.com/riscv/sail-riscv/blob/2dfc4ff9f2bed3dcd0a3e8748211c99099e70ab7/model/riscv_insts_base.sail#L159` [55] for full code.

The actual semantics of the `ITYPE` instructions is given by the following `execute` clause:

```
1  function clause execute (ITYPE (imm, rs1, rd, op)) = {
2    let rs1_val = X(rs1);
3    let immext : bits(64) = sign_extend(imm);
4    let result : bits(64) = match op {
5      RISCV_ADDI  => rs1_val + immext,
6      RISCV_SLTI  => zero_extend(bool_to_bits(rs1_val <_s immext)),
7      RISCV_SLTIU => zero_extend(bool_to_bits(rs1_val <_u immext)),
8      RISCV_ANDI  => rs1_val & immext,
9      RISCV_ORI   => rs1_val | immext,
10     RISCV_XORI  => rs1_val ^ immext
11   };
12   X(rd) = result;
13   RETIRE_SUCCESS
14 }
```

The code is beautifully simple and clear. However, the simplicity hides the following complexities:

- The function definition clause in Line 1 pattern matches on the `ast` data type, triggers when the instruction is `ITYPE`, and gives convenient, locally-scoped names to the components `imm`, `rs1`, `rd` and `op`, which were chosen to match the corresponding names in [76].
- Line 2 loads the register value to be added from `rs1`, the source register.
- Line 3 sign-extends the 12 bits value `imm` to the target register bitwidth (32 or 64 bits, depending on the version of RISC-V used). Two things are noteworthy about `sign_extend(imm)` and its use here. First, `sign_extend(imm)` is parameterised with source and target bitwidths, albeit subject to the requirement that the latter not be smaller than the former. This bitwidth parameterisation is visible in the function's type
  $$\texttt{forall 'n 'm, 'm} \geq \texttt{'n. (implicit('m), bits('n))} \rightarrow \texttt{bits('m)}$$
  which has type parameters `'n` and `'m` that range over integers (singleton types). The predicate $\texttt{'m} \geq \texttt{'n}$ is a refinement predicate, evaluated during type checking. We refer to the Sail manual [5] for a detailed explanation of Sail's type syntax. Second, while `sign_extend(imm)` takes two arguments, in Line 3 it is called with only one: the required target bitwidth `'m` is not specified in the call. The keyword `implicit` gives it away: Sail has a restricted form of implicit function parameterisation [37, 43], albeit resolved at compile-time. The resolution of implicit arguments is restricted to the expected output type, here `bits(64)`.
- Line 4 pattern matches on the symbolic name of the exact function to be executed.
- Line 5 triggers if we match `RISCV_ADDI`. In this case, the integer `imm` encoded in the instruction and suitably sign-extended, is added to the content in the source register by $\texttt{rs1_val} + \texttt{immext}$.
- Line 12 writes the result into the destination register `rd`.
- Line 13 returns an indication that the instruction has been successfully executed to the caller.
- Another subtlety is that the register access function $X(\dots)$ is overloaded, and reads from a register when on the right, but writes when on the left of an equality sign.

For this code, Sail offers strong *compile*-time guarantees, for example, we can neither assign bitvectors of different bitwidths, nor access a vector out-of-bounds.

2.4 How does the Sail compiler generate an ISS?

Sail generates a C-based ISS. Unfortunately, this ISS is slow: on a 2024 laptop (AMD Ryzen 7 PRO 7840U, with 32 GB main memory) it executes less than a million RISC-V instructions per second; QEMU simulates a few hundred million instructions in the same time. In this section we identify the root-causes of Sail's poor performance. We start with two possibly unsurprising remarks to set the scene.

```
1  uint64_t imm = ...;
2  ...
3  // let rs1_val = X(rs1);
4  uint64_t rs1_val;
5  rs1_val = rX_bits(rs1);
6  // let immext : xlenbits = sign_extend(imm);
7  uint64_t immext;
8  {
9    i_generic i_generic_64;
10   CREATE(i_generic)(&i_generic_64);
11   CONVERT_OF(i_generic, int64_t)(&i_generic_64, INT64_C(64));
12   bv_generic imm_bv_generic;
13   CREATE(bv_generic)(&imm_bv_generic);
14   CONVERT_OF(bv_generic, fbits)(&imm_bv_generic, imm, UINT64_C(12) , true);
15   bv_generic immext_bv_generic;
16   CREATE(bv_generic)(&immext_bv_generic);
17   sign_extend(&immext_bv_generic, i_generic_64, imm_bv_generic);
18   immext = CONVERT_OF(fbits, bv_generic)(immext_bv_generic, true);
19   KILL(bv_generic)(&immext_bv_generic);
20   KILL(bv_generic)(&imm_bv_generic);
21   KILL(i_generic)(&i_generic_64);
22 }
23 uint64_t result;
24 {
25   __label__ case_8084, finish_match_8083;
26   {
27     if ((RISCV_ADDI != op)) goto case_8084;
28     // rs1_val + immext
29     result = rs1_val + immext;
30     goto finish_match_8083;
31   }
32 case_8084: ;
33   ... // the other cases
34 }
35 {
36   wX_bits(rd, result);
37 }
38 cbz31056 = RETIRE_SUCCESS;
39   goto finish_match_8062;
```

Figure 3 Fragment of the `execute` clause of `ITYPE` instructions after compilation to C by Sail.

- Sail is a high-level language that emphasises precise expression of ISA semantics over concern for simulation speed at every step. High-level languages are harder to generate fast code from than low-level languages.
- Development of optimising compilers is labour intensive and there is a direct correlation between the level of optimisation achievable and developer hours spent. The Sail community chose to spend the scarce developer resources available in a university lab on verification-related parts of the toolchain.

Let us look at concrete code that shows where Sail leaves performance on the table. Figure 3 shows C generated from the `execute` clause of `ITYPE` instructions, discussed above. The code is lightly edited for readability, with Sail code added as comments before the corresponding C code lines. The computation in Line 29 of the result in the `ADDI` case is mapped to an addition on the host machine. Sign extending `imm`, the 12-bit immediate, is done by calling `sign_extend` in Line 17. The arguments of this call are of type `bv_generic`, which is a *heap*-allocated data structure storing bitvectors of arbitrary width. Converting `imm` to `bv_generic`, requires a heap allocation as part of the `CONVERT_OF` macro; `i_generic` is also heap allocated and represents integers of arbitrary size.[4] In Sail, both `bv_generic`

[4] Those types are called `lbits` and `sail_int` in the *actual* C code that Sail generates. We change their names for consistency with Section 3, where we discuss the datatypes in more detail.

and `i_generic` are implemented using the *GNU Multiple Precision Arithmetic library (GMP)* [69] for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers. We summarise the key reasons why the Sail-generated simulator is slow:

- **Bitvectors.** Sail tries to infer the width of the bitvector variables in the program. If they are statically known to fit into an unsigned 64-bit integer, the C backend will map them to the `uint64_t` type. However, many functions in a Sail spec are parametrised with bitwidths, for example `sign_extend`. The bitvectors in these functions are implemented using `bv_generic` and thus ultimately become GMP integers. This has a number of drawbacks: all operations on GMP-based bitvectors require costly heap allocations, because GMP integer types are always heap-allocated.
- **Integers.** Sail's integer type has arbitrary precision. As with bitvectors, Sail tries to infer whether a given integer variables can only have values that fit into a signed 64-bit integer. When that is possible, the C backend will use `int64_t`. However, where it is impossible for Sail to prove that the values fits into `int64_t`, the C backend uses an arbitrary-precision representation from GMP which requires heap allocations on every integer operation.
- **Interpretation overhead.** Finally, Sail generates an interpreter, which has to re-analyse the bits of the program being simulated again and again. Note that actual processors use caches to ameliorate this overhead, e.g., with an instruction cache.

This brings us to our key objective: **how can we improve simulation performance?**

2.5 RPython

Pydrofoil is implemented using the PyPy framework [53]. RPython [3] is PyPy's implementation language. RPython (short for Restricted Python) is a statically typable subset of Python2. The language was developed to aid the implementation of high-performance virtual machines for dynamic languages. To this end, RPython contains a generic meta-tracing just-in-time compilation infrastructure [15] that can be retargeted to different guest languages. Meta-tracing works by tracing through the main interpreter loop (written in the host language – here RPython) of the guest language interpreter [9, 23, 24, 42, 64]. The tracing JIT traces many iterations of the main interpreter loop, effectively unrolling it. This means that one trace contains several guest language instructions, which can then be optimised together. The trace is optimised, compiled to machine code, and then available to execute the corresponding guest language program fragment more efficiently. As ISA models, whether written in Sail or not, are essentially interpreters for an ISA, it seems reasonable to expect that the RPython meta-tracing JIT can speed up the Sail interpreter loop, and compile guest machine instructions to host machine instructions. This has been confirmed in the Pydgin project [39], which implements user-mode ISSs for subsets of Arm v6, MIPS and RISC-V by writing interpreters for them manually in RPython.

3 How is Pydrofoil implemented?

Pydrofoil's approach for better ISS performance is based on the following ideas:

- Better static optimisations of Sail code, in particular optimisations for bitvectors and integers, with the goal of minimising heap allocations.
- Leveraging run-time information with the help of the RPython tracing JIT compiler for further optimisations not available statically.

The first one is, at least in principle, also available to Sail's C backend.

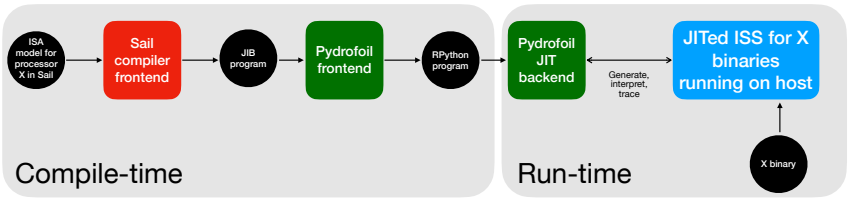


Figure 4 Pydrofoil multi-stage compilation pipeline from Sail to a JITed ISS.

3.1 Pydrofoil architecture

The Sail system emits an ISS in C to be able to use the existing ecosystem of optimising C compilers. In the same way, Pydrofoil reuses the Sail frontend, RPython and the RPython JIT to generate a faster ISS from a Sail model. Pydrofoil's overall architecture is shown in Figure 4.

Pydrofoil's input is JIB, the Sail compiler's textual intermediate representation [7]. When JIB is emitted, all of Sail's type checking has been done, all variables have explicit types and all pattern matching is compiled away into basic blocks and (conditional) gotos. This makes JIB easy for the Pydrofoil ahead-of-time (AOT) compiler to work with. From JIB, Pydrofoil generates its own SSA-based intermediate representation (IR) for Sail functions. These functions are optimised by Pydrofoil's static optimisation passes. Then Pydrofoil generates RPython source code from the optimised IR. The generated code makes use of Pydrofoil's runtime library of support functions that implement Sail datatypes, including problem cases like the aforementioned bitvectors, and arbitrary precision integers. The support library also implements the simulated main memory that the simulated processor interacts with. This resulting RPython code is saved to file, and can be executed with a normal Python2 implementation. This is not performant, but useful for testing. Pydrofoil further translates it into a binary with the existing RPython translation toolchain. During this translation, a tracing JIT compiler can be (optionally) inserted into the resulting binary. The following sections will elaborate on key steps of this process.

3.2 Static Pydrofoil IR optimisations

Once JIB is translated to Pydrofoil's SSA-based IR, we apply a number of standard compiler optimisations: constant folding, dead code elimination, common subexpression elimination, inlining, and scalar replacement of aggregates (using a simple form of partial escape analysis) [14, 62]. All static optimisations are applied until a fixpoint is reached, and further optimisation passes would not alter the IR.

3.2.1 Static Optimisations of bitvector and integer operations

We focus our attention on static optimisations for operations on the bitvector and integer data types, because of their importance for ISSs in general, and Sail in particular. JIB uses two different types for each of these:[5]

[5] In line with the previous section, we renamed two JIB types for consistency: `%bv_generic` is called `%bv` in JIB, whereas `%i_generic` is just called `%i`.

```
1  fn execute(mergez3var) {
2  ...
3    imm : %bv12
4    imm = ...
5    // let rs1_val = X(rs1);
6    rs1_val : %bv64
7    rs1_val = rX_bits(rs1)
8    // let immext : xlenbits = sign_extend(imm);
9    immext : %bv64
10   sail_int_64 : %i_generic
11   sail_int_64 = i64_to_i_generic(64) // cast
12   imm_bv_generic : %bv_generic
13   imm_bv_generic = imm               // cast
14   immext_bv_generic : %bv_generic
15   immext_bv_generic = sign_extend(sail_int_64, imm_bv_generic)
16   immext = immext_bv_generic         // cast
17 ...
```

Figure 5 JIB code for the `execute` clause of `ITYPE` instructions.

- **Bitvectors:** if JIB can infer precise bitwidth, then bitvector types like `%bv64` or `%bv16` of fixed width are used. Otherwise `%bv_generic` is used, a generic bitvector type where the width is not known at runtime (which turns into the `bv_generic`).
- **Integers:** if JIB can infer that an integer can always fit into a 64-bit signed machine integer then `%i64` is used. Otherwise `%i_generic`, a type of arbitrary precision integer, is used.

JIB code frequently casts between the generic and fixed-width variants of the respective representations. Casts do not change the underlying value, but cost a heap allocation when casting to a generic datatype, and heap memory reads when casting in the other direction. Both are expensive. Another reason why the generic forms `%bv_generic` and `%i_generic` are much less efficient at runtime is that the operations (e.g., addition for `%i_generic`) cannot be mapped to directly to host CPU operations, since we do not know their bitwidth. Here is a concrete example. The JIB code in Figure 5, lightly edited for clarity, corresponds to the beginning of `ITYPE`'s `execute` fragment that we already saw in Sail and C in Section 2.3, and Figure 3. Line 8 in Figure 5 shows the `sign_extend` from Sail in comments. JIB Lines 9–16, execute this `sign_extend`: first the immediate value of type `%bv12` is cast to a generic bitvector (Line 13), then sign-extended to 64 bits (Line 15), and finally cast back to `%bv64` (Line 16). This sign extension alone by default requires three heap allocations. The reason why these casts are necessary is that the `sign_extend` function takes a `bv_generic` and an `i_generic` and returns a `bv_generic`.[6]

For all built-in Sail functions and operations that deal with bitvectors and integers, Pydrofoil implements a bundle of variants. They all perform the same computation, but take more specific types as arguments. This enables Pydrofoil to avoid some heap allocations, as we discuss next. To optimise such operations on generic bitvectors and integers, Pydrofoil's static optimiser uses peephole rewrite rules that inspect the IR of a function and rewrite the operations on these types into more efficient forms. In the best case rewrites remove casts between generic and specific bitvector types (as well as machine integers and arbitrary integers) and use specialised operations on the types (when they are available in Pydrofoil's runtime library).

[6] Note that the JIB translation of `sign_extend` takes a bitvector and then an integer as input. We saw in Section 2.3 that the order is reversed in the eponymous Sail function. This is an artefact of Sail's resolution of implicit arguments, and irrelevant for compiler optimisations. It's implemented in the Sail prelude of the RISC-V model by having a small one-operation Sail helper function that has the implicit argument, which calls the JIB `sign_extend` operation with arguments reversed.

Here are Pydrofoil's variants of the `sign_extend` function, together with the rewrite rules that the static optimiser uses:

```
1  # functions
2
3  # nothing is known statically
4  sign_extend:             (%bv_generic, %i_generic) -> %bv_generic
5  # the target width is known to fit into an i64
6  sign_extend_g_i:         (%bv_generic, %i64) -> %bv_generic
7  # the target width is a constant
8  sign_extend_g_c<n>:      (%bv_generic) -> %bv<n>
9  # the argument bitvector has a constant width
10 sign_extend_bv_c<n, m>:  (%bv<n>) -> %bv<m>
11
12 # rewrite rules
13 sign_extend(b, i64_to_i_generic(i))          -> sign_extend_g_i(b, i)
14 sign_extend_g_i(b, c)                       -> sign_extend_g_c<c>(b) [if c <= 64]
15 sign_extend_g_c<c>(cast(b:%bv<n>, %bv_generic)) -> sign_extend_bv_c<n, c>(b, n, c)
```

After applying the rewrite rules to the `sign_extend` call in the example code on Line 15 in Figure 5, it can be replaced by a call to `sign_extend_bv_c`$\langle 12, 64\rangle$. The latter takes a `%bv12` as argument, and returns a `%bv64`. This means we get rid of all three heap allocations. The resulting code is equivalent to the following JIB code and does not contain casts:

```
1  fn execute(mergez3var) {
2  ...
3    imm : %bv12
4    imm = ...
5    // let rs1_val = X(rs1);
6    rs1_val : %bv64
7    rs1_val = rX_bits(rs1)
8    // let immext : xlenbits = sign_extend(imm);
9    immext : %bv64
10   immext = sign_extend_bv_c<12, 64>(imm)
11 ...
```

To be able to narrow the generic integer type `%i_generic` to `%i64`, Pydrofoil also performs a range analysis. If the analysis finds an integer variable where the range fits into a signed 64-bit int, Pydrofoil narrows the type to `%i64`. An example is the RISC-V `mulw` instruction, which multiplies two signed 32-bit bitvectors by converting them to `%i_generic` first. The result of the multiplication of these two integers must fit into an `%i64`, so Pydrofoil narrows the integer types and rewrites the multiplication operation accordingly.

3.2.2 Interaction with inlining

Inlining helps to make bitvector and integer operation rewrites more effective. Many Sail functions are small helper functions that operate on generic bitvectors and integers, passed as arguments. After inlining such small functions into their use contexts, the concrete bitvector widths of the argument can often be inferred from the calling context. Then the bitvector operations in the inlined function are specialised to smaller bitwidths.

Consider another `ITYPE` operation as an example: the `slti` case uses the `operator <_s`, to perform the signed comparison. It is defined as follows in Sail:

```
1  val operator <_s  : forall 'n, 'n > 0. (bits('n), bits('n)) -> bool
2  function operator <_s  (x, y) = signed(x) < signed(y)
```

Here `signed` is a Sail function that takes a bitvector of width n, and returns the corresponding signed integer. This code is compiled to the JIB code in the top part of Figure 6. It implements a signed comparison of bitvectors and operates on generic bitvectors. In the concrete context where the function is called, the bitvector widths are mostly known. For example, its use in the `slti` instruction can be seen in the bottom half of Figure 6.

```
1  val (operator <_s): (%bv_generic, %bv_generic) ->  %bool
2
3  fn (operator <_s)(x, y) {
4    // signed(x) < signed(y)
5    z40 : %i_generic
6    z40 = signed(x)
7    z41 : %i_generic
8    z41 = signed(y)
9    return = lt_int(z40, z41)
10   end;
11 }
```

```
1    jump @neq(RISCV_SLTI, z0) goto 280
2    // RISCV_SLTI  => zero_extend(bool_to_bits(rs1_val <_s immext)),
3    z2 : %bv1
4    z6 : %bool
5    z7 : %bv_generic
6    z7 = rs1_val       // cast
7    z8 : %bv_generic
8    z8 = immext        // cast
9    z6 = (operator <_s)(z7, z8)
10   z2 = bool_to_bits(z6)
11   ...
```

Figure 6 JIB code for `operator <_s` (top half) and its use in the JIB code of the `execute` clause of the `slti` instruction (bottom half).

The arguments to `operator <_s` are cast from `%bv64` to `%bv_generic` (Lines 6 and 8). After inlining the call to `operator <_s`, the calls to `signed` in its body can therefore be optimised to a variant that takes a `%bv64` and returns an `%i64`, which then allows the `lt_int` operation to be replaced by a version that operates on `%i64`. This removes *all* heap allocations from the inlined body of `operator <_s` that replaces the call-site.

3.2.3 Function specialisation

In cases where a function is too large[7] to be inlined, Pydrofoil tries to specialise it to the specific bitvector widths and `%i64` types of its arguments at the function's call-sites. The specialiser will go over all the non-inlinable function calls in the program and check whether any of the arguments are of type `%bv_generic` or `%i_generic` and either constant, or the result of casts from more specific types. For such calls, the target function will be copied into a more specific version. The copied function takes the more specific argument types. The body of the copy is then optimised to take the newly available bitvector widths of its arguments into account. Sometimes, as the result of specialisation, the return type of a function can become more specific, enabling the caller to be optimised further.

3.2.4 When do the static bitvector and integer optimisations fail?

Peephole rewrites for bitvector and integer operations work only within single functions. They can only be applied when the arguments of such operations are cast from more specific types in the same function where the operation happens. Inlining and function specialisation help with peephole rewrites being restricted to single functions. But this is sometimes still insufficient for giving peephole rewrites enough context to do their work. In more complicated cases, particularly if the arguments of bitvector operations are read out of a `union` or a

[7] We use an inlining limit of 25 JIB operations in up to 4 basic blocks at the moment.

```
val mem_read : forall 'n, 0 < 'n <= max_mem_access . (AccessType(ext_access_type),
    physaddr, int('n), bool, bool, bool) -> MemoryOpResult(bits(8 * 'n))

enum word_width = {BYTE, HALF, WORD, DOUBLE}

union clause ast = LOAD : (bits(12), regidx, regidx, bool, word_width, bool, bool)

val extend_value : forall 'n, 0 < 'n <= xlen. (bool, bits('n)) -> xlenbits
function extend_value(is_unsigned, value) = if is_unsigned then zero_extend(value)
    else sign_extend(value)

mapping size_bytes : word_width <-> {1, 2, 4, 8} = {
  BYTE   <-> 1,
  HALF   <-> 2,
  WORD   <-> 4,
  DOUBLE <-> 8,
}

function clause execute (LOAD(imm, rs1, rd, is_unsigned, width, aq, rl)) = {
  let offset : xlenbits = sign_extend(imm);
  let width_bytes = size_bytes(width);
  let addr = X(rs1) + offset;
  // simplified, the real code deals with virtual memory and checks for alignment
  match mem_read(Read(Data), addr, width_bytes, aq, rl, false) {
    Ok(result) => { X(rd) = extend_value(is_unsigned, result); RETIRE_SUCCESS },
    Err(e)     => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
  }
}
```

Figure 7 Sail code for the RISC-V instructions that load from main memory into a register.

`struct`, more powerful static analysis would be needed to be able to optimise the operations to more specific variants. An example is the `LOAD` constructor in the RISC-V Sail model, which implements loading data from simulated main memory into a register. The Sail code can be seen in Figure 7.

The result of calling `mem_read` is a `union` where one of the variants stores a `%bv_generic` at the JIB level. The width of the result of the read from simulated main memory is data-dependent on some bits of the encoded instruction. Therefore it cannot be specialised at compile-time to the `width_bytes` parameter. This also means that the bitwidth of the result is unknown to the static optimisations.

Similarly, it is much harder to track concrete bitvector widths when the values pass through a `union` or are stored into a `struct`. It is conceivable to handle such cases with more powerful intra-procedural static analysis, and taking the information the liquid types into account. We leave this as future work, see Section 5.

3.3 RPython code generation

After Sail functions have been optimised, we generate RPython source code from each. This is straightforward, the only complexity being that the IR control flow graphs use basic blocks with (conditional) jumps between them, and RPython only supports structured control flow. Pydrofoil compiles that to the standard construction of a program counter variable and an infinite loop with one condition for every basic block.[8]

[8] This is an old approach [18], see Harel [30] for a thorough historical discussion. There are better approaches to generating block-structured code out of arbitrary control flow graphs [49, 81] and we plan to switch to one of them.

3.4 Adding a JIT compiler with RPython

To gain efficiency beyond what an interpreter-based simulator can offer, we use RPython's meta-tracing JIT compiler. It traces the execution of the ISS through one loop of the simulated guest program and turns the resulting traces into host machine code. In this way, RPython's tracing JIT acts as a trace-based dynamic binary translator for the simulated instruction set architecture.

RPython's meta-tracing JIT needs a few annotations in the outermost execution loop of the Sail model in question. In particular, the tracing JIT needs to know what the core execution loop is, and which register of the simulated machine stores the program counter. The meta-tracing JIT uses the program counter to detect backwards jumps in the guest program in order to identify loops at the guest program level.

The meta-tracing JIT can further optimise the generic bitvector and integer operations that the static optimisations of Section 3.2 did not improve. Since it traces a concrete execution path of guest machine code, all used bitvector widths are observable by the JIT. The JIT can thus specialise the generic bitvector operations at runtime to the observed widths. This yields performance improvements because the JIT can remove some of the remaining heap allocations of bitvectors and integers using escape analysis [14]. However, it is still preferable to statically remove as many generic bitvector operations as possible, as otherwise the JIT has to continually remove them, every time the ISS is executed.

Consider the following JIT trace, generated for a single RISC-V `addi` instruction (the details of this trace are not important):

```
1  ...
2  i106 = getfield_gc_i(p1, field=s4)
3  ...
4  # c.addi s4. 0x8
5  i144 = getarrayitem_gc_i(ConstPtr(
       ptr142), 26)
6  i146 = int_and(i144, 65535)
7  i148 = int_and(i144, 3)
8  i150 = int_eq(i148, 3)
9  guard_false(i150)
10 i152 = int_eq(i146, 1)
11 guard_false(i152)
12 i154 = uint_rshift(i146, 5)
13 i156 = int_and(i154, 1)
14 i158 = uint_rshift(i146, 6)
15 i160 = int_and(i158, 1)
16 i162 = int_lshift(i156, 1)
17 i163 = int_or(i162, i160)
18 i165 = uint_rshift(i146, 7)
19 i167 = int_and(i165, 15)
20 i169 = uint_rshift(i146, 11)
21 i171 = int_and(i169, 3)
22 i173 = int_lshift(i171, 2)
23 i174 = int_or(i173, i163)
24 i176 = int_lshift(i167, 4)
25 i177 = int_or(i176, i174)
26 i178 = int_is_zero(i177)
27 guard_false(i178)
28 i180 = uint_rshift(i146, 13)
29 i181 = int_is_zero(i180)
30 guard_true(i181)
31 i182 = int_is_zero(i148)
32 guard_false(i182)
33 i184 = int_and(i165, 31)
34 i186 = uint_rshift(i146, 12)
35 i188 = uint_rshift(i146, 2)
36 i190 = int_and(i188, 31)
37 i192 = int_lshift(i186, 5)
38 i193 = int_or(i192, i190)
39 i194 = int_is_zero(i193)
40 guard_false(i194)
41 i195 = int_is_zero(i184)
42 guard_false(i195)
43 i197 = int_eq(i148, 1)
44 guard_true(i197)
45 i199 = int_xor(i193, 32)
46 i201 = int_sub(i199, 32)
47 i203 = int_and(i201, 4095)
48 guard_value(i184, 20)
49 i206 = int_xor(i203, 2048)
50 i208 = int_sub(i206, 2048)
51 i209 = int_add(i106, i208)
52 i211 = int_add(i2, 2)
53 i213 = int_add(i0, 2)
54 setfield_gc(p1, i209, field=s4)
55 i215 = int_eq(i213, 10000)
56 guard_false(i215)
57 # next guest instruction
```

The trace does not contain heap allocations, which is good. However, the trace is long, given that it performs only a single guest addition. The problem is the `getarrayitem_gc_i` instruction at the start, which is the instruction fetch from simulated main memory. A large part of the trace (Lines 6-50) is manipulating the result of that memory read to perform the instruction decoding. This happens despite the program counter register being constant for that point of the trace. In the following section we will look at an optimisation for removing those *instruction-decoding* trace instructions.

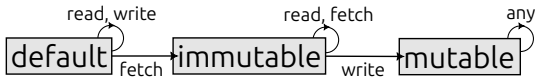


Figure 8 Diagram of the states a 64-bit word of simulated main memory can be in.

3.5 Making the main memory simulation JIT-friendly

One big problem for the JIT compiler is that while it knows the concrete value of the guest program counter, that is not enough for optimising away the decoding of the instruction stored at that program counter.

The reason is the possibility of *self-modifying code*. In practice true self-modifying code is rare, and so we want the JIT to exploit this fact. To do this, we track the status of every part of the simulated main memory. Currently, this tracking happens at the granularity of 8 bytes.[9] Every 64-bit word in simulated main memory can be in one of three states: `status_default`, `status_immutable`, `status_mutable`. All the 64-bit words start out in `status_default` state. If a `status_default` 64-bit word is read or written, its status is unchanged, and the memory read proceeds normally. However, if a `status_default` is read from during instruction fetch, it transitions to `status_immutable`. The JIT will constant-fold reads from `status_immutable` 64-bit words and return the concrete value of the bits stored in the 64-bit word. To behave correctly in the face of self-modifying code, the simulated main memory checks on every write whether a `status_immutable` 64-bit word is being written to.[10] When a `status_immutable` 64-bit word is changed, the generated host machine code is potentially no longer valid, because it was generated under the assumption that `status_immutable` 64-bit words are immutable. Therefore all relevant host machine code is invalidated in such a situation, and the modified 64-bit word is marked as `status_mutable`. `status_mutable` 64-bit words are assumed to potentially contain mutable code, so the JIT will never constant-fold reads from them during host machine code generation. Figure 8 shows a state diagram of the various states every 64-bit word of the simulated main memory can be in, and the transitions it can go through.

After adding this status tracking logic to the simulated main memory, the trace for the `addi` instruction in Section 3.4 gets much simpler. Since the memory read of the instruction bits happens from a `status_immutable` 64-bit word, the bits of the guest instruction are constant. Therefore the trace operations that decode those bits all get constant-folded away. This kind of trace length reduction due to memory status tracking is typical: on average the trace for one guest instruction gets 3–4 times shorter with memory status tracking. The remaining trace looks like this:

```
i35 = getfield_gc_i(p1, field=s4)
...
# c.addi s4, 0x8
i44 = int_add(i35, 8)
i64 = int_add(i0, 2)
setfield_gc(p1, i44, field=s4)
i66 = int_eq(i64, 10000)
guard_false(i66)
```

[9] It is possible that other granularities work better. Pydrofoil could e.g., do this kind of tracking on the granularity of guest pages and take the permission bits from the page tables into account.

[10] This extra check makes all writes to simulated main memory slower, but the trade-off is worth it because instruction fetch is common: there are more instruction fetches than memory writes.

Now the trace looks good! The `addi s4, 0x8` instruction is compiled to a `int_add(..., 8)` instruction, which is its closest equivalent in the RPython trace IR. There are a few extra instructions around to read and write the value of the register field in the `p1` `struct` which contains the simulated machine's state. In addition, there are three instructions that check whether 10,000 instructions have been executed, which would make it time to tick the simulated clock (we discuss the issue of clocks in more details in Section 5).

3.6 Improving the RPython JIT integer optimisations

While developing Pydrofoil, we identified a number of missing trace optimisations for bit manipulation operations in the RPython JIT. This may not be surprising: the RPython JIT was developed for dynamic language implementations where such operations are rare. In contrast, bit manipulation is pervasive in processors, and, as a consequence, in the Pydrofoil traces the JIT needs to optimise. In order to adapt the JIT's optimising passes to such workloads, we have added a number of new integer optimisations. Let us give an example. In processors, individual bits are often used to hold permissions, e.g., whether a specific address range allows non-aligned access. Typically such a permission bit rarely changes, if ever. Nevertheless the permission bit is checked on every access. In hardware such a check is cheap. In software simulation of hardware, checking the bit again and again is expensive. In principle, a JIT can determine during trace-optimisation if a permission bit changes from one instruction to the next. If not, all but the first check can be elided. However JITs targeting high-level languages typically do not track value-constancy *at the bit-level*.

We adapt *Tristate numbers*, used in the Linux kernel by the eBPF verifier to reason about which bits in a variable have constant values [71], to the RPython JIT. An equivalent approach is also used in other compilers such as LLVM, where it is called *Known Bits* [68]. The Tristate number abstraction lets the JIT track values of constant bits of interest in integer and bitvector variables (which map to the same JIT trace IR type) when optimising traces. This enables the removal of permission checks where the JIT can prove that the relevant bits cannot change.

A concrete example for why this is useful are alignment checks. If two load instructions are executed that use the same base register, e.g. like this:

```
ld t0, 0x8(a0)
ld t1, 0x10(a0)
```

Then both loads have to check whether the pointer the value comes from is word-aligned, which means that the lowest bits are zero. The Tristate number analysis lets the JIT conclude that if the first load is aligned, the second one is aligned too, and so the alignment check can be removed from the second load. Here is a sketch[11] of what this looks like in a JIT trace:

```
# ld t0, 0x8(a0) # first load
i1 = getfield_gc_i(p1, field=a0)
i2 = int_add(i1, 8) # add offset to get the actual pointer
i3 = int_and(i2, 7) # alignment check: extract last three bits
i4 = int_is_zero(i3) # check that they are zero
guard_true(i4) # now the JIT knows that i1 and i2 are of the form ???...???000
... # do the actual load from memory

# ld t1, 0x10(a0) # second load
i12 = int_add(i1, 16) # must be of the form ???...???000, because both i1 and 16 are
i13 = int_and(i12, 7) # alignment check for second load: i13 is 0
i14 = int_is_zero(i13) # always 1
guard_true(i14) # can be removed by the JIT's opimizer because i14 is constant 1
... # do the actual load from memory
```

[11] The actual trace is significantly more complicated, due to bounds checks and address translation.

4 Benchmarks

To evaluate the performance of Pydrofoil and quantify the impact of our domain-specific optimisations and JIT compilation, we conducted a comprehensive benchmarking study. Our evaluation, constrained by available resources, aims to be a representative set of industrial ISS workloads, and has four main parts.

- Comparison of Pydrofoil with the Sail-generated ISS, using RISC-V 64.
- Ablations to understand the contribution of each individual domain-specific Pydrofoil optimisation, using RISC-V 64.
- Comparison of Pydrofoil with QEMU and Spike, two handwritten industry-standard ISSs.
- Preliminary benchmarks using Arm v9.4a and CHERIoT ISAs.

We use two classes of benchmark: SPEC CPU and booting Linux (excluding CHERIoT, described below). Both are widely used in computer architecture, and motivated next.

- SPEC, the Standard Performance Evaluation Corporation benchmark [63], is a suite of standardised tests evaluating the performance of computer systems. We focus on the SPEC CPU benchmarks [17], which measure the performance of a system's processor and memory subsystem. We restrict our attention to SPECint, and avoid the floating point arithmetic in SPEC CPU because currently the RISC-V model lacks a Sail implementation of the floating point extensions. Instead, the RISC-V model relies on the C-based Berkeley SoftFloat library [31], and benchmarking floating points would *not* exercise Pydrofoil[12].
- Booting Linux is a critical benchmark due to its prevalence in ISS applications and its challenge for JIT compilers, as the boot process lacks the long-running loops that JITs excel at optimising. Typically, the boot process executes a lot of code that is never seen again, and so this benchmark helps to establish JIT compilation latency.

Each SPEC benchmark was integrated into a complete Linux 6.7.0 boot image, using the initial ramdisk [79] (`initrd`). This approach is necessary because the Sail models do not contain emulated disk devices that could have been used used for storing programs. Unfortunately, due to the limitation of a few hundred MB on `initrd` sizes, we were not able to run `657.xz_s.1`. For the `625.x264` variants we had the same problem but moved input file generation into the Linux image. We mark those SPEC benchmarks with an asterisk (*) when we discuss them in Section 4.3.

Due to the deterministic nature of Sail models, and their lack of access to the host clock, all timing was performed externally on the host system, measuring the simulator's total wall-clock runtime. Therefore, our SPEC benchmark times also include the Linux 6.7.0 boot process until user-mode (about 138 million instructions).

Hardware and software used for benchmarking. All Sail compilation uses the following parameters affecting optimisation: `-dno_cast -O -Oconstant_fold -memo_z3`. Generated C was compiled with `gcc`. For pragmatic reasons, we use three different compute setups (indicating for each benchmark which is used).

- TaiShan server with Kunpeng 920 Arm v8.4 cores, 500 GB main memory, running Ubuntu 18.04.6.
- AMD Ryzen 7 PRO 7840U with 32 GB main memory, running Ubuntu 24.04.6.
- AMD Ryzen 9 7950x3d, with 128GB main memory, running Ubuntu 24.04.1.

[12] A pure Sail implementation of the floating point extensions is in preparation. When it becomes part of the official RISC-V model, it makes sense to revisit our benchmarks.

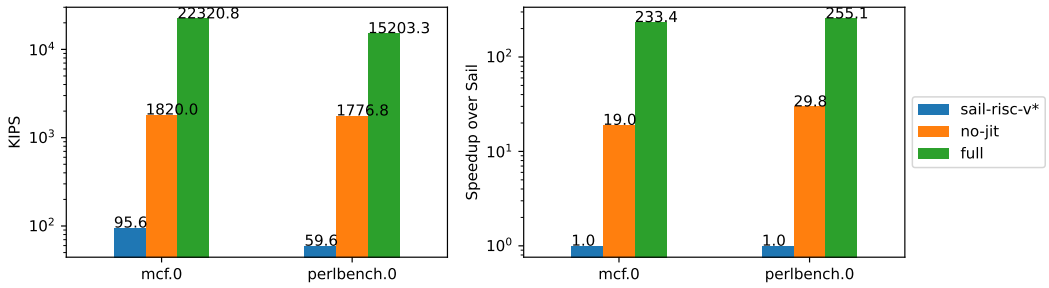


Figure 9 Instructions/second comparison between Sail and Pydrofoil. The left plot shows KIPS (= kilo instructions per second), the right the speedup over Sail. Higher is better.

4.1 Comparing Pydrofoil and Sail performance for the RISC-V model

Pydrofoil's most important goal was generating faster ISSs than Sail itself. This section compares Pydrofoil with Sail. We are resource constrained: the Sail-generated RISC-V ISS is too slow! For example `605.mcf_s.0`, which executes 1.2 trillion instructions, takes roughly six months on the TaiShan server. Therefore, we decided to run the comparison on only two arbitrarily chosen SPEC benchmarks, `605.mcf.0` and `600.perlbench.0`. We stopped the benchmarks for the Sail-generated ISS after 20 billion instructions. Pydrofoil is run to conclusion on the same benchmarks. We compare the instructions per second executed by both systems, extrapolated from the first 20 billion for Sail.

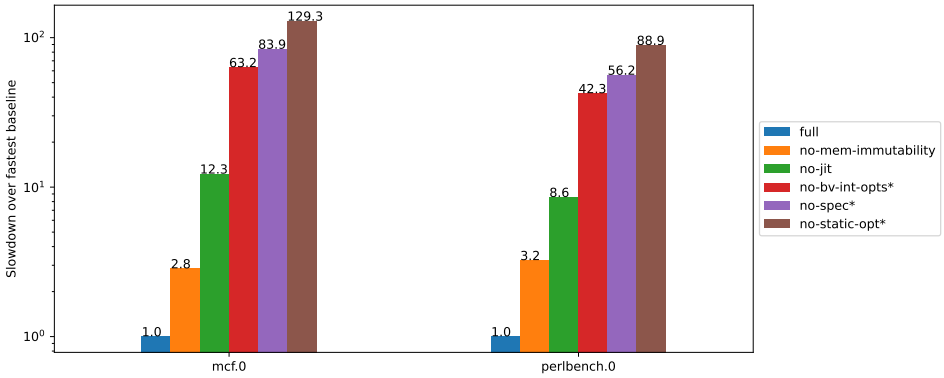
Results. This benchmark uses the TaiShan server. The results of this can be seen in Figure 9. We see that Pydrofoil is 233× and 255× faster than the Sail RISC-V ISS on both benchmarks. Even without the JIT, Pydrofoil is at least 19× faster than Sail.

4.2 Ablations: benchmarking Pydrofoil variants

To quantify the effect of our optimisations, we conduct ablation studies: we systematically disable successively more optimisations and measure the resulting slowdowns. This gives us a quantitative approximation of the relative impact of each optimisation, providing insights into their individual and collective value. We use the following variants.

- `full`: The full Pydrofoil simulator, running the JIT and the AOT with all optimisation enabled.
- `no-mem-immutability`: Disables memory immutability tracking (Section 3.5).
- `no-jit`: Additionally disables the JIT (Section 3.4) and works only as an interpreter at runtime.
- `no-bv-int-opts`: Additionally disables the static optimisations that replace generic bitvector and generic integer operations with more specialise variants operating on 64-bit integers (Section 3.2.1).
- `no-spec`: Additionally disables inlining (Section 3.2.2) and function specialisation (Section 3.2.3).
- `no-static-opt`: Additionally disables all remaining static optimisations (Section 3.2).

Unfortunately, we were unable to run all ablations against the SPEC CPU benchmarks as planned. The problem is that the ablations with many of the optimisations disabled run much too slow to finish in a realistic time frame. For example the aforementioned



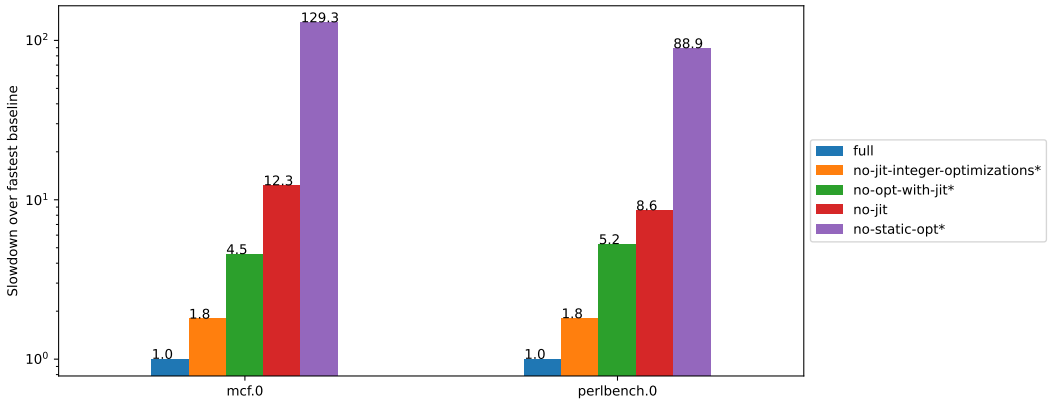
Benchmark Implementation	KIPS mcf.0	KIPS perlbench.0	slowdown mcf.0	slowdown perlbench.0
full	22320.79	15203.28	1.00	1.00
no-mem-immutability	7836.32	4701.09	2.85	3.23
no-jit	1820.04	1776.76	12.26	8.56
no-bv-int-opts*	353.13	359.32	63.21	42.31
no-spec*	266.06	270.47	83.89	56.21
no-static-opt*	172.60	170.92	129.32	88.95

Figure 10 Benchmark results for ablations, as graph and table. The y-axis of the plot is logarithmic. Benchmark results with * are extrapolated from the first 20 billion instructions.

`605.mcf_s.0` takes several months with `no-static-opt`. We deal with this as we did in the previous comparison and run the ablations on the same arbitrarily chosen SPEC benchmarks, `605.mcf.0` and `600.perlbench.0`. We stopped the benchmarks for the slowest ablations after 20 billion instructions. The remaining ablations are run to conclusion on the same benchmarks. We compare the instructions per second executed by both systems, extrapolated from the first 20 billion for the slowest ablations.

Results. This benchmark uses the TaiShan server. The results are shown in Figure 10. We see clearly that the JIT is fundamental for performance, it provides (together with memory immutability tracking) a 12×/8× performance boost. The static bitvector and integer optimisations are the most useful static optimisations. Disabling them slows Pydrofoil down by about a further 5×. Inlining, function specialisation and all other static optimisations are less important. Turning the runtime-switchable integer representation off leads to a further slowdown of more than 2×. We also investigated what is more important: static optimisations or JIT compilation. To that end, we add another ablation, `no-opt-with-jit` It is outside of the sequence of ablations in the previous subsection, because it disables all static optimisations but leaves the JIT-compiler active. The results are in Figure 11. We see that this variant is 4.54×/5.25× slower than `full`. This slowdown is less than `no-jit`, which removes the JIT but keeps the static optimisations. We conclude that the JIT can partly compensate for missing static optimisations (but not vice versa). The JIT does this by performing some, but not all, of the missing static optimisations at runtime.

Unfortunately we were unable to benchmark the additional integer optimisations from Section 3.6: Tristate numbers are now too embedded into the RPython JIT to be easily disabled. To get an upper bound of the effect those optimisations have, we ran an ablation



Benchmark	KIPS		slowdown	
Implementation	mcf.0	perlbench.0	mcf.0	perlbench.0
full	22320.79	15203.28	1.00	1.00
no-jit-integer-optimisations*	12488.84	8454.55	1.79	1.80
no-opt-with-jit*	4912.80	2898.55	4.54	5.25
no-jit	1820.04	1776.76	12.26	8.56
no-static-opt*	172.60	170.92	129.32	88.95

Figure 11 JIT-relevant ablation results. The y-axis of the plot is logarithmic.

`no-jit-integer-optimisations` that disables *all* integer optimisations of the RPython JIT (also in Figure 11). Doing that gives a slowdown of 1.79× and 1.80×. Since the RPython JIT also had integer optimisations before the changes motivated by Pydrofoil (including a range analysis done by the JIT at runtime), we expect the effect of Tristate numbers to be below that number. Therefore it is a less important optimisation compared to any of the Pydrofoil static optimisation and compared to having a JIT in the first place.

4.3 Comparison with QEMU and Spike

To assess the practical performance of Pydrofoil, we benchmark against two established, industrial-strength ISSs. Due to the limited availability of ISS generators capable of compiling Sail specifications, this comparison uses simulators not derived from Sail. We selected QEMU [11] and Spike[13]. QEMU is a widely adopted ISS, implemented in C and incorporating significant manual optimisation. It supports multiple prominent ISAs, including RISC-V, and functions as a core component within various processor simulation frameworks. QEMU utilises a JIT compilation engine that translates guest instructions into host instruction sequences with manually developed, ISA-specific translators. Over two decades of development effort have contributed to QEMU's high simulation performance. While not designed for formal verification, its widespread adoption positions QEMU as a practical performance ceiling for Pydrofoil. Conversely, Spike is manually implemented in C++ specifically for the RISC-V ISA and is widely used within the RISC-V ecosystem. It operates as an interpreter without a JIT compiler, enhancing execution speed through the caching of decoded instructions.

We use SPEC and booting Linux as benchmarks, and RISC-V as ISA.

[13] `https://github.com/riscv-software-src/riscv-isa-sim`

Benchmark	time pydrofoil	time qemu	time spike	speedup qemu	speedup spike
600.perlbench_s.0	25462s	1121s	14075s	22.7×	1.8×
600.perlbench_s.1	49089s	760s	13505s	64.6×	3.6×
600.perlbench_s.2	14026s	692s	14232s	20.3×	1.0×
602.gcc_s.0	31757s	993s	16619s	32.0×	1.9×
602.gcc_s.1	35479s	651s	11263s	54.5×	3.2×
602.gcc_s.2	30758s	627s	10455s	49.1×	2.9×
605.mcf_s.0	13248s	6095s	30219s	2.2×	0.4×
620.omnetpp_s.0	58203s	1267s	33816s	45.9×	1.7×
623.xalancbmk_s.0	14044s	3456s	32109s	4.1×	0.4×
625.x264_s.0*	15664s	278s	7281s	56.3×	2.2×
625.x264_s.1*	92698s	947s	1331s	97.9×	69.7×
625.x264_s.2*	69464s	768s	1953s	90.5×	35.6×
631.deepsjeng_s.0	39604s	1749s	34459s	22.6×	1.1×
641.leela_s.0	34928s	1738s	21686s	20.1×	1.6×
648.exchange2_s.0	17023s	1159s	12661s	14.7×	1.3×
657.xz_s.0	49566s	2603s	47487s	19.0×	1.0×
Geometric mean				26.7×	2.3×

Figure 12 Performance comparison of QEMU, Spike and Pydrofoil on SPEC benchmarks, in seconds. Speedup is the factor by which QEMU, resp. Spike, is faster than Pydrofoil. All benchmark times include booting Linux. Benchmarks marked with * also include the time to generate their input files.

Results. This benchmark uses the AMD Ryzen 9 setup. The results are in Figure 12. We see that QEMU's performance is significantly ahead of Pydrofoil's: QEMU is between 2× and 100× faster than Pydrofoil, with a geometric mean of 26.7× . This is clearly disappointing, it shows that for many cases Pydrofoil is not good enough at removing the overhead of Sail. Section 5 gives concrete suggestions on how to reduce the performance gap with QEMU.

Spike also generally outperforms Pydrofoil. The relative speedup of Spike compared to Pydrofoil ranges from 0.4× (so Pydrofoil is faster) to 96.7×. The geometric mean of the speedup across all benchmarks is 2.3×. Pydrofoil exhibits significantly lower performance relative to Spike on the `625.x264_s.1/2` benchmarks, in particular. The underlying causes of these performance differences, especially the slowdowns observed for Pydrofoil, warrant future investigation.

4.4 Preliminary benchmarks for Pydrofoil-Arm and Pydrofoil-CHERIoT

All benchmarks above use RISC-V. This raises the question how dependent on RISC-V the performance numbers are. This benchmark presents preliminary performance results for two other ISAs, Arm v9.4-a [52] and CHERIoT [1, 2, 19]. For both ISAs, we compared the performance of the Pydrofoil-generated ISS with that of the Sail C backend.

The Arm v9.4-a specification in Sail is obtained by transpilation from Arm's ASL [51] specification, and is *substantially* bigger and more complex than RISC-V. For example the JIB obtained by translation of the Arm model is 20× larger than the corresponding RISC-V. As Arm v9.4-A benchmark, we boot Linux 6.0.7, but only up to the initialisation of the `init` process. This executes 24.6 million instructions.

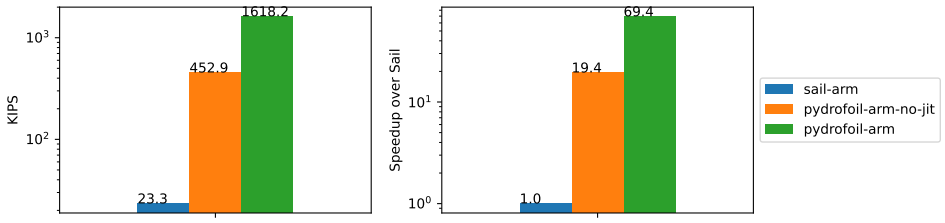


Figure 13 Benchmark results for the Arm ISS, booting Linux. The left plot shows KIPS, the right the speedup over Sail. Higher is better.

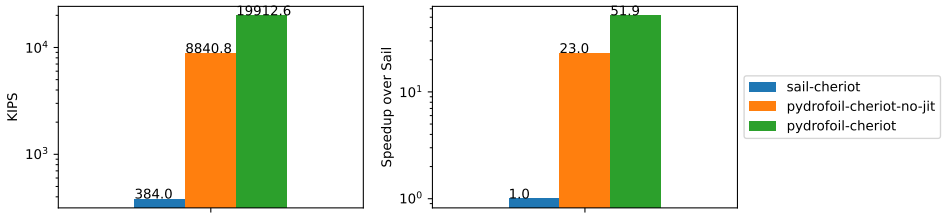


Figure 14 Benchmark results for the CHERIoT ISS, running the allocation-benchmark. The left plot shows KIPS, the right the speedup over Sail. Higher is better.

In contrast to the complex Arm ISA, CHERIoT is a simple 32-bit RISC-V micro-controller without virtual memory and caches, co-designed with a real-time operating system (RTOS), all with a strict focus on security. CHERIoT is a capability machine, based on (a small variant of) CHERI [78]. Capabilities can be seen as a hardware implementation of "fat pointers": conventional integer pointers are replaced by capabilities, which are memory addresses together with permissions constraining capability use. This enables the processor to check every memory access to ensure that it is not out-of-bounds (spatial memory safety). Capabilities can be used to mitigate a wide range of memory safety bugs and vulnerabilities. CHERIoT comes with a Sail specification of its ISA semantics [46], and the ISS that Sail generates is currently the only available ISS for CHERIoT. For CHERIoT we boot its RTOS [45] and then run CHERIoT's "allocation-benchmark"[14]. Booting the RTOS takes 490,000 instructions, while the "allocation-benchmark" takes 274 million.

Results. This benchmark uses the AMD Ryzen 7 setup. The results for Arm are in Figure 13. While Pydrofoil still manages to produce a much faster ISS for Arm than the Sail C backend (by a factor of almost 70×), in terms of instructions per second the performance is significantly worse than that of the RISC-V variant.

The CHERIoT results are in Figure 14. The speedup of Pydrofoil-CHERIoT is about 50×, even lower than that of Arm. This seems to be at least partly caused by Pydrofoil's poor handling of the fact that the CHERIoT-Sail-model uses a Sail `struct` type to store the CHERIoT capability registers (in RISC-V and Arm most registers are bitvectors).

[14] The "allocator-benchmark" can be found in [44]. For each power of 2 from 32 to 131072, the benchmark tries to allocate 1 MB with `malloc`, and then deallocates it using `free`. It also prints (to the emulated character device) the time it took to allocate and deallocate each 1 MB.

4.5 Threats to validity

This study, while providing insights into Pydrofoil's performance, is subject to several threats to validity. The primary limitation stems from the computational intensity of benchmarking ISSs, which operate at significantly lower speeds than the hardware they model. A comprehensive evaluation of Pydrofoil across a broad range of benchmarks requires computational resources and time that exceed our current capacity. Consequently, our findings are based on a subset of representative benchmarks, potentially limiting the scope of our conclusions. Second, the selected benchmarks, while intended to be representative of typical computer architecture workloads, were not specifically designed for ISS evaluation. This misalignment may not fully exercise Pydrofoil's capabilities, potentially under-representing its strengths and weaknesses. We acknowledge the need for benchmarks specifically tailored to evaluate ISS generators. Third, to mitigate limitations in existing tools, we employed certain experimental shortcuts. Notably, we did not perform multiple benchmark runs to assess statistical variation[15]. Finally, our primary focus on the RISC-V ISA introduces a potential limitation to the generalisability of our findings. While we included preliminary evaluations for Arm and CHERIoT, these are subject to methodological constraints. The Arm Sail model, derived from ASL through transpilation, may introduce runtime checks due to ASL's weaker typing, potentially confounding the attribution of performance characteristics. This highlights the challenges of comparing ISSs derived from different specification languages. The CHERIoT evaluation is based on a single, small-scale benchmark, whose results may not generalise well. Nevertheless, we believe that our work demonstrates that ISA specifications in Sail can be compiled into performant ISSs.

4.6 Summary of benchmarking

To address research questions RQ2 and RQ3, we conducted extensive benchmarking, aiming to quantify Pydrofoil's performance. Our ablation studies demonstrate that Pydrofoil's hybrid architecture, combining AOT and JIT compilation with domain-specific optimisations, is a viable approach for accelerating ISSs generated from Sail. Our smaller-scale benchmarks across Arm and CHERIoT ISAs corroborate the trends observed with RISC-V, giving us additional confidence in the robustness of our approach. However, comparisons with QEMU highlight a significant remaining performance gap, indicating substantial room for improvement to match the speed of hand-crafted full-system emulators.

Notably, the considerable resource demands of these benchmarks suggest a need for more efficient and feasible benchmarking strategies.

5 Suggestions for future research

While the present work has demonstrated significant performance gains, achieving a >250× speedup over the Sail-generated ISS, Pydrofoil remains 26.7× slower than QEMU. Spink et al. (2020) [60] report a 2× speedup over QEMU by employing a combination of AOT and partial evaluation techniques, similar in principle to Pydrofoil, albeit compiling from a low-level ADL, which is easier to optimise for than Sail. These discrepancies highlight the potential for further optimisation, and motivate the roadmap presented below, which is intended to bring Pydrofoil to performance parity with QEMU. It might also provide insights for the development of high-performance ISS generators for other high-level ADLs.

[15] Due to the required computational intensity. We hope to fix this in later versions of this work.

Clocks. Simulation of the processor clock is a known inefficiency for ISS acceleration [58]. The RISC-V Sail model ticks its clock every 100 instructions. This number was chosen arbitrarily, and is *not* part of the RISC-V instruction set architecture [76, 77]. Checking whether the next clock tick is reached in every guest instruction is expensive. Moreover, the JIT must generate control flow edges from *all* instruction to the clock logic. QEMU optimises this by only checking for clock ticks at the start of generated basic blocks, and deoptimises whenever a timer event happens elsewhere [48]. It is interesting to investigate if QEMU's approach or the mitigations discussed in [58] improve Pydrofoil performance.

Address translation. Real processors cache the results of address translation in the translation lookaside buffer (TLB), a dedicated hardware cache. Simulation has no TLB, so every memory access must simulate a full page table walk. This is a known problem in ISS research [11, 59]. We believe that a JIT with suitable domain-specific optimisations can lower the cost of simulated page table walks, for example by exploiting the predictability arising from contiguous addresses.

JIT warmup. The time it takes the JIT to generate optimised machine code is called warmup [10], and is the result of a fine balance between the cost and benefits of optimisations. PyPy's tracing mechanism was optimised for general purposed workloads: in PyPy, one Python bytecode produces 6-7 trace operations on average, which the optimiser reduces to about 2. Contrast this with Pydrofoil: tracing one RISC-V guest instruction produces more than 600 trace operations, which are optimised down to about 65 on average. The trade-offs driving the JIT optimisation design in PyPy are likely to be suboptimal for ISS workloads, hence have significant potential for improvement.

Type-erasure. Sail's liquid types can express strong constraints on bitvector widths. Sail's current compilation chain mostly erases them. Consider `size_bytes` from Section 3.2.4 with signature $\texttt{size_bytes} : \texttt{word_width} \leftrightarrow \{1, 2, 4, 8\}$. In other words, any value obtained by calling `size_bytes` is constrained to values $1, 2, 4$ or 8 in the Sail source. This strong constraint is lost in JIB: all the Pydrofoil AOT sees is that `size_bytes` returns `%i64`. If we want to optimise further, the fact that `%i64` stores one of $1, 2, 4, 8$ needs to be re-discovered by static analysis, either at compile-time, or by the JIT at run-time. Such strong constraints are prevalent in ISAs. Given the increasing interest in liquid types in other languages [36, 70], we believe that a better exploitation of the information from liquid types in the compilation toolchain might deliver better optimisations even beyond Sail.

6 Related work

6.1 DSLs used in computer architecture

There are a variety of DSLs that have been created for processor descriptions. Most relevant here are the aforementioned *architecture description languages*. Examples include nML [22], ISDL [28], EXPRESSION [29], SystemC [32], ArchC [54], and GenC [72]. They are often described as being high-level, and are actively used to derive ISSs, especially in commercial tools. Compared to Sail, and with the exception of GenC, they all betray their origins in layers of the processor design stack lower than ISA: they invariably also contain facilities to describe low-level detail, whether pipeline layout, timing constraints, or data-path conflicts. In their context of origin, computer architecture, they are considered higher-level, but only by contrast with RTL languages, such as Verilog and VHDL. Such DSLs are often

based on C/C++, a notable exception being SSL [20], which specifies the ISA semantics of machine instructions in Object-Z (an extension of Z specification language). However, this approach has not gotten much traction. It is only with Arm's ASL [51], Sail's most important predecessor, that a mature, ISA-only DSL emerged.

6.2 ISS acceleration

Evaluating Pydrofoil's performance against existing ISS solutions presents challenges due to the lack of directly comparable systems that utilise Sail. Proprietary tools such as Arm's FastModel [38] and Architecture Envelope Models [4], offer high-performance simulation capabilities. However, their internal architectures, including the extent to which they are automatically generated from Arm's ASL, remain largely undisclosed. Similarly, commercial tools from Synopsys (e.g., ASIP Designer [65], Platform Architect [67], ImperasDV [57,66]), MachineWare [16,26], and Codasip [21] rely on handwritten components or generation from lower-level ADLs like SystemC, nML, or Codal [21]. Many academic ISSs, while valuable, have not seen recent publications or updates to their codebase, making comparison difficult.

Pydgin. Pydgin [39], another meta-tracing and PyPy-based ISS, simulates only user-mode execution. This simplification omits features such as address translation, self-modifying code, and interrupt handling. Furthermore, Pydgin utilises its own Architecture Description Language, which offers a lower level of abstraction compared to Sail. These differences render direct performance comparisons difficult. In addition, due to the lack of recent updates to the Pydgin codebase, Pydgin no longer builds "out of the box". Even after resolving these build issues, we found that the resulting simulator could not correctly load and execute our SPEC benchmarks. For these reasons, we excluded Pydgin from our performance evaluation.

GenSim from the University of Edinburgh. GenSim [60,72–74] provides a retargetable and JIT-accelerated ISS framework. The GenSim toolchain, using the GenC specification language, and Captive dynamic binary translator (DBT) achieve a performance speedup of 2× over QEMU [61]. The performance gains are attributed to: (i) hardware-accelerated guest memory address translation [59]; (ii) partial evaluation [73], a technique related to meta-tracing [13, Chapter 12]; (iii) DBT optimised for ISS workloads [61]; (iv) parallelisation [35]; (v) use of a DSL with performance-focused constructs. We believe most of these techniques are general over the JIT/DBT system and thus applicable to Pydrofoil as an avenue for future work. However, while the hardware-acceleration of guest memory address translation provides substantial performance improvement, the implementation used by Captive is not possible in Pydrofoil. The technique requires arranging host page tables so guest address translations can be performed by the host's memory management unit. This use of hardware translation is faster than the simulation of multiple levels of page table walks in software. Unfortunately, it is only suitable for systems where the address space can be fully managed (Captive hosts the DBT in a bare-metal virtual machine, to gain full and unrestricted access to the host machine's hardware); as a user-mode application Pydrofoil is incapable of that. Additionally, this would harm one of Pydrofoil's features by no longer fully simulating the precise model provided, and instead requiring ahead-of-time knowledge of the guest's memory translation regime and a separate implementation for each host-guest architecture combination. The performance implications of compiling from Sail, compared to lower-level languages, remain a subject of ongoing investigation. A tool was made to translate Sail to GenC, forming a Sail frontend for GenSim, but was unsuccessful. The internal architecture of the GenSim toolchain scales poorly with input model size; the handwritten models are 20,000 lines of

GenC and compile in minutes, but attempts to use Sail translated to GenC resulted in tens of millions of lines and failed to compile even after days. The same issue occurred when developing a tool to translate Sail into Rust; the large volume of code being emitted caused the Rust compiler to use an unsustainable amount of memory. This appears to be a consistent challenge faced when working with high-level, machine-generated Sail specifications.

7 Conclusion

This paper introduces Pydrofoil, a tool designed to enhance the performance of ISSs based on Sail. We identify the primary bottlenecks of the current Sail system in Sections 2 and 3, addressing our first research question (RQ1). We propose a staged compiler architecture that utilises meta-tracing, as detailed in Section 3. Our benchmarks, presented in Section 4, demonstrate that Pydrofoil significantly improves simulation speed compared to Sail. However, in its current iteration, it does not yet reach the performance levels of a hand-tuned system like QEMU, answering RQ2 and RQ3. Future work aimed at closing this gap is outlined in Section 5, serving as a roadmap for achieving performance parity with QEMU.

References

1 Saar Amar, Tony Chen, David Chisnall, Felix Domke, Nathaniel Wesley Filardo, Kunyan Liu, Robert M. Norton, Yucong Tao, Murali Vijayaraghavan, Robert N. M. Watson, and Hongyan Xia. *CHERIoT Architecture specification*. CHERIoT Platform, version 0.6 (draft) edition, January 2025. URL: `https://cheriot.org/cheriot-sail/cheriot-architecture.pdf`.

2 Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. CHERIoT: Complete Memory Safety for Embedded Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, pages 641–653, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3613424.3614266`.

3 Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, pages 53–64, New York, NY, USA, 2007. Association for Computing Machinery. `doi:10.1145/1297081.1297091`.

4 Arm. Arm Architecture FVPs. `https://developer.arm.com/Tools%20and%20Software/Fixed%20Virtual%20Platforms/Arm%20Architecture%20FVPs`. Accessed February 2025.

5 Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, and Shaked Flur. The Sail instruction-set semantics specification language. `https://alasdair.github.io/`. Accessed February 2025.

6 Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2019. Proc. ACM Program. Lang. 3, POPL, Article 71. `doi:10.1145/3290384`.

7 Alasdair Armstrong, Brian Campbell, Ben Simner, Thibaut Perami, and Peter Sewell. Isla. `https://github.com/rems-project/isla`, Accessed Jan. 2025.

8 Krste Asanović and David A. Patterson. Instruction Sets Should Be Free: The Case For RISC-V. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley, August 2014. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html`.

9 Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A Transparent Dynamic Optimization System. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 46, February 2003. `doi:10.1145/349299.349303`.

10 Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual machine warmup blows hot and cold. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), October 2017. doi:10.1145/3133876.

11 Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association.

12 Martin Berger and Laurence Tratt. Program Logics for Homogeneous Generative Run-Time Meta-Programming. *Logical Methods in Computer Science (LMCS)*, 11(1:5), March 2015. doi:10.2168/LMCS-11(1:5)2015.

13 Carl Friedrich Bolz. *Meta-Tracing Just-in-Time Compilation for RPython.* PhD thesis, Heinrich-Heine-Universität Düsseldorf, 2014.

14 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Allocation removal by partial evaluation in a tracing JIT. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '11, pages 43–52, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1929501.1929508.

15 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1565824.1565827.

16 Nils Bosbach, Niko Zurstraßen, Rebecca Pelke, Lukas Jünger, Jan Henrik Weinstock, and Rainer Leupers. Towards High-Performance Virtual Platforms: A Parallelization Strategy for SystemC TLM-2.0 CPU Models. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, DAC '24, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3649329.3658257.

17 James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 41–42, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3185768.3185771.

18 Corrado Böhm and Giuseppe Jacopini. Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules. *Commun. ACM*, 9(5):366–371, May 1966. doi:10.1145/355592.365646.

19 CHERIoT Platform. https://cheriot.org/. Accessed February 2025.

20 C. Cifuentes and S. Sendall. Specifying the Semantics of Machine Instructions. In *Proceedings of the 6th International Workshop on Program Comprehension*, IWPC '98, page 126, USA, 1998. IEEE Computer Society.

21 Codasip. What is CodAL? https://codasip.com/2021/02/26/what-is-codal/. Accessed February 2025.

22 A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *Proceedings the European Design and Test Conference. ED&TC 1995*, pages 503–507, 1995. doi:10.1109/EDTC.1995.470354.

23 Joseph Allen Fisher. *The optimization of horizontal microcode within and beyond basic blocks: an application of processor scheduling with resources.* PhD thesis, New York University, USA, 1979. AAI8010348.

24 Joseph Allen Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. Comput.*, 30(7):478–490, July 1981. doi:10.1109/TC.1981.1675827.

25 John Nathan Foster. *Bidirectional Programming Languages.* PhD thesis, University of Pennsylvania, 2009. URL: https://repository.upenn.edu/handle/20.500.14332/32277.

26 MachineWare GmbH. Virtual Components Modeling Library (vcml). https://github.com/machineware-gmbh/vcml, Accessed February 2025.

27 Michael Gordon, Robin Milner, Lockwood Morris, Malcom Newey, and Christopher Wadsworth. A Metalanguage for Interactive Proof in LCF. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 119–130. ACM, 1978. doi:10.1145/512760.512773.

28 George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: an instruction set description language for retargetability. In *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, pages 299–302, New York, NY, USA, 1997. Association for Computing Machinery. doi:10.1145/266021.266108.

29 A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: a language for architecture exploration through compiler/simulator retargetability. In *Design, Automation and Test in Europe Conference and Exhibition, 1999. Proceedings (Cat. No. PR00078)*, pages 485–490, 1999. doi:10.1109/DATE.1999.761170.

30 David Harel. On folk theorems. *Commun. ACM*, 23(7):379–389, July 1980. doi:10.1145/358886.358892.

31 John Hauser. Berkeley SoftFloat. https://www.jhauser.us/arithmetic/SoftFloat.html. Accessed March 2025.

32 IEEE Standard for Standard SystemC Language Reference Manual, 2023. doi:10.1109/IEEESTD.2023.10246125.

33 RISC-V International. ISA Formal Spec Public Review Public. https://github.com/riscvarchive/ISA_Formal_Spec_Public_Review. Accessed February 2025.

34 RISC-V International. RISC-V Sail Model. https://github.com/riscv/sail-riscv. Accessed February 2025.

35 Stephen Kyle, Igor Böhm, Björn Franke, Hugh Leather, and Nigel Topham. Efficiently parallelizing instruction set simulation of embedded multi-core processors using region-based just-in-time dynamic binary translation. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 21–30, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2248418.2248422.

36 Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. Flux: Liquid Types for Rust. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi:10.1145/3591283.

37 Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. Implicit parameters: dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 108–118, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/325694.325708.

38 Arm Limited. Fast Models: Develop Software without Hardware. https://www.arm.com/products/development-tools/simulation/fast-models. Accessed February 2025.

39 Derek Lockhart, Berkin Ilbeyi, and Christopher Batten. Pydgin: generating fast instruction set simulators from simple architecture descriptions with meta-tracing JIT compilers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267, 2015. doi:10.1109/ISPASS.2015.7095811.

40 Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised Edition)*. MIT Press, 1997.

41 Prabhat Mishra and Nikil Dutt. *Processor Description Languages*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

42 James George Mitchell. *The design and construction of flexible and efficient interactive programming systems*. PhD thesis, Carnegie Mellon University, USA, 1970. AAI7104538.

43 Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. Simplicitly: foundations and applications of implicit function types. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158130.

44 CHERIoT Platform. alloc.cc. https://github.com/CHERIoT-Platform/cheriot-rtos/blob/44c1bdeba1278354ec379e36d87fa33c6dc10b0c/benchmarks/allocation/alloc.cc. Accessed February 2025.

45 CHERIoT Platform. CHERIoT RTOS. https://github.com/CHERIoT-Platform/cheriot-rtos/. Accessed February 2025.

46 CHERIoT Platform. Cheriot specification and sail model. https://github.com/CHERIoT-Platform/cheriot-sail. Accessed February 2025.

47 QEMU. System Emulation. https://www.qemu.org/docs/master/system/index.html. Accessed March 2025.

48 QEMU.org. TCG Instruction Counting. https://www.qemu.org/docs/master/devel/tcg-icount.html. accessed February 2025.

49 Norman Ramsey. Beyond Relooper: recursive translation of unstructured control flow to structured control flow (functional pearl). *Proceedings of the ACM on Programming Languages*, 6(ICFP):1–22, August 2022. doi:10.1145/3547621.

50 Allison Randal. The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers. *ACM Comput. Surv.*, 53(1), February 2020. doi:10.1145/3365199.

51 Alastair Reid. Trustworthy specifications of ARM v8-A and v8-M system level architecture. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pages 161–168, 2016. doi:10.1109/FMCAD.2016.7886675.

52 Cambridge University REMS Team. Github repo sail-arm. https://github.com/rems-project/sail-arm. Accessed February 2025.

53 Armin Rigo and Samuele Pedroni. PyPy's approach to virtual machine construction. In *DLS*, Portland, Oregon, USA, 2006. ACM. doi:10.1145/1176617.1176753.

54 S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. ArchC: a systemC-based architecture description language. In *16th Symposium on Computer Architecture and High Performance Computing*, pages 66–73, 2004. doi:10.1109/SBAC-PAD.2004.8.

55 RISC-V model: riscv_insts_base. https://github.com/riscv/sail-riscv/blob/2dfc4ff9f2bed3dcd0a3e8748211c99099e70ab7/model/riscv_insts_base.sail#L159. Accessed February 2025.

56 Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. *SIGPLAN Not.*, 43(6):159–169, June 2008. doi:10.1145/1379022.1375602.

57 Imperas Software. riscvOVPsim: A Complete, Fully Functional, Configurable RISC-V Simulator. https://github.com/riscv-ovpsim/imperas-riscv-tests/tree/v20230724/riscv-ovpsim. Accessed February 2025.

58 Tom Spink, Harry Wagstaff, and Björn Franke. Efficient asynchronous interrupt handling in a full-system instruction set simulator. In *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*, LCTES 2016, pages 1–10, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2907950.2907953.

59 Tom Spink, Harry Wagstaff, and Björn Franke. Hardware-Accelerated Cross-Architecture Full-System Virtualization. *ACM Trans. Archit. Code Optim.*, 13(4), October 2016. doi:10.1145/2996798.

60 Tom Spink, Harry Wagstaff, and Björn Franke. A Retargetable System-level DBT Hypervisor. *ACM Trans. Comput. Syst.*, 36(4), May 2020. doi:10.1145/3386161.

61 Tom Spink, Harry Wagstaff, Björn Franke, and Nigel Topham. Efficient code generation in a region-based dynamic binary translator. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '14, pages 3–12, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2597809.2597810.

62 Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 165–174, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/2544137.2544157.

63 Standard performance evaluation corporation (SPEC) homepage. https://www.spec.org/spec/. Accessed February 2025.

64 Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic native optimization of interpreters. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, IVME '03, pages 50–57, New York, NY, USA, 2003. Association for Computing Machinery. doi:10.1145/858570.858576.

65 Synopsys. ASIP Designer. https://www.synopsys.com/dw/ipdir.php?ds=asip-designer. Accessed February 2025.

66 Synopsys. ImperasDV: RISC-V Processor Verification Made Easy. https://www.synopsys.com/verification/imperasdv.html. Accessed February 2025.

67 Synopsys. Platform Architect: SoC architecture analysis and optimization for performance and power. https://www.synopsys.com/verification/virtual-prototyping/platform-architect.html. Accessed February 2025.

68 Jubi Taneja, Zhengyang Liu, and John Regehr. Testing static analyses for precision and soundness. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '20, pages 81–93, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3368826.3377927.

69 GMP Development Team. GMP: The GNU Multiple Precision Arithmetic Library. https://gmplib.org/. Accessed February 2025.

70 Niki Vazou. *Liquid Haskell: Haskell as a Theorem Prover*. PhD thesis, UC San Diego, 2016. URL: https://escholarship.org/uc/item/8dm057ws.

71 Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Sound, precise, and fast abstract interpretation with tristate numbers. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '22, pages 254–265. IEEE Press, 2022. doi:10.1109/CGO53902.2022.9741267.

72 Harry Wagstaff. *From High Level Architecture Descriptions to Fast Instruction Set Simulators*. PhD thesis, School of Informatics, University of Edinburgh, 2015.

73 Harry Wagstaff, Miles Gould, Björn Franke, and Nigel Topham. Early partial evaluation in a JIT-compiled, retargetable instruction set simulator generated from a high-level architecture description. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2013. doi:10.1145/2463209.2488760.

74 Harry Wagstaff, Tom Spink, and Björn Franke. Automated ISA branch coverage analysis and test case generation for retargetable instruction set simulators. In *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '14, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2656106.2656113.

75 Andrew Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, EECS Department, University of California, Berkeley, January 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html.

76 Andrew Waterman and Krste Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V International, December 2019. Document Version 20191214-draft.

77 Andrew Waterman and Krste Asanović, editors. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. RISC-V International, December 2021. Document Version 20211203.

78 Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9). Technical Report UCAM-CL-TR-987, University of Cambridge, Computer Laboratory, September 2023. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-987.pdf.

79 Wikipedia contributors. Initial ramdisk — Wikipedia, the free encyclopedia, 2025. [Online; accessed 2-March-2025]. URL: `https://en.wikipedia.org/w/index.php?title=Initial_ramdisk&oldid=1277729657`.

80 Hongwei Xi. Dependent ML: An approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, March 2007. `doi:10.1017/S0956796806006216`.

81 Alon Zakai. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, OOPSLA '11, pages 301–312, New York, NY, USA, October 2011. Association for Computing Machinery. `doi:10.1145/2048147.2048224`.

Ensuring Convergence and Invariants Without Coordination

Dina Borrego
NOVA School of Science and Technology, Caparica, Portugal
Vrije Universiteit Brussel, Belgium

Nuno Preguiça
NOVA School of Science and Technology, Caparica, Portugal

Elisa Gonzalez Boix
Vrije Universiteit Brussel, Belgium

Carla Ferreira
NOVA School of Science and Technology, Caparica, Portugal

Abstract

The CAP theorem demonstrates a trade-off between consistency and availability (and, by extension, latency) in systems where network partitions are unavoidable, such as in cloud computing and local-first software. While adopting weak consistency can preserve availability, it may result in inconsistencies that compromise application correctness. Replicated data types provide a principled, coordination-free approach to guarantee convergence but do not consider application invariants. Existing methods for maintaining invariants in replicated systems either rely on coordination – undermining the benefits of weak consistency – or suffer from limited applicability. This paper introduces the No-Op framework, a generic approach for enforcing consistency without coordination while guaranteeing both convergence *and* invariant preservation. The core idea of the No-Op approach is to resolve conflicts among concurrent operations by prioritising one operation over the other according to programmer-defined conflict resolution policies. This prioritisation transforms the less-preferred operation into a no-side-effect operation, ensuring conflict-free execution. We formalise the model underlying the No-Op framework and introduce a replication protocol built upon it, accompanied by a formal proof of correctness for both the framework and the protocol. Furthermore, we demonstrate the framework's applicability by showcasing the design of widely used replicated data types and the preservation of a wide range of application invariants.

2012 ACM Subject Classification Computing methodologies → Distributed computing methodologies; Software and its engineering → Correctness; Theory of computation → Distributed algorithms; Computer systems organization → Availability

Keywords and phrases distributed systems, conflict resolution, RDTs, invariant preservation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.4

Supplementary Material *Software (ECOOP 2025 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.11.2.1`

Funding This work is partially supported by PRT/BD/154519/2022 awarded by the EUTOPIA European University Alliance, UID/04516/NOVA LINCS with the financial support of FCT.IP and the European Commission through the TaRDIS project (agreement ID 101093006).

Acknowledgements We would like to thank Kevin De Porre for the helpful discussions about VeriFx and the anonymous reviewers for their valuable feedback, which helped improve the paper.

1 Introduction

Replication is a key technique in data management systems that enables high availability, fault tolerance, low latency, and scalability. In cloud storage systems [28, 19, 5, 16], data is typically geo-replicated across multiple geographic locations to ensure low latency for clients.

© Dina Borrego, Nuno Preguiça, Elisa Gonzalez Boix, and Carla Ferreira;
licensed under Creative Commons License CC-BY 4.0

Similarly, local-first software [27] allows clients to maintain a local copy of the data, enabling fast execution and continuous operation, even when disconnected. However, achieving these benefits relies on clients accessing a replica without coordinating with others. As a result, replication creates a trade-off between consistency and latency [1].

Weak consistency [45, 2, 34, 10, 43, 27] sacrifices consistency in favour of low latency and offline availability. In these models, operations are first executed locally on a replica (without coordination), with their effects propagating asynchronously to other replicas. This execution model leads to different execution orders across replicas, requiring a mechanism to guarantee that all replicas eventually converge to the same state. Replicated data types (RDTs) [40, 18, 13, 26, 9] offer a principled, coordination-free approach to ensure that, after the execution of the same set of updates, replicas converge to the same state.

However, state convergence alone does not ensure application correctness, as combining the effects of concurrent operations may violate application invariants and lead to an incorrect state. Although synchronising all operations (i.e., employing strong consistency [11, 23]) can preserve invariants, this approach significantly compromises system performance [4].

Much research [15, 30, 7, 21, 31, 25, 18] has focused on reducing coordination by synchronising only operations that might violate invariants, typically identified through static analysis tools. Non-conflicting operations, in contrast, are executed without requiring coordination. An alternative strategy is to move coordination outside the critical execution path, building on escrow and reservation techniques [36, 8, 7, 33, 38, 41]. By reserving the right to execute specific operations in advance, replicas allow operations to be safely executed without synchronisation, provided the necessary rights are locally available. Otherwise, coordination is required to acquire the rights from other replicas, or the operation must abort.

Finally, recent research has focused on maintaining invariants using coordination-free mechanisms [35, 6]. These mechanisms enable asynchronous execution, automatically resolving invariant violations using automatic resolution policies. These approaches build on the principles of RDTs, extending them from ensuring data convergence to preserving invariants: concurrent operations are executed without coordination while guaranteeing that replicas converge to a state where application invariants remain upheld. However, while some solutions, such as Antidote SQL [35], focus solely on referential integrity, others, like IPA [6], address invariant conflicts but rely on RDTs to achieve convergence.

In this paper, we propose a coordination-free consistency framework that addresses both data convergence and invariant preservation in a unified way. Our approach builds on a simple observation: when concurrent operations occur, their effects can be combined to ensure that replicas converge without violating invariants unless the operations conflict with one another. In cases of conflicting operations, it becomes necessary to prioritise the effects of one operation over the other. Our framework conceptually achieves this by transforming the not-preferred operation into an operation with no side effect (or a No-Op).

To validate our claim, we employ two complementary methods. First, we formally define our No-Op approach and provide proof establishing its correctness. Second, we demonstrate its applicability by showing that this simple model can encode popular RDTs and enforce a wide range of invariants, including referential integrity.

We introduce a replication protocol that adopts this framework to maintain consistency in replicated applications. The protocol was implemented in VeriFx [17], a high-level programming language with built-in automated proof capabilities, enabling validation of its correctness through automated proofs on specific data type implementations. These results corroborate the formal proofs provided, confirming the correctness of the No-Op approach.

To summarise, the main contributions of this paper are:

- The No-Op generic framework for enforcing consistency without coordination, achieving both convergence and invariant preservation in a unified way.
- A replication protocol that integrates the No-Op framework.
- Formal proofs of correctness for the proposed framework and replication protocol.
- The applicability of the No-Op framework to the design of popular RDTs and the maintenance of general application invariants.

The remainder of this paper is organised as follows. Section 2 offers an overview of the No-Op approach, followed by Section 3, which introduces its formal model and a formal proof of its correctness. Section 4 details the No-Op replication protocol and provides the corresponding correctness proofs. Section 5 explores the applicability of the proposed solution. Section 6 discusses limitations of the No-Op approach and possible solutions to avoid these limitations. Then, Section 7 discusses related work. Finally, Section 8 concludes the paper.

2 Overview

This section overviews our approach using an album management system as a running example. This application manages a collection of artists and their albums, supporting operations such as adding, removing, and updating artists, as well as adding and removing albums. This example was previously used in [35] and is similar to an example from [16].

The goal of our No-Op approach is to guarantee state convergence and invariant preservation without requiring coordination. Figure 1 and Figure 2 illustrate a convergence and an invariant conflict that could occur when implementing the album management system. Consider that the application's state is modelled as a set of pairs (artist, country) representing artists and their respective countries, along with a set of elements representing albums. Figure 1 shows a convergence conflict. Initially, both replicas are in an equivalent state, each containing one artist (Sam, UK). Replica 1 updates the artist, resulting in a state with one artist (Sam, USA). Replica 2 removes the artist, leaving its state empty. After propagating both updates, the replicas have applied all operations yet reached divergent states. Figure 2 shows how a referential integrity invariant can be violated. The application must ensure that albums are not associated with non-existent artists. Starting again from an initial state where both replicas contain one artist (Sam, UK), replica 1 adds an album, A1, for Sam, updating its state to include the album. Meanwhile, replica 2 removes the artist, leaving its state empty. After propagating both updates, both replicas converge to a state containing the album but not the artist. Therefore, even though the replicas converge after applying all operations, the referential integrity invariant is violated, as the replicas' state contains an album whose associated artist does not exist.

The core idea of the No-Op approach is to avoid by design *conflicting operations*, i.e., operations that violate state convergence or application invariants. When two conflicting operations are executed concurrently, one is converted into a no-side-effect operation (i.e., a No-Op) based on a predefined conflict resolution policy. These policies vary based on the application's intended semantics, and it is the programmer's responsibility to select the most suitable one for the application. The No-Op approach implements conflict resolution using an auxiliary operation called *block*, which encodes the conflict resolution policy defined by the programmer. A *block* is triggered atomically when a conflicting operation is executed and specifies which operations cannot run concurrently due to conflicts. The system then transforms any operation that conflicts with the *block* into a No-Op.

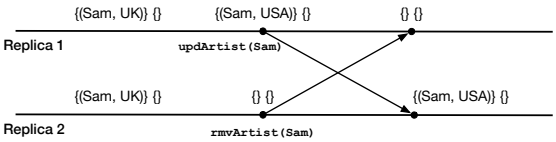


Figure 1 Convergence conflict.

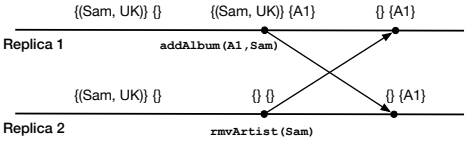


Figure 2 Invariant conflict.

By leveraging conflict resolution policies, the No-Op approach enforces both convergence and invariant preservation in a unified manner. Let us revisit Figure 1 and Figure 2 to see which conflict resolution policies could be applied to solve these conflicts. A conflict resolution policy for solving the convergence conflict in Figure 1 is to prioritise the remove operation by discarding the update operation. This behaviour aligns with the remove-wins policy used in existing remove-wins CRDT sets [40, 9]. Our No-Op approach can encode remove-wins semantics by having remove operations trigger a *block* on updates for the same artist. The *block* on updates will render any concurrent update on the deleted artist a No-Op.

An alternative policy for the convergence conflict is a privilege-based policy suitable for systems where users are assigned privilege levels. Under this policy, both update and remove operations trigger a *block*. An update operation triggers a *block* on all remove operations for the same artist issued by users with lower priority. Similarly, a remove operation triggers a *block* on all update operations for the same artist submitted by users of lower priority. When users share the same privilege level, the policy must specify how to resolve ties. For example, a total order could determine the winning operation among users with equal privilege levels.

Note that an operation only issues a *block* if it can win. For example, under the remove-wins semantics, the update operation never issues a *block* because it never wins against any other operation. In contrast, in a privilege-based policy, both update and remove operations issue a *block*, as either operation can be the winner depending on the user's privilege level.

Similar to convergence conflicts, various conflict resolution policies can be applied to address invariant conflicts. In the case of the referential integrity conflict shown in Figure 2, we can prioritise the operation that adds the album, rendering the remove artist operation ineffective. This behaviour corresponds to an update-wins conflict resolution policy similar to the one used in Antidote SQL [35]. Alternatively, we can prioritise removing the artist by dropping the add album operation, corresponding to remove-wins semantics. These policies are implemented akin to the policies for handling convergence conflicts: the winning operation type issues a *block* on the losing operation type.

2.1 The No-Op Approach by Example

We will now illustrate how the No-Op approach works through a concrete execution trace of the album management system shown in Figure 3.

The execution trace consists of five operations involving two replicas. Replica 1 executes an `addArtist(Sam)` operation, which gets delivered to both replicas. Next, two concurrent `addAlbum` operations are executed and delivered to both replicas. Finally, two concurrent operations follow: `updateArtist` in replica 1 and `rmvArtist` in replica 2.

The No-Op approach stores operations in a graph to determine when a conflict occurs and applies the appropriate conflict resolution policy. This graph forms the core of the No-Op mechanism, representing a partial order of operations that accurately captures concurrency and causality of operations. Figure 3 depicts both replicas' execution graphs for the aforementioned execution trace. Solid circles represent local operations; dashed ones,

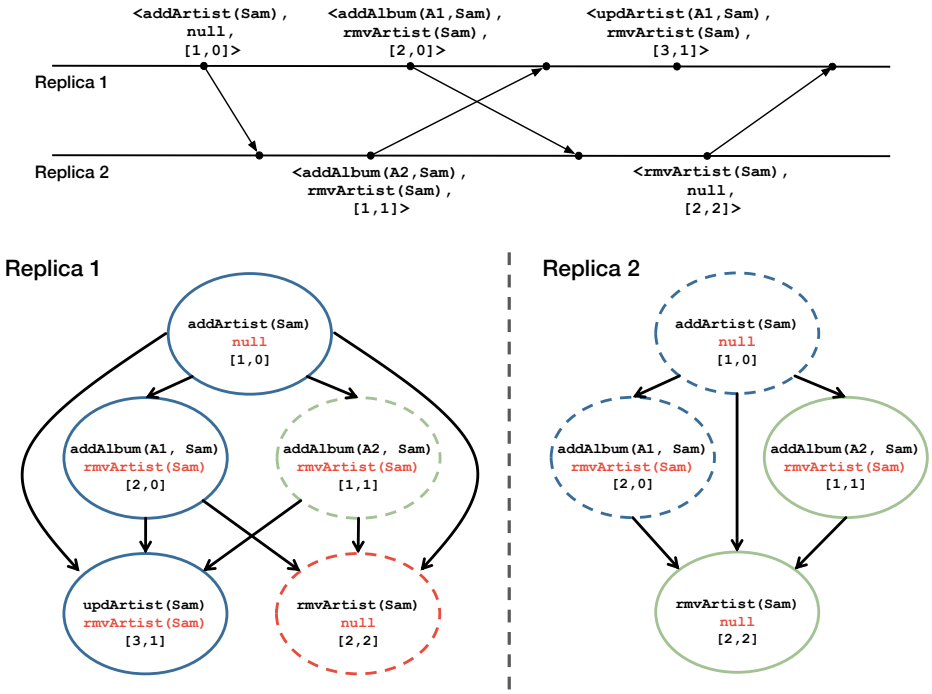


Figure 3 Execution graph for an execution trace of the album management system.

delivered remote operations. Blue, green, and red indicate operations from replica 1, replica 2, and No-Ops, respectively. Each vertex lists the operation call, the *block* operation, and its timestamp. In this example, both referential integrity and convergence conflicts are resolved following the update-wins semantics.

The first node in the operation graph corresponds to the `addArtist(Sam)` operation, which got delivered to both replicas before any subsequent concurrent operations were executed, causally preceding all other operations in the graph. Since the operation was originally executed at replica 1, the node in replica 2 for this operation is dashed. Following this, the two replicas concurrently execute two `addAlbum` operations. Both operations issue a *block* on concurrent `rmvArtist(Sam)` to uphold the referential integrity invariant (according to the update-wins semantics). However, the operations do not conflict, as neither operation blocks the other. Consequently, after propagation, both graphs have two concurrent `addAlbum` operations (one local and one remote), but neither operation is marked as a No-Op.

Lastly, two additional operations are executed concurrently. Replica 1 executes an `updArtist(A1, Sam)` operation, which issues a *block* on concurrent `rmvArtist` on the updated artist, namely `Sam`. Concurrently, replica 2 executes a `rmvArtist(Sam)` operation that does not block any operation. After this operation is propagated to replica 1, the conflict between the *block* on `rmvArtist(Sam)` and the delivered `rmvArtist(Sam)` operation is detected, marking the latter as a No-Op to resolve the conflict. Since the `updArtist` operation has not yet been delivered to replica 2, the graphs currently diverge. Once the `updArtist` operation is delivered to replica 2, the system will also detect the conflict, and the local `rmvArtist(Sam)` operation will be marked as a No-Op.

Note that there are no edges between concurrent operations (e.g., the two concurrent `addAlbum` operations), so there are different possible serialisations of the execution graph. For example, in the graph of replica 2, there are two possible serialisations: i) `addArtist(Sam)`,

`addAlbum(A1, Sam)`, `addAlbum(A2, Sam)`, and `rmvArtist(Sam)`, and ii) `addArtist(Sam)`, `addAlbum(A2, Sam)`, `addAlbum(A1, Sam)`, and `rmvArtist(Sam)`. However, as we prove in Section 3.1, all serialisations are safe and equivalent.

2.2 The No-Op Approach Workflow

We now present the envisioned workflow for developing replicated systems that guarantee state convergence and maintain application invariants using the No-Op approach. This workflow follows a three-step process, which we outline below.

Step 1. Identifying Conflicting Operations. Given the application state and the operations, programmers start by identifying conflicting operations within the system. We assume programmers can leverage existing analysis tools, such as Hamsaz [24] or Ordana [18], to detect conflicts between operations automatically.

Step 2. Defining Conflict Resolution Policies. For each pair of conflicting operations identified by the analysis tool, programmers must define a conflict resolution policy. These policies determine how conflicts are resolved, ensuring consistency while preserving the intended system behaviour. As noted in [37] and illustrated earlier in this section, different concurrency semantics can be applied to solving a conflict, with some being more suitable for specific applications. Our prototype implementation offers a range of conflict resolution policies (e.g., add-wins, remove-wins, last-writer-wins) as shown later in Section 5. These predefined policies enable programmers to adapt them when implementing new data types. Section 4.1.1 shows conflict resolution policy examples.

Step 3. Verification. After defining all conflict resolution policies, we foresee programmers verifying the correctness of the data type or application. In our work, we use VeriFx [17] to validate the correctness of our implementation, though other verification tools could be used. Note that correctness is independent of the conflict resolution semantics, i.e., VeriFx does not validate whether a conflict resolution policy aligns with the programmer's intended semantics. For instance, if a programmer intends to implement an add-wins set but defines a remove-wins policy, the system will enforce the policy as specified without detecting a mismatch. Since we aim to provide flexibility in defining conflict resolution policies, we assume that the programmer correctly specifies policies that align with the application's intended behaviour. To simplify, developers can use or adapt our predefined standard policies covering common concurrency semantics.

3 Model

We now present our model and provide a formal proof of its correctness. The No-Op framework ensures strong convergence and safety, guaranteeing that replicas remain consistent while preserving application invariants without coordination. Strong convergence [39] ensures that correct replicas processing the same set of updates – potentially in different orders – reach equivalent states. Safety ensures that the replicated state consistently upholds the invariants.

We consider a causally consistent database fully replicated across multiple data centres. Replicas may also be stored in mobile devices, which may temporarily be offline. We assume a fail-recover model that preserves durable state and excludes Byzantine faults.

The application state comprises a set of objects, each of which can be accessed and modified at any time through application-defined operations. In turn, the replica state comprises two components: i) the initial application state, and ii) a partial order of operations that tracks concurrency and allows the identification of conflicts as they arise. The current application state of a replica is calculated by applying all operations in partial order to the initial state.

Algorithm 1 Execution model of an update operation under the No-Op approach.

types
$m = \langle op, block \rangle$ ▷ a message is a pair containing operation *op* and the *block* operation
1: **on** operation$_i(op)$
2: $m \leftarrow$ PREPARE$_i(op)$
3: **if** $m \neq \perp$ **then** ▷ m is set to $\perp$ when the operation's precondition is not satisfied
4: bcast$_i(m)$
5: **on** deliver$_i(m, t)$ ▷ we assume causal delivery
6: $c \leftarrow \langle op, block, t \rangle$ ▷ call c contains the operation *op*, the *block* operation, and the timestamp t
7: EFFECT$_i(c)$

An operation consists of a sequence of reads and writes that a replica executes atomically. In line with causal consistency, operations are propagated between replicas using a broadcasting mechanism that guarantees causal delivery. Causally-related operations are ordered according to the "happens-before" relation [29], whilst concurrent operations remain unordered relative to each other. Consequently, the execution order of concurrent operations may differ across replicas, potentially leading to consistency conflicts. These conflicts fall into two categories: i) convergence conflicts, which occur when replicas reach different states by applying concurrent operations in different orders, and ii) invariant conflicts, which arise when concurrent operations violate application-specific invariants.

An operation is *conflicting* if it can cause convergence or invariant conflicts. As such, read-only operations, which do not modify the application state, are non-conflicting.

Algorithm 1 outlines the execution model of the No-Op framework. It draws inspiration from operation-based CRDTs [40] and divides the execution of an update operation into two phases: the prepare phase and the effect phase.

When a user invokes *op* on replica i, the operation$_i$ event on that replica is triggered. This event triggers the prepare phase, executed exclusively on the replica where the operation was invoked. During this phase, the system validates the operation's preconditions. If the preconditions are satisfied, the PREPARE$_i$ returns a message m containing the operation *op* and a *block* operation, codifying the conflict resolution policy to handle conflicts involving the submitted operation. Based on the predefined policy, the *block* specifies which operations cannot be executed concurrently with the operation that issued it. In other words, the *block* encodes the less preferred operations. Consequently, there may be operations where the *block* is null, meaning that it represents the losing operation in the conflict. Conversely, an operation may have a *block* on multiple operations, indicating that it conflicts with and prevails over several others. Then, bcast$_i$ broadcasts the message to all replicas by calling the reliable causal broadcast API (which will add a timestamp to the message).

When a message m with timestamp t is delivered to replica i, the deliver$_i$ event is triggered. The event first creates a tagged call c, associating the message with its timestamp. As the message includes the operation call and the issued *block*, the tagged call is a triple consisting of the operation call, the *block*, and the timestamp. EFFECT$_i$ incorporates the tagged call (or simply call) into the partial order of operations and resolves any resulting local conflicts. A local conflict occurs when the replica detects that the delivered operation has a *block* against a local operation or vice versa. When the conflict is detected, the system transforms the blocked operation into a no-side-effect operation, called a No-Op. This transformation ensures that only the preferred operation (the one that issued the *block*) affects the application state, thus maintaining consistency and enforcing the predefined conflict resolution policy. Conflicts

can also arise with operations that have already been transformed into No-Ops. For instance, if a No-Op has a *block* for a newly received operation, a conflict exists between the two operations, and the system resolves it by transforming the new operation into a No-Op.

3.1 Model Correctness

We now prove that the No-Op framework guarantees strong convergence and safety. To do so, we assume three key prerequisites: i) all replicas receive all updates, ii) updates are delivered in causal order, and ii) a resolution policy is defined for every pair of conflicting operations.

As mentioned before, our model represents the partial order of operations as an execution graph, $G = \langle V, E \rangle$. G is a labelled directed acyclic graph (DAG) where vertices (V) represent tagged operation calls, and the edges (E) denote sequential dependencies between these calls. No-Ops are maintained in a set N, which consists of all the vertices marked as No-Ops. Concurrent operations remain unconnected in the graph, allowing for multiple topological orders. We prove that all topological orders are safe serialisations (Theorem 5) and that replicas receiving the same set of calls will converge to the same state (Theorem 7).

For simplicity, the proofs focus on the effect phase, as the prepare phase only creates the message to be broadcast to all replicas and does not modify the replica state. We assume the prepare phase has been executed, and the *block* has been issued according to the predefined conflict resolution policies. Additionally, the proofs do not distinguish between local and remote replicas, as only EFFECT_i modifies the state.

▶ **Lemma 1.** *Two replicas that observed the same calls have the same execution graph and the same set of No-Ops:*

$$\forall r_1 = \langle G_1, N_1 \rangle,\ r_2 = \langle G_2, N_2 \rangle :$$
$$(G_1 = \langle V_1, E_1 \rangle \land G_2 = \langle V_2, E_2 \rangle \land V_1 = V_2) \implies (E_1 = E_2 \land N_1 = N_2)$$

Proof. The graphs are equivalent iff, upon receiving the same set of calls, they contain the same vertices and edges. When executing the effect phase, the replica adds the call to its execution graph. Consequently, two replicas that observe the same calls will have identical vertices in their execution graphs. We now show that the graphs contain identical edges across replicas. The effect phase computes the edges between sequential calls by comparing each new call c with every other call in the graph. Causal consistency ensures that operations are causally delivered, allowing all replicas to process sequential operations in the same order. As a result, all calls preceding c are consistently compared to c, generating the necessary edges. Thus, all replicas maintain the same edges in their execution graphs. As the vertices and the edges are the same, we can infer that both graphs, G_1 and G_2, are identical.

Lastly, we demonstrate that even when replicas process concurrent calls in different orders, the model marks the same operations as No-Ops. When a tagged call c is added to the graph, the model checks whether c conflicts with its concurrent operations. For each pair of concurrent operations, we distinguish two scenarios. First, if one operation issues a *block* against the other, the blocked operation is marked as a No-Op. Second, if the operations do not conflict, no further action is taken. Since we assume that all operations are eventually delivered, all updates are processed, ensuring all blocks are also processed. Consequently, after processing the same set of operations, all replicas will have identified the same No-Ops. ◀

To demonstrate that all topological orders are safe serialisations, we first define safety based on the definition from [18] adapted to our model.

▶ **Definition 2** (Safe calls). *Two concurrent calls are safe iff applying them in any order preserves the operations' preconditions and invariants. Otherwise, they are unsafe.*

▶ **Definition 3** (Safe operations). *Two operations are safe iff all pairs of concurrent calls to those operations are safe.*

▶ **Definition 4** (Safe serialisation). *A serialisation of correct calls is safe iff every pair of concurrent calls is either safe or includes at least one call that has been marked as a No-Op.*

▶ **Theorem 5.** *All topological orderings of an execution graph G are safe serialisations.*

Proof. By induction on the length of the topological order.

Base Case. No calls occurred, so the topological ordering is empty and trivially safe.

Induction Step. Assume a safe topological order of dimension n derived from an execution graph. When a message is delivered, the effect phase incorporates the new call into the execution graph and calculates the No-Ops according to the issued blocks. We distinguish two cases: either the new call is safe or unsafe.

Case 1. If the new call is safe, indicating no conflicts with any other concurrent call in the execution graph, we can add the call to the graph, ensuring that the resulting topological order(s) are safe serialisations.

Case 2. If the new call is unsafe, the new call invariant conflicts with at least a concurrent operation in the execution graph. Since the call is conflicting, it either appended a *block* during the prepare phase or the conflicting calls did. The blocked operations are then transformed into No-Ops, ensuring that the resulting topological order(s) are safe serialisations. Therefore, assuming the prepare phase was correctly executed, and conflict resolution policies are defined for all conflicts, a *block* must exist for every pair of conflicting operations, ensuring safety.

Starting from an execution graph with a safe topological order of dimension n, our model builds an expanded graph whose topological order of dimension $n+1$ is a safe serialisation. ◀

▶ **Lemma 6.** *Two topological orderings from the same graph G converge to equivalent states.*

Proof. We follow a proof by contradiction. Assume two topological orderings, t_1 and t_2, that result in divergent states. We know that, in both $t1$ and $t2$, all sequential operations appear in the same order. However, concurrent operations may appear in different orders. Given the assumption that the states diverge, there must be at least two concurrent non-commutative calls, c_1 and c_2, appearing in different orders in t_1 and t_2. Let us assume, without loss of generality, that $t_1[c_1] < t_1[c_2]$ and $t_2[c_1] \not< t_2[c_2]$, where $t_i[c_i]$ represents the position of call c_i in the topological ordering t_i. Since c_1 and c_2 are non-commutative, executing them in different orders would lead to divergent outcomes. However, according to the conflict resolution policy, non-commutative operations will issue a *block* in the prepare phase. Therefore, one operation will block the other for all pairs of non-commutative conflicting operations. Consequently, either c_1 or c_2 would become a No-Op. As No-Ops do not affect the application state, it follows that c_1 and c_2 commute, which contradicts our initial assumption that c_1 and c_2 are non-commutative and lead to divergent states. By processing all operations in the graph this way, we ensure that all remaining operations are commutative, guaranteeing that any topological ordering will yield equivalent results. ◀

▶ **Theorem 7.** *Two replicas that observed the same tagged calls converge to equivalent states.*

Proof. Assuming both graphs have observed the same calls, it follows from Lemma 1 that they have the same execution graph. Additionally, by Lemma 6, any topological order within the same graph leads to an equivalent state. Considering the graphs are equal, we can infer that their topological orders are equivalent, thus converging to an equivalent state. ◀

4 No-Op Replication Protocol

We now introduce a coordination-free replication protocol that leverages the No-Op approach to ensure strong convergence and uphold invariants in distributed systems without requiring coordination. The protocol enables replicas to execute operations without synchronisation and handles conflicts as they arise. As explained in Section 2, conflicts are identified and resolved using a *block* operation that encodes a programmer-defined resolution policy. After resolving the conflict, *at most one* of the conflicting operations affects the state.

Algorithm 2 provides an overview of the replication algorithm that implements the model proposed in Section 3. Each replica maintains an execution graph, a No-Op (losing calls) set, and the application's initial state. As in the model, the execution graph is a labelled directed acyclic graph (DAG) in which vertices represent tagged operation calls, and edges denote the sequential dependencies between them. When two concurrent operations conflict – one operation blocks the other – the algorithm designates the blocked operation as a No-Op, ensuring that only the preferred operation affects the application's state.

Since each replica only maintains the initial state, the current state of the application must be calculated based on the operations in the execution graph. The GETSTATE function computes the current state by executing the operations according to the partial order maintained in the graph. The function begins by determining a topological order of the graph. Starting from the initial state, each operation in the topological order is applied, except for No-Op calls, which are ignored and left unexecuted.

The PREPARE function, executed exclusively by the replica where the operation was submitted, verifies whether the operation is enabled in its current state and, if necessary, issues a *block*. The function begins by calculating the replica's local state and evaluating the precondition. If the precondition does not hold, the operation cannot execute, and the function returns $\perp$. Otherwise, it returns a message containing the operation and the corresponding *block*. The *block* is determined based on the operation, the current state, and the predefined conflict resolution policies. In some cases, the application's state does not influence the *block*. However, specific policies may use the state to decide which operations to block (as discussed later in Section 5.2.4).

The EFFECT function operates on all replicas and incorporates the received call into each replica's local operation graph. The function begins by adding the edges between the new call c and all preceding calls in the graph. Since the model assumes causal delivery, no calls occurring after c exist in the graph at this stage. Next, the function checks if the new call conflicts with its concurrent operations. Conflicts are detected using the OPSCONFLICT function, which returns *true* if one operation has a *block* on the other. If a conflict is detected, the algorithm gets the blocked operation through the GETBLOCKEDOP function and adds it to the No-Ops set. Finally, the function adds the call to the set of vertices.

The algorithm allows a replica to validate an operation's precondition while concurrently delivering a remote operation. This is not a problem, as any invalidation of the precondition by the remote operation indicates a conflict, which the system detects and resolves when the effect function integrates the local operation into the graph.

Algorithm 2 The No-Op algorithm main functions.

```
    replica local state
      G = ⟨V, E⟩                                   ▷ execution graph
      N                                            ▷ set of calls marked as No-Ops
      σ_0                                          ▷ initial state
 1: function PREPARE(op)                           ▷ check if an operation can be executed in the current state
 2:     σ ← GETSTATE()                             ▷ calculate the current state based on the execution graph
 3:     if PRE(op, σ) then                         ▷ check if the operation precondition holds in the current state
 4:         b ← GETBLOCK(op, σ)                    ▷ generate the block according to the conflict resolution policy
 5:         return ⟨op, b⟩                         ▷ return the operation and the block if the precondition is satisfied
 6:     else
 7:         return ⊥                               ▷ indicate the operation is not enabled in the current state
 8: function EFFECT(c)                             ▷ integrate call c into the execution graph
 9:     for v ∈ V do                               ▷ determine edges and No-Ops generated by call c
10:         if v ≺ c then                          ▷ call v happens-before call c
11:             E ← E ∪ {⟨v, c⟩}                   ▷ add edge between call v and call c
12:         else if v ∥ c ∧ OPSCONFLICT(v, c) then ▷ operation blocks the concurrent operation
13:             l ← GETBLOCKEDOP(v, c)             ▷ get the blocked operation
14:             N ← N ∪ {l}                        ▷ add the losing call l to the No-Op set
15:     V ← V ∪ {c}                                ▷ add call c to the graph vertices
16: function GETSTATE()                            ▷ calculate the current state
17:     to ← GETTOPOLOGICALORDER()                 ▷ get a topological order from the graph
18:     σ ← σ_0
19:     for c ∈ to do                              ▷ apply all calls according to the topological order
20:         if c ∉ N then                          ▷ No-Ops are ignored and are not executed
21:             σ ← EXECUTE(σ, c)                  ▷ execute call c over state σ
22:     return σ                                   ▷ return the current state
```

4.1 Implementation in VeriFx

We implemented the core of the No-Op replication protocol in VeriFx [17], a programming language for RDTs with automated proof capabilities. VeriFx allows developers to implement RDTs in a high-level language atop functional collections and express correctness properties that are verified automatically. Alongside the algorithm implementation, we developed formal proofs to automatically verify the algorithm's correctness properties when applied to specific data types or application implementations. The implementation considers only the effect phase of Algorithm 2. As in the formal proofs in Section 3.1, we assume the prepare phase was correctly executed according to the programmer-defined conflict resolution policies.

4.1.1 Conflict Resolution Policies Specification

We start by detailing how conflict resolution policies are specified in our VeriFx implementation. These policies are implemented by means of two functions:

- The `blocksEncoding` function verifies whether an operation call is valid. It takes `TaggedOp` type, combining the operation and a corresponding *block*, and defines the *block* value according to the operation type. Since we assume that the prepare phase was correctly executed and VeriFx will generate any possible combination of operation and *block*, we use this function to restrict the set of operations considered in the proofs to those that are correct (i.e., those whose blocks match GETBLOCK in Algorithm 2). Programmers must define this function.

- The `isBlockedBy` function takes two calls and defines the condition under which a given operation instance is blocked by another. It corresponds to the opsConflict in Algorithm 2. Typically, an operation blocks all instances of another operation of a given type. Accordingly, the default implementation of this function returns `true` if a given operation has a *block* that matches the other operation. For some data types (e.g., the LWW Register), an operation may not block all instances of another type. In such cases, the programmer must provide their custom implementation.

The implementation of `blocksEncoding` and `isBlockedBy` depends on the data type semantics. We now illustrate how programmers specify conflict resolution policies through these functions in the context of two concrete data types.

Add-Wins Set. The implementation of `blocksEncoding` for a replicated set defining an add-wins policy is shown below:

```
1  def blocksEncoding(v1: TaggedOp[Time, ID, SetOp[V]]): Boolean =
2    v1.op match {
3      case Add(e) => v1.b == new Rmv(e)
4      case Rmv(_) => v1.b == new NullOp[V]()
5      case _ => false
6    }
```

Given an operation, the `blocksEncoding` function encodes how blocks should have been assigned during the prepare phase. In this case, if the operation is an `Add(e)`, the *block* is equal to `Rmv(e)`. Conversely, if the operation is a `Rmv(_)`, the *block* is null (i.e., the operation does not block any operation) since it represents the losing side of the conflict. In VeriFx, we represent this null value using a special operation called `NullOp`. Accordingly, for the case of `Rmv`, the function returns `true` if the associated *block* is `NullOp`. The last case ensures pattern matching covers all possible scenarios, including `NullOp`. Since `NullOp` does not correspond to an actual operation, there are no operations of this type, and the function returns `false`. This encoding guarantees that only operations with correctly triggered blocks are considered. For example, it ensures that in the proofs we only consider `Add(e)` operations that correctly blocks `Rmv(e)`, and excludes any `Add(e)` that does not block any operation.
In the add-wins set, the *block* matches precisely the operation that should be transformed into a No-Op, i.e., the removes on the same element. Therefore, we can rely on the default implementation of the `isBlockedBy` function.

LWW Register. Listing 1 shows the specification of a register defining a last-writer-wins (LWW) policy. The `blocksEncoding` function defines that an `Assign` operation must block other concurrent `Assign` operations that assign the same value to the register (i.e., `Assign(5)` would only block concurrent `Assign(5)` operations). However, to implement the LWW policy, an `Assign` operation should block all operations lower in the total order for the LWW, regardless of the assigned value. For instance, a *block* on `Assign(5)` must block all concurrent instances of `Assign(e)` operations that are older in the total order, independently of the value of `e`. We specify this requirement in the custom implementation of the `isBlockedBy` function where blocking is defined by operation identifiers (`opID`). As shown in lines 8–10, the operation in the first call (`op1.op`) is considered blocked if both `op1.op` and the *block* of the second call (`op2.b`) are of type `Assign(_)`, regardless of their arguments (denoted by the underscore in pattern matching), and if the identifier of the first call (`op1.opID`) is smaller than that of the second (`op2.opID`). Line 14 ensures exhaustive pattern matching. In the LWW register, it only matches `NullOp`, which is never triggered and thus never evaluated.

Listing 1 Conflict resolution policy specification for a last-writer-wins register.

```
def blocksEncoding(v1: TaggedOp[Time, ID, RegOp[V]]): Boolean =
  v1.op match {
    case Assign(x) => v1.b == new Assign(x)
    case _ => false
  }

override def isBlockedBy(op1: TaggedOp[Time, ID, RegOp[V]],
                         op2: TaggedOp[Time, ID, RegOp[V]]) = {
  op1.op match {
    case Assign(_) => op2.b match {
      case Assign(_) => this.smaller(op1.opID, op2.opID)
      case _ => false
    }
    case _ => false
  }
}
```

4.1.2 Implementation and Verification of the No-Op Replication Protocol

Our implementation closely follows the formal model, with a few optimisations. In the model, the prepare phase assigns blocks according to conflict resolution policies, and all *block* operations are issued during this phase. In the VeriFx implementation, we assume the prepare phase has been correctly applied. We encode this assumption using the `blocksEncoding` function, which specifies which operations are invalid and should not be considered in the verification process. Additionally, rather than issuing all *block* operations, our implementation only allows one type of concurrent operation to be blocked. Therefore, we need the `isBlockedBy` function to determine whether a specific operation instance is blocked.

To prove the algorithm's correctness, we must guarantee that the implementation upholds the properties of the model proposed in Section 3: strong convergence and safety. In VeriFx, we encoded these properties through two graph-based properties. The first property aligns with the property proven in Lemma 1, demonstrating that applying two concurrent operations in different orders on the same state produces an identical graph and the same No-Ops. We only need to verify this property for concurrent operations, as sequential operations are processed in the same order due to the assumption of causal delivery. The second property encodes the structural characteristics of the graph. It ensures that edges connect all sequential operations and that, for any conflicting pair of concurrent operations, one operation is marked as a No-Op and added to the set of No-Ops. This second property encodes the properties of Theorems 5 and 7.

Listing 2 presents the proofs used to verify the algorithm's correctness. The proofs employ three helper functions present in VeriFx to facilitate the verification process of RDTs:

- The `compatible` function returns true if the two operations denote different operations.
- The `enabledOp` function assesses whether an operation is enabled in the current state. It first checks that the operation has not yet been delivered and that all predecessor operations have already been delivered. This verification is needed because our model assumes causal delivery of operations. Additionally, the function ensures the tagged operation call is correct using the `blocksEncoding` function.
- The `reachable` function defines the states the system can attain. It integrates the structural properties of the graph, ensuring that conflicts between concurrent operations are appropriately resolved and that edges exist between sequential operations.

Listing 2 Proof verifying the algorithm's correctness under concurrent operations, leveraging graph-based properties.

```
proof noOpStrongConverges[Time, ID] {
  forall(s: T[Time, ID], x: TaggedOp[Op, ID, Time],
                         y: TaggedOp[Op, ID, Time]) {
    (s.compatible(x, y) && s.reachable() &&
     s.enabledOp(x) && s.enabledOp(y) &&
     s.execGraph.timeProps.concurrent(x.stamp, y.stamp)) =>:
       s.effect(x).effect(y).equals(s.effect(y).effect(x))
  }
}

proof noOpIsSafe[Time, ID] {
  forall(s: T[Time, ID], x: TaggedOp[Time, ID, Op]) {
    (s.reachable() && s.enabledOp(x)) =>: s.effect(x).reachable()
  }
}
```

The proof `noOpStrongConverges` ensures that starting from a reachable state with two concurrent operations enabled, applying these operations in different orders results in the same graph state. The proof `noOpIsSafe` verifies that integrating an enabled operation into the graph preserves the safety properties encoded in the `reachable` function.

Like our manual proofs (cf. Section 3.1), the VeriFx proofs assume that all possible conflicts have a defined conflict resolution policy. Thus, the analysis may incorrectly indicate that the data type is correct if the resolution policies do not cover all conflicts.

We used the VeriFx implementation of our approach to implement some data types (discussed later in Section 5) and successfully verified their correctness. In this work, data type implementations are used exclusively for verification purposes. However, VeriFx also supports the transpilation of the data types into mainstream languages such as Scala and JavaScript. Thus, if the No-Op algorithm were integrated into a database, developers could design and verify their data types in VeriFx, subsequently transpiling them into a mainstream language for seamless database integration.

4.2 Graph Compaction

Although the algorithm maintains consistency properties, i.e., ensuring convergence and preserving invariants, it is impractical for real-world use due to the linear growth of each replica's state with the number of operations. To make our approach practical and more efficient, we leverage the concept of causal stability [9] to determine when an operation call can be committed and removed from the graph. This optimisation reduces space and prevents the need to reapply the entire operation history for every incoming call. By committing causally stable calls, we ensure that the graph remains compact, improving efficiency while preserving the correctness guarantees of the original algorithm.

An operation is *causally stable* at a replica i when the replica can no longer receive any additional concurrent operations. Once stabilised, the operation can no longer be transformed into a No-Op nor cause other operations to become No-Ops. Therefore, replicas can locally commit causally stable calls without requiring coordination.

The commitStableCalls function, described in Algorithm 3, implements the causal stabilisation of our approach. The function takes a stable timestamp t as its argument and begins by identifying the stable calls in the graph. The getStableCalls function identifies all calls with a timestamp less than or equal to t and returns them as a sequence of operations. This sequence respects the causal ordering defined by the graph. Next, each stable call in the sequence is removed from the graph and applied to the state σ_0. If an operation is a

Algorithm 3 Commiting causally stable calls.

```
1: function commitStableCalls(t)
2:     stableCalls ← getStableCalls(t)          ▷ get the stable calls
3:     for c_s ∈ stableCalls do                 ▷ we assume stableCalls respect causal ordering
4:         V ← V \ {c_s}                        ▷ remove the stable call from the vertices
5:         E ← E \ in(c_s) \ out(c_s)           ▷ remove incoming and outgoing edges
6:         if c_s ∉ N then                      ▷ the stable call is not a No-Op
7:             σ_0 ← execute(σ_0, c_s)          ▷ apply the stable call c_s in the state σ_0
8:         else                                 ▷ the stable call is a No-Op
9:             N ← N \ {o_s}                    ▷ remove the stable call from the set of No-Ops
```

No-Op, it does not affect the stable state, as it has no side effects.

Some applications may benefit from being notified when operations are committed. For example, in an online sales application, a developer may want a purchase to remain in an *accepted pending* state until it is final. Only after the operation is committed (and can no longer be modified) should a notification be sent to the customer, confirming that the purchase has been successfully processed. To support this behaviour, our framework can be extended to let developers register callbacks on operations, which are triggered asynchronously when the operation is committed. Note that the callback notification can be sent as soon as a single replica commits the operation; there's no need to wait for every replica to commit.

4.2.1 Causal Stabilisation and No-Op Operations

During execution, conflicts are resolved by transforming operation calls into no-side-effect operations, known as No-Ops. Since conflicts are handled locally by each replica as they arise, different replicas may detect and resolve them in different orders due to variations in operation arrival. To ensure determinism, the computation of No-Ops must consider all concurrent operations, including those already marked as No-Ops. Consequently, although No-Ops do not modify the application state, they must remain in the graph until causal stabilisation to ensure deterministic conflict resolution and prevent divergence.

To illustrate why No-Op operations must remain in the graph, consider the example in Figure 4, based on the albums management system introduced in Section 2. Initially, the database contains an artist named *Sam*. Now, consider three concurrent operations: `addAlbum(A1, Sam)`, which adds an album *A1* by *Sam*; `rmvArtist(Sam)`, which removes *Sam* from the database; and `updArtist(Sam)`, which updates *Sam*'s information. The operations are executed on replicas 1, 2, and 3, respectively. For simplicity, the figure depicts only two of the three replicas, showing their stable and current state. Replica 1 first executes `addAlbum(A1, Sam)`(denoted as `c1` in the graph), and then it receives the `rmvArtist(Sam)` operation. When the effect phase for `rmvArtist(Sam)` is triggered, replica 1 detects a conflict between the operation and the *block* issued by call `c1`. Consider that the conflict is resolved by removing `c2` from the graph (as the operation would be marked as a No-Op). When `updArtist(Sam)` is delivered to replica 1, it is added to the graph without conflict.

Let us turn our attention to the execution graph in replica 2. Consider that replica 2 executed the `rmvArtist(Sam)` operation, but its effect phase is only triggered after the effect phase of the two remote operations: `addAlbum(A1, Sam)` and `updArtist(Sam)`. First, `addAlbum(A1, Sam)` is processed and added to the graph (denoted as `c1`). Later `updArtist(Sam)` is also added (denoted as `c3`) as the operation does not conflict with `c1`. However, two conflicts arise when the effect phase of `rmvArtist(Sam)` is triggered. The *block*

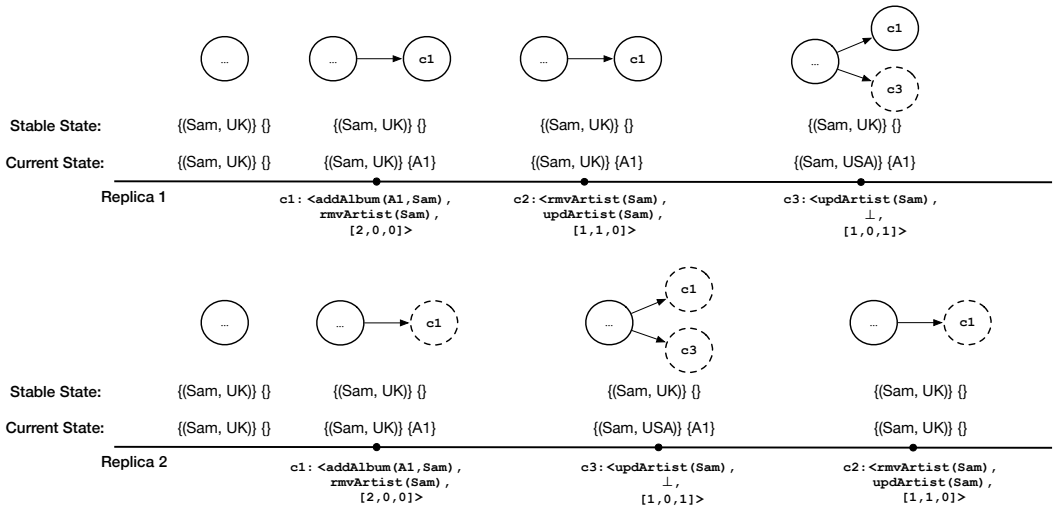


Figure 4 Convergence problems caused by removing No-Op operations from the graph before stabilisation.

on `updArtist(Sam)` issued by call `c3` leads to the removal of call `c2` from the graph. On the other hand, the *block* issued by `addAlbum(A1, Sam)` on `rmvArtist(Sam)` also results in call `c2` being removed from the graph. After executing the same set of operations, the state of the replicas diverges. The problem is that removing operations from the graph when marked as a No-Op before they are causally stable does not guarantee convergence. As such, our algorithm retains unstable No-Op operations in the graph. Maintaining these operations, along with comparing each operation against all its concurrent operations – including those already marked as No-Ops– ensures consistent conflict resolution and prevents divergence.

To optimise metadata storage in the graph, we can replace the operation marked as No-Op with null (i.e., store $\langle \perp, \textit{block}, \text{timestamp} \rangle$). This optimisation is possible because the only information required for a No-Op is the associated *block* operation and the timestamp to detect causal stability. Retaining the operation information is thus unnecessary since a No-Op does not affect the system state.

5 Applicability of the Solution

This section demonstrates how the No-Op framework supports the implementation of CRDTs (Section 5.1), enforces a wide variety of application invariants (Section 5.2), and enables the development of distributed geo-replicated applications such as the RUBiS system (Section 5.3).

The No-Op approach supports a wide range of conflict resolution policies, allowing flexibility in handling concurrent operations. Examples include operation-type-based policies, such as update-wins and delete-wins, where one type of operation takes precedence over the other. Additionally, it supports total-order-based policies, such as the last-writer-wins and first-writer-wins policies, which resolve conflicts based on a total order of operations. Finally, privilege-based policies can resolve conflicts based on criteria, such as user roles, giving precedence to operations initiated by higher-privileged users within the system. Besides these static policies, the framework supports state-dependent policies, where decisions are made based on the system's current state. Ultimately, our mechanism's generality allows programmers to define any policy expressible in terms of blocks.

All policies and mechanisms outlined in this section were implemented and verified using VeriFx. This confirms the correctness of our approach and demonstrates its practicality for

real-world applications.

5.1 CRDTs

Conflict-free Replicated Data Types (CRDTs) [40] are the most well-known class of RDTs. They automatically resolve conflicts by leveraging mathematical properties to ensure convergence by design. These data types can be classified into two categories: those with commutative operations, which naturally avoid conflicts, and those with non-commutative operations, which require conflict resolution policies.

Commutative data types inherently avoid convergence conflicts and do not require conflict resolution policies. Under causal delivery assumptions, if all operations are commutative, they can be applied directly to the state without needing an execution graph. Examples of such data types include the grow-only set [40] and the counter [40]. Since these data types are conflict-free, no operations become No-Ops, making them immediately causally stable and directly applicable to the stable state. In other words, these data types can be implemented as a sequential set and a sequential counter, respectively.

In contrast, non-commutative data types require embedding a conflict resolution policy within the data structure to prioritise specific operations when conflicts arise. These CRDTs leverage conflict resolution policies, making their implementation straightforward with the No-Op framework. For instance, to implement an add-wins set, the conflict resolution policy specifies that add operations take precedence over remove operations on the same element (cf. Section 4.1.1). During the prepare phase of an add operation, the No-Op framework generates a *block* on concurrent removes targeting the same element. As a result, those remove operations are transformed into No-Ops.

Overall, CRDTs typically rely on operation-type-based or total-order-based policies, both supported by the No-Op framework, allowing diverse CRDT implementations. We have implemented in our VeriFx implementation different registers, flags, counters, and sets.

5.2 Invariant Maintenance

This section describes how our model can enforce a wide variety of invariants, ranging from simple invariants on single objects to more complex invariants over multiple objects. To illustrate invariant's examples, we use the albums management system presented in Section 2.

5.2.1 Equality and Inequality of Attributes

Equality and inequality of attributes [4] restrict the values assigned to an attribute. In our running example, an example of attribute equality is defining that the *genre* of an album can only take predefined values, such as *"rock"*, *"pop"*, *"jazz"*, or *"classical"*. This invariant type does not generate conflicts between operations, not requiring any conflict resolution policy to enforce it. Since this invariant does not generate conflicts, it is trivially maintained within our framework without requiring any conflict resolution policy.

5.2.2 Uniqueness

Uniqueness [4, 7, 6] includes all constraints regarding attribute uniqueness and is commonly used to denote primary keys or identifiers. For instance, in our running example, requiring artist identifiers to be unique is a typical uniqueness invariant.

We can classify the identifiers into system-generated identifiers and user-chosen identifiers. In system-generated identifiers, we assume a mechanism that automatically generates them,

ensuring their uniqueness [7, 30]. By generating unique identifiers, the system guarantees that no conflicts related to identifier uniqueness will arise, eliminating the need for a policy to solve uniqueness conflicts. For user-chosen identifiers, two operations conflict if attempting to add distinct objects with the same identifier (insertions with different identifiers do not conflict). When n operations attempt to add different objects with the same identifier, $n-1$ operations need to be converted into No-Ops, allowing only one operation to be successfully executed, thereby preserving the invariant. Therefore, the No-Op approach can resolve these conflicts through a total-order-based policy. Each operation blocks all preceding operations, as determined by a total order (e.g., enforced by Lamport clocks), that attempt to add a different object with the same identifier. As a result, after processing the operations, the No-Op framework converts all but one into No-Ops, ensuring the invariant is preserved.

5.2.3 Referential Integrity

Referential integrity [4, 7, 6] defines dependency relationships between objects, with a parent object representing the referenced entity and a child object referencing the parent. In the context of the running example, a referential integrity constraint could ensure that albums by nonexistent artists cannot exist in the system.

Conflicts arise when an operation inserts or updates an object that references a parent object which has been concurrently deleted. To manage these conflicts without coordination, Antidote SQL [35] introduced two conflict resolution policies: update-wins and delete-wins. Under the update-wins policy, the conflict is resolved by keeping the parent object in the database. In contrast, the delete-wins policy resolves the conflict by removing both the parent object and the concurrently inserted or updated child.

Our model supports the implementation of both policies. In the update-wins policy, insert and update operations on child objects take precedence by blocking the parent deletion, rendering it a No-Op. In the delete-wins policy, the parent deletion takes precedence, blocking insert and update operations on child objects and transforming them into No-Ops.

Furthermore, our model enables the implementation of variations of these policies by simply adjusting how the operations are defined. Consider the update-wins policy. If the operation that deletes the parent object also removes its children, applying the policy would prevent both actions from being executed. As a result, after resolving the conflict, both the parent and its children remain in the database. In contrast, if the operation that deletes the parent first triggers a separate operation to delete its children, the outcome changes. Since deleting the children is an independent operation, it does not conflict with inserting a new child. As a result, the parent remains in the database, while the deleted children do not. This behaviour aligns with the update-wins semantics proposed in Antidote SQL.

5.2.4 Disjunctions

A disjunction invariant requires at least one of several conditions to be true. A violation occurs when all true predicates concurrently transition to false. Therefore, conflicting operations are the operations that transition a predicate from true to false.

Consider an extension to the album management system where each album can contain multiple tracks but must always include at least one. This version introduces two additional operations: adding a track to an album and removing a track from an album. Given this property, operations that remove different tracks conflict, as they may violate the invariant. For example, if an album initially contains two tracks and two operations attempt to remove them concurrently, the album would end up with no tracks, which breaks the invariant.

The disjunction invariant illustrates how issuing blocks based on the system's state can be beneficial. The remove track operation is self-conflicting, which makes it responsible

for triggering blocks. However, traditional policies, such as operation-type or total-order-based, are overly restrictive, limiting execution to at most one operation. Total-order-based policies allow only a single operation to proceed, while operation-based policies prevent all operations from executing, as self-conflicting operations block each other. However, enforcing a disjunction invariant only requires ensuring that at least one predicate remains true.

To address this, we propose a state-based policy where the *block* operation is issued based on the application's current state. This approach preserves the truth value of a predicate by transforming any operation that would transition it to false into a No-Op. By blocking all operations that would make the predicate false, the policy ensures that at least one predicate remains true, thereby maintaining the invariant.

In the example, the *block* operation would guarantee that a specific track could not be removed from an album. In this scenario, the *block* operation would block all operations that transition the predicate "track x is in album y" to false. Consequently, any operation that concurrently attempts to remove track x from album y would conflict with the *block* operation, as it alters the truth value of the blocked predicate. The conflicting remove operation would then become a No-Op.

The specification of such a state-based policy follows a similar approach to the LWW register shown in Section 4.1.1. The specification is provided in Appendix A in [12]. Assuming the application defines the predicate to be blocked and returns the selected predicate to the framework, the framework then issues blocks on all operations that would transition the predicate to false.

Issuing blocks based on the system's state enables the implementation of more fine-grained policies, reducing the number of conflicts. However, it is important to note that some operations may still be unnecessarily undone, depending on how the blocks are assigned. For instance, if the predicate to be blocked is chosen randomly, different operations may block multiple predicates, turning several operations into No-Ops. In contrast, if all operations attempt to block the same predicate, only the operations affecting that specific predicate are undone, minimising the number of No-Ops.

5.2.5 Numerical Invariants

Numerical invariants [4, 7, 6] are usually associated with attributes manipulated by increment or decrement operations, and where there must be some control over the values of the attribute since there is a lower or upper bound. An example of a numerical invariant is guaranteeing that a user's balance does not become negative.

Although the No-Op approach preserves numerical invariants, it can be overly restrictive. Due to the self-conflicting nature of the operations, they block each other and, in some cases, generate more No-Ops than necessary. Consider a counter with a lower bound of 0 and an initial value of 100. Decrements are inherently self-conflicting. Applying the No-Op approach would require decrements to trigger blocks on other decrements. If each decrement blocked another, none of the operations would execute, as the decrements would cancel each other out. Similarly, policies based on a total order would allow only one operation to succeed. For example, if five concurrent decrements of 10 units each were issued, only one decrement would be applied (e.g., the latter following the total order). This behaviour is overly restrictive, as executing all decrements would maintain the invariant and respect the lower bound.

To overcome this limitation, we propose integrating escrow techniques [36, 7, 8] into the model. These techniques enable replicas to pre-allocate permissions for executing future updates, ensuring constraints are maintained. For example, consider a counter $x = 2$ with the invariant $x \geq 0$. In this case, the counter can be decremented by up to two units without violating the invariant. This decrement capacity represents the available rights in the system,

which are distributed among replicas. In a system with three replicas, the rights could be allocated as follows: replica 1 and replica 2 each hold one right, while replica 3 has none. If a decrement operation is submitted to replica 1, it can safely execute as it acquires an available right, preserving the invariant. Conversely, if a decrement operation is submitted to replica 3, it is aborted due to the lack of rights.

If the system supports escrow mechanisms, the necessary rights for an operation would be acquired in advance during the prepare phase, ensuring that the operation can be safely executed across replicas. In other words, escrow techniques eliminate conflicts by converting conflicting operations into non-conflicting ones. As a result, operations that previously required conflict resolution policies can now be executed safely within our framework, as conflicts no longer arise. A bounded counter [8] can, therefore, be implemented as a sequential counter, eliminating the need for an execution graph, much like in commutative CRDTs.

5.2.6 Aggregation Constraints

Aggregation constraints [6] enable the establishment of limits on the size of collections or enumerable attributes. Since the counted objects are in the database, aggregation constraints can be expressed as a disjunction of predicates. For instance, the example provided for the disjunctions – the number of songs in an album must be greater than one – is also an example of an aggregation constraint, as this property restricts the size of a collection of tracks.

Aggregation constraints comprise both lower and upper bounds. In lower bounds, the operations that remove an element from the collection are self-conflicting. Therefore, blocks are triggered whenever an element is removed from the collection. The number of blocked operations depends on the specified limit. For example, if the limit is 1, removing an element from the collection triggers a *block* on operations that remove a specific element. If the limit is 2, removing an element must block the removal of two different elements. Therefore, it triggers blocks on operations that remove one of the two blocked elements.

Conversely, in upper bounds, the conflicting operations are the operations that insert elements into the collection. However, identifying which operations to block is challenging, as it would require preemptively blocking any potential insertion, including block insertion of elements that may not yet exist in the system. Since there is no mechanism to block specific, non-existent elements, the *block* must be generic, restricting all insert operations. This approach can be overly restrictive and shares the same limitations as those faced with numerical invariants. In the future, we plan to explore alternative strategies to reduce the number of blocked operations without relying on escrow techniques.

5.2.7 Linear Resources

Linear resources [7] are applied to partitionable objects/resources to ensure no overlap. For example, a linear resource can be applied to ensure that a seat on a plane is not booked by more than one person. This type of invariant functions similarly to uniqueness, allowing only one operation from a set of concurrent operations to succeed. Consequently, any policy based on a total order is well-suited for maintaining this type of invariant.

5.3 Real-World Example

Ensuring application correctness requires preserving all defined invariants throughout execution and guaranteeing that all replicas converge after applying the same operations. So far, we have demonstrated how the No-Op framework supports the development of CRDTs and enables the creation of data structures that automatically maintain invariants. In this section, we bring these concepts together to illustrate how the No-Op framework can be used

	registerUser(u_2, ...)	unregisterUser(u_2)	update(u_2, ...)	placeBid(a_2, u_2, b_2, ...)	closeAuction(a_2)
registerUser(u_1, ...)	$u_1 = u_2$				
unregisterUser(u_1)	✓	✓			
update(u_1, ...)	✓	$u_1 = u_2$	$u_1 = u_2$		
placeBid(a_1, u_1, b_1, ...)	✓	$u_1 = u_2$	✓	$a_1 = a_2 \wedge b_1 = b_2 \wedge u_1 \neq u_2$	
closeAuction(a_1)	✓	✓	✓	$a_1 = a_2$	✓

Figure 5 Analysis of conflicts for the RUBiS system. Convergence conflicts are shown in yellow, whereas invariant conflicts are shown in red.

for developing a real-world distributed application. To this end, we develop an eBay-like auction system akin to the RUBiS system [14].

RUBiS supports operations such as registering and unregistering users, updating user information, opening and closing auctions, and bidding on open auctions. Additionally, it allows for direct sales, with operations to list items for sale and to purchase items. Based on the supported operations, we define five key invariants for RUBiS:

- System-assigned identifiers must be unique.
- User-chosen nicknames must be unique.
- Item stock cannot be negative.
- Each bid on an open auction must be associated with an existing user.
- There cannot be two bids with the same value for the same auction from two users.

Figure 5 illustrates all considered conflicting operations and their potential conflicts. Read operations are excluded, as they are safe-commutative and do not modify the database. The operations `openAuction`, `sellItem`, and `storeBuyNow` were analysed but are not shown in the figure since they do not conflict with any other operation. In the figure, yellow rectangles represent convergence conflicts, while red rectangles indicate invariant conflicts. Additionally, the relevant attributes responsible for each conflict are specified. For instance, conflicts between two user registration operations occur only when both use the same identifier.

Our RUBiS implementation in VeriFx assumes the system operates with escrow techniques and that its identifier generation mechanism ensures unique identifiers. These mechanisms guarantee that system-assigned identifiers are unique and that the item stock remains non-negative. As a result, there are no conflicting operations concerning these two invariants.

Recall that the No-Op approach relies on programmer-defined conflict resolution policies to handle conflicts between operations. The framework applies these policies at runtime to trigger blocks, which are used to detect and resolve conflicts. In the remainder of this section, we describe the resolution policies chosen for each identified conflict in the RUBiS system.

We opted for the last-writer-wins policy to resolve the uniqueness conflict between two `registerUser` operations. Similarly, we applied this same policy to address the convergence conflict caused by two `updateUser` operations targeting the same user. In both cases, the conflicts stem from self-conflicting operations, and only one operation from a set of concurrent operations must be executed. Consequently, any total-order-based policy is appropriate for resolving these conflicts.

For the same reasons, the uniqueness conflict caused by two users placing bids of the same value can also be resolved using a total-order-based policy. However, in this case, we have defined a policy that prioritises users based on the number of bids they have won. Specifically, the user who has won more auctions takes priority. If both users have won the same number of auctions, the policy resolves the tie using a total order over the operations.

Conflicts between `unregisterUser` and `placeBid` operations, which affect referential integrity, can be resolved using any of the policies described in Section 2. In our VeriFx im-

plementation, we adopted the remove-wins policy. This decision aligns with the application's semantics, favouring the removal of the user and their associated bids over forcing the user to remain in the system, where they could win the auction.

In convergence conflicts between `closeAuction` and `placeBid`, we defined `closeAuction` as the winning operation. Prioritising `closeAuction` ensures the auction closes properly, avoiding inconsistencies such as reopening the auction due to a concurrent bid. If a `placeBid` operation is delivered late due to a failure, it could inadvertently reopen the auction long after it has been closed. By using this policy, we avoid these situations.

Finally, for conflicts involving `updateUser` and `unregisterUser` operations, we selected `updateUser` as the winning operation.

6 Discussion

This section examines the practical implications of the No-Op approach, including its applicability to different applications and the impact of poorly designed conflict resolution policies. Additionally, it explores the trade-off between optimisation and policy determinism.

6.1 When and How to Apply the No-Op Approach

The No-Op approach resolves conflicts by discarding operations, transforming them into no-side-effect operations. Each replica maintains its local view of the system and, using only local knowledge, determines which operations should be transformed into No-Ops to ensure convergence while preserving invariants.

The model may result in operations being discarded after a response has already been sent to the client. For example, consider the referential integrity conflict in Figure 2. Under an update-wins semantics, a client may initially observe an artist as removed but later find that the artist is still present. This occurs because the remove operation is transformed into a No-Op only after the concurrent update operation has been delivered and the conflict resolved. As some operations may be discarded after execution, this mechanism is unsuitable for applications that cannot tolerate such behaviour. Based on this, applications can be classified into two categories:

- **Applications that can handle reversals through compensation mechanisms.** These applications can use the framework alongside compensation mechanisms [20, 44]. When an operation is transformed into a No-Op, the system can trigger a compensation action to maintain consistency.
- **Applications that do not support operation reversals.** These applications require strong consistency and cannot operate under weak consistency models, making coordination the only viable solution.

Many conflict resolution policies prioritise certain operations over others. For example, in the last-writer-wins policy, after applying all updates, the *oldest* update does not affect the system's state, which is similar to transforming it into a No-Op. By explicitly transforming an operation into a No-Op, our approach has the advantage of clearly identifying the operations with no side-effect, making it easier to integrate a recovery or notification mechanism, such as the mentioned compensations.

In some cases, only a specific pair of operations may not support reversals, while all other conflicting pairs do. For example, in the RUBiS scenario, a programmer may believe that their application does not support reversals on `placeBid` and `closeAuction`, meaning neither operation can be transformed into No-Ops. In this case, they may opt for coordination to handle these operations [30]. However, using coordination to prevent the concurrent

execution of operations that cannot be transformed into a No-Op does not preclude applying the No-Op approach to resolve other conflicts within the application. In such scenarios, the No-Op framework can be augmented with strong consistency mechanisms without affecting the rest of the system. For example, coordination could be implemented using locks, ensuring their correctness while allowing the remaining operations in the application to continue following the No-Op framework. This hybrid approach maintains efficiency while addressing specific consistency requirements where necessary.

6.2 Challenges with Poorly Designed Conflict Resolution Policies

Although the No-Op approach is coordination-free, it may sometimes mark operations as No-Ops unnecessarily. This issue can arise for different reasons. Sometimes, the model is overly restrictive for specific invariants, such as numerical invariants (cf. Section 5.2.5). In others, it may stem from the design of the conflict resolution policies implemented by the programmer. This section examines how poorly designed policies can increase the number of No-Ops and explores strategies to identify policy definition improvements.

To illustrate how poorly designed conflict resolution policies can lead to unnecessary No-Ops, let us revisit the example in Figure 4. Consider that the programmer specifies that `addAlbum` wins in conflicts with `rmvArtist`. In turn, `rmvArtist` wins in conflicts with `updArtist`. When `addAlbum`, `rmvArtist`, and `updArtist` are executed concurrently, the algorithm marks both `rmvArtist` and `updArtist` as No-Ops. As a result, fewer concurrent operations can be executed, impacting performance. However, defining `rmvArtist` as the losing operation in both cases solves the conflicts and ensures that only one of the three operations becomes a No-Op. Redefining the conflict resolution policies can thus improve performance without compromising the application's correctness.

Poorly designed conflict resolution policies could, however, lead to more severe issues hindering the overall system progress. For example, operation op_1 marks operation op_2 as a No-Op, operation op_2 marks operation op_3 as a No-Op, and operation op_3, in turn, marks operation op_1 as a No-Op. While this circular dependency does not compromise correctness, it causes all conflicting operations to be marked as No-Ops, hampering progress.

A policy graph provides a systematic way to identify and address those issues. In a policy graph, vertices represent conflicting operations, while edges show how conflicts are resolved. Long path lengths in the graph highlight scenarios where an excessive number of operations are unnecessarily marked as No-Ops. Meanwhile, cycles in the graph indicate circular dependencies. Therefore, programmers should define conflict resolution policies that ensure a cycle-free policy graph while minimising path lengths to reduce inefficiencies. In future work, we plan to explore an automated analysis tool that generates and analyses the policy graph based on programmer-defined conflict resolution policies. This tool will provide feedback, guiding developers to the policies that can be refined to enhance performance.

6.3 Optimisation for Deterministic Resolution

The correctness of our model, described in Algorithms 1 and 2, does not depend on the deterministic nature of conflict resolution policies. The policies only determine how blocks are triggered. If a policy is non-deterministic, two replicas in the same state executing the same operation may generate different blocks. However, since blocks are generated locally during the prepare phase, they are triggered only once per operation, ensuring their assignment remains deterministic. For example, consider a policy that resolves conflicts by randomly selecting an operation to be transformed into a No-Op. In this case, the blocked operation is

chosen randomly during the prepare phase. However, since the prepare phase is executed only by the local replica, the same *block* assignment is propagated to all remote replicas. During the effect phase, each delivered operation is compared against all its concurrent operation calls, including those already marked as No-Ops. As a result, all replicas process blocks uniformly, ensuring No-Ops converge. Therefore, by design, the No-Op approach ensures that replicas processing the same operations produce identical No-Op sets. This is achieved through the block abstraction, which guarantees that conflicts are resolved deterministically regardless of policy non-determinism.

When implementing the model, the creation of blocks could be optimised. Rather than creating the *block* when an operation is submitted (in the prepare phase), it is possible to simply perform the effect of blocks – transform concurrent conflicting operations into No-Ops according to the policy – upon detecting a conflict when an operation is delivered (in the effect phase). This optimisation reduces the amount of metadata that must be broadcast over the network and minimises the data stored in the graph, as blocks are no longer appended to operations.

Note that this optimisation is only applicable to deterministic conflict resolution policies. Since the policy is applied directly during the effect phase, a non-deterministic policy could lead to divergence, as different replicas may resolve conflicts inconsistently. Consider again the policy that resolves conflicts by randomly transforming an operation into a No-Op. In this case, two replicas processing the same conflict could make different decisions, resulting in divergent execution graphs. Another example is state-dependent policies, which issue blocks based on the application's current state.

Optimising the model comes at the cost of imposing stronger assumptions on the conflict resolution policies. While this optimisation reduces metadata overhead and storage costs by eliminating blocks, it requires deterministic conflict resolution to ensure convergence. In contrast, the original model allows for greater flexibility in defining policies, including non-deterministic ones, at the cost of maintaining extra metadata. If the optimised version is adopted, the verification step should verify whether the resolution policies are deterministic.

7 Related Work

Ensuring correctness in applications built on weakly consistent storage is an active research area, with two core challenges: achieving state convergence across replicas and preserving application invariants. Traditional replicated data types (RDTs) [40, 13, 26, 9] provide a principled, coordination-free approach to ensure replicas converge to the same state after executing the same operations. However, these data types are typically hand-crafted for specific use cases. In contrast, our framework offers a unified approach that can be applied consistently to implement different policies. Pure operation-based CRDTs [9] also maintain a graph of operations, encoding conflict resolution in the read operations. Our approach differs fundamentally by resolving conflicts during the write operations. While these approaches for building RDTs guarantee convergence, they do not address invariant maintenance.

The remainder of this section reviews methods for enforcing invariants, classifying them into coordination-based, reservation-based, and coordination-free approaches, and examines their characteristics and trade-offs.

Coordination-based approaches. Coordination-based approaches rely on synchronisation to prevent operations from violating invariants. As described in [4], invariant confluence provides a necessary and sufficient condition for determining whether invariants can be maintained under concurrent operations. This principle identifies the operations requiring coordination: non-$\mathcal{I}$-confluent operations. Several techniques aim to minimise the level

of coordination required. Lucy [46] achieves this by segmenting the state space and restricting transactions within each segment, ensuring each segment becomes $\mathcal{I}$-confluent. Similarly, Blazes [3] identifies locations needing coordination and automatically synthesises application-specific coordination code to reduce synchronisation overhead. Other approaches focus on generating replicated objects that inherently maintain invariants. For instance, Hamsaz [24] and Hampa [32] automatically synthesise replicated objects by coordinating unsafe operations. CISE [21] provides an automated proof rule to verify whether invariants are preserved on replicated databases under a given replication protocol. To ensure invariant safety, CISE coordinates conflicting operations. Quelea [42] determines the weakest consistency model that upholds invariants by statically analysing contracts to assign the most efficient and sound consistency level to each operation. Similarly, Q9 [25] identifies the weakest consistency model needed to prevent invariant violations. Indigo [7] gives programmers more control by allowing them to either coordinate unsafe operations or automatically repair invariants post-facto. It also introduces a multi-level mask reservation mechanism, using fine-grained locking to prevent specific conflicting operations. ECROs [18] reduce synchronisation by reordering conflicting operations, but it still relies on coordination when reordering is impossible. LoRe [22] automatically verifies developer-supplied safety properties for local-first applications, resolving invariant violations due to concurrency by adding coordination logic as needed. Coordination-based approaches optimise synchronisation but remain conservative, since they still rely on coordination to uphold invariants. In contrast, we propose a coordination-free alternative with the No-Op approach, which eliminates the need for synchronisation entirely, offering a novel and efficient solution for maintaining consistency in replicated systems.

Reservation-based approaches. Reservation-based approaches [36, 8, 7, 33, 38, 41] reduce coordination costs by moving synchronisation outside the execution path of operations. By reserving the rights to perform specific operations in advance, replicas can execute operations safely without requiring immediate coordination. However, these mechanisms assume the necessary rights are available at the local replica. Otherwise, coordination is needed to acquire rights from other replicas or operations are aborted. Escrow transactions [36] apply this concept to enforce numerical invariants. The difference between the current value of a numerical variable and its bound determines the number of rights distributed among replicas. The bounded counter CRDT [8] builds on escrow transactions by implementing this mechanism asynchronously to uphold numerical invariants. Partition lock reservation [7] extends these ideas to enforce linear resource constraints. It allows replicas to reserve rights over non-overlapping value intervals, enabling operations within these intervals to execute locally without coordination. However, operations that require values outside the reserved intervals must synchronise with other replicas. Although these mechanisms maximise concurrency for operations that might otherwise violate invariants, they still rely on locks or coordination when local rights are exhausted. In our work, we incorporate escrow techniques specifically for numerical invariants, as the No-Op approach can be overly conservative in such cases. By pre-allocating execution rights, replicas can safely execute concurrent operations without unnecessary No-Ops, ensuring safety. This integration reduces the constraints of the No-Op model for numerical invariants while preserving its coordination-free nature for other invariants.

Coordination-free approaches. Coordination-free approaches maintain invariants by transforming non-$\mathcal{I}$-confluent operations into $\mathcal{I}$-confluent operations. Existing approaches, as well as our approach, do so by introducing conflict resolution mechanisms. IPA [6] resolves conflicts by extending conflicting operations with updates that ensure invariant preservation. It leverages static analysis to identify conflicts and suggests to programmers conflict

resolution policies, enabling them to choose the policy that best suits their application. Compared to IPA, our No-Op approach introduces two key differences. First, while IPA resolves conflicts by adding updates, our approach discards conflicting operations. Second, IPA relies on CRDTs for convergence, whereas our approach natively supports convergence and invariant preservation without additional structures. Antidote SQL [35] enforces referential integrity through resolution policies. However, these policies are limited to referential integrity and cannot be extended to other invariants. In contrast, as shown in Section 5.2, our approach provides broader applicability, enabling the preservation of a wider range of invariants.

8 Conclusion

This paper introduced a coordination-free consistency framework that unifies data convergence and invariant preservation through the novel No-Op approach. Conflicts are resolved by designating one operation in each conflict as a no-side-effect operation. By transforming non-preferred operations into no-side-effect operations, the No-Op approach ensures consistency and safety across all replicas even under concurrent updates, as established by formal proofs. To the best of our knowledge, this is the first unified approach that ensures strong convergence and invariant preservation while eliminating the need for synchronisation.

Our approach allows programmers to define custom conflict resolution policies tailored to the specific semantics of their applications. Additionally, it eliminates the need to design and implement new data structures to support these policies, thereby simplifying the development of highly available distributed systems. We implemented the framework in the VeriFx programming language to demonstrate its practicality, leveraging its automated proof engine to validate each data-type implementation. This implementation confirmed the manual proofs of our approach, bridging the gap between theoretical guarantees of consistency and real-world applicability. Additionally, we showcased its ability to encode popular CRDTs and enforce a wide range of invariants, including referential integrity. Finally, we demonstrate our approach for developing real-world applications, such as the geo-distributed RUBiS system.

While the No-Op approach is coordination-free, it can sometimes unnecessarily transform operations into No-Ops. To address this challenge, we propose to integrate escrow techniques to minimise superfluous No-Ops and leverage policy graphs to identify inefficiencies.

References

1 Daniel Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer*, 45(2):37–42, February 2012. doi:10.1109/MC.2012.33.

2 Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: Definitions, implementation, and programming. *Distrib. Comput.*, 9(1):37–49, March 1995. doi:10.1007/BF01784241.

3 Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. Blazes: Coordination analysis and placement for distributed programs. *ACM Trans. Database Syst.*, 42(4), October 2017. doi:10.1145/3110214.

4 Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, November 2014. doi:10.14778/2735508.2735509.

5 Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January*

9-12, 2011, Online Proceedings, pages 223–234. www.cidrdb.org, 2011. URL: http://cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf.

6 Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno Preguiça. Ipa: invariant-preserving applications for weakly consistent replicated databases. *Proc. VLDB Endow.*, 12(4):404–418, December 2018. doi:10.14778/3297753.3297760.

7 Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2741948.2741972.

8 Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno M. Preguiça. Extending eventually consistent cloud databases for enforcing numeric invariants. In *34th IEEE Symposium on Reliable Distributed Systems, SRDS 2015, Montreal, QC, Canada, September 28 - October 1, 2015*, pages 31–36. IEEE Computer Society, 2015. doi:10.1109/SRDS.2015.32.

9 Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Pure operation-based replicated data types. *CoRR*, abs/1710.04469, 2017. arXiv:1710.04469.

10 Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 1–10. ACM Press, 1995. doi:10.1145/223784.223785.

11 Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

12 Dina Borrego, Nuno Preguiça, Elisa Gonzalez Boix, and Carla Ferreira. Ensuring convergence and invariants without coordination (appendix), 2025. URL: https://soft.vub.ac.be/Publications/2025/vub-tr-soft-25-02.pdf.

13 Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud types for eventual consistency. In James Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 283–307. Springer, 2012. doi:10.1007/978-3-642-31057-7_14.

14 Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of EJB applications. In Mamdouh Ibrahim and Satoshi Matsuoka, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002*, pages 246–261. ACM, 2002. doi:10.1145/582419.582443.

15 Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008. doi:10.14778/1454159.1454167.

16 James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), August 2013. doi:10.1145/2491245.

17 Kevin De Porre, Carla Ferreira, and Elisa Gonzalez Boix. VeriFx: Correct Replicated Data Types for the Masses. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:45, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2023.9.

18 Kevin De Porre, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix. ECROs: Building global scale systems from sequential code. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi:10.1145/3485484.

19 Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1294261.1294281.

20 Hector Garcia-Molina and Kenneth Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, December 1987. doi:10.1145/38714.38742.

21 Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 371–384, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2837614.2837625.

22 Julian Haas, Ragnar Mogk, Elena Yanakieva, Annette Bieniusa, and Mira Mezini. Lore: A programming model for verifiably safe local-first software. *ACM Trans. Program. Lang. Syst.*, 46(1), January 2024. doi:10.1145/3633769.

23 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. doi:10.1145/78969.78972.

24 Farzin Houshmand and Mohsen Lesani. Hamsaz: replication coordination analysis and synthesis. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290387.

25 Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. Safe replication through bounded concurrency verification. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276534.

26 Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. Mergeable replicated data types. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi:10.1145/3360580.

27 Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, pages 154–178, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3359591.3359737.

28 Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010. doi:10.1145/1773912.1773922.

29 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. doi:10.1145/359545.359563.

30 Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 265–278. USENIX Association, 2012. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li.

31 Cheng Li, Nuno M. Preguiça, and Rodrigo Rodrigues. Fine-grained consistency for geo-replicated systems. In Haryadi S. Gunawi and Benjamin C. Reed, editors, *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 359–372. USENIX Association, 2018. URL: https://www.usenix.org/conference/atc18/presentation/li-cheng.

32 Xiao Li, Farzin Houshmand, and Mohsen Lesani. Hampa: Solver-aided recency-aware replication. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I*, pages 324–349, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-53288-8_16.

33 Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency. In Ratul Mahajan and Ion Stoica, editors, *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 503–517. USENIX Association, 2014. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/liu_jed.

34 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2043556.2043593.

35 Pedro Lopes, João Sousa, Valter Balegas, Carla Ferreira, Sérgio Duarte, Annette Bieniusa, Rodrigo Rodrigues, and Nuno M. Preguiça. Antidote SQL: relaxed when possible, strict when necessary, 2019. arXiv:1902.03576.

36 Patrick E. O'Neil. The escrow transactional method. *ACM Trans. Database Syst.*, 11(4):405–430, December 1986. doi:10.1145/7239.7265.

37 Nuno M. Preguiça. Conflict-free replicated data types: An overview. *CoRR*, abs/1806.10254, 2018. arXiv:1806.10254.

38 Nuno M. Preguiça, José Legatheaux Martins, Miguel Cunha, and Henrique João L. Domingos. Reservations for conflict avoidance in a mobile database system. In Daniel P. Siewiorek, Mary Baker, and Robert T. Morris, editors, *Proceedings of the First International Conference on Mobile Systems, Applications, and Services, MobiSys 2003, San Francisco, CA, USA, May 5-8, 2003*, pages 43–56. USENIX, 2003. doi:10.1145/1066116.1189038.

39 Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2011. doi:10.1007/978-3-642-24550-3_29.

40 Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical report, INRIA, 2011.

41 Liuba Shrira, Hong Tian, and Douglas B. Terry. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In Valérie Issarny and Richard E. Schantz, editors, *Middleware 2008, ACM/IFIP/USENIX 9th International Middleware Conference, Leuven, Belgium, December 1-5, 2008, Proceedings*, volume 5346 of *Lecture Notes in Computer Science*, pages 42–61. Springer, 2008. doi:10.1007/978-3-540-89856-6_3.

42 KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 413–424, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2737924.2737981.

43 Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2043556.2043592.

44 D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, 29(5):172–182, December 1995. doi:10.1145/224057.224070.

45 Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009. doi:10.1145/1435417.1435432.

46 Michael Whittaker and Joseph M. Hellerstein. Interactive checks for coordination avoidance. *Proc. VLDB Endow.*, 12(1):14–27, September 2018. doi:10.14778/3275536.3275538.

Compositional Bug Detection for Internally Unsafe Libraries

A Logical Approach to Type Unsoundness

Pedro Carrott
Imperial College London, UK

Sacha-Élie Ayoun
Imperial College London, UK

Azalea Raad
Imperial College London, UK

Abstract

Rust is a modern programming language marrying the best of language design by offering fine-grained control over system resources (thus enabling efficient implementations), while simultaneously ensuring memory safety and data-race freedom. However, ensuring type safety in *internally unsafe* libraries remains an important and non-trivial challenge, where unsafe features enable low-level control but can introduce undefined behaviour (UB). Existing works on reasoning about unsafe code either use *verification* techniques to show *correctness* (i.e. safety, useful only when the unsafe code is indeed correct), or use *bug detection* techniques to show *incorrectness* (i.e. unsafety) but fail to do so *automatically* (to avoid developer burden), *compositionally* (to support libraries without a main function), *soundly* (without false positives) and *generally* (rather than applying to specific patterns).

We close this gap by developing RUXt, a *fully automatic*, *compositional* bug detection technique for detecting UB in internally unsafe Rust libraries with a formal *inadequacy* theorem (formalised and proven in Rocq) providing a *no-false-positives guarantee*. RUXt leverages the Rust type system to under-approximate the set of safe values for user-defined types and detect safety violations without requiring manual annotations by the user. By encoding well-typed values in Rust as separation logic assertions, RUXt enables compositional reasoning about types to refute unsound type signatures and detect UB. The inadequacy theorem of RUXt ensures that each UB detected by RUXt in an internally unsafe library is a true safety violation that can be triggered by a *safe* program, i.e. one that solely interacts with the safe API of the library. To this end, when RUXt identifies a UB, it additionally produces a *safe program* witnessing the UB. In addition, we develop a prototype implementation in OCaml that can analyse programs written in a small model of Rust, and apply it to several case studies, showcasing its ability to detect true safety violations.

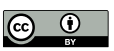
2012 ACM Subject Classification Theory of computation → Program analysis; Theory of computation → Separation logic; Theory of computation → Programming logic; Theory of computation → Automated reasoning; Theory of computation → Program specifications; Software and its engineering → General programming languages

Keywords and phrases Rust, bug detection, static analysis, program logics, under-approximation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.5

Supplementary Material *Software (ECOOP 2025 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.11.2.9`

Funding This project was sponsored by the Defense Advanced Research Projects Agency, (DARPA), Information Innovation Office (I2O), Program BAA HR001124S0003, under Cooperative Agreement No. HR00112420359. Disclaimer: The content of the information does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred. Raad is further supported by the UKRI Future Leaders Fellowship MR/V024299/1, by the EPSRC grant EP/X037029/1 and by VeTSS.



© Pedro Carrott, Sacha-Élie Ayoun, and Azalea Raad;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

Programming languages such as C and C++ provide developers with fine-grained control over system resources, thereby enabling efficient implementations, albeit at the cost of *safety*. By contrast, modern languages prioritise safety through high-level abstractions, compromising on low-level control. A long-standing goal in programming language design has been to bridge this divide and create a language that delivers both safety features and control over system resources. The Rust language [27] is the most significant industrial effort toward achieving this goal to date. Rust enables zero-cost abstractions for common systems programming patterns and operates without relying on a garbage collector. Moreover, Rust offers the safety features of high-level languages, ensuring *memory safety* and *data-race freedom*, thus guaranteeing the absence of *undefined behaviours* (UB).

To do this, Rust uses an ownership-based type system, enforcing a strict discipline that prevents mutation of aliased data. This discipline, *enforced statically* at compile time, ensures the absence of UB in Rust programs. Unfortunately, the Rust type system can be overly conservative at times, making it too restrictive to implement basic low-level data structures such as `Vec`, or synchronisation mechanisms such as `Mutex`. To circumvent this, Rust provides *unsafe* blocks and functions that unlock "unsafe superpowers" to bypass the Rust compiler's safety checks. Using unsafe code, developers can implement, for instance, the aforementioned `Vec` data structure, providing a *safe encapsulation* of the unsafe code. That is, using the safe `Vec` API, developers should not be able to trigger any undefined behaviour.

Effectively, defining such an *internally unsafe* data structure *extends* the Rust type system with a new type whose safety is not monitored by the compiler. That is, while the compiler *enforces* the safety guarantees for safe code, in the case of internally unsafe code it simply *assumes* them to hold, and the onus is on the library developer to ensure their safety.

Ensuring the above, however, is error-prone and far from straightforward, as evidenced by the subtle bugs discovered in the literature [3, 4, 17]. This underlines the need for formal methods and rigorous techniques for reasoning about unsafe code. Indeed, there has been a large body of work on reasoning about unsafe code, falling into one of two main categories (see §7 for a comprehensive discussion): **(i)** *verification* techniques aimed at showing the *correctness* (safety) of unsafe code [10, 1, 7, 23]; or **(ii)** *bug detection* techniques aimed at showing the *incorrectness* (unsafety) of unsafe code [2, 35, 5]. Specifically, the approaches in i are useful when the unsafe code under consideration is correct and they guarantee the absence of *false negatives* (i.e. missed bugs). However, they require extensive annotation efforts from the user, they simply fail when the code is indeed unsafe, and they cannot be used to identify the causes of safety violations. By contrast, the approaches in ii focus on detecting the sources of safety violations.

A bug detection technique (in category ii) should *ideally* (1) be *fully automatic* ("push-button") and require no user input or annotations – this ensures that the technique can be used out of the box, without imposing a learning curve on the user or requiring a background in formal methods; (2) only detect *true positives* (genuine safety violations) without any *false positives* (spurious reports of safety violations); (3) be applicable to a wide range of scenarios rather than specific patterns; and (4) be *compositional* (i.e. support analysing incomplete programs rather than accepting only full programs with a main function) to ensure its scalability and thus wide applicability to large codebases.

Unfortunately, however, no existing bug detection technique meets all above criteria. Specifically, Rudra [2], Kani [35] and TraitInv [5] all suffer from false positives. Moreover, while Rudra and TraitInv are automatic, Kani is *not automatic* and relies on *user-supplied sym-*

bolic tests for checking the desired properties. It only supports first-order, non-compositional properties (i.e. not ownership properties) and cannot reason about type safety. On the other hand, Rudra and TraitInv have limited applicability: Rudra uses linter-like *heuristics* to detect three specific patterns that are *likely* to be safety violations, while TraitInv can only detect violations of simple, first-order properties in the implementations of six standard *traits*, e.g. that a partial equality function is transitive and symmetric.

Our approach: RUXt. Our main contribution bridges this gap: RUXt is a *fully automatic*, *compositional* tool for detecting UB in (internally unsafe) Rust libraries, accompanied by a formal *true-positives* theorem proven in Rocq, guaranteeing that all reported UB are true safety violations.

To do this, we leverage the Rust type system: the type signature of a Rust function *is* indeed a safety specification/annotation provided by the developer (in contrast to type annotations in languages such as C). To this end, we draw inspiration from the semantic model of Rust types as formalised in RustBelt [19]. Specifically, we focus on RUXtBelt, a large fragment of the RustBelt model, enhanced with a clear account of errors necessary for bug detection. As in RustBelt, references in RUXtBelt do not have a different semantics from raw pointers, allowing us to maintain a simple formalism by avoiding special aliasing semantics such as Stacked Borrows [18] or Tree Borrows [37]. As our focus here is bug detection rather than verification, we can safely forgo such semantics, at the cost of missing some safety violations, without compromising on our true-positives result.

As in RustBelt, in RUXtBelt we use *separation logic* [14] and represent well-typed values as resources, enabling *compositional* reasoning about types. This way, given a library that introduces a new type `T`, we can reason about it in isolation and independently of a concrete client by considering only the `T` *type space*, namely the set of all allowed (safe) values in `T`. To ensure type soundness, the `T` type space *disallows* values that lead to UB, and we can find such UB by solving a *reachability* problem: we can explore the `T` type space to show that a set of values modeled by a separation logic assertion are reachable only through *safe* code. Specifically, given a function `fn f(x: U) -> T`, all values reachable by executing `f` *must* belong to the `T` type space, as long as the input belongs to the `U` type space. On the other hand, if we can show that UB is reachable using inputs that *must* belong to the `U` type space, then we have found a safety violation. By iteratively exploring and enlarging our knowledge of the type spaces through symbolically executing functions from the library, we can reliably detect type safety bugs fully automatically.

Our reachability analysis introduces the notion of *type subvariants*, namely a *subset* of the `T` type space, thus *under-approximating* the `T` type space. We obtain this under-approximation by leveraging *incorrectness separation logic* (ISL) [29, 31] to derive reachable values and infer type subvariants. We thus develop RISL (Rust incorrectness separation logic, pronounced *rizzle*), a compositional program logic for detecting UB and show that it is *sound* against our RUXtBelt model. RISL is inspired by ISL and similarly enjoys a *true-positives* theorem, meaning that all type safety bugs found by RISL are indeed true safety violations.

We then develop RUXt as a symbolic execution engine that operates on RISL assertions. RUXt automates the process of deriving RISL triples (and thus detecting UB). We prove that RUXt enjoys a *no-false-positives* guarantee through our *inadequacy theorem*: every UB found is a true type unsoundness in the library. That is, when RUXt detects UB, it guarantees that it is possible to construct a trace of safe calls to the library witnessing the UB. This is a direct consequence of *under-approximating* type spaces.

```rust
struct Even { val: i32 }

fn zero() -> Even {
    Even { val: 0 }
}
unsafe fn succ(x: Even) -> Even {
    Even { val: x.val + 1 }
}
fn next(x: Even) -> Even {
    unsafe { succ(succ(x)) }
}
fn noop(x: Even) -> () {
    if x.val % 2 != 0 {
        unsafe { *(0 as *mut i32) = 1 }
    }
}
```

```c
typedef struct { int val } Even;

Even zero() {
    return (Even){.val = 0};
}
Even succ(Even x) {
    return (Even){.val = x.val + 1};
}
Even next(Even x) {
    return succ(succ(x));
}
void noop(Even x) {
    if (x.val % 2 != 0) {
        *((int *)0) = 1;
    }
}
```

Figure 1 Library for the `Even` type implemented in Rust (left), alongside its C analogue (right).

The underlying theory behind RUXt and the proofs of all our theorems are fully mechanised in Rocq. We also develop a prototype implementation of RUXt in OCaml, and we conduct three small case studies, showing that it can detect subtle safety violations resulting from the interaction of several independently-safe functions of a library.

Outline. The rest of this article is organised as follows. We present an intuitive account of our contributions in §2. In §3 we describe the RUXt algorithm and our prototype implementation in OCaml, and provide an informal argument for why every UB detected by RUXt is indeed a safety violation that can always be triggered. In §4 we present several examples of UB found by RUXt. In §5 we describe our RISL logic for detecting UB. In §6 we present our RUXtBelt model, show that RISL is sound with respect to RUXtBelt and then present our inadequacy theorem *formally* showing that RUXt enjoys a guarantee of no false positives. Finally, we discuss related work and conclude in §7.

2 Overview

We present an intuitive account of our contributions through an example inspired by a case study from RefinedRust [10], as shown in Fig. 1 (left). We define a new type in Rust, `Even`, as an abstraction of even integers, along with a *safe* API for interacting with values of this type. We discuss the specificities and challenges of creating safe abstractions in Rust by showing an analogous C implementation of `Even` and highlighting the differences. Using this illustrative example, we first explain *why* Rust enables the detection of a new class of bugs, thanks to its more expressive type system, and then demonstrate *how* we can leverage Rust types to detect such bugs automatically.

2.1 Type Safety Bugs

We implement the `Even` type as a `struct` encapsulating an `i32` value (a 32-bit integer) in the `val` field. We also implement a library for interacting with `Even` through an API comprising three *safe* functions: (1) `zero` encapsulates 0 as the base value for `Even` and returns it; given an `Even` value, (2) `next` returns a new `Even` value encapsulating the next even integer, and

(3) `noop` writes to an arbitrary pointer if the encapsulated integer is odd. The `zero` and `next` functions encode constructors for well-formed `Even` values: every `Even` value they return encapsulates an integer that is guaranteed to be even. As a result, `noop` effectively performs no operations on safe inputs: the encapsulated integer is never odd by construction. Were this not the case, the function would attempt to write to an arbitrary memory address, resulting in undefined behaviour (UB). Note that `noop` is typed as a *safe* function, even though it is implemented with *unsafe* code; the `unsafe` block is needed as manipulating raw pointers is, in general, unsafe. In this example, however, our usage of unsafe features is safely encapsulated by ensuring that we never perform the unsafe write operation for safe inputs and thus we never observe UB.

The library implementation also includes the `succ` function which we have not yet mentioned. Given an `Even` value as input, `succ` increments the encapsulated integer and returns the result as a value of type `Even`. This behaviour clearly breaks the "evenness" abstraction that we desire for `Even`, and thus `succ` is marked as unsafe. This unsafe function is only used as an auxiliary function in the implementation of `next`, where it is called twice within an unsafe block, ensuring that its usage is safely encapsulated. Although `succ` itself does not contain any unsafe code, it allows odd integers to be encapsulated as `Even` values, breaking the requirements for the safety of `noop`. Therefore, defining `succ` as a safe function invalidates the safety of `noop`, meaning that both functions *cannot co-exist* as safe functions for the library to be *type-safe*.

By maintaining this boundary between safe and unsafe code, one can extend the language with new types that carry rich information besides structural layouts. For instance, `noop` restricts the `val` field of `Even` values to be even, since this is the only way of ensuring `noop` is safe. In C, this kind of information is unattainable as usage of unsafe features is unrestricted and thus safety cannot be expressed programmatically. The C analogue of our Rust library in Fig. 1 (right) exemplifies this: in `noop` the unsafe write is no longer wrapped in an unsafe block. Since we cannot distinguish safe functions from unsafe ones in C, one could assume that all functions are safe by default and apply the same reasoning as in Rust, but we have already shown that `succ` and `noop` cannot be simultaneously safe. Therefore, one must assume that all C functions are unsafe by default, meaning that we cannot leverage the type of `noop` to obtain information about the `Even` type.

Effectively, type signatures in Rust, including the `unsafe` keyword, are *specifications* written by the developer. In particular, safe functions guarantee the *absence* of UB for safe inputs, and we can invalidate such a specification by showing the *presence* of UB for such inputs, hence showing that the library violates safety. Functions such as `noop` and `succ` (if both are declared safe) constitute such a safety violation. Naturally, we refer to these as *type safety bugs*. These bugs are not detectable in C: unlike in Rust, analysis tools can only rely on the structural layout of data and not on the implicit *meaning* that programmers attribute to their programs.

2.2 Finding Real Bugs in Rust Libraries

The state-of-the-art Infer:Pulse analyser [31, 24] classifies memory safety bugs into two categories: (1) **manifest:** a bug that occurs in all calling contexts; and (2) **latent:** a bug that occurs only in specific calling contexts. Infer:Pulse always reports manifest bugs as they are *context-independent*: if a function *always* exhibits UB, then it is not safe to include it in a library. By contrast, a latent bug is only reported if the calling context that triggers it is reachable from a program entry point, e.g. the `main` function. In other words, Infer:Pulse cannot report *context-dependent* bugs as real bugs in libraries, relying on specific codebases to report real bugs that are latent.

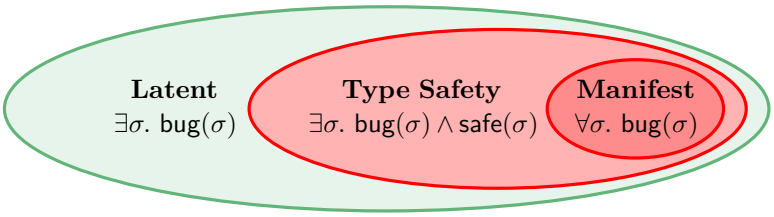


Figure 2 Taxonomy of bug categories. The variable σ represents a calling context, where $\mathsf{bug}(\sigma)$ expresses that UB is observable in σ and $\mathsf{safe}(\sigma)$ expresses that σ is obtained solely from safe code. Every type safety bug is a latent bug. Every manifest bug is a type safety bug: if every context triggers a bug, then the bug occurs in both safe and unsafe contexts. Manifest and type safety bugs are highlighted in red as they are always true bugs, in contrast to latent bugs.

We show that, in Rust, context-dependent bugs can be reported as real bugs as long as we restrict the conditions on calling contexts. In particular, type safety bugs are a class of context-dependent bugs where the calling context is obtained by executing only safe code, *including calls to internally unsafe functions*. For example, by making `succ` safe, we can construct an apparently safe program that encapsulates an odd integer as an `Even` value, resulting in a context where `noop` exhibits UB. Although this type safety bug could be reported as a real bug in the Rust implementation, Infer:Pulse would merely identify it as a latent bug in the C analogue. That is, as type safety is not expressible in C, it is impossible to restrict the calling context as in Rust.

As illustrated in Fig. 2, reasoning about type safety allows us to report more bugs: certain bugs that would be dismissed as latent in C can be reported as type safety bugs in Rust. Indeed, it is possible to detect *strictly more* real bugs than what Infer:Pulse currently reports, since every manifest bug is also a type safety bug. This opens the door to a new realm of analysis techniques for sound bug detection, which is the main contribution of our work.

2.3 Reasoning about Type Safety

Type safety, also known as *type soundness*, is a fundamental property of safe programming languages: well-typed programs (i.e. programs that type-check) will always exhibit well-defined behaviour (no UB). Conversely, one can think of a type safety bug as *type unsoundness*.

The RustBelt project [19] provided a semantic model of Rust types, mechanising and proving the type soundness of the *safe* fragments of Rust (which can be syntactically type-checked). RustBelt further provides a framework for *verifying the type soundness* of internally unsafe libraries (these are not covered by the type soundness proof), provided of course that they are indeed sound. While we focus on *detecting type unsoundness* in internally unsafe libraries, we introduce one of the main ideas behind RustBelt, from which we draw inspiration: *separation logic invariants* [14].

Safe values in RustBelt are interpreted as resources in separation logic, enabling *compositional* reasoning about types. This means that given a library that introduces a new type, we can reason about it in isolation and independently of a concrete client. To ensure type soundness is preserved when introducing a new type, RustBelt relies on user-defined *type invariants* that *over-approximate* the set of allowed values for a given type, which we refer to as *type spaces*. That is, while type spaces define the *exact* set of values inhabiting a given type, type invariants over-approximate types and thus a type invariant is a superset of its associated type space, as illustrated in Fig. 3. Hereafter, we refer to a value in a type space as a *safe value*. Type soundness guarantees that a safe program moves between safe values in

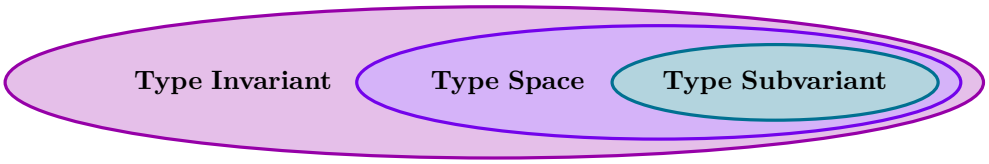


Figure 3 Type spaces represent the *exact* set of safe values of a given type; type invariants *over-approximate* type spaces; type subvariants *under-approximate* type spaces.

type spaces. For example, since the `zero` function is safe, its output must necessarily belong to the `Even` type space. Similarly, given a value in the `Even` type space as input, the `next` function produces another value in the `Even` type space. That is, as both `zero` and `next` are safe functions, every even integer must inhabit the `Even` type space.

2.4 A Logical Approach to Type Unsoundness

We present RUXt, a *compositional*, *fully automated* analysis algorithm and tool for detecting *type safety bugs* in (internally unsafe) Rust libraries. RUXt does not require explicit definitions of type invariants from the user. Instead, it reduces the problem of detecting type unsoundness to a reachability problem. By leveraging type signatures and incorrectness separation logic [31], it automatically infers a set of safely reachable values for each type. The set of reachable values detected by our analysis constitutes an *under-approximation* of the type space, which we call a *type subvariant*. Specifically, we grow type subvariants by iteratively performing under-approximate compositional symbolic execution (à la Infer:Pulse [24] or Gillian [25]) on the library functions. In turn, if the function `f(x: T)` can reach UB from an input `x` that belongs to a subvariant of `T`, then the library must be type-unsound. The relation between type spaces, type invariants and type subvariants is represented pictorially in Fig. 3.

Importantly, RUXt enjoys a guarantee of *no false positives*: every bug found corresponds to a true type unsoundness in the library. That is, when RUXt detects UB, it guarantees that it is possible to construct a trace of syntactically-safe calls to the library witnessing the UB. Furthermore, exploring type spaces is drastically more efficient than exploring the space of all possible *traces*, as we only need to find the safe *values*; that is, while many traces may construct the same safe value, we can mitigate repeated work using each safe value only once as an input to a function execution.

RUXt's symbolic execution engine is based on the set of proof rules that we define for RISL, our incorrectness separation logic for Rust, underpinned by our model of Rust (RUXtBelt), presented in §5. RISL allows us to derive an under-approximate postcondition for each safe function, given initial type subvariants for each of its inputs. In turn, we can use these postconditions to grow the type subvariants for the output types of the functions.

For instance, take the `Even` library defined in Fig. 1 (left). We demonstrate how RUXt can detect type unsoundness by describing the analysis step by step. Remember that the unsoundness arises when the `succ` function is marked as safe.

1. Initially, the subvariant for the `Even` type is empty (no safe values are known).
2. Because `next`, `noop`, and `succ` receive `Even` values as input, we cannot symbolically execute them without first inferring a non-empty subvariant for `Even`. However, `zero` does not require any input, and we can therefore symbolically execute starting from the `emp` assertion, capturing an empty heap (executing `zero` should not need any resources). We can then immediately infer that the output `Even { val: 0 }` must be a safe value, and we thus add it to our subvariant for `Even`.

3. We now have a non-empty subvariant for `Even` and can symbolically execute `next` from this subvariant, learning that the value `Even { val: 2 }` is also safe. Our newly-grown subvariant contains both `Even { val: 0 }` and `Even { val: 2 }`.
4. If we now symbolically execute `succ` from this subvariant, we learn that the values `Even { val: 1 }` and `Even { val: 3 }` are also safe.
5. Finally, if we symbolically execute `noop` from this subvariant, we can detect UB when dereferencing a null-like pointer, as the `Even` type space now includes non-even values.

As soon as we perform the last step and detect the UB, we can report the type unsoundness in the library, with a guarantee that the bug is real and can be triggered by a syntactically-safe program. Specifically, RUXt returns the following program as a *witness* of the UB:

```
fn exhibit_ub() {
    let x = {
        let x = {
            zero()  // witness of Even { val : 0 }
        };
        succ(x)  // witness of Even { val : 1 }
    };
    noop(x);  // witness of UB
}
```

If, however, the `succ` function is correctly marked as unsafe, our analysis does not use it to grow the subvariant for `Even` and never learns that the `Even` type space contains non-even values. In this case, the library is sound, and we do not report any type unsoundness.

Note that, in this case, we would provide a notion of *fuel* to RUXt to prevent it from looping indefinitely, capping its number of iterations. In general, doing so is always sound: in the worst case we fail to detect bugs, but we would never report false positives.

3 RUXt: A Compositional Analysis for Type Safety Refutation

We present our algorithm for compositionally and automatically detecting type unsoundness in internally unsafe Rust libraries. To this end, we introduce the RUXt programming language (§3.1), formally describe type spaces and subvariants (§3.2) and present the RUXt algorithm and discuss our prototype implementation in OCaml (§3.3).

3.1 RUXt Programming Language

Our programming language, λ_{RUXt}, is inspired by those of both RustBelt (λ_{Rust} [19]) and ISL [31]. We consider a heap-manipulating *expression* language in the style of λ_{Rust}, similar to the *command* language in ISL.

$$
\begin{array}{l}
\textsc{Term} \ni t ::= v \mid x \in \textsc{Var} \qquad \textsc{Val} \ni v ::= z \in \mathbb{Z} \mid b \in \mathbb{B} \mid l \in \textsc{Loc} \mid () \\
\textsc{Pure} \ni p ::= t \mid \ominus p \mid p_1 \oplus p_2 \qquad \ominus ::= -_{\mathbb{Z}} \mid \neg_{\mathbb{B}} \qquad \oplus ::= +_{\mathbb{Z}} \mid \leq_{\mathbb{Z}} \\
\textsc{Expr} \ni e ::= p \mid \mathbf{alloc} \mid \mathbf{free}(t) \mid t_1 \leftarrow t_2 \mid {}^*t \mid f(\overline{t}) \mid \\
\qquad\qquad \mathbf{assume}(t) \mid \mathbf{error} \mid e_1 + e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2
\end{array}
$$

The λ_{RUXt} (program) expressions are defined by the e grammar above and include *pure expressions*, standard heap-manipulating constructs for allocation (**alloc**), deallocation (**free**(t)), storing (heap mutation, $t_1 \leftarrow t_2$) and loading (heap lookup, ${}^*t$), as well as function calls ($f(\overline{t})$, see below), assume expressions (**assume**(t)), explicit error expressions (**error**),

non-deterministic choice ($e_1 + e_2$) and let bindings (**let** $x = e_1$ **in** e_2). Pure expressions (p) have no side effects and comprise *terms* as well as unary/binary operations on terms. A term t is either a *value* or a *program variable* (x) meant to be substituted with concrete values. A value v may be an integer z, a boolean b, a heap location l or the unit value ($()$). Note that as heap-manipulating and assume expressions contain terms instead of pure expressions, unary and binary operations must first be evaluated through a let binding, creating a new variable.

Other standard language constructs such as sequential composition, reference allocation, assert expressions and deterministic conditionals can be encoded using the above:

$$
\begin{aligned}
e_1;\ e_2 &\triangleq \textbf{let}\ _ = e_1\ \textbf{in}\ e_2 \\
\textbf{ref}(t) &\triangleq \textbf{let}\ x = \textbf{alloc}\ \textbf{in}\ x \leftarrow t;\ x \\
\textbf{assert}(t) &\triangleq \textbf{let}\ x = \neg_{\mathbb{B}} t\ \textbf{in}\ \textbf{assume}(t) + (\textbf{assume}(x);\ \textbf{error}) \\
\textbf{if}\ t\ \textbf{then}\ e_1\ \textbf{else}\ e_2 &\triangleq \textbf{let}\ x = \neg_{\mathbb{B}} t\ \textbf{in}\ (\textbf{assume}(t);\ e_1) + (\textbf{assume}(x);\ e_2)
\end{aligned}
$$

Functions and Loops. As λ_{RUXt} supports function calls ($f(\overline{t})$), we omit loop expressions ($e^\star$), since the same behaviour can be obtained with recursive functions. Since λ_{RUXt} is not higher-order, functions must be externally provided as an *implementation context*, γ, mapping function identifiers to their concrete implementations. We write $f(\overline{x})\{e\} \in \gamma$ to denote that γ contains an implementation for f, where e is the f body and $\overline{x}$ are its arguments (i.e. free variables in e). For example, the `Even` library would correspond to an implementation context γ such that $\mathsf{dom}(\gamma) = \{\texttt{zero}, \texttt{succ}, \texttt{next}, \texttt{noop}\}$ and $\texttt{next}(x)\{$ **let** $x = \texttt{succ}(x)$ **in** $\texttt{succ}(x)$ $\} \in \gamma$.

Type signatures. A function implementation is annotated with a type signature, recorded in a *signature context*, Δ. Signature contexts contain type signatures for *safe* functions only, as they are the only ones required to preserve type safety. Therefore, unsafe functions are excluded from signature contexts, even though they still appear in implementation contexts. We write $f(\overline{\tau}) \to \tau \in \Delta$ to denote that Δ contains a signature for f where $\overline{\tau}$ are its input types and τ is its output type. In the case of `Even` with γ described above, we have $\mathsf{dom}(\Delta) = \mathsf{dom}(\gamma) \setminus \{\texttt{succ}\}$ and $\texttt{next}(\texttt{Even}) \to \texttt{Even} \in \Delta$. In essence, excluding `succ` from Δ means that we cannot leverage type information from `succ` to infer anything about `Even`.

3.2 Type Spaces and Subvariants

Recall that we leverage signature contexts to construct type subvariants by inferring safe values of the types in a library. To this end, we introduce the notion of a *summary* and define a type subvariant as a set of summaries. We track the subvariants associated with each library type through a *summary context*.

Summaries. A summary, ς, is of the form $\mathsf{Summary}(\lambda v.\ P \mid e)$, where e is a *safe program* (described below) that *witnesses* value v (i.e. executing e yields v) constrained by the separation logic (SL) *assertion* P. Rather than using *concrete* values, v is a *symbolic* value: it serves as a binder for P, allowing us to represent an infinite number of concrete values in a single assertion. We define assertions as follows:

$$
\begin{aligned}
\textsc{Asrt} \ni P, Q ::= {} & \llcorner \pi \lrcorner \mid \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \exists\, x.\ P \mid \\
& \mathsf{emp} \mid l \mapsto v \mid l \not\mapsto \mid l \mapsto_? \mid P * Q
\end{aligned}
$$

Our assertion language is standard and includes truth, falsity, conjunction, disjunction, implication and existential quantification. A purely logical proposition[1] π may be lifted to an assertion as $\llcorner \pi \lrcorner$. Our assertions also include standard SL assertions for describing an empty heap (emp), a single-cell heap ($l \mapsto v$, comprising a single location l that holds value v) and composite heaps via the separating conjunction $*$ ($P * Q$, describing a heap that can be decomposed into two parts, one described by P and another by Q). We further include the negative heap assertions from ISL, $l \not\mapsto$, and the unknown heap assertions $l \mapsto_?$, respectively describing a heap comprising a single location l that has been freed or is uninitialised.

For every summary $\mathsf{Summary}(- \mid e)$, e must be a *safe program*: a program e is safe if it is *syntactically* well-typed (i.e. e type-checks) and comprises solely a sequence of calls to the safe functions in the API of the current library. Furthermore, a safe program should require no initial resources to execute safely (i.e. should execute safely starting from emp).

A summary context Σ is a map from types to their subvariants (set of summaries). For example, the `zero` function always returns the value 0 in an empty heap, and thus when the summary context is given by Σ, then $\mathsf{Summary}(\lambda v.\ \llcorner v = 0 \lrcorner \mid \texttt{zero}()) \in \Sigma(\texttt{Even})$. Note that the `Even` type is particularly simple and does not make use of the heap. In §4, we describe a more complex linked-list example where the constraining assertion is not pure.

Summary inference. We infer summaries using compositional symbolic execution [24, 25]: starting with an empty summary context Σ, we iteratively execute *safe* functions from the library to extend Σ. Given a safe function f with return type τ, every output value of f must be a safe value of τ (i.e. in the τ subvariant), and thus we can extend $\Sigma(\tau)$ to include the output value. More concretely, let f be the safe function we consider at the current iteration, where f takes n arguments of types $\tau_1 \cdots \tau_n$ and has output type τ. We then proceed as follows:

1. For each input type τ_i, we pick one summary $\varsigma_i = \mathsf{Summary}(\lambda v_i.\ P_i \mid e_i)$ from $\Sigma(\tau_i)$. At this step we pick a function f only if we have at least one summary for each of its input types, which may not always be the case. The strategy for selecting f is immaterial for soundness of our approach, and thus we omit such details from our formalism. Nevertheless, each library typically has *constructor* functions which do not require an input, or require inputs of other types, for which our analysis would compute subvariants.
2. We combine $v_1 \cdots v_n$ into a list of symbolic values $\overline{v}$ to pass as arguments to f, and define a *precondition* $P \triangleq P_1 * \cdots * P_n$ for f. This yields a *safe context* for executing f: as the input values are safe by construction and f is safe, it *should* execute safely, returning a safe value of type τ. However, recall that this is not always the case when the library is internally unsafe.
3. We sequentially compose witness programs $e_1 \cdots e_n$, followed by a call to f on the associated values, resulting in the program $e \triangleq \textbf{let}\ x_i = e_1\ \textbf{in}\ \ldots \textbf{let}\ x_n = e_n\ \textbf{in}\ f(\overline{x})$. Note that as the function call is syntactically well-typed and every program e_i is a safe program, the composite program e is also safe by construction. Program e will serve as a witness program for the result of executing f on these safe input values, whether it be a safe execution or not.
4. As f may be *internally* unsafe, its execution may reach an error. To distinguish safe executions from unsafe ones, we record *termination (exit) conditions* as follows:

$$\textsc{Exit} \ni \epsilon ::= \mathsf{ok}(v) \mid \mathsf{err} \mid \mathsf{miss}(l)$$

[1] That is, any term definable in Rocq with type Prop, such as equality assertions or other similar relations.

TryRefute$(\gamma, \Delta, \Sigma) \to$ SummCtx + Expr

1: $(f, \overline{\varsigma}, \tau) \leftarrow$ pickCandidate(Δ, Σ)
2: $(\overline{v}, P, \overline{e}) \leftarrow$ constructContext$(\overline{\varsigma})$
3: $e \leftarrow$ constructWitness$(\overline{e}, f)$
4: **for** $[\epsilon : Q] \in$ execute$(\gamma, f(\overline{v}), P)$ **do**
5: **match** ϵ **with**
6: **case** ok(v)
7: $\varsigma \leftarrow$ Summary$(\lambda v.\ Q \mid e)$
8: $\Sigma \leftarrow \Sigma[\tau \mapsto \varsigma, \Sigma(\tau)]$
9: **case** err $\mid$ miss$(-)$
10: **return** e
11: **return** Σ

Summary contexts $\boxed{\gamma, \Delta \vdash \Sigma}$

R-Empty
$$\gamma, \Delta \vdash \emptyset$$

R-Infer
$$\frac{\gamma, \Delta \vdash \Sigma \quad \text{TryRefute}(\gamma, \Delta, \Sigma) = \Sigma'}{\gamma, \Delta \vdash \Sigma'}$$

Figure 4 The RUXt refutation procedure (left) and rules for inferring well-formed summary contexts (right).

5. We analyse the resulting exit condition ϵ and proceed accordingly.
6. If ϵ is *successful*, denoted by ok(v), the return value v is a *symbolic* value (as the inputs are themselves symbolic) and represents multiple safe values of τ; therefore,
   7. the symbolic execution engine constrains v by returning a postcondition Q, whereby we construct a new summary $\varsigma =$ Summary$(\lambda v.\ Q \mid e)$, where e (constructed as described above) is the safe program that witnesses ς as reachable; and
   8. we extend the subvariant of τ in Σ with ς.

   In the future iterations, we can then use this summary as a safe value for functions that require an input of type τ. Note that we may re-execute previously executed functions with the new input summary Summary$(\lambda v.\ Q \mid e)$: UB may now be reachable from this new summary, even if it was not reachable for prior safe inputs.
9. Otherwise ϵ is *unsuccessful*, given by either **(i)** err: denoting a memory safety error (e.g. performing a double free); or **(ii)** miss(l): denoting an access to a location l for which we do not own sufficient resources (e.g. an out-of-bounds access). We elaborate further on the distinction between these two cases later in §5.
   10. Finally, we report the unsuccessful termination as a type unsoundness (safety violation), returning e as a witness program that triggers the unsoundness.

Note that we can soundly identify an unsuccessful exit as a true bug. Specifically, as we only consider safe functions with safe values (summaries) as input, we solely interact with the safe API of the library. As such, if a safe function does not terminate successfully on safe inputs, then the library API does not encapsulate its usage of unsafe code, resulting in UB and rendering the library type-unsound. In other words, by considering only safe contexts, we can identify latent bugs as type safety bugs thanks to the expressivity of the type system.

3.3 The RUXt Algorithm

We present RUXt, our compositional, fully automatic analysis for detecting safety violations in internally unsafe libraries. The RUXt algorithm comprises a meta-loop that calls the TryRefute procedure (Fig. 4, left) at each iteration. Each execution of TryRefute infers summaries for a safe function and updates Σ to some Σ', as discussed in §3.2, following the rules in Fig. 4 (right). This meta-loop is performed until a type unsoundness is found

or a predefined limit ("fuel") is reached. Although the search is not exhaustive and some bugs may go undetected, we prioritise the guarantee of true positives by only exploring an under-approximation of the possible interactions with the library.

Refutation procedure. The TryRefute procedure is parametrised by the current summary context, Σ, along with an implementation context γ (for function implementations) and a signature context Δ (for function type signatures), returning either the updated summary context or a witness program if a type unsoundness is found. Specifically, lines 1–10 in TryRefute correspond to steps 1–10 on page 10.

The pickCandidate procedure (line 1) selects a safe function f and its associated input summaries and output type (see step 1 on p. 10); that is, $f(\overline{\tau})\to\tau \in \Delta$ and $\forall i.\ \overline{\varsigma}_i \in \Sigma(\overline{\tau}_i)$. We then construct a safe context using constructContext (line 2) and a witness program using constructWitness (line 3), as we described previously (see steps 2 and 3 on p. 10, respectively). Given P, we then call execute (line 4) to symbolically execute the function call $f(\overline{v})$, which returns a *set* of postconditions (due to non-determinism). For each postcondition $[\epsilon : Q]$, we perform a case analysis (lines 5–10) on its exit condition ϵ, as discussed in steps 5–10 on p. 11. Finally, if no type unsoundness is found by the end of the meta-loop, the final summary context is returned (line 11).

Well-formed summary contexts. The judgement $\gamma, \Delta \vdash \Sigma$ (Fig. 4, right) denotes a *well-formed* summary context: every summary in Σ is reachable by interacting solely with the safe functions given in implementation context γ and signature context Δ. The R-Empty rule states that an empty summary context is always well-formed. The R-Infer rule allows a well-formed summary context to be extended: given a well-formed Σ, if TryRefute returns an updated summary context Σ', then Σ' is also well-formed. Since TryRefute maintains the previous summaries and only extends the summary context with new ones that are reachable from a safe function, well-formedness is preserved across iterations.

Soundness. The soundness of RUXt depends on the soundness of its underlying symbolic execution engine. Indeed, the execute procedure must derive a valid under-approximation of the values that are reachable by the program. To ensure this, we develop the RISL program logic (§5) and prove that it is sound with respect to our under-approximate semantics (§6).

The RISL soundness allows us to prove our main result, the *inadequacy theorem* (see Theorem 7): if TryRefute returns a witness program for a library, then the witness is a safe program for which UB is observable. The returned witness thus demonstrates that UB can emerge by interacting solely with the safe API of the library. In other words, we show that any violation reported by RUXt proves the existence of a true safety bug in the library.

Handling references. Until now, we have not discussed how RUXt handles *reference* types. Namely, given a Rust function `fn f(x: &mut T)`, the RUXt algorithm cannot infer anything about type `T` from `f` mutating `x`, since `T` is not the output type. Therefore, rather than perform the analysis directly on `f`, we define a *safe wrapper* `f_wrap` that receives and returns a value of type `T`, and run RUXt on `f_wrap` instead. The wrapper simply calls the original function and returns the mutated input: `fn f_wrap(mut x: T) -> T { f(&mut x); x }`. The intuition for this wrapper function is that any safe mutating operation should always re-establish the safety of its input value, and we can use this information to grow our subvariants. Indeed, since `f_wrap` type-checks, it *must* be safe and therefore our analysis can immediately be applied to the wrapper without further proof. We demonstrate this wrapper transformation through a linked list example in §4.

These wrappers can be constructed fully automatically directly in Rust and then type-checked by the compiler. Moreover, similar wrappers can also be defined for other reference types. For example, a function `fn f(&mut T) -> &mut U` that returns a mutable reference can be wrapped into a function that assigns an arbitrary value to the returned reference: `fn f_wrap(mut x: T, y: U) -> T { *(f(&mut x)) = y; x }`. Again, the intuition is that assigning an arbitrary value to the reference should not break the safety of the input structure. Shared references can be handled similarly by simply checking that *reading* from the reference does not trigger UB on safe inputs. Note that if a function receives or returns several references, we can apply these transformations to each reference separately and analyse each wrapper independently to grow our subvariants.

Even if the wrappers use mutable references in their body, the analysis can still be performed, since references are semantically equivalent to raw pointers in our model (similarly to RustBelt). However, RUXt is thus unable to detect specific classes of bugs related to *aliasing semantics* from stronger semantic models such as Stacked Borrows [18] or Tree Borrows [37]. Nevertheless, our approach remains sound as we are not compromising on our promise of true positives: all bugs reported by RUXt are true safety violations, even if other UBs allowed by Stacked Borrows or Tree Borrows remain undetected.

RUXt prototype implementation. We develop a prototype implementation of RUXt in OCaml. The execute procedure is implemented using the technique of *compositional symbolic execution for incorrectness reasoning* [25], a form of symbolic execution that allows reasoning about fragments of states that correspond to separation logic assertions. A similar technique is implemented in other tools for correctness and/or incorrectness reasoning such as VeriFast [15], Viper [28] or Gillian [26]. The engine was designed to follow the RISL proof rules (see §5), ensuring that, starting from a precondition P and executing a function body e, the engine constructs only postconditions $[\epsilon : Q]$ such that the triple $\vdash [P]\, e\, [\epsilon\colon Q]$ holds.

Furthermore, in the implementation, the analysis is initialised with valid subvariants for the primitive types such as integers and booleans. For instance, the initial subvariant for the `int` type is given by the single summary $\mathsf{Summary}(\lambda v.\ v \in \mathbb{Z} \mid v)$, where v represents all integers, as constrained by the assertion. In §4, we exemplify further the use of such symbolic summaries with a constructor function for the `Even` type that receives an integer as parameter.

Finally, since the algorithm presented in Fig. 4 may loop indefinitely, our implementation receives an initial fuel parameter to limit the number of iterations.

4 Case Studies

To demonstrate the feasibility and applicability of RUXt, we run our prototype implementation on three λ_{RUXt} case studies and show that we can automatically detect type unsoundness in ill-defined libraries. First, we target the unsoundness in our running example `Even` (§2), and next we consider an extension of `Even` with a more general constructor. Finally, we define a `List` library in λ_{RUXt} with a function that causes an unsoundness. Note that as our prototype does not currently construct witness programs, we omit these from the summaries shown in this section.

4.1 Even Integer

We re-implement the functions presented in Fig. 1 within our prototype. Our implementation successfully finds the unsoundness when `succ` is marked as safe, and accurately does not when it is marked as `unsafe`.

Even integer extension. The `Even` example in §2 is simplified and does not fully demonstrate the benefits of using symbolic execution, as all inferred values are concrete. To illustrate the power of symbolic execution, we replace the `zero` constructor with a new function `new` that takes an integer `x` and returns it if `x` is even; otherwise, it increments `x` and returns it.

```
fn new(x: int) -> Even {
  if (x % 2 = 0) { x }
  else { x + 1 }
}
```

Here, `int` is a base case: we know that the *exact* type space (i.e. neither over- nor under-approximate) of a value n of type `int` is "$n \in \mathbb{Z}$". We can use this type space as a subvariant to construct the input context for `new`. We then obtain the following subvariant for `Even` by symbolically executing `new` from that context, obtaining two branches and hence two summaries:

$$\Sigma(\texttt{Even}) = \{\mathsf{Summary}(\lambda n.\ n \in \mathbb{Z} \wedge n\%2 = 0), \mathsf{Summary}(\lambda n.\ n \in \mathbb{Z} \wedge (n-1)\%2 \neq 0)\}$$

This newly obtained subvariant covers exactly the entire type space of `Even`. Using a simple SMT check, we can then prove that the concrete value returned by `zero` is included in this subvariant. This allows us to remove (or never add) the concrete value as a subvariant, thereby reducing path explosion. The same approach could be used for other base cases that are "native" to the language (e.g. the slice or `Box` types), where the type spaces are known exactly and can be formulated as simple separation logic assertions.

4.2 Linked List

We define a simple linked list library, `List`, in λ_{RUXt}. Unlike `Even`, linked lists are real-world data structures that are infamously difficult to implement soundly in Rust. Within `List` we introduce a subtle unsoundness and show that our prototype can detect it automatically. The `List` library contains the following functions:

- `fn new() -> List`: creates a new empty list, a null pointer represented as the value 0;
- `fn push(l: List) -> List`: adds a newly allocated node to the front of (input list) `l`;
- `fn drop(l: List)`: iterates through `l` and frees each node until a null pointer is found;
- `fn cycle(l: List) -> List`: overrides the null pointer terminating `l` with a pointer to its first element, creating a *cycle* in the list.

In a real Rust implementation, `push` and `cycle` would receive a mutable reference to the list. However, as discussed in §3.3 (p. 12), the analysis can only be performed on wrappers. Since our prototype does not yet automate this transformation, we implement `push` and `cycle` as the actual wrappers. For example, we transform the Rust implementation (receiving a mutable reference to the list), `Rust_push`, by wrapping it in our `push` implementation, where `inline_Rust_push(&mut l)` denotes inlining the body of `Rust_push(&mut l)`: `fn push(mut l: List) -> List { inline_Rust_push(&mut l); l }`.

The first three functions (`new`, `push` and `drop`) are standard and exist in the `LinkedList` implementation of Rust standard library. However, the `cycle` function is not standard and indeed introduces unsoundness: dropping a cyclic linked list will lead to a use-after-free error the second time the iteration reaches a node in the cycle[2].

[2] An alternative implementation of `drop` which would check for cycles would not suffer from this issue. However, the current implementation of the `LinkedList` structure in the Rust standard library does not check for cycles when iterating.

RUXt can detect this unsoundness as follows.

1. At first, we can only pick the `new` function: it requires no arguments, while all other functions require an argument of type `List` for which we are yet to infer a subvariant. By executing `new`, we infer that the empty linked list – i.e. a `null` pointer – must be a safe value of type `List`. In other words, thus far we have inferred the type subvariant:

$$\Sigma(\texttt{List}) = \{\mathsf{Summary}(\lambda v.\ \llcorner v = 0 \lrcorner)\}$$

2. Later on, we pick `push` and execute it from our current `List` subvariant, learning a new subvariant for lists of size one:

$$\Sigma(\texttt{List}) = \{\mathsf{Summary}(\lambda v.\ \llcorner v = 0 \lrcorner), \mathsf{Summary}(\lambda l.\ l \mapsto 0)\}$$

3. Subsequently, we pick `cycle` and learn a new subvariant:

$$\Sigma(\texttt{List}) = \{\mathsf{Summary}(\lambda v.\ \llcorner v = 0 \lrcorner), \mathsf{Summary}(\lambda l.\ l \mapsto 0), \mathsf{Summary}(\lambda l.\ l \mapsto l)\}$$

4. Finally, we pick `drop`, and using our last subvariant we detect a use-after-free UB.

4.3 Key Takeaways

These examples highlights several key aspects of the kinds of unsoundness detected by RUXt:

- In Rust, functions such as `cycle` would *require* unsafe code, as safe Rust prevents the kind of aliasing required for a cyclic linked list. Importantly, reasoning about this aliasing is made substantially simpler thanks to separation logic.
- The `cycle` and `succ` functions are not inherently wrong. For instance, in the `List` case, the unsoundness arises from the interaction of `cycle` and `drop`; the library would be sound without either of these functions. Safety is, inherently, an inter-procedural property [16]. In our prototype, we also check that no unsoundness is found if either function is removed from the library, or if either is marked as unsafe.
- We do not need to explore all traces as we build subvariants. As demonstrated with the `zero` and `new` functions of the `Even` library, some subvariants might be removed from our Σ when a strictly larger subvariant is found. We believe that this compositional approach dramatically reduces path explosion issues in comparison to whole-program type-guided approaches such as Crabtree [34], described in more details in §7.

5 The RISL Program Logic

We now present RISL (Rust incorrectness separation logic), our under-approximate logic for deriving reachable states in RUXt. Our proof rules are standard and similar to those of ISL [31], with the caveat that λ_{RUXt} programs are not commands, but rather expressions that reduce to values on successful termination. Unlike ISL, RISL supports function calls and is the first separation logic to reason about missing resources (the miss exit condition).

5.1 Proof Rules

We present the RISL proof rules in Fig. 5. Intuitively, a RISL triple $\Gamma \vdash [P]\ e\ [\epsilon\colon Q]$ is interpreted similarly to ISL: every state in the postcondition Q is reachable by executing program e starting from some state in the precondition P, where ϵ is the exit condition, as described in §3.2 (p. 10). To reason about function *calls*, RISL allows reusing of derived specifications through a *specification context*, Γ, mapping each function to a list of *under-approximate* specifications. More concretely, $\left[(\overline{v})\ P \,\middle|\, \epsilon\colon Q\right] \in \Gamma(f)$ denotes that $[\epsilon : Q]$ is reachable by executing $f(\overline{v})$ starting from P (i.e. $\Gamma \vdash [P]\ f(\overline{v})\ [\epsilon\colon Q]$ holds).

Under-approximate triples $\boxed{\Gamma \vdash [P]\, e\, [\epsilon\colon Q]}$

S-Val
$\Gamma \vdash [\mathsf{emp}]\, v\, [\mathsf{ok}(v)\colon \mathsf{emp}]$

S-Assume
$\Gamma \vdash [\mathsf{emp}]\ \mathbf{assume}(\mathsf{true})\ [\mathsf{ok}\colon \mathsf{emp}]$

S-Error
$\Gamma \vdash [\mathsf{emp}]\ \mathbf{error}\ [\mathsf{err}\colon \mathsf{emp}]$

S-Minus
$$\frac{\Gamma \vdash [\mathsf{emp}]\, p\, [\mathsf{ok}(z)\colon \mathsf{emp}]}{\Gamma \vdash [\mathsf{emp}]\ -_{\mathbb{Z}}\, p\, [\mathsf{ok}(-z)\colon \mathsf{emp}]}$$

S-Add
$$\frac{\Gamma \vdash [\mathsf{emp}]\, p_1\, [\mathsf{ok}(z_1)\colon \mathsf{emp}] \quad \Gamma \vdash [\mathsf{emp}]\, p_2\, [\mathsf{ok}(z_2)\colon \mathsf{emp}]}{\Gamma \vdash [\mathsf{emp}]\, p_1 +_{\mathbb{Z}} p_2\, [\mathsf{ok}(z_1 + z_2)\colon \mathsf{emp}]}$$

S-Not
$$\frac{\Gamma \vdash [\mathsf{emp}]\, p\, [\mathsf{ok}(b)\colon \mathsf{emp}]}{\Gamma \vdash [\mathsf{emp}]\ \neg_{\mathbb{B}} p\, [\mathsf{ok}(\neg b)\colon \mathsf{emp}]}$$

S-Le
$$\frac{\Gamma \vdash [\mathsf{emp}]\, p_1\, [\mathsf{ok}(z_1)\colon \mathsf{emp}] \quad \Gamma \vdash [\mathsf{emp}]\, p_2\, [\mathsf{ok}(z_2)\colon \mathsf{emp}]}{\Gamma \vdash [\mathsf{emp}]\, p_1 \leq_{\mathbb{Z}} p_2\, [\mathsf{ok}(z_1 \leq z_2)\colon \mathsf{emp}]}$$

S-Free
$\Gamma \vdash [l \mapsto v]\ \mathbf{free}(l)\ [\mathsf{ok}\colon l \not\mapsto]$

S-Store
$\Gamma \vdash [l \mapsto v']\, l \leftarrow v\, [\mathsf{ok}\colon l \mapsto v]$

S-Load
$\Gamma \vdash [l \mapsto v]\, {}^{*}l\, [\mathsf{ok}(v)\colon l \mapsto v]$

S-FreeUninit
$\Gamma \vdash [l \mapsto_{?}]\ \mathbf{free}(l)\ [\mathsf{ok}\colon l \not\mapsto]$

S-StoreUninit
$\Gamma \vdash [l \mapsto_{?}]\, l \leftarrow v\, [\mathsf{ok}\colon l \mapsto v]$

S-LoadUninit
$\Gamma \vdash [l \mapsto_{?}]\, {}^{*}l\, [\mathsf{err}\colon l \mapsto_{?}]$

S-FreeFreed
$\Gamma \vdash [l \not\mapsto]\ \mathbf{free}(l)\ [\mathsf{err}\colon l \not\mapsto]$

S-StoreFreed
$\Gamma \vdash [l \not\mapsto]\, l \leftarrow v\, [\mathsf{err}\colon l \not\mapsto]$

S-LoadFreed
$\Gamma \vdash [l \not\mapsto]\, {}^{*}l\, [\mathsf{err}\colon l \not\mapsto]$

S-FreeMiss
$\Gamma \vdash [\mathsf{emp}]\ \mathbf{free}(l)\ [\mathsf{miss}(l)\colon \mathsf{emp}]$

S-StoreMiss
$\Gamma \vdash [\mathsf{emp}]\, l \leftarrow v\, [\mathsf{miss}(l)\colon \mathsf{emp}]$

S-LoadMiss
$\Gamma \vdash [\mathsf{emp}]\, {}^{*}l\, [\mathsf{miss}(l)\colon \mathsf{emp}]$

S-Alloc
$\Gamma \vdash [\mathsf{emp}]\ \mathbf{alloc}\ [\mathsf{ok}(l)\colon l \mapsto_{?}]$

S-Call
$$\frac{[(\overline{v})\ P \mid \epsilon\colon Q] \in \Gamma(f)}{\Gamma \vdash [P]\, f(\overline{v})\, [\epsilon\colon Q]}$$

S-LetCut
$$\frac{\Gamma \vdash [P]\, e_1\, [\epsilon\colon Q] \quad \epsilon \neq \mathsf{ok}(-)}{\Gamma \vdash [P]\ \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\, [\epsilon\colon Q]}$$

S-Choice
$$\frac{\Gamma \vdash [P]\, e_i\, [\epsilon\colon Q] \quad \exists\, i \in \{1, 2\}}{\Gamma \vdash [P]\, e_1 + e_2\, [\epsilon\colon Q]}$$

S-Let
$$\frac{\Gamma \vdash [P]\, e_1\, [\mathsf{ok}(v)\colon R] \quad \Gamma \vdash [R]\, e_2[v/x]\, [\epsilon\colon Q]}{\Gamma \vdash [P]\ \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\, [\epsilon\colon Q]}$$

S-Disj
$$\frac{\Gamma \vdash [P_1]\, e\, [\epsilon\colon Q_1] \quad \Gamma \vdash [P_2]\, e\, [\epsilon\colon Q_2]}{\Gamma \vdash [P_1 \vee P_2]\, e\, [\epsilon\colon Q_1 \vee Q_2]}$$

S-Frame
$$\frac{\Gamma \vdash [P]\, e\, [\epsilon\colon Q] \quad \mathsf{frameable}(\epsilon, R)}{\Gamma \vdash [P * R]\, e\, [\epsilon\colon Q * R]}$$

S-Exists
$$\frac{\Gamma \vdash [P]\, e\, [\epsilon\colon Q]}{\Gamma \vdash [\exists\, x.\ P]\, e\, [\epsilon\colon \exists\, x.\ Q]}$$

S-Cons
$$\frac{\Gamma' \subseteq \Gamma \quad \models (P' \Rightarrow P) \quad \Gamma' \vdash [P']\, e\, [\epsilon\colon Q'] \quad \models (Q \Rightarrow Q')}{\Gamma \vdash [P]\, e\, [\epsilon\colon Q]}$$

Figure 5 The RISL proof rules.

Plain values always reduce to the value itself successfully (S-Val). Pure expressions reduce to the result of the underlying operation on the reduction values of its operands (S-Minus, S-Not, S-Add and S-Le). Assume expressions only terminate if the term evaluates to true[3] (S-Assume). Error expressions always terminate erroneously (S-Error).

[3] For readability, when programs reduce to () (unit value), we simply write ok rather than ok(()).

For heap-manipulating expressions, the successful cases (with ok) are analogous to their ISL counterparts (S-Free, S-Store, S-Load and S-Alloc). As in ISL, accessing previously freed locations results in a memory safety issue and thus erroneous termination (S-FreeFreed, S-StoreFreed and S-LoadFreed). As we model uninitialised memory, compared to ISL we further have rules to account for accessing uninitialised memory. Specifically, while freeing and overwriting uninitialised locations terminates successfully (S-FreeUninit and S-StoreUninit), loading from such locations leads to UB (S-LoadUninit). We further introduce new rules to reason about missing resources (via the miss exit condition). For instance, when attempting to free a location l for which we do not own the necessary resources in the precondition, i.e. we have emp rather than $l \mapsto - \vee l \mapsto_?$, then we terminate unsuccessfully due to missing location l, denoted by the $\mathsf{miss}(l)$ exit condition in S-FreeMiss. The S-StoreMiss and S-LoadMiss describe analogous scenarios when storing to or loading from l. This miss exit condition could be surprising to readers familiar with ISL; we discuss it in more details shortly.

Let bindings correspond to sequential composition in ISL. As in ISL, S-LetCut captures the short-circuiting semantics of sequential executions: when the execution of the first expression e_1 terminates unsuccessfully, then the let expression **let** $x = e_1$ **in** e_2 also executes unsuccessfully. The S-Let rule captures the case where executing e_1 terminates successfully with value v, in which case (unlike in ISL) v is propagated to e_2 through variable substitution.

The S-Choice rule states that the states in Q are reachable when executing $e_1 + e_2$ if they are reachable from P when executing *either* branch. As in incorrectness (separation) logic, this is due to the *under-approximate* nature of RISL.

Missing resources. The miss exit condition is a novel aspect of RISL, allowing us to derive triples when the proof context does not carry sufficient resources to execute a program successfully. By contrast, in ISL, missing resources do not lead to a specific exit condition, but rather to a failure to derive a triple.

This is because unlike existing analyses based on ISL, RUXt is designed to reason about the soundness of Rust libraries, where types *capture the resources required for safe execution*. That is, if RUXt computes a subvariant assertion P for a given type `T`, then P *should* contain enough resources to execute the body of any safe function `f(x: T)`. Therefore, if we can derive a triple with a miss exit for the body of a function `f(x: T)` starting from a computed subvariant for `T`, we have shown that the resources associated with `T` are insufficient for the function to execute safely, and thus we can report it as a type safety violation.

Note that additional care is needed to ensure that triples with miss exits do not break the S-Frame rule. Specifically, when applying S-Frame to a triple with $\epsilon = \mathsf{miss}(l)$, we must ensure the framed-on resources R do not invalidate $\mathsf{miss}(l)$ by framing the missing location l. This is captured by the $\mathsf{frameable}(\epsilon, R)$ premise of the S-Frame rule, defined as follows:

$$\mathsf{frameable}(\epsilon, R) \triangleq \forall\, l.\ \epsilon = \mathsf{miss}(l) \Rightarrow \neg\, \mathsf{sat}(R \wedge (\mathsf{True} * (l \mapsto - \vee l \not\mapsto \vee l \mapsto_?)))$$

Note that when $\epsilon \in \{\mathsf{ok}, \mathsf{err}\}$, the $\mathsf{frameable}(\epsilon, R)$ premise is immediately satisfied, ensuring that we can *always* frame on additional resources R in such cases, as in ISL. That is, in these cases our frame rule is identical to that of ISL (which cannot account for missing resources). By contrast, in the case of $\epsilon = \mathsf{miss}(l)$, the right-hand side of the implication above ensures that R does not express ownership of l, i.e. does not contain $l \mapsto -$ or $l \not\mapsto$ or $l \mapsto_?$ (among other resources, as denoted by True).

Function calls. The S-Call rule allows specifications from the Γ context to be directly applied when they match a given function call. Rather than executing $f(\overline{v})$ directly, one can consult the specifications of f in Γ and check if there exists a specification $\left[(\overline{v})\ P \,\middle|\, \epsilon\colon Q\right] \in \Gamma(f)$ for the same input values $\overline{v}$.

Specification contexts $\boxed{\gamma \prec_{\mathsf{S}} \Gamma}$

$$\textsc{S-Empty}\quad \gamma \prec_{\mathsf{S}} \mathsf{empty}(\gamma) \qquad \textsc{S-Extend}\quad \frac{\gamma \prec_{\mathsf{S}} \Gamma \quad f(\overline{x})\{e\} \in \gamma \quad \Gamma \vdash [P]\, e[\overline{v}/\overline{x}]\, [\epsilon\colon Q]}{\gamma \prec_{\mathsf{S}} \Gamma\big[f \mapsto \big[(\overline{v})\ P \,\big|\, \epsilon\colon Q\big],\ \Gamma(f)\big]}$$

Figure 6 Proof rules for well-formed specification contexts.

The S-Cons rule further allows the specification context Γ' to be extended to Γ so long as Γ contains all specifications that Γ' does (i.e. $\Gamma' \subseteq \Gamma \triangleq \forall\ f.\ \Gamma'(f) \subseteq \Gamma(f)$). This ensures that Γ preserves any specification that may have been used to prove the original triple.

Note that RISL rules are only sound provided that the given specification context is consistent with the concrete implementation context under analysis. Therefore, we next show how to construct specification contexts soundly by deriving specifications in RISL.

5.2 Well-Formed Specification Contexts

The rules in Fig. 6 allow one to construct *well-formed* specification contexts. A specification context Γ is well-formed with respect to an implementation context γ, written $\gamma \prec_{\mathsf{S}} \Gamma$, when every specification in $\Gamma(f)$ holds of (can be derived for) the concrete implementation in $\gamma(f)$, for all f in γ. We write $\mathsf{empty}(\gamma)$ to denote an *empty specification context* for γ, i.e. one where $\Gamma(f)$ is an empty list, for all functions f in γ. An empty specification context for γ is vacuously well-formed with respect to γ (S-Empty). When $\gamma \prec_{\mathsf{S}} \Gamma$ holds, we can extend Γ (in a well-formed fashion) with a specification $\big[(\overline{v})\ P \,\big|\, \epsilon\colon Q\big]$ for f in γ, provided that we can prove it in RISL by directly executing the f body with concrete input values $\overline{v}$ (S-Extend).

In other words, by iterating over functions and deriving RISL (under-approximate) specifications for concrete inputs, we can dynamically construct Γ, tailor-made for a specific implementation context γ. This way, we can also reason about (mutually) recursive functions for a bounded number of iterations (i.e. under-approximate them) by first deriving specifications for non-recursive calls, which can then be used to derive specifications for the recursive calls. Moreover, note that the S-Cons rule ensures that derived RISL triples remain valid when extending the specification context, which in turn ensures that RISL is sound under specification context extension.

In the RUXt algorithm, we obtain reachable states by following the RISL proof (Fig. 5) and refinement (Fig. 6) rules. Indeed, every derived postcondition $[\epsilon : Q] \in \mathsf{execute}(\gamma, f(\overline{v}), P)$ on line 3 in Fig. 4 (left) implies that there exists Γ such that $\gamma \prec_{\mathsf{S}} \Gamma$ and $\Gamma \vdash [P]\, f(\overline{v})\, [\epsilon\colon Q]$ hold. As such, the soundness of RUXt follows directly from the soundness of RISL with respect to our under-approximate RUXtBelt semantics, which we describe next in §6.

6 RUXtBelt: The Semantic Model of RUXt

We present RUXtBelt, our under-approximate semantic model underpinning RISL and RUXt. As discussed in §1, RUXtBelt covers a large fragment of RustBelt. Unlike RustBelt, however, in RUXtBelt we explicitly account for memory safety errors (e.g. use after free) as this is required for bug detection. Specifically, while the RustBelt semantics simply gets "stuck" when encountering a memory safety issue, in RUXtBelt we directly identify them as errors using our unsuccessful exit conditions.

In §6.1 we present the RUXtBelt operational semantics. In §6.2 we define the semantic interpretation of RISL triples and prove the RISL proof system sound against the RUXtBelt semantics. Finally in §6.3 we prove that the RUXt algorithm is sound through our inadequacy theorem, stating that every UB detected by RUXt is a true safety violation in the library.

6.1 RUXtBelt Operational Semantics

We present two characterisations of our RUXtBelt semantics. First, we describe our *big-step* semantics that explicitly accounts for unsuccessful termination, but does not distinguish between the two kinds of unsuccessful termination (err or miss). Specifically, in our big-step semantics we assume that we have the *full* heap at all times and do not miss any resources, i.e. that there are no additional resources in the frame. As such, when accessing a location l that is not within the underlying heap, we simply terminate with err (and not miss(l)). We thus refer to this big-step semantics as the RUXtBelt *full semantics*.

We next present the RUXtBelt *instrumented* (big-step) semantics – inspired by JaVerT2.0 [9] and Gillian [26] – that can operate on *partial* heaps and thus distinguishes between unsuccessful termination due to memory safety errors (err) and missing resources (miss). We then establish a semantics preservation lemma (Lemma 1) stating that our instrumented semantics can simply be erased to our big-step semantics, in that if a program e terminates with exit condition ϵ under the instrumented semantics, then (1) either $\epsilon = \mathsf{ok}$ and e also terminates with ok in our big-step semantics; or (2) $\epsilon \neq \mathsf{ok}$ (i.e. it is either err or miss) and e terminates with err in our big-step semantics.

As we show below, our instrumented semantics simplifies the task of proving RISL sound: miss RISL triples directly correspond to miss judgements in our instrumented semantics. As such, we first prove RISL sound against our instrumented semantics, and subsequently show that RUXt is sound against our full semantics through our semantics preservation lemma.

RUXtBelt big-step semantics. We define our RUXtBelt big-step operational semantics in Fig. 7, through judgements of the form $\gamma \vdash \langle h \mid e \rangle \Downarrow \langle h' \mid \epsilon \rangle$, where γ is an implementation context (for function implementations), e is a λ_{RUXt} program (expression), ϵ is an exit condition and h, h' are *program states*. A program state is a heap, $h \in \textsc{Heap} \triangleq \textsc{Loc} \rightharpoonup \textsc{Val} \uplus \{☣, \varnothing\}$, mapping locations to either values in Val or designated values for uninitialised (☣) and freed ($\varnothing$) locations. The $\gamma \vdash \langle h \mid e \rangle \Downarrow \langle h' \mid \epsilon \rangle$ judgement states that h' is reachable with exit condition ϵ by evaluating e starting from h, under the implementation context γ.

We further define partial evaluation functions for terms ($\lfloor \cdot \rfloor_{\mathsf{t}} : \textsc{Term} \rightharpoonup \textsc{Val}$) and pure expressions ($\lfloor \cdot \rfloor_{\mathsf{p}} : \textsc{Pure} \rightharpoonup \textsc{Val}$) as follows:

$$\lfloor v \rfloor_{\mathsf{t}} \triangleq v \qquad \lfloor -_{\mathbb{Z}} p \rfloor_{\mathsf{p}} \triangleq -\lfloor p \rfloor_{\mathsf{p}} \textbf{ if } \lfloor p \rfloor_{\mathsf{p}} \in \mathbb{Z} \qquad \lfloor p_1 +_{\mathbb{Z}} p_2 \rfloor_{\mathsf{p}} \triangleq \lfloor p_1 \rfloor_{\mathsf{p}} + \lfloor p_2 \rfloor_{\mathsf{p}} \textbf{ if } \lfloor p_1 \rfloor_{\mathsf{p}}, \lfloor p_2 \rfloor_{\mathsf{p}} \in \mathbb{Z}$$
$$\lfloor t \rfloor_{\mathsf{p}} \triangleq \lfloor t \rfloor_{\mathsf{t}} \qquad \lfloor \neg_{\mathbb{B}} p \rfloor_{\mathsf{p}} \triangleq \neg \lfloor p \rfloor_{\mathsf{p}} \textbf{ if } \lfloor p \rfloor_{\mathsf{p}} \in \mathbb{B} \qquad \lfloor p_1 \leq_{\mathbb{Z}} p_2 \rfloor_{\mathsf{p}} \triangleq \lfloor p_1 \rfloor_{\mathsf{p}} \leq \lfloor p_2 \rfloor_{\mathsf{p}} \textbf{ if } \lfloor p_1 \rfloor_{\mathsf{p}}, \lfloor p_2 \rfloor_{\mathsf{p}} \in \mathbb{Z}$$

Most of the rules in Fig. 7 are standard. Pure expressions, assume expressions and explicit errors do not interact with the heap and leave it unchanged. Specifically, pure expressions simply reduce to their corresponding value (O-Pure), **assume**(t) is only reducible if t evaluates to true (O-Assume) and **error** terminates unsuccessfully with err (O-Error).

Heap allocation always returns a fresh, uninitialised location (O-Alloc). As in ISL, deallocation updates the value of the freed location to the designated $\varnothing$ value (O-Free). Storing to a heap location l is successful when l is not deallocated but may be potentially uninitialised (O-Store), while loading from l is only successful if l is initialised, in which case it returns the value at l (O-Load). Accessing a previously freed location or loading from an uninitialised location terminates erroneously (O-FreeErr, O-StoreErr and O-LoadErr).

Big-step evaluation $\boxed{\gamma \vdash \langle h \mid e \rangle \Downarrow \langle h' \mid \epsilon \rangle}$

$$\textsc{O-Pure}\ \frac{\lfloor p \rfloor_{\mathsf{p}} = v}{\gamma \vdash \langle h \mid p \rangle \Downarrow \langle h \mid \mathsf{ok}(v) \rangle} \qquad \textsc{O-Assume}\ \frac{\lfloor t \rfloor_{\mathsf{t}} = \mathsf{true}}{\gamma \vdash \langle h \mid \mathbf{assume}(t) \rangle \Downarrow \langle h \mid \mathsf{ok} \rangle} \qquad \textsc{O-Error}\ \frac{}{\gamma \vdash \langle h \mid \mathbf{error} \rangle \Downarrow \langle h \mid \mathsf{err} \rangle}$$

$$\textsc{O-Alloc}\ \frac{l \notin \mathsf{dom}(h)}{\gamma \vdash \langle h \mid \mathbf{alloc} \rangle \Downarrow \langle h[l \mapsto \text{☣}] \mid \mathsf{ok}(l) \rangle} \qquad \textsc{O-Free}\ \frac{\lfloor t \rfloor_{\mathsf{t}} = l \quad h(l) \in \textsc{Val} \uplus \{\text{☣}\}}{\gamma \vdash \langle h \mid \mathbf{free}(t) \rangle \Downarrow \langle h[l \mapsto \varnothing] \mid \mathsf{ok} \rangle}$$

$$\textsc{O-Store}\ \frac{\lfloor t_1 \rfloor_{\mathsf{t}} = l \quad h(l) \in \textsc{Val} \uplus \{\text{☣}\} \quad \lfloor t_2 \rfloor_{\mathsf{t}} = v}{\gamma \vdash \langle h \mid t_1 \leftarrow t_2 \rangle \Downarrow \langle h[l \mapsto v] \mid \mathsf{ok} \rangle} \qquad \textsc{O-Load}\ \frac{\lfloor t \rfloor_{\mathsf{t}} = l \quad h(l) = v}{\gamma \vdash \langle h \mid {}^{*}t \rangle \Downarrow \langle h \mid \mathsf{ok}(v) \rangle}$$

$$\textsc{O-FreeErr}\ \frac{\lfloor t \rfloor_{\mathsf{t}} = l \quad h(l) = \varnothing}{\gamma \vdash \langle h \mid \mathbf{free}(t) \rangle \Downarrow \langle h \mid \mathsf{err} \rangle} \qquad \textsc{O-StoreErr}\ \frac{\lfloor t_1 \rfloor_{\mathsf{t}} = l \quad h(l) = \varnothing}{\gamma \vdash \langle h \mid t_1 \leftarrow t_2 \rangle \Downarrow \langle h \mid \mathsf{err} \rangle} \qquad \textsc{O-LoadErr}\ \frac{\lfloor t \rfloor_{\mathsf{t}} = l \quad h(l) \in \{\varnothing, \text{☣}\}}{\gamma \vdash \langle h \mid {}^{*}t \rangle \Downarrow \langle h \mid \mathsf{err} \rangle}$$

$$\textsc{O-FreeMiss}\ \frac{\lfloor t \rfloor_{\mathsf{t}} = l \quad l \notin \mathsf{dom}(h)}{\gamma \vdash \langle h \mid \mathbf{free}(t) \rangle \Downarrow \langle h \mid \mathsf{err} \rangle} \qquad \textsc{O-StoreMiss}\ \frac{\lfloor t_1 \rfloor_{\mathsf{t}} = l \quad l \notin \mathsf{dom}(h)}{\gamma \vdash \langle h \mid t_1 \leftarrow t_2 \rangle \Downarrow \langle h \mid \mathsf{err} \rangle} \qquad \textsc{O-LoadMiss}\ \frac{\lfloor t \rfloor_{\mathsf{t}} = l \quad l \notin \mathsf{dom}(h)}{\gamma \vdash \langle h \mid {}^{*}t \rangle \Downarrow \langle h \mid \mathsf{err} \rangle}$$

$$\textsc{O-Choice}\ \frac{\gamma \vdash \langle h \mid e_i \rangle \Downarrow \langle h' \mid \epsilon \rangle \quad \exists\, i \in \{1, 2\}}{\gamma \vdash \langle h \mid e_1 + e_2 \rangle \Downarrow \langle h' \mid \epsilon \rangle} \qquad \textsc{O-LetCut}\ \frac{\gamma \vdash \langle h \mid e_1 \rangle \Downarrow \langle h' \mid \epsilon \rangle \quad \epsilon \neq \mathsf{ok}(-)}{\gamma \vdash \langle h \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rangle \Downarrow \langle h' \mid \epsilon \rangle}$$

$$\textsc{O-Let}\ \frac{\gamma \vdash \langle h \mid e_1 \rangle \Downarrow \langle h'' \mid \mathsf{ok}(v) \rangle \quad \gamma \vdash \langle h'' \mid e_2[v/x] \rangle \Downarrow \langle h' \mid \epsilon \rangle}{\gamma \vdash \langle h \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rangle \Downarrow \langle h' \mid \epsilon \rangle}$$

$$\textsc{O-Call}\ \frac{f(\overline{x})\{e\} \in \gamma \quad \gamma \vdash \langle h \mid e[\overline{t}/\overline{x}] \rangle \Downarrow \langle h' \mid \epsilon \rangle}{\gamma \vdash \langle h \mid f(\overline{t}) \rangle \Downarrow \langle h' \mid \epsilon \rangle}$$

Figure 7 The full RUXtBelt semantics; the instrumented RUXtBelt semantics is obtained by replacing err in O-FreeMiss, O-StoreMiss and O-LoadMiss with miss(l).

Moreover, accessing locations that are not part of the (full) heap also terminate erroneously (O-FreeMiss, O-StoreMiss and O-LoadMiss). That is, the full heap must contain all resources required for successful termination or it terminates erroneously.

The remaining judgements closely mimic RISL rules in Fig. 5. Starting from h, non-deterministic choice $e_1 + e_2$ reduces to h' with ϵ if *either* e_1 *or* e_2 does so. A let binding short-circuits if e_1 terminates unsuccessfully (O-LetCut); otherwise, e_1 propagates its value v to e_2 by substituting the binder x (O-Let). A call to f is reduced by substituting its arguments with input terms in its body when f is in the implementation context (O-Call).

Instrumented semantics. We next define our *instrumented* semantics that operates on incomplete heaps and distinguishes err and miss conditions. Our instrumented semantics comprises judgements of the $\gamma \vdash \langle h \mid e \rangle \Downarrow_i \langle h' \mid \epsilon \rangle$ (note the i subscript in $\Downarrow$) and only

differs from the full semantics by yielding a $\mathsf{miss}(l)$ exit in O-FreeMiss, O-StoreMiss and O-LoadMiss. In effect, the full semantics maps every miss termination in the instrumented semantics to an err one, preserving all other outcomes, as we show in the following lemma.

▶ **Lemma 1** (Semantics preservation). *For all γ, h, h', e and ϵ, if $\gamma \vdash \langle h \mid e \rangle \Downarrow_i \langle h' \mid \epsilon \rangle$, then one of the following holds:*

1. $\epsilon = \mathsf{ok}(v)$ *and* $\gamma \vdash \langle h \mid e \rangle \Downarrow \langle h' \mid \mathsf{ok}(v) \rangle$.
2. $\epsilon \neq \mathsf{ok}(-)$ *and* $\gamma \vdash \langle h \mid e \rangle \Downarrow \langle h' \mid \mathsf{err} \rangle$.

As we show below, our instrumented semantics enjoys strong frame preservation properties, namely *frame subtraction* (Lemma 2) and *frame addition* (Lemma 3). The former describes the effects of removing fragments from the underlying heap, while the latter accounts for heap extensions. We write $h_s \# h_F$ to denote that h_s and h_F are *disjoint* (i.e. have disjoint domains) and can thus be composed (combined) through the disjoint union operator $\uplus$.

▶ **Lemma 2** (Frame subtraction). *For all γ, h_s, h_F, h'_s, e and ϵ, if $\gamma \vdash \langle h_s \uplus h_F \mid e \rangle \Downarrow_i \langle h'_s \uplus h_F \mid \epsilon \rangle$ and $h_s \# h_F$ holds, then one of the following holds:*

1. $\gamma \vdash \langle h_s \mid e \rangle \Downarrow_i \langle h'_s \mid \epsilon \rangle$ *and* $h'_s \# h_F$.
2. *There exist h', l such that $\gamma \vdash \langle h_s \mid e \rangle \Downarrow_i \langle h' \mid \mathsf{miss}(l) \rangle$, $h' \# h_F$ and $l \in \mathsf{dom}(h_F)$.*

▶ **Lemma 3** (Frame addition). *For all γ, h, h', e and ϵ, if $\gamma \vdash \langle h \mid e \rangle \Downarrow_i \langle h' \mid \epsilon \rangle$, then for all h_F such that $h' \# h_F$ holds, one of the following holds:*

1. $\gamma \vdash \langle h \uplus h_F \mid e \rangle \Downarrow_i \langle h' \uplus h_F \mid \epsilon \rangle$ *and* $h \# h_F$.
2. *There exists l such that $\epsilon = \mathsf{miss}(l)$ and $l \in \mathsf{dom}(h_F)$.*

Lemma 2 states that removing a fragment from the heap under execution either (1) preserves the behaviour of the program; or (2) terminates unsuccessfully due to a missing location in the frame. Conversely, Lemma 3 states that extending the underlying heap (1) preserves the behaviour of the program unless (2) the program terminates with a miss before adding the frame, and the frame contains the missing location. This is precisely why we need the $\mathsf{frameable}(\epsilon, R)$ requirement in the premise of the S-Frame rule, i.e. to ensure that the extension R does not include missing resources that may change the termination condition.

These lemmas demonstrate how (when dealing with incomplete heaps in our instrumented semantics) extending and shrinking the underlying heap may affect the termination condition with regards to missing resources. By contrast, in our full semantics we assume that the underlying heap is always complete, and thus we need not account for extending or shrinking it. As our instrumented semantics considers partial (incomplete) heaps, it closely captures the *compositional* nature of our RISL proof rules, where in each rule the pre- and post-conditions only describe the (partial) heap fragments needed for executing a given λ_{RUXt} program, and the underlying heap (resources) may always be extended using the S-Frame rule. As such, it is simpler to first show RISL sound against our instrumented semantics, as we show next in §6.2, and then relate it to our full semantics using Lemma 1 for our inadequacy result in §6.3.

6.2 Semantic RISL Triples and Soundness

We next formalise the *semantic interpretation* of RISL triples. Recall that RISL triples (Fig. 5) are given using assertions. As such, we first present the semantics of RISL assertions.

Semantic interpretation $(\!|\cdot|\!) : \textsc{Asrt} \to \textsc{Heap} \to \textsc{Prop}$

$$(\!|\llcorner \pi \lrcorner|\!) \triangleq \lambda\, h.\ h = \emptyset \wedge \pi \qquad (\!|\mathsf{True}|\!) \triangleq \lambda\, _.\ \top \qquad (\!|\mathsf{False}|\!) \triangleq \lambda\, _.\ \bot$$
$$(\!|P \wedge Q|\!) \triangleq \lambda\, h.\ (\!|P|\!)(h) \wedge (\!|Q|\!)(h) \qquad (\!|P \vee Q|\!) \triangleq \lambda\, h.\ (\!|P|\!)(h) \vee (\!|Q|\!)(h)$$
$$(\!|P \Rightarrow Q|\!) \triangleq \lambda\, h.\ (\!|P|\!)(h) \Rightarrow (\!|Q|\!)(h) \qquad (\!|\exists\ x.\ P|\!) \triangleq \lambda\, h.\ \exists\ x.\ (\!|P|\!)(h)$$
$$(\!|l \mapsto v|\!) \triangleq \lambda\, h.\ h = \{l \mapsto v\} \qquad (\!|l \mapsto_? |\!) \triangleq \lambda\, h.\ h = \{l \mapsto \lightning\} \qquad (\!|l \not\mapsto|\!) \triangleq \lambda\, h.\ h = \{l \mapsto \varnothing\}$$
$$(\!|\mathsf{emp}|\!) \triangleq \lambda\, h.\ h = \emptyset \qquad (\!|P * Q|\!) \triangleq \lambda\, h.\ \exists\ h_P, h_Q.\ h = h_P \uplus h_Q \wedge (\!|P|\!)(h_P) \wedge (\!|Q|\!)(h_Q)$$

Figure 8 Assertion semantics.

Assertion semantics. We define the semantics of assertions in Fig. 8 through an interpretation function, $(\!|\cdot|\!)$, that maps assertions to predicates (Prop in Rocq) on heaps (i.e. our program states). Pure assertions require no heap resources and thus assert ownership of an empty heap. Semantics of truth and falsity do not depend on the heap, semantics of conjunction, disjunction, implication and existential quantification are defined inductively and as expected. For separation logic (SL) assertions, emp simply expresses ownership of an empty heap, while $l \mapsto v$ expresses ownership of a singleton heap mapping l to v. Similarly, $l \not\mapsto$ and $l \mapsto_?$ express ownership of a singleton heap mapping l to $\varnothing$ and $\lightning$, respectively. Finally, the separating conjunction requires the heap to be split into two disjoint sub-heaps, each satisfying one sub-assertion. An assertion P is *valid*, written $\models P$, if it holds of *every* heap; P is *satisfiable*, written $\mathsf{sat}(P)$, if it holds of *some* heap:

$$\models P \triangleq \forall\ h.\ (\!|P|\!)(h) \qquad \mathsf{sat}(P) \triangleq \exists\ h.\ (\!|P|\!)(h)$$

Semantic RISL triples. A *semantic* triple (*cf.* *syntactic* RISL triples in Fig. 5), $\gamma \vDash [P]\, e\, [\epsilon\colon Q]$, states that every heap h_q satisfying Q is reachable under ϵ from some heap h_p satisfying P by executing e under implementation context γ:

$$\gamma \vDash [P]\, e\, [\epsilon\colon Q] \triangleq \forall\ h_q.\ (\!|Q|\!)(h_q) \Rightarrow \exists\ h_p.\ (\!|P|\!)(h_p) \wedge \gamma \vdash \langle h_p \mid e \rangle \Downarrow_i \langle h_q \mid \epsilon \rangle$$

Note that while the syntactic RISL triples (Fig. 5) are defined using a *specification* context Γ, semantic triples (above) are given using an *implementation* context γ. To give a semantic meaning to our RISL triples, we thus lift the notion of a semantic triple with γ, to one using Γ, provided that Γ is *well-formed* with respect to γ. Recall that in §5.2 we provided a *logical* notion of this well-formedness, $\gamma \prec_{\mathsf{S}} \Gamma$, that can be derived using the rules in Fig. 6. We analogously define a *semantic* notion of well-formedness as follows. We define the semantics of the refinement relation from §5.2 as follows:

$$\gamma \lessdot_{\mathsf{S}} \Gamma \triangleq \forall f, \overline{v}, P, \epsilon, Q.\ \left[(\overline{v})\ P \mid \epsilon\colon Q\right] \in \Gamma(f) \Rightarrow \exists\ \overline{x}, e.\ f(\overline{x})\{e\} \in \gamma \wedge \gamma \vDash [P]\, e[\overline{v}/\overline{x}]\, [\epsilon\colon Q]$$

This ensures that every specification $\left[(\overline{v})\ P \mid \epsilon\colon Q\right]$ of function f in Γ is the result of directly executing the function body with the given input values.

We next lift the notion of semantic triples (with γ) to *semantic* RISL *triples* (with Γ):

$$\Gamma \vDash [P]\, e\, [\epsilon\colon Q] \triangleq \forall\ \gamma.\ \gamma \lessdot_{\mathsf{S}} \Gamma \Rightarrow \gamma \vDash [P]\, e\, [\epsilon\colon Q]$$

That is, as RISL only relies on derived specifications and not on concrete implementations, a semantic RISL triple with Γ holds when the corresponding semantic triple holds for all implementation contexts γ with respect to which Γ is semantically well-formed.

Soundness of RISL. In Lemma 4 below we prove that (i) every RISL triple that can be derived using the rules in Fig. 5 is a valid semantic RISL triple.; and (ii) every well-formed specification context Γ constructed using the rules in Fig. 6 is semantically well-formed.

▶ **Lemma 4.** *For all Γ, γ, P, Q, e and ϵ:*
1. *if $\Gamma \vdash [P]\, e\, [\epsilon\colon Q]$ is derived using the rules in Fig. 5, then $\Gamma \vDash [P]\, e\, [\epsilon\colon Q]$ holds; and*
2. *if $\gamma \prec_{\mathsf{S}} \Gamma$ is derived using the rules in Fig. 6, then $\gamma \lll_{\mathsf{S}} \Gamma$ holds.*

Finally, we prove that the RISL proof system is sound by showing that every triple derived for a given implementation using the RISL rules in Fig. 5 is a (sound) semantic triple.

▶ **Theorem 5** (RISL soundness). *For all Γ, γ, P, Q, e and ϵ, if $\gamma \prec_{\mathsf{S}} \Gamma$ and $\Gamma \vdash [P]\, e\, [\epsilon\colon Q]$ is derived using the rules in Fig. 5, then $\gamma \vDash [P]\, e\, [\epsilon\colon Q]$ holds.*

Crucially, RUXt relies on this result to justify the soundness of its summary inference. Specifically, every postcondition $[\epsilon : Q] \in \mathsf{execute}(\gamma, f(\overline{v}), P)$ derived by its underlying symbolic execution engine (underpinned by RISL) on line 3 (Fig. 4, left) must ensure that $\gamma \vDash [P]\, f(\overline{v})\, [\epsilon\colon Q]$ holds, which follows immediately from the RISL soundness above (Theorem 5). That is, by implementing the RISL proof system in its underlying symbolic execution engine, the RUXt algorithm thus guarantees that every summary it infers is a sound under-approximation of the reachable states.

6.3 Soundness of RUXt

We next show that RUXt is sound by proving its no-false-positives guarantee: when RUXt detect a UB with a witness program e, then e is indeed a safe program that results in UB, witnessing that the usage of unsafe code is not encapsulated by the safe API of the library. To this end, using the RISL soundness result (Theorem 5), in Theorem 6 below we first show that every summary derived by RUXt yields a sound RISL triple starting from an initial empty heap and terminating successfully. Furthermore, the witness program associated with inferred summary is guaranteed to be a safe program, meaning that the corresponding safe value can be constructed by solely interacting with the safe API of the library. More concretely, given an implementation context γ, a signature context Δ and a summary context Σ, if RUXt derives $\gamma, \Delta \vdash \Sigma$ using the rules in Fig. 4 (right) with $\mathsf{Summary}(\lambda v.\ Q \mid e)$ in Σ, then for every concrete value v (1) e is a safe program; and (2) $\gamma \vDash [\mathsf{emp}]\, e\, [\mathsf{ok}(v)\colon Q]$ holds.

▶ **Theorem 6** (Summary soundness). *For all γ, Δ, Σ, Q, e, if $\gamma, \Delta \vdash \Sigma$ is derived using the rules in Fig. 4 and $\mathsf{Summary}(\lambda v.\ Q \mid e) \in \Sigma$, then e is a safe program and $\gamma \vDash [\mathsf{emp}]\, e\, [\mathsf{ok}(v)\colon Q]$ holds for all v.*

Finally, using the soundness of summary inference in RUXt, we prove our key *inadequacy* theorem: given a well-formed summary context, if the refutation procedure reports a type unsoundness, then the returned witness is a safe program that leads to UB.

▶ **Theorem 7** (Inadequacy). *For all γ, Δ, Σ, e, if $\gamma, \Delta \vdash \Sigma$ is derived using the rules in Fig. 4 and $\mathsf{TryRefute}(\gamma, \Delta, \Sigma)$ returns a witness program e, then e is a safe program and $\gamma \vdash \langle \emptyset \mid e \rangle \Downarrow \langle h \mid \mathsf{err} \rangle$, for some h.*

Note that our inadequacy theorem states that e results in an error in our *full* semantics starting from an empty heap. Although summaries are only shown to be reachable using the *instrumented* semantics (in the conclusion of Theorem 6), our semantics preservation lemma (Lemma 1) allows us to exchange any judgement of the instrumented semantics into

one in our full semantics (by leaving successful judgements unchanged, while transforming unsuccessful ones with miss or err into erroneous ones with err), provided that we operate on the *full* heap. Indeed, the empty heap should suffice for safe programs to execute safely, and thus RUXt can soundly report miss exits as true UB. Concretely, our inadequacy theorem states that *any UB reported by* RUXt *is a true safety violation in the library*, that can be triggered by its returned witness program.

7 Related and Future Work

RUXt exists in a rich ecosystem of tools and techniques for analysing unsafe Rust code, as well as techniques that combine type-guided reasoning with under-approximate reasoning. We discuss these in turn, categorising Rust analysis tools into three categories: compositional verification, whole-program analysis and bug-finding tools.

Compositional type-safety verification for Rust. There are a number of tools that aim to *verify* the type safety of Rust programs. RustBelt [19] is an Iris mechanisation [20, 22] that provides a formal model of Rust's type system and provides a framework for compositionally verifying the type safety of internally unsafe libraries. Later, RefinedRust [10] (based on Lithium [33]), VeriFast for Rust [7] (based on VeriFast [15]) and Gillian-Rust [1] (based on Gillian [8, 26]) are all underpinned by RustBelt and aim to verify the type safety of Rust code, each with their own set of caveats and trade-offs. Verus [23] is a language (embedded in Rust) for writing formally verified code that can be extracted to idiomatic Rust code. However, Verus cannot verify existing unsafe Rust code that has not been written within Verus. While these tools are powerful, they all require a substantial amount of manual effort on the part of users and, as verification tools, are prone to false positives.

Whole-program analysis for Rust. Miri is a Rust interpreter distributed with the Rust compiler. It allows for the concrete execution of Rust programs and can detect many UB such as out-of-bounds memory accesses or more subtle Rust-specific issues, e.g. the violation of aliasing rules [18]. However, Miri is a testing tool, it requires the user to write tests, and can only explore one execution path.

Kani [35] is a whole-program bounded model checker for Rust. It performs symbolic execution, and as such can explore infinitely many execution paths. However, Kani still requires users to write tests and first-order logic assertions, and it is prone to *both* false positives (due to its over-approximate nature) *and* false negatives (as it is bounded). Moreover, Kani is unsound with respect to several Rust-specific features, e.g. the detection of uninitialised memory accesses, or the violation of aliasing rules. Neither Miri nor Kani can detect type unsoundness in Rust programs, as they cannot reason compositionally about library invariants.

Bug-finding tools for Rust. We know of three automatic bug-finders for Rust: Rudra [2], TraitInv [5] and Crabtree [34]. Rudra uses linter-like heuristics to detect three specific unsafe patterns in Rust code that are known to often lead to type unsoundness. It is automatic and was ran on the entirety of the Rust openly-available packages (`crates.io`), finding more than 250 new safety issues. However, its scope is limited to the detection of the three patterns, and it is prone to false positives.

TraitInv automatically synthesises symbolic tests that check certain properties expected to hold for eight core Rust traits (e.g. that `PartialEq` is symmetric and transitive). TraitInv then runs these tests using Kani and reports violations. Unfortunately, TraitInv's approach is

limited to these specific traits, and supporting new traits requires manual effort as it cannot infer new trait invariants. Furthermore, since it relies on Kani, TraitInv is also prone to false positives; it can only be extended to support first-order properties and cannot be used to enforce separation logic (ownership) properties such as that required by the `Copy` trait.

Crabtree is an automatic test synthesiser that uses a type-guided approach to increase coverage of the library under test. It can generate tests that make use of advanced Rust features such as higher-order trait functions; it uses Miri to execute the generated tests and can detect type unsoundness in the library if a test fails. Its type-guided approach is somewhat similar to ours. However, instead of inferring type subvariants, Crabtree notes when it finds a *path* that generates a value of a type not previously seen in the test suite, and uses that information to call functions that have not yet been covered because of the lack of input values for this type. This approach is, unlike ours, not compositional, and still quickly leads to combinatorial (path) explosion.

Type refutation and under-approximation. The recent development of Incorrectness Logic [31] has led to the exploration of reasoning techniques that combine type systems with under-approximation, starting with a theoretical experiment from Ramsay and Walpole [32] who introduced a two-sided type system that allows for refutation of type assignments.

Next, HayStack [38] is an extension of LiquidHaskell [36], a refinement type system for Haskell. HayStack is similar to RUXt, in that it infers subsets of the refinement type spaces that are sufficient to refute the refinement types provided by the user. However, it differs from RUXt in several key aspects. First, HayStack is used to provide explanations for type errors when the user provides a refinement type annotation that cannot be proven by LiquidHaskell. However, as refinement type checking is inherently over-approximate, inability to type-check does not guarantee that the program is incorrect. Furthermore, HayStack can only help finding counter-examples in Haskell programs that have been annotated with refinement types and is therefore not fully automatic. Finally, LiquidHaskell refinement types are first-order and cannot express separation logic (ownership) properties.

Finally, Qian et al. [30] propose a semantic type refuter that can find type unsoundness in a simple functional language. However, their approach requires knowledge of the *semantic typing of all existing types*, which must be defined using first-order logic. As such, they cannot find type unsoundness in internally-unsafe libraries, where type invariants must be inferred from the code itself and must be expressed using separation logic.

Symbolic execution for bug-finding. There is an extensive literature on symbolic execution for bug-finding, which can be split into two categories: dynamic and static symbolic execution. None of these techniques have been applied to Rust, or leverage the main novelty in our approach: the use of type-guided reasoning to refute the type safety of a program.

Dynamic first-order symbolic execution (or concolic execution) has been used to automatically detect bugs and generate tests using tools such as KLEE [6], DART [12, 11, 13], and more [39, 21]. However, our prototype implementation is based on static compositional symbolic execution, which, in the context of bug-finding, is implemented in Infer:Pulse [24] and Gillian [26, 25]. However, as explained in §2, these tools are only able to soundly detect *manifest* bugs, while our approach can detect a larger class of bugs, namely *type unsafety*.

Future Work. In the future, we will extend and build on our work here in several ways. First, we will extend our λ_{RUXt} language and RUXtBelt formalisation to include support for relevant Rust features such as polymorphism and higher-order functions. Second, we

will extend RUXt accordingly to include support for these features, and further ensure that UB reports are accompanied by witness programs. Finally, we will perform a large-scale evaluation of RUXt by applying it to real Rust codebases.

References

1 Sacha-Élie Ayoun, Xavier Denis, Petar Maksimović, and Philippa Gardner. A Hybrid Approach to Semi-Automated Rust Verification. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025. doi:10.1145/3729289.

2 Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: Finding memory safety bugs in rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 84–99, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3477132.3483570.

3 Ariel Ben-Yehuda. std::thread::joinguard (and scoped) are unsound because of reference cycles. rust issue #24292, 2015. URL: https://github.com/rust-lang/rust/issues/24292.

4 Christophe Biocca. std::vec::intoiter::as_mut_slice borrows &self, returns &mut of contents. rust issue #39465, 2017. URL: https://github.com/rust-lang/rust/issues/39465.

5 Twain Byrnes, Yoshiki Takashima, and Limin Jia. Automatically enforcing rust trait properties. In Rayna Dimitrova, Ori Lahav, and Sebastian Wolff, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 210–223, Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-50521-8_10.

6 Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, USA, December 2008. USENIX Association. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

7 Nima Rahimi Foroushaani and Bart Jacobs. Modular Formal Verification of Rust Programs with Unsafe Blocks, December 2022. doi:10.48550/arXiv.2212.12976.

8 José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. Gillian, part i: A multi-language platform for symbolic execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 927–942, New York, NY, USA, June 2020. Association for Computing Machinery. doi:10.1145/3385412.3386014.

9 José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. JaVerT 2.0: Compositional symbolic execution for JavaScript. *Proceedings of the ACM on Programming Languages*, 3(POPL):66:1–66:31, January 2019. doi:10.1145/3290379.

10 Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. Refinedrust: A type system for high-assurance verification of rust programs. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024. doi:10.1145/3656422.

11 Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 47–54, New York, NY, USA, January 2007. Association for Computing Machinery. doi:10.1145/1190216.1190226.

12 Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005. doi:10.1145/1064978.1065036.

13 Patrice Godefroid, Aditya Nori, and Sriram Rajamani. Compositional May-Must Program Analysis: Unleashing The Power of Alternation, January 2009.

14 Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 14–26, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/360204.375719.

15 Bart Jacobs, Jan Smans, and Frank Piessens. A Quick Tour of the VeriFast Program Verifier. In Kazunori Ueda, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 304–311, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-17164-2_21.

16 Ralf Jung. The Scope of Unsafe. https://www.ralfj.de/blog/2016/01/09/the-scope-of-unsafe.html, January 2016.

17 Ralf Jung. Mutexguard<cell<i32» must not be sync. rust issue #41622, 2017. URL: https://github.com/rust-lang/rust/issues/41622.

18 Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: An aliasing model for Rust. *Proceedings of the ACM on Programming Languages*, 4(POPL):41:1–41:32, December 2019. doi:10.1145/3371109.

19 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158154.

20 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, November 2018. doi:10.1017/S0956796818000151.

21 Yunho Kim, Shin Hong, and Moonzoo Kim. Target-driven compositional concolic testing with function summary refinement for effective bug detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 16–26, New York, NY, USA, August 2019. Association for Computing Machinery. doi:10.1145/3338906.3338934.

22 Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proceedings of the ACM on Programming Languages*, 2(ICFP), July 2018. doi:10.1145/3236772.

23 Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust Programs using Linear Ghost Types. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):85:286–85:315, April 2023. doi:10.1145/3586037.

24 Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.*, 6(OOPSLA1), April 2022. doi:10.1145/3527325.

25 Andreas Lööw, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Caroline Cronjäger, Petar Maksimović, and Philippa Gardner. Compositional Symbolic Execution for Correctness and Incorrectness Reasoning. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, volume 313 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:28, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2024.25.

26 Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. Gillian, Part II: Real-World Verification for JavaScript and C. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 827–850, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-81688-9_38.

27 Nicholas D. Matsakis and Felix S. Klock. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104, New York, NY, USA, October 2014. Association for Computing Machinery. doi:10.1145/2663171.2663188.

28 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, pages 41–62, Berlin, Heidelberg, 2016. Springer. doi:10.1007/978-3-662-49122-5_2.

29 Peter W. O'Hearn. Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL):10:1–10:32, December 2019. doi:10.1145/3371078.

30 Kelvin Qian, Scott Smith, Brandon Stride, Shiwei Weng, and Ke Wu. Semantic-Type-Guided Bug Finding. *Software Artifact for Semantic-Type-Guided Bug Finding*, 8(OOPSLA2):348:2183–348:2210, October 2024. doi:10.1145/3689788.

31 Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. Local reasoning about the presence of bugs: Incorrectness separation logic. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 225–252, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-53291-8_14.

32 Steven Ramsay and Charlie Walpole. Ill-Typed Programs Don't Evaluate. *Proceedings of the ACM on Programming Languages*, 8(POPL):2010–2040, January 2024. doi:10.1145/3632909.

33 Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. RefinedC: Automating the foundational verification of C code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 158–174, New York, NY, USA, June 2021. Association for Computing Machinery. doi:10.1145/3453483.3454036.

34 Yoshiki Takashima, Chanhee Cho, Ruben Martins, Limin Jia, and Corina S. Păsăreanu. Crabtree: Rust API Test Synthesis Guided by Coverage and Type. *Artifact Package for Paper "Crabtree: Rust API Test Synthesis Guided by Coverage and Type"*, 8(OOPSLA2):293:618–293:647, October 2024. doi:10.1145/3689733.

35 The Kani Team. How Open Source Projects are Using Kani to Write Better Software in Rust | AWS Open Source Blog. https://aws.amazon.com/blogs/opensource/how-open-source-projects-are-using-kani-to-write-better-software-in-rust/, November 2023.

36 Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282, New York, NY, USA, August 2014. Association for Computing Machinery. doi:10.1145/2628136.2628161.

37 Neven Villani, Johannes Hostert, Derek Dreyer, and Ralf Jung. Tree Borrows, November 2024. URL: https://jhostert.de/assets/pdf/papers/villani2024trees.pdf.

38 Robin Webbers, Klaus Von Gleissenthall, and Ranjit Jhala. Refinement Type Refutations. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):962–987, October 2024. doi:10.1145/3689745.

39 Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V. Krishnamurthy, and Paul Yu. UBITect: A precise and scalable method to detect use-before-initialization bugs in Linux kernel. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pages 221–232, New York, NY, USA, November 2020. Association for Computing Machinery. doi:10.1145/3368089.3409686.

Detecting Functionality-Specific Vulnerabilities via Retrieving Individual Functionality-Equivalent APIs in Open-Source Repositories

Tianyu Chen
Peking University, Beijing, China

Zeyu Wang
Huawei Cloud Computing Technologies Co., Ltd., Guiyang, China

Lin Li
Huawei Cloud Computing Technologies Co., Ltd., Guiyang, China

Ding Li
Peking University, Beijing, China

Zongyang Li
Peking University, Beijing, China

Xiaoning Chang
Huawei Cloud Computing Technologies Co., Ltd., Guiyang, China

Pan Bian
Huawei Cloud Computing Technologies Co., Ltd., Guiyang, China

Guangtai Liang
Huawei Cloud Computing Technologies Co., Ltd., Guiyang, China

Qianxiang Wang
Huawei Cloud Computing Technologies Co., Ltd., Guiyang, China

Tao Xie[1]
Peking University, Beijing, China

Abstract

Functionality-specific vulnerabilities, which mainly occur in Application Programming Interfaces (APIs) with specific functionalities, are crucial for software developers to detect and avoid. When detecting individual functionality-specific vulnerabilities, the existing two categories of approaches are ineffective because they consider only the API bodies and are unable to handle diverse implementations of functionality-equivalent APIs. To effectively detect functionality-specific vulnerabilities, we propose APISS, the first approach to utilize API doc strings and signatures instead of API bodies. APISS first retrieves functionality-equivalent APIs for APIs with existing vulnerabilities and then migrates Proof-of-Concepts (PoCs) of the existing vulnerabilities for newly detected vulnerable APIs. To retrieve functionality-equivalent APIs, we leverage a Large Language Model for API embedding to improve the accuracy and address the effectiveness and scalability issues suffered by the existing approaches. To migrate PoCs of the existing vulnerabilities for newly detected vulnerable APIs, we design a semi-automatic schema to substantially reduce manual costs. We conduct a comprehensive evaluation to empirically compare APISS with four state-of-the-art approaches of detecting vulnerabilities and two state-of-the-art approaches of retrieving functionality-equivalent APIs. The evaluation subjects include 180 widely used Java repositories using 10 existing vulnerabilities, along with their PoCs. The results show that APISS effectively retrieves functionality-equivalent APIs, achieving a Top-1 Accuracy of 0.81 while the best of the baselines under comparison achieves only 0.55. APISS is highly efficient: the manual costs are within 10 minutes per vulnerability and the end-to-end runtime overhead of testing one candidate API is less than 2 hours. APISS detects 179 new vulnerabilities and receives 60 new CVE IDs, bringing high value to security practice.

2012 ACM Subject Classification Security and privacy → Software security engineering

Keywords and phrases Application Security, Vulnerability Detection, Large Language Model

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.6

[1] Corresponding author

© Tianyu Chen, Zeyu Wang, Lin Li, Ding Li, Zongyang Li, Xiaoning Chang, Pan Bian, Guangtai Liang, Qianxiang Wang, and Tao Xie;
licensed under Creative Commons License CC-BY 4.0

39th European Conference on Object-Oriented Programming (ECOOP 2025).
Editors: Jonathan Aldrich and Alexandra Silva; Article No. 6; pp. 6:1–6:27

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Funding This work was partially supported by National Natural Science Foundation of China under Grant No. 92464301. Tao Xie is also affiliated with the School of Computer Science, Peking University; Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China.

1 Introduction

Application Programming Interfaces (APIs) in open source software are critical and widely used in software development [40], while the usage of APIs also increases the burden on developers to avoid security vulnerabilities among the APIs. Functionality-specific vulnerabilities, as one common type of API vulnerabilities, are defined as those that mainly occur in APIs with specific functionalities. For example, `Path Traversal` vulnerabilities (i.e., unauthorized access to a restricted directory) likely occur in an `unzip` API if it forgets to check whether the paths in its inputs (an uploaded zip file) are permitted. These vulnerabilities are often caused by lacking crucial operations during implementation, such as input validation, so similar vulnerabilities might occur among individual APIs with equivalent/similar functionalities (referred to as functionality-equivalent APIs).

However, the existing approaches, which fall into two categories, are ineffective in detecting functionality-specific vulnerabilities, as these approaches consider only the API bodies. First, some approaches [43, 26] aim to detect recurring vulnerabilities, which come from reused code base or shared code logic. However, these approaches find only vulnerabilities that have both similar functionalities and similar implementations with existing ones, thus suffering from high false negatives when encountering functionality-equivalent APIs with diverse implementations. Second, some other approaches [41, 52, 39, 8, 38, 37, 42, 31, 17, 7] aim to detect vulnerabilities other than recurring ones, typically via deep learning, which learns code features of vulnerable APIs/methods of existing vulnerabilities. However, these approaches are ineffective because the APIs affected by these vulnerabilities have diverse implementations without a common pattern. Additionally, these approaches cannot provide Proof-of-Concepts (PoCs) for the detected vulnerabilities, leading to high manual costs in vulnerability validation.

To effectively detect functionality-specific vulnerabilities, we propose APISS, the first approach that utilizes API doc strings and signatures (instead of focusing on API bodies), consisting of two steps: retrieving functionality-equivalent APIs for existing vulnerabilities and then migrating PoCs of the existing vulnerabilities to exploit new vulnerabilities. Our rationale comes from our observation of functionality-equivalent APIs: these APIs have similar doc strings and signatures because an API's input/output format (i.e., the API name, parameters, and output type included in its signature) is determined by this API's functionality and the doc string is used to explain each component of this API's signature. Thus, the doc strings and signatures of APIs with existing vulnerabilities can be utilized to retrieve functionality-equivalent APIs with potential functionality-specific vulnerabilities.

There are two main challenges in the two steps of APISS, respectively. First, the main challenge of retrieving functionality-equivalent APIs is the effectiveness and scalability issues. The effectiveness issue of existing approaches [29, 11, 6, 47] is caused by their overemphasis on the similarities of API bodies, not being able to handle diverse implementations of functionality-equivalent APIs. The scalability issue of these existing approaches comes from their training/fine-tuning on specific datasets, e.g., mapping a Java API to a C# API [29, 11, 6] or a Swift one [47]. Second, the main challenge of migrating the PoCs of existing vulnerabilities is the high manual costs. Migrating the PoCs requires building the environment and API dependencies of the project repository that includes a newly-detected

vulnerable API. Given that newly-detected vulnerable APIs come from different repositories and have different context dependencies, environment/dependency building incurs high manual costs.

To address the preceding two challenges, we design two respective techniques based on our observations on functionality-equivalent APIs and PoCs of functionality-specific vulnerabilities. First, we leverage a Large Language Model (LLM) for API embedding to address the effectiveness and scalability issues due to the enormous achievements of LLMs. For each pair of APIs, we combine the embedding distances of each component (doc string, signature, body) between them to determine whether they are functionality-equivalent. Our observation is the similarities among API doc strings and signatures of functionality-equivalent APIs. Second, we design a semi-automatic schema to migrate the PoC of existing vulnerabilities, thus substantially reducing manual costs. Our observation is that a vulnerability PoC consists of two parts: input construction (constructing a specific object) and API invocation (invoking the target API) where the former remains the same during migration and the latter often accounts for one or two lines of PoCs and can be semi-automatically migrated by the algorithm of parameter mapping [45].

We evaluate APISS on 180 widely used Java repositories using 10 existing vulnerabilities along with their PoCs. Additionally, we compare APISS's effectiveness with four state-of-the-art approaches of detecting vulnerabilities and two state-of-the-art approaches of retrieving functionality-equivalent APIs. Our evaluation results show that APISS is effective in detecting new vulnerabilities as the best of the baselines under comparison can detect only 58% of the vulnerabilities detected by APISS. Additionally, APISS effectively retrieves functionality-equivalent APIs, achieving a Top-1 Accuracy of 0.81 while the best of the baselines under comparison achieves only 0.55. APISS is highly efficient: its manual costs are within 10 minutes per vulnerability and the end-to-end runtime overhead of analyzing one candidate API is less than 2 hours. APISS has detected 170 new vulnerabilities and received 60 new CVE IDs, bringing high value to security practice.

In summary, this paper makes the following main contributions:

- To the best of our knowledge, we are the first to utilize API doc strings and signatures instead of focusing on API bodies to retrieve functionality-equivalent APIs and then detect functionality-specific vulnerabilities.
- We propose an effective technique of API mapping to retrieve functionality-equivalent APIs mainly based on the embedding distance of API doc strings and signatures, and achieve a Top-1 Accuracy of 0.81 while the best of the baselines under comparison achieves only 0.55.
- We design a semi-automatic schema to reduce the manual costs of migrating the PoC of existing vulnerabilities within 10 minutes per vulnerability.
- We implement a prototype of APISS and show its high value to security practice by detecting 179 new vulnerabilities and receiving 60 new CVE IDs.

2 Preliminaries

In this section, we provide definitions for three key concepts to improve accessibility for a broader audience.

CVE (Common Vulnerabilities and Exposures): It is maintained by the MITRE Corporation. CVEs are used to identify, define, and catalog publicly disclosed cybersecurity vulnerabilities. Once a vulnerability is discovered, developers can request a unique CVE ID from MITRE. Once assigned, this ID enables consistent reference to the vulnerability across databases and platforms. CVEs are widely used by security practitioners to monitor relevant vulnerabilities and track vulnerability patches and affected packages.

CWE (Common Weakness Enumeration): It is also maintained by MITRE. CWE provides a classification system for types of software vulnerabilities. As of now, there are 943 unique CWE types, offering a standardized framework for understanding software vulnerabilities.

PoC (Proof-of-Concept): A PoC is a minimal, verifiable demonstration that confirms the existence and exploitability of a specific security vulnerability. In our paper, we define a PoC as **a single unit test case that demonstrates such exploitability**. Due to the potential security risks, PoCs are often withheld by practitioners to prevent attackers from exploiting vulnerabilities before affected users can apply necessary fixes.

3 Motivation

In this section, we use a motivating example to introduce functionality-specific vulnerabilities and functionality-equivalent APIs. We further leverage this example to demonstrate why existing approaches fail to identify these vulnerabilities. Then we explain our observations on functionality-equivalent APIs and our insights of APISS.

3.1 A Motivating Example

Listing 1 and Listing 2 show a motivating example of two functionality-equivalent APIs with similar functionality-specific vulnerabilities, CVE-2023-27603[1] and CVE-2023-33484[2]. Specifically, CVE-2023-27603 is an existing Scala vulnerability whose PoC is available while CVE-2023-33484 is a Java vulnerability newly detected by APISS based on the former vulnerability.

Functionality-Specific Vulnerabilities. Functionality-specific vulnerabilities are defined as vulnerabilities that mainly occur in APIs with specific functionalities. These vulnerabilities are often caused by a lack of crucial operations during implementation, such as input validation. Listing 1 is the patch of an example functionality-specific vulnerability, CVE-2023-27603, which affects the `fileToUnzip` API in a widely used third-party repository, `Apache Linkis`. This vulnerability corresponds to a representative type of functionality-specific vulnerability, CWE-22 (i.e., improper limitation of a pathname to a restricted directory). Lines 16-21 in Listing 1 of this patch explain how this vulnerability is exploited and fixed. Specifically, the vulnerable API `fileToUnzip` forgets to check whether the file paths (`filePath` in Line 16) in the input zip file (`zipFilePath`) contain improper relative paths, e.g., "`..`", which has the risk of unzipping into a restricted directory (in Line 23).

Functionality-Equivalent APIs. Functionality-equivalent APIs refer to a collection of APIs that provide equivalent/similar functionalities, such as "unzip a zip file" or "parse a JSON file". Take Listing 1 and Listing 2 as examples. These two APIs, which come from two repositories (`Apache Linkis` and `Light4J`), respectively, perform the same functionality: unzipping an input zip file to a specific path.

Based on the preceding definitions, functionality-equivalent APIs often share similar or even the same functionality-specific vulnerabilities. For example, the `unzip` API in Listing 2 is also vulnerable to a path traversal vulnerability, CVE-2023-33484, and is fixed by invoking the `validateZipOut` method before unzipping. However, the implementations of these APIs might diverge substantially. For example, `fileToUnzip`'s source code spans only 27 lines while that of `unzip` spans 46 lines.

[1] `https://nvd.nist.gov/vuln/detail/CVE-2023-27603`
[2] `https://github.com/networknt/light-4j/issues/1760`

3.2 Limitations of Existing Approaches

Recurring-Vulnerability Detection Approaches. Some existing approaches [43, 26, 36] are designed to detect recurring vulnerabilities, which come from reused code or shared code logic. For example, MVP [43] utilizes hash values based on the code property graphs of known vulnerable APIs to match similar APIs in open-source repositories. However, these approaches rely heavily on matching the preceding implementation-based features. This reliance may produce high false negatives in detecting functionality-specific vulnerabilities while the affected APIs have diverse implementations.

Deep-Learning Approaches. Some other approaches [41, 52, 39, 8, 38, 37, 42] learn code features extracted from vulnerable APIs/methods including existing vulnerabilities. For example, Devign [52] uses a Graph Neural Network (GNN) to learn the code property graphs (e.g., control/data-flow graphs) of vulnerable APIs. These approaches can effectively detect vulnerabilities caused by incorrect implementation, such as use-after-free, because the source code of their affected APIs has specific patterns, e.g., a pointer operation after a `free` API invocation. However, these approaches are ineffective in detecting functionality-specific vulnerabilities because their affected APIs have diverse implementations without a common pattern. Additionally, these existing approaches also face limitations in generating PoCs for the detected vulnerabilities. A recent study [46] shows that deep-learning models, including LLMs, struggle to generate inputs to cover a specific code branch, let alone generate inputs to exploit a specific vulnerability.

3.3 Observation of Functionality-Equivalent APIs

Considering that an API's source code comprises not only its implementation (i.e., its body) but also a doc string and a signature, we explore the similarities and disparities among these elements in the APIs affected by existing functionality-specific vulnerabilities.

For example, in Listing 1 and Listing 2, we present a pair of functionality-equivalent APIs. Notably, while both APIs share comparable doc strings and signatures, their bodies diverge. Specifically, Line 7 in Listing 1 and Line 8 in Listing 2 represent their signatures. These signatures directly point out their intended functionality of `unzip` and include similar input parameters, such as the zip file's path/name and the destination directory for extraction. Moreover, their doc strings (Lines 1-6 in Listing 1 and Lines 1-7 in Listing 2) explain the purpose of these parameters, thus helping us classify them as functionality-equivalent APIs. However, in terms of their bodies (Lines 8-32 in Listing 1 and Lines 9-19 in Listing 2), there are noticeable disparities, e.g., their differing lengths (27 lines in the former and 46 lines in the latter).

3.4 Our Insight

Our observation reveals that functionality-equivalent APIs often share similarities in their doc strings and signatures, and disparities in their implementations (bodies). Based on this observation, in this paper, we propose to utilize API doc strings and signatures to retrieve functionality-equivalent APIs, thus contributing to detecting functionality-specific vulnerabilities.

A main characteristic of doc strings and signatures is that they are closer to natural language than code. Doc strings are written in natural language and serve the purpose of explaining API functionalities and the purpose of each parameter. Signatures are part of the API source code, primarily consisting of API names and parameters without structured information, e.g., control/data-flow dependencies.

Listing 1 The Vulnerability Patch of CVE-2023-27603 (an Existing Scala Vulnerability).[3]

```
/**
 * @param zipFilePath
 *   Path of the zip file to be extracted
 * @param unzipDir
 *   Path of the destination directory
 */
def fileToUnzip(zipFilePath: String, unzipDir: String): Unit = {
    val zipIn = new ZipInputStream(new FileInputStream(zipFilePath))
    Utils.tryFinally {
    val destDir = new File(unzipDir)
    if (!destDir.exists()) {
        destDir.mkdir()
    }
    var entry = zipIn.getNextEntry
    while (entry != null) {
-       val filePath = destDir.getPath + File.separator + entry.
    getName
+       var entryName = entry.getName
+       if (entryName.contains(".." + File.separator)) {
         throw new IOException("Zip entry contains illegal characters:
              " + entryName)
+       }
+       val filePath = destDir.getPath + File.separator + entryName
         if (!entry.isDirectory) {
           extractFile(zipIn, filePath)
         } else {
           val dir = new File(filePath)
           dir.mkdir
         }
         zipIn.closeEntry()
         entry = zipIn.getNextEntry
       }
    }(IOUtils.closeQuietly(zipIn))
}
```

Considering the preceding characteristic of doc strings and signatures, we propose to leverage a Large Language Model (LLM) to retrieve functionality-equivalent APIs due to an LLM's significant advancements in natural language and code comprehension. Our proposed approach involves separately embedding each combination of elements (doc string, signature, and body) of a given API as its embedding features. For each API pair, we compute the distances between their respective embedding features and average them as their weighted distances. Notably, our empirical results in Section 5.4 indicate that the embedding features with API bodies have a negligible contribution.

Advantages Over Existing API Mapping Approaches. Our proposed approach has two main advantages when compared to existing work [29, 11, 6, 50, 47], which models the task of retrieving functionality-equivalent APIs as an API mapping task. First, existing

[3] https://github.com/apache/linkis/pull/4279/commits/b3a16c4f1d4ab66d412b0d55bd8c0c35b48775b5

Listing 2 The Vulnerability Patch of CVE-2023-33484 (a Newly-Detected Java Vulnerability).[4]

```
/**
 * Unzips the specified zip file to the specified destination
     directory.
 * Replaces any files in the destination, if they already exist.
 * @param zipFilename the name of the zip file to extract
 * @param destDirname the directory to unzip to
 * @throws IOException IOException
 */
public static void unzip(String zipFilename, String destDirname) {
... //17 lines
    final Path destFile = Paths.get(destDir.toString(), file.toString
        ());

+   // validate unzip cross file
+   validateZipOutputFile(zipFilename, new File(destDirname,file.
    toString()), new File(destDirname));

    if(logger.isDebugEnabled()) logger.debug("Extracting file %s to %
        s", file, destFile);
    Files.copy(file, destFile, StandardCopyOption.REPLACE_EXISTING);
    return FileVisitResult.CONTINUE;
... //19 lines
}
```

approaches are less effective. Similar to vulnerability-detection approaches, existing API mapping approaches also overemphasize API bodies without being able to handle diverse implementations of functionality-equivalent APIs. Second, existing approaches are less generalizable because they are trained/fine-tuned on specific datasets, e.g., mapping first-party APIs in Java libraries to C# ones [29, 11, 6, 50] or Swift ones [47].

4 Approach

In this section, we propose APISS, the first approach to detect functionality-specific vulnerabilities via retrieving functionality-equivalent APIs. Unlike existing deep-learning and recurring-vulnerability detection approaches, APISS utilizes API doc strings and signatures to retrieve functionality-equivalent APIs and subsequently detect and exploit new functionality-specific vulnerabilities.

The workflow of APISS is illustrated in Figure 1. The input of APISS consists of two parts, an existing vulnerability (a.k.a., source vulnerability), including its affected API and its Proof-of-Concept (PoC), and a series of candidate APIs on which APISS intends to detect functionality-specific vulnerabilities if any exist. The output of APISS is a series of new vulnerabilities, each including an affected API and its corresponding PoC.

As shown in Figure 1, APISS contains two steps. First, APISS retrieves functionality-equivalent APIs among these candidate APIs by leveraging the embedding of each combination of API elements (doc string, signature, and body). Second, for each retrieved functionality-

[4] https://github.com/networknt/light-4j/pull/1768/commits/dff1dcb7f87622f86c953c9a7e639077d694a291

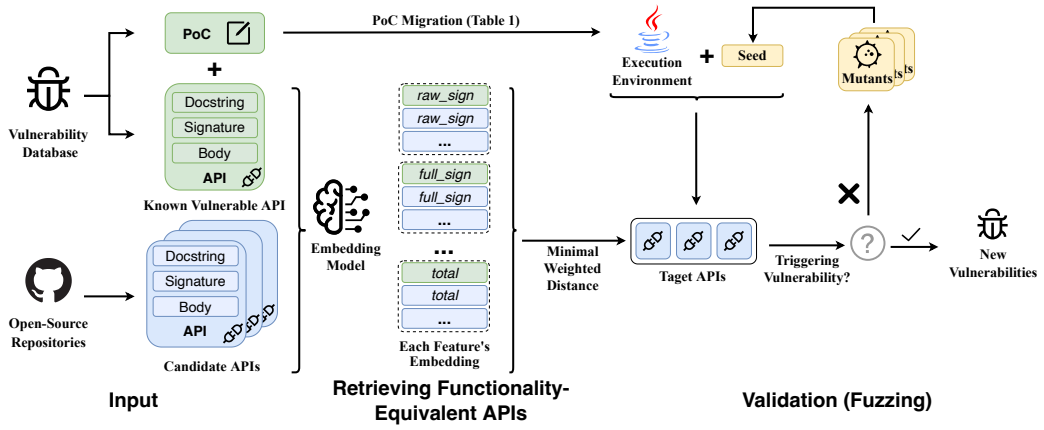


Figure 1 High-level framework of APISS.

equivalent API, APISS migrates the PoC of the source vulnerabilities. Additionally, for functionality-equivalent APIs corresponding to specific vulnerability types, APISS employs a fuzzing engine to conduct fuzzing based on the migrated PoC.

4.1 Retrieving Functionality-Equivalent APIs

Retrieving functionality-equivalent APIs is the basis of APISS as these APIs are more susceptible to vulnerabilities (similar to the given existing vulnerability) than other existing APIs.

Given that the source code of an API comprises three elements (its doc string, signature, and body), our key insight for retrieving functionality-equivalent APIs is to combine the embedding distances of these API elements. Specifically, we mainly focus on the embedding distances corresponding to API doc strings and signatures due to their similarities among functionality-equivalent APIs (as shown in Section 3.3). Considering the significant advancements achieved by Large Language Models (LLMs), we propose to leverage an LLM to embed various combinations of these three elements, termed as API features. To further investigate the contribution of each element, we also consider bodies when selecting features. In Section 5.4, we show that features including API bodies make only marginal contributions.

Specifically, we consider the following eight features:

- *raw_sign*: the raw signature, e.g., `public static void unzip (String zipFilename, String destDirname)` in Listing 2
- *full_sign*: the full signature (including the package and class names), e.g., `com.networknt.utility.NioUtils.unzip (String zipFilename, String destDirname)` in Listing 2
- *doc*: the doc string, e.g., Lines 1-7 in Listing 2
- *body*: the body, e.g., Lines 9-19 in Listing 2
- *raw_sb*: the combination of *raw_sign* and *body*
- *raw_ds*: the combination of *raw_sign* and *doc*
- *full_ds*: the combination of *full_sign* and *doc*
- *total*: the combination of *full_sign*, *doc*, and *body*

Pseudocode. Algorithm 1 presents the pseudocode of APISS' module of retrieving functionality-equivalent APIs. The algorithm takes as input a source API of an existing vulnerability ($srcAPI$), a set of candidate APIs ($candidateAPIs$), the weight of each feature ($\mathcal{W}_i$), and the desired number of functionality-equivalent APIs (N). This algorithm outputs N functionality-equivalent APIs associated with $srcAPI$.

Algorithm 1 Retrieving Functionality-Equivalent APIs.

input : $srcAPI$, a source API (of an existing vulnerability)
input : $candidateAPIs$, candidate APIs
input : $\mathcal{W}_i$, the weight of each feature
input : N, target number of functionality-equivalent APIs
output : $equivAPIs$, $srcAPI$'s functionality-equivalent APIs

```
// Obtain one code line's minimal depth in AST
1 Function getEmbeddings (API) :
2     features ← [raw_sign, full_sign, doc, body, raw_sb, raw_ds, full_ds, total];
3     for i ∈ features do
          // Each feature's normalized embedding
4         E_i(API) ← LLM(API_i).last_hidden_states[−1];
5         Ê_i(API) = E_i(API) / ||E_i(API)||;
6     E(API) ← [Ê_1, ..., Ê_F];
7     return E(API)
// Searching the top N closest APIs
8 E(src) ← getEmbeddings(srcAPI);
9 for dst ∈ candidateAPIs do
10    E(dst) ← getEmbeddings(dst);
11    D(src, dst) ← Σ_{i∈features} W_i * ||E_i(src) − E_i(dst)||_2;
12 equivAPIs ← argmin_{dst}[N]{D(src, dst)};
13 return equivAPIs;
```

In Lines 1–7, we compute each feature's embedding of a given API. In Line 4, we compute the raw embedding of each API feature, $\mathcal{E}_i(API)$. Then in Line 5, we normalize each embedding vector, denoted as $\hat{\mathcal{E}}_i(API)$. Our rationale for normalization is to ensure that the distances of each feature are within the same ranges.

In Lines 8–13, we compare the API embeddings of $srcAPI$ with those of $candidateAPIs$. In Lines 8–10, we generate the embeddings of both the $srcAPI$ and each candidate API using the `getEmbeddings` method. Then in Line 10, we define the distance of each pair of API embeddings as the weighted distance of each feature. Here, the distance of each feature is defined as the L2 norm [27, 12] of their embeddings' subtraction, $||\mathcal{E}_i(src) - \mathcal{E}_i(dst)||_2$.

Weight Optimization. The performance of Algorithm 1 depends on the selection of feature weights ($\mathcal{W}_i$). To determine the optimal set of values, we employ a global optimization technique. Specifically, we utilize the Basin-hopping algorithm [24], which has been widely applied in high-dimensional search problems [33, 48]. The optimization is performed on a dataset comprising Java and C# APIs (as shown in Table 6), and the resulting weights are subsequently used in the vulnerability detection process.

Acceleration by Vector Database. To enhance retrieval efficiency and mitigate the runtime costs of Algorithm 1, we utilize a vector database [20, 14]. Algorithm 1 requires calculating $|srcAPIs|$ * $|candidateAPIs|$ pairs of distances for all $|srcAPIs|$ as the distance between the src API and each dst API is computed in Lines 9-11. On the contrary, a vector database substantially reduces the runtime cost to $|srcAPIs| + |candidateAPIs|$ by storing

and retrieving the embeddings of $candidateAPIs$ while querying each $srcAPI$. During implementation, we choose Faiss[5] as our vector database and modify the selection criteria equivalently in Lines 11-12:

$$\begin{aligned} \textbf{Define: } \hat{\mathcal{E}}(API) = &< Weight_1 * \hat{\mathcal{E}}_1, \ldots, Weight_F * \hat{\mathcal{E}}_F > \\ &\underset{dst \in APIs}{argmin} \sum_{i \in Features} Weight_i * ||\hat{\mathcal{E}}_i(src) - \hat{\mathcal{E}}_i(dst)||_2 \\ &\Leftrightarrow \underset{dst}{argmin} \, ||\hat{\mathcal{E}}(src) - \hat{\mathcal{E}}(dst)||_2 \end{aligned} \tag{1}$$

where $\mathcal{E}_i(API)$ is the i-th feature embedding of API, $\hat{\mathcal{E}}_i(API)$ is the normalized embedding, and $\hat{\mathcal{E}}(API)$ is the concatenated embeddings of all API's features. Thus, we store only $\hat{\mathcal{E}}(dst)$ of all candidate APIs in our vector database.

4.2 Migrating PoCs of Existing Vulnerabilities

In this step, we migrate the PoC for each retrieved functionality-equivalent API from that of its source API. Our goal is to reduce the substantial human labor of manually constructing PoCs. Specifically, the input for this step includes a PoC from a source API and one of its functionality-equivalent APIs, while the output is an executable PoC for this functionality-equivalent API.

Our Observation. Our solution leverages our observation that a functionality-specific vulnerability's PoC generally consists of two parts: input construction and API invocation. Input construction, i.e., creating specific objects or data structures, remains consistent across functionality-equivalent APIs. For example, in the PoCs of CVE-2023-27603 and CVE-2023-33484 (Listing 3), the construction of a zip file containing a malicious file is identical (Lines 12–30 in both PoCs). This zip file attempts to access a restricted path using the relative address "../../a/b/c/poc.txt". On the contrary, API invocation, i.e., the actual invocation of the targeted API, varies minimally and comprises a smaller code portion of the PoC. Lines 7–8 demonstrate that the "unzip" method is called in the former CVE and "NioUtils.unzip" is called in the latter, both aiming to unzip the malicious zip file. Notably, the parameters for these invocations, including the path of the zip file and the targeted extraction path, remain identical/similar across different APIs.

Migration Procedure. The preceding observation highlights the feasibility of adapting PoCs between functionality-equivalent APIs with minimal adjustment. Thus, we design a semi-automatic schema to migrate the PoC of existing vulnerabilities with two steps. First, for each type of vulnerability, we collect a representative affected API with its PoC, and manually create a heuristic-based template for PoC migration. The details are listed in Table 1. Second, for the API invocation part, we migrate the usage of each source API into the destination API based on a simple parameter-matching algorithm [45]. As for those cases where automatic migration fails, we manually migrate them. Specifically, we look up the documents in their corresponding repositories or directly consult with search engines.

[5] `https://faiss.ai/index.html`

Listing 3 The PoC of CVE-2023-27603 and CVE-2023-33484.

```
//API invocation
public static void main(String[] args) throws IOException {
    zip();
    String zipFilename = "./testData/unzip/poc.zip";
    String destDirname = "./testData/unzip";
    //The core lines of both PoCs
    unzip(zipFilename, destDirname)(fs) // CVE-2023-27603
    NioUtils.unzip(zipFilename, destDirname);  // CVE-2023-33484
}

// Input construction
public static void zip() {
    ZipOutputStream zos = null;
    try {
        zos = new ZipOutputStream(new FileOutputStream( "./testData/
            unzip/poc.zip"));
        String srcFile = "../../a/b/c/poc.txt";
        String destFile = "./testData/unzip/poc.txt";
        zos.putNextEntry(new ZipEntry(srcFile));
        FileInputStream in = new FileInputStream(destFile);
        int len;
        byte[] buf = new byte[1024];
        while ((len = in.read(buf)) != -1) {
            zos.write(buf, 0, len);
        }
        zos.closeEntry();
        in.close();
    } catch (Exception e) {
        throw new RuntimeException("zip␣error␣from␣ZipUtils", e);
    } ...//9 Lines
}
```

API Fuzzing. We employ API fuzzing to further exploit functionality-specific vulnerabilities in functionality-equivalent APIs whose source APIs are associated with specific vulnerability types. Our rationale is that identical inputs may not trigger the same or similar vulnerabilities in different APIs if their functionalities diverge slightly. Thus, APISS employs a fuzzer to generate more inputs to exploit potential vulnerabilities. We focus our fuzzing efforts on APIs whose inputs are amenable to mutation, as some constructed inputs cannot be effectively fuzzed. The vulnerabilities whose affected APIs require fuzzing are detailed in the "Need Fuzzing" column of Table 1.

5 Evaluation

Our evaluation answers the following three research questions about APISS:

- **RQ1**: How effective is APISS in detecting real-world vulnerabilities when compared with baseline approaches?
- **RQ2**: How effective is APISS' API mapping module when compared with baseline approaches?
- **RQ3**: What are the manual/runtime costs of APISS?

Table 1 Heuristics of PoC Migration for Various Types of Vulnerabilities.

CWE Type	Characteristics of PoC	Fuzzing
CWE-121	Reuse the deep-nested JSON file in the "load" API	✓
CWE-22	Reuse the malicious zip file in the "unzip" API	✗
CWE-400/835	Reuse the test case with a vulnerable method sequence	✓
CWE-200	Reuse the malicious file in the "download" API	✗
CWE-502/611	Reuse the vulnerable JSON/XML files in the "parse" API	✓
CWE-94	Reuse the injected code in the execution API	✗
CWE-358	Reuse the initialization API of `collections` with various sizes	✓
CWE-434	Reuse the malicious file in the "upload" API	✗
CWE-275/79	Reuse the malicious XSS request in the "content_editing" API	✗
CWE-122/917	Reuse the malicious string expression in the "parse" API	✗

5.1 Background Information of Our Manual Efforts

Considering that the reliability of our detected vulnerabilities and the costs of our manual efforts highly depend on the expertise of our engineers, we explain the background of our manual efforts here. Evaluating APISS involves 10 software engineers, each with at least two years of Java development and one year of vulnerability-related tasks, e.g., vulnerability mining and fuzzing. Specifically, our engineers come from one of the major IT companies. This company has more than 20,000 engineers and more than 100 papers in top-tier conferences and journals on software engineering and security. The entire process of manually verifying existing vulnerabilities and their PoCs takes about 30 human-hours, and the entire process of detecting, validating, and reporting new vulnerabilities takes about 50 human-hours.

5.2 Dataset Collection

To evaluate APISS, we collect our dataset with two components: the affected APIs and PoCs of existing vulnerabilities, and a set of candidate APIs.

Vulnerability and PoC Collection. We collect source vulnerabilities from a Java vulnerability dataset used by VulFixMiner [51], MiDas [30], and CompVPD [10], including 1,359 vulnerabilities from 127 Java repositories. For each vulnerability in this dataset, we search for its related issue from the commit message in its patch, and the reference links in its vulnerability reports in NVD[6]. Then, for each issue with an executable PoC, we manually validate this PoC. For each type of functionality vulnerability, we choose the earliest PoC as the template one for migration (as shown in Table 2).

Candidate-API Collection. We extract 42,612 candidate APIs from 180 randomly sampled Java repositories on GitHub. Specifically, we adopt tree-sitter[7], a widely-used parser, to extract the elements of each API, including doc string, signatures, and bodies.

[6] `https://nvd.nist.gov`
[7] `https://tree-sitter.github.io/tree-sitter/`

Table 2 Each CWE Type's Source API.

CWE Type	CVE ID	Repository	API
CWE-121	CVE-2020-36518	jackson-databind	createContextual
CWE-22	CVE-2023-27603	apache_linkis	fileToUnzip
CWE-400/835	CVE-2018-18854	spray-json	JsObject
CWE-200	CVE-2021-21364	swagger-codegen	prepareDownloadFile
CWE-502/611	CVE-2018-21234	jodd	map2bean
CWE-94	CVE-2021-41269	cron-utils	parse
CWE-358	CVE-2018-1275	spring-framework	filterSubscriptions
CWE-434	CVE-2019-17352	jfinal	parse
CWE-275/79	CVE-2021-23899	json-sanitizer	sanitizeString
CWE-122/917	CVE-2022-24847	geoserver	getJNDIDataSource

Table 3 New Vulnerabilities Detected by APISS (TP and FP are True Positives and False Positives).

CWE Type	New CVE ID	Confirmed Without CVE ID	During Reporting	Reported by Others	TP	FP	Precision
CWE-121	24	8	59	7	98	53	0.65
CWE-22	2	7	2	0	11	31	0.26
CWE-400/835	14	0	15	0	29	8	0.78
CWE-200	2	0	0	0	2	37	0.05
CWE-502/611	1	2	0	3	6	24	0.20
CWE-94	3	0	0	2	5	33	0.13
CWE-358	0	0	11	0	11	17	0.39
CWE-434	3	0	0	0	3	15	0.17
CWE-275/79	10	0	0	2	12	12	0.50
CWE-122/917	1	1	0	0	2	33	0.06
Total	60	18	87	14	179	263	0.40

5.3 RQ1: How effective is APISS in detecting real-world vulnerabilities when compared with baseline approaches?

This research question aims to show the effectiveness of APISS in terms of detecting new vulnerabilities in real-world settings. Then, we take the vulnerabilities detected by APISS as ground truths to investigate the effectiveness of baselines (i.e., how many of them can be detected by our baselines).

Baselines. We compare APISS with three state-of-the-art deep-learning models: GGNN [23] (used by Devign [52], Funded [38], and ReVeal [8]), GCN [21] (used by IVDetect [22]), and RGCN [9] (used by Vu1SPG [49]). Although these models are originally trained on C/C++ vulnerability datasets, we retrain them on the preceding Java vulnerability dataset to ensure a fair comparison in detecting Java vulnerabilities. We also compare APISS with another state-of-the-art recurring-vulnerability detection approach, MVP [43]. We exclude AHPH [26] and ReurScan [36] as they are specialized for detecting specific recurring vulnerabilities (non-functionality ones).

Table 4 Vulnerabilities Detected by APISS with CVE IDs.

CVE ID	CWE Type	Repository	CVE ID	CWE Type	Repository
CVE-2023-33484	CWE-22	light-4j	CVE-2023-36067	CWE-121	groovy-json
CVE-2023-29638	CWE-79	WinterChenS/my-site	CVE-2023-36068	CWE-121	groovy-xml
CVE-2023-29641	CWE-79	pandao/editor	CVE-2023-36068	CWE-121	groovy-xml
CVE-2023-29643	CWE-79	perfree/PerfreeBlog	CVE-2023-35104	CWE-121	logansquare
CVE-2023-29636	CWE-79	ZHENFENG13/My-Blog	CVE-2023-35103	CWE-121	progsbase
CVE-2023-29639	CWE-79	ZHENFENG13/My-Blog	CVE-2023-35110	CWE-121	jjson
CVE-2023-29637	CWE-79	Qbian61/forum-java	CVE-2023-35116	CWE-121	jackson
CVE-2023-29635	CWE-434	Antabot/White-Jotter	CVE-2023-36073	CWE-94	Icefrog-core
CVE-2023-29652	CWE-434	RuoYi-Vue	CVE-2023-36074	CWE-94	bus-core
CVE-2023-33544	CWE-22	hawtio	CVE-2023-36071	CWE-121	com.grunka.json
CVE-2023-33546	CWE-121	janino-compiler/janino	CVE-2023-36075	CWE-611	ueboot-core
CVE-2023-33695	CWE-200	hutool	CVE-2023-41574	CWE-835	json-simple
CVE-2023-34670	CWE-200	nutzam/nutz	CVE-2023-42277	CWE-835	hutool-json
CVE-2023-29874	CWE-79	LovebuildJ/book-manager	CVE-2023-42276	CWE-835	hutool-json
CVE-2023-29875	CWE-79	LovebuildJ/book-manager	CVE-2023-42278	CWE-835	hutool-json
CVE-2023-29876	CWE-79	LovebuildJ/book-manager	CVE-2023-47382	CWE-400	aalto-xml
CVE-2023-29877	CWE-275	LovebuildJ/book-manager	CVE-2023-47381	CWE-400	aalto-xml
CVE-2023-29878	CWE-434	vran-dev/databasir	CVE-2023-50540	CWE-121	json-path
CVE-2023-34609	CWE-121	flexjson	CVE-2023-50546	CWE-121	json-path
CVE-2023-34610	CWE-121	json-io	CVE-2023-50570	CWE-835	ipaddress
CVE-2023-34612	CWE-121	ph-json	CVE-2023-50571	CWE-94	easy-rules-mvel
CVE-2023-34611	CWE-121	mjson	CVE-2023-50572	CWE-917	jline-groovy
CVE-2023-34615	CWE-121	JSONUtil	CVE-2023-51079	CWE-835	mvel
CVE-2023-34613	CWE-121	sojo	CVE-2023-51076	CWE-835	jxls
CVE-2023-34614	CWE-121	jsonij	CVE-2023-50635	CWE-835	hutool-all
CVE-2023-34616	CWE-121	progsbase	CVE-2023-50636	CWE-835	hutool-all
CVE-2023-34617	CWE-121	genson	CVE-2023-51074	CWE-121	json-path
CVE-2023-34620	CWE-121	hjson	CVE-2023-51075	CWE-835	hutool-core
CVE-2023-34623	CWE-121	jtidy	CVE-2023-51080	CWE-835	hutool-core
CVE-2023-34624	CWE-121	htmlcleaner	CVE-2023-51084	CWE-835	hyavijava

New Vulnerabilities Detected by APISS. At the time of writing, APISS has detected 179 new vulnerabilities. As shown in Table 3, 60 have received new Common Vulnerabilities & Exposures Identifiers (CVE IDs), 18 have been confirmed by developers without CVE IDs, 87 are currently under review, and 14 have been reported by others. Vulnerabilities with CVE IDs are listed in Table 4. Among the 60 vulnerabilities with new CVE IDs, 15 are classified as high-risk and 3 as critical based on the CVSS Base Score [35]. This result highlights APISS' effectiveness in detecting new vulnerabilities, bringing high value to security practice.

Difference Among Various Vulnerability Types. Table 3 shows that the majority of vulnerabilities detected by APISS belong to CWE-121 (Stack-based Buffer Overflow) and CWE-400/835 (Uncontrolled Resource Consumption/Infinite Loop). According to our manual investigation of the PoCs of these vulnerabilities, they are triggered by unexpected JSON inputs, especially those that are deeply nested. For example, Listing 5 shows the PoC of CVE-2023-34610, a new vulnerability (belonging to CWE-121) detected by APISS. Its input is constructed in Lines 3-22 and has the following format: "$[[\ldots[\ 0\ ]]\ldots]$" (9999 and 9999).

APISS' False Positives. Table 3 also shows the false-positive numbers of each type of vulnerabilities. APISS incorrectly identifies 263 false-positive vulnerable APIs, achieving a high precision score of 0.40. It is mainly because we discard functionality-equivalent APIS for which APISS fails to generate its PoC.

Listing 4 An Example False-Positve Vulnerability.

```
import org.junit.Test;
import cn.hutool.core.net.*;
import java.util.List;
public class Ipv4UtilFuzzerList {
    // oom
    @Test
    public void listFuzzerTest() {
        try {
            List<String> result = Ipv4Util.list("213.3.13.2/2/",
                false);
        } catch (Exception e) {
        }
    }
}

-------------------------------------------------
Error messages:
java.lang.OutOfMemoryError: Java heap space
        at cn.hutool.core.net.Ipv4Util.list(Ipv4Util.java:141)
```

An Example of a False-Positive Vulnerability. Listing 4 presents an example of a false-positive vulnerability. The issue arises due to the use of a subnet mask of `/2`, which results in an excessively large capacity (i.e., 1,073,741,822) when initializing an ArrayList. The maintainer has acknowledged that a validation check could be added to Ipv4Util to handle such illogical inputs and prevent errors.

However, the maintainer does not classify this case as a vulnerability, stating that "A subnet mask is typically used to divide an IP address range into smaller networks, so a `/2` subnet is too large for any practical use, particularly for a public IP address such as `213.3.13.2`."

It is worth noting that false positives count for a small portion because most of them are crash-like vulnerabilities, which can be automatically verified. Such false positives rejected by repository maintainers account for a small fraction (i.e., less than 20%). Given the overall end-to-end effort (80 human-hours, as detailed in Section 5.1) and the effectiveness of our approach (identifying 179 new vulnerabilities and 60 new CVE IDs), the occurrence of such false positives remains within an acceptable range.

Comparison With Baselines. Table 5 illustrates the proportion of vulnerabilities detected by APISS and also detected by existing baselines. Overall, our baseline approaches can detect at most 58% of these vulnerabilities while MVP can detect 41% of them (corresponding to their recall values). Additionally, these baseline approaches encounter substantial false positives. MVP achieves a precision score of only 0.41 while the precision scores of existing deep-learning approaches (GCN, RGCN, and GGNN) are 0.02, 0.04, and 0.03, respectively. Such false positives will lead to a substantial manual costs when these identified vulnerabilities are validated. This result indicates that APISS is more effective in detecting functionality-specific vulnerabilities when compared with baselines. The main reason for their ineffectiveness is

[8] `https://nvd.nist.gov/vuln/detail/CVE-2023-34610`

Listing 5 The PoC of CVE-2023-34610.[8]

```
public class PoC {
    //Input Construction
    public final static int TOO_DEEP_NESTING = 9999;
    public final static String TOO_DEEP_DOC = _nestedDoc(
        TOO_DEEP_NESTING, "[␣", "]␣", "0");

    public static String _nestedDoc(int nesting, String open, String
        close, String content) {
        StringBuilder sb = new StringBuilder(nesting * (open.length()
            + close.length()));
        for (int i = 0; i < nesting; ++i) {
            sb.append(open);
            if ((i & 31) == 0) {
                sb.append("\n");
            }
        }
        sb.append("\n").append(content).append("\n");
        for (int i = 0; i < nesting; ++i) {
            sb.append(close);
            if ((i & 31) == 0) {
                sb.append("\n");
            }
        }
        return sb.toString();
    }
    //API invocation
    public static void main(String[] args) {
        String jsonString = TOO_DEEP_DOC;
        JsonReader.jsonToJava(jsonString);
    }
}
```

their overemphasis on API bodies disregarding the various implementations of functionality-equivalent APIs. On the contrary, APISS utilizes the similarities among API doc strings and signatures, thus successfully detecting these vulnerabilities.

5.4 RQ2: How effective is APISS' API mapping module when compared with baseline approaches?

This research question evaluates the effectiveness of APISS' API mapping module. In the evaluation, we choose two representative LLMs as the embedding model in APISS, ChatGPT (text-embedding-ada-002) [1] and CodeLLaMa (with 7B parameters) [34]. For each embedding model, we evaluate the effectiveness of each LLM with five feature combinations (configurations): `All Features` (using the optimal feature weights determined by Basin-hopping), `Signature+Doc String`, `Signature Only`, `Doc String Only`, and `Body Only`.

To quantitatively investigate the contribution of each embedding feature, we conduct Shapley Additive exPlanations (SHAPs) [3, 32], a game-theoretic technique to interpret the outputs of machine learning models. SHAP values help us understand how each feature contributes to APISS' effectiveness, thus providing evidence of focusing on doc strings and signatures instead of bodies.

Table 5 Proportions of Vulnerabilities Detected by Baselines.

CWE Type	MVP				GCN			
	TP	FP	Precision	Recall	TP	FP	Precision	Recall
CWE-121	52	50	0.51	0.53	18	874	0.02	0.18
CWE-22	5	4	0.56	0.45	4	141	0.03	0.36
CWE-400/835	10	9	0.53	0.34	5	116	0.04	0.17
CWE-200	0	8	0.00	0.00	0	1	0.00	0.00
CWE-502/611	0	4	0.00	0.00	3	144	0.02	0.50
CWE-94	2	0	1.00	0.40	1	44	0.02	0.20
CWE-358	4	20	0.17	0.36	1	75	0.01	0.09
CWE-434	0	0	0.00	0.00	0	0	0.00	0.00
CWE-275/79	0	9	0.00	0.00	0	0	0.00	0.00
CWE-122/917	0	3	0.00	0.00	0	0	0.00	0.00
Total	73	107	0.41	0.41	32	1,395	0.02	0.18

CWE Type	RGCN				GGNN			
	TP	FP	Precision	Recall	TP	FP	Precision	Recall
CWE-121	66	2,318	0.03	0.67	45	1,740	0.03	0.46
CWE-22	9	43	0.17	0.82	9	85	0.10	0.82
CWE-400/835	16	35	0.31	0.55	5	59	0.08	0.17
CWE-200	1	5	0.17	0.50	0	1	0.00	0.00
CWE-502/611	3	1	0.75	0.50	3	2	0.60	0.50
CWE-94	2	1	0.67	0.40	1	71	0.01	0.20
CWE-358	7	44	0.14	0.64	2	77	0.03	0.18
CWE-434	0	1	0.00	0.00	0	0	0.00	0.00
CWE-275/79	0	38	0.00	0.00	0	44	0.00	0.00
CWE-122/917	0	6	0.00	0.00	0	0	0.00	0.00
Total	104	2,492	0.04	0.58	65	2,079	0.03	0.36

Baselines. We select two state-of-the-art API mapping approaches as our baselines, Api2Api [29] and SAR [6]. Both approaches employ a seeding step that is critical to their performance. For fair comparison, we evaluate all three seeding strategies (used in their paper): Random Seeds, k-Fold Seeds ($k = 1, 2, 3, 4$), and Signature-Based Seeds. For each strategy, we choose the optimal seed number that maximizes accuracy, based on the highest reported results in their paper [6].

Dataset. We use the same dataset, Java2CSharp [18], as our baselines. This dataset maps Java First-Party APIs to functionality-equivalent C# First-Party APIs. It includes 1570 Java APIs, 2457 C# APIs, and 860 pairs of functionality-equivalent APIs.

Given that the API source code in Java2CSharp is not available and the SAR repository[9] does not provide complete details of API bodies and doc strings, we recollect all First-Party Java and C# APIs from *jdk8u40*[10] and *.NET 4.8. for Windows January 2020 Update*[11].

[9] `https://github.com/bdqnghi/SAR_API_mapping`
[10] `https://hg.openjdk.org/jdk8u/jdk8u40/jdk/file/c7bbaa04eaa8/src`
[11] `https://referencesource.microsoft.com/DotNet48ZDP2.zip`

Table 6 The Effectiveness of API Mapping.

Baselines	Configuration	Top-1	Top-5
Api2Api	Random Seeds	0.19	0.24
	1-Fold Seeds	0.24	0.35
	2-Fold Seeds	0.34	0.35
	3-Fold Seeds	0.37	0.51
	4-Fold Seeds	0.43	0.64
	Signature-based Seeds	0.35	0.41
SAR	Random Seeds	0.44	0.50
	1-Fold Seeds	0.36	0.39
	2-Fold Seeds	0.45	0.50
	3-Fold Seeds	0.54	0.66
	4-Fold Seeds	0.59	0.77
	Signature-based Seeds	0.54	0.75
APISS-CodeLLaMa	All Features	**0.69**	**0.78**
	Signature+Doc String	**0.69**	**0.78**
	Signature Only	0.65	0.70
	Doc String Only	0.02	0.03
	Body Only	0.19	0.31
APISS-ChatGPT	All Features	**0.81**	**0.84**
	Signature+Doc String	0.74	0.78
	Signature Only	**0.81**	**0.84**
	Doc String Only	0.34	0.44
	Body Only	0.40	0.58

For each configuration of baselines (Api2Api and SAR), we choose the optimal seed number that maximizes accuracy.

Metrics. To keep the same setting with our baselines, we use Top-k Accuracy ($k = 1, 5$) as our evaluation metric. Top-k Accuracy measures the accuracy of API mappings by assessing whether the functionality-equivalent C# API for a given Java API appears within the top k positions of the predicted API list.

Results of Top-k Accuracy. Table 6 shows the Top-k Accuracy results of APISS and baselines. APISS substantially improves the API mapping accuracy with an increase of 22% in Top-1 Accuracy ($0.81 - 0.59$) and an increase of 7% in Top-5 Accuracy ($0.84 - 0.77$). This improvement is attributed to APISS' use of LLMs, which effectively comprehend API functionalities.

We also observe that comparing the embeddings of only signatures achieves higher accuracy than our baselines, while comparing the embeddings of only doc strings or bodies achieves relatively lower accuracy. This result indicates that API signatures are the most critical feature for API mapping than doc strings or bodies.

Explanation of Each Feature's Contribution. The contribution of each feature in APISS' embedding models, ChatGPT and CodeLLaMa, are depicted in Figure 2 and Figure 3. Each feature's contribution percentage is noted within brackets, and the scatter points represent the feature's contribution on each input API.

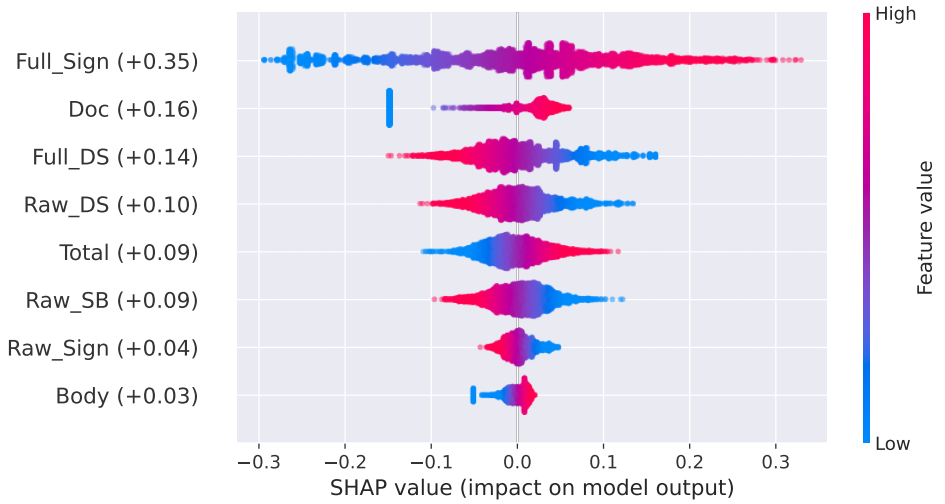


Figure 2 Each Feature's Contribution to API Mapping (Using ChatGPT's Embedding Model).

Both figures reveal that the three most important features are *full_sign*, *doc*, and *Full_DS*. These results highlight that both API doc strings and signatures (and their combination) mainly contribute to API mapping, while the API body has a negligible contribution (3% and 2%). Thus, the preceding results effectively support our proposal of utilizing API doc strings and signatures (instead of bodies) in API mapping.

Additionally, we notice two main differences from these figures. First, documentation strings show high value in smaller models in API mapping. For example, the most important feature for ChatGPT is *full_sign* (API signatures) while that of CodeLLaMa is *full_ds*, the combination of API doc strings and signatures. Second, features that include full signatures, which include package and class names (e.g., `com.networknt.utility.NioUtils`), are more effective than those with raw signatures. This result suggests that this contextual information enhances LLM's effectiveness in understanding API functionalities.

5.5 RQ3: What are the manual/runtime costs of APISS?

In this research question, we evaluate the manual and runtime costs of APISS. For manual costs, we detail the steps in the process where engineers detect new vulnerabilities, specifying their average time consumption. As for the runtime costs, we measure the end-to-end time costs of APISS on two main processes: (1) retrieving functionality-equivalent APIs, and (2) migrating PoCs and exploiting new vulnerabilities.

We perform all the evaluations on the system of Ubuntu 20.04. We use one Intel(R) Xeon(R) Gold 6248R@3.00GHz CPU, which contains 64 cores and 512GB memory. We use 1 Tesla A100 PCIe GPU with 40GB memory for API embedding.

Background Information of Our Manual Efforts. The end-to-end effectiveness of APISS highly depends on the expertise of our software engineers. Our evaluation, in which we have detected 179 new vulnerabilities, includes five software engineers. Each engineer has at least two years of experience in Java development and one year in fuzzing. The entire process of their manual costs takes approximately 50 man-days.

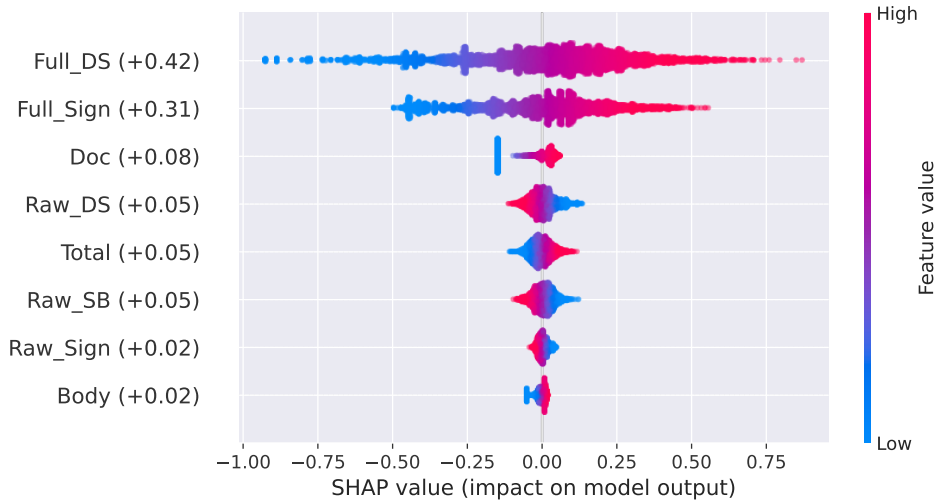


Figure 3 Each Feature's Contribution to API Mapping (Using CodeLLaMa for Embedding).

The Manual Costs of APISS. Figure 4 depicts the detailed workflow of how our engineers are involved in detecting new functionality-specific vulnerabilities. In this flowchart, steps requiring human intervention are highlighted in yellow. Adjacent to each step, we provide a breakdown of the average time costs.

The process of retrieving functionality-equivalent APIs, including extracting and validating the PoCs of existing vulnerabilities, requires approximately four hours per PoC. Our engineers read the vulnerability issue (if exists) and its URL links, and test the PoC within the affected software repository (on its vulnerable version). As shown in Section 5.3, a single validated PoC leads to the detection of an average of 17.9 new vulnerabilities, so the manual costs of four hours are affordable. Additionally, existing vulnerability databases, e.g., Snyk[12] and VeraCode[13], have already collected a large number of PoCs although they do not open-source these PoCs to avoid potential risks.

The procedure of migrating PoCs and exploiting new vulnerabilities involves three steps requiring human intervention. First, our engineers identify the usage for each functionality-equivalent API that cannot be directly invoked by our parameter-matching algorithm [45]. This task applies to 20% of APIs, with each requiring 3-5 minutes to address. Second, for each API triggered by a potential vulnerability, our engineers verify whether the API inputs are legal. This step is necessary for 19.6% of APIs, also taking 3-5 minutes per API. Third, once a vulnerability is confirmed, reporting each one takes approximately 15 minutes. We exclude the manual costs of this step as the vulnerability is already detected and exploited. Therefore, the manual costs of migrating a PoC to a new vulnerability is within 10 minutes.

The Runtime Costs of APISS. The runtime costs of APISS are shown in Table 7. For API mapping, APISS demonstrates high efficiency, with each step of embedding and retrieving taking less than one second. In the fuzzing process, we limit the maximum duration to two hours, determined based on the similarities between migrated inputs and potential vulnerability-triggering inputs. Consequently, the end-to-end fuzzing costs a total of 296 hours and successfully detects 144 vulnerabilities. These results confirm that the runtime costs of APISS are practical for real-world applications.

[12] `https://snyk.io`
[13] `https://www.veracode.com`

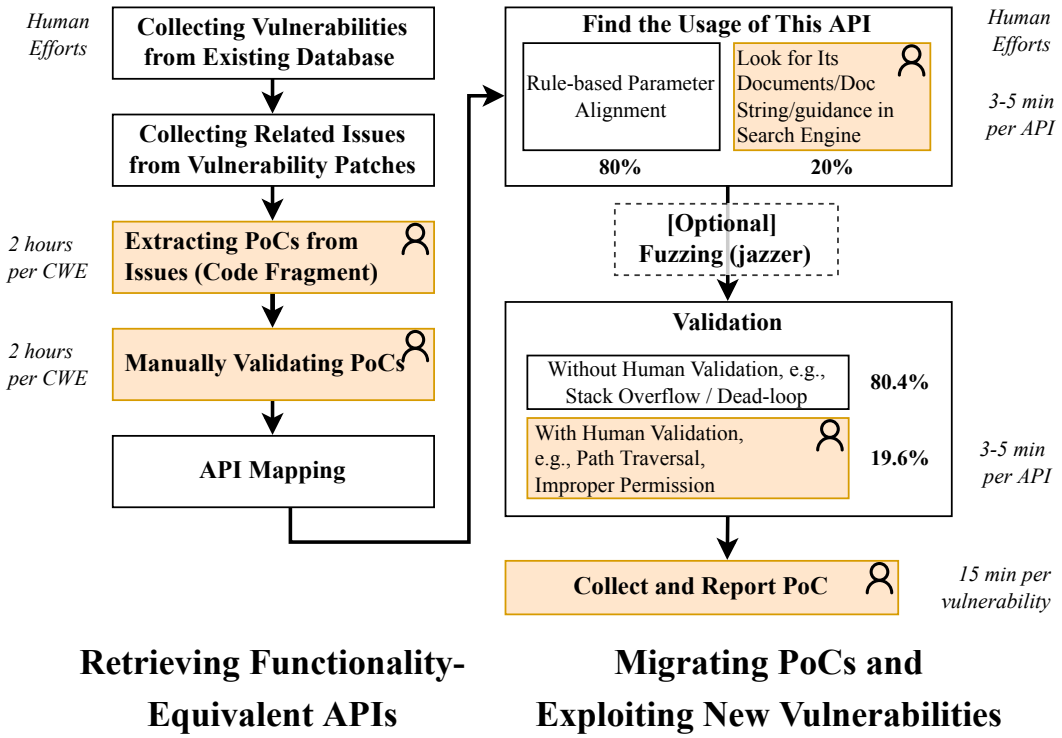


Figure 4 The Details of Human-Involved Procedure. White Boxes are Automatic Steps While Orange Boxes are Human-Involved Steps.

6 Discussion

Detecting Functionality-Specific Vulnerabilities of Other Programming Languages. We primarily focus on Java vulnerabilities due to the widespread usage of Java third-party libraries in software development [40]. APISS is designed with a high degree of generalization ability and is not restricted to Java vulnerabilities. For instance, APISS detects a new Java vulnerability (Listing 2) based on a Scala one (Listing 1). In Section 5.4, we also show that APISS can effectively retrieve functionality-equivalent APIs among various programming languages, e.g., Java and C#, without extra fine-tuning or training. Thus, APISS can be easily adapted to detect other programming languages' functionality-specific vulnerabilities.

Detecting Non-Functionality-Specific Vulnerabilities. Non-functionality-specific vulnerabilities also count for a large portion of vulnerabilities in third-party libraries. For example, among the Top 5 vulnerability types in 2023[14], CWE-787 (Out-of-bounds Write) and CWE-416 (Use-After-Write) are two representative non-functionality-specific ones because they do not occur in specific functionalities and remain widespread, especially in C/C++ repositories. Thus, the likelihood of being affected by these vulnerabilities is quite similar between functionality-equivalent APIs and the rest APIs. Now that APISS mainly focuses on retrieving candidate APIs based on their functionalities, we consider detecting such vulnerabilities as future work as they belong to different scopes of approaches.

[14] https://cwe.mitre.org/top25/archive/2023/2023_stubborn_weaknesses.html

Table 7 Hyper-Parameters and Runtime Costs of APISS.

API Mapping	CodeLLaMa	ChatGPT
Embedding Length	4,096 (default)	1,536 (default)
Embedding	465ms (per API)	554 ms (per API)
Initialize DB	0.4ms (per API)	0.1 ms (per API)
Retrieving	391ms (per API)	423 ms (per API)
Fuzzing Runtime	2 hours (per API)	296 hours (Total)

7 Threats to Validity

A major threat to external validity comes from the training dataset for evaluating existing deep-learning approaches. Existing deep-learning approaches [52, 8] are trained on two datasets including only C/C++ repositories while the vulnerabilities detected by APISS are mainly Java ones. To mitigate this threat, we choose a widely-used Java vulnerability dataset [51, 30, 10], which is general enough by including 1,359 Java vulnerabilities from 127 Java repositories. Thus, our reproduced models of existing deep-learning approaches reflect their effectiveness in detecting Java vulnerabilities. Additionally, the vulnerabilities in this dataset are before 2022 while those detected by APISS are later in 2023, so the potential risks of look-ahead bias [2] also do not exist.

The threats to internal validity are instrumentation effects that can bias our results. To reduce these threats, we manually inspect the intermediate results such as the functionality-equivalent APIs retrieved by APISS for dozens of sampled vulnerability patches, such as those in Listing 1 and Listing 2.

8 Related Work

8.1 Vulnerability Detection Approaches

Recurring-Vulnerability Detection Approaches. These approaches [43, 26, 36] are designed to detect recurring vulnerabilities, which come from reused code or shared code logic. For example, MVP [43] utilizes hash values based on the code property graphs of known vulnerable APIs to match similar APIs in open-source repositories.

However, these approaches rely heavily on matching the preceding implementation-based features. These approaches may produce high false negatives in detecting functionality-specific vulnerabilities as they rely heavily on matching the preceding implementation-based features while the affected APIs have diverse implementations. On the contrary, APISS considers functionality equivalence, a less strict constraint than implementation equivalence, to detect more vulnerabilities.

Deep-Learning Approaches. These approaches [41, 52, 39, 8, 38, 37, 42] determine whether a given method contains a potential vulnerability by learning code features of vulnerable APIs/methods of existing vulnerabilities. Devign [52] is a representative deep-learning approach. It uses Code Property Graphs (e.g., control/data-flow graphs) to represent the given method and designs a specialized Graph Neural Network (GNN) to determine whether this method is vulnerable.

These approaches can effectively detect vulnerabilities caused by incorrect implementation, such as use-after-free, because the source code of their affected APIs has specific patterns, e.g., a pointer operation after a `free` API invocation. However, deep-learning approaches are ineffective in detecting functionality-specific vulnerabilities because their affected APIs have diverse implementations without a common pattern, and they also face limitations in generating PoCs for detected vulnerabilities. On the contrary, APISS specifies the targeted vulnerabilities as functionality-specific ones, whose PoCs can be migrated from existing vulnerabilities.

8.2 API Mapping Approaches

API Mapping maps a library API to another functionality-equivalent one. Existing work [29, 11, 6, 50, 47] mainly focuses on translating an API of a specific programming language to another one. For example, SAR [6] translates first-party Java APIs to C# ones. Specifically, it learns the vector representation of both Java and C# APIs and trains a mapping model by adversarial learning. However, there are two main limitations when adopting existing API mapping approaches to retrieve functionality-equivalent APIs. First, for each pair of programming languages, these approaches need to train a specific model to translate an API between them, resulting in the scalability problem. Second, similar to existing vulnerability detection approaches, they also overemphasize the importance of API bodies. As shown in Section 5.4, API bodies hardly contribute to the effectiveness of API mapping. Thus, we design our API mapping module in APISS to address the preceding limitations.

8.3 API Fuzzing

API fuzzing [16, 4, 15, 44, 13, 19, 28, 5], as a special category of fuzzing approaches [53, 5, 25], is widely used to exploit API vulnerabilities. RULF [19] is a representative approach that fuzzes API vulnerabilities in Rust standard libraries. These approaches mainly use template-based or synthesized API sequences and randomly generated API inputs. Additionally, these approaches need to be executed for a long period to cover the input space as possible, thus resulting in low efficiency/effectiveness. On the contrary, APISS migrates the PoCs and inputs of existing vulnerabilities, so it can detect and trigger new functionality-specific vulnerabilities with higher probability within a short period.

9 Conclusion

In this paper, we have presented the first approach to utilize API doc strings and signatures instead of focusing on API bodies to retrieve functionality-equivalent APIs and then detect functionality-specific vulnerabilities. We have conducted a comprehensive evaluation to demonstrate APISS' effectiveness in retrieving functionality-equivalent APIs, achieving a Top-1 Accuracy of 0.81 while the best of the baselines under comparison achieves only 0.55. We have designed a semi-automatic schema to reduce the manual costs of migrating the PoC of existing vulnerabilities. We have demonstrated APISS' efficiency, costing less than 10 minutes of human labor per vulnerability, and the end-to-end runtime overhead of analyzing one candidate API is less than 2 hours. We have demonstrated APISS' high value to security practice by detecting 179 new vulnerabilities and receiving 60 new CVE IDs.

References

1 ChatGPT, 2023. URL: https://openai.com/chatgpt.

2 Maya Almaraz, Michelle Y Wong, and Wendy H Yang. Looking back to look ahead: a vision for soil denitrification research. *Ecology*, 101(1):e02917, 2020.

3 Liat Antwarg, Ronnie Mindlin Miller, Bracha Shapira, and Lior Rokach. Explaining anomalies detected by autoencoders using shapley additive explanations. *Expert Systems with Applications*, 186:115736, 2021. doi:10.1016/J.ESWA.2021.115736.

4 Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: stateful REST API fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, pages 748–758, 2019. doi:10.1109/ICSE.2019.00083.

5 Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017. doi:10.1145/3133956.3134020.

6 Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. SAR: learning cross-language API mappings with little knowledge. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 796–806, 2019.

7 Sicong Cao, Xiaobing Sun, Xiaoxue Wu, David Lo, Lili Bo, Bin Li, and Wei Liu. Coca: improving and explaining graph neural network-based vulnerability detection systems. *arXiv preprint arXiv:2401.14886*, 2024. doi:10.48550/arXiv.2401.14886.

8 Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296, 2022. doi:10.1109/TSE.2021.3087402.

9 Junjie Chen, Hongxu Hou, Jing Gao, Yatu Ji, and Tiangang Bai. RGCN: recurrent graph convolutional networks for target-dependent sentiment analysis. In *Proceedings of International Conference on Knowledge Science, Engineering and Management*, pages 667–675, 2019. doi:10.1007/978-3-030-29551-6_59.

10 Tianyu Chen, Lin Li, Taotao Qian, Zeyu Wang, Guangtai Liang, Ding Li, Qianxiang Wang, and Tao Xie. Identifying vulnerability patches by comprehending code commits with comprehensive change contexts. *arXiv preprint arXiv:2310.02530*, 2023. doi:10.48550/arXiv.2310.02530.

11 Lei Cui, Jiancong Cui, Yuede Ji, Zhiyu Hao, Lun Li, and Zhenquan Ding. API2Vec: learning representations of API sequences for malware detection. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 261–273, 2023. doi:10.1145/3597926.3598054.

12 HWJ Debeye and P Van Riel. Lp-norm deconvolution. *Geophysical Prospecting*, 38(4):381–403, 1990.

13 Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. Fuzzing deep-learning libraries via automated relational API inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 44–56, 2022. doi:10.1145/3540250.3549085.

14 Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. Retrieval-augmented generation for large language models: a survey. *arXiv preprint arXiv:2312.10997*, 2023.

15 Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. Intelligent REST API data fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 725–736, 2020. doi:10.1145/3368089.3409719.

16 Harrison Green and Thanassis Avgerinos. Graphfuzz: Library api fuzzing with lifetime-aware dataflow graphs. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1070–1081, 2022. doi:10.1145/3510003.3510228.

17 Hazim Hanif and Sergio Maffeis. Vulberta: simplified source code pre-training for vulnerability detection. In *Proceedings of 2022 International Joint Conference on Neural Networks*, pages 1–8, 2022. doi:10.1109/IJCNN55064.2022.9892280.

18 Java2CSharp. https://github.com/codejuicer/java2csharp, 2017.

19 Jianfeng Jiang, Hui Xu, and Yangfan Zhou. RULF: Rust library fuzzing via API dependency graph traversal. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, pages 581–592, 2021. doi:10.1109/ASE51524.2021.9678813.

20 Zhi Jing, Yongye Su, Yikun Han, Bo Yuan, Chunjiang Liu, Haiyun Xu, and Kehai Chen. When large language models meet vector databases: a survey. *arXiv preprint arXiv:2402.01763*, 2024.

21 Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016. arXiv:1609.02907.

22 Yi Li, Shaohua Wang, and Tien N Nguyen. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 292–303, 2021. doi:10.1145/3468264.3468597.

23 Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

24 Zhenqin Li and Harold A Scheraga. Monte carlo-minimization approach to the multiple-minima problem in protein folding. *Proceedings of the National Academy of Sciences*, 84(19):6611–6615, 1987.

25 Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018. doi:10.1109/TR.2018.2834476.

26 Miaoqian Lin, Kai Chen, and Yang Xiao. Detecting API post-handling bugs using code and description in patches. In *Proceedings of the 32nd USENIX Security Symposium*, pages 3709–3726, 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/lin.

27 Xiong Luo, Xiaohui Chang, and Xiaojuan Ban. Regression and classification using extreme learning machine based on l1-norm and l2-norm. *Neurocomputing*, 174:179–186, 2016. doi:10.1016/J.NEUCOM.2015.03.112.

28 Riyadh Mahmood, Jay Pennington, Danny Tsang, Tan Tran, and Andrea Bogle. A framework for automated API fuzzing at enterprise scale. In *Proceedings of 2022 IEEE Conference on Software Testing, Verification and Validation*, pages 377–388, 2022. doi:10.1109/ICST53961.2022.00018.

29 Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. Exploring API embedding for API usages and applications. In *Proceedings of the 39th International Conference on Software Engineering*, pages 438–449, 2017. doi:10.1109/ICSE.2017.47.

30 Truong Giang Nguyen, Thanh Le-Cong, Hong Jin Kang, Ratnadira Widyasari, Chengran Yang, Zhipeng Zhao, Bowen Xu, Jiayuan Zhou, Xin Xia, Ahmed E. Hassan, Xuan-Bach D. Le, and David Lo. Multi-granularity detector for vulnerability fixes. *IEEE Transactions on Software Engineering*, 49(8):4035–4057, 2023. doi:10.1109/TSE.2023.3281275.

31 Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. Regvd: Revisiting graph neural networks for vulnerability detection. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 178–182, 2022. doi:10.1145/3510454.3516865.

32 Yasunobu Nohara, Koutarou Matsumoto, Hidehisa Soejima, and Naoki Nakashima. Explanation of machine learning models using shapley additive explanation and application for real data in hospital. *Computer Methods and Programs in Biomedicine*, 214:106584, 2022. doi:10.1016/J.CMPB.2021.106584.

33 Gustavo G Rondina and Juarez LF Da Silva. Revised basin-hopping monte carlo algorithm for structure optimization of clusters and nanoparticles. *Journal of Chemical Information and Modeling*, 53(9):2282–2298, 2013. doi:10.1021/CI400224Z.

34 Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. CodeLlama: open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

35 Karen Scarfone and Peter Mell. An analysis of CVSS version 2 vulnerability scoring. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 516–525, 2009. doi:10.1109/ESEM.2009.5314220.

36 Youkun Shi, Yuan Zhang, Tianhao Bai, Lei Zhang, Xin Tan, and Min Yang. RecurScan: detecting recurring vulnerabilities in PHP web applications. In *Proceedings of the ACM Web Conference 2024*, pages 1746–1755, 2024. doi:10.1145/3589334.3645530.

37 Benjamin Steenhoek, Hongyang Gao, and Wei Le. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024. doi:10.1145/3597503.3623345.

38 Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security*, 16:1943–1958, 2020. doi:10.1109/TIFS.2020.3044773.

39 Wenbo Wang, Tien N Nguyen, Shaohua Wang, Yi Li, Jiyuan Zhang, and Aashish Yadavally. DeepVD: toward class-separation features for neural network vulnerability detection. In *Proceedings of the 45th International Conference on Software Engineering*, pages 2249–2261, 2023. doi:10.1109/ICSE48619.2023.00189.

40 Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An empirical study of usages, updates and risks of third-party libraries in Java projects. In *Proceedings of 2020 IEEE International Conference on Software Maintenance and Evolution*, pages 35–45, 2020. doi:10.1109/ICSME46990.2020.00014.

41 Xin-Cheng Wen, Yupan Chen, Cuiyun Gao, Hongyu Zhang, Jie M Zhang, and Qing Liao. Vulnerability detection with graph simplification and enhanced graph representation learning. *arXiv preprint arXiv:2302.04675*, 2023. doi:10.48550/arXiv.2302.04675.

42 Xin-Cheng Wen, Xinchen Wang, Cuiyun Gao, Shaohua Wang, Yang Liu, and Zhaoquan Gu. When less is enough: Positive and unlabeled learning model for vulnerability detection. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, pages 345–357, 2023. doi:10.1109/ASE56229.2023.00144.

43 Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. MVP: detecting vulnerabilities using patch-enhanced vulnerability signatures. In *Proceedings of the 29th USENIX Security Symposium*, pages 1165–1182, 2020.

44 Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W Godfrey. Docter: documentation-guided fuzzing for testing deep learning API functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 176–188, 2022. doi:10.1145/3533767.3534220.

45 Shengzhe Xu, Ziqi Dong, and Na Meng. Meditor: inference and application of API migration edits. In *Proceedings of 2019 IEEE/ACM 27th International Conference on Program Comprehension*, pages 335–346, 2019. doi:10.1109/ICPC.2019.00052.

46 Chen Yang, Junjie Chen, Bin Lin, Jianyi Zhou, and Ziqi Wang. Enhancing LLM-based test generation for hard-to-cover branches via program analysis. *arXiv preprint arXiv:2404.04966*, 2024. doi:10.48550/arXiv.2404.04966.

47 Zejun Zhang, Minxue Pan, Tian Zhang, Xinyu Zhou, and Xuandong Li. Deep-diving into documentation to develop improved Java-to-Swift API mapping. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 106–116, 2020. doi:10.1145/3387904.3389282.

48 Yafan Zhao, Xin Chen, and Jun Li. Tgmin: A global-minimum structure search program based on a constrained basin-hopping algorithm. *Nano Research*, 10:3407–3420, 2017.

49 Weining Zheng, Yuan Jiang, and Xiaohong Su. Vu1SPG: vulnerability detection based on slice property graph representation learning. In *Proceedings of 2021 IEEE 32nd International Symposium on Software Reliability Engineering*, pages 457–467, 2021. doi:10.1109/ISSRE52982.2021.00054.

50 Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 195–204, 2010. doi:10.1145/1806799.1806831.

51 Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E Hassan. Finding a needle in a haystack: automated mining of silent vulnerability fixes. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, pages 705–716, 2021. doi:10.1109/ASE51524.2021.9678720.

52 Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.

53 Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys*, 54(11s):1–36, 2022. doi:10.1145/3512345.

Monadic Type-And-Effect Soundness

Francesco Dagnino
DIBRIS, Università di Genova, Italy

Paola Giannini
DiSSTE, Università del Piemonte Orientale, Vercelli, Italy

Elena Zucca
DIBRIS, Università di Genova, Italy

Abstract

We introduce the abstract notions of *monadic operational semantics*, a small-step semantics where computational effects are modularly modeled by a monad, and *type-and-effect system*, including *effect types* whose interpretation lifts well-typedness to its monadic version. In this meta-theory, as usual in the non-monadic case, we can express progress and subject reduction, and provide a proof, given once and for all, that they imply soundness. The approach is illustrated on a lambda calculus with generic effects, equipped with an expressive type-and-effect system We provide proofs of progress and subject reduction, parametric on the interpretation of effect types. In this way, we obtain as instances many significant examples, such as checking exceptions, preventing/limiting non-determinism, constraining order/fairness of outputs. We also provide an extension with constructs to raise and handle computational effects, which can be instantiated to model different policies.

2012 ACM Subject Classification Theory of computation → Operational semantics; Theory of computation → Type structures

Keywords and phrases Effects, monads, type soundness

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.7

Related Version *Full Version*: `https://arxiv.org/abs/2504.10159` [9]

Funding This work was partially funded by the MUR project "T-LADIES" (PRIN 2020TL3X8X) and has the financial support of the Università del Piemonte Orientale.

1 Introduction

It would be hard to overstate the impact on foundations of programming languages of, on one hand, the idea that computational effects can be modeled by monads [30, 31], and, on the other hand, the technique based on progress and subject reduction to prove the soundness of a type system with respect to a small-step operational semantics [47].

Moggi's seminal work [30, 31] recognized monads as the suitable structure to modularly describe the denotational semantics of effectful languages. The key idea was the distinction between *pure* (effect-free) and *monadic* (effectful) expressions, also called *computations*, the latter getting semantics in a monad. Haskell has firstly[1] shown that such an approach can be fruitfully adopted in a mainstream language, through a monad type constructor allowing to encapsulate effectful code. However, the structure of a monad[2] does not include operations for *raising* effects, which need to be defined ad-hoc in instances. *Algebraic and generic effects* [33, 34, 35], instead, explicitly consider operations to raise effects, interpreted by additional structure on the monad. Such an approach, combined with handlers [37, 38, 4, 40], has been exploited in fully-fledged programming languages, e.g., in Scala and OCaml 5.

[1] Many other languages have then supported a monad pattern, e.g., Scheme, Python, Racket, Scala, F#.
[2] In Haskell, methods of the `Monad` typeclass.

© Francesco Dagnino, Paola Giannini, and Elena Zucca;
licensed under Creative Commons License CC-BY 4.0

To provide guarantees on, besides the result, the computational effects possibly raised by a computation, type systems are generalized to *type-and-effect systems*. A great many of these have been designed for specific calculi, modelling effects by relying on auxiliary structures, e.g., memory in imperative calculi, and providing ad-hoc soundness proofs; Katsumata [25] has provided a unified view of such systems, however based on denotational semantics. In this paper, instead, we provide an *operational meta-theory of monadic type-and-effect soundness*, analogous to the one mentioned above for usual type soundness based on small-step semantics, progress and subject reduction [47]. To this end, we provide abstract notions of small-step monadic semantics, type-and-effect system, and soundness, as detailed below.[3]

Operational semantics. We design a language semantics which is *monadic*, since effects are, as customary, expressed by a monad, and simultaneously *small-step*, since we define sequences of reduction steps. To this end, we start from a reduction from language expressions to monadic expressions (in a given monad) required to be deterministic, and extend such a relation to a *total* function, so to be able to combine steps by Kleisli composition, similarly to the approach in [17]. In this way, reduction sequences are always infinite, so termination is conventionally represented by monadic elements called *results*, which always reduce to themselves without raising any effect. On top of the reduction, we define the *finitary* semantics of an expression, which is either the monadic result reached in many steps, if any, or divergence. This semantics does not describe the computational effects raised by infinite computations. Hence, we define an *infinitary* semantics, obtained, as customary, as the supremum of a chain of approximants, provided that the monad has the necessary structure.

Type-and-effect system. As done in [7, 6] for standard type systems, we abstractly model a *type-and-effect system* as a family of predicates over expressions, indexed by types and *effect types*, statically approximating the computational effects that may be raised during evaluation. Effect types are required to form an ordered monoid, as typically assumed in effect systems [32, 29] and proposed as algebraic structure by [25]. The relation between an effect type and the allowed computational effects is specified by a family of *predicate liftings* [21]. In this way the transformation from a predicate to a monadic one associated to a given effect type is independent from the predicate and its universe.

Soundness. We provide abstract definitions of monadic progress and monadic subject reduction, and a proof, given once and for all, that they imply soundness. The latter means that, if a monadic element is the result of a well-typed expression, then it should be well-typed, that is, satisfy the lifting through the effect type of well-typedness of values.

We illustrate the approach on Λ_Σ, a lambda calculus with generic effects, equipped with an expressive type-and-effect system. We provide proofs of progress and subject reduction parametrically on the interpretation of effect types. In this way, we obtain as instances many significant examples, such as checking exceptions, preventing/limiting non-determinism, constraining order/fairness of outputs on different locations. We also provide an extension with constructs to handle effects, which can be instantiated as well to model different policies.

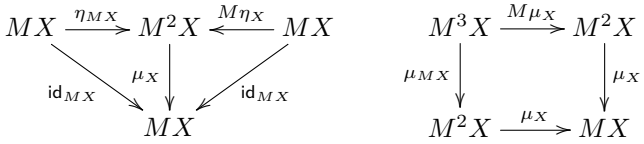
[3] The term "effect" is used in literature both as synonym of computational effect, and in the context of type-and-effect systems, as a static approximation of the former. We will use "effect" when there is no ambiguity, otherwise "computational effect" and "effect type", respectively.

Outline. Section 2 reports the background on monads. Section 3 introduces monadic operational semantics, exemplified through Λ_Σ in Section 4, where we also design a type-and-effect system, discussing its soundness. The approach is formalized by the abstract framework in Section 5; the proof technique introduced there is applied in Section 6 to Λ_Σ. Finally, in Section 7 we enhance the example by handlers, and in Section 8 we discuss related and future work, and summarize the contributions. Omitted proofs can be found in the extended version [9].

2 Preliminaries on monads

Monads [13, 43] are a fundamental notion in category theory, enabling an abstract and unified study of algebraic structures. Since Moggi's seminal papers [30, 31], they have also become a major tool in computer science, especially for describing the semantics of computational effects, and integrating them in programming languages in a structured and principled way. In this section, we recall basic notions about monads, and provide some examples. We will focus on monads on the category of sets and functions, denoted by $\mathcal{Set}$, referring the reader to standard textbooks [41] for a detailed introduction in full generality.

A *monad* $\mathbb{M} = \langle M, \eta, \mu \rangle$ (on $\mathcal{Set}$) consists of a functor $M \colon \mathcal{Set} \to \mathcal{Set}$ and two natural transformations $\eta : \mathsf{Id} \Rightarrow M$ and $\mu : M^2 \Rightarrow M$ such that, for every set X, the following diagrams commute:



The functor M specifies, for every set X, a set MX of monadic elements built over X, in a way that is compatible with functions. The map η_X, named *unit*, embeds elements of X into monadic elements in MX, and the map μ_X, named *multiplication*, flattens monadic elements built on top of other monadic elements into plain monadic elements.

From these data, one can derive an operation on functions of type $X \to MY$, dubbed *Kleisli extension*, which is crucial for modelling computational effects using monads. For all sets X, Y, we have a function $(-)^\dagger \colon (X \to MY) \to (MX \to MY)$, defined by $f^\dagger = \mu_Y \circ Mf$, that is, first we lift f through M to apply it to monadic elements and then we flatten the result using μ_Y. It is easy to see that the operation $(-)^\dagger$ satisfies the following equations for all $f \colon X \to MY$ and $g \colon Y \to MZ$:

$$\eta_X^\dagger = \mathsf{id}_{MX} \qquad f^\dagger \circ \eta_X = f \qquad g^\dagger \circ f^\dagger = (g^\dagger \circ f)^\dagger$$

Actually, a monad can be equivalently specified in the form of a *Kleisli triple* $\langle M, \eta, (-)^\dagger \rangle$ [28], where M is a mapping on sets, η is a family of functions $\eta_X \colon X \to MX$, for every set X, and $(-)^\dagger$ is a family of functions $(-)^\dagger \colon (X \to MY) \to (MX \to MY)$, for all sets X, Y, satisfying the three equations above. In particular we have $\mu_X = \mathsf{id}^\dagger_{MX}$.

Functions of type $X \to MY$ are called *Kleisli functions* and play a special role: they can be regarded as "effectful functions" from X to Y, raising effects described by the monad $\mathbb{M}$. Indeed, from the Kleisli extension, we can define a composition on Kleisli functions, known as Kleisli composition: given $f \colon X \to MY$ and $g \colon Y \to MZ$ we set

$$g * f = g^\dagger \circ f = \mu_Z \circ Mg \circ f$$

Intuitively, $g * f$ applies f followed by g, sequentially composing the effects they may raise. It is immediate to see that Kleisli composition is associative and η_X is the identity Kleisli function on the set X, that is, η_X is the function raising no effects.

We introduce some useful notation, corresponding to standard operations of monadic types in languages, where such types are assigned to expressions with effects. Given $\alpha \in MX$, $f\colon X \to MY$ and $g\colon X \to Y$, we set

$$\begin{array}{ll} - \gg\!= -\colon MX \to (X \to MY) \to MY & \alpha \gg\!= f = f^\dagger(\alpha) \\ \mathsf{map}\colon (X \to Y) \to MX \to MY & \mathsf{map}\, g\, \alpha = Mg(\alpha) \end{array}$$

The operator $\gg\!=$ is also called bind. As its definition shows, it can be seen as an alternative description of the Kleisli extension, where the parameters are taken in inverse order. This view corresponds, intuitively, to the sequential composition of two expressions with effects, where the latter depends on a parameter *bound* to the result of the former. The operator map describes the effect of the functor M on functions. That is, the lifting of function g through M is applied to a monadic value α. Note that bind and map are interdefinable:

$$\alpha \gg\!= f = \mu_Y(\mathsf{map}\, f\, \alpha) \qquad\qquad \mathsf{map}\, g\, \alpha = \alpha \gg\!= (\eta_Y \circ g)$$

Furthermore, we can express Kleisli composition using bind: $(g * f)(x) = f(x) \gg\!= g$.

In the following examples we characterize the monads by defining bind rather than multiplication μ since this is often more insightful, and customary in programming languages.

▶ **Example 1** (Exceptions)**.** Let us fix a set Exc. The monad $\mathbb{E}_{\mathsf{Exc}} = \langle E_{\mathsf{Exc}}, \eta^{\mathbb{E}_{\mathsf{Exc}}}, \mu^{\mathbb{E}_{\mathsf{Exc}}} \rangle$ is given by $E_{\mathsf{Exc}}X = \mathsf{Exc} + X$, and

$$\eta^{\mathbb{E}_{\mathsf{Exc}}}(x) = \iota_2(x) \qquad\qquad \alpha \gg\!= f = \begin{cases} f(x) \text{ if } \alpha = \iota_1(x) \\ \alpha \text{ otherwise } (\alpha = \iota_2(\mathsf{e}) \text{ for some } \mathsf{e} \in \mathsf{Exc}) \end{cases}$$

where $+$ denotes disjoint union (coproduct) and ι_1, ι_2 the left and right injections, respectively. We will omit the reference to the set Exc when it is clear from the context.

▶ **Example 2** (Classical Non-Determinism)**.** The monad $\mathbb{P} = \langle P, \eta^{\mathbb{P}}, \mu^{\mathbb{P}} \rangle$ is given by $PX = \wp(X)$, that is, PX is the set of all subsets of X, and

$$\eta^{\mathbb{P}}(x) = \{x\} \qquad\qquad \alpha \gg\!= f = \bigcup_{x \in \alpha} f(x)$$

A variant of this monad is the list monad $\mathbb{L} = \langle L, \eta^{\mathbb{L}}, \mu^{\mathbb{L}} \rangle$, where the set LX of (possibly infinite) lists over X is coinductively defined by the following rules: $\epsilon \in L(X)$ and, if $x \in X$ and $l \in L(X)$, then $x\colon l \in L(X)$. We use the notation $[x_1, \ldots, x_n]$ to denote the finite list $x_1\colon \ldots\colon x_n\colon \epsilon$. Then, the unit is given by $\eta^{\mathbb{L}}_X(x) = [x]$ and the monadic bind is corecursively defined by the following clauses: $\epsilon \gg\!= f = \epsilon$ and $(x\colon l) \gg\!= f = f(x)(l \gg\!= f)$, where juxtaposition denotes the concatenation of possibly infinite lists.

▶ **Example 3** (Probabilistic Non-Determinism)**.** Denote by DX the set of probability subdistributions α over X with countable support, i.e., $\alpha\colon X \to [0..1]$ with $\sum_{x \in X} \alpha(x) \leq 1$ and $\mathsf{supp}(\alpha) = \{x \in X \mid \alpha(x) \neq 0\}$ countable set. We write $r \cdot \alpha$ for the pointwise multiplication of a subdistribution α with a number $r \in [0, 1]$. The monad $\mathbb{D} = \langle D, \eta^{\mathbb{D}}, \mu^{\mathbb{D}} \rangle$ is given by

$$\eta^{\mathbb{D}}(x) = y \mapsto \begin{cases} 1 & y = x \\ 0 & \text{otherwise} \end{cases} \qquad\qquad \alpha \gg\!= f = \sum_{x \in X} \alpha(x) \cdot f(x)$$

▶ **Example 4** (Output/Writer). Let $\langle \mathsf{Out}, \cdot, \varepsilon \rangle$ be a monoid, e.g., the monoid of strings over a fixed alphabet. The monad $\mathbb{O} = \langle O, \eta^{\mathbb{O}}, \mu^{\mathbb{O}} \rangle$ is given by $OX = \mathsf{Out} \times X$ and

$$\eta^{\mathbb{O}}(x) = \langle \varepsilon, x \rangle \qquad \langle o, x \rangle \gg\!= f = \langle o \cdot \pi_1(f(x)), \pi_2(f(x)) \rangle$$

Combining this monad with the exception monad of Example 1, we obtain the pointed output monad, whose underlying functor is given by $O'X = \mathsf{Out} \times (X + \{\bot\})$.

▶ **Example 5** (Global State). Let S be a set of states. The monad $\mathbb{S} = \langle S, \eta^{\mathbb{S}}, \mu^{\mathbb{S}} \rangle$ is given by $SX = \mathsf{S} \to \mathsf{S} \times X$ and

$$\eta^{\mathbb{S}}(x) = s \mapsto \langle s, x \rangle \qquad \alpha \gg\!= f = s \mapsto f(\pi_2(\alpha(s)))(\pi_1(\alpha(s)))$$

We can combine this monad with the exception monad of Example 1 obtaining $\mathbb{S}_{\mathsf{Exc}} = \langle S_{\mathsf{Exc}}, \eta^{\mathbb{S}_{\mathsf{Exc}}}, \mu^{\mathbb{S}_{\mathsf{Exc}}} \rangle$ where $S_{\mathsf{Exc}} = \mathsf{S} \to (\mathsf{S} \times X) + \mathsf{Exc}$ and $\eta^{\mathbb{S}_{\mathsf{Exc}}}_X(x) = s \mapsto \eta^{\mathbb{E}_{\mathsf{Exc}}}_{\mathsf{S} \times X}(\langle s, x \rangle)$ and $\alpha \gg\!= f = s \mapsto (\alpha(s)) \gg\!=_{\mathbb{E}_{\mathsf{Exc}}} (x \mapsto f(x)(s))$. This combination yields a monad thanks to the fact that $\mathbb{S}$ determines a monad transformer [27, 22].

3 Monadic operational semantics

In this section we abstractly describe a framework for (deterministic) monadic operational semantics, adapting from [16, 17].

▶ **Definition 6.** *Let $\mathcal{L}$ be a triple $\langle \mathsf{Exp}, \mathsf{Val}, \mathsf{ret} \rangle$, called a* language*, with* Exp *the set of* expressions*,* Val *the set of* values*, and* $\mathsf{ret} \colon \mathsf{Val} \to \mathsf{Exp}$ *an injective function. A* monadic operational semantics *for $\mathcal{L}$ consists of:*

- *a monad $\mathbb{M} = \langle M, \eta, \mu \rangle$*
- *a relation $\to \subseteq \mathsf{Exp} \times M\mathsf{Exp}$, called* monadic (one-step) reduction*, such that*
  - *$\to$ is a partial function and*
  - *for all $v \in \mathsf{Val}$, $\mathsf{ret}(v) \not\to$.*

The set Exp contains expressions that can be executed, while Val contains values produced by the computation. The inclusion ret identifies the expressions representing successful termination with a given value. The elements of $M\mathsf{Exp}$, called *monadic expressions*, are the counterpart of expressions in the monad $\mathbb{M}$. The relation $\to$ models single computation steps, which transform expressions into monadic ones, thus possibly raising computational effects. Finally, the first requirement on $\to$ ensures that it is deterministic, while the latter one that expressions representing values cannot be reduced.

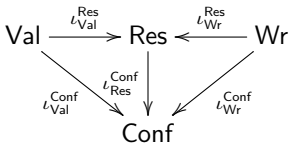
Assume a monadic operational semantics $\langle \mathbb{M}, \to \rangle$ for a language $\langle \mathsf{Exp}, \mathsf{Val}, \mathsf{ret} \rangle$. In standard (small-step) operational semantics, starting from the one-step reduction we can model computations as (either finite of infinite) sequences of reduction steps. In particular, finite computations are obtained by the reflexive and transitive closure $\to^\star$ of the one-step reduction. Starting from the *monadic* one-step reduction, which is a relation from a set to a different one, there is no transitive closure in the usual sense.

In the solution proposed in [16], the monadic reduction can be an arbitrary relation; however, this requires a relational extension of the monad [3]. On the other hand, given a relation $\to \subseteq \mathsf{Exp} \times M\mathsf{Exp}$ which is a *total* function, we can define, by iterating Kleisli composition, a relation $\to^\star \subseteq \mathsf{Exp} \times M\mathsf{Exp}$ which plays the role of transitive closure, as in [17].

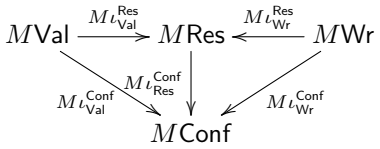
Our aim here is to define $\to^\star$ taking the second approach, which does not require relational extensions. Unfortunately, the monadic reduction, exactly as the standard one, is by its own nature a partial function, where some expressions, representing terminated computations,

cannot be reduced. Notably, those representing successful termination with a value, and others, intuitively corresponding to stuck computations. To obtain a total function, we extend the monadic reduction to *configurations* (expressions, values, or a special result wrong). In particular, expressions representing terminating computations reduce to (the monadic embedding of) a value, and wrong, respectively. In this way, we can define the transitive closure by Kleisli composition, as formally detailed below.

Set $\mathsf{Res} = \mathsf{Val} + \mathsf{Wr}$, where $\mathsf{Wr} = \{\mathsf{wrong}\}$, that is, a result r is either a value, modelling successful termination, or wrong, modelling a stuck computation. Then, we consider the set $\mathsf{Conf} = \mathsf{Exp} + \mathsf{Res}$ of *configurations*, ranged over by c. We have the following commutative diagram of coproduct injections:



As customary, with a slight abuse of notation, we identify elements with their images along such injections. We use E, V, R and C to range over monadic expressions, monadic values, monadic results and monadic configurations, respectively, that is, elements of $M\mathsf{Exp}$, $M\mathsf{Val}$, $M\mathsf{Res}$ and $M\mathsf{Conf}$. Since monads on $\mathcal{S}et$ preserve injections [1], by applying M to the diagram above we get another diagram of injections:



In the following, we use some shortcuts for the application of such injections: notably, we write $\hat{\mathrm{E}}$ for $M\iota^{\mathsf{Conf}}_{\mathsf{Exp}}(\mathrm{E})$, $\hat{\mathrm{R}}$ for $M\iota^{\mathsf{Conf}}_{\mathsf{Res}}(\mathrm{R})$ and $\hat{\mathrm{V}}$ for $M\iota^{\mathsf{Res}}_{\mathsf{Val}}(\mathrm{V})$.

We can now extend the monadic reduction $\to$ to configurations, getting the relation $\xrightarrow[\mathsf{step}]{} \subseteq \mathsf{Conf} \times M\mathsf{Conf}$ shown in Figure 1. As said above, reduction is extended to expressions

$$(\textsc{exp})\ \frac{e \to \mathrm{E}}{e \xrightarrow[\mathsf{step}]{} \hat{\mathrm{E}}} \qquad (\textsc{ret})\ \frac{}{\mathsf{ret}(v) \xrightarrow[\mathsf{step}]{} \eta_{\mathsf{Conf}}(v)}$$

$$(\textsc{wrong})\ \frac{e \not\to \quad e \neq \mathsf{ret}(v) \text{ for all } v \in \mathsf{Val}}{e \xrightarrow[\mathsf{step}]{} \eta_{\mathsf{Conf}}(\mathsf{wrong})} \qquad (\textsc{res})\ \frac{}{r \xrightarrow[\mathsf{step}]{} \eta_{\mathsf{Conf}}(r)}$$

Figure 1 Monadic (one-step) reduction on configurations.

which represent terminated computations, which reduce to the monadic embedding of the corresponding value or wrong, respectively; moreover, it is extended to results (either values or wrong) as well, which conventionally reduce to their monadic embedding.

It is immediate to see that $\xrightarrow[\mathsf{step}]{}$ is (the graph of) a total function from Conf to $M\mathsf{Conf}$, which we simply write step. Clearly step is a Kleisli function for $\mathbb{M}$, hence we can define the "monadic reflexive and transitive closure" $\xrightarrow[\mathsf{step}]{}^{\star} \subseteq \mathsf{Conf} \times M\mathsf{Conf}$ of $\xrightarrow[\mathsf{step}]{}$ as follows:

$$(\textsc{refl})\ \frac{}{c \xrightarrow[\mathsf{step}]{}^{\star} \eta_{\mathsf{Conf}}(c)} \qquad (\textsc{step})\ \frac{c \xrightarrow[\mathsf{step}]{}^{\star} \mathrm{C}}{c \xrightarrow[\mathsf{step}]{}^{\star} \mathrm{C} \gg\!= \mathsf{step}}$$

These rules are analogous to those defining the reflexive and transitive closure of a standard one-step relation. In (REFL) a configuration reduces, rather than to itself, to its monadic counterpart. In rule (STEP), $\xrightarrow{\mathsf{step}}{}^{\star}$ is combined with $\xrightarrow{\mathsf{step}}$, rather than by standard composition, through the $\gg=$ operator. That is, a computation is extended by one step through a monadic binding of the previously computed monadic configuration C to the step function.

Equivalently, we can define the Kleisli n-th iteration step^n of the step function by setting $\mathsf{step}^0 = \eta_{\mathsf{Conf}}$ and $\mathsf{step}^{n+1} = \mathsf{step} * \mathsf{step}^n$. Then, the following holds:

▶ **Proposition 7.** *$c \xrightarrow{\mathsf{step}}{}^{\star} \mathrm{C}$ if and only if $\mathsf{step}^n(c) = \mathrm{C}$ for some $n \in \mathbb{N}$.*

In a similar way, we can define a small-step reduction on monadic configurations. Recall that the Kleisli extension of step gives the function $\mathsf{step}^{\dagger} \colon M\mathsf{Conf} \to M\mathsf{Conf}$.

▶ **Definition 8.** *The* small-step reduction induced by $\xrightarrow{\mathsf{step}}$ *is the relation $\Rightarrow$ on $M\mathsf{Conf}$ defined by: $\mathrm{C} \Rightarrow \mathrm{C}'$ iff $\mathsf{step}^{\dagger}(\mathrm{C}) = \mathrm{C}'$.*

Then, since $\Rightarrow$ is a relation on $M\mathsf{Conf}$, we can consider its (standard) reflexive and transitive closure $\Rightarrow^{\star} \subseteq M\mathsf{Conf} \times M\mathsf{Conf}$, which describes computations on monadic configurations.

▶ **Proposition 9.** *$\mathrm{C} \Rightarrow^{\star} \mathrm{C}'$ if and only if $\mathrm{C} \gg= \mathsf{step}^n = \mathrm{C}'$ for some $n \in \mathbb{N}$.*

Combining Propositions 7 and 9, we get the following corollary which relates $\xrightarrow{\mathsf{step}}{}^{\star}$ and $\Rightarrow^{\star}$.

▶ **Corollary 10.** *The following are equivalent:*

1. $c \xrightarrow{\mathsf{step}}{}^{\star} \mathrm{C}$
2. $\eta_{\mathsf{Conf}}(c) \Rightarrow^{\star} \mathrm{C}$
3. $c \xrightarrow{\mathsf{step}}{}^{\star} \mathrm{C}' \Rightarrow^{\star} \mathrm{C}$, *for some* $\mathrm{C}' \in M\mathsf{Exp}$

To use the above machinery for describing the semantics of expressions, we essentially follow the approach in [17], with minor adjustments to fit our context.

First of all note that, being defined on top of a total function, $\Rightarrow^{\star}$ has no normal forms. However, monadic results should intuitively correspond to termination. Formally, this is a consequence of the proposition below, stating that a monadic configuration which is a result only reduces to itself; hence, when a monadic result is reached, its reduction continues with an infinite sequence of trivial reduction steps, which can be seen as a representation of termination. Hence, the outcome of a (terminating) computation is a monadic result.

▶ **Proposition 11.** *$\hat{\mathrm{R}} \Rightarrow \mathrm{C}$ if and only if $\mathrm{C} = \hat{\mathrm{R}}$.*

Thanks to the above proposition, we can prove the following, stating that the monadic result of a computation, if any, is unique.

▶ **Proposition 12.** *If $c \xrightarrow{\mathsf{step}}{}^{\star} \hat{\mathrm{R}}_1$ and $c \xrightarrow{\mathsf{step}}{}^{\star} \hat{\mathrm{R}}_2$, then $\mathrm{R}_1 = \mathrm{R}_2$.*

Hence, we can define a function $[\![-]\!]_{\star} \colon \mathsf{Exp} \to M\mathsf{Res} + \{\infty\}$ describing the semantics of expressions as follows:

$$[\![e]\!]_{\star} = \begin{cases} \mathrm{R} & \text{if } e \xrightarrow{\mathsf{step}}{}^{\star} \hat{\mathrm{R}} \\ \infty & \text{otherwise} \end{cases}$$

This is called *finitary semantics*, as it describes only monadic results that can be reached in finitely many steps. In other words, all diverging computations are identified and no information on computational effects they may produce is available. Even worse, when the monad supports some form of non-determinism, we may have computations that terminate in some cases and diverge in others, but the finitary semantics considers them as diverging, as they never reach a result after finitely many steps.

To overcome this limitation, again following [17], we introduce an *infinitary semantics*, which is able to provide more information on diverging computations. To achieve this, we need to assume more structure on the monad $\mathbb{M}$. Recall that, given a partially ordered set $\langle P, \sqsubseteq \rangle$, an ω-chain is an increasing sequence $(x_n)_{n\in\mathbb{N}}$ of points in P. We say that $\langle P, \sqsubseteq \rangle$ is an ω-CPO if it has a least element $\bot$ and every ω-chain in $\langle P, \sqsubseteq \rangle$ has a supremum $\bigsqcup_{n\in\mathbb{N}} x_n$. A function $f\colon \langle P, \sqsubseteq \rangle \to \langle P', \sqsubseteq' \rangle$ between ω-CPO is said to be ω-continuous if it preserves the least element, and suprema of ω-chains. Note that an ω-continuous function is necessarily monotone. Then, we have the following definition:

▶ **Definition 13.** *An ω-CPO-ordered monad $\mathbb{M} = \langle M, \sqsubseteq, \eta, \mu \rangle$ is a monad $\langle M, \eta, \mu \rangle$ together with a partial order $\sqsubseteq_X$ on MX, for every set X, such that*

1. *for every set X, the poset $\langle MX, \sqsubseteq_X \rangle$ is an ω-CPO and*
2. *for all sets X, Y, the Kleisli extension $(-)^\dagger\colon (X \to MY) \to (MX \to MY)$ is ω-continuous with respect to the pointwise extension of $\sqsubseteq_Y$ to function spaces $X \to MY$ and $MX \to MY$.*

▶ **Example 14.** The powerset and list monads of Example 2 are ω-CPO-ordered with the subset and prefix ordering, respectively. The subdistribution monad of Example 3 is ω-CPO-ordered with the pointwise ordering on subdistributions. The other monads of Section 2 can also be turned into ω-CPO-ordered monads, but require adjustements, typically a combination with the exception monad. For instance, in Example 4, the output monad is not ω-CPO-ordered in general, but its pointed version is ω-CPO-ordered when the underlying monoid is an ω-CPO and the multiplication is ω-continuous in the second argument.

From now on, we assume the monad $\mathbb{M}$ to have an ω-CPO-ordered structure. Our goal is to define a function $[\![-]\!]_\infty\colon \mathsf{Exp} \to M\mathsf{Res}$ modelling the infinitary semantics of expressions. To this end, we first define a function $\mathsf{res}\colon M\mathsf{Conf} \to M\mathsf{Res}$ extracting monadic results from monadic configurations. Let $\mathsf{res}_0\colon \mathsf{Conf} \to M\mathsf{Res}$ be the function given by

$$\mathsf{res}_0(c) = \begin{cases} \bot_{\mathsf{Res}} & c = e \\ \eta_{\mathsf{Res}}(r) & c = r \end{cases}$$

and set $\mathsf{res} = \mathsf{res}_0^\dagger$. The key point is that the total relation $\Rightarrow$ on monadic configurations is compatible with the order $\sqsubseteq_{\mathsf{Res}}$ under the application of res, as the following proposition shows.

▶ **Proposition 15.** *If $c \Rightarrow c'$ then $\mathsf{res}(c) \sqsubseteq_{\mathsf{Res}} \mathsf{res}(c')$.*

For every $e \in \mathsf{Exp}$ and $n \in \mathbb{N}$, we define $[\![e]\!]_n = \mathsf{res}(\mathsf{step}^n(e))$. From Propositions 7 and 9, we easily derive $\mathsf{step}^n(e) \Rightarrow \mathsf{step}^{n+1}(e)$, and, by Proposition 15, $[\![e]\!]_n \sqsubseteq_{\mathsf{Res}} [\![e]\!]_{n+1}$. Hence, the sequence $([\![e]\!]_n)_{n\in\mathbb{N}}$ is an ω-chain in $\langle M\mathsf{Res}, \sqsubseteq_{\mathsf{Res}} \rangle$ and so we define the infinitary semantics as

$$[\![e]\!]_\infty = \bigsqcup_{n\in\mathbb{N}} [\![e]\!]_n$$

Intuitively, $[\![e]\!]_n$ is the portion of the result that is reached after n reduction steps. Hence, the actual result is obtained as the supremum of all such approximations and it may be never reached, thus describing also the observable behaviour of possibly diverging computations.

We conclude this section by stating that infinitary and finitary semantics agree on terminating computations.

▶ **Proposition 16.** *If $[\![e]\!]_\star = r$, then $[\![e]\!]_\infty = r$.*

4 Example: a lambda calculus with generic effects

The aim of this section is twofold:

- to ilustrate the monadic operational semantics in Section 3 through a simple example
- to equip such example with a type-and-effect system, and to discuss how to express and prove type soundness with respect to finitary/infinitary semantics

To this end, we introduce Λ_Σ, a call-by-value λ-calculus with *generic effects*. Here Σ is a family of sets $\{\Sigma_k\}_{k\in\mathbb{N}}$ of k-ary operations raising effects. We choose generic rather than algebraic effects, thus avoiding explicit continuations, to have a style more convenient for a programmer [40], and a more significant monadic reduction.[4]

The syntax is shown in Figure 2. We use $\overline{v}$ as metavariable for sequences $v_1, \ldots, v_n$, and analogously for other sequences. We assume variables x, y, f, ..., using the last for variables

$$\begin{array}{lcll} v & ::= & x \mid \mathtt{rec}\, f.\lambda x.e \mid \ldots & \text{value} \\ e & ::= & v\, v' \mid op(\overline{v}) \mid \mathtt{return}\ v \mid \mathtt{do}\ x = e_1\,;\ e_2 \mid \ldots & \text{expression} \end{array}$$

Figure 2 Λ_Σ: fine-grain syntax.

denoting functions. We adopt, as customary, the fine-grain approach [26], where *values* are effect-free, whereas *expressions*, also called *computations*, may raise effects. Dots stand for additional, unspecified, constructs, such as operators of primitive types, conditional, etc.

To illustrate type soundness with respect to the infinitary semantics as well, the calculus includes recursive functions; notably, $\mathtt{rec}\, f.\lambda x.e$ is a function with parameter x and body e which can recursively call itself through the variable f. Standard lambda expressions can be recovered as those where f does not occur free in e, that is, when the function is non-recursive, and we will use the abbreviation $\lambda x.e$ for such expressions.

In this section, Exp and Val denote the sets of closed expressions and values of Λ_Σ, respectively. In the following, we define a monadic (one-step) reduction for the language, parametric on a monad $\mathbb{M} = \langle M, \eta, \mu\rangle$, being a relation $\rightarrow$ on $\mathsf{Exp} \times M\mathsf{Exp}$. As in Section 3, we use V and E to range over $M\mathsf{Val}$ and $M\mathsf{Exp}$, respectively.

This relation is modularly defined on top of a "pure" reduction $\rightarrow_p$ on $\mathsf{Exp} \times \mathsf{Exp}$. In our example, such relation only reduces function calls into the corresponding bodies, as shown in Figure 3; other rules should be added to deal with additional language constructs, as we will do for handlers in Section 7. `Do` expressions are, then, normal forms for the pure reduction, and will be handled by the rules of the monadic reduction.

$$(\textsc{app})\ \frac{}{v\, v' \rightarrow_p e[v/f][v'/x]}\ v = \mathtt{rec}\, f.\lambda x.e$$

Figure 3 Pure reduction.

Rules defining the monadic reduction are given in Figure 4. As mentioned, they are parametric on the underlying monad; more in detail, they depend on the following ingredients:

- The function $\eta_{\mathsf{Exp}}\colon \mathsf{Exp} \rightarrow M\mathsf{Exp}$ embedding language expressions into their counterpart in the monad, written simply η in this section.
- The function $\mathsf{map}\colon (\mathsf{Exp} \rightarrow \mathsf{Exp}) \rightarrow M\mathsf{Exp} \rightarrow M\mathsf{Exp}$ lifting functions from expressions to expressions to their counterpart in the monad.

[4] In the case of algebraic effects there would be no monadic reduction inside a context, as in rule (DO).

Moreover we assume, for each operation op with arity k, a partial function $\mathsf{run}_{op}{:}\mathsf{Val}^k \rightharpoonup M\mathsf{Val}$, returning a monadic value expressing the effects raised by a call of the operation. The function could be undefined, for instance when arguments do not have the expected types.

$$(\textsc{pure})\ \frac{e \to_p e'}{e \to \eta(e')} \qquad (\textsc{effect})\ \frac{}{op(\overline{v}) \to \mathsf{map}\,(\mathtt{return}\ [\,])\,\mathsf{run}_{op}(\overline{v})}$$

$$(\textsc{ret})\ \frac{}{\mathtt{do}\ x = \mathtt{return}\ v;\ e \to \eta(e[v/x])} \qquad (\textsc{do})\ \frac{e_1 \to \mathrm{E}}{\mathtt{do}\ x = e_1;\ e_2 \to \mathsf{map}\,(\mathtt{do}\ x = [\,];\ e_2)\,\mathrm{E}}$$

Figure 4 Monadic (one-step) reduction.

Rule (PURE) propagates a pure step, embedding its result in the monad. In rule (EFFECT), the effect is actually raised. To this end, we apply the function of type $M\mathsf{Val} \to M\mathsf{Exp}$ obtained by lifting, through map, the context `return` [ ] to the monadic value obtained from the call. Here we identify the context `return` [ ], which is an expression with a hole, with the function $v \mapsto \mathtt{return}\ [v]$ of type $\mathsf{Val} \to \mathsf{Exp}$. In rule (RET), when the first subterm of a `do` expression returns a value, the expression is reduced to the monadic embedding of the second subterm, after replacing the variable with the value. Rule (DO), instead, propagates the reduction of the first subterm. To take into account raised effects, we apply the function of type $M\mathsf{Exp} \to M\mathsf{Exp}$ obtained by lifting, through map, the context $\mathtt{do}\ x = [\,];\ e_2$ to the monadic expression obtained from e_1. Analogously to above, we identify the context $\mathtt{do}\ x = [\,];\ e_2$ with the function $e \mapsto \mathtt{do}\ x = [e];\ e_2$ of type $\mathsf{Exp} \to \mathsf{Exp}$.

The following property is needed to have an instance of the framework in Section 3.

▶ **Proposition 17** (Determinism). *If $e \to E_1$ and $e \to E_2$ then $E_1 = E_2$.*

We show now some examples of expressions and their monadic operational semantics. We assume the calculus to be extended with standard constructs, such `unit`, 0, `succ`, `true` and `false` constructors, `pred` selector, `iszero` test, and conditional. We write $e; e'$ for $\mathtt{do}\ x = e;\ e'$ when x does not occur free in e', and sometimes, to save space, n for $\mathtt{succ}^n\,0$.

▶ **Example 18.** Set, as underlying monad, the monad of exceptions introduced in Example 1, where $EX = X + \mathsf{Exc}$. For each $\mathsf{e} \in \mathsf{Exc}$, we assume an operation $\mathtt{raise}\langle \mathsf{e} \rangle$, with

$$\mathsf{run}_{\mathtt{raise}\langle \mathsf{e} \rangle} : \mathbf{1} \to M\mathsf{Val} \qquad \mathsf{run}_{\mathtt{raise}\langle \mathsf{e} \rangle} = \iota_2(\mathsf{e})$$

The function $\mathsf{predfun} = \lambda x.\mathtt{if}\ \mathtt{iszero}\ x\ \mathtt{then}\ \mathtt{raise}\langle \mathsf{PredZero} \rangle\ \mathtt{else}\ \mathtt{return}\ \mathtt{pred}\,x$ raises the exception PredZero when the argument is 0. The following are examples of small-step reduction sequences on monadic configurations[5]:

$$\begin{array}{lcl}
\mathsf{predfun}\,\mathtt{succ}\,0 & \Rightarrow & \mathtt{if}\ \mathtt{iszero}\ \mathtt{succ}\,0\ \mathtt{then}\ \mathtt{raise}\langle \mathsf{PredZero} \rangle\ \mathtt{else}\ \mathtt{return}\ 0 \\
& \Rightarrow & \mathtt{if}\ \mathtt{false}\ \mathtt{then}\ \mathtt{raise}\langle \mathsf{PredZero} \rangle\ \mathtt{else}\ \mathtt{return}\ 0 \\
& \Rightarrow & \mathtt{return}\ 0 \\
& \Rightarrow & 0 \\
\mathsf{predfun}\,0 & \Rightarrow & \mathtt{if}\ \mathtt{iszero}\ 0\ \mathtt{then}\ \mathtt{raise}\langle \mathsf{PredZero} \rangle\ \mathtt{else}\ \mathtt{return}\ 0 \\
& \Rightarrow & \mathtt{if}\ \mathtt{true}\ \mathtt{then}\ \mathtt{raise}\langle \mathsf{PredZero} \rangle\ \mathtt{else}\ \mathtt{return}\ 0 \\
& \Rightarrow & \mathtt{raise}\langle \mathsf{PredZero} \rangle \\
& \Rightarrow & \mathsf{PredZero}
\end{array}$$

[5] Where we omit the injections from monadic expressions and values.

In the first reduction sequence, all steps are derived by rules (PURE) in Figure 4 and (EXP) in Figure 1, except for the last one, which is derived by rule (RET) in Figure 1. Analogously in the second reduction sequence, where the last step is derived by rule (EFFECT) in Figure 4 and (EXP) in Figure 1. Note that, here and in the following examples, after reaching a monadic result the sequence of steps continues with an infinite sequence of steps, in the case above $0 \Rightarrow 0$ and $\mathsf{PredZero} \Rightarrow \mathsf{PredZero}$ steps.

▶ **Example 19.** Set, as underlying monad, the monad of non-determinism of Example 2, in the variant of the possibly infinite lists. We assume a constant operation `choose`, with

$$\mathsf{run}_{\mathtt{choose}} \colon \mathbf{1} \to M\mathsf{Val} \qquad \mathsf{run}_{\mathtt{choose}} = [\mathtt{true}, \mathtt{false}]$$

Then, $e = \mathtt{do}\ y = \mathtt{choose};\ \mathtt{if}\ y\ \mathtt{then}\ \mathtt{return}\ 0\ \mathtt{else}\ \mathtt{return}\ \mathtt{succ}\,0$ reduces as follows.[6]

$$\begin{array}{rcl}
[\,e\,] & \Rightarrow & [\,\mathtt{do}\ y = \mathtt{true};\ \mathtt{if}\ y\ \mathtt{then}\ \mathtt{return}\ 0\ \mathtt{else}\ \mathtt{return}\ \mathtt{succ}\,0, \\
& & \ \mathtt{do}\ y = \mathtt{false};\ \mathtt{if}\ y\ \mathtt{then}\ \mathtt{return}\ 0\ \mathtt{else}\ \mathtt{return}\ \mathtt{succ}\,0\,] \\
& \Rightarrow & [\,\mathtt{if}\ \mathtt{t}\ \mathtt{then}\ \mathtt{ret}\ 0\ \mathtt{else}\ \mathtt{ret}\ 1, \mathtt{if}\ \mathtt{f}\ \mathtt{then}\ \mathtt{ret}\ 0\ \mathtt{else}\ \mathtt{ret}\ 1\,] \\
& \Rightarrow & [\,\mathtt{return}\ 0, \mathtt{return}\ \mathtt{succ}\,0\,] \\
& \Rightarrow & [\,0, \mathtt{succ}\,0\,]
\end{array}$$

Given $\mathsf{chfun}^{\uparrow} = \mathtt{rec}\, f.\lambda x.\mathtt{do}\ y = \mathtt{choose};\ \mathtt{if}\ y\ \mathtt{then}\ \mathtt{return}\ x\ \mathtt{else}\ f\,\mathtt{succ}\,x$, the expression $\mathsf{chfun}^{\uparrow}\,0$ reduces as follows:

$$\begin{array}{rcl}
[\,\mathsf{chfun}^{\uparrow}\,0\,] & \Rightarrow & [\,\mathtt{do}\ y = \mathtt{choose};\ \mathtt{if}\ y\ \mathtt{then}\ \mathtt{return}\ 0\ \mathtt{else}\ \mathsf{chfun}^{\uparrow}\,\mathtt{succ}\,0\,] \\
& \Rightarrow & [\,\mathtt{do}\ y = \mathtt{true};\ \mathtt{if}\ y\ \mathtt{then}\ \mathtt{return}\ 0\ \mathtt{else}\ \mathtt{return}\ \mathsf{chfun}^{\uparrow}\,\mathtt{succ}\,0, \\
& & \ \mathtt{do}\ y = \mathtt{false};\ \mathtt{if}\ y\ \mathtt{then}\ \mathtt{return}\ 0\ \mathtt{else}\ \mathsf{chfun}^{\uparrow}\,\mathtt{succ}\,0\,] \\
& \Rightarrow & [\,\mathtt{if}\ \mathtt{t}\ \mathtt{then}\ \mathtt{ret}\ 0\ \mathtt{else}\ \mathtt{ret}\ 1, \mathtt{if}\ \mathtt{f}\ \mathtt{then}\ \mathtt{ret}\ 0\ \mathtt{else}\ \mathsf{chfun}^{\uparrow}\ 1\,] \\
& \Rightarrow & [\,\mathtt{return}\ 0, \mathsf{chfun}^{\uparrow}\,\mathtt{succ}\,0\,] \\
& \Rightarrow & [\,0, \mathsf{chfun}^{\uparrow}\,\mathtt{succ}\,0\,] \\
& \Rightarrow & [\,0, \mathtt{do}\ y = \mathtt{choose};\ \mathtt{if}\ y\ \mathtt{then}\ \mathtt{ret}\ \mathtt{succ}\,0\ \mathtt{else}\ \mathsf{chfun}^{\uparrow}\,\mathtt{s}\,\mathtt{s}\,0\,] \\
& \Rightarrow^{\star} & [\,0, \mathtt{succ}\,0, \mathsf{chfun}^{\uparrow}\,\mathtt{succ}\,\mathtt{succ}\,0\,] \\
& \cdots &
\end{array}$$

Note that the second reduction is non-terminating, in the sense that a monadic result (a list of values) is never reached. Hence, with the finitary semantics, we get $[\![\mathsf{chfun}^{\uparrow}\,0]\!]_{\star} = \infty$. With the infinitary semantics, instead, we get the following ω-chain:

$$[\ ], \ldots, [\,0\,], \ldots, [\,0, \mathtt{succ}\,0\,], \ldots, [\,0, \mathtt{succ}\,0, \ldots, \mathtt{succ}^n\,0\,], \ldots,$$

whose supremum is, as expected, the infinite list of the (values representing the) natural numbers. On the other end, given the function

$$\mathsf{chfun}^{\downarrow} = \mathtt{rec}\, f.\lambda x.\mathtt{if}\ \mathtt{iszero}\ x\ \mathtt{then}\ \mathtt{ret}\ x\ \mathtt{else}\ \mathtt{do}\ y = \mathtt{choose};\ \mathtt{if}\ y\ \mathtt{then}\ \mathtt{ret}\ x\ \mathtt{else}\ f\,\mathtt{pred}\,x$$

we get $[\![\mathsf{chfun}^{\downarrow}\,\mathtt{succ}^n\,0]\!]_{\star} = [\![\mathsf{chfun}^{\downarrow}\,\mathtt{succ}^n\,0]\!]_{\infty} = [\,\mathtt{succ}^n\,0, \ldots, 0\,]$.

▶ **Example 20.** Set, as underlying monad, the monad of probabilistic non-determinism of Example 3. We consider a discrete uniform distribution over a set of two elements and use the same function `choose`, now returning the list consisting of the values `true` and `false` with probability $\frac{1}{2}$, that we denote by $[\,\frac{1}{2} : \mathtt{true}, \frac{1}{2} : \mathtt{false}\,]$.

[6] We use `t`, `f`, `ret`, and `s`, for `true`, `false`, `return`, and `succ`, to save space.

Then, $[\,1 : e\,]$ and $[\,1 : \mathsf{chfun}^{\uparrow}\,0\,]$ reduce analogously to the previous example:

$$[\,1 : e\,] \Rightarrow^{\star} [\,\frac{1}{2} : 0, \frac{1}{2} : \mathtt{succ}\,0\,]$$

$$[\,[\,1 : \mathsf{chfun}^{\uparrow}\,0\,]\,] \Rightarrow^{\star} [\,\frac{1}{2} : 0, \frac{1}{4} : \mathtt{succ}\,0, \frac{1}{8} : \mathtt{succ}^2\,0, \frac{1}{16} : \mathtt{succ}^3\,0\,] \Rightarrow \ldots$$

Again, the second reduction is non-terminating, hence, with the finitary semantics, we get ∞, whereas, with the infinitary semantics, we get an ω-chain whose supremum is the infinite list where each (value representing the) number n has probability $\frac{1}{2^{n+1}}$.

▶ **Example 21.** Set, as underlying monad, the output monad of Example 4, in its pointed version. As a simple concrete choice, we take as elements of Out sequences of pairs $\langle \ell, \mathtt{succ}^n\,0\rangle$ where ℓ ranges over a fixed set Loc of *output locations* modeling, e.g., file names or output channels. We assume, for each ℓ, an operation $\mathtt{write}\langle\ell\rangle\colon \mathtt{Nat} \to \mathtt{Unit}$, with

$$\mathsf{run}_{\mathtt{write}\langle\ell\rangle}\colon \mathsf{Val} \to M\mathsf{Val} \qquad \mathsf{run}_{\mathtt{write}\langle\ell\rangle}(v) = \begin{cases} \langle\langle \ell, v\rangle, \mathtt{unit}\rangle & \text{if } v = \mathtt{succ}^n\,0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Given two distinct output locations ℓ, ℓ', and the functions

$$\mathsf{wfun}^{\uparrow} = \mathtt{rec}\,f.\lambda x.\mathtt{write}\langle\ell\rangle(x)\mathtt{;write}\langle\ell'\rangle(x)\mathtt{;}f\,\mathtt{succ}\,x$$
$$\mathsf{wfun}^{\downarrow} = \mathtt{rec}\,f.\lambda x.\mathtt{write}\langle\ell\rangle(x)\mathtt{;write}\langle\ell'\rangle(x)\mathtt{;if\ iszero}\ x\ \mathtt{then\ unit\ else}\ f\,\mathtt{pred}\,x$$

we get, as in the previous examples, the following semantics:

$$[\![\mathsf{wfun}^{\uparrow}\,0]\!]_{\star} = \infty$$
$$[\![\mathsf{wfun}^{\uparrow}\,0]\!]_{\infty} = \langle\langle \ell, 0\rangle \cdot \langle \ell', 0\rangle \cdot \langle \ell, \mathtt{succ}\,0\rangle \cdot \langle \ell', \mathtt{succ}\,0\rangle \cdot \ldots, \bot\rangle$$
$$[\![\mathsf{wfun}^{\downarrow}\,\mathtt{s}^n\,0]\!]_{\star} = [\![\mathsf{wfun}^{\downarrow}\,\mathtt{s}^n\,0]\!]_{\infty} = \langle\langle \ell, \mathtt{s}^n\,0\rangle \cdot \langle \ell', \mathtt{s}^n\,0\rangle \cdot \ldots \cdot \langle \ell, 0\rangle \cdot \langle \ell', 0\rangle, \mathtt{unit}\rangle$$

In the first two cases the reduction does not terminate, so no value is returned. With the finitary semantics also no effect is produced, whereas with the infinitary semantics the effect is the infinite sequence of outputs.

In order to equip the calculus (Figure 2) with a type-and-effect system, we need the ingredients shown in Figure 5. Besides types, which are functional types and additional unspecified

$$\begin{array}{lcll} T & ::= & \ldots \mid T \to_E T' & \text{type} \\ \Gamma & ::= & \overline{x : T} & \text{context} \end{array}$$

Figure 5 Types and contexts.

types, we consider *effect types* (*effects* when there is no ambiguity), ranged over by E, meant to be static approximations of the computational effects raised by an expression. As formally detailed below, effects are sets of (possibly infinite) sequences of operations. In this way, they are expressive enough to approximate computational effects in many different monads, as we will describe in Section 6, and we abstract away from details of a syntactic representation, which of course would be needed in a real language. Functional types are annotated with an effect, approximating the computational effects of calling the function. Finally, we assume operations to be typed; formally, for each op, we write $op\colon T_1 \ldots T_n \to T$.

Set Σ^{∞} the set of either finite or infinite sequences of operations. We use α, β to range over elements of Σ^{∞}, denote by ϵ the empty sequence, by $op{:}\alpha$ the sequence consisting of op followed by α, and by $\cdot$ sequence concatenation, coinductively defined by:

$$\epsilon\cdot\beta = \beta \qquad (op{:}\alpha)\cdot\beta = op{:}(\alpha\cdot\beta)$$

As customary, we write op for the sequence $op{:}\epsilon$.

An *effect* is a non-empty subset of Σ^∞. We denote by $\cdot$ composition of effects, defined by:

$$E{\cdot}E' = \{\alpha{\cdot}\beta \mid \alpha \in E, \beta \in E'\}$$

Absence of effects is modeled by the set $\{\epsilon\}$; the empty effect, if allowed, could be assigned to non-terminating computations which never call operations; however, since $E{\cdot}\emptyset = \emptyset$, effects assigned to a previous terminating computation would be lost.

The type-and-effect system is shown in Figure 6. The subtyping judgment has shape

$$\text{(SUB-FUN)}\ \frac{T_1' \le T_1 \quad T_2 \le T_2'}{T_1 \to_E T_2 \le T_1' \to_{E'} T_2'}\ E \subseteq E' \qquad \text{(SUB-REFL)}\ \frac{}{T \le T}$$

$$\text{(SUB-TRANS)}\ \frac{T \le T' \quad T' \le T''}{T \le T''} \qquad \text{(SUB-TE)}\ \frac{T \le T' \quad E \subseteq E'}{T!E \le T'!E'}$$

$$\text{(T-VAR)}\ \frac{}{\Gamma \vdash x : T}\ \Gamma(x) = T \qquad \text{(T-ABS)}\ \frac{\Gamma, f : T \to_E T', x : T \vdash e : T''!E'}{\Gamma \vdash \mathtt{rec}\, f.\lambda x.e : T \to_E T'}\ T''!E' \le T'!E$$

$$\text{(T-APP)}\ \frac{\Gamma \vdash v_1 : T_1 \to_E T \quad \Gamma \vdash v_2 : T_2}{\Gamma \vdash v_1\, v_2 : T!E}\ T_2 \le T_1 \qquad \text{(T-OP)}\ \frac{\Gamma \vdash v_i : T_i' \ \ \forall i \in 1..n}{\Gamma \vdash op(\overline{v}) : T!\{op\}}\ \begin{array}{l}\overline{v} = v_1, \ldots, v_n \\ op{:}\, T_1 \ldots T_n \to T \\ T_i' \le T_i\ \forall i \in 1..n\end{array}$$

$$\text{(T-RET)}\ \frac{\Gamma \vdash v : T}{\Gamma \vdash \mathtt{return}\ v : T!\{\epsilon\}} \qquad \text{(T-DO)}\ \frac{\Gamma \vdash e_1 : T_1!E_1 \quad \Gamma, x : T_2 \vdash e_2 : T!E_2}{\Gamma \vdash \mathtt{do}\ x = e_1\,;\ e_2 : T!E_1{\cdot}E_2}\ T_1 \le T_2$$

Figure 6 Type-and-effect system.

$T \le T'$. In (SUB-FUN) inclusion of effect types is propagated to functional types. Moreover subtyping is, as expected, covariant/contravariant on the result/parameter of functions. The other rules are standard.

The typing judgment for values has shape $\Gamma \vdash v : T$, since they have no effects. The judgment for expressions, instead, has shape $\Gamma \vdash e : T!E$.

Rule (T-VAR) is standard. In rule (T-ABS), a (possibly recursive) function gets a functional type, consisting of parameter/result types and effect, if the body, in a context where parameter and function are added with their types, gets a subtype and a subeffect of the result type and effect of the function. In rule (T-APP), an application gets the result type and the effect of the applied function, provided that the argument type is subtype of the expected one. In rule (T-OP), calling an operation raises the corresponding singleton effect, provided that the argument types are subtypes of the expected ones. In rule (T-RET), an expression representing a value has the trivial effect, and, in (T-DO), a sequential composition of two computations has the composition of the two effects.

▶ **Example 22.** We show some typing judgments which can be derived for the previous examples. We assume primitive types `Nat` and `Bool`, an empty type `Bot` subtype of any type, the singleton type `Unit` for the constant `unit`, and the obvious typing rule for conditional which takes the union of the effects of the two branches. Finally, we denote by α^n and α^ω a finite and infinite concatenation of αs, respectively.

1. In Example 18, with, for each $\mathsf{e} \in \mathsf{Exc}$, $\mathtt{raise}\langle \mathsf{e} \rangle{:}\, \mathbf{1} \to \mathtt{Bot}$,

$$\emptyset \vdash \mathsf{predfun} : \mathtt{Nat} \to_{\{\epsilon, \mathtt{raise}\langle \mathsf{PredZero} \rangle\}} \mathtt{Nat}$$

$$\emptyset \vdash \mathsf{predfun}\, v : \mathtt{Nat}!\{\epsilon, \mathtt{raise}\langle \mathsf{PredZero} \rangle\}\mathtt{if}\emptyset \vdash \mathtt{v} : \mathtt{Nat}$$

Note a significant feature of our type effects: differently from, e.g., Java checked exceptions, we can distinguish code which *may* raise an exception, as expressed by the effect $\{\epsilon, \texttt{raise}\langle\mathsf{PredZero}\rangle\}$, from code which *necessarily* raises an exception, as expressed by the effect $\{\texttt{raise}\langle\mathsf{PredZero}\rangle\}$, which is assigned, e.g., to the function $\lambda x.\texttt{raise}\langle\mathsf{PredZero}\rangle$. More in general, our type effects can *force* computational effects to be raised.

2. In Example 19, with $\texttt{choose}\colon \to \texttt{Bool}$,

$$\emptyset \vdash \mathsf{chfun}^{\uparrow} : \texttt{Nat} \to_{\{\texttt{choose}^n \mid n \geq 1\}} \texttt{Nat}$$
$$\emptyset \vdash \mathsf{chfun}^{\downarrow} : \texttt{Nat} \to_{\{\texttt{choose}^n \mid n \geq 0\}} \texttt{Nat}$$

Again, the effect of the first function forces non-determinism, differently from that of the second one. Apart from that, the two effects are very similar, even though calls of the first and second function always diverge and terminate, respectively. Indeed, as usual, effect types only provide a static approximation of the computational effects.

3. In Example 21, with, for each output location ℓ, $\texttt{write}\langle\ell\rangle\colon \texttt{Nat} \to \texttt{Unit}$,

$$\emptyset \vdash \mathsf{wfun}^{\uparrow} : \texttt{Nat} \to_{\{(\texttt{write}\langle\ell\rangle \cdot \texttt{write}\langle\ell'\rangle)^{\omega}\}} \texttt{Unit}$$
$$\emptyset \vdash \mathsf{wfun}^{\downarrow} : \texttt{Nat} \to_{\{(\texttt{write}\langle\ell\rangle \cdot \texttt{write}\langle\ell'\rangle)^{n} \mid n \geq 0\}} \texttt{Unit}$$

Here the difference between the effects of the two functions is even more significant: in the former, the sequence of two write calls is necessarily done infinitely many times, in the latter it can be done any arbitrary, yet finite, positive number of times. Moreover, in this case effects also provide an information on the the order among different write calls; for instance, here a $\texttt{write}\langle\ell\rangle$ call should be always followed by a $\texttt{write}\langle\ell'\rangle$ call.

We discuss now how to express and prove type soundness. Recall that the monadic operational semantics defined in Section 3 constructs, on top of the one-step reduction:

- a finitary semantics $[\![-]\!]_{\star}\colon \mathsf{Exp} \to M\mathsf{Res} + \{\infty\}$
- an infinitary semantics $[\![-]\!]_{\infty}\colon \mathsf{Exp} \to M\mathsf{Res}$

where $\mathsf{Res} = \mathsf{Val} + \mathsf{Wr}$, with the latter modelling a stuck computation. Hence, we expect a sound type-and-effect system to guarantee, first of all, that

(1) the (monadic) result of a well-typed expression is never wrong

analogously to what we expect for a standard type system. In the standard case, we also expect the result, if any, to be in agreement with the expression type. Here, since the expression has an effect as well, approximating the computational effects raised by its execution, we expect that

(2) the monadic result, if any, is in agreement with the expression type and effect

In finitary semantics, (2) imposes nothing on diverging expressions, since they have no monadic result, whereas, in infinitary semantics, (2) is significant for diverging expressions as well.

To formally express (2), we need to derive, from the well-typedness predicates (one for each type and effect), analogous predicates on monadic results. In the following section, this is achieved through a *predicate lifting* [21] λ, that is, a way to lift, for every set X, predicates over X to predicates over MX. Intuitively, λ adds requirements on the computational effects,expressed by an effect type, that is, lifting provides an *interpretation of effect types*.

5 Monadic type-and-effect soundness

The standard technique for proving type soundness with respect to a small-step operational semantics is as a consequence, by a simple inductive argument, of progress and subject reduction properties [47]. In this section, we introduce an analogous technique for monadic

operational semantics. Notably, we express progress and subject reduction for the monadic one-step reduction, and prove that they imply soundness. We develop our technique for *type-and-effect systems* [45, 32, 46, 29, 25], that is, formal systems providing an (over)approximation not only of the result of a computation, but also of its computational effects.

Following [7, 6], a type system can be abstractly seen as a family of predicates over expressions and values indexed by types. In a type-and-effect system, predicates over expressions will be indexed not only by types but also by *effect types*, describing the computational effects that expressions can produce during their evaluation, as defined below.

▶ **Definition 23.** *A* type-and-effect system $\Theta = \langle \mathsf{Ty}, \mathcal{E}, \mathsf{WT}^{\mathsf{E}}, \mathsf{WT}^{\mathsf{V}} \rangle$ *for a language* $\mathcal{L} = \langle \mathsf{Exp}, \mathsf{Val}, \mathsf{ret} \rangle$ *consists of the following data:*

- *a set* Ty *of* types
- *an ordered monoid* $\mathcal{E} = \langle \mathsf{Eff}, \preceq, \cdot, 1 \rangle$ *of* effect types
- *for every* $\tau \in \mathsf{Ty}$ *and* $\varepsilon \in \mathsf{Eff}$*, predicates* $\mathsf{WT}^{\mathsf{V}}_{\tau} \subseteq \mathsf{Val}$ *and* $\mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon} \subseteq \mathsf{Exp}$ *such that*
  - $\varepsilon \preceq \varepsilon'$ *implies* $\mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon} \subseteq \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon'}$ *and*
  - $\mathsf{ret}(v) \in \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon}$ *iff* $v \in \mathsf{WT}^{\mathsf{V}}_{\tau}$ *and* $1 \preceq \varepsilon$

The ordered monoid is a typical structure for effect systems [32, 29, 25]: 1 represents the absence of computational effects, $\varepsilon_1 \cdot \varepsilon_2$ represents the composition of computational effects described by ε_1 and ε_2, and $\varepsilon_1 \preceq \varepsilon_2$ states that the effect type ε_1 is more specific than ε_2.

The two families WT^{V} and WT^{E} are, for each index, predicates over values and expressions, respectively: $\mathsf{WT}^{\mathsf{V}}_{\tau}$ is the set of values of type τ, and $\mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon}$ is the set of expressions of type τ which may raise effects described by ε. The first requirement, that is, monotonicity with respect to the order, states that the latter actually models if $\varepsilon_1 \preceq \varepsilon_2$, then ε_1 is really more specific than ε_2. The second requirement states that an expression which is the embedding of a value has the same type, and an effect type which is not forcing any effect.[7]

Consider now an operational semantics $\langle \mathbb{M}, \rightarrow \rangle$, with $\mathbb{M} = \langle M, \mu, \eta \rangle$, and focus, e.g., on reduction from expressions to monadic expressions. To express type preservation, we should define, for each τ and ε, the monadic counterpart of $\mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon}$, being a predicate on $M\mathsf{Exp}$. The key idea is to obtain such predicate by applying a *predicate lifting* [21], that is, a way to lift, for every set X, predicates over X to predicates over MX, adding requirements on the computational effects modeled by the monad. In our case, for each effect type ε, the predicate lifting modularly models the meaning of ε, that is, the computational effects approximated by ε, independently from the set X and the predicate A, as formally detailed below.

For a set X, we denote by $\mathcal{P}(X)$ the poset of all subsets (a.k.a. predicates) on X, ordered by subset inclusion. For a function $f\colon X \to Y$, we have a monotone function $\mathcal{P}_f\colon \mathcal{P}(Y) \to \mathcal{P}(X)$, given by the inverse image: for $A \subseteq Y$, $\mathcal{P}_f(A) = \{x \in X \mid f(x) \in A\}$. That is, $\mathcal{P}_f$ is a predicate transformer, giving, for each predicate A on Y, the weakest condition elements of X should satisfy to be mapped by f in elements satisfying A. These data determine a functor $\mathcal{P}\colon \mathcal{S}et^{\mathrm{op}} \to \mathcal{P}os$, where $\mathcal{P}os$ denotes the category of posets and monotone functions.

▶ **Definition 24** (Interpretation of effect types)**.** *Let* $\mathbb{M} = \langle M, \mu, \eta \rangle$ *be a monad, and* $\mathcal{E} = \langle \mathsf{Eff}, \preceq, \cdot, 1 \rangle$ *an ordered monoid of effect types. Then, an* interpretation *of* $\mathcal{E}$ *in* $\mathbb{M}$ *consists of a family* λ *of monotone functions* $\lambda^{\varepsilon}_X\colon \mathcal{P}(X) \to \mathcal{P}(MX)$*, for every* $\varepsilon \in \mathsf{Eff}$ *and set* X*, such that*

1. $\lambda^{\varepsilon}_X(\mathcal{P}_f(A)) = \mathcal{P}_{Mf}(\lambda^{\varepsilon}_Y(A))$*, for every* $A \subseteq Y$ *and function* $f\colon X \to Y$
2. $\varepsilon \preceq \varepsilon'$ *implies* $\lambda^{\varepsilon}_X(A) \subseteq \lambda^{\varepsilon'}_X(A)$*, for every* $A \subseteq X$*,*
3. $A \subseteq \mathcal{P}_{\eta X}(\lambda^{1}_X(A))$*, for every* $A \subseteq X$*,*
4. $\lambda^{\varepsilon}_{MX}(\lambda^{\varepsilon'}_X(A)) \subseteq \mathcal{P}_{\mu X}(\lambda^{\varepsilon \cdot \varepsilon'}_X(A))$*, for every* $A \subseteq X$*.*

[7] For instance, in Example 22(1), we have $\{\epsilon\} \preceq \{\epsilon, \mathtt{raise}\langle \mathsf{PredZero} \rangle\}$, whereas $\{\epsilon\} \not\preceq \{\mathtt{raise}\langle \mathsf{PredZero} \rangle\}$.

The family $\lambda = (\lambda^{\varepsilon})_{\varepsilon \in \mathsf{Eff}}$ is a family of predicate liftings for the monad $\mathbb{M}$, indexed by effect types. For a subset $A \subseteq X$, the subset $\lambda^{\varepsilon}_X(A) \subseteq MX$ contains monadic elements which agree with A and whose computational effects are described by ε.

Item 1 states that λ^{ε}_X is natural in X, that is, for every $\varepsilon \in \mathsf{Eff}$, we have a natural transformation $\lambda^{\varepsilon} : \mathcal{P} \Rightarrow \mathcal{P} \circ M^{\mathrm{op}}$.[8] The naturality on X ensures that the semantics of each effect type is independent from the specific set X, thus depending only on the functor M.

Item 2 states that λ^{ε}_X is monotone with respect to the order on effects, that is, computational effects described by ε are also described by ε'.

Item 3 states that monadic elements in the image of η_X contain computational effects described by 1, that is, no computational effect.

Finally, in Item 4 we consider elements of M^2X whose computational effects are described by lifting predicates to MX through ε', and then by lifting through ε. By flattening such elements through $\mu_X \colon M^2X \to MX$ we obtain elements whose computational effects are described by the composition $\varepsilon \cdot \varepsilon'$.

▶ Remark 25. The monad $\mathbb{M}$ with an interpretation λ determine a structure on the functor $\mathcal{P}$, which can be described as a graded/parametric monad [14] on $\mathcal{P}$ in an appropriate 2-category (see e.g., [10]). Equivalently, λ determines a graded/parametric monad above $\mathbb{M}$ [25, Def. 2.6] along the fibration obtained from $\mathcal{P}$ by the Grothendieck construction [18].

▶ **Example 26.** Consider the exception monad $\mathbb{E}_{\mathsf{Exc}}$ of Example 1 and the ordered monoid $\langle \wp(\mathsf{Exc} + \{\mathsf{none}\}), \subseteq, \cdot, \{\mathsf{none}\}\rangle$ where $\mathsf{E}_1 \cdot \mathsf{E}_2 = (\mathsf{E}_1 \setminus \{\mathsf{none}\}) \cup \mathsf{E}_2$, if $\mathsf{none} \in \mathsf{E}_1$, and $\mathsf{E}_1 \cdot \mathsf{E}_2 = \mathsf{E}_1$, otherwise. For every $\mathsf{E} \in \wp(\mathsf{Exc} + \{\mathsf{none}\})$, set X, and $A \subseteq X$, the assigment

$$\lambda^{\mathsf{E}}_X(A) = \begin{cases} A + (\mathsf{E} \setminus \{\mathsf{none}\}) & \text{if } \mathsf{none} \in \mathsf{E} \\ \mathsf{E} & \text{otherwise} \end{cases}$$

determines an interpretation of effect types into $\mathbb{E}_{\mathsf{Exc}}$. Intuitively, the interpretation of E requires exceptions possibly raised to be in E, and, if it is allowed that no exception be raised ($\mathsf{none} \in \mathsf{E}$), requires the predicate A to be satisfied.

▶ **Example 27.** Consider the powerset monad $\mathbb{P}$ of Example 2.

1. Taking the ordered monoid $\langle \{0,1\}, \leq, \vee, 0\rangle$, for every set X and $A \subseteq X$, the following assignments determine two interpretations of effect types into $\mathbb{P}$:

$$\forall^1_X(A) = \{B \in PX \mid B \subseteq A\},$$
$$\exists^1_X(A) = \{B \in PX \mid B = \emptyset \text{ or } B \cap A \neq \emptyset\} \textit{and}$$
$$\forall^0_X(A) = \exists^0_X(A) = \{B \in PX \mid B \subseteq A \text{ and } \sharp B \leq 1\}$$

   where $\sharp B$ is the cardinality of B. Intuitively, in both cases, the interpretation of 0 disallows non-determinism, while the interpretation of 1 requires the predicate A to be always satisfied, according to $\forall$, and satisfied in at least one case, according to $\exists$.
2. Taking instead the ordered monoid $\langle \mathbb{N} \cup \{\infty\}, \leq, \cdot, 1\rangle$, we can give a finer version of $\forall$:

$$\forall^n_X(A) = \{B \in PX \mid B \subseteq A \text{ and } \sharp B \leq n\}$$
$$\forall^\infty_X(A) = \{B \in PX \mid B \subseteq A\}$$

   In this way, we can quantify the level of non-determinism in terms of the maximum number of possible outcomes.

[8] Here $M^{\mathrm{op}} \colon \mathcal{S}et^{\mathrm{op}} \to \mathcal{S}et^{\mathrm{op}}$ denotes the functor defined exactly as M but on the opposite category.

Similar interpretations can be defined for the list and subdistribution monads of Example 3.

▶ **Example 28.** Consider the output monad $\mathbb{O}$ of Example 4 for the monoid $\langle A^\infty, \cdot, \epsilon\rangle$ of possibly infinite words over A and the ordered monoid $\langle \mathbb{N}\cup\{\infty\}, \leq, +, 0\rangle$ of effect types. For a word $\sigma \in A^\infty$, we write $|\sigma|$ for its length, which is an element of $\mathbb{N}\cup\{\infty\}$. For every $n \in \mathbb{N}\cup\{\infty\}$, set X and $A \subseteq X$, the assignment $\lambda^n_X(A) = \{\langle\sigma, x\rangle \in OX \mid x \in A, |\sigma| \leq n\}$ determines an interpretation of effect types into $\mathbb{O}$. Intuitively, such interpretation imposes an upper bound (or none) to the length of the outputs.

▶ **Example 29.** Let $\mathcal{E}$ be an ordered monoid of effect types and λ an interpretation of $\mathcal{E}$ into a monad $\mathbb{M}$. Let $\mathcal{E}'$ be another ordered monoid. To give an interpretation of $\mathcal{E}'$ into $\mathbb{M}$, it suffices to give a *lax monoid homomorphism* $f\colon \mathcal{E}' \to \mathcal{E}$, that is, a monotone function $f\colon \langle \mathsf{Eff}', \preceq'\rangle \to \langle \mathsf{Eff}, \preceq\rangle$ such that $1 \preceq f(1')$ and $f(\varepsilon'_1)\cdot f(\varepsilon'_2) \preceq f(\varepsilon'_1\cdot\varepsilon'_2)$. Then, we can define an interpretation ρ of $\mathcal{E}'$ into $\mathbb{M}$ by setting $\rho^{\varepsilon'} = \lambda^{f(\varepsilon')}$ for all $\varepsilon' \in \mathsf{Eff}'$.

Let us fix a monadic operational semantics $\langle \mathbb{M}, \to\rangle$ for a language $\mathcal{L} = \langle \mathsf{Exp}, \mathsf{Val}, \mathsf{ret}\rangle$, a type-and-effect system $\Theta = \langle \mathsf{Ty}, \mathcal{E}, \mathsf{WT}^{\mathsf{E}}, \mathsf{WT}^{\mathsf{V}}\rangle$ for $\mathcal{L}$, and an interpretation λ of $\mathcal{E}$ into $\mathbb{M}$.

Then, we can formally state monadic progress and monadic subject reduction.

▶ **Definition 30** (Monadic Progress). *The type-and-effect system Θ has* monadic progress *if $e \in \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon}$ implies either $e = \mathsf{ret}(v)$ for some $v \in \mathsf{Val}$, or $e \to \mathrm{E}$ for some $\mathrm{E} \in M\mathsf{Exp}$.*

▶ **Definition 31** (Monadic Subject Reduction). *The type-and-effect system Θ has* monadic subject reduction *if $e \in \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon}$ and $e \to \mathrm{E}$ imply $\mathrm{E} \in \lambda^{\varepsilon_1}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon_2})$ for some $\varepsilon_1\cdot\varepsilon_2 \preceq \varepsilon$.*

Monadic progress is standard: a well-typed expression either represents a value or can reduce. Monadic subject reduction, instead, takes into account effects: if an expression of type τ and effect ε reduces to a monadic expression E, then E "has type τ and effect ε" as well, meaning that: ε can be decomposed as $\varepsilon_1\cdot\varepsilon_2$ and E contains computational effects described by ε_1 and expressions of type τ and effect ε_2. In other words, the type τ is preserved and the effect ε is an upper bound of the computational effects produced by the current reduction step, described by ε_1, composed with those produced by future reductions, described by ε_2.

Our next step is expressing type-and-effect soundness. In standard small-step semantics, soundness means that, starting from a well-typed expression, if termination, that is, an expression which cannot be reduced, is reached, then such expression should be a well-typed value. In our monadic operational semantics, termination is conventionally represented by monadic results. Hence, an analogous statement is that, starting from a well-typed expression, if termination, that is, a monadic result, is reached, then this should be a well-typed result, meaning that is satisfies the lifting through the effect type of well-typedness of values.

Again slightly abusing the notation, we will consider predicates $\mathsf{WT}^{\mathsf{V}}_\tau$ on values also as predicates on results. Note that in particular $\mathsf{wrong} \notin \mathsf{WT}^{\mathsf{V}}_\tau$ for all $\tau \in \mathsf{Ty}$.

▶ **Definition 32** (Finitary type-and-effect soundness). *The type-and-effect system Θ is* finitarily sound *if $e \in \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon}$ and $[\![e]\!]_\star = R$ imply $R \in \lambda^{\varepsilon}_{\mathsf{Res}}(\mathsf{WT}^{\mathsf{V}}_\tau)$.*

This notion of soundness is very general: whenever an expression of type τ and effect ε evaluates to a monadic result, this belongs to the interpretation of ε applied to (the image of) values of type τ. Hence, the nature of the soundness property heavily depends on the interpretation λ of effect types. For instance, considering the interpretations for the powerset monad of Example 27, $\forall$ and $\exists$ induce induce a notion of must-soundness, and may-soundness, respectively: the former ensures that the evaluation of a well-typed expression never reaches wrong, while the latter only that it either diverges or reaches at least a well-typed value.

More specifically, it is not guaranteed that the monadic result is actually a monadic value. Formally, viewing $M\mathsf{Val}$ as a subset of $M\mathsf{Res}$, the inclusion $\lambda^{\varepsilon}_{\mathsf{Res}}(\mathsf{WT}^{\mathsf{V}}_{\tau}) \subseteq M\mathsf{Val}$ does not hold in general, as happens for instance with the $\exists$ interpretation. However, we can recover this property when the interpretation λ enjoys an additional condition, as detailed below.

Given $f\colon X \to Y$, the mapping $\mathcal{P}_f\colon \mathcal{P}(Y) \to \mathcal{P}(X)$ has a left adjoint $\mathcal{P}^f\colon \mathcal{P}(X) \to \mathcal{P}(Y)$, that is, a monotone function such that, for every $A \subseteq X$ and $B \subseteq Y$, $\mathcal{P}^f(A) \subseteq B$ if and only if $A \subseteq \mathcal{P}_f(B)$. The function $\mathcal{P}^f$ is the direct image along f, that is, for $A \subseteq X$, $\mathcal{P}^f(A) = \{f(x) \mid x \in A\}$. Then, the following is an easy observation.

▶ **Proposition 33.** *If λ satisfies*
5. $\lambda^{\varepsilon}_{Y}(\mathcal{P}^f(A)) \subseteq \mathcal{P}^{Mf}(\lambda^{\varepsilon}_{X}(A))$ *for $f\colon X \to Y$ and $A \subseteq X$*
then $\lambda^{\varepsilon}_{\mathsf{Res}}(\mathsf{WT}^{\mathsf{V}}_{\tau}) \subseteq M\mathsf{Val}$.

Proof. Recall that we are implicitly using an inclusion $\iota^{\mathsf{Res}}_{\mathsf{Val}}\colon \mathsf{Val} \to \mathsf{Res}$. Making it explicit, the thesis becomes $\lambda^{\varepsilon}_{\mathsf{Res}}(\mathcal{P}^{\iota^{\mathsf{Res}}_{\mathsf{Val}}}(\mathsf{WT}^{\mathsf{V}}_{\tau})) \subseteq \mathcal{P}^{\iota^{\mathsf{Res}}_{\mathsf{Val}}}(M\mathsf{Val})$. This follows from $\lambda^{\varepsilon}_{\mathsf{Res}}(\mathcal{P}^{\iota^{\mathsf{Res}}_{\mathsf{Val}}}(\mathsf{WT}^{\mathsf{V}}_{\tau})) \subseteq \mathcal{P}^{M\iota^{\mathsf{Res}}_{\mathsf{Val}}}(\lambda^{\varepsilon}_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_{\tau})) \subseteq \mathcal{P}^{M\iota^{\mathsf{Res}}_{\mathsf{Val}}}(M\mathsf{Val})$. ◀

Note that the inclusion in Item 5 is actually an equality, since the converse always holds thanks to Item 1 of Definition 24, as $\mathcal{P}^f$ is the left adjoint of $\mathcal{P}_f$. This ensures that a monadic result $\mathrm{R} \in \lambda^{\varepsilon}_{\mathsf{Res}}(\mathsf{WT}^{\mathsf{V}}_{\tau})$ contains only values of type τ, hence, in particular, cannot contain wrong. In fact, the $\exists$ interpretation of Example 27 does not satisfy Item 5 of Proposition 33.

From now on, we assume that Θ has monadic progress and monadic subject reduction, and our goal is to prove that they imply type-and-effect soundness.

We first extend the type-and-effect system to configurations, defining $\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon} \subseteq \mathsf{Conf}$ as

$$\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon} = \begin{cases} \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon} + \mathsf{WT}^{\mathsf{V}}_{\tau} & \text{if } 1 \preceq \varepsilon \\ \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon} & \text{otherwise} \end{cases}$$

Note that wrong is never a well-typed configuration, while configurations which are values of type τ are well-typed with type τ and effect ε only when ε is larger than 1, that is, the type effect does not force raising effects.

Then, we should extend monadic progress and monadic subject reduction to the reduction relation $\xrightarrow[\mathsf{step}]{}$. However, since it is a total function, it trivially enjoys progress, hence, we only have to deal with subject reduction. In the proof, we also use monadic progress of the monadic reduction $\to$ on expressions to ensure that wrong, which is ill-typed, is not produced.

▶ **Lemma 34.** *If $c \in \mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon}$ and $c \xrightarrow[\mathsf{step}]{} \mathrm{C}$, then $\mathrm{C} \in \lambda^{\varepsilon_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\varepsilon_2})$ with $\varepsilon_1 \cdot \varepsilon_2 \preceq \varepsilon$.*

Proof. We split cases on the shape of c.

- $c = e$. From $e \in \mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon}$ we derive $e \in \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon}$. By monadic progress, either $e = \mathsf{ret}(v)$ or $e \to \mathrm{E}$. In the former case, $\mathrm{C} = \eta_{\mathsf{Conf}}(v)$ and, by Definition 23, $1 \preceq \varepsilon$ and $v \in \mathsf{WT}^{\mathsf{V}}_{\tau}$. Hence, the thesis follows by Definition 24(3), taking $\varepsilon_1 = 1$ and $\varepsilon_2 = \varepsilon$. In the latter case, $\mathrm{C} = \hat{\mathrm{E}} = M\iota^{\mathsf{Conf}}_{\mathsf{Exp}}(\mathrm{E})$ and, by monadic subject reduction, $\mathrm{E} \in \lambda^{\varepsilon_1}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon_2})$, with $\varepsilon_1 \cdot \varepsilon_2 \preceq \varepsilon$. From $\mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon_2} = \mathcal{P}_{\iota^{\mathsf{Conf}}_{\mathsf{Exp}}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon_2})$, by Definition 24(1),

  $$\mathrm{E} \in \lambda^{\varepsilon_1}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon_2}) = \lambda^{\varepsilon_1}_{\mathsf{Exp}}(\mathcal{P}_{\iota^{\mathsf{Conf}}_{\mathsf{Exp}}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon_2})) = \mathcal{P}_{M\iota^{\mathsf{Conf}}_{\mathsf{Exp}}}(\lambda^{\varepsilon_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon_2}))$$

  which implies that $\hat{\mathrm{E}} = M\iota^{\mathsf{Conf}}_{\mathsf{Exp}}(\mathrm{E}) \in \lambda^{\varepsilon_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon_2})$, as needed.
- $c = r$. Since $r \in \mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon}$, we have $r \neq \mathsf{wrong}$, hence $r = v$ and this implies that $v \in \mathsf{WT}^{\mathsf{V}}_{\tau}$ and $1 \preceq \varepsilon$. By definition of $\xrightarrow[\mathsf{step}]{}$, we also know that $\mathrm{C} = \eta_{\mathsf{Conf}}(v)$, hence, the thesis follows from Definition 24(3) taking $\varepsilon_1 = 1$ and $\varepsilon_2 = \varepsilon$. ◀

Then, we obtain the following result, showing a form of soundness for the multistep reduction on configurations.

▶ **Theorem 35.** *If* $c \in \mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon}$ *and* $c \xrightarrow{\mathsf{step}}{}^{\star} \mathrm{C}$*, then* $\mathrm{C} \in \lambda^{\varepsilon_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon_2})$ *with* $\varepsilon_1 \cdot \varepsilon_2 \preceq \varepsilon$*.*

Proof. By induction on rules defining $\xrightarrow{\mathsf{step}}{}^{\star}$.

(REFL). We have $\mathrm{C} = \eta_{\mathsf{Conf}}(c)$. By Definition 24(3), $c \in \mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon} \subseteq \mathcal{P}_{\eta_{\mathsf{Conf}}}(\lambda^{1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon}))$, which implies $\mathrm{C} = \eta_{\mathsf{Conf}}(c) \in \lambda^{1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon})$. This proves the thesis since $1 \cdot \varepsilon \preceq \varepsilon$.

(STEP). We know that $c \xrightarrow{\mathsf{step}}{}^{\star} \mathrm{C}_1$ and $\mathrm{C} = \mathrm{C}_1 \gg\!= \mathsf{step}$. By induction hypothesis, $\mathrm{C}_1 \in \lambda^{\varepsilon_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon_2})$ with $\varepsilon_1 \cdot \varepsilon_2 \preceq \varepsilon$. By Lemma 34, we derive $\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon_2} \subseteq \mathcal{P}_{\mathsf{step}}(\lambda^{\varepsilon'_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon'_2}))$ with $\varepsilon'_1 \cdot \varepsilon'_2 \preceq \varepsilon_2$. Then, using Items 1 and 4 of Definition 24, we have

$$\begin{aligned}
\mathrm{C}_1 \in \lambda^{\varepsilon_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon_2}) &\subseteq \lambda^{\varepsilon_1}_{\mathsf{Conf}}(\mathcal{P}_{\mathsf{step}}(\lambda^{\varepsilon'_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon'_2}))) = \mathcal{P}_{M\mathsf{step}}(\lambda^{\varepsilon_1}_{M\mathsf{Conf}}(\lambda^{\varepsilon'_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon'_2})))\\
&\subseteq \mathcal{P}_{M\mathsf{step}}(\mathcal{P}_{\mu_{\mathsf{Conf}}}(\lambda^{\varepsilon_1\cdot\varepsilon'_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon'_2}))) = \mathcal{P}_{\mu_{\mathsf{Conf}}\circ M\mathsf{step}}(\lambda^{\varepsilon_1\cdot\varepsilon'_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon'_2}))\\
&= \mathcal{P}_{(\mathsf{step})^{\dagger}}(\lambda^{\varepsilon_1\cdot\varepsilon'_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon'_2}))
\end{aligned}$$

This implies that $\mathrm{C} = (\mathsf{step})^{\dagger}(\mathrm{C}_1) \in \lambda^{\varepsilon_1\cdot\varepsilon'_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon'_2})$, hence the thesis follows observing that $(\varepsilon_1 \cdot \varepsilon'_1) \cdot \varepsilon'_2 \preceq \varepsilon_1 \cdot \varepsilon_2 \preceq \varepsilon$. ◀

▶ **Theorem 36** (Finitary type-and-effect soundness)**.** *If* $e \in \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon}$ *and* $[\![e]\!]_{\star} = R$ *then* $R \in \lambda^{\varepsilon}(\mathsf{WT}^{\mathsf{V}}_{\tau})$*.*

Theorem 36 states that monadic progress and monadic subject reduction imply soundness with respect to the finitary semantics. To state an analogous result for infinitary semantics, the interpretation of effect types has to take into account the additional structure of the monad.

▶ **Definition 37.** *Let* $\mathbb{M} = \langle M, \sqsubseteq, \mu, \eta \rangle$ *be an* ω*-CPO-ordered monad. An interpretation* λ *of* $\mathcal{E}$ *in* $\mathbb{M}$ *is* ω-CPO-ordered *if, for every effect type* $\varepsilon \in \mathsf{Eff}$*, set* X*, and* $A \subseteq X$*, we have*

1. $\bot_X \in \lambda^{\varepsilon}_X(A)$ *and*
2. *for every* ω*-chain* $(\alpha_n)_{n\in\mathbb{N}}$ *in* MX*,* $\alpha_n \in \lambda^{\varepsilon}_X(A)$ *for all* $n \in \mathbb{N}$ *implies* $\bigsqcup_{n\in\mathbb{N}} \alpha_n \in \lambda^{\varepsilon}_X(A)$*.*

For example, the interpretations in Example 27 are ω-CPO-ordered and also that in Example 28 can be turned into an ω-CPO-ordered one if applied to the pointed output monad. Finally, the construction of Example 29 applies to ω-CPO-ordered interpretations as well.

From now on, we assume that the monad $\mathbb{M}$ has an ω-CPO-ordered structure and the interpretation λ of effect types is ω-CPO-ordered as well.

▶ **Definition 38** (Infinitary type-and-effect soundness)**.** *The type-and-effect system* Θ *is* infinitarily sound *if* $e \in \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon}$ *implies* $[\![e]\!]_{\infty} \in \lambda^{\varepsilon}_{\mathsf{Res}}(\mathsf{WT}^{\mathsf{V}}_{\tau})$*.*

Infinitary soundness states that the limit behaviour of an expression of type τ and effect type ε is a monadic result belonging to the interpretation of ε applied to values of type τ. Observations in Proposition 33 applies to infinitary soundness as well.

In order to prove that monadic progress and monadic subject reduction imply infinitary soundness, we first need a simple property of the function $\mathsf{res} = \mathsf{res}^{\dagger}_0$, introduced at page 8, which is at the basis of the definition of the infinitary semantics.

▶ **Lemma 39.** *If* $\mathrm{C} \in \mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon}$ *then* $\mathsf{res}_0(c) \in \lambda^{\varepsilon}_{\mathsf{Res}}(\mathsf{WT}^{\mathsf{V}}_{\tau})$*.*

▶ **Theorem 40** (Infinitary type-and-effect soundness)**.** *If* $e \in \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon}$ *then* $[\![e]\!]_{\infty} \in \lambda^{\varepsilon}_{\mathsf{Res}}(\mathsf{WT}^{\mathsf{V}}_{\tau})$*.*

6 Example of soundness proof

We show an instance of the technique introduced in the previous section, by proving monadic progress (Theorem 42) and monadic subject reduction (Theorem 45), hence, type-and-effect soundness, for our example. Recall that monadic reduction in Section 4 is parametric on a monad $\mathbb{M}$, and, for each operation op with arity k, a partial function $\mathsf{run}_{op}{:}\mathsf{Val}^k \rightharpoonup M\mathsf{Val}$.

The type-and-effect system defined in Section 4 is an example of Definition 23, where, omitting empty environments and environments in judgments for simplicity:

- Ty is the set of types T as in Figure 5
- $\mathcal{E} = \langle \mathsf{Eff}, \subseteq, \cdot, \{\epsilon\}\rangle$ where Eff is the set of non-empty subsets of Σ^∞
- $\mathsf{WT}^{\mathsf{E}}_{T,E}(e)$ iff $\vdash e : T'!E'$ for some T', E' such that $T'!E' \leq T!E$.
- $\mathsf{WT}^{\mathsf{V}}_{T}(v)$ iff $\vdash v : T'$ for some T' such that $T' \leq T$

In order to prove progress and subject reduction properties, we need a last parameter, that is, an interpretation λ of effect types. Since the proof is parametric on the computational effects raised by operations, these two parameters should agree, as described below.

for each $op\colon T_1 \dots T_n \to T$, and $\overline{v}$ such that $\vdash \overline{v} : \overline{T}'$ and $\overline{T}' \leq \overline{T}$

(RUN) $\quad \mathsf{run}_{op}(\overline{v}) \in \lambda^{\{op\}}_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_{T})$

▶ **Example 41.** We describe interpretations of the effect types suitable for the examples in Section 4. Such interpretations are defined by first mapping[9] the effect types into one of the ordered monoids in Section 5, and then taking the interpretation of the latter into the monad; in this way, as described in Example 29, we get an interpretation of the original effect types. In other words, for an instantiation of the calculus on specific monad and operations, sets of possibly infinite sequences could be reduced to simpler effect types, as exemplified below.

1. In Example 18, we reduce effect types to sets whose elements are either exceptions or none:

$$[\![\epsilon]\!] = \{\mathsf{none}\} \qquad [\![\mathtt{raise}\langle \mathsf{e}\rangle{:}\alpha]\!] = \{\mathsf{e}\} \qquad [\![E]\!] = \bigcup_{\alpha\in E} [\![\alpha]\!]$$

That is, effect types are mapped into those of Example 26, so that, if $[\![E]\!] = \mathsf{E}$, then

$$\lambda^{E}_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_{T}) = \begin{cases} \mathsf{WT}^{\mathsf{V}}_{T} + \mathsf{E} \text{ if } \epsilon \in E \\ \mathsf{E} \text{ otherwise} \end{cases}$$

In this way, monadic values[10] (either values or exceptions) are well-typed if they are either exceptions in E, or, if it is allowed that no exception be raised ($\mathsf{none} \in \mathsf{E}$), well-typed values. Note that an expression such as, e.g., $\mathtt{raise}\langle \mathsf{e}\rangle\mathtt{;raise}\langle \mathsf{e}'\rangle$, gets the effect (reducing to) $\{\mathsf{e}\}$, highlighting the fact that $\mathtt{raise}\langle \mathsf{e}'\rangle$ cannot be reached.

2. In Example 19, the simplest interpretation is to reduce effect types to either 0 or 1:

$[\![\epsilon]\!] = 0$
$[\![\mathtt{choose}{:}\alpha]\!] = 1$
$[\![E]\!] = 1$ if $[\![\alpha]\!] = 1$ for some $\alpha \in E, 0$ otherwise

[9] In all the examples it is easy to see that the mapping is a lax monoid homomorphism.
[10] We explain how the lifting works on values; of course the same applies to expressions and configurations.

That is, effect types are mapped into those of Example 27(1), so that, if $[\![E]\!] = 0$, then

$$\lambda^E_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_T) = \lambda^0_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_T) = \{\mathsf{V} \in P\mathsf{Val} \mid \mathsf{V} \subseteq \mathsf{WT}^{\mathsf{V}}_T \text{ and } \sharp\mathsf{V} \leq 1\}$$

If, instead, $[\![E]\!] = 1$, then we can choose

$$\text{either } \lambda^E_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_T) = \forall^1_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_T) = \{\mathsf{V} \in P\mathsf{Val} \mid \mathsf{V} \subseteq \mathsf{WT}^{\mathsf{V}}_T\}$$
$$\text{or } \lambda^E_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_T) = \exists^1_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_T) = \{\mathsf{V} \in P\mathsf{Val} \mid \mathsf{V} = \emptyset \text{ or } \mathsf{V} \cap \mathsf{WT}^{\mathsf{V}}_T \neq \emptyset\}$$

In this way, monadic values (sets of values, representing possibile results of a computation) are well-typed with a type effect (reducing to) 0 if they have at most one element, and this element, if any, is well-typed; in other words, the computation is deterministic. On the other hand, they are well-typed with a type effect (reducing to) 1 if all the values in the set are well-typed, or there is at least one well-typed value, respectively.

3. A finer interpretation for Example 19 is to reduce effect types to the ordered monoid $\langle \mathbb{N} \cup \{\infty\}, \leq, \cdot, 1\rangle$ of Example 27(2), thus controlling the level of non-determinism:

$$[\![\mathtt{choose}^n]\!] = 2^n \text{ for } n \in \mathbb{N}, [\![\mathtt{choose}^\omega]\!] = \infty$$
$$[\![E]\!] = \sup\{[\![\alpha]\!] \mid \alpha \in E\}$$

Indeed, each call can be seen as a node in a binary tree of choices. In this way, if $[\![E]\!] = k \in \mathbb{N} \cup \{\infty\}$, then monadic values (sets of values, representing possibile results of a computation) are well-typed with E if there are at most 2^k values, hence possible results, in the set and these are all well typed.

4. In Example 21, a possible interpretation of a sequence of $\mathtt{write}\langle\ell\rangle$ is its length:

$$[\![\mathtt{write}\langle\ell\rangle^{\mathtt{n}}]\!] = \mathtt{n} \text{ for } \mathtt{n} \in \mathbb{N} \cup \{\infty\}$$
$$[\![E]\!] = \sup\{[\![\alpha]\!] \mid \alpha \in E\}$$

That is, effect types are mapped into $\mathbb{N} \cup \{\infty\}$ as done in Example 28, so that, if $[\![E]\!] = n$, then $\lambda^E_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_T) = \lambda^n_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_T) = \{\langle\sigma, v\rangle \in O\mathsf{Val} \mid v \in \mathsf{WT}^{\mathsf{V}}_T \text{ and } \mid \sigma \mid \leq n\}$. In this way, an upper bound (or none) is imposed on the length of the produced outputs.

5. A finer interpretation for Example 21 is obtained by taking effect types as they are, and $\lambda^E_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_T) = \{\langle\sigma, v\rangle \in O\mathsf{Val} \mid v \in \mathsf{WT}^{\mathsf{V}}_T, \mathsf{extract}(\sigma) \in E\}$ where, if $\sigma = \langle\ell_1, n_1\rangle \ldots \langle\ell_k, n_k\rangle$, then $\mathsf{extract}(\sigma) = \mathtt{write}\langle\ell_1\rangle \ldots \mathtt{write}\langle\ell_{\mathtt{k}}\rangle$.In this way, effect types can express properties about the order, or the fairness, in which write operations to different output locations can be performed. Similar sophisticated properties can be expressed in cases where different operations can be performed, e.g., reading and updating in the global state monad.

We state now monadic progress and monadic subject reduction for the type-and-effect system in Section 4; as shown in Section 5, they imply monadic soundness. We report only the proof of monadic subject reduction; other proofs and lemmas they depend on are given in [9].

▶ **Theorem 42** (Monadic Progress). *If $e \in \mathsf{WT}^{\mathsf{E}}_{T,E}$ then either $e = \mathtt{return}\ v$ or $e \to_E$.*

The proof of monadic subject reduction uses the standard substitution lemma, and subject reduction for the pure relation $\to_p$ defined in Figure 3. Both properties do not involve any monadic ingredient, and are proved by standard techniques.

▶ **Lemma 43** (Substitution). *If $\Gamma, \overline{x} : \overline{T} \vdash e : T!E$ and $\overline{T'} \leq \overline{T}$, then $\vdash \overline{v} : \overline{T'}$ implies $\Gamma \vdash e[\overline{v}/\overline{x}] : T'!E'$ with $T'!E' \leq T!E$.*

▶ **Lemma 44** (Subject Reduction). *If $\vdash e : T!E$ and $e \to_p e'$ then $\vdash e' : T'!E'$ with $T'!E' \leq T!E$.*

▶ **Theorem 45** (Monadic Subject Reduction). *If* $e \in \mathsf{WT}^{\mathsf{E}}_{T,E}$ *and* $e \to \mathrm{E}$ *then* $\mathrm{E} \in \lambda^{E_1}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{T,E_2})$ *for some* E_1 *and* E_2 *such that* $E_1 \cdot E_2 \subseteq E$.

Proof. From $e \in \mathsf{WT}^{\mathsf{E}}_{T,E}$ we get $\vdash e : T'!E'$ and $T'!E' \leq T!E$. By induction on the reduction rules of Figure 4.

(pure) In this case $e \to_p e'$ and $\mathrm{E} = \eta(e')$. From $\vdash e : T'!E'$ and Lemma 44 we get $\vdash e' : T_1!E_1$ and $T_1!E_1 \leq T'!E'$ and, by transitivity of $\leq$, $e' \in \mathsf{WT}^{\mathsf{E}}_{T,E}$. From Definition 24(3) we derive $\eta(e') \in \lambda^{\{\epsilon\}}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{T,E})$ with $\{\epsilon\} \cdot E \subseteq E$.

(effect) In this case $e = op(\overline{v})$ and $\mathrm{E} = \mathsf{map}\,(\mathtt{return}\ [\ ])\,\mathrm{V}$, with $\mathrm{V} = \mathsf{run}_{op}(\overline{v})$. From rule (T-OP), $op\colon T_1 \dots T_n \to T'$ and $\vdash \overline{v} : \overline{T}'$ and $\overline{T}' \leq \overline{T}$ and $E' = \{op\}$. Hence, by rule (RUN), $\mathrm{V} \in \lambda^{\{op\}}_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_{T'})$. Let $f\colon \mathsf{Val} \to \mathsf{Exp}$ be defined by $f(v) = \mathtt{return}\ v$, then $\mathrm{E} = \mathsf{map}\,(\mathtt{return}\ [\ ])\,\mathrm{V} = Mf(\mathrm{V})$. From Definition 24(1) and $\mathsf{WT}^{\mathsf{V}}_{T'} = \mathcal{P}_f(\mathsf{WT}^{\mathsf{E}}_{T',\{\epsilon\}})$

$$\mathrm{V} \in \lambda^{\{op\}}_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_{T'}) = \lambda^{\{op\}}_{\mathsf{Val}}(\mathcal{P}_f(\mathsf{WT}^{\mathsf{E}}_{T',\{\epsilon\}}) = \mathcal{P}_{Mf}(\lambda^{\{op\}}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{T',\{\epsilon\}}))$$

This implies $\mathrm{E} = Mf(v) \in \lambda^{\{op\}}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{T',\{\epsilon\}})$. Since $T' \leq T$, $\mathsf{WT}^{\mathsf{E}}_{T',\{\epsilon\}} \subseteq \mathsf{WT}^{\mathsf{E}}_{T,\{\epsilon\}}$, hence by monotonicity of $\lambda^{\{op\}}_{\mathsf{Exp}}$, we get $\mathrm{E} \in \lambda^{\{op\}}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{T,\{\epsilon\}})$, with $\{op\}\cdot\{\epsilon\} = \{op\} \subseteq E$.

(ret) In this case $e = \mathtt{do}\ x = \mathtt{return}\ v\mathtt{;}\ e'$ and $\mathrm{E} = \eta(e'[v/x])$. From rules (T-DO) and (T-RET), $\vdash v : T_1$ and $x : T_1' \vdash e' : T'!E'$, with $T_1 \leq T_1'$. By Lemma 43, we get $\vdash e'[v/x] : T''!E''$ with $T''!E'' \leq T'!E'$. Hence, $T''!E'' \leq T!E$ and so $e'[v/x] \in \mathsf{WT}^{\mathsf{E}}_{T,E}$. Finally, from Definition 24(3), $\eta(e'[v/x]) \in \lambda^{\{\epsilon\}}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{T,E})$ with $\{\epsilon\}\cdot E = E$.

(do) In this case $e = \mathtt{do}\ x = e_1\mathtt{;}\ e_2$ and $\mathrm{E} = \mathsf{map}\,(\mathtt{do}\ x = [\ ]\mathtt{;}\ e_2)\,\mathrm{E}_1$ and $e_1 \to \mathrm{E}_1$. From rule (T-DO), $\vdash e_1 : T_1!E_1$ and $x : T_1' \vdash e_2 : T'!E_2$ with $E' = E_1 \cdot E_2$ and $T_1 \leq T_1'$. Hence, from $e_1 \in \mathsf{WT}^{\mathsf{E}}_{T_1,E_1}$, by induction hypothesis we get that $\mathrm{E}_1 \in \lambda^{E_1'}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{T_1,E_2'})$ with $E_1' \cdot E_2' \subseteq E_1$. Let $f\colon \mathsf{Exp} \to \mathsf{Exp}$ be defined by $f(\hat{e}) = \mathtt{do}\ x = \hat{e}\mathtt{;}\ e_2$, hence $\mathrm{E} = Mf(\mathrm{E}_1)$. By rule (T-DO), we know that $\hat{e} \in \mathsf{WT}^{\mathsf{E}}_{T_1,\hat{E}}$ implies $f(\hat{e}) \in \mathsf{WT}^{\mathsf{E}}_{T',\hat{E}\cdot E_2} \subseteq \mathsf{WT}^{\mathsf{E}}_{T,\hat{E}\cdot E_2}$, that is, $\mathsf{WT}^{\mathsf{E}}_{T_1,\hat{E}} \subseteq \mathcal{P}_f(\mathsf{WT}^{\mathsf{E}}_{T,\hat{E}\cdot E_2})$. From Definition 24(1) and monotonicity of $\lambda^{E_1'}_{\mathsf{Exp}}$ we get

$$\mathrm{E}_1 \in \lambda^{E_1'}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{T_1,E_2'}) \subseteq \lambda^{E_1'}_{\mathsf{Exp}}(\mathcal{P}_f(\mathsf{WT}^{\mathsf{E}}_{T,E_2'\cdot E_2})) = \mathcal{P}_{Mf}(\lambda^{E_1'}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{T,E_2'\cdot E_2}))$$

that is, $\mathrm{E} = Mf(\mathrm{E}_1) \in \lambda^{E_1'}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{T,E_2'\cdot E_2})$, and we get the thesis since $E_1'\cdot E_2'\cdot E_2 \subseteq E_1 \cdot E_2 \subseteq E'$. ◀

The results hold for the core calculus in Figure 2, for an arbitrary family Σ of operations. The calculus, the type system and the proofs can be modularly extended by just considering cases for additional constructs, as we will do in Section 7 for handlers. Extending the subtyping relation, instead, requires some care to preserve the needed properties.[11]

7 Handlers

We extend Λ_Σ with *handlers*, showing how our framework can deal with more sophisticated language features and, at the same time, how proofs can be modularly extended. In particular, it is important to illustrate that monadic semantics can incorporate handlers. Constructs and terminology are inspired by those for algebraic effects, see, e.g., [40]; however, the approach is different since our calculus, being based on generic effects, has no explicit continuations.

[11] For instance, adding $\mathtt{Bot} \leq T$ for all T as in Example 18 is sound since $\mathtt{Bot}$ is an empty type.

$$
\begin{array}{lcll}
e & ::= & \ldots \mid \texttt{handle}\ e\ \texttt{with}\ h & \text{expression with handler} \\
h & ::= & \overline{c}, x \mapsto e & \text{handler} \\
c & ::= & op(\overline{x}) \mapsto_{\mu} e & \text{clause} \\
\mu & ::= & \texttt{c} \mid \texttt{s} & \text{mode}
\end{array}
$$

Figure 7 Syntax of handlers.

The syntax is reported in Figure 7. A handler specifies a *final expression*, and a sequence of *clauses*, assumed to be a map, that is, there can be at most one clause for an operation. Such a clause, if any, handles a call of the operation by executing the clause expression. After that, the final expression is either executed or not depending on the *mode*, either `c` or `s`, for "continue" and "stop", respectively. As illustrated in the following examples, a `c`-clause replaces an effect with an alternative behaviour in a continuous manner, whereas in `s`-clauses handling the computational effect interrupts the normal flow of execution.

$$h = \overline{c}, x \mapsto e'$$

$$\text{(WITH-DO)}\ \frac{}{\texttt{handle do}\ y = e_1\texttt{;}\ e_2\ \texttt{with}\ h \to_p \texttt{handle}\ e_1\ \texttt{with}\ \overline{c}, y \mapsto (\texttt{handle}\ e_2\ \texttt{with}\ h)}$$

$$\text{(WITH-RET)}\ \frac{}{\texttt{handle return}\ v\ \texttt{with}\ h \to_p \texttt{do}\ x = \texttt{return}\ v\texttt{;}\ e'}$$

$$\text{(WITH-CONTINUE)}\ \frac{}{\texttt{handle}\ op(\overline{v})\ \texttt{with}\ h \to_p \texttt{do}\ x = e[\overline{v}/\overline{x}]\texttt{;}\ e'}\ op(\overline{x}) \mapsto_{\texttt{c}} e \in \overline{c}$$

$$\text{(WITH-STOP)}\ \frac{}{\texttt{handle}\ op(\overline{v})\ \texttt{with}\ h \to_p e[\overline{v}/\overline{x}]}\ op(\overline{x}) \mapsto_{\texttt{s}} e \in \overline{c}$$

$$\text{(WITH-FWD)}\ \frac{}{\texttt{handle}\ op(\overline{v})\ \texttt{with}\ h \to_p \texttt{do}\ x = op(\overline{v})\texttt{;}\ e'}\ op \notin \overline{c}$$

$$\text{(WITH-CTX)}\ \frac{e \to_p e'}{\texttt{handle}\ e\ \texttt{with}\ h \to_p \texttt{handle}\ e'\ \texttt{with}\ h}$$

Figure 8 Pure reduction with handlers.

The pure reduction extended with handlers is shown in Figure 8. The behaviour of an expression with handler depends on the shape of the handled expression.

In case of a `do` composition of two subexpressions, the `do` is eliminated by reducing to the first subexpression with as final expression the second one; clauses are propagated to both the subexpressions. In case of a `return`, the handler is eliminated by reducing to the `do` composition of the handled expression and the final expression.

In case of an operation call, the behaviour depends on whether a matching clause is found or not. If it is found, then the clause expression is executed, after replacing parameters by arguments, as shown in rules (WITH-CONTINUE) and (WITH-STOP). In a `c`-clause, the final expression is executed as well. If there is no matching clause, instead, the handler is eliminated, by reducing to the `do` composition of the operation call and the final expression. The outcome is that the operation call is forwarded to be possibly handled by an outer level. gFinally, the contextual rule is as expected.

To extend the type-and-effect system, we rely on *filter functions* associated to handlers, which describe how they transform effects, by essentially replacing operations matching some clause with the effect of the clause expression.

To this end, first we define a *(handler) filter* H to be the information about transforming effects which can be extracted from a handler, as shown in the top section of Figure 9. Then, given a filter H, we define the associated function $\widehat{\mathcal{F}}_H\colon \mathsf{Eff} \to \mathsf{Eff}$. This function, as shown in the bottom section of Figure 9, is obtained on top of the function $\mathcal{F}_H\colon \Sigma^\infty \to \mathsf{Eff}^\infty$ which transforms a single possibly infinite sequence of operations into a possibly infinite sequence of effects. The latter is transformed into a unique effect by taking the possibly infinite concatenation of its elements, denoted $\bullet^\infty\colon \mathsf{Eff}^\infty \to \mathsf{Eff}$. Finally, the function is extended to effects (sets of sequences) in the obvious way.

$$\begin{array}{lcll} H & ::= & \overline{C}, E & \text{filter} \\ C & ::= & op \mapsto_\mu E & \text{clause filter} \end{array}$$

$H = op_1 \mapsto_{\mu_1} E_1 \dots op_n \mapsto_{\mu_n} E_n, E$

$\mathcal{F}_H\colon \Sigma^\infty \to \mathsf{Eff}^\infty$ coinductively defined by:

$\mathcal{F}_H(\epsilon) = E$

$\mathcal{F}_H(op_i{:}\alpha) = E_i{:}\mathcal{F}_H(\alpha) \qquad i \in 1..n, \mu_i = \mathsf{c}$

$\mathcal{F}_H(op_i{:}\alpha) = E_i \qquad i \in 1..n, \mu_i = \mathsf{s}$

$\mathcal{F}_H(op{:}\alpha) = \{op\}{:}\mathcal{F}_H(\alpha) \qquad op \neq op_i \text{ for all } i \in 1..n$

$\widehat{\mathcal{F}}_H\colon \mathsf{Eff} \to \mathsf{Eff}$

$\widehat{\mathcal{F}}_H(E) = \bigcup_{\alpha \in E} \{\bullet^\infty \mathcal{F}_H(\alpha)\}$

Figure 9 Filters.

$$\text{(T-WITH)}\ \frac{\Gamma \vdash e : T!E \quad \Gamma;\, T' \vdash h : T''!H}{\Gamma \vdash \texttt{handle}\ e\ \texttt{with}\ h : T''!\mathcal{F}_H(E)}\ T \leq T'$$

$$\text{(T-HANDLER)}\ \frac{\Gamma, x : T \vdash e' : T'!E' \quad \Gamma;\, T'' \vdash c_i : C_i}{\Gamma;\, T \vdash c_1 \dots c_n, x \mapsto e' : T''!C_1 \dots C_n, E'}\ T' \leq T''$$

$$\text{(T-CONTINUE)}\ \frac{\Gamma, \overline{x}{:}\overline{T} \vdash e : T''!E'}{\Gamma;\, T' \vdash op(\overline{x}) \mapsto_{\mathsf{c}} e : op \mapsto_{\mathsf{c}} E'}\ \begin{array}{l} op{:}\,\overline{T} \to T \\ T'' \leq T \end{array}$$

$$\text{(T-STOP)}\ \frac{\Gamma, \overline{x}{:}\overline{T} \vdash e : T''!E'}{\Gamma;\, T' \vdash op(\overline{x}) \mapsto_{\mathsf{s}} e : op \mapsto_{\mathsf{s}} E'}\ \begin{array}{l} op{:}\,\overline{T} \to T \\ T'' \leq T' \end{array}$$

Figure 10 Typing rules for handlers.

In Figure 10 we show the typing rules for expressions with handlers. In rule (T-WITH), in order to typecheck an expression with handler, first we get the type and effect of the handled expression. The type is used to typecheck the handler, as (subtype of the) type of the parameter of the final expression, see rule (T-HANDLER). Typechecking the handler we get a type, being that of the final expression, which will be the type of the whole expression. Moreover, we extract from the handler a filter, which is used to transform the effect E of the handled expression, getting the resulting effect of the whole expression. In detail, as formally described in Figure 9, the filter transforms any sequence of operations in E by replacing the first operation matching some clause, if any, with the effect of the clause expression; then, the remaining sequence is disregarded if the clause is s, otherwise filtered in turn. If the sequence to be filtered is finite, and no matching s-clause is found, then the final effect is appended in the end.

In rule (T-HANDLER), as said above, the type on the left of the judgment is used as type of the parameter of the final expression, whose type will be returned by the handler. This type is also needed to typecheck s-clauses, see below. The filter extracted from the handler consists in a clause filter for each clause, and the effect of the final expression.

For each clause, the extracted filter consists of the operation name, mode, and effect of the expression, as shown in rules (T-CONTINUE) and (T-STOP). A c-clause is meant to provide alternative code to be executed before the final expression, hence the type of the clause expression should be (a subtype of) the return type of the operation. In a s-clause, instead, the result of the clause expression becomes that of the whole expression with handler, hence the type of the former should be (a subtype of) the latter.

▶ **Example 46.** We show handlers for some of the previous examples. A handler of shape $\overline{c}, x \mapsto \texttt{return}\ x$ is abbreviated by $\overline{c}$.

1. Set $h = \texttt{raise}\langle\mathsf{PredZero}\rangle() \mapsto_{\mathsf{s}} \texttt{return}\ 0$. Then

$$\begin{array}{lcl} \texttt{handle}\ \mathsf{predfun}\,0\ \texttt{with}\ h & \Rightarrow^{\star} & \texttt{handle}\ \texttt{raise}\langle\mathsf{PredZero}\rangle\ \texttt{with}\ h \\ & \Rightarrow & \texttt{return}\ 0 \\ & \Rightarrow & 0 \end{array}$$

   As shown in Example 22(1), we get the judgment $\emptyset \vdash \mathsf{predfun}\,0 : \texttt{Nat}!\{\epsilon, \texttt{raise}\langle\mathsf{PredZero}\rangle\}$. On the other hand, with the handler we get

$$\emptyset \vdash \texttt{handle}\ \mathsf{predfun}\,0\ \texttt{with}\ h : \texttt{Nat}!\{\epsilon\}$$

   since $\widehat{\mathcal{F}}_H(\{\epsilon, \texttt{raise}\langle\mathsf{PredZero}\rangle\}) = \{\epsilon\}$ where $H = \texttt{raise}\langle\mathsf{PredZero}\rangle \mapsto_{\mathsf{s}} \{\epsilon\}, \{\epsilon\}$ is the filter extracted from h. As the reader could expect, an s-clause is appropriate in this case. With a c-clause, see rule (T-CONTINUE), the type of the clause expression should be (a subtype of) the return type of the operation, which is `Bot`. Since no value has type `Bot`, no value could be returned, as already noted in [35].[12]

2. Assuming the function even: `Nat` $\to$ `Bool` checking the parity of a number, set

$$h_1 = \texttt{write}\langle\ell'\rangle(x) \mapsto_{\mathsf{c}} \texttt{write}\langle\ell\rangle(x)$$
$$h_2 = \texttt{write}\langle\ell'\rangle(x) \mapsto_{\mathsf{c}} \texttt{if}\ \mathsf{even}(x)\ \texttt{then return}\ x\ \texttt{else write}\langle\ell\rangle(x)$$

   Then

$$[\![\texttt{handle}\ \mathsf{wfun}^{\uparrow}\,0\ \texttt{with}\ h_1]\!]_{\infty} = \langle\langle\ell,0\rangle\cdot\langle\ell,0\rangle\cdot\langle\ell,1\rangle\cdot\langle\ell,1\rangle\cdot\ldots\cdot\langle\ell,n\rangle\cdot\langle\ell,n\rangle\cdot\ldots,\bot\rangle$$
$$[\![\texttt{handle}\ \mathsf{wfun}^{\uparrow}\,0\ \texttt{with}\ h_2]\!]_{\infty} = \langle\langle\ell,0\rangle\cdot\langle\ell,1\rangle\cdot\langle\ell,1\rangle\cdot\ldots\langle\ell,2k\rangle\cdot\langle\ell,2k{+}1\rangle\cdot\langle\ell,2k{+}1\rangle\cdot\ldots,\bot\rangle$$
$$[\![\texttt{handle}\ \mathsf{wfun}^{\downarrow}\,n\ \texttt{with}\ h_1]\!]_{\star} = \langle\langle\ell,n\rangle\cdot\langle\ell,n\rangle\cdot\ldots\cdot\langle\ell,0\rangle\cdot\langle\ell,0\rangle,\texttt{unit}\rangle$$
$$[\![\texttt{handle}\ \mathsf{wfun}^{\downarrow}\,2k\ \texttt{with}\ h_2]\!]_{\star} = \langle\langle\ell,2k\rangle\cdot\langle\ell,2k{-}1\rangle\cdot\langle\ell,2k{-}1\rangle\ldots\cdot\langle\ell,1\rangle\cdot\langle\ell,1\rangle\cdot\langle\ell,0\rangle,\texttt{unit}\rangle$$

   In this case, a c-clause is appropriate, since the aim is to continuously handle the $\texttt{write}\langle\ell'\rangle$ operation. By the typing judgments shown in Example 22(3), we get

$$\emptyset \vdash \mathsf{wfun}^{\uparrow}\,0 : \texttt{Unit}!\{(\texttt{write}\langle\ell\rangle\cdot\texttt{write}\langle\ell'\rangle)^{\omega}\}$$
$$\emptyset \vdash \mathsf{wfun}^{\downarrow} : \texttt{Unit}!\{(\texttt{write}\langle\ell\rangle\cdot\texttt{write}\langle\ell'\rangle)^{\mathtt{n}} \mid \mathtt{n} \geq 1\}$$

[12] Hence, the clause expression could only be another raise or a diverging expression.

On the other hand, with the handler we get, with $\alpha ::= \epsilon \mid \texttt{write}\langle\ell\rangle$

$$\emptyset \vdash \texttt{handle wfun}^{\uparrow}\, 0 \texttt{ with } h_1 : \texttt{Unit!}\{(\texttt{write}\langle\ell\rangle \cdot \texttt{write}\langle\ell\rangle)^{\omega}\}$$
$$\emptyset \vdash \texttt{handle wfun}^{\uparrow}\, 0 \texttt{ with } h_2 : \texttt{Unit!}\{\alpha \cdot \texttt{write}\langle\ell\rangle)^{\omega}\}$$
$$\emptyset \vdash \texttt{wfun}^{\downarrow}\, n : \texttt{Unit!}\{\alpha \cdot \texttt{write}\langle\ell'\rangle)^{\texttt{n}} \mid \texttt{n} \geq \texttt{1}\}$$

As already noted, effect types only provide a static approximation of the computational effects; notably, in the last two judgments, the effect type contains other sequences besides the two which can be actually performed, depending on the argument.

Soundness for handlers. The results of Section 6 can be extended to handlers. For monadic subject reduction we only need to show subject reduction for the newly introduced rules, since they are pure. The proofs of the results are in the extended version [9].

▶ **Lemma 47** (Monadic Progress for handlers). *Set e of shape* `handle _ with _`.
If $\vdash e : T!E$ then $e \to \mathrm{E}$ for some $\mathrm{E} \in M\mathsf{Exp}$.

▶ **Lemma 48** (Subject Reduction for handlers). *Set e of shape* `handle _ with _`.
If $\vdash e : T!E$ and $e \to_p e'$, then $\vdash e' : T'!E'$ such that $T'!E' \leq T!E$.

8 Related work and conclusion

Monadic semantics. The idea that monads can model computational effects in programming languages goes back to the pioneering Moggi's work [30, 31]. He showed that one can use (strong) monads to organise the denotational semantics of effectful languages, interpreting impure expressions as functions (actually arrows of an arbitrary category) returning monadic values, which can be sequenced by Kleisli composition. However, the structure of a monad does not include any operation for actually raising computational effects, which thus need to be defined ad-hoc in specific instances. Moreover, monads are difficult to combine, requiring non trivial notions like monad transformers [27, 22].

To overcome these difficulties and make the model closer to the syntax, Plotkin and Power [33, 34, 35] introduced algebraic effects which, instead, explicitly consider operations to raise computational effects. These can be interpreted by additional structure on the monad and, moreover, when equipped with an equational theory, they actually determine a monad, which provides a syntactic model for the language. Thus, one reduces the problem of combining monads to the much easier problem of combining theories [20], greatly increasing modularity.

An alternative, essentially equivalent, way of interpreting algebraic operations is by means of runners, a.k.a. comodels [39, 36, 44, 2]. Roughly, runners describe how operations are executed by the system, that is, how they transform the environment where they are run. This essentially amounts to giving an interpretation of operations in the state monad. More general runners, where the system is modelled in a more expressive way, are considered by [2], where the state monad is combined with errors and system primitive operations.

On the operational side, algebraic effects are typically treated as uninterpreted operations, that is, the evaluation process just builds a tree of operation calls [23, 42, 48, 40]. Monadic operational semantics for λ-calculi with algebraic effects are also considered, mainly in the form of a monadic definitional interpreter (see, e.g., [27, 12, 15, 11, 8]). That is, they directly define a function from expressions to monadic values, which essentially corresponds to our infinitary semantics. Small-step approaches are also considered by [16, 17]. The former tackles a different problem, that is, studying monadic rewriting systems, which require the

use of sophisticated relational techniques, and thus restricts the class of available monads. We can avoid these difficulties since we focus on deterministic rewriting, which can be addressed using just sets and functions. The latter, instead, studies a specific calculus where, as already noticed, the way sequences of steps are constructed is very close to ours; however, they do not need to introduce wrong in configurations, as type errors are prevented syntactically.

Type-and-effect systems. Type-and-effect systems, or simply effect systems [45, 32, 46, 29, 25], are the most popular way of statically controlling computational effects. Many have been designed for specific notions of computational effect and implemented in mainstream programming languages, the most well-known being the mechanism of Java checked exceptions. Katsumata [25] recognized that effect systems share a common algebraic structure, notably they form an ordered monoid, and gave them denotational semantics through parametric monads, using a structure equivalent to our notion of interpretation (see Remark 25).

Effect handlers. Plotkin and Pretnar [37, 38] introduced effect handlers as a generalisation of exception handling mechanisms. They are an extremely powerful programming abstraction, allowing to describe the semantics of algebraic operations in the language itself, thus enabling the simulation of several effectful programs, such as stream redirection or cooperative concurrency [38, 24, 4, 40]. When a call to an algebraic operation is caught, the alternative code can resume the original computation using a form of continuation-passing style. Other forms of handlers have been considered, notably, shallow handlers [24, 19], where only the first call to an operation is handled. Our handlers are inspired by those for algebraic effects, see, e.g., [40]; however, the approach is different since our calculus has no explicit continuations.

Summary. In the research on foundations of programming languages, it is a routine task to describe execution through a small-step reduction, and prove progress and subject reduction for the type system. Can this be smoothly combined with the long-established approach where computational effects are modularly modeled by a monad, so to enjoy all the advantages of separation of concerns? The answer provided in this paper is yes. Notably, we provide a meta-theory defining abstract notions of monadic small-step semantics and type-and-effect system, and prove that type-and-effect soundness is implied by progress and subject reduction properties, with an inductive argument similar to the standard one.

This overall achievement relies on two key specific contributions. On one hand, we provide a canonical way to construct, on top of a monadic reduction, a small-step operational semantics where computations, even though always represented by infinite sequences, can be distinguished as either non-terminating, or successfully terminating, or stuck. On the other hand, we provide a formal model of the "meaning" of effect types, independent in principle from the underlying language and type system.

Discussion and future work. The way we define the "transitive closure" of a monadic reduction, which is a relation from a set to a different one, is similar, as said, to that proposed by [17]. Notably, such reduction is assumed to be deterministic, since starting from an arbitrary relation would require a relational extension of the monad [3]. Confluence as well would require strong assumptions on the monad, notably some form of commutativity, ruling out most of the relevant examples. Moreover, the aim here is to prove soundness for programming languages, which typically adopt a deterministic evaluation strategy. Differently from [17], we provide a language independent definition; moreover, whereas they consider an intrinsically total reduction, in this paper, as mentioned above, we address the additional problem to characterize stuck computations, as needed to express soundness.

In our framework, non-termination is always possibile, rather than be considered as an effect. This is essentially a choice we made, possibly influenced by the fact that in standard soundness we have three possible outcomes: non-termination, termination with a value, and stuck. The coinductive Delay monad [5] could be an alternative approach to define the infinitary semantics, assuming a way to be combined with the monad modeling computational effects, that is, a distributive law. The relationship between these two approaches, as far as we know, is not clear, and is an interesting direction to be investigated.

Our definition of ω-CPO-ordered monad is given for monads on $\mathcal{S}et$. A challenging and relevant problem is to consider a category different from $\mathcal{S}et$; our feeling is that the notion could be generalized by considering a monad $\mathbb{M}$ on a category $\mathcal{C}$ such that the Kleisli category $\mathcal{C}^{\mathbb{M}}$ is CPO-enriched.

In this paper, where the focus is different, we did not study decidability of the type-and-effect system; we did not even provide a syntactic representation of effects, which are considered semantic entities, notably possibly infinite sets of possibly infinite sequences. Of course decidability is a very important issue to be investigated; the first step should then be to choose a finite representation, e.g., by means of a system of guarded equations.

We illustrated our approach by a lambda-calculus with generic effects. Clearly, it would be important to investigate how other calculi can be formalized as to take advantage of the meta-theory. Notably, we plan to apply the approach to an object-oriented calculus.

Finally, here we considered non-standard handlers, as our calculus is based on generic effects and so it does not use explicit continuations. Hence, it would be nice to investigate the precise relationship between them and handlers for algebraic effects used in the literature.

References

1 Jirí Adámek, Nathan J. Bowler, Paul Blain Levy, and Stefan Milius. Coproducts of monads on set. *CoRR*, abs/1409.3804, 2014. arXiv:1409.3804.

2 Danel Ahman and Andrej Bauer. Runners in action. In Peter Müller, editor, *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020*, volume 12075 of *Lecture Notes in Computer Science*, pages 29–55. Springer, 2020. doi:10.1007/978-3-030-44914-8_2.

3 Michael Barr. Relational algebras. In S. MacLane, H. Applegate, M. Barr, B. Day, E. Dubuc, Phreilambud, A. Pultr, R. Street, M. Tierney, and S. Swierczkowski, editors, *Reports of the Midwest Category Seminar IV*, number 137 in Lecture Notes in Mathematics, pages 39–55, Berlin, Heidelberg, 1970. Springer Berlin Heidelberg.

4 Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015. doi:10.1016/J.JLAMP.2014.02.001.

5 Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), 2005. doi:10.2168/LMCS-1(2:1)2005.

6 Francesco Dagnino. A meta-theory for big-step semantics. *ACM Transactions on Computational Logic*, 23(3):20:1–20:50, 2022. doi:10.1145/3522729.

7 Francesco Dagnino, Viviana Bono, Elena Zucca, and Mariangiola Dezani-Ciancaglini. Soundness conditions for big-step semantics. In Peter Müller, editor, *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020*, volume 12075 of *Lecture Notes in Computer Science*, pages 169–196. Springer, 2020. doi:10.1007/978-3-030-44914-8_7.

8 Francesco Dagnino and Francesco Gavazzo. A fibrational tale of operational logical relations: Pure, effectful and differential. *Logical Methods in Computer Science*, 20(2), 2024. doi:10.46298/LMCS-20(2:1)2024.

9 Francesco Dagnino, Paola Giannini, and Elena Zucca. Monadic type-and-effect soundness (extended version). *CoRR*, 2025. URL: http://arxiv.org/abs/2504.10159.

10 Francesco Dagnino and Giuseppe Rosolini. Doctrines, modalities and comonads. *Mathematical Structures in Computer Science*, pages 1–30, 2021. doi:10.1017/S0960129521000207.

11 Ugo Dal Lago and Francesco Gavazzo. A relational theory of effects and coeffects. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–28, 2022. doi:10.1145/3498692.

12 David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. Abstracting definitional interpreters (functional pearl). *Proceedings of the ACM on Programming Languages*, 1(ICFP):12:1–12:25, 2017. doi:10.1145/3110256.

13 Samuel Eilenberg and John C. Moore. Adjoint functors and triples. *Illinois Journal of Mathematics*, 9(3):381–398, 1965. doi:10.1215/ijm/1256068141.

14 Soichiro Fujii, Shin-ya Katsumata, and Paul-André Melliès. Towards a formal theory of graded monads. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures, 19th International Conference, FoSSaCS 2016*, volume 9634 of *Lecture Notes in Computer Science*, pages 513–530. Springer, 2016. doi:10.1007/978-3-662-49630-5_30.

15 Francesco Gavazzo. Quantitative behavioural reasoning for higher-order effectful programs: Applicative distances. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018*, pages 452–461. ACM, 2018. doi:10.1145/3209108.3209149.

16 Francesco Gavazzo and Claudia Faggian. A relational theory of monadic rewriting systems, part I. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021*, pages 1–14. IEEE, 2021. doi:10.1109/LICS52264.2021.9470633.

17 Francesco Gavazzo, Riccardo Treglia, and Gabriele Vanoni. Monadic intersection types, relationally. In Stephanie Weirich, editor, *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024*, volume 14576 of *Lecture Notes in Computer Science*, pages 22–51. Springer, 2024. doi:10.1007/978-3-031-57262-3_2.

18 Alexander Grothendieck. Catégories fibrées et descente. In *Revêtements étales et groupe fondamental*, pages 145–194. Springer, 1971.

19 Daniel Hillerström and Sam Lindley. Shallow effect handlers. In Sukyoung Ryu, editor, *Proceedings of the 16th Asian Symposium on Programming Languages and Systems, APLAS 2018*, volume 11275 of *Lecture Notes in Computer Science*, pages 415–435. Springer, 2018. doi:10.1007/978-3-030-02768-1_22.

20 Martin Hyland, Gordon D. Plotkin, and John Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357(1-3):70–99, 2006. doi:10.1016/J.TCS.2006.03.013.

21 Bart Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*, volume 59 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2016. doi:10.1017/CBO9781316823187.

22 Mauro Jaskelioff and Eugenio Moggi. Monad transformers as monoid transformers. *Theoretical Computer Science*, 411(51-52):4441–4466, 2010. doi:10.1016/J.TCS.2010.09.011.

23 Patricia Johann, Alex Simpson, and Janis Voigtländer. A generic operational metatheory for algebraic effects. In *Proceedings of the 25th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2010*, pages 209–218. IEEE Computer Society, 2010. doi:10.1109/LICS.2010.29.

24 Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP 2013*, pages 145–158. ACM, 2013. doi:10.1145/2500365.2500590.

25 Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In Suresh Jagannathan and Peter Sewell, editors, *Proceedings of the 41st ACM/SIGPLAN Symposium on Principles of Programming Languages, POPL 2014*, pages 633–646. ACM, 2014. doi:10.1145/2535838.2535846.

26 Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, 2003. doi:10.1016/S0890-5401(03)00088-9.

27 Sheng Liang, Paul Hudak, and Mark P. Jones. Monad transformers and modular interpreters. In Ron K. Cytron and Peter Lee, editors, *Proceedings of the 22nd ACM/SIGPLAN Symposium on Principles of Programming Languages, POPL 1995*, pages 333–343. ACM Press, 1995. doi:10.1145/199448.199528.

28 Ernest G. Manes. *Algebraic Theories*. Graduate Texts in Mathematics. Springer, 1976. doi:10.1007/978-1-4612-9860-1.

29 Daniel Marino and Todd D. Millstein. A generic type-and-effect system. In Andrew Kennedy and Amal Ahmed, editors, *TLDI'09: Types in Languages Design and Implementatio*, pages 39–50. ACM Press, 2009. doi:10.1145/1481861.1481868.

30 Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the 4th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 1989*, pages 14–23. IEEE Computer Society, 1989. doi:10.1109/LICS.1989.39155.

31 Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. doi:10.1016/0890-5401(91)90052-4.

32 Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design, Recent Insight and Advances*, volume 1710 of *Lecture Notes in Computer Science*, pages 114–136. Springer, 1999. doi:10.1007/3-540-48092-7_6.

33 Gordon D. Plotkin and John Power. Adequacy for algebraic effects. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FoSSaCS 2001*, volume 2030 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001. doi:10.1007/3-540-45315-6_1.

34 Gordon D. Plotkin and John Power. Notions of computation determine monads. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures, 5th International Conference, FoSSaCS 2002*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002. doi:10.1007/3-540-45931-6_24.

35 Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003. doi:10.1023/A:1023064908962.

36 Gordon D. Plotkin and John Power. Tensors of comodels and models for operational semantics. In Andrej Bauer and Michael W. Mislove, editors, *The 24th Conference on Mathematical Foundations of Programming Semantics, MFPS 2008*, volume 218 of *Electronic Notes in Theoretical Computer Science*, pages 295–311. Elsevier, 2008. doi:10.1016/J.ENTCS.2008.10.018.

37 Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems - 18th European Symposium on Programming, ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009. doi:10.1007/978-3-642-00590-9_7.

38 Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013. doi:10.2168/LMCS-9(4:23)2013.

39 A. John Power and Olha Shkaravska. From comodels to coalgebras: State and arrays. In Jirí Adámek and Stefan Milius, editors, *Proceedings of the Workshop on Coalgebraic Methods in Computer Science, CMCS 2004*, volume 106 of *Electronic Notes in Theoretical Computer Science*, pages 297–314. Elsevier, 2004. doi:10.1016/J.ENTCS.2004.02.041.

40 Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. In Dan R. Ghica, editor, *The 31st Conference on Mathematical Foundations of Programming Semantics, MFPS 2015*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 19–35. Elsevier, 2015. doi:10.1016/J.ENTCS.2015.12.003.

41 Emily Riehl. *Category theory in context*. Courier Dover Publications, 2017.

42 Alex Simpson and Niels F. W. Voorneveld. Behavioural equivalence via modalities for algebraic effects. *ACM Transactions on Programming Languages and Systems*, 42(1):4:1–4:45, 2020. doi:10.1145/3363518.

43 Ross Street. The formal theory of monads. *Journal of Pure and Applied Algebra*, 2(2):149–168, 1972. doi:10.1016/0022-4049(72)90019-9.

44 Tarmo Uustalu. Stateful runners of effectful computations. In Dan R. Ghica, editor, *The 31st Conference on Mathematical Foundations of Programming Semantics, MFPS 2015*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 403–421. Elsevier, 2015. doi:10.1016/J.ENTCS.2015.12.024.

45 Philip Wadler. The marriage of effects and monads. In Matthias Felleisen, Paul Hudak, and Christian Queinnec, editors, *3rd ACM SIGPLAN International Conference on Functional Programming, ICFP 1998*, pages 63–74. ACM, 1998. doi:10.1145/289423.289429.

46 Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Transactions on Computational Logic*, 4(1):1–32, 2003. doi:10.1145/601775.601776.

47 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. doi:10.1006/inco.1994.1093.

48 Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):51:1–51:32, 2020. doi:10.1145/3371119.

An Effectful Object Calculus

Francesco Dagnino
DIBRIS, Università di Genova, Italy

Paola Giannini
DiSSTE, Università del Piemonte Orientale, Italy

Elena Zucca
DIBRIS, Università di Genova, Italy

Abstract

We show how to smoothly incorporate in the object-oriented paradigm constructs to raise, compose, and handle effects in an arbitrary monad. The underlying pure calculus is meant to be a representative of the last generation of OO languages, and the effectful extension is manageable enough for ordinary programmers; notably, constructs to raise effects are just special methods. We equip the calculus with an expressive type-and-effect system, which, again by relying on standard features such as inheritance and generic types, allows a simple form of effect polymorphism. The soundness of the type-and-effect system is expressed and proved by a recently introduced technique, where the semantics is formalized by a one-step reduction relation from language expressions into monadic ones, so that it is enough to prove progress and subject reduction properties on this relation.

2012 ACM Subject Classification Theory of computation → Operational semantics; Theory of computation → Type structures

Keywords and phrases Object calculi, handlers, type-and-effect systems

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.8

Related Version *Full Version*: `https://doi.org/10.48550/arXiv.2504.15936` [11]

Funding This work was partially funded by the MUR project "T-LADIES" (PRIN 2020TL3X8X) and has the financial support of the Università del Piemonte Orientale.

1 Introduction

Every modern programming language needs to support a wide range of computational effects, such as exceptions, input/output, interaction with a memory and classical or probabilistic non-determinism. Several approaches have been proposed for designing language constructs dealing with different computational effects in a uniform and principled way. The monad pattern [31, 32, 46] and algebraic effects and handlers [34, 35, 36, 38, 39, 6, 41] stand out for their impact on real world programming, and are now incorporated into many mainstream languages such as Haskell, OCaml 5 and Scala. However, this area of research focuses on the functional paradigm. The aim of this paper, instead, is to show that constructs to raise, compose, and handle effects can be smoothly incorporated into object-oriented languages; in fact, the specific features of the paradigm help in allowing a simple effectful extension.

To illustrate the effectful extension, we choose a pure object calculus designed for the purpose, rather than Featherweight Java (FJ)[1] [21], for more than twenty years the paradigmatic calculus for Java-like languages. As FJ, our calculus only includes distinctive ingredients, such as nominal types, inheritance, and objects, and no imperative features. However, following the current trend in OO languages, and inspired by recent work [49, 43, 50], object values are, rather than class instances with fields, *object literals*, extending a sequence

[1] In its version with generic types.

© Francesco Dagnino, Paola Giannini, and Elena Zucca;
licensed under Creative Commons License CC-BY 4.0

of nominal types with additional methods. In other words, we select, as subset of Java features, interfaces with default methods and anonymous classes, rather than classes with fields and constructors. This choice leads to generality, in the sense that the language design encompasses both Java-like and functional calculi. More precisely, FJ can be easily encoded by representing fields as constant methods, and, at the same time, as in seminal object calculi [1], the simply-typed lambda calculus can be seen as a language subset, since functions are a special case of stateless objects.

The effectful extension builds on generic effects [36] and handlers [38], carefully adapted to be more familiar to OO programmers. Constructs for raising effects are just method calls of a special kind, called *magic*.[2] They are made available through predefined interfaces, and do not specify an implementation, like abstract ones. However, rather than being deferred to subtypes, their implementation is provided "by the system". For instance, throwing an exception is a magic call `Exception.throw()`. This fits naturally in the OO paradigm, where computations happen in the context of a program. Again to be familiar, the construct to handle effects is a generalized try-block with catch clauses. A raised effect (a magic call) is caught by a clause when the receiver's type is (a subtype of) that specified in the clause, as for Java exceptions, generalized to arbitrary magic calls. To this end, besides those interrupting the normal flow of execution, we allow catch clauses replacing an effect with an alternative behaviour in a continuous manner. Altogether, the construct provides a good compromise between simplicity and expressivity: notably, it does not explicitly handle continuations, as in handlers of algebraic effects [38, 39, 24, 19], yet expressing a variety of handling mechanisms.

We endow our calculus with a type-and-effect system where effect types[3] again generalize sets of exceptions, as in `throws` clauses, to sets of *call-effects*, of shape $T.\mathsf{m}[\overline{T}]$, approximating the computational effects of a call of m in type T, with types $\overline{T}$ instantiating the type variables of the method. This approach, inspired by [15], provides more expressivity than the effect systems for algebraic effects [5, 6, 41], which only track the *names* of operations raising effects, without any type information. Moreover, T can be a type variable X, expressing a call-effect not yet specified, which will become, when X is instantiated to a specific type, the effect type declared there for the method. This allows a rather sophisticated approximation of effects, encompassing a simple form of *effect polymorphism* [26, 20, 29, 7, 9, 8].

Last but not least, the soundness of the type-and-effect system is expressed and proved by a newly introduced approach [12], building on a recent line of research on *monadic operational semantics* for effectful languages [17, 18]. That is, the semantics is formalized by a one-step reduction from language expressions into monadic ones, parameterized by a monad modeling the capabilities provided by the system. In this way, we can uniformly deal with a wide range of effects, by just providing an interpretation of magic methods in the monad. Then, applying a technique from [12], to ensure type-and-effect soundness it is enough to prove progress and subject reduction on the monadic one-step reduction. Accordingly with its shape, subject reduction roughly means that reducing an expression to a monadic one preserves its type-and-effect; hence, to express and prove this property, we need *monadic typing judgments*, which can be derived by *lifting* the non-monadic ones.

We describe the language in Section 2, and its semantics in Section 3. The type-and-effect system is illustrated in Section 4, and its soundness is shown in Section 5. Section 6 discusses related work and Section 7 summarizes our contribution and outlines future work. Auxiliary definitions and omitted proofs can be found in the extended version [11].

---

[2] This terminology is taken from [50], and also informally used in Java reflection and Python.

[3] The term "effect" is used in literature both as synonym of computational effect, and in the context of type-and-effect systems, as a static approximation of the former. We will use "effect" when there is no ambiguity, otherwise "computational effect" and "effect type", respectively.

2 Language

To have a smoother presentation, we first provide the syntax, discuss the key features, and show some examples, for the effect-free subset of the language; then we illustrate the constructs to raise, compose, and handle computational effects.

Effect-free language. As anticipated, in our calculus objects are not created by invoking constructors, but directly obtained by extending a sequence of nominal types with additional methods. In this way, the notions of class (more in general, nominal type) and object almost coincide. The only difference is that nominal types are top-level entities in a program, possibly generic (declaring type variables), and abstract (including non-implemented methods). On the other hand, objects can be seen as anonymous types declared on-the-fly in the code.

Syntax and types are given in Figure 1. We assume *type names* N, *method names* m, *type variables* X, Y, and *variables* x, y. The metavariable $\overline{td}$ stands for a sequence $td_1 \dots td_n$, and analogously for other overlined metavariables. An empty sequence will be denoted by ϵ.

P	::=	$\overline{td}$	program
td	::=	$\mathsf{N}[\overline{Y} \lhd \overline{T}] \lhd \overline{N}\{\overline{md}\}$	(nominal) type declaration
md	::=	$\mathsf{m} : \mathsf{abs}\, MT \mid \mathsf{m} : \mathsf{def}\, MT\, \langle x\, \overline{x}, e\rangle$	method declaration
e	::=	$x \mid \overline{N}\{\overline{md}\} \mid e.\mathsf{m}[\overline{T}](\overline{e})$	expression: variable, object, method call
T, U	::=	$X \mid \overline{N}\{s\}$	type
N	::=	$\mathsf{N}[\overline{T}]$	nominal type
s	::=	$\overline{\mathsf{m}} : \overline{\mathsf{k}}\ \overline{MT}$	signature (structural type)
k	::=	$\mathsf{abs} \mid \mathsf{def}$	(method) kind
MT	::=	$[\overline{X} \lhd \overline{U}]\overline{T} \to T$	method type

Figure 1 Effect-free syntax and types.

A program P is a sequence of declarations of *(nominal) types*[4]. This sequence is assumed to be a map, that is, type names are distinct. Hence, we can safely use the notations $P(\mathsf{N})$ and $\mathsf{dom}(P)$, as we will do for other (sequences representing) maps.

A type declaration $\mathsf{N}[\overline{Y} \lhd \overline{T}] \lhd \overline{N}\{\overline{md}\}$ introduces a generic nominal type $\mathsf{N}[\overline{Y} \lhd \overline{T}]$, inheriting from all those in $\overline{N}$. We assume that the notation $\overline{Y} \lhd \overline{T}$ represents a map from type variables to types (their bounds), that is, type variables are distinct and the two sequences have the same length; moreover, $\overline{N}$ represents a set, that is, order and repetitions are immaterial, and $\overline{md}$ represents a map, that is, method names are distinct. Analogous assumptions hold in an object $\overline{N}\{\overline{md}\}$ and a method type $[\overline{X} \lhd \overline{U}]\overline{T} \to T$.

Method declarations can be either abstract (abs), that is, only specifying a method type, or defined (def), providing the (variables to be used as) parameters and the method body. The programmer can choose an arbitrary variable x for the first parameter in defined methods to denote the current object, rather than a fixed name as, e.g., this in Java.

In the calculus, we adopt a uniform syntax for abstract and defined methods, to simplify the technical treatment; a more realistic language would likely use a different concrete syntax, for instance declaring parameters together with their types.

▶ Remark 1. Lambda-expressions can be encoded in the calculus, similarly to what is done in Java; that is, the object $\{\mathsf{apply} : \mathsf{def}\ T \to U\ \langle _\, x, e\rangle\}$ can be abbreviated $\lambda x{:}T.e$. Note that, in this way, encoded functions are possibly recursive and higher-order.

[4] We adopt this terminology to avoid more connoted terms such as class, interface, trait.

Types are either type variables X or *object types*, of shape $\overline{N}\{s\}$. Nominal types are (instantiations of) those declared in the program. Object types are a form of intersection types, present also in Scala 3, denoting, without introducing a name, a type which is a subtype of all types in $\overline{N}$, additionally providing methods in the signature s. Nominal types can be seen as special object types. We assume that a signature s represents a map from method names to method kinds and types, that is, method names are distinct and the three sequences have the same length. Hence, we can safely use the notations $s(\mathsf{m})$ and $\mathsf{dom}(s)$.

Note that type annotations written by the programmer are allowed to be arbitrary types, rather than only nominal. In this respect, our calculus offers more generality than Java.

In the following, we will write $\overline{N}$ for $\overline{N}\{\}$; note that such a syntactic form can be seen both as an object and the corresponding object type. Moreover, we will write Object for $\{\}$, which, as a type, is the top of the subtyping relation.

Examples. In the code examples, we will use a number of other obvious conventions and abbreviations, such as omitting square brackets around an empty sequence in type/method declarations and method calls, and writing the pair of sequences $\overline{X} \triangleleft \overline{T}$ as a sequence of pairs, omitting bounds which are Object. Finally, in both code and formalism, we will often use the wildcard _ to indicate that some variable or meta-variable does not matter.

▶ **Example 2.** As an example of a program, we show the classic encoding of booleans and natural numbers in the object-oriented paradigm, through an inheritance hierarchy, using the visitor pattern for defining functions over them.

```
Bool {
  if: abs [X Y◁ThenElse[X]] Y -> X
  not: def -> Bool <b, b.if[Bool ThenElse[Bool]](
    ThenElse[Bool]{
      then: def -> Bool <_, False>
      else: def -> Bool <_, True>
    }
  )
}

True ◁ Bool { if: def [X Y◁ThenElse[X]] Y -> X <_ te, te.then()> }
False ◁ Bool { if: def [X Y◁ThenElse[X]] Y -> X <_ te, te.else()> }

ThenElse[X] {
  then: abs -> X
  else: abs -> X
}
```

Let us focus first on the method `if`, which takes a parameter of (a subtype of) `ThenElse[X]`, expected to provide two alternative results of type `X`. The method, abstract in `Bool`, is implemented in `True` and `False` by selecting the `then` and `else` alternative provided by the argument, respectively. As said above, we use the wildcard as variable for the current object, since it does not occur in the body.

The type `ThenElse[X]` declares methods `then` and `else` to be implemented in its subtypes. To illustrate the language features, we define `not` in `Bool` by invoking `if` on the current object with, as argument, an object which implements `ThenElse[Bool]`. Clearly `not` could declared abstract in `Bool` and defined in the obvious way in the two subtypes.

Turning now to naturals, a similar encoding through an inheritance hierarchy could be:

```
Nat {
  succ: def -> Nat <n, Succ{pred: def -> Nat <_, n>}
  match: abs [X] NatMatch[X] -> X
}
```

```
Zero ◁ Nat { match: def [X] NatMatch[X] -> X <_ nm, nm.zero()> }
Succ ◁ Nat {
  pred: abs -> Nat
  match: def [X] NatMatch[X] -> X <n nm, nm.succ(n.pred())>
}

NatMatch[X] { zero: abs -> X   succ: abs Nat -> X }
```

Since in the calculus data are uniformly implemented by stateless objects, numbers greater than 0 are encoded as objects which extend `Succ` by implementing the predecessor method; that is, if $\texttt{Nat}_n$ is the object encoding number n, then $n+1$ is encoded by $\texttt{Nat}_n\texttt{.succ()}$, which evaluates to $\texttt{Succ\{pred: def -> Nat <_, Nat}_n\texttt{>\}}$.

The type `NatMatch[X]` declares methods `zero` and `succ` to be implemented in subtypes, offering a programming schema for definitions given by arithmetic induction. For instance:

```
Even ◁ NatMatch[Bool] {
  zero: def -> Bool <_,True>
  succ: def Nat -> Bool <even n, n.match(even).not()>
}
```

A different encoding of natural numbers could be provided with no inheritance hierarchy:

```
Nat {
  match: def [X] NatMatch[X] -> X <_ nm, nm.zero()>
  succ: def -> Nat
    <n, Nat{match: def [X] NatMatch[X] -> X <_ nm, nm.succ(n)>}}
}
```

In this version, 0 is encoded by `Nat`, and, if $\texttt{Nat}_n$ is the object encoding number n, then $n+1$ is encoded by $\texttt{Nat.succ(Nat}_n\texttt{)}$, which evaluates to

$$\texttt{Nat\{ match: def [X] NatMatch[X] -> X <_ nm, nm.succ(Nat}_n\texttt{)>\} \}}$$

Effectful language. In Figure 2 we extend the language by adding constructs to raise, compose, and handle effects. As customary, we adopt a fine-grain syntax [27], where *values*

P	::=	$\overline{td}$	program
td	::=	$\mathsf{N}[\overline{Y} \triangleleft \overline{T}] \triangleleft \overline{N}\{\overline{md}\}$	type declaration
md	::=	$\mathsf{m} : \mathsf{abs}\, MT \mid \mathsf{m} : \mathsf{def}\, MT\, \langle x\, \overline{x}, e\rangle \mid \mathsf{m} : \mathsf{mgc}\, MT$	method declaration
v	::=	$x \mid \overline{N}\{\overline{md}\}$	value
e	::=	$v.\mathsf{m}[\overline{T}](\overline{v}) \mid \texttt{return}\ v \mid \texttt{do}\ x = e_1\texttt{;}\ e_2 \mid \texttt{try}\ e\ \texttt{with}\ h$	expression
h	::=	$\overline{c}, \langle x, e\rangle$	handler
c	::=	$N.\mathsf{m} : [\overline{X}]\langle x\, \overline{x}, e\rangle_\mu$	(catch) clause
μ	::=	$\mathsf{c} \mid \mathsf{s}$	mode
k	::=	$\mathsf{abs} \mid \mathsf{def} \mid \mathsf{mgc}$	(method) kind

Figure 2 Fine-grain syntax for the effectful language.

are effect-free, whereas expressions may raise effects, and are also called *computations*. Methods can be, besides abstract and defined, *magic* (abbreviated mgc). Magic methods are made available to the programmer in the type declarations composing the program, and do not specify an implementation, like abstract methods. However, rather than being deferred to subtypes, their implementation is provided "by the system". We add the standard operators `return` for embedding values into computations, and `do` for composing computations

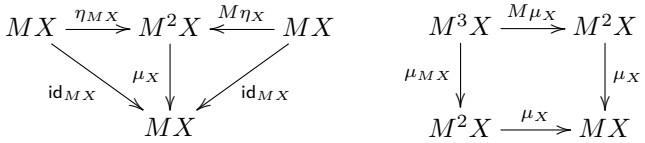
sequentially passing the result of the former to the latter. Finally, we provide a try-block enclosing a computation with a *handler*, consisting of a sequence of *(catch) clauses*, and a *final expression*, parametric on a variable. A clause specifies a type and method name, expected to identify a magic method declaration; the clause may catch calls of such method, by executing the *clause expression* instead, parametric on type variables and variables, analogously to a method body. After that, the final expression is either executed or not depending on the *mode*, either c or s, for "continue" and "stop", respectively, of the clause. As illustrated in the following examples, a c-clause replaces an effect with an alternative behaviour in a continuous manner, whereas in s-clauses handling the computational effect interrupts the normal flow of execution.

3 Monadic Operational Semantics

In this section, following the approach in [12], we define a *monadic operational semantics* for the language, parametric on an underlying monad.

Monads. First of all we recall basic notions about monads, referring to standard textbooks [42] for a detailed presentation.

A *monad* $\mathbb{M} = \langle M, \eta, \mu \rangle$ (on $\mathcal{Set}$) consists of a functor $M : \mathcal{Set} \to \mathcal{Set}$ and two natural transformations $\eta : \mathsf{Id} \Rightarrow M$ and $\mu : M^2 \Rightarrow M$ such that, for every set X, the following diagrams commute:



The functor M specifies, for every set X, a set MX of monadic elements built over X, in a way compatible with functions. The map η_X, named *unit*, embeds elements of X into monadic elements in MX, and the map μ_X, named *multiplication*, flattens monadic elements built on top of other monadic elements into plain monadic elements.

Functions of type $X \to MY$ are called *Kleisli functions* and play a special role: they can be seen as "effectful functions" from X to Y, raising effects described by the monad $\mathbb{M}$. Given a Kleisli function, by *Kleisli extension* we can define:

$$f^\dagger : MX \to MY \quad f^\dagger = \mu_Y \circ Mf$$

that is, first we lift f through M to apply it to monadic elements and then we flatten the result using μ_Y. Moreover, given $\alpha \in MX$, $f : X \to MY$ and $g : X \to Y$, we set

$$\begin{array}{ll} - \gg\!= - : MX \to (X \to MY) \to MY & \alpha \gg\!= f = f^\dagger(\alpha) \\ \mathsf{map} : (X \to Y) \to MX \to MY & \mathsf{map}\, g\, \alpha = Mg(\alpha) \end{array}$$

The operator $\gg\!=$ is also called bind. As its definition shows, it can be seen as an alternative description of the Kleisli extension, where the parameters are taken in inverse order. This view corresponds, intuitively, to the sequential composition of two expressions with effects, where the latter depends on a parameter *bound* to the result of the former. The operator map describes the effect of the functor M on functions. That is, the lifting of function g through M is applied to a monadic value α.

$$\text{(INVK)}\ \frac{}{v.\mathsf{m}[\overline{T}](\overline{v}) \to_p e[\overline{T}/\overline{X}][v/x][\overline{v}/\overline{x}]}\ \mathsf{lookup}(v,\mathsf{m}) = \langle \overline{X}, x, \overline{x}, e\rangle$$

$h = \overline{c}, \langle x, e'\rangle$

$$\text{(TRY-RET)}\ \frac{}{\texttt{try return } v \texttt{ with } h \to_p \texttt{do } x = \texttt{return } v\texttt{; } e'}$$

$$\text{(TRY-DO)}\ \frac{}{\texttt{try do } y = e_1\texttt{; } e_2 \texttt{ with } h \to_p \texttt{try } e_1 \texttt{ with } \overline{c}, \langle y, \texttt{try } e_2 \texttt{ with } h\rangle}$$

$$\text{(CATCH-CONTINUE)}\ \frac{}{\begin{array}{l}\texttt{try } v.\mathsf{m}[\overline{T}](\overline{v}) \texttt{ with } h \to_p \\ \qquad \texttt{do } x = e[\overline{T}/\overline{X}][v/x][\overline{v}/\overline{x}]\texttt{; } e'\end{array}}\ \begin{array}{l}\mathsf{lookup}(v,\mathsf{m}) = \langle \mathsf{mgc}, _\rangle \\ \mathsf{cmatch}(v.\mathsf{m}[\overline{T}](\overline{v}), \overline{c}) = [\overline{X}]\langle x\,\overline{x}, e\rangle_{\mathsf{c}}\end{array}$$

$$\text{(CATCH-STOP)}\ \frac{}{\texttt{try } v.\mathsf{m}[\overline{T}](\overline{v}) \texttt{ with } h \to_p e[\overline{T}/\overline{X}][v/x][\overline{v}/\overline{x}]}\ \begin{array}{l}\mathsf{lookup}(v,\mathsf{m}) = \langle \mathsf{mgc}, _\rangle \\ \mathsf{cmatch}(v.\mathsf{m}[\overline{T}](\overline{v}), \overline{c}) = [\overline{X}]\langle x\,\overline{x}, e\rangle_{\mathsf{s}}\end{array}$$

$$\text{(FWD)}\ \frac{}{\texttt{try } v.\mathsf{m}[\overline{T}](\overline{v}) \texttt{ with } h \to_p \texttt{do } x = v.\mathsf{m}[\overline{T}](\overline{v})\texttt{; } e'}\ \begin{array}{l}\mathsf{lookup}(v,\mathsf{m}) = \langle \mathsf{mgc}, _\rangle \\ \mathsf{cmatch}(v.\mathsf{m}[\overline{T}](\overline{v}), \overline{c}) \text{ undefined}\end{array}$$

$$\text{(TRY-CTX)}\ \frac{e_1 \to_p e_2}{\texttt{try } e_1 \texttt{ with } h \to_p \texttt{try } e_2 \texttt{ with } h}$$

Figure 3 Pure reduction.

Monadic reduction. Let Val and Exp be the sets of closed values and expressions, respectively. In the following, we define a monadic (one-step) reduction for the language, being a relation $\to$ on $\mathsf{Exp} \times M\mathsf{Exp}$, parametric on a monad $\mathbb{M} = \langle M, \eta, \mu\rangle$. More in detail, its definition depends on the following ingredients:

- The function $\eta_{\mathsf{Exp}} : \mathsf{Exp} \to M\mathsf{Exp}$ embedding language expressions into their counterpart in the monad, written simply η in the following.
- The function $\mathsf{map} : (\mathsf{Exp} \to \mathsf{Exp}) \to M\mathsf{Exp} \to M\mathsf{Exp}$ lifting functions from expressions to expressions to their counterpart in the monad.

Moreover we assume:

- For every magic method m declared in N, a partial function $\mathsf{run}_{\mathsf{N},\mathsf{m}} : \mathsf{Val} \times \mathsf{Val}^\star \rightharpoonup M\mathsf{Val}$, returning a monadic value expressing the effects raised by a call. The function could be undefined, for instance, when arguments do not have the expected types.

The monadic reduction is modularly defined on top of a "pure" reduction $\to_p$ on $\mathsf{Exp} \times \mathsf{Exp}$, which transforms calls of non-magic methods into the corresponding bodies, as usual, and try-blocks into either magic calls or `do` expressions, by distributing and possibly applying catch clauses; magic calls and `do` expressions are, then, normal forms for the pure reduction.

Rules defining the pure reduction are given in Figure 3. The abbreviation h for $\overline{c}, \langle x, e'\rangle$ is intended to be used in all the rules below. The lookup function models method look-up, searching for the method declaration corresponding to a call, with two different successful outcomes: either a defined method, and the result are its type variables, variables, and body, or a magic method, necessarily in a type declaration, and the result is an mgc tag, and the type name. The cmatch function is defined, on a pair consisting of a magic call and a sequence of clauses, when there is a (first) clause catching the call; in this case, the corresponding clause expression is returned. The formal definitions are given in Figure 4.

Rule (INVK) is the standard rule for method invocation. That is, method look-up finds a defined method, and the call reduces to the method's body where type variables and variables are replaced by actual types and arguments. The other rules model the behaviour of a computation enclosed in a try-block.

$$\mathsf{lookup}(\overline{N}\{\overline{md}\}, \mathsf{m}) = \begin{cases} [\overline{X}]\langle x\,\overline{x}, e\rangle & \text{if } \overline{md}(\mathsf{m}) = \mathsf{m} : \mathsf{def}\ [\overline{X} \triangleleft _]_ \to _\,\langle x\,\overline{x}, e\rangle \\ \mathsf{lookup}(\overline{N}, \mathsf{m}) & \text{otherwise} \end{cases}$$

$$\mathsf{lookup}(N_1 \dots N_n, \mathsf{m}) = \begin{cases} \mathsf{lookup}(N_i, \mathsf{m}) & \text{if } \mathsf{lookup}(N_i, \mathsf{m}) \text{ defined for a unique } i \in 1..n \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathsf{lookup}(\mathsf{N}[\overline{T}], \mathsf{m}) = \begin{cases} [\overline{X}]\langle x\,\overline{x}, e[\overline{T}/\overline{Y}]\rangle & \text{if } \overline{md}(\mathsf{m}) = \mathsf{m} : \mathsf{def}\ [\overline{X} \triangleleft _]_ \to _\,\langle x\,\overline{x}, e\rangle \\ \langle \mathsf{mgc}, \mathsf{N}\rangle & \text{if } \overline{md}(\mathsf{m}) = \mathsf{m} : \mathsf{mgc}\ [_ \triangleleft _]_ \to _ \\ \mathsf{lookup}(\overline{N}[\overline{T}/\overline{Y}], \mathsf{m}) & \text{otherwise} \end{cases}$$

$$\text{where } P(\mathsf{N}) = \mathsf{N}[\overline{Y} \triangleleft _] \triangleleft \overline{N}\{\overline{md}\}$$

$$\mathsf{cmatch}(v.\mathsf{m}[\overline{T}](\overline{v}), c) = \begin{cases} [\overline{X}]\langle x\,\overline{x}, e\rangle_\mu & \text{if } c = N.\mathsf{m} : [\overline{X}]\langle x\,\overline{x}, e\rangle_\mu \text{ and } v\ \mathsf{instof}\ N \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathsf{cmatch}(v.\mathsf{m}[\overline{T}](\overline{v}), c\,\overline{c}) = \begin{cases} \mathsf{cmatch}(v.\mathsf{m}[\overline{T}](\overline{v}), \overline{c}) & \text{if } \mathsf{cmatch}(v.\mathsf{m}[\overline{T}](\overline{v}), c) \text{ undefined} \\ \mathsf{cmatch}(v.\mathsf{m}[\overline{T}](\overline{v}), c) & \text{otherwise} \end{cases}$$

$$\mathsf{cmatch}(v.\mathsf{m}[\overline{T}](\overline{v}), \epsilon) = \text{undefined}$$

Figure 4 Method look-up and matching.

When the enclosed computation has no effects, rule (TRY-RET), the try-block is reduced to the sequential composition of this computation with the final expression. In case of a `do` composition of two computations, the `do` is eliminated by reducing it to a try-block enclosing the first computation, with as final expression another try-block enclosing the second one; clauses are propagated to both.

When the computation calls a magic method, the behaviour depends on whether a matching clause is found. If it is found, then the clause expression is executed, after replacing parameters by arguments, see rules (CATCH-CONTINUE) and (CATCH-STOP). In a `c`-clause, the final expression is then executed. If no matching clause is found, then in rule (FWD) the try-block is reduced to the sequential composition of the magic call with the final expression.

Finally, rule (TRY-CTX) is the standard contextual rule. Note that, as already mentioned, in the pure reduction there are no rules for magic calls and `do` expressions. Indeed, the pure reduction only handles expressions when they are enclosed in a try-block, so that effects which would be raised by magic calls can be possibly caught.

The lookup function is defined in Figure 4. When a method is invoked on an object receiver, it is first searched among the method declarations in the object itself (first clause). Objects cannot declare magic methods, since they are made available to the programmer through predefined interfaces, as illustrated by examples later. If not found, then look-up is propagated to the parent types. In this case, there should be exactly one parent type where method look-up is successful. In a nominal type, analogously, the method is first searched among those defined in the corresponding declaration. If a defined method is found, then the type parameters of the declaration are replaced by the corresponding type arguments in the nominal type. If a magic method is found, necessarily in a type declaration, then the result is an mgc tag, and the type name. If the method is not found, then look-up is propagated to the parents, again replacing type parameters with the corresponding type arguments.

The cmatch function is defined in Figure 4 as well. We write v instof N meaning that the dynamic type of v, which can be extracted from v by just erasing method bodies, defined in the obvious way, is a subtype of N. Note that the extraction of the dynamic type and the subtyping check, being part of the runtime semantics, are purely syntactic. Notably, the extracted type could violate constraints on conflicting methods and overriding (which will be

$$\text{(PURE)}\ \frac{e \to_p e'}{e \to \eta(e')} \qquad \text{(MGC)}\ \frac{}{v.\mathsf{m}[\overline{T}](\overline{v}) \to \mathsf{map}\,(\texttt{return}\ [\,])\,\mathsf{run}_{\mathsf{N},\mathsf{m}}(v,\overline{v})}\ \mathsf{lookup}(v,\mathsf{m}) = \langle \mathsf{mgc}, \mathsf{N}\rangle$$

$$\text{(RET)}\ \frac{}{\texttt{do}\ x = \texttt{return}\ v;\ e \to \eta(e[v/x])} \qquad \text{(DO)}\ \frac{e_1 \to \mathrm{E}}{\texttt{do}\ x = e_1;\ e_2 \to \mathsf{map}\,(\texttt{do}\ x = [\,];\ e_2)\,\mathrm{E}}$$

Figure 5 Monadic (one-step) reduction.

checked by the type system) or even be ill-formed. If a (first) matching clause is found, then such clause provides an alternative body with its type parameters and parameters, which are instantiated with the corresponding arguments in the magic call. This resembles very much what happens for the call of a defined method, except that look-up is performed in the catch clauses, following the syntactic order.

In Figure 5 we give the rules for the monadic reduction. An expression is reduced to a monadic expression either by propagating a pure step, embedding its result in the monad, as shown in rule (PURE), or by interpreting in the monad magic calls and `do` expressions, as shown in the following rules.

In rule (MGC), method look-up finds a magic method in a type declaration named N, and the call reduces to the corresponding monadic expression. In other words, the effect is actually raised. To this end, we apply the function of type $M\mathsf{Val} \to M\mathsf{Exp}$ obtained by lifting, through map, the context $\texttt{return}\ [\,]$ to the monadic value obtained from the call. Here we identify the context $\texttt{return}\ [\,]$, which is an expression with a hole, with the function $v \mapsto \texttt{return}\ [v]$ of type $\mathsf{Val} \to \mathsf{Exp}$.

Rules (RET) and (DO) are the monadic version of the standard ones for these constructs. More in detail, given a `do` expression, when the first subterm does not raise effects, just returning a value, the expression can be reduced to the monadic embedding of the second subterm, after replacing the variable with the returned value, as shown in rule (RET). Rule (DO), instead, propagates the reduction of the first subterm, taking into account possibly raised effects. To this end, we apply the function of type $M\mathsf{Exp} \to M\mathsf{Exp}$ obtained by lifting, through map, the context $\texttt{do}\ x = [\,];\ e_2$, to the monadic expression obtained from e_1. Analogously to above, we identify the context $\texttt{do}\ x = [\,];\ e_2$, which is an expression with a hole, with the function $e \mapsto \texttt{do}\ x = [e];\ e_2$ of type $\mathsf{Exp} \to \mathsf{Exp}$.

Monadic small-step reduction and semantics. Following the approach in [12], we can define, on top of the monadic one-step reduction $\to$ on $\mathsf{Exp} \times M\mathsf{Exp}$:

- a small-step reduction on *monadic configurations*
- a *finitary semantics* of expressions
- assuming an appropriate structure on the monad, an *infinitary semantics* of expressions.

To apply the construction in [12], the monadic one-step reduction is required to be deterministic, and this is the case indeed.

▶ **Proposition 3** (Determinism). *If $e \to E_1$ and $e \to E_2$ then $E_1 = E_2$.*

We outline the construction, referring to [12] for technical details and proofs.

First of all, we set $\mathsf{Conf} = \mathsf{Exp} + \mathsf{Res}$, where $\mathsf{Res} = \mathsf{Val} + \mathsf{Wr}$, $\mathsf{Wr} = \{\mathsf{wrong}\}$. That is, a *configuration* c is either an expression or a result r, which, in turn, is either a value, modelling successful termination, or wrong, modelling a stuck computation. Then, we extend the monadic reduction $\to$ to configurations, getting the relation $\xrightarrow[\mathsf{step}]{} \subseteq \mathsf{Conf} \times M\mathsf{Conf}$ shown in Figure 6. Expressions which represent terminated computations reduce to the monadic embedding of the corresponding value or wrong, respectively; moreover, results (either values or wrong) conventionally reduce to their monadic embedding.

$$(\text{EXP})\ \frac{e \to \mathrm{E}}{e \xrightarrow[\mathsf{step}]{} \mathrm{E}} \qquad (\text{RET})\ \frac{}{\texttt{return}\ v \xrightarrow[\mathsf{step}]{} \eta_{\mathsf{Conf}}(v)}$$

$$(\text{WRONG})\ \frac{e \not\to \quad e \neq \texttt{return}\ v \text{ for all } v \in \mathsf{Val}}{e \xrightarrow[\mathsf{step}]{} \eta_{\mathsf{Conf}}(\mathsf{wrong})} \qquad (\text{RES})\ \frac{}{r \xrightarrow[\mathsf{step}]{} \eta_{\mathsf{Conf}}(r)}$$

Figure 6 Monadic (one-step) reduction on configurations.

Now, we can define a relation $\Rightarrow$ on $M\mathsf{Conf}$ as follows:

$\mathrm{C} \Rightarrow \mathrm{C}'$ iff $\mathsf{step}^\dagger(\mathrm{C}) = \mathrm{C}'$.

In this way, computations on monadic configurations are described, as usual in small-step style, by sequences of $\Rightarrow$ steps.

Then, the finitary semantics is the function $[\![-]\!]_\star : \mathsf{Exp} \to M\mathsf{Res} + \{\infty\}$ defined as follows[5]:

$$[\![e]\!]_\star = \begin{cases} \mathrm{R} & \text{if } \eta_{\mathsf{Conf}}(e) \Rightarrow^\star \mathrm{R} \\ \infty & \text{otherwise} \end{cases}$$

where $\Rightarrow^\star$ is the reflexive and transitive closure of $\Rightarrow$. This semantics describes only monadic results that can be reached in finitely many steps. In other words, all diverging computations are identified and no information on computational effects they may produce is available.

To overcome this limitation, we introduce an *infinitary semantics*. As formally detailed in [12], we assume, for each set X, a partial order on MX with a least element $\bot_X$ and suprema $\bigsqcup$ of ω-chains. Let $\mathsf{res}_0 : \mathsf{Conf} \to M\mathsf{Res}$ be the function given by

$$\mathsf{res}_0(c) = \begin{cases} \bot_{\mathsf{Res}} & c = e \\ \eta_{\mathsf{Res}}(r) & c = r \end{cases}$$

For every $e \in \mathsf{Exp}$ and $n \in \mathbb{N}$, we define $[\![e]\!]_n = \mathsf{res}_0{}^\dagger(\mathrm{C})$ iff $\eta_{\mathsf{Conf}}(e) \Rightarrow^n \mathrm{C}$. Note that $[\![e]\!]_n$ is well-defined for all $n \in \mathbb{N}$ since $\Rightarrow$ is (the graph of) a total function and so is its n-th iteration. The sequence $([\![e]\!]_n)_{n\in\mathbb{N}}$ turns out to be an ω-chain, so we define the infinitary semantics as

$$[\![e]\!]_\infty = \bigsqcup_{n\in\mathbb{N}} [\![e]\!]_n$$

Intuitively, $[\![e]\!]_n$ is the portion of the result that is reached after n reduction steps. Hence, the actual result is obtained as the supremum of all such approximations, thus describing also the observable behaviour of possibly diverging computations.

Examples. We show now examples of type declarations providing the user interface to raise effects in an underlying monad, together with some reductions and semantics. Booleans, natural numbers, conditional, and checking that a number is even, are encoded as shown in Example 2, except that, for brevity, we assume that `not` in `True` directly returns `False`. We recall the (inductive) definition of $\texttt{Nat}_n$, the object encoding the natural number n:

$\texttt{Nat}_0 = \texttt{Zero}$ $\qquad$ $\texttt{Nat}_{n+1} = \texttt{Succ\{pred: def -> Nat <_, Nat}_n\texttt{>\}}$

Finally, a handler of shape $\overline{c}, \langle x, \texttt{return}\ x\rangle$ is abbreviated by $\overline{c}$.

[5] Here and in rule (EXP) we omit the injection into monadic configurations.

▶ **Example 4** (Exceptions)**.** Let us fix a set Exc. The monad $\mathbb{E}_{\mathsf{Exc}} = \langle E_{\mathsf{Exc}}, \eta^{\mathbb{E}_{\mathsf{Exc}}}, \mu^{\mathbb{E}_{\mathsf{Exc}}} \rangle$ is given by $E_{\mathsf{Exc}}X = X + \mathsf{Exc}$, and

$$\eta^{\mathbb{E}_{\mathsf{Exc}}}(x) = x \qquad \alpha \gg\!= f = \begin{cases} f(x) & \text{if } \alpha = x \in X \\ \alpha & \text{otherwise } (\alpha = \mathsf{e} \in \mathsf{Exc}) \end{cases}$$

where $+$ denotes disjoint union (coproduct) and, for simplicity, we omit the injections.

We define a nominal type `Exception` of the (objects representing) exceptions. More formally, we assume that, for each v such that v instof `Exception`, that is, the dynamic type of v is a subtype of `Exception`, there is an associated exception in Exc, denoted $\mathsf{exc}(v)$. As a minimal example, we define, besides `Exception`, a subtype `MyException`, and assume[6] $\mathsf{exc}(\texttt{Exception}) = \mathsf{E}$, and $\mathsf{exc}(\texttt{MyException}) = \mathsf{MyE}$, with E, MyE elements of Exc.

```
Exception { throw : mgc[X] -> X }

MyException◁Exception { throw : mgc[X] -> X }
```

Both types declare a magic method `throw` that can be called to raise the exception corresponding to the receiver object. Note that the method `throw` declares a type variable as result type, since it could be called in an arbitrary context. We have:

- $\mathsf{run}_{\texttt{Exception},\texttt{throw}} : \mathsf{Val} \times \mathsf{Val}^\star \rightharpoonup M\mathsf{Val}$ only defined on $\langle v, \epsilon \rangle$ with v instof `Exception`
  $\mathsf{run}_{\texttt{Exception},\texttt{throw}}(v, \epsilon) = \mathsf{exc}(v)$
- $\mathsf{run}_{\texttt{MyException},\texttt{throw}} : \mathsf{Val} \times \mathsf{Val}^\star \rightharpoonup M\mathsf{Val}$ only defined on $\langle v, \epsilon \rangle$ with v instof `MyException`
  $\mathsf{run}_{\texttt{MyException},\texttt{throw}}(v, \epsilon) = \mathsf{exc}(v)$

The `throw` magic method in `MyException` has the same behaviour of that in the parent type; however, redefining the method will be significant in the type-and-effect system (Section 4), to precisely track the possibly raised exceptions.

The type `My` below defines a method that, depending on whether its argument (a natural number) is either even or odd, returns a natural or calls the method `throw` of `MyException`.

```
My {
  m : def Nat -> Nat<x y, y.match(Even).if[Nat ThenElse[Nat]](MyTE)}>
}

MyTE = ThenElse[Nat]{
    then : def [Nat] ->Nat <_, return One>
    else : def [Nat] ->Nat <_, Exception.throw[Nat]()>
}

One = Nat₁ = Succ{pred : def -> Nat <_,Zero>}
```

In the following reductions, we use the abbreviations `MyTE` and `One` above and write `if` for `if`[`Nat ThenElse`[`Nat`]].

Consider the expression:

$$e_1 = \texttt{try}\,\texttt{My.m}(\texttt{One})\,\texttt{with}\,c_1 \text{ where } c_1 = \texttt{Exception.throw} : [X]\langle x, \texttt{return One}\rangle_{\mathsf{s}}$$

[6] In this simple example exc only depends on the exception type; generally, it could depend on a specific instance, for instance if exceptions provide error messages or other information.

We have the following small-step reduction sequences on monadic configurations[7]:

$$
\begin{array}{rcl}
e_1 & \Rightarrow & \texttt{try}\,\texttt{One.match(Even).if(MyTE)}\,\texttt{with}\,c_1 \\
& \Rightarrow & \texttt{try}\,\texttt{Even.succ(One.pred()).if(MyTE)}\,\texttt{with}\,c_1 \\
& \Rightarrow & \texttt{try}\,\texttt{One.pred().match(Even).not().if(MyTE)}\,\texttt{with}\,c_1 \\
& \Rightarrow & \texttt{try}\,\texttt{Zero.match(Even).not().if(MyTE)}\,\texttt{with}\,c_1 \\
& \Rightarrow & \texttt{try}\,\texttt{Even.zero().not().if(MyTE)}\,\texttt{with}\,c_1 \\
& \Rightarrow & \texttt{try}\,\texttt{True.not().if(MyTE)}\,\texttt{with}\,c_1 \\
& \Rightarrow & \texttt{try}\,\texttt{False.if(MyTE)}\,\texttt{with}\,c_1 \\
& \Rightarrow & \texttt{try}\,\texttt{Exception.throw[Nat]()}\,\texttt{with}\,c_1 \\
& \Rightarrow & \texttt{return One} \Rightarrow \texttt{One}
\end{array}
$$

where all steps are derived by rules (PURE) in Figure 5 and (EXP) in Figure 6, except the last one, which is derived by rule (RET) in Figure 6.

Considering now:

$$e_2 = \texttt{try}\,\texttt{My.m(One)}\,\texttt{with}\,c_2 \text{ where } c_2 = \texttt{MyException.throw} : [X]\langle x\,, \texttt{return One}\rangle$$

we have

$$
\begin{array}{rcl}
e_2 & \Rightarrow^\star & \texttt{try}\,\texttt{False.if(MyTE)}\,\texttt{with}\,c_2 \text{ first 8 steps as for } e_1 \\
& \Rightarrow & \texttt{do}\ x = \texttt{Exception.throw()}\texttt{; return Zero} \\
& \Rightarrow & \mathsf{E} \text{ since} \\
& & \texttt{do}\ x = \texttt{Exception.throw()}\texttt{; return Zero} \to \mathsf{E} \text{ by rule (DO)}
\end{array}
$$

As the reader may have noted, the s-clause models the expected behaviour of exceptions, which, even when caught, interrupt the normal flow of execution. In the following example we show a different interface, still using the exception monad, paired with c-clauses, which replace an effect with an alternative behaviour in a continuous manner.

▶ **Example 5** (Failure).[8] Consider the exception monad with $\mathsf{Fail} \in \mathsf{Exc}$, and the following type declarations, with $\mathsf{run}_{\texttt{Failure,fail}}$ returning Fail.

```
Failure[X]{ fail : mgc -> X }

String{ ...  toNat: def -> Nat < ... > }

Test {
 sumAsNat: def String String -> Nat
   <_ s1 s2,do n1=s1.toNat();do n2=s2.toNat();do n=n1.sum(n2);return n>
}
```

Method `toNat`, whose implementation is omitted, is expected to return the natural number represented by a string, if any; otherwise, Fail is raised, through the magic call `Failure[Nat].fail()`. We assume a method `sum` in `Nat` returning the sum of two numbers.

The expression `Test.sumAsNat(s1,s2)` clearly raises Fail if one of the two strings does not represent a natural number. However, we can catch such effect returning a default value:

```
try Test.sumAsNat(s1,s2) with Failure[Nat].fail : <_ , return Zero>c
```

[7] We omit the injections from monadic expressions and values.
[8] This example is a minimal version of one in [9].

In this way, we always get a result; in particular, if `s1` does not represent a natural number, and `s2` represents the natural number `n`, we get `n`, as expected. Instead, with a `s`-clause, we would get `Zero`, without performing the sum. An example of reduction sequence is below.[9]

$e = \texttt{try Test.sumAsNat("1","a") with } c \text{ where } c = \texttt{Failure[Nat].fail} : \langle _ , \texttt{return Zero} \rangle_{\texttt{c}}$

$$
\begin{array}{rcl}
e & \Rightarrow & \texttt{try do n1="1".toNat();do n2="a".toNat();do n=n1.sum(n2);ret n with } c \\
& \Rightarrow & \texttt{try "1".toNat() with } c, \langle \texttt{n1} , \texttt{try do n2="a".toNat();do n=n1.sum(n2);ret n with } c \rangle \\
& \Rightarrow^{\star} & \texttt{try ret One with } c, \langle \texttt{n1} , \texttt{try do n2="a".toNat();do n=n1.sum(n2);ret n with } c \rangle \\
& \Rightarrow & \texttt{do n1 = return One; try do n2 = "a".toNat(); return n1.sum(n2) with } c \\
& \Rightarrow & \texttt{try do n2="a".toNat();do n=One.sum(n2);ret n with } c \\
& \Rightarrow & \texttt{try "a".toNat() with } c, \langle \texttt{n2} , \texttt{try do n=One.sum(n2);ret n with } c \rangle \\
& \Rightarrow^{\star} & \texttt{try Failure[Nat].fail with } c, \langle \texttt{n2} , \texttt{try do n=One.sum(n2);ret n with } c \rangle (*) \\
& \Rightarrow & \texttt{do n2 = return Zero; try do n=One.sum(n2);ret n with } c \\
& \Rightarrow & \texttt{try do n=One.sum(Zero);ret n with } c \\
& \Rightarrow & \texttt{try One.sum(Zero) with } c, \langle \texttt{n} , \texttt{ret n} \rangle \\
& \Rightarrow^{\star} & \texttt{try return One with } c, \langle \texttt{n} , \texttt{ret n} \rangle \\
& \Rightarrow & \texttt{do n = return One; return n} \\
& \Rightarrow & \texttt{return One} \Rightarrow \texttt{One}
\end{array}
$$

If the catch `c`-clause were a `s`-clause, then the expression at line $(*)$ would reduce to `return Zero` and therefore the reduction would produce `Zero`.

It is worthwhile to compare the interfaces:

```
Exception { throw : mgc[X] -> X }

Failure[X]{ fail : mgc String -> X }
```

`Exception` offers a generic method `throw`, so that a magic call `Exception.throw()` can occur in any context. On the other hand, `Failure` is a parametric type, so that a magic call `Failure[T].fail()` can only occur where a `T` is expected.

▶ **Example 6** (Non-determinism). The monad $\mathbb{L} = \langle L, \eta^{\mathbb{L}}, \mu^{\mathbb{L}} \rangle$ is given by LX the set of (possibly infinite) lists over X, coinductively defined by the following rules: $\epsilon \in L(X)$ and, if $x \in X$ and $l \in L(X)$, then $x : l \in L(X)$. We use the notation $[x_1, \ldots, x_n]$ to denote the finite list $x_1 : \ldots : x_n : \epsilon$. Then, the unit is $\eta^{\mathbb{L}}_X(x) = [x]$, and the bind is corecursively defined as follows: $\epsilon \gg\!= f = \epsilon$ and $(x : l) \gg\!= f = f(x)(l \gg\!= f)$, where juxtaposition denotes the concatenation of possibly infinite lists. We define the nominal type `Chooser` declaring the magic method `choose`, with $\mathsf{run}_{\texttt{Chooser,choose}}$ returning the list (monadic value) consisting of the values `True` and `False` and we use the abbreviations `MyTE1` and $\texttt{MyTE}^y$ defined below.

```
Chooser { choose : mgc -> Bool }

My {
  m1 : def -> Nat  <_, do z = Chooser.choose();
                        z.if[Nat ThenElse[Nat]](MyTE1)>
  m2 : def Nat -> Nat <_ y, do z = Chooser.choose();
                              z.if[Nat ThenElse[Nat]](MyTE^y)>
}

MyTE1 = ThenElse[Nat]{
  then : def [Nat] -> Nat <_, return One>
  else : def [Nat] -> Nat <_, return Zero>
}
```

[9] We use `ret` for `return` to save space.

```
MyTE^y = ThenElse[Nat]{
  then : def [Nat] ->Nat <_, return y>
  else : def [Nat] ->Nat <_, My.m2(y.succ())>
}

Two = Nat_2 = Succ{ pred: def ->Nat <_,Succ{pred: def ->Nat <_,Zero>}> }
```

We have the following small-step reduction sequences:

$$
\begin{array}{lll}
[\,\texttt{My.m1()}\,] & \Rightarrow & [\,\texttt{do z=Chooser.choose();z.if(MyTE1)}\,] \\
& \Rightarrow & [\,\texttt{do z=ret True;z.if(MyTE1)}, \texttt{do z=ret False;z.if(MyTE1)}\,] \\
& \Rightarrow & [\,\texttt{True.if(MyTE1)}, \texttt{False.if(MyTE1)}\,] \\
& \Rightarrow & [\,\texttt{ret One}, \texttt{ret Zero}\,] \\
& \Rightarrow & [\,\texttt{One}, \texttt{Zero}\,] \\
[\,\texttt{My.m2(Zero)}\,] & \Rightarrow & [\,\texttt{do z=Chooser.choose();z.if(MyTE}^{\texttt{y}}\texttt{)}\,] \\
& \Rightarrow & [\,\texttt{do z=ret True;z.if(MyTE}^{\texttt{y}}\texttt{)}\,,\ \texttt{do z=ret False;z.if(MyTE}^{\texttt{y}}\texttt{)}\,] \\
& \Rightarrow & [\,\texttt{True.if(MyTE}^{\texttt{y}}\texttt{)}\,,\ \texttt{False.if(MyTE}^{\texttt{y}}\texttt{)}\,] \\
& \Rightarrow & [\,\texttt{ret Zero}\,,\ \texttt{My.m2(Zero.succ())}\,] \\
& \Rightarrow & [\,\texttt{Zero}\,,\ \texttt{My.m2(One)}\,] \\
& \Rightarrow & [\,\texttt{Zero}\,,\ \texttt{do z=Chooser.choose();z.if(MyTE}^{\texttt{y}}\texttt{)}\,] \\
& \Rightarrow & [\,\texttt{Zero}\,,\ \texttt{do z=ret True;z.if(MyTE}^{\texttt{y}}\texttt{)}\,,\ \texttt{do z=ret False;z.if(MyTE}^{\texttt{y}}\texttt{)}\,] \\
& \Rightarrow & [\,\texttt{Zero}\,,\ \texttt{True.if(MyTE}^{\texttt{y}}\texttt{)}\,,\ \texttt{False.if(MyTE}^{\texttt{y}}\texttt{)}\,] \\
& \Rightarrow & [\,\texttt{Zero}\,,\ \texttt{ret One}\,,\ \texttt{My.m2(One.succ())}\,] \\
& \Rightarrow & [\,\texttt{Zero}\,,\ \texttt{One}\,,\ \texttt{My.m2(Two)}\,] \\
& \Rightarrow^{\star} & [\,\texttt{Zero}\,,\ \texttt{One}\,,\ \texttt{Two}\,,\ \texttt{My.m2(Two.succ())}\,] \\
& \dots &
\end{array}
$$

Note that the second reduction is non-terminating, in the sense that a monadic result (a list of values) is never reached. Hence, with the finitary semantics, we get $[\![[\,\texttt{My.m2(Zero)}\,]]\!]_{\star} = \infty$. With the infinitary semantics, instead, we get the following ω-chain:

$$\texttt{[]}, \dots, \texttt{[Zero]}, \dots, \texttt{[Zero, One]}, \dots, \texttt{[Zero, One, Two]}, \dots, \texttt{[Nat}_0\texttt{, ..., Nat}_n\texttt{]}, \dots,$$

whose supremum is the infinite list of the (objects representing the) natural numbers.

▶ **Example 7.** Denote by DX the set of probability subdistributions α over X with countable support, i.e., $\alpha : X \to [0..1]$ with $\sum_{x\in X} \alpha(x) \leq 1$ and $\texttt{supp}(\alpha) = \{x \in X \mid \alpha(x) \neq 0\}$ countable set. We write $r \cdot \alpha$ for the pointwise multiplication of a subdistribution α with a number $r \in [0, 1]$. The monad $\mathbb{D} = \langle D, \eta^{\mathbb{D}}, \mu^{\mathbb{D}}\rangle$ is given by

$$\eta^{\mathbb{D}}(x) = y \mapsto \begin{cases} 1 & y = x \\ 0 & \text{otherwise} \end{cases} \qquad \alpha \gg\!= f = \textstyle\sum_{x\in X} \alpha(x) \cdot f(x)$$

We use the same nominal type `Chooser` with the magic method `choose`, now returning the distribution consisting of the values `True` and `False` with probability $\frac{1}{2}$, that we denote by $[\,\frac{1}{2} : \texttt{return True}, \frac{1}{2} : \texttt{return False}\,]$.

The expressions $[\,1 : \texttt{My.m1()}\,]$ and $[\,1 : \texttt{My.m2(Zero)}\,]$ can be reduced analogously to the previous example:

$[\,1 : \texttt{My.m1()}\,] \Rightarrow^{\star} [\,\frac{1}{2} : \texttt{One}, \frac{1}{2} : \texttt{Zero}\,]$
$[\,1 : \texttt{My.m2(Zero)}\,] \Rightarrow^{\star} [\,\frac{1}{2} : \texttt{Zero}, \frac{1}{4} : \texttt{One}, \frac{1}{8} : \texttt{Two}, \frac{1}{16} : \texttt{My.m2(Two.succ())}\,]\ \dots$

Again, the second reduction is non-terminating, hence, with the finitary semantics, we get $[\![[\,1 : \texttt{My.m2(Zero)}\,]]\!]_{\star} = \infty$. With the infinitary semantics we get the following ω-chain:

$$\texttt{[]}, \dots, [\tfrac{1}{2}\texttt{:Zero]}, \dots, [\tfrac{1}{2}\texttt{:Zero, }\tfrac{1}{4}\texttt{:One]}, \dots, [\tfrac{1}{2}\texttt{:Nat}_0\texttt{, ..., }\tfrac{1}{2^{n+1}}\texttt{:Nat}_n\texttt{]}, \dots,$$

whose supremum is the infinite list where each (object representing the) number n has probability $\frac{1}{2^{n+1}}$.

4 Type-and-effect System

In order to equip the language with a type-and-effect system, first of all we extend the syntax and the signatures, as shown in Figure 7. In method declarations, method types are replaced by *method type-and-effects*, where an *effect* component is added; we maintain the same meta-variable for simplicity. They are considered equal up-to α-renaming. That is, $[\overline{X} \triangleleft \overline{U}]\overline{T} \to T!E = ([\overline{Y} \triangleleft \overline{U'}]\overline{T'} \to T'!E')[\overline{X}/\overline{Y}]$.

MT	::=	$[\overline{X} \triangleleft \overline{U}]\overline{T} \to T!E$	method type-and-effect
md	::=	$\mathsf{m} : \mathsf{def}\ MT\ \langle x\,\overline{x}, e\rangle \mid \mathsf{m} : \mathsf{abs}\ MT \mid \mathsf{m} : \mathsf{mgc}\ MT$	method declaration
E	::=	$\bullet \mid \top \mid E \vee E' \mid T.\mathsf{m}[\overline{T}]$	effect
s	::=	$\overline{\mathsf{m}} : \overline{\mathsf{k}}\ \overline{MT}$	signature
k	::=	$\mathsf{abs} \mid \mathsf{def} \mid \mathsf{mgc}$	(method) kind

Figure 7 Adding effects.

Effects are the empty effect, the top effect, union of effects, and *call-effects*. For magic methods, this component is assumed to have a canonical form, hence can be omitted in the concrete syntax. Notably, for a method $\mathsf{m} : \mathsf{mgc}\,[\overline{X} \triangleleft _]_ \to _$ in the declaration of the nominal type $\mathsf{N}[\overline{Y} \triangleleft _]$, the effect is $\mathsf{N}[\overline{Y}].\mathsf{m}[\overline{X}]$. In signatures, the information associated to method names is analogously extended; moreover, the additional kind mgc is considered.

Φ	::=	$\overline{X} \leq \overline{T}$	type environment
Γ	::=	$\overline{x} : \overline{T}$	environment

Figure 8 Syntax of (type) environments.

Type environments and environments, defined in Figure 8, are assumed to be maps, from type variables to types (their bounds), and from variables to types, respectively. Hence (type) variables are distinct, and the two sequences have the same length.

Before the formal details, we illustrate the most distinctive feature of our type system.

Variable call-effects and simplification. As shown in Figure 7, except for $\top$, effects are essentially (representations of) sets of call-effects, with $\bullet$ the empty set and $\vee$ the union. Sets of "atomic" effects are a rather natural idea, generalizing what happens, e.g., in Java `throws` clauses. What is interesting here is the nature of such atomic effects, and the role of $\top$. A call-effect $T.\mathsf{m}[\overline{T}]$ is a static approximation of the computational effects of a call to m with receiver of type T and type arguments $\overline{T}$. More precisely, we only allow call-effects which are:

magic $T.\mathsf{m}[\overline{T}]$ with T object type (that is, not of shape X), and m magic in T. As expected, this means that this magic method could be possibly invoked, raising the corresponding computational effect; for instance, as will be shown in Example 8, a call-effect `Exception.throw` denotes that an exception could be possibly thrown.

variable $X.\mathsf{m}[\overline{T}]$. This call-effect can be assigned to code parametric on the type variable X; the meaning is that, for each instantiation of X with an object type T, this becomes, through a non-trivial process called *simplification*, the effect of m in T. In other words, $X.\mathsf{m}[\overline{T}]$ is a parametric effect, which can be made concrete in the types which replace X.

In addition to sets of call-effects, we include the *top* effect, which plays the role of default for (typically abstract) methods which do not pose constraints on the effects in implementations. In a sense, this generalizes the meaning of a `throws Exception` clause in Java.

We illustrate now the above features on an example, notably showing how (simplification of) variable call-effects allows a very precise approximation.

▶ **Example 8.** Consider again the encoding of booleans in Example 2 (Section 2), where we added effect annotations.

```
Bool { if : abs [X Y◁ThenElse[X]] Y -> X ! Y.then ∨ Y.else }

True ◁ Bool {
  if : def [X Y◁ThenElse[X]] Y -> X ! Y.then <_ te, return te.then()>
}

False ◁ Bool {
  if : def [X Y◁ThenElse[X]] Y -> X ! Y.else <_ te, return te.else()>
}

ThenElse[X] {
  then: abs -> X ! ⊤
  else: abs -> X ! ⊤
}
```

The abstract method if in `Bool` has, as parameter type, the type variable `Y`, expected to be instantiated with subtypes of `ThenElse[X]`, hence providing methods `then` and `else`. As effect, the method declares the union of two variable call-effects, `Y.then` ∨ `Y.else`. This means that the computational effects of this method can only be those propagated from calling either `then` or `else` on the argument. Subtypes `True` and `False` specializes the effect of `if` as expected, since they select the `then` and the `else` alternative provided by the argument, respectively.

In the type `ThenElse[X]` the methods `then` and `else` are abstract, and their implementation in subtypes is allowed to raise arbitrary effects, as denoted by ⊤. This makes sense, since we would like to instantiate `ThenElse[X]` on types implementing these methods in arbitrary ways.

For instance, consider the following code, where `b` is an expression of type `Bool`, and `Exception`, `MyTE` are the type and object introduced in Example 4, where the latter has been annotated with effects, and `MyTEType` is the corresponding object type.

```
b.if[Nat, MyTEType](MyTE)

MyTE = ThenElse[Nat]{
  then: def  -> Nat ! • <_, return Zero>
  else: def  -> Nat ! MyException.throw[Nat] <_, MyException.throw()>
}

MyTEType =
ThenElse[Nat]{
  then: def -> Nat ! •,
  else: def -> Nat ! MyException.throw[Nat]
}
```

The argument passed to `if` is an object implementing `then` by returning `Zero`, hence declaring no effects, whereas `else` calls a magic method, and declares the corresponding effect.

The effect computed for the call is, as intuitively expected, `MyException.throw[Nat]`. This happens thanks to effect simplification, which will be formally specified in Figure 10. Indeed, looking for method `if` in the receiver's type `Bool` gives the effect `X.then` ∨ `X.else`, which is instantiated to the argument type, giving `MyTEType.then` ∨ `MyTEType.else`. This effect would be highly inaccurate. However, `MyTEType.then` and `MyTEType.else` are simplified to the effects of the corresponding methods, that is, • and `MyException.throw[Nat]`, respectively.

In conclusion, our type-and-effect system supports a form of effect polymorphism which does not need additional ingredients, such as, e.g., explicit effect variables. Indeed, the effect of a method can be parametric on those of methods called on type variables in the context (variable call-effects): for instance, the effect of `if` is parametric on `Y.then` and `Y.else`. This

only relies on existing language features, notably on the OO paradigm. The key point is that a method is identified by, besides its name, the type where it is declared. Hence, to be parametric on the effect of a method, it is enough to be parametric on the type where it is declared, and this is for free since we have type variables. In other words, whereas effect variables stand for arbitrary effects, variable call-effects stand for effects declared by a method, and standard instantiation is complemented here by a non-trivial step of simplification.

Key formal definitions. The typing rules for values and expressions are given in Figure 9. The typing judgment for values has the shape $\Phi;\Gamma \vdash v : T$, since values have no effects; the one for expressions, instead, has the shape $\Phi;\Gamma \vdash e : T!E$.

$$\text{(T-VAR)}\ \frac{}{\Phi;\Gamma \vdash x : T}\ \Gamma(x) = T \qquad \text{(T-OBJ)}\ \frac{\Phi;\Gamma;\overline{N}\{s\} \vdash \overline{md}\diamond}{\Phi;\Gamma \vdash \overline{N}\{\overline{md}\} : \overline{N}\{s\}}\ \begin{array}{l}\Phi \vdash \overline{md} \rightsquigarrow s\\ \Phi \vdash \overline{N}\{s\} \rightsquigarrow s'\\ \mathsf{NoMgc}(s)\\ \mathsf{NoAbs}(s')\end{array}$$

$$\text{(T-INVK)}\ \frac{\Phi;\Gamma \vdash v_0 : T_0 \qquad \Phi;\Gamma \vdash v_i : T_i'\ \forall i \in 1..n}{\Phi;\Gamma \vdash v_0.\mathsf{m}[\overline{T}](v_1,\ldots,v_n) : T[\overline{T}/\overline{X}]!E'}\ \begin{array}{l}\mathsf{mtype}_\Phi(T_0,\mathsf{m})=_\ [\overline{X} \lhd \overline{U}]\,T_1 \ldots T_n \to T!E\\ \Phi \vdash \overline{T} \le \overline{U}[\overline{T}/\overline{X}]\\ \Phi \vdash T_i' \le T_i[\overline{T}/\overline{X}]\ \forall i \in 1..n\\ \Phi \vdash E[\overline{T}/\overline{X}] \Downarrow E'\end{array}$$

$$\text{(T-RET)}\ \frac{\Phi;\Gamma \vdash v : T}{\Phi;\Gamma \vdash \texttt{return}\ v : T!\bullet} \qquad \text{(T-DO)}\ \frac{\Phi;\Gamma \vdash e : T!E \qquad \Phi;\Gamma, x : T \vdash e' : T'!E'}{\Phi;\Gamma \vdash \texttt{do}\ x = e;\ e' : T'!E \vee E'}$$

$$\text{(T-TRY)}\ \frac{\Phi;\Gamma \vdash e : T!E \qquad \Phi;\Gamma;T \vdash h : T'!H}{\Phi;\Gamma \vdash \texttt{try}\ e\ \texttt{with}\ h : T'!\mathcal{F}_H(E)}$$

$$\text{(T-HANDLER)}\ \frac{\Phi;\Gamma, x : T \vdash e' : T'!E' \qquad \Phi;\Gamma;T'' \vdash c_i \Rightarrow C_i\ \forall i \in 1..n}{\Phi;\Gamma;T \vdash c_1 \ldots c_n, \langle x, e'\rangle : T''!C_1 \ldots C_n, E'}\ \Phi \vdash T' \le T''$$

$$\text{(T-CONTINUE)}\ \frac{\Phi;\Gamma, x : N_x, \overline{x}{:}\overline{T} \vdash e : T''!E}{\Phi;\Gamma;\ T' \vdash N_x.\mathsf{m} : [\overline{X}]\langle x\,\overline{x}, e\rangle_{\mathsf{c}} \Rightarrow T_x.\mathsf{m} : [\overline{X}]\langle E\rangle}\ \begin{array}{l}\mathsf{mtype}_\Phi(N_x,\mathsf{m})=\mathsf{mgc}\,[\overline{X} \lhd \overline{U}]\overline{T} \to T!_\\ \Phi \vdash T'' \le T\end{array}$$

$$\text{(T-STOP)}\ \frac{\Phi;\Gamma, x : N_x, \overline{x}{:}\overline{T} \vdash e : T''!E}{\Phi;\Gamma;\ T' \vdash N_x.\mathsf{m} : [\overline{X}]\langle x\,\overline{x}, e\rangle_{\mathsf{s}} \Rightarrow T_x.\mathsf{m} : [\overline{X}]\langle E\rangle}\ \begin{array}{l}\mathsf{mtype}_\Phi(N_x,\mathsf{m})=\mathsf{mgc}\,[\overline{X} \lhd \overline{U}]\overline{T} \to _!_\\ \Phi \vdash T'' \le T'\end{array}$$

Figure 9 Typing rules for values and expressions.

They rely on the following auxiliary notations:

1. $\Phi \vdash E \Downarrow E'$ meaning that the effect E can be simplified to E'
2. $\mathcal{F}_h$ the *filter function* associated to the handler h
3. $\Phi \vdash T \rightsquigarrow s$ meaning that we can safely extract a signature s from a type; the notation $\mathsf{mtype}_\Phi(T,\mathsf{m})=\mathsf{k}\,MT$ is an abbreviation for $\Phi \vdash T \rightsquigarrow s$ and $s(\mathsf{m}) = \mathsf{k}\,MT$
4. $\Phi \vdash T \le T'$ the subtyping relation

We illustrate the typing rules and notations (1) and (2), whereas the formal definitions of (3) and (4), which are almost standard, are given in the extended version [11]. The reader should only know that (3) models extracting the structural type information; in this phase, constraints about no conflicting method definitions and safe overriding are checked. Then, typing rules model another phase where code (method bodies) is typechecked against this type information.[10]

[10] Differently from other type systems for Java-like languages [21], here the two phases cannot be sequenced, since we still need signature extraction for the types introduced "on the fly" by objects.

$$(\text{EMPTY})\ \frac{}{\Phi \vdash \bullet \Downarrow \bullet} \qquad (\text{TOP})\ \frac{}{\Phi \vdash \top \Downarrow \top} \qquad (\text{VAR})\ \frac{}{\Phi \vdash X.\mathsf{m}[\overline{T}] \Downarrow X.\mathsf{m}[\overline{T}]}\ X \in \mathsf{dom}(\Phi)$$

$$(\text{MGC})\ \frac{}{\Phi \vdash T.\mathsf{m}[\overline{T}] \Downarrow T.\mathsf{m}[\overline{T}]}\ \mathsf{mtype}_\Phi(T, \mathsf{m}) = \mathsf{mgc}\,[\overline{X} \triangleleft _]_ \to _!E$$

$$(\text{SIMPLIFY-NON-MGC})\ \frac{\Phi \vdash E[\overline{T}/\overline{X}] \Downarrow E' \quad \mathsf{mtype}_\Phi(T, \mathsf{m}) = \mathsf{k}\,[\overline{X} \triangleleft _]_ \to _!E \quad \mathsf{k} \neq \mathsf{mgc}}{\Phi \vdash T.\mathsf{m}[\overline{T}] \Downarrow E'}$$

$$(\text{SIMPLIFY-UNION})\ \frac{\Phi \vdash E_i \Downarrow E'_i \ \ i \in 1..2}{\Phi \vdash E_1 \vee E_2 \Downarrow E'_1 \vee E'_2}$$

Figure 10 Simplification of effects.

Rule (T-VAR) is straightforward. An object is well-typed, rule (T-OBJ), if a signature can be safely extracted from its type. This essentially means that there are no conflicts among the parent types, and they are safely overriden by the method declarations in the object (first and second side conditions). Moreover, an object cannot declare magic methods (third side condition), and should provide an implementation for all its (either inherited or declared) methods (last side condition). Finally, in the premise, (bodies of) declared methods should be well-typed with respect to an enclosing type which is the object type.

In rule (T-INVK), the type-and-effect of the invoked method is found in the (signature extracted from) the receiver's type, as expressed by the first side condition. In the second side condition, the type annotations in the call should (recursively) satisfy the bounds for the corresponding type variables, and, in the third side condition, the argument types should be subtypes of the corresponding parameter types where type variables have been replaced by the corresponding type annotations. The type assigned to the call is the return type of the method, with the same replacement. The method effect is instantiated by replacing the type variables with the type annotations. However, the effect assigned to the call is obtained by a further *simplification step* (last side condition).

The formal definition of simplification is given in Figure 10, where $\Phi \vdash E \Downarrow E'$ means that the (not necessarily simplified) effect E is simplified to E', which only contains call-effects which are either magic or variable, as described before.

We assume that effect annotations are written by the programmer (or preliminarily reduced) in a simplified shape. Non-simplified effects can, however, appear during typechecking, when a method declared with an effect E which is generic, that is, depending on the method's type variables $\overline{X}$, is invoked with actual type arguments[11] Indeed, in this case, instantiating the type variables, in some variable call-effect $X.\mathsf{m}[\overline{T}]$ in E, X can become an object type providing a (non-magic) method m, allowing to simplify to the corresponding effect of m in the object type. This allows us to give a more refined approximations of the effects raised at runtime, as illustrated by Example 8 above, where the non-simplified effect `MyTEType.then` $\vee$ `MyTEType.else` is simplified to $\bullet \vee$ `MyException.throw[Nat]`. To ensure decidability of typechecking, an issue we do not deal with in this paper, termination of effect simplification should be enforced by some standard technique, essentially by forbidding (mutual) recursion in effect annotations.

[11] This happens in rule (T-INVK) in Figure 9.

$$
\begin{array}{lcll}
h & ::= & \overline{c}, \langle x, e\rangle & \text{handler} \\
H & ::= & \overline{C}, E & \text{filter} \\
c & ::= & N.\mathsf{m} : [\overline{X}]\langle x\,\overline{x}, e\rangle_{\mu} & \text{catch clause} \\
C & ::= & N.\mathsf{m} : [\overline{X}]\langle E\rangle & \text{clause filter}
\end{array}
$$

$\mathcal{F}_H$ defined by:
$\mathcal{F}_H(E) = \mathcal{F}_{\overline{C}}(E) \vee E'$ if $H = \overline{C}, E'$

$\mathcal{F}_{\overline{C}}$ defined by:

$$\mathcal{F}_{\overline{C}}(\bullet) = \bullet \qquad \mathcal{F}_{\overline{C}}(\top) = \top \qquad \mathcal{F}_{\overline{C}}(E_1 \vee E_2) = \mathcal{F}_{\overline{C}}(E_1) \vee \mathcal{F}_{\overline{C}}(E_2)$$

$$E = T.\mathsf{m}[\overline{T}] \qquad \mathcal{F}_{C\overline{C}}(E) = \begin{cases} \mathcal{F}_C(E) & \text{if } \mathcal{F}_C(E) \text{ defined} \\ \mathcal{F}_{\overline{C}}(E) & \text{otherwise} \end{cases} \qquad \mathcal{F}_{\epsilon}(E) = E$$

$$C = N.\mathsf{m} : [\overline{X}]\langle E\rangle \qquad \mathcal{F}_C(T.\mathsf{m}[\overline{T}]) = \begin{cases} E[\overline{T}/\overline{X}] & \text{if } \Phi \vdash T \leq N \\ \text{undefined} & \text{otherwise} \end{cases}$$

Figure 11 Filters.

In rule (T-RET), a computation which is the embedding of a value has no effects, and, in (T-DO), a sequential composition of two computations has the union of the two effects.

Typing rules for try-blocks rely on *filter functions* associated to handlers, which describe how they transform effects, by essentially replacing calls of magic methods matching some clause with the effect of the clause expression, as formally defined in Figure 11. Filter functions allow to typecheck try-blocks in a very precise way, since effects of the expression in a clause are only added if the clause could be possibly applied. This generalizes to arbitrary effects what happens for Java catch clauses; however, in Java this is part of the analysis of unreachable code, whereas here it is a feature of the type system. Filters are not a novelty of this type-and-effect system, since they were firstly used in [12] for a functional calculus; however, it is worthwhile to describe them in detail since they are a new feature, and the general idea is applied here to rather different effects, leading to another formal definition.

As shown in Figure 11, filters are the type information which can be extracted from a handler, consisting of a sequence of clause filters and a final effect. The filter function $\mathcal{F}_H$ associated to H transforms an effect by first applying the sequence of clause filters, and then adding the final effect. The transformation applying a sequence of clause filters is defined inductively. The significant case is a call-effect, which is either transformed by a (first) matching clause filter, or remains unaffected. A clause filter matches a call-effect with the same method name and a subtype of the nominal type; the call-effect is replaced by the effect of the clause filter, where type variables have been substituted by the types in the call-effect.

In rule (T-TRY), in order to typecheck a try-block, first we get the type and effect of the enclosed expression. This type is then used to typecheck the handler, as type of the parameter of the final expression, see rule (T-HANDLER). By typechecking the handler we get a type, being a supertype of the final expression, which will be the type of the whole expression. Moreover, we extract from the handler a filter, which is used to transform the effect of the enclosed expression, getting the resulting effect of the whole expression.

In rule (T-HANDLER), as said above, the type on the left of the judgment is used as type of the parameter of the final expression, required to be a subtype of that of the handler. This latter type is also needed to typecheck s-clauses, see below. The filter extracted from the handler consists in a clause filter for each clause, and the effect of the final expression.

The filter extracted from a clause keeps the first three components (nominal type, method name, and type variables), and adds the effect obtained typechecking the clause expression, as shown in rules (T-CONTINUE) and (T-STOP). A c-clause is meant to provide alternative code

$$\text{(T-METHS)}\ \frac{\Phi;\Gamma;\ T \vdash md_i \diamond\ \forall i \in 1..n}{\Phi;\Gamma;\ T \vdash md_1 \dots md_n \diamond}$$

$$\text{(T-METH)}\ \frac{\Phi, \overline{X} \lhd \overline{U};\Gamma, x : T_x, \overline{x}{:}\overline{T} \vdash e : T'!E'}{\Phi;\Gamma;\ T_x \vdash \mathsf{m} : \mathsf{def}\ [\overline{X} \lhd \overline{U}]\overline{T} \to T!E\ \langle x\,\overline{x}, e\rangle \diamond}\ \Phi, \overline{X} \lhd \overline{U} \vdash T'!E' \leq T!E$$

$$\text{(T-PROG)}\ \frac{\vdash td_i \diamond\ \forall i \in 1..n}{\vdash td_1 \dots td_n \diamond}$$

$$\text{(T-NTYPE)}\ \frac{\overline{Y} \lhd \overline{T};\emptyset;\mathsf{N}[\overline{Y}] \vdash \mathsf{GetDef}(\overline{md}) \diamond}{\vdash \mathsf{N}[\overline{Y} \lhd \overline{T}] \lhd \overline{N}\{\overline{md}\} \diamond}\ \vdash \mathsf{N}[\overline{Y} \lhd \overline{T}] \lhd \overline{N}\{\overline{md}\} \rightsquigarrow _$$

Figure 12 Typing rules for method and type declarations.

to be executed before the final expression, hence the type of the clause expression should be (a subtype of) the return type of the operation. In a s-clause, instead, the result of the clause expression becomes that of the whole expression with handler, hence the type of the former should be (a subtype of) the latter.

Referring to Example 4, note that catching an exception with a c-clause would be ill-typed. Indeed, the type of the clause expression should be (a subtype of) the return type of `throw`, which is a type variable X. Since no value has type X, no value could be returned[12], as already noted in [36].

In Figure 12 we show the typing rules for method and type declarations, which are mainly straightforward. The typing judgment for method definitions has shape $\Phi;\Gamma;\ T \vdash \overline{md}\diamond$. In rule (T-METH), a method definition is well-typed if the body is well-typed with respect to a type environment enriched by the type variables with their bounds, and an environment enriched by the variable denoting the current object with the enclosing type, and parameters with the corresponding parameter types. The type-and-effect of the body should be a sub-type-and-effect of that declared for the method. The typing judgment for type declarations has shape $\vdash \overline{td}\diamond$. In rule (T-NTYPE), a type declaration, assumed to have passed the extraction phase (side condition), is well-typed if its defined method declarations, denoted by $\mathsf{GetDef}(\overline{md})$, are well-typed with respect to the type environment consisting of the type variables with their bounds, the empty environment, and the declared type as enclosing type. Here the environment is empty since these method declarations are top-level, whereas in those inside objects, handled in rule (T-OBJ), there can be variables declared at an outer level.

5 Type-and-effect Soundness

In this section, we express and prove soundness of our type-and-effect system, by applying definitions and results in [12]. We focus on explaining the concepts, referring to [12] for detailed formal definitions and proofs.

Informal introduction. In Section 3 we defined, on top of the monadic one-step reduction:

- a finitary semantics $[\![-]\!]_\star : \mathsf{Exp} \to M\mathsf{Res} + \{\infty\}$
- an infinitary semantics $[\![-]\!]_\infty : \mathsf{Exp} \to M\mathsf{Res}$

In standard soundness we expect the result, if any, to be in agreement with the expression type. Here, since the expression has also an effect, approximating the computational effects raised by its execution, we expect the monadic result, if any, to be in agreement with the expression type and effect.

[12] Hence, the clause expression could only be another `throw` or a diverging expression.

Let us write $\vdash e : T!E$ for $\emptyset;\emptyset \vdash e : T!E$, and analogously for $\vdash v : T$, and other ground judgments. Moreover, $\vdash r : T$ only holds if $r = v$ and $\vdash v : T$ holds. Note that the result wrong is never well-typed, that is, $\vdash \mathsf{wrong} : T$ does not hold for any type T.

To formally express the above soundness requirement, we need analogous typing judgments[13] $\vdash_E \mathrm{R} : T$, one for each type and effect, on monadic results. Assuming to have such judgments, we can express the soundness results as follows.

▶ **Theorem 9** (Finitary type-and-effect soundness). *$\vdash e : T!E$ and $[\![e]\!]_\star = r$ imply $\vdash_E r : T$.*

▶ **Theorem 10** (Infinitary type-and-effect soundness). *$\vdash e : T!E$ implies $\vdash_E [\![e]\!]_\infty : T$.*

Note that finitary soundness is vacuous when the finitary semantics of an expression is ∞. On the other hand, the infinitary semantics is always a monadic result.

Before explaining how the monadic typing judgments can be derived from the non-monadic ones, we show some examples.

▶ **Example 11.** Consider the exception monad as in Example 4. We have $E\mathsf{Res} = \mathsf{Res} + \mathsf{Exc}$, that is, monadic results are either values, or wrong, or exceptions. We expect the typing judgment $\vdash_E \mathrm{R} : T$ to be defined as follows:

$$\text{(T-VAL)}\ \frac{\vdash v : T}{\vdash_E v : T} \qquad \text{(T-EXC)}\ \frac{}{\vdash_E \mathsf{e} : T}\ \mathsf{e} \in [\![E]\!]$$

That is, a monadic result is well-typed with a given type-and-effect if it is either a well-typed value with the type, or an exception belonging to (the set denoted by) the effect. Note that wrong is ill-typed. The set of exceptions $[\![E]\!]$ represented by E is defined by:

$$[\![\bullet]\!] = \emptyset \qquad [\![\top]\!] = \mathsf{Exc} \qquad [\![E \vee E']\!] = [\![E]\!] \cup [\![E']\!]$$
$$[\![\mathsf{N}.\texttt{throw}]\!] = \{\mathsf{e} \mid \mathsf{e} = \mathsf{exc}(v), v\ \mathsf{instof}\ \mathsf{N}\}$$

That is, call-effects of the `throw` method in N denote the set of the exceptions (represented by objects) of subtypes of N, and the other operators have the obvious set-theoretic meaning.

▶ **Example 12.** Consider the monad of non-determinism as in Example 6. We have $L\mathsf{Res}$ the set of (possibly infinite) lists of results (values or wrong). We show two different ways to define the typing judgment for monadic results, denoted $\vdash^{\mathsf{q}}_E \mathrm{R} : T$ for $\mathsf{q} ::= \forall \mid \exists$.

$$\text{(T-NO-RES)}\ \frac{}{\vdash^{\mathsf{q}}_E \epsilon : T} \qquad \text{(T-DET)}\ \frac{\vdash v : T}{\vdash^{\mathsf{q}}_E [v] : T}\ [\![E]\!] = 0$$
$$\text{(T-}\forall\text{)}\ \frac{\vdash r : T\ \forall r \in \mathrm{R}}{\vdash^{\forall}_E \mathrm{R} : T}\ [\![E]\!] = 1 \qquad \text{(T-}\exists\text{)}\ \frac{\vdash r : T}{\vdash^{\exists}_E \mathrm{R} : T}\ [\![E]\!] = 1\ r \in \mathrm{R}$$

where $[\![E]\!]$ is 1 if non-determinism is allowed, 0 otherwise, as defined below:

$$[\![\bullet]\!] = 0 \qquad [\![\top]\!] = 1 \qquad [\![E \vee E']\!] = [\![E]\!] \vee [\![E']\!]$$
$$[\![\texttt{Chooser.choose}[\ ]]\!] = 1$$

That is, monadic results (representing possible results of a computation) are well-typed with a type effect (denoting) 0 if they have at most one element, and this element, if any, is well-typed; in other words, the computation is deterministic. On the other hand, they are well-typed with a type effect (denoting) 1 if all the results in the list are well-typed

[13] The effect is written under the turnstile to emphasize that this typing judgment is obtained by lifting the non-monadic one, as will be described in the following.

values, or there is either no result, or at least one well-typed value, respectively. The two interpretations express "must" and "may" soundness of a non-deterministic computation: with $\forall$, we require every possible result to be well-typed, whereas, with $\exists$, it is enough to have a well-typed result, and the others could be even wrong.

Interpretation of effect types. Monadic typing judgments can be derived from non-monadic ones by providing an *interpretation of effect types* into the considered monad.

Let us denote by Eff the set of effects, assumed to be simplified (Figure 10), ground, and well-formed, considered modulo the equivalence induced by the subtyping preorder[14]. As customary, we write E for the element of Eff it represents, i.e., its equivalence class. Then, it is easy to see that Eff is the carrier of an ordered monoid $\mathcal{E}$, where the unit is $\bullet$, the multiplication, which turns out to be idempotent and commutative, is $\vee$, and the order is $\leq$.

For a set X, we denote by $\mathcal{P}(X)$ the poset of all subsets (a.k.a. predicates or properties) on X, ordered by subset inclusion. For a function $f : X \to Y$, we have a monotone function $\mathcal{P}_f : \mathcal{P}(Y) \to \mathcal{P}(X)$, given by the inverse image: for $A \subseteq Y$, $\mathcal{P}_f(A) = \{x \in X \mid f(x) \in A\}$.

▶ **Definition 13** (Interpretation of effect types). *Let $\mathbb{M} = \langle M, \mu, \eta\rangle$ be a monad. Then, an* interpretation *of $\mathcal{E}$ in $\mathbb{M}$ consists of a family λ of monotone functions $\lambda^E_X : \mathcal{P}(X) \to \mathcal{P}(MX)$, for every $E \in \mathsf{Eff}$ and set X, such that*

1. *$\lambda^E_X(\mathcal{P}_f(A)) = \mathcal{P}_{Mf}(\lambda^E_Y(A))$, for every $A \subseteq Y$ and function $f : X \to Y$*
2. *$E \leq E'$ implies $\lambda^E_X(A) \subseteq \lambda^{E'}_X(A)$, for every $A \subseteq X$*
3. *$A \subseteq \mathcal{P}_{\eta_X}(\lambda^\bullet_X(A))$, for every $A \subseteq X$*
4. *$\lambda^E_{MX}(\lambda^{E'}_X(A)) \subseteq \mathcal{P}_{\mu_X}(\lambda^{E\vee E'}_X(A))$, for every $A \subseteq X$.*

The family $\lambda = (\lambda^E)_{E\in\mathsf{Eff}}$ is a family of predicate liftings [22] for the monad $\mathbb{M}$, indexed by effect types. That is, for each effect type E, λ^E transforms predicates on X into predicates on MX, and can be regarded as the semantics of the effect type E.

Item 1 states that λ^E_X is natural in X. This ensures that the semantics of each effect type is independent from the specific set X. Item 2 states that λ^E_X is monotone with respect to the order on effects, that is, computational effects described by E are also described by E'. Item 3 states that monadic elements in the image of η_X contain computational effects described by $\bullet$, that is, no computational effect. Finally, in Item 4 we consider elements of M^2X whose computational effects are described by lifting predicates to MX through E', and then by lifting through E. By flattening such elements through $\mu_X : M^2X \to MX$ we obtain elements whose computational effects are described by the composition $E \vee E'$.

In the proof of soundness for our calculus, we only need Item 3 and Item 1, instantiated as will be detailed in Lemma 16. Item 2 and Item 4 are only required in the general framework in [12] to derive soundness from progress and subject reduction.

We show now how the previous examples can be obtained through an appropriate interpretation of effect types.

In Example 11, we can take $\lambda^E_X(A) = A + [\![E]\!]$. That is, the interpretation of an effect type E transforms a predicate A on X into a predicate on MX which holds either on elements which satisfy the original predicate, or exceptions in the set denoted by E.

In Example 12, we can take the following two interpretations $\forall$ and $\exists$:

if $[\![E]\!] = 1$, then

[14] That is, we consider effects as sets of call-effects, plus the top effect. For the formal definition of well-formedness and subtyping, which are standard, see [11].

$\forall^E_X(A) = \{B \in PX \mid B \subseteq A\}$ and
$\exists^E_X(A) = \{B \in PX \mid B = \emptyset \text{ or } B \cap A \neq \emptyset\}$

if $[\![E]\!] = 1$, then $\forall^0_X(A) = \exists^0_X(A) = \{B \in PX \mid B \subseteq A \text{ and } \sharp B \leq 1\}$

That is, in both cases, the interpretation of 0 forbids non-determinism, while the interpretation of 1 requires the predicate A to be always satisfied, according to $\forall$, and satisfied in at least one case, according to $\exists$.

Proof of type-and-effect soundness. Thanks to a general result proved in [12], to prove Theorem 9 and Theorem 10 we can use a technique similar to that widely used to prove soundness of a type system with respect to a small-step semantics, that is, it is enough to prove the progress and subject reduction properties, stated below. For the infinitary soundness, the interpretation of effect types has to respect the additional structure of the monad, that is, the least monadic element should be always well-typed, and the monadic typing judgment should be closed with respect to suprema of ω-chains. We refer to [12] for the formal definition of such requirement, which trivially holds in our case.

Let us denote by $\vdash_{\hat{E}} \mathrm{V} : T$ and $\vdash_{\hat{E}} \mathrm{E} : T!E$ the typing judgments on monadic values and expressions obtained by lifting the non-monadic ones through $\hat{E}$.

▶ **Theorem 14** (Monadic Progress). *$\vdash e : T!E$ implies either $e = \mathtt{return}\ v$ for some $v \in \mathsf{Val}$, or $e \to \mathrm{E}$ for some $\mathrm{E} \in M\mathsf{Exp}$.*

▶ **Theorem 15** (Monadic Subject Reduction). *$\vdash e : T!E$ and $e \to \mathrm{E}$ imply $\vdash_{\hat{E}} \mathrm{E} : T!E'$ for some $T', \hat{E}, E'$ such that $T'!\hat{E} \vee E' \leq T!E$.*

Monadic progress is standard: a well-typed expression either is (the embedding of) a value or can reduce. For monadic subject reduction, if a well-typed expression reduces to a monadic one, then the monadic expression has a more specific type, and the expression effect is an upper bound of the computational effects produced by the current reduction step, described by $\hat{E}$, composed with those produced by future reductions, described by E'.

Remarkably enough, the proofs of monadic progress and subject reduction can be driven by the usual rule-based reasoning, without any need to know about Definition 13, thanks to Lemma 16 below. Recall that the interpretation of effects *lifts* predicates to monadic predicates, in our case typing judgments to monadic typing judgments. Lemma 16 shows that it is even possible to lift, in a sense, the type system itself. That is, we can derive a typing rule for each operator on monadic elements used in the semantics. The proofs in the following only rely on such monadic type system.

▶ **Lemma 16.** *The following rules can be derived.*

$$(\text{T-UNIT})\ \frac{\vdash e : T!E}{\vdash_\bullet \eta(e) : T!E} \qquad (\text{T-RET-LIFT})\ \frac{\vdash_{\hat{E}} \mathrm{V} : T}{\vdash_{\hat{E}} \mathsf{map}\,(\mathtt{return}\ [\,])\ \mathrm{V} : T!\bullet}$$

$$(\text{T-DO-LIFT})\ \frac{\vdash_{\hat{E}} \mathrm{E} : T!E \qquad \emptyset; x : T \vdash e' : T'!E'}{\vdash_{\hat{E}} \mathsf{map}\,(\mathtt{do}\ x = [\,];\ e')\mathrm{E} : T'!E \vee E'}$$

Proof. By instantiating Definition 13(3) with $X = \mathsf{Exp}$ and $A = \{e \mid \vdash e : T!E\}$, we get $A \subseteq \{e \mid \vdash_\bullet \eta(e) : T!E\}$, that is, $\vdash e : T!E$ implies $\vdash_\bullet \eta(e) : T!E$, as expressed by rule (T-UNIT).

We show how to derive (T-RET-LIFT), (T-DO-LIFT) can be derived analogously. Since, by rule (T-RET), $\vdash v : T$ implies $\vdash \mathtt{return}\ v : T!\bullet$, by the monotonicity of $\lambda^{\hat{E}}$ we have that, set $Pre = \{v \mid \vdash v : T\}$, $Cons = \{v \mid \vdash \mathtt{return}\ v : T!\bullet\}$, $\lambda^{\hat{E}}_{\mathsf{Val}}(Pre) \subseteq \lambda^{\hat{E}}_{\mathsf{Val}}(Cons)$. Since $\lambda^{\hat{E}}_{\mathsf{Val}}(Pre) = \{\mathrm{V} \mid$

$\vdash_{\hat{E}} \mathrm{V} : T\}$, and, by Definition 13(1), instantiated with function (context) $\texttt{return}\ [\,] : \mathsf{Val} \to \mathsf{Exp}$ and $A = \{e \mid \vdash e{:}T!\bullet\}$, $\lambda^{\hat{E}}_{\mathsf{Val}}(\mathit{Cons}) = \{\mathrm{V} \mid \vdash_{\hat{E}} \mathsf{map}\,(\texttt{return}\ [\,])\ \mathrm{V} : T!\bullet\}$, we get that $\vdash_{\hat{E}} \mathrm{V} : T$ implies $\vdash_{\hat{E}} \mathsf{map}\,(\texttt{return}\ [\,])\ \mathrm{V} : T!\bullet$, as expressed by rule (T-RET-LIFT). ◀

Lemma 16 derives a typing rule for each monadic operator used in the semantics. In addition, the semantics uses monadic constants, which are the results of some $\mathsf{run}_{\mathsf{N},\mathsf{m}}$ function, depending on (the magic methods declared in) the specific program. Hence, like standard constants, we have to assume that these are well-typed. To this end, we require a typing rule for such constants, as given below. Essentially, since the (canonical) effect of a magic method m declared in N should be an approximation of the computational effects raised by a call, formalized by the monadic value obtained as result of $\mathsf{run}_{\mathsf{N},\mathsf{m}}$, then such monadic value should be well-typed with respect to the type-and-effect of the method.

$$\text{(T-RUN)}\ \frac{\vdash v_0 : T_0 \qquad \vdash \overline{v} : \overline{T'}}{\vdash_{\mathsf{N}[\overline{T_Y}].\mathsf{m}[\overline{T_X}]} \mathsf{run}_{\mathsf{N},\mathsf{m}}(\overline{v}) : T[\overline{T_Y}/\overline{Y}][\overline{T_X}/\overline{X}]}\ \begin{array}{l}\mathsf{mtype}(\mathsf{N}[\overline{Y} \lhd \overline{U_Y}] \lhd _\{_\}, \mathsf{m}){=}\mathsf{mgc}\ MT\\ MT = [\overline{X} \lhd \overline{U_X}]\overline{T} \to T!\mathsf{N}[\overline{Y}].\mathsf{m}[\overline{X}]\\ T_0 \le \mathsf{N}[\overline{T_Y}]\\ \overline{T_Y} \le \overline{U_Y}[\overline{T_Y}/\overline{Y}]\\ \overline{T'} \le \overline{T}[\overline{T_Y}/\overline{Y}][\overline{T_X}/\overline{X}]\\ \overline{T_X} \le \overline{U_X}[\overline{T_Y}/\overline{Y}][\overline{T_X}/\overline{X}]\end{array}$$

More in detail, given a magic method m declared in N (first and second side conditions, where the effect is the canonical one), the result of a $\mathsf{run}_{\mathsf{N},\mathsf{m}}$ function is well-typed if the first argument has a subtype of one obtained by instantiating the type variables in N, and each other argument has a subtype of that obtained by instantiating the type variables in N and those in m with the corresponding parameter type. The type and the effect of the result are obtained by an analogous instantiation of variables of the return type-and-effect.

We provide below the proof of monadic subject reduction (Theorem 15), noteworthy since it relies on the monadic typing rules, that is, those derived in Lemma 16 and rule (T-RUN). The proofs of monadic progress and subject reduction for the pure relation (Lemma 17) can be found in [11], together with the standard inversion and substitution lemmas.

▶ **Lemma 17** (Subject Reduction). *If* $\vdash e{:}T!E$ *and* $e \to_p e'$, *then* $\vdash e'{:}T'!E'$ *and* $T'!E' \le T!E$.

Proof of Theorem 15. Assuming $\vdash e{:}T!E$ and $e \to \mathrm{E}$, we have to prove that $\vdash_{\hat{E}} \mathrm{E} : T!E'$ for some $T', \hat{E}, E'$ such that $T'!\hat{E} \vee E' \le T!E$. By induction on the reduction rules of Figure 5.

(pure) E is $\eta(e')$ and $e \to_p e'$. From $\vdash e{:}T!E$ and Lemma 17 we get $\vdash e'{:}T'!E'$ with $T'!E' \le T!E$. From Lemma 16 by rule (T-UNIT) we derive $\vdash_{\bullet} \eta(e') : T'!E'$, and $T'!\bullet \vee E \le T!E$.

(mgc) e is $v_0.\mathsf{m}[\overline{T_X}](\overline{v})$ and $\mathrm{E} = \mathsf{map}\,(\texttt{return}\ [\,])\,\mathsf{run}_{\mathsf{N},\mathsf{m}}(v, \overline{v})$ and $\mathsf{lookup}(v_0, \mathsf{m}) = \langle \mathsf{mgc}, \mathsf{N}\rangle$. By inversion we get $\vdash v_0{:}T_0$ and $\vdash \overline{v}{:}\overline{T}$, and, from the definition of lookup, $\mathsf{mkind}(T_0, \mathsf{m}) = \mathsf{mgc}$. Hence, again by inversion,

- $\mathsf{mtype}(td, \mathsf{m}){=}\mathsf{mgc}\,[\overline{X} \lhd \overline{U_X}]\overline{T''} \to T''!\mathsf{N}[\overline{Y}].\mathsf{m}[\overline{X}]$ for some $td = \mathsf{N}[\overline{Y} \lhd \overline{U_Y}] \lhd _\{_\}$
- there is $\overline{T_Y}$ such that $T_0 \le \mathsf{N}[\overline{T_Y}]$ and $\overline{T_Y} \le U_Y[\overline{T_Y}/\overline{Y}]$
- $\overline{T_X} \le \overline{U_X}[\overline{T_Y}/\overline{Y}][\overline{T_X}/\overline{X}]$ and $\overline{T'} \le \overline{T''}[\overline{T_Y}/\overline{Y}][\overline{T_X}/\overline{X}]$ and $T = T''[\overline{T_Y}/\overline{Y}][\overline{T_X}/\overline{X}]$ and $E = \mathsf{N}[\overline{T_Y}].\mathsf{m}[\overline{T_X}]$.

Since the premises of rule (T-RUN) are satisfied, we get $\vdash_{E} \mathsf{run}_{\mathsf{N},\mathsf{m}}(v, \overline{v}) : T$. From Lemma 16 by rule (T-RET-LIFT) we get $\vdash_{E} \mathsf{map}\,(\texttt{return}\ [\,])\ \mathsf{run}_{\mathsf{N},\mathsf{m}}(v, \overline{v}) : T!\bullet$, and $T!E \vee \bullet \le T!E$.

(ret) e is $\texttt{do}\ x = \texttt{return}\ v;\ e'$ and $\mathrm{E} = \eta(e'[v/x])$. By inversion $\vdash v{:}T_1$ and $\emptyset; x{:}T_1 \vdash e'{:}T!E$. Therefore, by substitution, we get $\vdash e'[v/x]{:}T!E$. From Lemma 16 by (T-UNIT) we derive $\vdash_{\bullet} \eta(e'[v/x]) : T!E$, and $T!\bullet \vee E \le T!E$.

(do) $e = \mathsf{do}\ x = e_1\,;\ e_2$ and $\mathrm{E} = \mathsf{map}\,(\mathsf{do}\ x = [\ ]\,;\ e_2)\,\mathrm{E}_1$ and $e_1 \to \mathrm{E}_1$. By inversion $E' = E_1 \vee E_2$ and $\vdash e_1 : T_1 ! E_1$ and $\emptyset; x : T_1 \vdash e_2 : T ! E_2$. By induction hypothesis $\vdash_{\hat{E}_1} \mathrm{E}_1 : T_1' ! E_1'$ for some $\hat{E}_1$ and E_1' with $T_1' ! \hat{E}_1 \vee E_1' \leq T_1 ! E_1$. From Lemma 16 by rule (T-DO-LIFT) we get $\vdash_{\hat{E}_1} \mathsf{map}\,(\mathsf{do}\ x = [\ ]\,;\ e')\mathrm{E} : T ! E_1' \vee E_2 \mathrm{U}$. Therefore, we get the thesis noting that $T ! \hat{E}_1 \vee E_1' \vee E_2 \leq T ! E_1 \vee E_2 \leq T ! E$. ◀

6 Related Work

Design of the pure language. As mentioned, the pure calculus we choose as basis for the effectful extension is inspired by recent work [49, 43, 50] proposing an object-oriented paradigm based on interfaces with default methods and anonymous classes, rather than class instances with fields. In particular, classless objects were also adopted in *interface-based programming* [49], enabling multiple inheritance by decoupling state and behaviour. Then, in *λ-based object-oriented programming* [43], objects are classless and stateless, but limited to be lambdas implementing a single abstract method. In the Fearless language [50], as in our calculus, this approach is generalized to arbitrary Java anonymous classes with no fields.

Design of the effectful language. In our effectful language, constructs to raise effects, similarly to generic effects [36], are just method calls of a special kind, called *magic*. As mentioned, this terminology is taken from Fearless [50], and also informally used in Java reflection and Python. Concerning constructs for handling effects, handlers of algebraic effects [38, 39, 24, 19] are an extremely powerful programming abstraction, describing the semantics of algebraic operations in the language itself, thus enabling the simulation of several effectful programs, such as stream redirection or cooperative concurrency [39, 24, 6, 41]. The code executed when a call is caught can resume the original computation, using a form of continuation-passing style. Differently from exception handling, all occurrences of an operation within the scope of a handler are handled. Other forms of handlers have been considered, e.g., shallow handlers [24, 19], where only the first call is handled. The handling mechanism in our language is meant to provide a good compromise between simplicity and expressivity: notably, it does not explicitly handle continuations, as in handlers of algebraic effects, hence we cannot resume the original computation more than once; yet we can express a variety of handling mechanisms.

Type-and-effect system. Type-and-effect systems, or simply effect systems [47, 33, 48, 30, 25], are a popular way of statically controlling computational effects. Many of them have been designed for specific notions of computational effect and also implemented in mainstream programming languages, the most well-known being the mechanism of Java exceptions. Katsumata [25] recognized that effect systems share a common algebraic structure, notably they form an ordered monoid, and gave them denotational semantics through parametric monads, using a structure equivalent to our notion of interpretation.

Our effects are essentially sets, like in formal models of Java exceptions [3], tracking which magic calls an expression is allowed to do. Differently from effect systems for algebraic effects [5, 6, 41], also implemented in Eff [4], and inspired by [15], we enrich the effects with type information, not using just the name of magic methods. By using this type information, which is polymorphic, we also allow effects to depend on the effect of other methods, smoothly enabling a form of effect polymorphism. Effect systems supporting effect polymorphism have been widely studied for algebraic effects [26, 20, 29, 7, 9, 8], especially in the form of row polymorphism. A precise comparison with these systems is left for future work.

Semantics for effectful languages. Algebraic effects are typically treated as uninterpreted operations, that is, the evaluation process just builds a tree of operation calls [23, 44, 41]. Monadic operational semantics for λ-calculi with algebraic effects are also considered, mainly in the form of a monadic definitional interpreter (see, e.g., [28, 14, 16, 13, 10]). That is, they directly define a function from expressions to monadic values, which essentially corresponds to our infinitary semantics. Small-step approaches are also considered by [17, 18].

An alternative way of interpreting algebraic operations is by means of runners, a.k.a. comodels [40, 37, 45, 2]. Roughly, runners describe how operations are executed by the system, that is, how they transform the environment where they are run. This essentially amounts to giving an interpretation of operations in the state monad. More general runners, where the system is modelled in a more expressive way, are considered by [2], where the state monad is combined with errors and system primitive operations.

The motivation to have methods interpreted in a monad, rather than implemented in code, is the same nicely illustrated for runners by [2]. They observe that some kinds of effects, such as input/output, even though represented in the language as algebraic operations (as in Eff) or a monad (as in Haskell), are handled by an operating system functionality. Hence, semantics lacks modularity, which can be recovered by introducing runners. The same modularity is achieved in our calculus by providing the interpretations of magic methods as an independent ingredient of the semantics, which can be regarded as the operational counterpart of (effectful) runners. Differently from runners of [2], our magic methods can be interpreted in an arbitrary monad. Moreover, our interpretation is a parameter of the operational semantics, while runners of [2] are described in the language itself through a mechanism similar to effect handlers, with a limited use of continuations.

7 Conclusion

This paper provides a positive answer to the question:

> *Is it possible to smoothly incorporate in the object-oriented paradigm constructs to raise, compose, and handle effects in an arbitrary monad?*

In particular, we consider a pure OO paradigm based on interfaces with default methods and anonymous classes, exemplified by a higher-order calculus subsuming both FJ and λ-calculus. The effectful extension is designed to be familiar for OO programmers, with magic methods to raise effects, and an expressive, yet simple, handling mechanism. Effect types are designed to be familiar, being sets of call-effects generalizing exception names; the key novel feature is that type effects containing type variables can be refined in concrete calls, through a non-trivial process called *simplification*. The semantics and the soundness proof are given by a novel approach [12], providing a significant application for a language with a complex type system (both nominal and structural types, subtyping and generic types). Instantiating such an approach, we derive from the type system a *monadic type system*, allowing to carry the proof of subject reduction by the usual rule-based reasoning.

The integration of effectful features turned out to be, in fact, even smoother than in the functional case, fitting naturally in the OO paradigm, where computations happen in the context of a given program (class table in Java). That is, magic methods are, from the point of view of the programmer, as other methods offered by the environment (system, libraries), with the only difference that their calls can be handled. Another advantage is that, since methods "belong to a type", atomic effects (call-effects in our paper) naturally include a type, rather than being just operation names as in previous literature handling the functional case. Thus, effect polymorphism is obtained for free from the fact that types can be type variables, without any need of ad-hoc effect variables.

Concerning the specific calculus, we could have likely used FJ (with generic types) as well. However we have, besides nominal, structural and intersection types, hence a paradigm partly object-based, as in seminal object calculi [1], rather than class-based. This leads to more flexibility, hence the tracking of effects is more flexible as well (can be done on an object's basis). In particular, FJ is not higher-order, meaning that functions are not first-class values as customary in functional languages. Formally, in the translation of λ-calculus shown in Remark 1, any arbitrary function is encoded by an object literal; without object literals, only a finite collection of functions could be encoded, each one by a class, in a program.

This is a foundational paper, hence we focus on the introduction of design features, formalisation, and results through a toy language. Of course, any step towards an implementation and/or the application to a realistic language would be an important contribution, as it would be to mechanize some proofs. Other directions for future work are outlined below.

We plan to analyze better the handling mechanism proposed in the paper, and its relation with other approaches. The type-and-effect system we design is, as said above, novel and expressive; however, it is very simple in the sense that effect types are essentially sets. Hence, it is not adequate to approximate computational effects where the order matters, such as, e.g., sequences of write operations. A type-and-effect system with a non-commutative operator on effects is shown in [12]; however, effects there are purely semantics, and a syntactic representation should be investigated in order to apply the approach to real languages.

Finally, the interpretation of magic methods could return monadic expressions, rather than values. This would enable a more interactive behaviour with the system; for instance, the semantics of a magic method, instead of returning an unrecoverable error, could return a call to the method `throw` of an exception, which then could be handled by the program.

References

1 Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. *Information and Computation*, 125(2):78–102, 1996. doi:10.1006/INCO.1996.0024.

2 Danel Ahman and Andrej Bauer. Runners in action. In Peter Müller, editor, *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020*, volume 12075 of *Lecture Notes in Computer Science*, pages 29–55. Springer, 2020. doi:10.1007/978-3-030-44914-8_2.

3 Davide Ancona, Giovanni Lagorio, and Elena Zucca. A core calculus for Java exceptions. In Linda M. Northrop and John M. Vlissides, editors, *Proceedings of the ACM International Conference on Object-Oriented Programming: Systems, Languages and Applications, OOPSLA 2001*, pages 16–30. ACM Press, 2001. doi:10.1145/504282.504284.

4 Andrej Bauer and Matija Pretnar. Eff. URL: https://www.eff-lang.org/.

5 Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, 10(4), 2014. doi:10.2168/LMCS-10(4:9)2014.

6 Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015. doi:10.1016/J.JLAMP.2014.02.001.

7 Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Abstracting algebraic effects. *Proceedings of the ACM on Programming Languages*, 3(POPL):6:1–6:28, 2019. doi:10.1145/3290319.

8 Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):126:1–126:30, 2020. doi:10.1145/3428194.

9 Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in scala. *Journal of Functional Programming*, 30:e8, 2020. doi:10.1017/S0956796820000027.

10 Francesco Dagnino and Francesco Gavazzo. A fibrational tale of operational logical relations: Pure, effectful and differential. *Logical Methods in Computer Science*, 20(2), 2024. doi:10.46298/LMCS-20(2:1)2024.

11 Francesco Dagnino, Paola Giannini, and Elena Zucca. An effectful object calculus (extended version). *CoRR*, 2025. doi:10.48550/arXiv.2504.15936.

12 Francesco Dagnino, Paola Giannini, and Elena Zucca. Monadic type-and-effect soundness. In Jonathan Aldrich and Alexandra Silva, editors, *39th European Conference on Object-Oriented Programming, ECOOP 2025*, volume 333 of *LIPIcs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.ECOOP.2025.1.

13 Ugo Dal Lago and Francesco Gavazzo. A relational theory of effects and coeffects. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–28, 2022. doi:10.1145/3498692.

14 David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. Abstracting definitional interpreters (functional pearl). *Proceedings of the ACM on Programming Languages*, 1(ICFP):12:1–12:25, 2017. doi:10.1145/3110256.

15 Isaac Oscar Gariano, James Noble, and Marco Servetto. Calle: an effect system for method calls. In Hidehiko Masuhara and Tomas Petricek, editors, *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019*, pages 32–45. ACM, 2019. doi:10.1145/3359591.3359731.

16 Francesco Gavazzo. Quantitative behavioural reasoning for higher-order effectful programs: Applicative distances. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018*, pages 452–461. ACM, 2018. doi:10.1145/3209108.3209149.

17 Francesco Gavazzo and Claudia Faggian. A relational theory of monadic rewriting systems, part I. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021*, pages 1–14. IEEE, 2021. doi:10.1109/LICS52264.2021.9470633.

18 Francesco Gavazzo, Riccardo Treglia, and Gabriele Vanoni. Monadic intersection types, relationally. In Stephanie Weirich, editor, *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024*, volume 14576 of *Lecture Notes in Computer Science*, pages 22–51. Springer, 2024. doi:10.1007/978-3-031-57262-3_2.

19 Daniel Hillerström and Sam Lindley. Shallow effect handlers. In Sukyoung Ryu, editor, *Proceedings of the 16th Asian Symposium on Programming Languages and Systems, APLAS 2018*, volume 11275 of *Lecture Notes in Computer Science*, pages 415–435. Springer, 2018. doi:10.1007/978-3-030-02768-1_22.

20 Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. Continuation passing style for effect handlers. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017*, volume 84 of *LIPIcs*, pages 18:1–18:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.FSCD.2017.18.

21 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the ACM International Conference on Object-Oriented Programming: Systems, Languages and Applications, OOPSLA 1999*, pages 132–146. ACM Press, 1999. doi:10.1145/320384.320395.

22 Bart Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*, volume 59 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2016. doi:10.1017/CBO9781316823187.

23 Patricia Johann, Alex Simpson, and Janis Voigtländer. A generic operational metatheory for algebraic effects. In *Proceedings of the 25th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2010*, pages 209–218. IEEE Computer Society, 2010. doi:10.1109/LICS.2010.29.

24 Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP 2013*, pages 145–158. ACM, 2013. doi:10.1145/2500365.2500590.

25 Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In Suresh Jagannathan and Peter Sewell, editors, *Proceedings of the 41st ACM/SIGPLAN Symposium on Principles of Programming Languages, POPL 2014*, pages 633–646. ACM, 2014. doi:10.1145/2535838.2535846.

26 Daan Leijen. Type directed compilation of row-typed algebraic effects. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM/SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*, pages 486–499. ACM, 2017. doi:10.1145/3009837.3009872.

27 Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, 2003. doi:10.1016/S0890-5401(03)00088-9.

28 Sheng Liang, Paul Hudak, and Mark P. Jones. Monad transformers and modular interpreters. In Ron K. Cytron and Peter Lee, editors, *Proceedings of the 22nd ACM/SIGPLAN Symposium on Principles of Programming Languages, POPL 1995*, pages 333–343. ACM Press, 1995. doi:10.1145/199448.199528.

29 Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM/SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*, pages 500–514. ACM, 2017. doi:10.1145/3009837.3009897.

30 Daniel Marino and Todd D. Millstein. A generic type-and-effect system. In Andrew Kennedy and Amal Ahmed, editors, *TLDI'09: Types in Languages Design and Implementatio*, pages 39–50. ACM Press, 2009. doi:10.1145/1481861.1481868.

31 Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the 4th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 1989*, pages 14–23. IEEE Computer Society, 1989. doi:10.1109/LICS.1989.39155.

32 Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. doi:10.1016/0890-5401(91)90052-4.

33 Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design, Recent Insight and Advances*, volume 1710 of *Lecture Notes in Computer Science*, pages 114–136. Springer, 1999. doi:10.1007/3-540-48092-7_6.

34 Gordon D. Plotkin and John Power. Adequacy for algebraic effects. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FoSSaCS 2001*, volume 2030 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001. doi:10.1007/3-540-45315-6_1.

35 Gordon D. Plotkin and John Power. Notions of computation determine monads. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures, 5th International Conference, FoSSaCS 2002*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002. doi:10.1007/3-540-45931-6_24.

36 Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003. doi:10.1023/A:1023064908962.

37 Gordon D. Plotkin and John Power. Tensors of comodels and models for operational semantics. In Andrej Bauer and Michael W. Mislove, editors, *The 24th Conference on Mathematical Foundations of Programming Semantics, MFPS 2008*, volume 218 of *Electronic Notes in Theoretical Computer Science*, pages 295–311. Elsevier, 2008. doi:10.1016/J.ENTCS.2008.10.018.

38 Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems - 18th European Symposium on Programming, ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009. doi:10.1007/978-3-642-00590-9_7.

39 Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013. doi:10.2168/LMCS-9(4:23)2013.

40 A. John Power and Olha Shkaravska. From comodels to coalgebras: State and arrays. In Jirí Adámek and Stefan Milius, editors, *Proceedings of the Workshop on Coalgebraic Methods in Computer Science, CMCS 2004*, volume 106 of *Electronic Notes in Theoretical Computer Science*, pages 297–314. Elsevier, 2004. doi:10.1016/J.ENTCS.2004.02.041.

41 Matija Pretnar. An introduction to algebraic effects and handlers. Invited tutorial paper. In Dan R. Ghica, editor, *The 31st Conference on Mathematical Foundations of Programming Semantics, MFPS 2015*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 19–35. Elsevier, 2015. doi:10.1016/J.ENTCS.2015.12.003.

42 Emily Riehl. *Category theory in context*. Courier Dover Publications, 2017.

43 Marco Servetto and Elena Zucca. λ-based object-oriented programming (pearl). In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021*, volume 194 of *LIPIcs*, pages 21:1–21:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ECOOP.2021.21.

44 Alex Simpson and Niels F. W. Voorneveld. Behavioural equivalence via modalities for algebraic effects. *ACM Transactions on Programming Languages and Systems*, 42(1):4:1–4:45, 2020. doi:10.1145/3363518.

45 Tarmo Uustalu. Stateful runners of effectful computations. In Dan R. Ghica, editor, *The 31st Conference on Mathematical Foundations of Programming Semantics, MFPS 2015*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 403–421. Elsevier, 2015. doi:10.1016/J.ENTCS.2015.12.024.

46 Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995. doi:10.1007/3-540-59451-5_2.

47 Philip Wadler. The marriage of effects and monads. In Matthias Felleisen, Paul Hudak, and Christian Queinnec, editors, *3rd ACM SIGPLAN International Conference on Functional Programming, ICFP 1998*, pages 63–74. ACM, 1998. doi:10.1145/289423.289429.

48 Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Transactions on Computational Logic*, 4(1):1–32, 2003. doi:10.1145/601775.601776.

49 Yanlin Wang, Haoyuan Zhang, Bruno C. d. S. Oliveira, and Marco Servetto. Classless Java. In Bernd Fischer and Ina Schaefer, editors, *Generative Programming: Concepts and Experiences, GPCE 2016*, pages 14–24. ACM Press, 2016. doi:10.1145/2993236.2993238.

50 Nick Webster, Marco Servetto, and Michael Homer. The Fearless journey [Draft]. *CoRR*, abs/2405.06233, 2014. doi:10.48550/arXiv.2405.06233.

Event Race Detection for Node.js Using Delay Injections

Andre Takeshi Endo
Federal University of São Carlos, Brazil

Anders Møller
Aarhus University, Denmark

Abstract

Node.js is a widely used platform for building JavaScript server-side web applications, desktop applications, and software engineering tools. Its asynchronous execution model is essential for performance, but also gives rise to event races, which cause many subtle bugs that can be hard to detect and reproduce. Current solutions to expose such races are based on modifications of the source code of the Node.js system or on guided executions using complex happens-before modeling.

This paper presents a simpler and more effective approach called NACD that works by dynamically instrumenting core asynchronous operations in the Node.js runtime system to inject delays and thereby reveal event race bugs. It consists of a small, robust runtime instrumentation module implemented in JavaScript that is configured by a flexible JSON model of the essential parts of the Node.js API. Experimental results show that NACD can reproduce event race bugs with higher probability and fewer runs than state-of-the-art tools.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging

Keywords and phrases JavaScript, race conditions, flaky tests, event races, callback interleaving

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.9

Supplementary Material *Software*: `https://github.com/andreendo/nacd`
archived at `swh:1:dir:d2c0ec68822296734f18d0607ea05be6f8ac9637`

Funding *Andre Takeshi Endo*: Supported by grant #2023/00577-8, São Paulo Research Foundation (FAPESP), Brazil.

1 Introduction

Node.js[1] is an asynchronous, event-driven JavaScript runtime designed for building scalable network applications. Its single-threaded, non-blocking I/O architecture makes it well suited for server-side web applications, microservices, desktop applications, and command-line tools. Node.js comes with a lean core API that offers access to file systems, networking, cryptography, timers, and processes, and is supported by a vast ecosystem of open-source libraries and frameworks available via the npm[2] package manager.

The single-threaded asynchronous execution model of Node.js brings new challenges regarding concurrency [18]. To avoid blocking the main thread, Node.js delegates I/O or high processing tasks to worker threads that run in the background. These tasks are initiated through asynchronous calls, and their results are processed through JavaScript callbacks acting as event handlers. The single-threaded model makes a Node.js application immune to traditional data races. However, the timing of worker threads and event handlers is non-deterministic, which makes Node.js applications prone to so-called event races.

[1] `https://nodejs.org/`
[2] `https://www.npmjs.com/`

© Andre Takeshi Endo and Anders Møller;
licensed under Creative Commons License CC-BY 4.0

Event races may cause system crashes, data corruption, application hangs, and security vulnerabilities. Detecting whether a program is susceptible to such harmful event races is often challenging. Approaches like NodeAV [6], NRace [7] and NodeRT [30] adopt a predictive strategy in which a happens-before model is built and used to predict event races from individual executions. Not all event races are harmful, so predictive approaches also employ mechanisms to filter out harmless races. Still, a key limitation of these predictive techniques is that they often produce many false positives.

Another group of approaches employ dynamic exploration or fuzzing of interleavings to produce executions that expose harmful event races. Node.fz [9] is based on a modification of the internal parts of Node.js written in C/C++ code to shuffle the task and event queues and thereby fuzz the scheduling of worker threads and event handlers. This approach is limited to reordering entries that are present in the scheduler queues, and it is difficult to maintain as Node.js evolves.[3] Differently, NodeRacer [11] works on the JavaScript side. It first observes a sample run, builds a happens-before graph, and then uses it in subsequent runs to explore different callback interleavings by instrumenting the application code to selectively postpone event handler executions. Although NodeRacer has shown to be effective in many cases, by design it only explores interleavings of the events seen in the sample run. As shown in Section 2, this prevents detection of certain event race errors.

We need an approach that (1) generates witnesses in the form of crashes or test failures whenever potential event race errors are reported (unlike NodeAV, NRace, and NodeRT), (2) does not require modifications of the Node.js source code (unlike Node.fz), and (3) is not limited to reordering of scheduler queues (unlike Node.fz) or event handler callbacks (unlike NodeRacer). This paper presents an event race detection technique named NACD[4] that satisfies these requirements. It is inspired by Node.fz and NodeRacer, but with some key differences that enhance maintainability and efficacy. The key idea is to fuzz the scheduling of event handlers by dynamically injecting random delays around both before and after the asynchronous functions in the built-in Node.js modules. This is achieved purely using JavaScript code, without modifications of the Node.js source code, and it avoids the complications of implementing happens-before computation and the overhead of program code instrumentation. By introducing delays rather than merely attempting to reorder events, more bugs can be found.

In summary, this paper makes the following main contributions:

1. We describe the design of NACD: a novel approach to event race detection for Node.js applications. It consists of a JavaScript component that dynamically instruments the Node.js module loading mechanism and is configured using a JSON model of the asynchronous operations in the Node.js API.
2. We present an experimental evaluation based on benchmarks from prior work, demonstrating that the approach has a high bug reproduction ratio and tends to find bugs with fewer runs compared to Node.fz and NodeRacer.

The remainder of the paper is organized as follows. Section 2 gives a motivating example. Section 3 describes the proposed approach, while Section 4 shows the main implementation details. Section 5 presents an experimental evaluation we conducted and the results obtained. Section 6 discusses the main limitations. The related work is presented in Section 7, and Section 8 makes the concluding remarks.

[3] The Node.fz implementation was based on Node.js v0.12.7, which is 32,720 commits from the latest version at the time of writing.

[4] Node.js Asynchronous Callback Delayer.

```
1   const fse = require('fs-extra');
2
3   it('should delete without a callback', (done) => { /*c1*/
4     const filePath = TEST_DIR + 'file.txt';
5     createFileSync(filePath, 'some text');
6
7     const existsChecker = setInterval(() => { /*c2*/
8       fse.pathExists(filePath, (itDoes) => { /*c3*/
9         if (!itDoes) {
10          clearInterval(existsChecker);
11          done();
12        }
13      });
14    }, 25);
15    fse.remove(file);
16  });
```

Figure 1 Motivating example.

2 Motivating Example

Figure 1 contains JavaScript code that is subjected to a race condition; this motivating example is based on a previously unknown race condition detected by NACD in the widely used Node.js package called fs-extra.[5] The code defines a test case for function `fse.remove` that receives the file path as argument and removes the file asynchronously. The test, here named `c1` (lines 3–16), intends to check if the removal is successful. Argument `done` is a function that signals to the test runner when the test is completed.[6] The test starts in lines 4–5 creating a text file that will be removed afterwards. In line 7, a timer is started[7] and its callback `c2` (lines 7–14) is called every 25 milliseconds. Line 15 executes the function under test that will remove the created file asynchronously. To test this operation, callback `c2` checks in line 8 if the file still exists, using an asynchronous call to `pathExists` with callback `c3`. Within `c3`, if the file does not exist (line 9), the timer is stopped (line 10) and the test ends successfully by calling function `done` in line 11.

Due to an existing race condition, this test is flaky as it may pass or fail non-deterministically [20]. This issue was reported and a pull request fixing it was accepted and merged.[8] Figure 2 illustrates the callback ordering, using directed edges to indicate happens-before relations between callbacks (nodes). The first row represents trivial executions (i.e., passing test runs) in which `c2` is enqueued only once by the timer, and `c3` stops the timer before `c2` is scheduled again. The race occurs if function `fse.pathExists` (line 8) takes long enough for the timer (line 7) to enqueue a new instance of its callback, referred to as `c2'`. The nodes and edges in the second row represent event handlers that only exist in this situation. When `c2'` is scheduled to run, it calls function `fse.pathExists` again, provoking a second instance of its callback, referred to as `c3'`. Both instances of `c3` are eventually invoked, and function `done` is called twice, which makes the test runner report a test failure even though the file has been removed.

The package `fs-extra` provides extra features for file system manipulation in Node.js. It uses other packages like `graceful-fs`, but ultimately those features depend on the Node.js built-in module `fs`. This scenario is predominant in Node.js applications, where application

[5] `https://www.npmjs.com/package/fs-extra`
[6] This mechanism is defined by testing frameworks like Mocha and Jest to test asynchronous code.
[7] `setInterval` is a Node.js function that starts a timer whose associated callback is called repeatedly after a number of milliseconds. It can be stopped by calling function `clearInterval`.
[8] `https://github.com/jprichardson/node-fs-extra/pull/736`

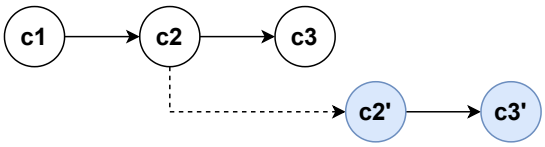


Figure 2 Callback order for motivating example.

code may use numerous third-party packages, but the asynchronous behavior comes from the built-in modules of Node.js. Notice in the example that, while `fs` is not used directly, several of its functions are called. An ordinary run of this test indirectly calls seven different `fs` functions with asynchronous behavior. In particular, package `fs-extra`'s function `fse.pathExists` is a promise-supported wrapper that uses built-in function `fs.access`.

Current versions of state-of-the-art tools cannot uncover the race bug illustrated in the example. Approaches based on happens-before relations (e.g., NodeRacer, NodeRT) rely on a logging phase to collect a trace and build happens-before relations. As previously discussed, an ordinary run will trace data to build happens-before relations similarly to the first row in Figure 2. If there is no potential interleaving between the callbacks, these approaches will fail to flag or explore this event race. Node.fz, which applies a different strategy, is also not capable of revealing this event race because it is not compatible with the Node.js version used by the project and cannot introduce sufficiently long delays [11].

3 Approach

Our approach to uncover event races, such as the one described in the previous section, is based on the observation that the asynchronous behavior originates from the core API provided by the Node.js built-in modules. By dynamically instrumenting those functions, we can introduce delays that will help to explore different callback interleavings and, as a consequence, increase the likelihood of exposing race bugs. Although centered on Node.js, the proposed approach is sufficiently general to be adapted to any software runtime that follows a single-threaded asynchronous model.

Figure 3 gives an overview of the proposed approach. A Node.js application consists of application code together with library code in the form of npm packages. Both application code and library code interact with Node.js built-in modules in order to access features related to networking, file system, cryptography, compression, etc. The key idea behind NACD is to inject delays in the asynchronous core API of the built-in modules, so that different interleavings of callbacks are explored, independently of the application and library code. For instance, delaying the callback of built-in function `fs.access` would be enough for `c2'` to be scheduled and provoke the event race in Figures 1-2. This requires no manipulation of application or library code, and explores callback interleavings within both levels.

To obtain a clean and extensible implementation of NACD, we first design a model of the API, describing its asynchronous behavior. As the Node.js built-in modules provide different API styles to perform asynchronous tasks, the model is based on a number of code patterns that are described in Section 3.1. Using this model, a runtime system instruments the API to inject delays that foster the execution of different callback scheduling. The runtime system and delay injection mechanisms are presented in Section 3.2.

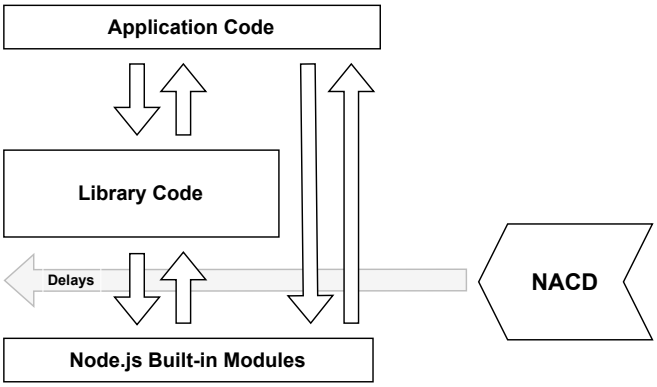


Figure 3 Overview of the proposed approach.

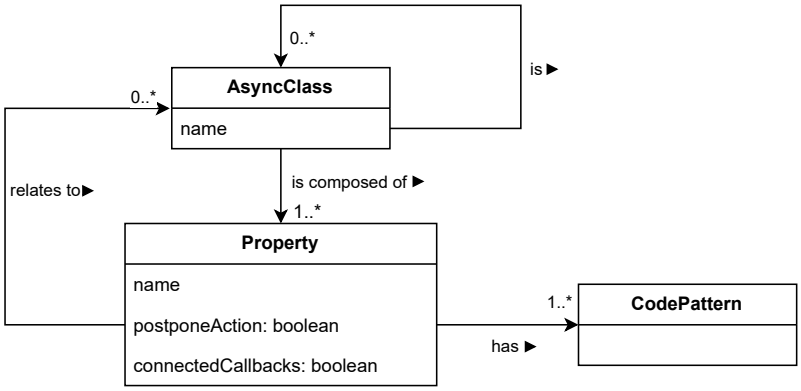


Figure 4 Model representation.

3.1 Modeling Node.js Asynchronous API

To perform the modeling of Node.js asynchronous API required to implement NACD, we have carefully studied the built-in modules and how they use asynchrony. Figure 4 shows how the proposed model is structured. It consists of a collection of *async classes*, each describing a module (e.g., `zlib`) or a class in the Node.js standard library (e.g., `zlib.ZlibTransformInterface`). Each async class has a name and is composed of *properties*. An async class can also inherit properties from other async classes, specified using the relationship `is`. For instance, several objects from the core API are event emitters, so we have an async class `EventEmitter` and several other classes inheriting from it.

The properties of an async class describe its functions and other objects. Each property may involve different forms of asynchronous behavior; we group these different styles in code patterns. Additionally, each property can be flagged as `postponeAction` or as `connectedCallbacks`, the meaning of which is explained below. Properties can also be related to other async classes; for example, a function may return an object with asynchronous behavior specified by another async class. Representative examples are presented in the following.

Next, we describe the asynchronous code patterns found in Node.js built-in modules. The identified code patterns are presented as follows together with illustrative example code.

Simple callback (CB). This pattern represents the most common case in which a function is associated with an asynchronous task and receives a callback as one of its arguments, which is scheduled to run in the future as a reaction to the task. Figure 5a shows an example where function `readFile` of the `fs` module is used to read the contents of a file and callback `c4` is called when the file contents are ready.

```
18 const fs = require('fs')
19 ...
20 fs.readFile('file.txt', (err, data) => { /*c4*/
21   ...
22 })
```

(a) Example code.

```
23 {
24   "moduleName": "fs",
25   "properties": [
26     {
27       "name": "readFile",
28       "type": [ "CB" ],
29       "postponeAction": true
30     },
31     ...
32   ]
33 }
```

(b) Model snippet.

Figure 5 Simple callback pattern.

Figure 5b shows how this code pattern is represented in the model. Using a JSON object for built-in module `fs` (line 24), it has an array of properties (starting in line 25). Function `readFile` (line 27) is identified with type `CB` (line 28), and the `postponeAction` is set to true, which instructs NACD to inject delays not only when `c4` is about to be executed but also before the file read action itself, thereby enabling exploration of alternative schedules that involve external actions.

Returned object (RO). This represents the case in which a function returns an object that is an instance of an async class. We refer to such objects as *async objects*. In Figure 6a, function `createGzip` of `zlib` returns an object (i.e., `gzip`) that supports compression using gzip (line 36). Object `gzip` is known within the API to have asynchronous behavior, as is later used to register callbacks for asynchronous events (lines 38–39).

Figure 6b shows how this code pattern is modeled. A JSON object for `createGzip` (line 44) with type `RO` (line 45) is defined as a property of `zlib` (line 41). Particularly, this code pattern requires attribute `returnedObject` (line 47). This attribute points to a different class (e.g., `zlib.ZlibTransformInterface` in lines 53–57) which has its own async behavior specified (omitted here). Consequently, the functions called in lines 38–39 of Figure 6a are identified by the model of `zlib.ZlibTransformInterface`.

Object creation (OC). This code pattern is similar to the previous case; here a class is employed to instantiate an async object. In Figure 7a, class `Agent` of built-in module `http` is used to instantiate an async object named `agent` (line 60).

```
34 const zlib = require('zlib')
35 ...
36 const gzip = zlib.createGzip()
37 ...
38 gzip.on('data', () => { ... })
39 gzip.on('end', () => { ... })
```

(a) Example code.

```
40 {
41   "moduleName": "zlib",
42   "properties": [
43     {
44       "name": "createGzip",
45       "type": [ "RO" ],
46       "postponeAction": false,
47       "returnedObject": "zlib.ZlibTransformInterface"
48     },
49     ...
50   ]
51 },
52 ...
53 {
54   "moduleName": "zlib.ZlibTransformInterface",
55   "is": [ "stream.Transform" ],
56   "properties": [ ... ]
57 }
```

(b) Model snippet.

Figure 6 Returned object pattern.

Figure 7b shows how this code pattern is modeled. A JSON object for `Agent` (line 67) with type `OC` (line 68) is defined as a property of `http` (line 64). Similarly to the previous code pattern, attribute `returnedObject` (line 70) has a reference for a different async class (`http.Agent` in lines 76–79).

Object property (`OP`). In this case, an async class has a property that is an async object itself. In Figure 8a, object `childProcess` is returned to represent a new process created using function `spawn` of built-in module `child_process` (line 82). This object has property `stdout` which is an async object itself for the standard output of the process (line 84).

Figure 8b shows how this code pattern is modeled. A JSON object for `child_process.ChildProcess` (line 98) is defined as the `returnedObject` attribute of function `spawn` (line 91). Among the properties of `child_process.ChildProcess`, `stdout` (line 101) is defined with type `OP`, requiring attribute `AsyncObjectProperty` that links to a different async class (i.e., `stream.ReadStream`).

Callback object (`CO`). In this case, the argument of a callback is an async object. In Figure 9a, function `createServer` of built-in module `net` is called to create a server, passing a callback for new connections (line 110). The argument of this callback, namely `socket`, is an async object itself.

Figure 9b shows how this code pattern is modeled. Using a JSON object for built-in module `net` (line 115), `createServer` (line 118) is defined with type `CO` (line 119). This code pattern requires attribute `callbackObjects` (line 121) with an array of async classes related to the arguments of its callback. In this example, the argument of `createServer`'s callback is a socket, represented in the model by async class `net.Socket` (lines 127–130).

```
58 const http = require('http')
59 ...
60 const agent = new http.Agent()
61 ...
62 agent.on('free', () => { ... })
```

(a) Example code.

```
63 {
64   "moduleName": "http",
65   "properties": [
66     {
67         "name": "Agent",
68         "type": [ "OC" ],
69         "postponeAction": false,
70         "returnedObject": "http.Agent"
71     },
72     ...
73   ]
74 },
75 ...
76 {
77   "moduleName": "http.Agent",
78   "is": [ "EventEmitter" ]
79 }
```

(b) Model snippet.

Figure 7 Object creation pattern.

Returned promise (RP). Some built-in modules support the use of JavaScript promises.[9] A promise is standard JavaScript built-in object that provides an abstraction for the result of an asynchronous computation [3]. Its eventual completion or failure is referred to as the promise being resolved or rejected, respectively. In either case, we say that the promise is settled.

In Figure 10a, function `resolve` is loaded from the promise API of built-in module `dns` (line 131). This function triggers a task to resolve a hostname, `'example.org'`, and returns a promise, which is asynchronously settled with an array of resource records. The result of this function is processed using `promise.then` in lines 134–135, while the function may be also used in the context of the JavaScript async-await style in line 140.

Figure 10b shows how this code pattern is modeled. The promise API is defined as an object property (line 149) of built-in module `dns` (line 144). Within `dns.Promises` (lines 155–164), function `resolve` (line 159) is identified with type `RP` (line 160).

NACD injects delays only into the promises returned by the core API of Node.js built-in modules. No special treatment is necessary for promise chains; delaying the settlement of the first promise in a chain automatically also affects the remaining promises in the chain.

Combining code patterns. Code patterns may be combined to represent the async behavior of more complex API functions. Figure 11 shows an example with an HTTP GET request. (To avoid redundancy from previously explanations, we omit the model code in the examples presented here.) In line 167, function `get` of built-in module `http` is used to request a JSON file from a web site. Making an HTTP request is an asynchronous operation, and a callback is also passed as the last argument. As mentioned before, this represents the code pattern

[9] `https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Promise`

```
80 const { spawn } = require('child_process')
81 ...
82 const childProcess = spawn('ls', ['-l', '/usr'])
83 ...
84 childProcess.stdout.on('data', (data) => { ... })
```

(a) Example code.

```
85 {
86   "moduleName": "child_process",
87   "properties": [
88     {
89       "name": "spawn",
90       "type": [ "RO" ],
91       "returnedObject": "child_process.ChildProcess"
92     },
93     ...
94   ]
95 },
96 ...
97 {
98   "moduleName": "child_process.ChildProcess",
99   "properties": [
100    {
101      "name": "stdout",
102      "type": [ "OP" ],
103      "AsyncObjectProperty": "stream.ReadStream"
104    },
105    ...
106  ]
107 }
```

(b) Model snippet.

Figure 8 Object property pattern.

simple callback. This callback receives as an argument an async object `res`, which has its own asynchronous behavior (lines 168–169); this characterizes the code pattern *callback object*. Finally, function `http.get` returns object `clientRequest` (line 167), which also has asynchronous behavior (line 172), corresponding to code pattern *returned object*. Notice that for function `http.get` and other functions in the built-in modules, two or more code patterns may be present. The proposed model represents those cases with an array that identifies all patterns in attribute `type`, as well as other pattern-specific attributes.

Connected callbacks. I/O operations and CPU-intensive tasks in Node.js applications are treated outside JavaScript code and concurrently by the workers. Therefore, an order among their callbacks cannot be established [11, 6]. Nevertheless, this does not hold with event emitter objects from the core API. The event-driven architecture of Node.js is implemented by special objects called *event emitters*.[10] With such objects, callbacks can be associated with a named event; once an event is emitted, its associated callbacks are scheduled to run.

Figure 12 shows an example of such a case. First, a readable stream object (that is, an event emitter) is created for some JSON file (line 175). Then, callbacks `cbData`, `cbEnd`, and `cbClose` are registered for events `'data'`, `'end'`, and `'close'`, respectively. The readable stream emits several `'data'` events for each chunk of data read from the file, emits `'end'` when there is no more data to consume, and finally emits `'close'` when the underlying file

[10] `https://nodejs.org/api/events.html`

```
const net = require('net')
...
net.createServer(function (socket) {
    socket.on('data', (data) => { ... })
    socket.on('error', () => { ... })
})
```

(a) Example code.

```
{
  "moduleName": "net",
  "properties": [
    {
      "name": "createServer",
      "type": [ "CO" ],
      "postponeAction": false,
      "callbackObjects": [ "net.Socket" ]
    },
    ...
  ]
}
...
{
  "moduleName": "net.Socket",
  ...
}
```

(b) Model snippet.

Figure 9 Callback object pattern.

descriptor is closed. So, callbacks `cbData`, `cbEnd`, and `cbClose` are asynchronous and related to I/O operations, but are ordered in the context of the event emitter object. We call them *connected callbacks*. This case is indicated in the model using the `connectedCallbacks` flag.

A key observation here is that for event emitters from the core API, the order in which its *connected* callbacks are scheduled by Node.js needs to be preserved when inserting the delays. So, we specify such cases in our model, and NACD's instrumentation provides the needed information for the delays being injected while respecting such order. To do so, we use a mechanism based on queues, described in Section 3.2.

Streams. All streams in the Node.js core API are also event emitters. As such, they can draw asynchronous behavior through their event emitters. However, streams may also propagate asynchrony by other means, as illustrated in Figure 13. A readable stream from an XML file is instantiated (line 195) and, using the `pipe` method, a writable stream `parser` (line 196) provided by a third-party library is attached. The callbacks in lines 197–198 are from `parser`, but are also asynchronous since they react based on the readable stream. To handle such cases, NACD injects delays in modeled functions of the stream objects so that different callback interleavings may be explored even in the presence of streams. In this example, the model represents the readable stream's `_read` function that fetches data from the underlying resource.

3.2 Runtime System

Using the model of the asynchronous behavior of the Node.js API, the NACD runtime system installs hooks in the asynchronous operations that we intend to delay. Such hooks, named `onRun`, are invoked right before an asynchronous operation is about to run; at this moment,

```
const { resolve } = require('dns').promises
...
// example 1
resolve('example.org')
    .then((records) => { ... })

// example 2
async function () {
    ...
    const records = await resolve('example.org')
    ...
};
```

(a) Example code.

```
{
  "moduleName": "dns",
  "properties": [
    {
      "name": "promises",
      "type": [ "OP" ],
      "AsyncObjectProperty": "dns.Promises"
    },
    ...
  ]
},
...
{
  "moduleName": "dns.Promises",
  "properties": [
    {
      "name": "resolve",
      "type": [ "RP" ]
    },
    ...
  ]
}
```

(b) Model snippet.

Figure 10 Returned promise pattern.

NACD may decide to inject a delay. This step occurs at runtime and on-demand, as NACD intercepts the Node.js application's accesses to the core API. The implementation details are presented in Section 4.

The model specified in Section 3.1 defines which functions and objects are tracked, providing the needed information to install `onRun` hooks. Algorithm 1 illustrates what those hooks look like. As discussed, function `onRun` is invoked for each asynchronous operation NACD tracks. The first argument `op` is an object with all the needed information to run the asynchronous operation; in most cases, `op` refers to a callback, but may also refer to a promise (see pattern *returned promise*). The second argument `connected` is a Boolean value that specifies whether or not `op` is a connected callback. If `connected` is true, then the third argument `objectID` comes with an integer that uniquely identifies the object with which `op` may have other connected callbacks. During the instrumentation, NACD uses the model's information to identify connected callbacks, and it keeps track of instantiation of async objects so that unique IDs (`objectID`) are correctly assigned. In line 200, NACD checks if `op` is connected and invokes either `decideSimpleDelay` or `decideConnectedDelay`. These two cases are explained as follows.

```
165 const http = require('http')
166
167 const clientRequest = http.get('http://some.site/file.json', (res) => {
168    res.on('data', (chunk) => { ... })
169    res.on('end', () => { ... })
170 })
171
172 clientRequest.on('close', () => { ... })
```

Figure 11 Example of combining different code patterns.

```
173 const fs = require('fs')
174
175 const readStream = fs.createReadStream('some_file.json')
176 let rawData = ''
177 let pkg
178
179 readStream.on('data', function cbData(chunk) {
180    rawData += chunk
181 })
182
183 readStream.on('end', function cbEnd() {
184    pkg = JSON.parse(rawData)
185 })
186
187 readStream.on('close', function cbClose() {
188    useObject(pkg)
189 })
```

Figure 12 Example of connected callbacks.

Simple delays. This is the simple case where the delay NACD applies does not depend on any other operation. Algorithm 2 describes how NACD decides whether or not to delay an asynchronous operation `op`. Function `decideSimpleDelay` is called when an asynchronous operation `op` is about to run. First, function `makeChoice` makes random choices and returns a Boolean variable (`delay`) and an integer (`timeout`) (line 206). If `delay` is true (line 207), `op` is delayed for `timeout` miliseconds (line 208); otherwise, `op` is run immediately (line 210). We discuss how the random choices of function `makeChoice` are implemented in Section 4.

In NACD, this delay mechanism is applied in three different operations:

1. Callback: The async callback is completely independent; as such, any delay injected in it should not interfere with the execution, apart from delaying it.
2. Registration: In some cases we modeled, it is possible to delay the registration (start) of an asynchronous task. For instance, when deleting a file, NACD delays when this operation is actually run, not its callback. This may reveal race conditions outside of Node.js. These cases are marked with `postponeAction` in the model.
3. Promise: In this case, the returned promise has its fullfilment delayed. So, we simulate the case where the asynchronous task takes a longer time to complete.

Connected callbacks. We now describe how to inject delays while preserving the order of connected callbacks. In Algorithm 3, function `decideConnectedDelay` is invoked with callback `cb` and `objectID` that uniquely identifies the object with which `cb` has other connected callbacks. As this function is called before `cb` occurs, NACD labels it as not scheduled (line 213), retrieves a queue, or creates one if does not exist, identified by `objectID` (line 214). The retrieved queue is referred to as `q`. Then, it pushes `cb` to the end of this queue (line 215), and calls function `scheduleFirstOf` passing queue `q` as argument (line 216).

```
190 const fs = require('fs')
191 const expat = require('node-expat')
192
193 const parser = new expat.Parser('UTF-8')
194
195 fs.createReadStream('some.xml')
196   .pipe(parser)
197     .on('startElement', (name, attrs) => { ... })
198     .on('endElement', (name) => { ... })
```

Figure 13 Piping streams.

Algorithm 1 Function `onRun`.

```
199 Function onRun(op, connected, objectID)
200     if ¬ connected then
201         decideSimpleDelay(op)
202     else
203         decideConnectedDelay(op, objectID)
204     end
```

Function `scheduleFirstOf` is defined in lines 218–231. Initially, line 219 starts a loop that repeats while queue `q` is not empty. Within the loop, it peeks the first element `cb` of the queue, without removing it (line 220). If `cb` is not scheduled yet (line 221), NACD starts the scheduling and delay injection process. First, `cb` is labelled as scheduled so that this occurs only once (line 222). Similarly to function `decideSimpleDelay`, NACD decides to delay or not callback `cb` (lines 223–228). The difference here is that line 229 is run right after `cb` is actually run (delayed or not). So, `cb` is dropped from the queue (line 229), and the next callback (if exists) is processed in the next iteration of the queue loop (line 219). The idea is to push callbacks that share the same `objectID` to the same queue, while scheduling the first element to run and removing it when it is actually run. For instance, if NACD decides to delay the execution of a callback `c1`, the call to `scheduleFirstOf` will be waiting in line 225. Meanwhile, function `decideConnectedDelay` may be invoked again for a connected callback `c2`, which is then pushed to the same queue (line 215), followed by another call to `scheduleFirstOf` (line 216). At this point, NACD sees that the first element in the queue (i.e., `c1`) has already been scheduled but not yet executed, taking no action (false branch in line 221). When the waiting period in line 225 ends and `c1` is actually executed, `c1` is removed from the queue (line 229). Since `c2` remains in the queue, another loop iteration (line 219) occurs to process `c2`. This mechanism ensures that the queue preserves the callback ordering originally defined by the Node.js runtime, while NACD's function `scheduleFirstOf` delays callbacks, one at a time, without disrupting the queue order.

4 Implementation

NACD is implemented as a Node.js Command Line Interface (CLI) tool, with its main modules comprising approximately 1.3 KLoC written in JavaScript. The tool takes an entry script as input, which runs a Node.js application with potential event races. Entry scripts may be automated tests for the Node.js application being analyzed. We have successfully used NACD with automated tests created with well-known testing frameworks, such as Mocha, Jest, Jasmine, and Karma.

Algorithm 2 Function `decideSimpleDelay`.

```
205 Function decideSimpleDelay(op)
206 |  {delay, timeout} ← makeChoice()
207 |  if delay then
208 |  |  runAfterDelay(op, timeout)
209 |  else
210 |  |  runImmediately(op)
211 |  end
```

Algorithm 3 Function `decideConnectedDelay`.

```
212 Function decideConnectedDelay(cb, objectID)
213 |  cb.scheduled ← false
214 |  q ← getQueue(objectID)
215 |  q.enqueue(cb)
216 |  scheduleFirstOf(q)
217
218 Function scheduleFirstOf(q)
219 |  while q.isNotEmpty() do
220 |  |  cb ← q.peekFirst()
221 |  |  if ¬ cb.scheduled then
222 |  |  |  cb.scheduled ← true
223 |  |  |  {delay, timeout} ← makeChoice()
224 |  |  |  if delay then
225 |  |  |  |  runAfterDelay(cb, timeout)
226 |  |  |  else
227 |  |  |  |  runImmediately(cb)
228 |  |  |  end
229 |  |  |  q.removeFirst()
230 |  |  end
231 |  end
```

As mentioned in Section 3.1, the model of the Node.js Asynchronous API is stored in JSON files. To identify the core API that triggers asynchronous operations, we manually inspected the Node.js documentation,[11] source code,[12] and type definitions.[13] This research was conducted based on Node.js v.10. To avoid bias, the benchmarks in Section 5 were not used for this task. At the time of writing, the NACD's model includes 19 JSON files with around 2.2 KLoC, representing 46 async classes and 424 properties.

The runtime system of NACD modifies the Node.js module system to intercept module imports made by the application under test. When a module is imported, NACD checks its internal model and, if the module is one of interest, it injects specific hooks for the async classes and properties (the `onRun` hook shown in Section 3.2). To realize this functionality,

[11] `https://nodejs.org/api`
[12] `https://github.com/nodejs/node`
[13] `https://definitelytyped.org`

we leverage JavaScript's built-in Proxy[14] and Reflect[15] APIs. This design avoids the need for the fine-grained instrumentation techniques used in tools like NodeRacer, NodeRT, and NRace. Function `makeChoice` of Section 3.2 introduces a 50/50 probability of injecting a random delay between 0 and 500 miliseconds. NACD only instruments calls to the Node.js core API; it requires no modeling of third-party packages.

5 Evaluation

The evaluation of the proposed approach is based on the following research questions:

- **RQ1** To what extent is NACD capable of reproducing race bugs in Node.js applications?
- **RQ2** How many runs does NACD take to reveal a race bug?
- **RQ3** What is the overhead imposed by NACD?

These questions intend to analyze different aspects of NACD in comparison with similar state-of-the-art tools available for race detection in Node.js applications. **RQ1** aims to assess fault detection capabilities by analyzing how often the tools can reproduce known race bugs. In **RQ2**, we focus on how many runs each tool would take to first reveal a race bug. Finally, **RQ3** targets the runtime overhead observed when executing the race detection tools.

5.1 Experimental Setting

To answer **RQ1**, we made an experimental comparison of NACD with state-of-the-art tools Node.fz [9] and NodeRacer [11]. To do so, we reused the experimental package with 24 benchmarks provided by NodeRacer [11]. Each benchmark is a specific version of an open source Node.js application with a race bug, and an automated test that passes in ordinary runs (by using vanilla Node.js) and fails when the race bug is explored. Node.fz and NodeRacer were run with their default settings [9, 11]. As in [9, 11], we measured how many times a tool reveals the bug in 100 runs, i.e., bug reproduction ratio. The three tools make random decisions, so this step was repeated 30 times and median values with median absolute deviation (MAD) were calculated for the bug reproduction ratio of each tool, per benchmark. Using the same setup for **RQ2**, we measured the number of runs until the test first fails. It is desirable for a tool to reveal a bug with the fewest number of runs possible. In **RQ3**, we collected runtime data for the tools across 100 runs using GNU Time. We refer to the wall clock runtime (the "real" output of GNU Time) as *elapsed time*, and the actual CPU time consumed during execution without delays (the sum of "user" and "sys" outputs from GNU Time) as *CPU time*. To compute the overhead for these two metrics, we use benchmark data gathered from runs with vanilla Node.js as baseline. Analyzing overhead costs alone may fail to take the bug detection capabilities of each tool into account. For this reason, we also compute the overhead divided by the bug reproduction ratio collected for RQ1.

The development, tests, and experiments were conducted on a machine with an Intel i7-1260P (16 cores), 32GB of RAM, running Ubuntu 22.04. The NACD implementation and the experimental package are available at:

`https://github.com/andreendo/nacd`

[14] `https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Proxy`
[15] `https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Reflect`

5.2 Analysis of Results

RQ1 – Bug Reproduction. Table 1 summarizes the comparison of NACD with Node.fz and NodeRacer. The first four columns characterize the benchmark: its ID, the project name, the GitHub issue ID that describes the race bug, and the number of lines of JavaScript code (LoC[16]). The next three columns show the results for each tool, each cell contains the median value (and its MAD between parentheses) of the bug reproduction ratio, i.e., how many times the tool is capable of revealing the bug in 100 runs. Overall, the three tools presented small variations in the results, as their MADs were at most 4% for Node.fz, 6% for NodeRacer, and 5% for NACD.

Node.fz was capable of running in only 8 benchmarks. This is because Node.fz was built on top of an early version of Node.js, so benchmarks adopting newer features are not compatible. For the 8 benchmarks, Node.fz's bug reproduction ratios were smaller than NodeRacer and NACD.

NodeRacer and NACD were compatible with all benchmarks. Benchmark #24 is a special case, as it has no race bug. Nevertheless, NodeRacer sometimes schedules the event handlers in a way that is impossible in ordinary executions, causing test failure to be mistakenly reported (i.e., false alarms) in 75% of the runs. We included the benchmark in the RQ1 experiments to show that NACD is not subjected to the same issue, as the test passed in all runs (ratio 0%). For the following discussion, the analyses are based on the 23 remaining benchmarks.

NACD had better bug reproduction ratio in 18 benchmarks (78%), in which it increased the ratio by on average +36% relative to NodeRacer. The improvements varied from +6% for Benchmark #3 to up to +79% in Benchmark #19. Proportionally, the greatest improvements occurred for Benchmarks #21 and #22. In both cases, NodeRacer would take 100 runs to expose the bug, while around 3 runs would be enough for NACD.

On the other hand, NodeRacer had a better performance in 5 benchmarks (22%), in which NACD had decreased the ratio by on average −28% when comparing to NodeRacer. The major difference was for Benchmark #11. NACD had a median bug reproduction ratio of 0% (though it had a ratio 1% in two of the 30 repetitions). This means that NACD could not reveal this race bug consistently. This occurred due to a file stream that is piped to another transform stream, a corner case for NACD (we discuss this limitation in Section 6). Yet, it is possible to extend NACD with a model of the library involved in the async behavior. With this extension, NACD is able to reproduce the bug consistently, in 100% of the runs. For the remaining 4 benchmarks, we manually inspected the code, logs and supporting artifacts generated by both tools, but found no discernible pattern. This may be explained by the fact that NACD and NodeRacer employ strategies that are fundamentally different, along with the random elements in their designs. Those factors introduce sufficient variation even in the same computing environment, as reflected in the MADs shown in Table 1.

Table 1 partially reproduces the bug reproduction experiments in [11], concerning Benchmarks #1–#11. For Node.fz, the results are essentially equal for 4 benchmarks (#2, #4, #6, #10), slightly better for #1 (from 8% to 11%), and slightly worse for #3 (from 26% to 15%) and #7 (from 1% to 0%). As for NodeRacer, the results are essentially equal for 2 benchmarks (#2, #6), slightly worse in 6 cases (with decreases varying from −2% to −6%), and better for Benchmarks #3 (+1%), #9 (+16%), and #10 (+19%). Overall, the ratios are similar with minor variations, though we recognize that the tool designs and the computing environment do have an impact.

[16] Data collected with the cloc tool.

Table 1 Bug reproduction ratio.

Benchmarks				Tools		
ID	Project Name	Issue[1]	LoC	Node.fz[2]	NodeRacer	NACD
#1	*agentkeepalive*	23	1.8K	11% (±2%)	57% (±3%)	71% (±3%)
#2	*fiware-pep-steelskin*	269	6.1K	0% (±0%)	48% (±2%)	87% (±2%)
#3	*Ghost*	1834	30K	15% (±2%)	92% (±1%)	98% (±1%)
#4	*node-mkdirp*	2	0.2K	0% (±0%)	46% (±5%)	93% (±2%)
#5	*nes*	18	3.4K	-	54% (±5%)	100% (±0%)
#6	*node-logger-file*	1	0.9K	9% (±2%)	84% (±2%)	76% (±4%)
#7	*socket.io*	1862	2.4K	0% (±0%)	17% (±2%)	84% (±2%)
#8	*del*	43	0.2K	-	20% (±2%)	41% (±5%)
#9	*linter-stylint*	63	0.2K	-	44% (±5%)	57% (±4%)
#10	*node-simplecrawler*	298	3.9K	0% (±0%)	83% (±2%)	64% (±5%)
#11	*xlsx-extract*	7	1K	-	53% (±5%)	0% (±0%)
#12	*get-port*	23	0.4K	-	49% (±5%)	13% (±2%)
#13	*live-server*	262	0.9K	-	19% (±2%)	49% (±4%)
#14	*bluebird*	1417	13.4K	20% (±4%)	71% (±6%)	84% (±2%)
#15	*express*	3536	1.8K	-	63% (±2%)	87% (±2%)
#16	*socket.io-client*	3358	1.5K	-	49% (±5%)	26% (±3%)
#17	*mongo-express-1*	499	2.9K	-	23% (±2%)	90% (±2%)
#18	*mongo-express-2*	499‡	2.9K	-	61% (±5%)	91% (±2%)
#19	*mongo-express-3*	500	2.9K	-	5% (±2%)	84% (±2%)
#20	*mongo-express-4*	500‡	2.9K	-	6% (±2%)	83% (±4%)
#21	*nedb-1*	610	7.3K	-	1% (±1%)	36% (±2%)
#22	*nedb-2*	610‡	7.3K	-	1% (±1%)	39% (±5%)
#23	*node-archiver*	388	1.3K	-	1% (±1%)	9% (±2%)
#24	*objection.js*	(*)	55.9K	-	75% (±24%)	0% (±0%)

[1]The issue IDs have links to the corresponding GitHub pages.

[2]Node.fz did not run with benchmarks marked with '-', as it is based on an outdated version of Node.js.

‡This bug was reported in a repeated issue, but is a different case (flaky test).

*This benchmark has no issue, reported as a false alarm in [11].

Response to RQ1: NACD is capable of uncovering race bugs without false alarms, being on average more effective than state-of-the-art tools Node.fz and NodeRacer. Particularly, NACD had the best bug reproduction ratio in 78% of the benchmarks, with improvements from 6% to up to 79%.

RQ2 – Number of runs until first failure. We also analyze the number of runs required for the race bug to first manifest (i.e., the test fails). This provides insight on how quickly each tool can uncover potential race bugs. Table 2 summarizes these results for each benchmark. Under each tool (columns 2–4), each cell shows the median number of runs until the first failure (and its MAD between parentheses). Similar to RQ1, Node.fz had results inferior to those of the others, and failed to reveal the bugs in Benchmarks #2, #4, and #10. NACD and NodeRacer had similar performance in 7 cases, where a single run was typically sufficient to uncover the bug. NACD had the best results in 12 cases, whereas NodeRacer outperformed it in 4 cases (#9, #11, #12, and #16). In Benchmarks #11, #12, and #16, NodeRacer also had a higher bug reproduction ratio, while NACD had a better ratio for Benchmark #9.

Table 2 Runs required for the bug to first manifest.

Benchmark ID	Node.fz[1]	NodeRacer	NACD
#1	6.5 (±5.1)	1.0 (±0.0)	1.0 (±0.0)
#2	**	2.0 (±1.4)	1.0 (±0.0)
#3	6.0 (±5.1)	1.0 (±0.0)	1.0 (±0.0)
#4	**	2.0 (±1.4)	1.0 (±0.0)
#5	-	2.0 (±1.4)	1.0 (±0.0)
#6	9.5 (±7.4)	1.0 (±0.0)	1.0 (±0.0)
#7	64.0 (±35.5)	4.5 (±3.7)	1.0 (±0.0)
#8	-	3.0 (±2.9)	2.0 (±1.4)
#9	-	1.5 (±0.7)	2.0 (±1.4)
#10	**	1.0 (±0.0)	1.0 (±0.0)
#11	-	2.0 (±1.4)	42.0 (±28.2)
#12	-	1.0 (±0.0)	6.5 (±5.1)
#13	-	4.5 (±2.2)	1.0 (±0.0)
#14	3.5 (±3.7)	1.0 (±0.0)	1.0 (±0.0)
#15	-	1.0 (±0.0)	1.0 (±0.0)
#16	-	1.0 (±0.0)	3.0 (±2.2)
#17	-	4.5 (±2.2)	1.0 (±0.0)
#18	-	1.0 (±0.0)	1.0 (±0.0)
#19	-	10.0 (±11.8)	1.0 (±0.0)
#20	-	12.5 (±11.1)	1.0 (±0.0)
#21	-	44.5 (±38.5)	3.5 (±3.7)
#22	-	39.0 (±26.6)	2.0 (±1.4)
#23	-	41.0 (±31.8)	9.0 (±8.1)

[1]Node.fz did not run with benchmarks marked with '-'.

**The bug was not revealed in 100 runs.

Observe in Table 2 that a higher median value is associated with greater variation (MAD). This suggests that results tend to vary more when the tool requires more runs to initially detect a bug. Also, notice that the bug reproduction ratio (Table 1) appears to be negatively correlated with the number of runs. For instance, ratios exceeding 50% are associated with fewer than 2 runs. On the other hand, lower ratios (close to 1%) require more than 40 runs to reveal the bug for the first time.

By aggregating the results of the 23 benchmarks, across all runs in applicable benchmarks, NACD revealed the bug within the first 25 runs in 95.7% of cases, compared to 87.3% for NodeRacer and 48.3% for Node.fz. Figure 14 shows the aggregated number of runs taken to uncover the bug, with the data limited to the first 25 runs for clarity. For each tool, it brings the boxplot, its distribution using violin plot, and the mean represented by a blue X. On average, Node.fz took 13.1 runs (median 7), its mean is out of the interquartile range due to outliers greater than 25. Next, NodeRacer needed fewer runs, 7.8 on average (median 2). It also had outliers to went up to 100 runs, making the furthest mean from the interquartile range. This wide range is due to NodeRacer having more benchmarks with values exceeding 3 than NACD, specifically the high values in Benchmarks #19–#23 (Table 2). NACD outperformed the previous tools, as it took on average 2.5 runs (median 1) to uncover

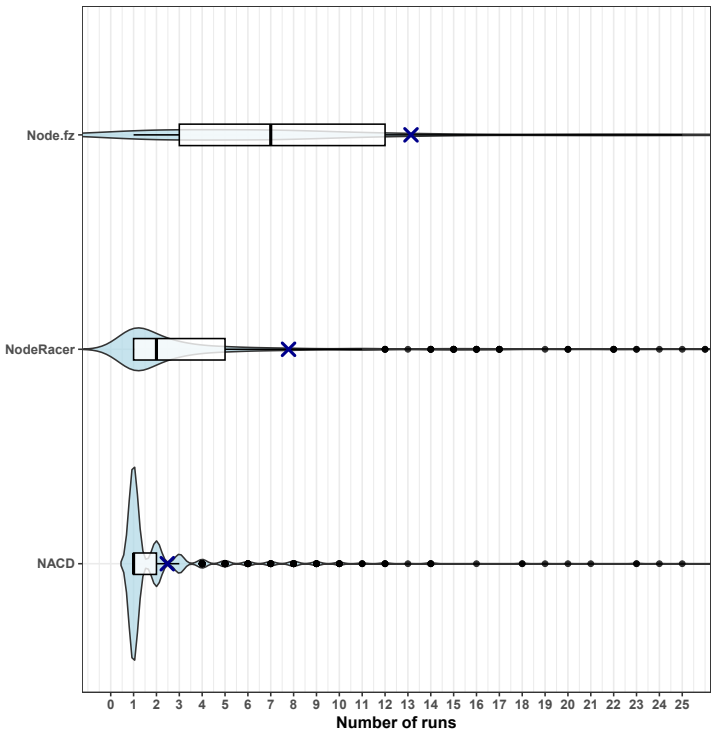


Figure 14 Runs required for the bug to first manifest.

the race bug. Its mean is the one closest to the interquartile range, since most data is concentrated on range 1–2, and only one outlier is above 25 (in one run for Benchmark #11, NACD took 61 runs).

Response to RQ2: To trigger the first failure, NACD performed as well as or better in 82.6% of the benchmarks, while NodeRacer achieved similar or better results in 47.8%. When considering the aggregated results across all 23 benchmarks, NACD can provoke the first failure faster than other state-of-the-art tools, taking an average of 2.5 runs.

RQ3 – Overhead. Using vanilla Node.js as a reference, the elapsed time for 100 runs across all benchmarks was on average 121.1 seconds, ranging from 64.9s in Benchmark #8 to 452.8s in Benchmark #13. For CPU time, the average was 29.8 seconds, ranging from 13.7s in Benchmark #4 to 51.7s in Benchmark #13. Table 3 summarizes the overhead-related results.

By aggregating the results of the applicable benchmarks, Figure 15 shows the overhead introduced by the tools with respect to executions with vanilla Node.js. As for the *elapsed time* observed in Figure 15a and listed in columns 2–4 of Table 3, Node.fz imposes the smallest overhead (median: 1.0x), followed by NodeRacer (median: 2.6x), and NACD (median: 3.7x). One reason for NACD's higher elapsed time is that it explores more opportunities to inject delays. Additionally, some benchmarks manifest race bugs through timeouts and hangs (observed in higher overhead of Benchmarks #2, #6, #7, #11, and #21), so NACD's high bug reproduction ratio (observed in RQ1) has a negative impact on this metric.

Concerning *CPU time* shown in Figure 15b and listed in columns 5–7 of Table 3, NACD generates the least overhead (median: 2.0x), followed by NodeRacer (median: 2.3x) and Node.fz (median: 3.2x). Surprisingly, Node.fz consumed more CPU time, as half of its

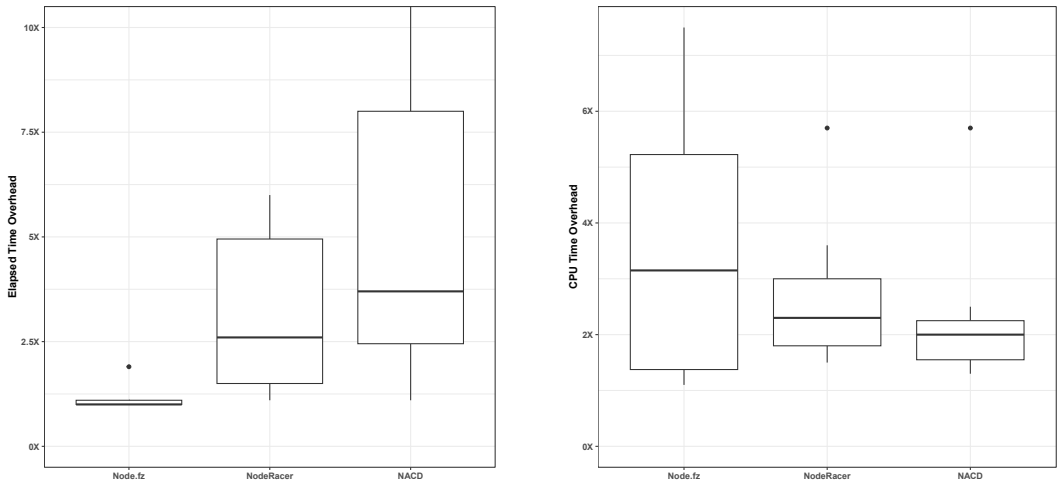
Table 3 Overhead results.

	Overhead						Overhead / Bug Ratio					
	Elapsed Time			CPU Time			Elapsed Time			CPU Time		
Ben. ID	*Node.fz*	*NodeRacer*	*NACD*	*Node.fz*	*NodeRacer*	*NACD*	*Node.fz*	*NodeRacer*	*NACD*	*Node.fz*	*NodeRacer*	*NACD*
#1	1.0	1.7	1.4	5.2	2.5	2.2	0.090	0.030	0.020	0.472	0.044	0.031
#2	1.1	19.3	102.8	1.4	3.2	2.3	**	0.402	1.182	**	0.067	0.026
#3	1.1	3.2	3.0	1.1	1.8	2.0	0.073	0.035	0.031	0.073	0.020	0.020
#4	1.0	2.5	3.0	1.8	2.3	2.1	**	0.054	0.032	**	0.050	0.023
#5	-	1.9	2.1	-	1.9	1.6	-	0.035	0.021	-	0.035	0.016
#6	1.9	13.8	22.5	5.3	2.5	2.2	0.211	0.164	0.296	0.588	0.030	0.029
#7	1.0	6.0	22.6	1.3	2.4	2.2	**	0.353	0.269	**	0.141	0.026
#8	-	3.6	3.8	-	3.2	2.0	-	0.180	0.093	-	0.160	0.049
#9	-	1.1	1.1	-	2.2	2.4	-	0.025	0.019	-	0.050	0.042
#10	1.0	1.4	2.8	4.5	2.5	2.1	**	0.017	0.044	**	0.030	0.033
#11	-	2.6	18.0	-	2.0	37.1	-	0.049	**	-	0.038	**
#12	-	2.7	2.3	-	1.8	1.7	-	0.055	0.177	-	0.037	0.131
#13	-	1.6	3.9	-	5.7	5.7	-	0.084	0.080	-	0.300	0.116
#14	1.0	1.3	1.1	7.5	3.6	1.9	0.500	0.018	0.013	3.750	0.051	0.023
#15	-	1.4	2.6	-	1.5	1.5	-	0.022	0.030	-	0.024	0.017
#16	-	3.4	2.3	-	3.5	1.9	-	0.069	0.088	-	0.071	0.073
#17	-	5.5	3.7	-	1.8	1.3	-	0.239	0.041	-	0.078	0.014
#18	-	5.0	3.8	-	1.8	1.4	-	0.082	0.042	-	0.030	0.015
#19	-	5.2	3.8	-	1.8	1.4	-	1.040	0.045	-	0.360	0.017
#20	-	4.9	3.9	-	1.8	1.4	-	0.817	0.047	-	0.300	0.017
#21	-	1.4	15.7	-	2.9	2.5	-	1.400	0.436	-	2.900	0.069
#22	-	1.4	12.1	-	3.1	2.5	-	1.400	0.310	-	3.100	0.064
#23	-	1.9	3.3	-	1.9	1.5	-	1.900	0.367	-	1.900	0.167

**It is not calculated since the bug reproduction ratio is 0.

benchmarks exhibited overheads exceeding 4.5x (Benchmark #10), moving the median upward. We surmise that since Node.fz is based on an outdated version of Node.js, it lacks optimizations introduced in newer releases, which affects certain benchmarks. Since NACD employs lightweight instrumentation and does not perform happens-before computations, it achieves lower CPU time overhead than NodeRacer in 19 out of the 23 benchmarks (82.6%). For the remaining 17.4%, NodeRacer was marginally better in Benchmarks #3, #9, and #13, while NACD exhibited a significant overhead of 37.1x in Benchmark #11, a limitation further discussed in Section 6.

By aggregating the results of the applicable benchmarks, Figure 16 illustrates the overhead related to the bug reproduction ratio. This provides a clearer perspective to the cost-effectiveness relation by examining the overhead through the lens of the tools' bug reproduction capabilities. As for the *elapsed time per bug reproduction ratio* shown in Figure 16a and listed in columns 8–10 of Table 3, NACD exhibits the smallest overhead (median: 0.05x), followed by NodeRacer (median: 0.08x), and Node.fz (median: 0.15x). Although there is a reasonable overlap in the interquartile ranges of the 3 tools, the NACD's range is below 0.25x



(a) Elapsed time. **(b)** CPU time.

Figure 15 Overhead introduced by the tools (measured relative to vanilla Node.js).

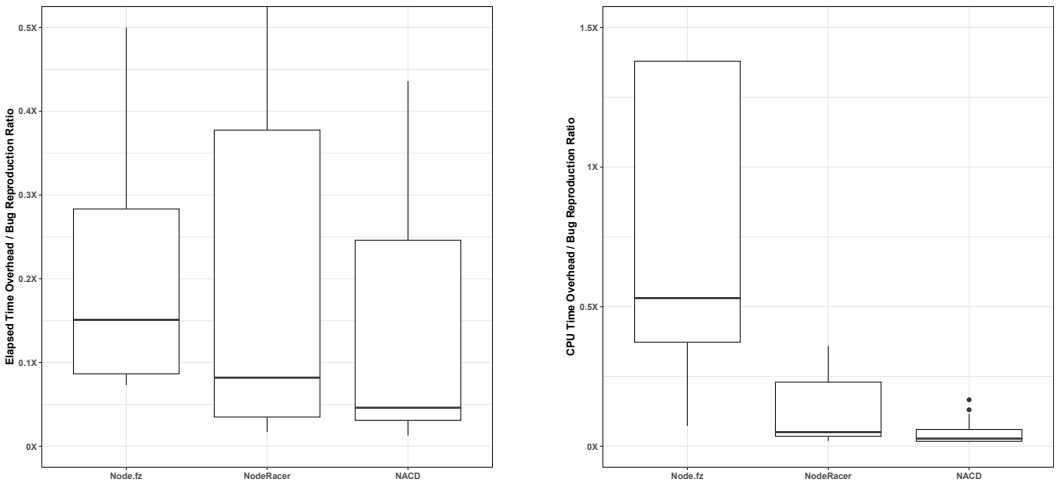
and its median is the lowest among the tools. In addition, NACD achieves the lowest overhead in 16 out of 23 benchmarks. Among the 7 benchmarks in which NodeRacer outperformed (#2, #6, #10, #11, #12, #15, and #16), it also had a higher bug reproduction ratio than NACD in 5 of them. In Benchmarks #2 and #15, NodeRacer exhibited lower overhead in terms of elapsed time (see columns 2–4 in Table 3).

Concerning *CPU time per bug reproduction ratio* presented in Figure 16b and listed in columns 11–13 of Table 3, NACD again has the lowest overhead (median: 0.03x), followed by NodeRacer (median: 0.05x) and Node.fz (median: 0.53x). Among the tools, NACD also exhibits the least data dispersion, as point out by its smallest interquartile range. Furthermore, NACD achieves the lowest overhead in 18 out of 23 benchmarks. Among the 5 benchmarks where NodeRacer performed better (#3, #10, #11, #12, #16), it also achieved the higher bug reproduction ratio in 4 of them. The improved performance in Benchmark #3 is due to NodeRacer's lower CPU time overhead (1.8x) compared to NACD's (2.0x) (see columns 5–7 in Table 3).

Response to RQ3: Node.fz and NodeRacer exhibit lower elapsed time overhead compared to NACD, while NACD consumes lower CPU time in most benchmarks. When considering overhead in relation to bug detection, NACD presents significantly better performance in terms of both elapsed time and CPU time for most benchmarks.

5.3 Threats to Validity

We here discuss threats to the validity of the experimental results. The implementation may be subjected to potential flaws, so we took several steps to minimize this threat. NACD was implemented with a logging feature so that all actions performed are logged for post-mortem analyses. Using this feature, we tested various examples across different functions within the Node.js core modules. Additionally, NACD comes with a suite of automated tests to verify the behavior of its classes.



(a) Elapsed time.

(b) CPU time.

Figure 16 Overhead in relation to bug reproduction ratio.

Node.fz and NodeRacer allow parameter adjustments that may alter their behavior. We adopted their default configuration, without any parameter fine-tuning. We believe this is a fair approach as most practitioners will likely use the out-of-the-box tools, and fine-tuning requires some expertise. Nevertheless, different configurations may yield varying results. For example, Node.fz could be parameterized to achieve better results under certain hypothetical circumstances. However, this is a challenging task, as it must be performed for each benchmark, and Node.fz has over 10 parameters that control complex operations within the Node.js runtime. All tools are subjected to randomness and may impact the results. This risk is mitigated by repeating the data collection for each pair tool-benchmark 30 times.

The benchmarks used in our experiments may not be fully representative, and results may not generalize in different contexts. We utilized the benchmarks provided by the NodeRacer study, which include race bugs from previous studies of Davis et al. [9] and Wang et al. [27], as well as additional bugs extracted from open source projects. All benchmarks are based on real-world Node.js projects, with corresponding issues documenting the race conditions.

We anticipate that the computing environment in which the experiments are run may also have some impact. To assist in replicating the results in future research, we have provided an experimental package, along with a detailed description of environment (e.g., computer configurations, software versions, and a Docker file).

6 Discussion

Evolution of Node.js core API. Node.js is an active distributed development project facilitated by the OpenJS foundation. As such, the platform has been actively evolved and the core API is subjected to modifications. As NACD relies on a model of the core API, the model itself needs to be updated to keep track of advances in the Node.js platform. Currently, this task can be performed manually by updating the NACD JSON model of the API. While inspecting the documentation, we simultaneously identified code patterns and developed the proposed approach. As a result, we do not have an accurate estimate of the effort spent building the current model. However, we predict that, once all code patterns are known, modeling each relevant function would take approximately 1–2 minutes. For this task, it

would be worthwhile exploring automation. A principled way could involve dynamic analysis to inspect the objects returned by the core API. We also anticipate that LLMs with access to up-to-date documentation may assist in this task.

Timers and scheduling functions. Node.js has several functions to define timers and schedule future tasks (e.g., `setTimeout`, `setImmediate`, `setInterval`, `process.nextTick`). These functions introduce asynchronous behavior and may involve complex happens-before relations [11, 30, 7], but we intentionally left them out of NACD to avoid this complexity. While event races may occur exclusively among them, most are in some way related to some I/O operation [27]. In the benchmarks, we did not miss any event race due to this design decision.

No observation phase. NACD starts to explore callback interleavings in the first run, as it does not require an observation phase like NodeRacer. This feature avoids false positives due to non-deterministic test setup (as in Benchmark #24), since NACD will inject delays only based on the current execution without querying happens-before relations built in a previous run. The absence of an observation phase also makes NACD more useful for executions that are hard to reproduce, such as a performance test that triggers multiple requests.

Limitation to handle pipe streams. As noted in Section 5, NACD was unable to consistently reproduce the event race in Benchmark #11. The race originates from the following context: there is a stream that reads data from a compressed (zip) file. This stream is piped into a transform stream, provided by the `unzip2` package, which splits the data and emits specific events for each XML file within the zip file. Each XML file is then parsed using the event emitter API of the `node-expat` package, where the event race occurs.

NACD can inject delays into the execution of the zip file stream, as the benchmark code uses the API provided by Node.js built-in module `fs`. Under normal circumstances, these delays help explore subsequent callback intervealings, even when the stream is piped to other streams. However, in this case, the `unzip2` transform stream emits an event only after all data chunks for an XML file have been read. This behavior prevents the injected delays from propagating to subsequent streams. In other words, the code being tested cancels the injected delays beyond that point, which afterwards effectively behave just like vanilla Node.js. A potential solution is to include the involved third-party packages in NACD's model. With this approach, we were able to reproduce the race bug consistently.

Asynchronous behavior out of the core API. In the Node.js and npm ecosystem, there are packages that work as wrappers for code in C/C++ or other programming languages. Asynchronous behavior may come from such a third-party code, and not from the Node.js built-in modules. In such situations, NACD would not know about it and no delay would be injected. While we did not observe this case in the benchmarks, this may occur in practice. To handle it, the developer would need to extend NACD's model by including information about the functions and classes that have asynchronous behavior in such packages.

7 Related Work

Race detection in JavaScript applications. The particularities of the single-threaded asynchronous model of JavaScript applications have been widely investigated to enhance programming tools and environments [29, 16, 17, 24, 3, 4, 5, 19, 26]. Specifically, the literature on race detection for JavaScript applications can be categorized into two main groups: client-side and server-side.

Race detection in client-side applications has been extensively explored. Tools like WebRacer [21] and EventRacer [22] leverage the WebKit browser framework to collect dynamic information and apply predictive techniques based on happens-before (HB) relations to detect event races. Both tools implement filtering mechanisms to minimize the detection of harmless races. To better identify harmful races, WAVE [13] and R4 [14] employ mechanisms to obtain observable manifestations, known as witness runs, that characterize such races. Similarly, RClassify [28], InitRacer [2], and AjaxRacer [1] aim to generate witness runs but rely on JavaScript code instrumentation instead of browser modifications.

In general, race detectors for client-side JavaScript applications need to simulate user actions, handle browser-specific API and integrated technologies like HTML and CSS, and adopt oracle mechanisms to flag harmful races. In contrast, NACD is designed for server-side applications in Node.js and does not adopt HB relations; yet, it aims for a witness run typically in the form of a failed test. Notably, many of those tools assume that the application under test lacks an automated test suite. However, with the widespread use of end-to-end (E2E) web testing in industry [15], we surmise that an approach with a design similar to NACD could be developed to detect race bugs also in client-side JavaScript applications.

In contrast to client-side tools, race detectors for server-side JavaScript start with an existing test or script that runs the application with potential event races. This is a plausible requirement since test suites are common in Node.js application development.

Given a test, some approaches rely on predictive strategies that, in general, observe the test execution as a reference run, reason about it using HB relations and heuristics, and report likely event races [6, 7, 30]. NodeAV [6] is the first initiative and targets violations on event groups that are supposed to be processed together but are not due to a race, i.e., atomicity violations. To obtain an execution trace, NodeAV initially instruments the application using Node.js experimental API Async Hooks along with the dynamic analysis framework Jalangi [23] to track reads and writes of memory locations and files. Using the trace, it establishes HB relations and the violation detection occurs by inferring atomicity intentions.

Differently from NodeAV, NRace [7] does not focus on a specific type of event race. Its detection method is based on conflicting operations that access memory or files and relies on optimizations to construct the HB graph faster than previous work. Nevertheless, its design is similar to NodeAV as it also adopts the Async Hooks API and Jalangi. NRace also applies some pattern-based heuristics to detect potential benign races and prune them.

NodeRT [30] advances the HB graph construction and race detection in order to reduce the overhead with respect to NRace. To do so, it simplifies the existing HB relation rules, and a partial HB-graph is built while the trace collection is performed. It also uses the Async Hooks API, but the instrumentation is performed using NodeProf [25].[17] The tool implements some matching rules that identify and remove race candidates that are false positives.

Although efforts have been made to reduce false positives, these predictive detectors still flag a substantial number of harmless races [30]. This is undesirable in practice because it often demands considerable developer effort to debug the code. Even worse, there is a high likelihood that the flagged issue is a harmless event race, leading to wasted resources. These tools primarily focus on the detection at the application code level, which leaves potential races that occur within library code or in its interaction with application code as an area for further investigation.

[17] NodeProf is a dynamic analysis framework for Node.js, built on top of GraalVM. Empirical results give evidence that NodeProf is faster than Jalangi [25].

NACD follows another approach to server-side race detection that instead performs dynamic exploration or fuzzing of callback interleavings [9, 11]. This kind of tools can be computationally expensive when many iterations are needed to uncover a race bug. On the other hand, by design, these techniques avoid false positives and provide actionable information in the form of witness executions of failed tests.

Node.fz [9] seeks to perturb the execution of an application by fuzzing the internal event scheduling mechanism of Node.js. Based on modifications in certain internal components of Node.js, Node.fz shuffles the queues related to the event loop, worker pool tasks, and done operations. By doing so, it intends to explore alternative schedules, amplify the non-determinism in Node.js, and expose event race bugs. A drawback of Node.fz is that it functions with a one-thread worker pool, which may limit the exploration of interleavings when the actual execution involves several workers. While NACD also avoids the use of HB relations, similar to Node.fz, it manipulates the execution using the knowledge of the asynchronous APIs provided by built-in modules. Unlike Node.fz, this is performed entirely through JavaScript code, without changing Node.js internals. This design choice improves the maintainability concerning the evolving Node.js ecosystem.

NodeRacer [11] operates in three distinct phases. In the observation phase, NodeRacer instruments all functions at the application code level, gathers asynchronous information using Async Hooks, and produces a log file. In the next phase, an HB-graph is created by applying happens-before relation rules while processing the log file. Finally, in the guided execution phase, reruns are executed using a dynamic HB-graph to decide whether to postpone the scheduled callbacks. If the HB-graph indicates that a callback may interleave with others, NodeRacer randomly decides whether to postpone the callback. As previously discussed, NACD can explore more callback interleavings since it does not rely on a previously observed run to postpone callbacks (see Section 2), and injects delays at the Node.js API level.

Other related work. As illustrated by the motivating example in Section 2, event races sometimes manifest as flaky tests. Several techniques have been developed specifically to detect concurrency-related flaky tests [10, 8]. Ganji et al. [12] propose code coverage criteria that are specific for asynchronous operations in JavaScript programs. Arteca et al. [5] have introduced an approach to generate automated tests for JavaScript code with asynchronous callbacks. It may be interesting to combine such techniques with event race detection tools like NACD.

8 Conclusion

In this paper, we have introduced NACD, an approach and tool to explore potential callback interleavings in Node.js applications. The main innovation of NACD comes from understanding that most of the asynchronous behavior in Node.js programs originates from the Node.js built-in modules. This leads to a simple and extensible design, consisting of a model of the asynchronous behavior present in the Node.js modules, combined with a JavaScript runtime system that injects random delays when the asynchronous functions and objects of those modules are used by applications and libraries. Experimental results using 24 benchmarks from prior work show that NACD is capable of uncovering race bugs more effectively than existing state-of-the-art tools.

One direction for future work is to improve the delay injection mechanism to make more informed decisions using contextual information (like code patterns, API used, run state) and history of previous runs; this has potential to reduce the need for more runs. As race

bugs in JavaScript and Node.js applications can be complex bugs, more research could be conducted to provide support for visualization, execution replay, pinpointing root causes, and proposing fixes. Such research may help to shed some light on the bug detection variations observed across different tools. It may also be interesting to conduct larger-scale studies to investigate how tools like NACD can support the diagnosis of open issues related to event race bugs. Finally, we anticipate that the ideas in the design of NACD can be applied not only to other JavaScript runtimes like Deno and Bun, but also to other software platforms that employ similar single-threaded event-driven architectures, such as, Flutter for Dart or FastAPI for Python.

References

1 Christoffer Quist Adamsen, Anders Møller, Saba Alimadadi, and Frank Tip. Practical AJAX race detection for JavaScript web applications. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 38–48. ACM, 2018. doi:10.1145/3236024.3236038.

2 Christoffer Quist Adamsen, Anders Møller, and Frank Tip. Practical initialization race detection for JavaScript web applications. *PACMPL*, 1(OOPSLA):66:1–66:22, 2017. doi:10.1145/3133890.

3 Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. Finding broken promises in asynchronous JavaScript programs. *PACMPL*, 2(OOPSLA):162:1–162:26, 2018. doi:10.1145/3276532.

4 Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for JavaScript. *ACM Comput. Surv.*, 50(5):66:1–66:36, 2017. doi:10.1145/3106739.

5 Ellen Arteca, Sebastian Harner, Michael Pradel, and Frank Tip. Nessie: Automatically testing JavaScript APIs with asynchronous callbacks. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1494–1505. ACM, 2022. doi:10.1145/3510003.3510106.

6 Xiaoning Chang, Wensheng Dou, Yu Gao, Jie Wang, Jun Wei, and Tao Huang. Detecting atomicity violations for event-driven Node.js applications. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 631–642. IEEE / ACM, 2019. doi:10.1109/ICSE.2019.00073.

7 Xiaoning Chang, Wensheng Dou, Jun Wei, Tao Huang, Jinhui Xie, Yuetang Deng, Jianbo Yang, and Jiaheng Yang. Race detection for event-driven Node.js applications. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 480–491. IEEE, 2021. doi:10.1109/ASE51524.2021.9678814.

8 Marcello Cordeiro, Denini Silva, Leopoldo Teixeira, Breno Miranda, and Marcelo d'Amorim. Shaker: a tool for detecting more flaky tests faster. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 1281–1285. IEEE, 2021. doi:10.1109/ASE51524.2021.9678918.

9 James C. Davis, Arun Thekumparampil, and Dongyoon Lee. Node.fz: Fuzzing the server-side event-driven architecture. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 145–160. ACM, 2017. doi:10.1145/3064176.3064188.

10 Zhen Dong, Abhishek Tiwari, Xiao Liang Yu, and Abhik Roychoudhury. Flaky test detection in Android via event order exploration. In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 367–378. ACM, 2021. doi:10.1145/3468264.3468584.

11 André Takeshi Endo and Anders Møller. NodeRacer: Event race detection for Node.js applications. In *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*, pages 120–130. IEEE, 2020. doi:10.1109/ICST46399.2020.00022.

12 Mohammad Ganji, Saba Alimadadi, and Frank Tip. Code coverage criteria for asynchronous programs. In Satish Chandra, Kelly Blincoe, and Paolo Tonella, editors, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1307–1319. ACM, 2023. doi:10.1145/3611643.3616292.

13 Shin Hong, Yongbae Park, and Moonzoo Kim. Detecting concurrency errors in client-side JavaScript web applications. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 61–70. IEEE Computer Society, 2014. doi:10.1109/ICST.2014.17.

14 Casper Svenning Jensen, Anders Møller, Veselin Raychev, Dimitar Dimitrov, and Martin T. Vechev. Stateless model checking of event-driven applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 57–73. ACM, 2015. doi:10.1145/2814270.2814282.

15 Maurizio Leotta, Boni García, Filippo Ricca, and Jim Whitehead. Challenges of end-to-end testing with Selenium WebDriver and how to face them: A survey. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2023, Dublin, Ireland, April 16-20, 2023*, pages 339–350. IEEE, 2023. doi:10.1109/ICST57152.2023.00039.

16 Matthew C. Loring, Mark Marron, and Daan Leijen. Semantics of asynchronous JavaScript. In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages, Vancouver, BC, Canada, October 23 - 27, 2017*, pages 51–62. ACM, 2017. doi:10.1145/3133841.3133846.

17 Magnus Madsen, Ondrej Lhoták, and Frank Tip. A model for reasoning about JavaScript promises. *PACMPL*, 1(OOPSLA):86:1–86:24, 2017. doi:10.1145/3133910.

18 Luciano Mammino and Mario Casciaro. *Node.js Design Patterns – Second Edition*. Packt Publishing, 2nd edition, 2016.

19 Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. Detecting JavaScript races that matter. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 381–392. ACM, 2015. doi:10.1145/2786805.2786820.

20 Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. A survey of flaky tests. *ACM Trans. Softw. Eng. Methodol.*, 31(1):17:1–17:74, 2022. doi:10.1145/3476105.

21 Boris Petrov, Martin T. Vechev, Manu Sridharan, and Julian Dolby. Race detection for web applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 251–262. ACM, 2012. doi:10.1145/2254064.2254095.

22 Veselin Raychev, Martin T. Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 151–166. ACM, 2013. doi:10.1145/2509136.2509538.

23 Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 488–498. ACM, 2013. doi:10.1145/2491411.2491447.

24 Thodoris Sotiropoulos and Benjamin Livshits. Static analysis for asynchronous JavaScript programs. In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom.*, volume 134 of *LIPIcs*, pages 8:1–8:30. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ECOOP.2019.8.

25 Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. Efficient dynamic analysis for Node.Js. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 196–206, New York, NY, USA, 2018. ACM. doi:10.1145/3178372.3179527.

26 Alexi Turcotte, Michael D. Shah, Mark W. Aldrich, and Frank Tip. DrAsync: Identifying and visualizing anti-patterns in asynchronous JavaScript. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 774–785. ACM, 2022. doi:10.1145/3510003.3510097.

27 Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. A comprehensive study on real world concurrency bugs in Node.js. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 520–531. IEEE Computer Society, 2017. doi:10.1109/ASE.2017.8115663.

28 Lu Zhang and Chao Wang. RClassify: Classifying race conditions in web applications via deterministic replay. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 278–288. IEEE / ACM, 2017. doi:10.1109/ICSE.2017.33.

29 Yunhui Zheng, Tao Bao, and Xiangyu Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, pages 805–814. ACM, 2011. doi:10.1145/1963405.1963517.

30 Jingyao Zhou, Lei Xu, Gongzheng Lu, Weifeng Zhang, and Xiangyu Zhang. NodeRT: Detecting races in Node.js applications practically. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 1332–1344. ACM, 2023. doi:10.1145/3597926.3598139.

Bottom-Up Synthesis of Memory Mutations with Separation Logic

Kasra Ferdowsi
University of California San Diego, CA, USA

Hila Peleg
Technion, Haifa, Israel

Abstract

Programming-by-Example (PBE) is the paradigm of program synthesis specified via input-output pairs. It is commonly used because examples are easy to provide and collect from the environment. A popular optimization for enumerative synthesis with examples is Observational Equivalence (OE), which groups programs into equivalence classes according to their evaluation on example inputs. Current formulations of OE, however, are severely limited by the assumption that the synthesizer's target language contains only *pure* components with no side-effects, either enforcing this in their target language, or ignoring it, leading to an incorrect enumeration. This limits their ability to use realistic component sets.

We address this limitation by borrowing from Separation Logic, which can compositionally reason about heap mutations. We reformulate PBE using a restricted Separation Logic: Concrete Heap Separation Logic (CHSL), transforming the search for programs into a proof search in CHSL. This lets us perform bottom-up enumerative synthesis without the need for expert-provided annotations or domain-specific inferences, but with three key advantages: we (i) preserve *correctness* in the presence of memory-mutating operations, (ii) *compact* the search space by representing many concrete programs as one under CHSL, and (iii) perform a provably correct OE-reduction.

We present SOBEQ (*Side-effects in OBservational EQuivalence*), a bottom-up enumerative algorithm that, given a PBE task, searches for its CHSL derivation. The SOBEQ algorithm is proved correct with no purity assumptions: we show it is guaranteed to lose no solutions. We also evaluate our implementation of SOBEQ on benchmarks from the literature and online sources, and show that it produces high-quality results quickly.

2012 ACM Subject Classification Software and its engineering → Automatic programming

Keywords and phrases Program synthesis, observational equivalence

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.10

Related Version *Full Version*: `https://zenodo.org/records/15271843`

Supplementary Material *Software (Source Code)*: `https://github.com/peleghila/sobeq`
Software (ECOOP 2025 Artifact Evaluation approved artifact):
`https://doi.org/10.4230/DARTS.11.2.2`

Funding This paper was published with the support of the Israel Science Foundation (ISF) #651/23.

Acknowledgements The authors thank Nadia Polikarpova for her help with formalisms, and Ilya Sergey and Yotam M. Y. Feldman for reading many drafts of this work and asking very good questions.

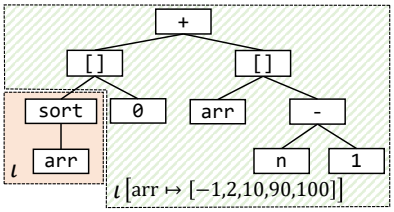
1 Introduction

Program Synthesis is the task of automatically generating a program that satisfies a given specification. A popular specification modality for synthesizing general-purpose programs is input-output examples, where a solution is defined as a program that, evaluated on each provided input, produces the provided output. This is commonly referred to as

© Kasra Ferdowsi and Hila Peleg;
licensed under Creative Commons License CC-BY 4.0

AST size 5:	⟨15⟩ `n + arr[0]`	⟨2⟩ `arr[n - 1]`	···		
AST size 3:	⟨10⟩ `arr[0]`	⟨[]⟩ `arr.slice(n)`	⟨4⟩ `n - 1`	···	
AST size 2:	⟨[−1, 2, 10, 90, 100]⟩ `arr.sort()`	···			
AST size 1:	⟨[10, 100, 90, −1, 2]⟩ `arr`	⟨5⟩ `n`	⟨0⟩ `0`	⟨1⟩ `1`	···

(a)



(b)

Figure 1 (a) Classical OE bank of programs with their observed evaluation results used to enumerate pure programs. Size 2 contains `arr.sort()` that sorts in-place, making compositions with it incorrect. (b) AST of the target program, divided into the variable states in which each subexpression is evaluated.

Programming-by-Example (PBE) [23, 16, 7, 21, 26]. PBE is notable because, unlike logic-based specifications [49, 56, 29], it does not require specialized knowledge to use, and values for it can be collected from a programming environment. It has been used to develop synthesizers for both end-users [23, 60] and programmers [17, 16, 12].

Consider a synthesis specification for a JavaScript program where, given an array of integers `arr` of length `n`, we want to return the sum of the smallest and largest elements of the array. We might specify this problem with the example:

$$\{\texttt{arr} \mapsto [10, 100, 90, -1, 2], \texttt{n} \mapsto 5\} \to 99$$

If we assume the value in `arr` is disposable, e.g., the program may sort it in place, the following is a solution that we would like to synthesize:

```
arr.sort()[0] + arr[n-1]
```

To solve this synthesis task, we can select a set of *components* (operations, library functions, etc.) and search the space of all programs constructed from those components. There are many ways to search, including constructing a representation of the space that can be traversed [15, 39, 23] and encoding the problem for a solver [56, 30]. Such approaches place heavy constraints on which components can be included. But the simplest form of search is to enumerate the space of programs [13, 19, 25, 52] by applying the components, then test resulting programs. Because the space of programs is astronomical, this approach is prohibitively slow when implemented naively. Moreover, most mechanisms to prune the space are domain-specific and restricted by the manual effort for handling each component individually.

Bottom-up Enumeration. One technique that does not require any domain-specific effort is *bottom-up enumeration* with an *Observational Equivalence* (OE) reduction [59, 2]. OE uses components to combine smaller programs into increasingly larger ones. To prune its search space, it maintains a bank of programs that are representatives of the *equivalence classes* of observationally equivalent programs, i.e., programs that, on the given inputs, evaluate to the same outputs.

New programs are obtained by applying components from the component set to smaller sub-expressions already in the bank. For example, in Fig. 1(a) applying `+` to the representative of equivalence classes ⟨5⟩ and ⟨10⟩ will yield the program `n + arr[0]`, which will be banked as the representative of equivalence class ⟨15⟩. The bank lets the enumeration memoize the

evaluation results of each program, and use them when evaluating larger programs, rather than re-evaluating each full program from scratch. The space is pruned by keeping only one program per equivalence class. Despite discarding many programs, this ensures no solution that existed in the un-reduced space is lost. Previous works use variations of this technique in synthesizers for C [59], Python [17, 16], JavaScript [45], Java [22], Selenium [36], and OCaml [2, 40].

Enumeration with heap mutations. The standard assumption of OE is that all components are *pure* and cause no side-effects [40, 17, 60, 59, 2, 36], and any mutations of the state in the target program are performed outside the synthesis task. Other synthesizers make assumptions on the intermediate states [16, 47], reducing the problem to synthesis of pure programs.

This is because OE has a fundamental limitation when handling mutating programs. Notice that JavaScript's `sort()` works in-place. In the example above, the call to `arr[n-1]` evaluates to a different value if evaluated before `arr.sort()` versus after it. A classical OE enumerator that performs its OE-reduction based on variable values in the inputs would consider `arr[n-1]` to be in the equivalence class $\langle 2 \rangle$. However, unlike its equivalent in languages like Python, `arr.sort()` also returns a self-reference, allowing the result to be composed into larger expressions, as in our target program. If the enumeration itself does not account for this, it will use the wrong memoized value computed using the unsorted value of `arr` in its candidates, yielding a result that does not actually satisfy the specifications. Moreover, the enumeration may see another program whose value is *really* $\langle 2 \rangle$, discard `arr[n-1]`, and never find this solution.

In other words, the classical bottom-up enumeration with OE described above is incorrect with memory mutations. This was also a problem in previous work: some synthesizers ignore the issue, leading to an incorrect enumeration [22, 45]. Peleg et al. [45], for example, allow JavaScript's in-place `sort()`, breaking their own purity assumption.

The key question addressed by this work, then, is how to correctly enumerate mutating programs without losing solutions while maintaining enough pruning to make the process tractable.

The Problem with Mutations. Classical OE considers only one memory state, the state both before and after evaluation of any expression. A composition with memory mutations, however, must consider a potentially different *after-state*. Fig. 1(b) shows this: sub-programs after a mutation are evaluated in a modified *before-state*, the after-state of the mutation. Since composition by the synthesizer simulates evaluation, composition with a mutating expression requires that the synthesizer to be aware of modified before-states. The synthesizer will need to generate complex expressions, e.g., `arr[n-1]` in Fig. 1(b), in a modified state.

Naively, we could store alongside each program not only its evaluation results, as in classical OE, but also its before- and after-state. While this would correct the equivalence relation and ensure programs are not erroneously discarded, the enumeration would still need to enumerate programs over and over: the program `n-1` that yields the result 4 would be enumerated both when `arr` is in its initial state and its modified state, and, as we do not know what the sequence of mutations in the target program will be (if any), in many other states as well. This is not only costly, as it creates many copies of the program in different equivalence classes, it is unnecessarily wasteful, since `n-1` is independent of the value of `arr`. We need an alternative equivalence reduction that both composes *valid* evaluation sequences, and constructs a *compact* program space.

Our approach. We introduce SOBEQ (*Side-effects in OBservational EQuivalence*), to our knowledge the first bottom-up proof-directed synthesis technique. SOBEQ is a bottom-up enumerative algorithm for expressions proven correct in the presence of mutating components.

A key insight of SOBEQ is that mutating operations only mutate a small portion of the state. If the variable state, usually considered as a variable valuation or a stack, is considered as a heap, then *Separation Logic*'s [51, 44] concept of *local reasoning* can be used to reason about programs in the synthesizer's space. The *Frame Rule* gives us the tools for a *compositional* application of operations, in much the same way that bottom-up synthesis already does for pure programs. Thanks to the Frame Rule, a compact representation of programs suffices, rather than considering the full variable valuation every time.

We define Concrete Heap Separation Logic (CHSL), a representation of the operational semantics of the language in the style of Separation Logic. For example, in CHSL we describe the mutating program `n++` evaluated on the example's input as the triple $\{\texttt{n} \mapsto 5\}\texttt{n++}\{\texttt{n} \mapsto 6; 5\}$ where the precondition $\texttt{n} \mapsto 5$ describes `n`'s initial value, and the postcondition $\texttt{n} \mapsto 6; 5$ describes its value after mutation and its result. Instead of searching for a program, SOBEQ searches for a CHSL derivation from inputs to outputs. Separation Logic has been used in program synthesis before [49, 29], including where information from the environment can be leveraged as a SL specification [20]. But previous work performed the proof search *deductively*, applying manually-crafted inference rules. The concrete heaps of CHSL allow SOBEQ to instead search the proof space bottom-up, composing larger programs from smaller ones, and relying only on program evaluation, like a classical OE search does. CHSL gives SOBEQ two key advantages over classical bottom-up synthesis.

Correct composition via logic. First, SOBEQ's bank stores CHSL triples rather than programs. To build larger programs, SOBEQ also applies each component to smaller triples in the bank, but unlike simple programs, triples can only be combined according to the rules of the logic. This means programs always compose in a way that forms a valid evaluation, and there is no risk of evaluating a program in the wrong state. Moreover, OE can be determined by simply ignoring the program and comparing the specifications surrounding each triple: two programs with the same pre- and postcondition are guaranteed to behave the same, and will also compose the same. This maintains SOBEQ's *correctness* in the presence of mutations.

Local reasoning. Second, thanks to the Frame Rule, CHSL can reason about the local effects of programs: triples do not need to store parts of the state that the program does not touch. For instance, $\{\texttt{n} \mapsto 5\}\texttt{n++}\{\texttt{n} \mapsto 6; 5\}$ does not specify anything about `arr`, because `n++` will behave the same way in all concrete states where the value of `n` is 5. This representation is *compact*, with each triple describing the behavior in many, potentially infinitely many, concrete states. This makes it much easier for SOBEQ to find programs that use variables after a mutation occurs, like our example, compared to the naive solution of enumerating using the full before- and after-states. This local reasoning also allows us to prioritize more general programs: ones that provide the same result and the same mutations for more concrete states. Discarding the less general programs lets us compact the space even further.

Our implementation. We implemented SOBEQ as a synthesizer for JavaScript programs that can create non-pure expressions, including sequences of expressions. We evaluated our synthesizer on 63 benchmarks curated from the literature, from STACKOVERFLOW and from competitive programming website LEETCODE. Compared to state of the art synthesis

$\texttt{arr} \mapsto [-1,2,10,90,100] * \texttt{n} \mapsto 5$:	$\{\texttt{arr} \mapsto [-1,2,10,90,100] * \texttt{n} \mapsto 5; 100\}$ `arr[n-1]`	⋯		
$\texttt{arr} \mapsto [10,100,90,-1,2] * \texttt{n} \mapsto 5$:	$\{\texttt{arr} \mapsto [10,100,90,-1,2] * \texttt{n} \mapsto 5; 2\}$ `arr[n-1]`	$\{\texttt{arr} \mapsto [10,100,90,-1,2] * \texttt{n} \mapsto 5; -1\}$ `arr[n-(1 + 1)]`	⋯	
$\texttt{arr} \mapsto [-1,2,10,90,100]$:	$\{\texttt{arr} \mapsto [-1,2,10,90,100]; [-1,2,10,90,100]\}$ `arr`	$\{\texttt{arr} \mapsto [-1,2,10,90,100]; -1\}$ `arr[0]`	⋯	
$\texttt{arr} \mapsto [10,100,90,-1,2]$:	$\{\texttt{arr} \mapsto [10,100,90,-1,2]; [10,100,90,-1,2]\}$ `arr`	$\{\texttt{arr} \mapsto [10,100,90,-1,2]; 10\}$ `arr[0]`		
	$\{\texttt{arr} \mapsto [-1,2,10,90,100]; [-1,2,10,90,100]\}$ `arr.sort()`	$\{\texttt{arr} \mapsto [-1,2,10,90,100]; -1\}$ `arr.sort()[0]`	⋯	
$\texttt{n} \mapsto 5$:	$\{\texttt{n} \mapsto 5; 5\}$ `n`	$\{\texttt{n} \mapsto 5; 4\}$ `n - 1`	$\{\texttt{n} \mapsto 6; 5\}$ `n++`	⋯
emp:	$\{emp; 0\}$ `0`	$\{emp; 1\}$ `1`	$\{emp; -1\}$ `-1`	⋯

Figure 2 The precondition bank used to enumerate the program in Fig. 1(b). Each program is additionally labeled with its postcondition comprising an assertion and its result.

of general mutations, SObEq returns more concise and higher quality solutions, with less overfitting and without performing spurious operations, on both tasks that require and do not require mutations. Importantly, SObEq is both deterministic, meaning it will more reliably integrate with a larger tool, and proved correct, meaning that it will never discard all solutions to a task while exploring the space.

Contributions. The main contributions of this paper are as follows:

- A formulation of the PBE problem with memory mutations as the search for a derivation in a restricted Separation Logic, CHSL.
- The SObEq algorithm, a bottom-up enumerative algorithm that searches for a derivation with a proven-correct OE-reduction.
- An implementation of SObEq in a JavaScript synthesizer, and an evaluation on 63 benchmarks curated from the literature, StackOverflow and LeetCode, which shows that SObEq outperforms state of the art on a variety of synthesis tasks with and without mutations, and highlights the class of tasks for which a naive approach would fail.

2 Overview

In this section, we present SObEq through the example introduced in Sec. 1. First, we present SObEq's notation for mutating programs, which is compact compared to the real space of programs. Next, we show how this helps us enumerate programs correctly. Finally, we show how ordering programs by generality helps us prune the space even further.

2.1 Mutating programs

We begin with a reminder of programs and their construction in Observational Equivalence [2, 59], then show how we modify them to correctly reason about side-effects.

Classical OE-reduction and the program bank. Bottom-up enumeration with an OE-reduction uses a component set $\mathcal{C}$ comprising functions, operators, variables, and literals, to search for a program specified by a vector of input-output examples, $\vec{\mathcal{E}}$, where each example is of the form $\iota \to \omega$. For each program we compute its equivalence class's label. In classical

OE, we use its evaluation result, $[\![p]\!](\iota)$, for each ι in $\vec{\mathcal{E}}$. In our example, the expression `arr` has the *observed behavior* $[10, 100, 90, -1, 2]$ on the provided input, so its equivalence class is $\langle[10, 100, 90, -1, 2]\rangle$.

To enumerate the space of programs, we use a *bank* to store representatives of equivalence classes, as seen in Fig. 1(a). For each function or operator $f \in \mathcal{C}$ with arity k, we collect all k-tuples of programs already in the bank and use them as arguments to apply f. This is usually done with some notion of iterations, e.g., as in Fig. 1(a), by increasing number of AST nodes. When enumerating programs of AST size 5, then, the enumerator considers the component array dereference (`[]`), which has an arity of 2. Among pairs of programs collected from the bank with a total size of 4 will be the program `arr`, representative of the equivalence class $\langle[10, 100, 90, -1, 2]\rangle$, and the program `n - 1`, representative of $\langle 4\rangle$. Applying array dereference yields `arr[n - 1]` with the observed behavior $\langle 2\rangle$. Crucially, we do not have to evaluate the full program `arr[n - 1]` to get this value: we can use observed values $[10, 100, 90, -1, 2]$ and 4 and only compute the final step.

When the enumeration finds a program from an equivalence class with no representative in the bank – i.e., no program in the bank has the same label – it is added to the bank to be used when constructing larger programs. This is the case with `arr[n - 1]`, which is added to the bank at size 5. Alternatively, if a representative of the equivalence class already exists in the bank, we discard the new program. If a program's observed values are equal to the provided outputs, the program is a solution. Since we only use discovered equivalence classes (i.e., programs) to construct larger programs, the effect of OE's pruning compounds.

The problems with side-effects. We quickly run into issues when $\mathcal{C}$ contains functions or operators that can cause side-effects. The next step in solving our example is to enumerate `arr.sort()[0] + arr[n - 1]`. But, as we are about to see, using the classical approach this program will not be evaluated correctly, and it may even be pruned away.

We already saw how the subprogram p_1 = `arr[n - 1]` was enumerated and labeled $\langle 2\rangle$, and p_2 = `arr.sort()[0]` will be enumerated similarly: at size 4, it will be composed from `arr.sort()` (size 2) and `0` (size 1), and added to the bank as a representative of $\langle -1\rangle$.

The first source of trouble is that OE is a dynamic programming algorithm: when constructing the larger p_2 `+` p_1, the enumerator uses the memoized observed values to compute: $\langle -1 + 2\rangle = \langle 1\rangle$. However, since p_2 modifies `arr`, p_2 `+` p_1 *actually* evaluates to 99 on the input ι. Worse, this erroneous value means the synthesizer will consider p_2 `+` p_1, our solution, equivalent to p_1 `+` p_2 so it may be discarded. While a seemingly simple solution is to just evaluate the program in full each time, ensuring labels are correct, this is not only expensive, depending on the operation, but more importantly, insufficient. If the synthesizer sees the constant value 2, e.g., by enumerating `1 + 1`, this would cause the larger – and seemingly equivalent – `arr[n - 1]` to be discarded. OE hinges on the enumeration only discarding a program when it already has a suitable replacement, but `arr[n - 1]` evaluated *after* `arr.sort()` would still have no equivalent program in the bank.

Rethinking the program representation. Our problem was not distinguishing between the *same* program when evaluated using *different* values of `arr`. We also saw in Sec. 1 the need for distinction between programs that return the same value, but their *effect* on variables is different, e.g., `n` and `n++` when evaluated using the same value of `n`.

We replace programs with triples in Concrete Heap Separation Logic (CHSL): the precondition records the variables before its evaluation, and the postcondition comprises an assertion recording the variables after evaluation and the result of the evaluation, like so:

$$\{\texttt{arr} \mapsto [10, 100, 90, -1, 2]\}\texttt{arr.sort()}\{\texttt{arr} \mapsto [-1, 2, 10, 90, 100]; [-1, 2, 10, 90, 100]\}$$

From here on, we name the original array value arr_{orig} and its sorted value arr_{sort}, and use them in our examples and figures for brevity. In this representation, we can tell apart these two triples:

$$\{\texttt{arr} \mapsto arr_{orig}\}\texttt{arr}\{\texttt{arr} \mapsto arr_{orig}; arr_{orig}\} \qquad \{\texttt{arr} \mapsto arr_{sort}\}\texttt{arr}\{\texttt{arr} \mapsto arr_{sort}; arr_{sort}\}$$

where the first is `arr` evaluated on the original array, returning the original array value, and the second is evaluated on a state where the array has been sorted, returning the sorted value. We use the second triple to construct larger programs evaluated after `sort()`.

Notice that since all values are concrete, CHSL does not need domain- or library-specific inference rules to connect the pre- and postcondition of the triple, but can rely entirely on evaluation.

Local reasoning. These pre- and postconditions do not encode fully-concrete states, they leave `n` unconstrained. This makes our representation of the space more compact: the triple $\{\texttt{arr} \mapsto arr_{orig}\}\texttt{arr.sort()}\{\texttt{arr} \mapsto arr_{sort}; arr_{sort}\}$ takes advantage of the fact that only constraining `arr` is sufficient to reason about the behavior of this program. We only need to constrain the *footprint* of the program.

CHSL includes the notion of the separating conjunction $*$ to indicate separate parts of the heap (or, in our case, variable store) that can be reasoned about locally. Thanks to aliasing restrictions of the synthesizer's target language detailed in Sec. 3.1, any two variables can be separated with $*$. E.g., the assertion $\texttt{arr} \mapsto arr_{orig} * \texttt{n} \mapsto 5$ describes the concrete initial state. Once we replace the programs in our bank with CHSL triples, we can now further borrow from Separation Logic when considering evaluation sequences.

2.2 Enumeration and heaps

A bottom-up enumeration composes larger programs from smaller ones by selecting a sequence of arguments to apply functions or operators f to. In a classical enumeration, our bank contains programs, so if we wish to apply `+` to two programs in it, this is only a matter of creating the new program p_1 `+` p_2. When executing p_1 `+` p_2 on some variable valuation, the runtime first evaluates p_1, then p_2, then their addition. The classical OE enumeration duplicates this by summing the evaluation result banked for p_1 to that banked for p_2. Without mutations, this is correct for all orderings of arguments to `+` as all the programs evaluate from the initial state and to the initial state, so their results can always be memoized in the bank.

Now that our bank comprises CHSL triples, however, the enumeration needs to apply f to triples, not programs. Here, we have to be more careful. Let us consider two programs in our space, `arr[0]` and `arr.sort()[0]` enumerated with classical OE in Fig. 1(a). In our new bank, they are now the triples:

$$tr_1 = \{\texttt{arr} \mapsto arr_{orig}\}\texttt{arr[0]}\{\texttt{arr} \mapsto arr_{orig}; 10\}$$
$$tr_2 = \{\texttt{arr} \mapsto arr_{orig}\}\texttt{arr.sort()[0]}\{\texttt{arr} \mapsto arr_{sort}; -1\}$$

First, we consider what applying `+` to a pair of triples means: applying it to (tr_1, tr_2) as arguments means trying to create `arr[0] + arr.sort()[0]`, or applying `+` to the program components of the triples. However, unlike the mutation-free case, not every pair of triples with those two programs can create this application. We expect a sequence of arguments to a component to describe a real evaluation of those arguments.

This is akin to creating a valid composition or sequence in Separation Logic, the postcondition of each argument must be equal to the precondition of the next one in the sequence; this means all mutations are accounted for. For (tr_1, tr_2), this is not a problem: $\texttt{arr} \mapsto arr_{orig}$ is

the after-state of tr_1 and the before-state of tr_2, which means that their sequence mutates the state from the precondition of tr_1 to the postcondition of tr_2, and they can be used in the application of any component, e.g., `+`, with the values memoized in the triple, so we add the two precomputed results $10 + -1$, and get:

$\{\texttt{arr} \mapsto arr_{orig}\}\texttt{arr[0] + arr.sort()[0]}\{\texttt{arr} \mapsto arr_{sort}; 9\}$

However, we cannot compose (tr_2, tr_1) in the same way: `arr.sort()` mutated the state, changing the value of `arr` to arr_{sort}, so tr_1, whose memoized result was computed where `arr` is arr_{orig}, cannot follow it. Requiring arguments to a component to form a valid sequence ensures that only valid executions are enumerated by the synthesizer. The enumerator will not try to compose (tr_2, tr_1) under any binary operator, but can – and will – select (tr_1, tr_2) as arguments to `+`. Once a new program is composed, the enumerator invokes the language interpreter to evaluate the composition from the values of its arguments, yielding a triple containing the result and compound effect of the arguments and operation. We denote this invocation the Eval inference rule.

If we partition our bank of programs by the program's precondition, as in Fig. 2, then the enumerator can always create correct sequences of arguments by construction, using a triple's postcondition to fetch all suitable next arguments according to their precondition.

Discovering more preconditions. Let us consider the last subprogram of our target program: `arr[n - 1]`. It is going to be added with `arr.sort()[0]`, which means that for that composition to be allowed, it must match the state after `arr` is sorted. Specifically, we consider the first argument of the array dereference, which will be needed to enumerate it:

$tr_3 = \{\texttt{arr} \mapsto arr_{sort}\}\texttt{arr}\{\texttt{arr} \mapsto arr_{sort}; arr_{sort}\}$

Importantly, this triple will never be enumerated if the enumerator only considers triples starting at a precondition describing the initial state, but without it we will never find the target program.

We have to ensure, therefore, that we also enumerate programs that have other preconditions. tr_3 is a necessary building block for such programs.

To this end, whenever a new assertion is discovered by the enumerator, e.g., when a mutating component is evaluated, creating a never-before-seen assertion in the postcondition, as enumerating $\{\texttt{arr} \mapsto arr_{orig}\}\texttt{arr.sort()}\{\texttt{arr} \mapsto arr_{sort}; arr_{sort}\}$ does, we get ready to enumerate programs that *begin* in this new assertion: we create tr_3, which any program starting at $\texttt{arr} \mapsto arr_{sort}$ will need to incorporate, and add it to the bank, which also adds $\texttt{arr} \mapsto arr_{sort}$ to the bank as a precondition. The next time the enumerator selects arguments to an operator, tr_3 will be available. This lets the enumerator create programs that begin at $\texttt{arr} \mapsto arr_{sort}$.

Programs with different footprints. Now that we have tr_3, the first argument to the array dereference, let us consider the second:

$\{\texttt{n} \mapsto 5\}\texttt{n - 1}\{\texttt{n} \mapsto 5; 4\}$

We enumerate this program at the initial value of `n`, i.e., without the need for its precondition to be discovered. However, when the enumerator tries to compose it with tr_3 into a sequence of arguments to `[]`, it runs into a problem: the two triples do not make a valid sequence. The assertion in the postcondition of the first argument, $\texttt{arr} \mapsto arr_{sort}$, is different than the precondition of the following argument, $\texttt{n} \mapsto 5$.

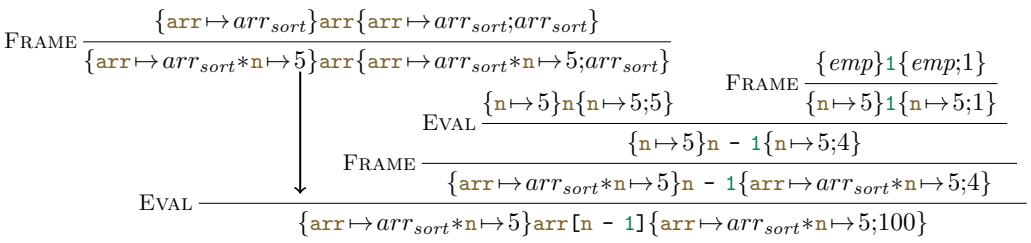


Figure 3 Using Frame and Eval to enumerate `arr[n - 1]`. Recall arr_{sort} denotes the constant value $[-1, 2, 10, 90, 100]$ for brevity.

However, we clearly see that these two assertions deal with separate parts of the heap, and do not interfere with each other. This is where we draw on Separation Logic's powerful tool: the Frame rule. Since these are separate parts of the heap, we can use the separating conjunction to create a unified heap in the pre- and postcondition of both triples. The more constrained triples are now composable, so the enumerator can apply array dereference and Eval, yielding:

$$\{\texttt{arr} \mapsto arr_{sort} * \texttt{n} \mapsto 5\}\texttt{arr[n - 1]}\{\texttt{arr} \mapsto arr_{sort} * \texttt{n} \mapsto 5; 100\}$$

The full derivation is shown in Fig. 3. Then similarly, the same extension – now adding $\texttt{n} \mapsto 5$ to tr_2 and nothing to the second argument – is applied to allow the enumerator to compose the target program. This is an important strength of our technique: the same triple for `n - 1` can compose with any value for `arr` – we will only enumerate it once!

We notice that in this case, this will discover a new assertion not as a postcondition as we did when `sort()` mutated `arr` but as a precondition. Once the program is added to the bank, the enumerator will continue to enumerate more programs with this new assertion as their precondition.

2.3 Equivalence and mutation

So far, we showed how the enumerator composes programs, but we did not show how the OE-reduction works. Now that we have a new representation of programs, we can re-define how each triple is observed and labeled, so that the label vector suits our needs.

In classical OE, the observed value for each example is the program's evaluation result, but this is now insufficient: while in the precondition $\texttt{arr} \mapsto [10, 100, 90, -1, 2]$ the program `arr[1]` evaluates to 100, if we discard all the following programs that evaluate to 100 we will also discard the `arr[n - 1]` that is evaluated on a sorted array. Likewise, `n` and `n++` with the same precondition are distinct in that they return the same value given the same precondition, but have a different effect on the state.

Generally, we can see that we can label a triple with its pre- and postcondition, including the result. Once those are identical, any programs that form a valid triple with them would be *interchangeable*, i.e., can be swapped for each other with no change to a larger evaluation. In Fig. 2, the precondition component is expressed by the precondition the triple is stored under in the bank, while the postcondition appears above each program.

Generality of equivalence classes. The equivalence class labels above define an *equivalence relation* for OE. However, in this setting, we can do even better: there are cases where we want to discard programs even though one is not equivalent to the other. Let us consider two triples:

$$tr_4 = \{\texttt{n} \mapsto 5\}\texttt{n - n}\{\texttt{n} \mapsto 5; 0\} \qquad tr_5 = \{emp\}\texttt{0}\{emp; 0\}$$

where *emp* is the unconstrained heap.

Both triples have the same evaluation result (0), and the same effect (no effect), but different assertions at the pre- and postconditions. But they are not completely different: if we select a *frame axiom* $R = \texttt{n} \mapsto 5$, we can bridge their difference: $\{emp * R\}\texttt{0}\{emp * R; 0\}$ is equivalent to tr_4. Under this condition, we say that tr_5 is more general than tr_4: it describes a program that behaves the same over more concrete states. If we let the more general equivalence class *subsume* less general ones, we further compact our representation of the space while remainting correct. While we cannot tractably apply this additional reduction to the entire space, we can at least discard less general programs once more general ones exist in the bank.

Temporary values and variables. This tactic cannot stand on its own: naively employing it means $tr_6 = \{emp\}\texttt{5}\{emp; 5\}$ subsumes $tr_7 = \{\texttt{n} \mapsto 5\}\texttt{n}\{\texttt{n} \mapsto 5; 5\}$ in the same way. However, in Fig. 2 similar subsumptions do not happen: $\{emp\}\texttt{-1}\{emp; -1\}$ does not subsume $\{\texttt{arr} \mapsto arr_{sort}\}\texttt{arr[0]}\{\texttt{arr} \mapsto arr_{sort}; -1\}$ and $\{\texttt{arr} \mapsto arr_{orig} * \texttt{n} \mapsto 5\}\texttt{arr[n - (1+1)]}\{\texttt{arr} \mapsto arr_{orig} * \texttt{n} \mapsto 5; -1\}$, which are still in the bank. Why?

We notice that if these subsumptions are allowed, the programs will no longer be interchangeable, and some will break entirely. Consider the application of `++`: while $\{\texttt{n} \mapsto 5\}\texttt{n++}\{\texttt{n} \mapsto 6; 5\}$ uses tr_7 as its argument, `5++` using tr_6 does not compile. Likewise, replacing `arr` within `arr.sort()` with the array literal with `arr`'s value would produce a different mutation of the heap, indicting they should be separate programs. The difference, which must be encoded into our representation, is what is *reachable* from state variables, and if so, from *where*.

Our evaluation result therefore comprises two components: the *value* returned by evaluation, and the *location* of that value. tr_3, then, returns $arr_{sort}@arr$, and tr_7 returns $5@n$. This ensures the interchangeability within `++` or `sort`: the new program will not only compile, it will affect the same variable.

What, then, should be the location of a temporary variable, i.e., a variable that is not reachable from the stack? We must consider the location's purpose: to separate programs enough to preserve interchangeability when the heap is mutated. Two temporary values, on the other hand, are (inherently) interchangeable, so we do not want to over-separate them which would inflate the number of equivalence classes in the bank. We therefore give all temporary values the *same* location: $\bot$. This way, tr_4 and tr_5 both return $0@\bot$, preserving the subsumption, and tr_6 returns $5@\bot$ which separates it from tr_7.

Fig. 2 omits this for simplicity, but does still separate $\{emp\}\texttt{-1}\{emp; -1\}$ with result $-1@\bot$ from $\{\texttt{arr} \mapsto arr_{sort}\}\texttt{arr[0]}\{\texttt{arr} \mapsto arr_{sort}; -1\}$ with result $-1@arr[0]$ and from $\{\texttt{arr} \mapsto arr_{orig} * \texttt{n} \mapsto 5\}\texttt{arr[n-(1+1)]}\{\texttt{arr} \mapsto arr_{orig} * \texttt{n} \mapsto 5; -1\}$ with result $-1@arr[3]$.

Defining the Solution. We are almost finished: like classical OE, we continue iteratively enumerating programs into the bank until a solution is found. Our final step is to reconsider the definition of a solution to a synthesis specification.

Input variables	Identifiers $x \in \mathcal{V}$
Literal values	$v \in \mathcal{K}$
Assertion	$a \in \mathcal{A} ::= emp \mid x \mapsto v * \mathcal{A}$
Assertions	$\vec{a}, \vec{P}, \vec{Q} \in \mathcal{A} \times \cdots \times \mathcal{A}$
Location	$l \in \mathcal{M} ::= \bot \mid x \mid x[n]$, n is an integer literal
Result	$r \in \mathcal{R} ::= v@l$
Results	$\vec{r} \in \mathcal{R} \times \cdots \times \mathcal{R}$
Postcondition	$\{\vec{Q}; \vec{r}\}$

$$\frac{\{\vec{P_1}\}p_1\{\vec{P_2};\vec{r_1}\} \;\; \{\vec{P_2}\}p_2\{\vec{P_3};\vec{r_2}\} \;\; \cdots \;\; \{\vec{P_k}\}p_k\{\vec{P}_{k+1};\vec{r_k}\} \quad c \in \mathcal{C}, arity(c) = k \quad \forall i.(c(r_1^i, \ldots, r_k^i), P_{k+1}^i) \to (r^i, Q^i)}{\{\vec{P_1}\}c(p_1, \ldots, p_k)\{\vec{Q};\vec{r}\}} \textsc{Eval}$$

$$\frac{\{\vec{P}\}p\{\vec{Q};\vec{r}\}}{\{\vec{P} * \vec{R}\}p\{\vec{Q} * \vec{R};\vec{r}\}} \textsc{Frame}$$

Figure 4 CHSL syntax and inference rules.

When enumerating a program in a non-mutating space with an OE-reduction using the examples' inputs, a solution is a program that is labeled by (i.e., that evaluates to) the provided outputs. In SOBEQ this is insufficient: the enumerator can find a triple returning the correct value in *some* precondition. To be correct when evaluated on the concrete example inputs, we require that a solution's precondition describe the concrete initial state ι. In our example, `arr` $\mapsto arr_{orig} *$ `n` $\mapsto 5$ trivially describes ι – it is equal to it – but so does `n` $\mapsto 5$.

Thus, to be a solution, a triple must: 1) return the values of the example outputs, and 2) have a precondition that describes ι.

SOBEQ is, to our knowledge, the first proof-directed synthesis algorithm that works bottom-up. It is also the first bottom-up enumerative synthesizer for general mutating components with a correctness guarantee: if a solution exists in the space spanned by $\mathcal{C}$, SOBEQ will find an equivalent solution.

3 Enumerative Synthesis and Heap Mutations

In this section we define the basic elements with which we construct SOBEQ: programs and their evaluation, and the synthesis task.

3.1 Concrete Heap Separation Logic

In this section, we introduce Concrete Heap Separation Logic (CHSL), a notation for operational semantics in the style of Separation Logic. Its syntax and two inference rules are showin in Fig. 4.

The heap. In CHSL, we use a heap notation to reason about the concrete values assigned to variables from the input. We can reason locally about each individual variable because SOBEQ's target language does not allow assignments, and so as long as nothing is aliased at the input – we can enforce this when generating the synthesis task – each variable is separate from all others and can be separated from them with a separating conjunction, $*$.

CHSL assertions. CHSL reasons about concrete values assigned to variables. As such, a CHSL assertion comprises heaplets of the form $x \mapsto v$ where v is a concrete value.

This also means we can trivially transform a state (partial function) σ into an assertion describing an identical heap s.t. $x_1 \mapsto \sigma(x_1) * \cdots * x_n \mapsto \sigma(x_n)$ for all $x_i \in \text{dom}(\sigma)$. We use the function interchangeably with its assertion equivalent, i.e., σ as a shorthand for the assertion above.

For a vector of states $\vec{\sigma}$, corresponding to the vector of examples $\vec{\mathcal{E}}$, CHSL likewise uses a vector of assertions. We therefore also define a vector separating conjunction s.t. $\vec{P} * \vec{R} = \langle P_i * R_i \rangle_i$.

Values with location. In the presence of mutations, we must also consider that some values are reachable from our heap. As shown in Sec. 2, this is important for components that modify their arguments (e.g. `arr.sort()`), especially components that return a self-reference and are therefore composable, as in `arr.sort().reverse()`, where both `sort` and `reverse` mutate `arr`.

To support this, a result $r \in \mathcal{R}$ comprises two components: a *value* and that value's *location*, $l \in \mathcal{M}$ that can be a variable, a variable at a *concrete* index, or $\perp$. We use $\perp$ to indicate a value not reachable from any variable, i.e., a temporary value. We decompose the result as $r = v@l$. As with variable values, the values in results are concrete.

Triples. A triple in CHSL comprises a precondition $\vec{P} \in \mathcal{A}^k$ (where $|\vec{\mathcal{E}}| = k$), the program p, and a postcondition. Our postconditions have two components: i) an assertion $\vec{Q} \in \mathcal{A}^k$, and ii) the evaluation results of p on $\vec{P}$. Vectors of assertions and results form a conjunction: $\{\vec{P}\}p\{\vec{Q}; \vec{r}\}$ means all triples $\{P^1\}p\{Q^1; r^1\}, \{P^2\}p\{Q^2; r^2\}, \ldots, \{P^k\}p\{Q^k; r^k\}$ hold.

We call the subset of each assertion Q whose values are different from P the *effect* of the program on P. For example, in the triple $\{\texttt{x} \mapsto 1 * \texttt{y} \mapsto 2\}\texttt{x++ + y}\{\texttt{x} \mapsto 2 * \texttt{y} \mapsto 2; 3\}$ the effect is $\texttt{x} \mapsto 2$.

We consider $\{\vec{P}\}p\{\vec{Q}; \vec{r}\}$ to be a *valid* triple if it accurately tracks the evaluation of p on $\vec{P}$. In other words, assuming a (deterministic) small-step operational semantics for programs the behavior of which is defined by the interpreter of the language: $\rightarrow : (\mathbb{P}, \mathcal{A}) \rightarrow (\mathcal{R}, \mathcal{A})$, we say $\{\vec{P}\}p\{\vec{Q}; \vec{r}\}$ is valid if $\forall 1 \leq i \leq k.(p, P^i) \rightarrow (r^i, Q^i)$.

Because SObEq will be *enumerating* triples, it is crucial that it can construct valid triples.

The Eval rule. When constructing a new triple, the Eval rule denotes the use of the interpreter to ensure the resulting triple is valid. Because an evaluation of an AST node will first evaluate its children in sequence, then use the results to evaluate the node's operation, Eval requires a sequence of triples. As in the classical Hoare Logic Sequence rule, Eval requires matching midconditions. However, since we often want to form a sequence from triples where $Q_i \neq P_{i+1}$, we use the Frame rule to find a larger composed heap where the midconditions do match.

The Frame rule. Generally, all triples in the space contain only their footprint in their pre- and postcondition. As the example in Fig. 3 shows, in this compact representation midconditions may not match. We recall that the frame rule allows us to reason about a larger heap by equally extending the pre- and postcondition with an additional assertion called the *frame axiom*.

Fortunately, in CHSL, frame axioms are very easy to find. To prepare a pair of arguments for Eval, then, one of two things is true: either for every i, $Q_1^i \cup P_2^i$ is still a partial function (i.e., if the same variable appears in both, it maps to the same value) and R_1^i, R_2^i are found by their difference, or the two triples cannot form a valid sequence.

Combining Eval and Frame. When the SObEq enumeration will apply a function or operator f to k child triples, this will combine the two rules. Since not all valid sequences of possible arguments to f have matching midconditions, we apply the Frame rule to the entire sequence:

given a set of $k = \mathit{arity}(f)$ triples $tr_i = \{\vec{P_i}\}p_i\{\vec{Q_i}; \vec{r_i}\}$, Eval is applicable if there exist k frame axioms $\vec{R}_1, \ldots, \vec{R}_k$ where $\vec{Q}_1 * \vec{R}_1 = \vec{P}_2 * \vec{R}_2, \vec{Q}_2 * \vec{R}_2 = \vec{P}_3 * \vec{R}_3, \ldots, \vec{Q}_{k-1} * \vec{R}_{k-1} = \vec{P}_k * \vec{R}_k$. Some of these $\vec{R}_i$ may be $\vec{emp}$. Now that the midconditions of the triples match, Eval can be applied, yielding a triple that evaluates from the new shared precondition $\vec{P}_1 * \vec{R}_1$, and computing the result and precondition for $f(p_1, \ldots, p_k)$.

3.2 Problem definition

Since SObEq uses CHSL triples to solve a PBE problem, we must re-state the PBE problem for CHSL. In this section we define what it means for a CHSL triple to represent our target program, define a translation from a PBE task, and explain how additional specifications, constraining the program's effects on the state, can also be supported.

The Programming by Example (PBE) task is usually defined as follows: given a vector of input-output examples $\vec{\mathcal{E}} = \langle \iota_i \to \omega_i \rangle_i$, find a program p over a component set $\mathcal{C}$ where for every input ι_i, evaluating p on ι_i, denoted $[\![p]\!](\iota_i)$, is equal to the provided output ω_i. We first want to translate this into a goal of the shape $\{P\}_\{Q; r\}$.

Examples in CHSL. We denote the vector of input states $\vec{I} = \langle \iota_i \mid \iota_i \to \omega_i \rangle_i$. The direct translation of $\vec{\mathcal{E}}$ to CHSL, then, is $\{\vec{P}\}_\{_; \langle \omega_i @ _ \rangle_i\}$, where $\exists \vec{R}.\vec{P} * \vec{R} = \vec{I}$. In other words, the precondition needs to *describe* the initial states (it is a footprint, so it may not describe all input variables), any postcondition will satisfy our specification, and the correct values can be at any location.

Constraining effect. Users may want to constrain specific mutations as well as the value. We let the user pair with each example any constraints on the effect, e.g., the value of `n` must be 9, or `arr` must end in its initial value $[10, 100, 90, -1, 2]$. To do this, we let the user provide an *effect constraint* $q : \mathcal{V} \to \mathcal{K}$, constraining the target value for any variable, where if $x \notin \mathrm{dom}(q)$, x is unconstrained.

The SObEq task. A PBE user does not need to know about CHSL. Much like the original $\vec{\mathcal{E}}$, they provide a vector $\vec{\Phi}$ where each $\varphi \in \vec{\Phi}$ is a pair of example and effect constraint: $\varphi = (\iota \to \omega, q)$. If they are unconcerned with effects, this is identical to providing $\vec{\mathcal{E}}$.

We then turn $\vec{\Phi}$ into a synthesis goal in CHSL:

▶ **Definition 1** (SObEq synthesis goal). *Given a specification $\vec{\Phi}$, our goal $\mathcal{G}_{\vec{\Phi}}$ is:*

$$\mathcal{G}_{\vec{\Phi}} = \{\vec{P}\}_\{\langle q_i * _ \rangle_i; \langle \omega_i @ _ \rangle_i\}, \textit{ where } \exists \vec{R}.\vec{P} * \vec{R} = \vec{I}$$

The SObEq task is to find a triple $\{\vec{\sigma}\}p\{\vec{\sigma}'; \vec{r}\}$ that matches $\mathcal{G}_{\vec{\Phi}}$.

3.3 The program space

Now that we have SObEq's specifications, we define the space in which it performs the search. We consider the component set $\mathcal{C}$ as a union of three sets: $\mathcal{V} \cup \mathit{lits} \cup \mathcal{F}$, where $x \in \mathcal{V}$ are the variables in $\vec{I}$, $l \in \mathit{lits}$ are literals in the component set, and $f \in \mathcal{F}$ are the functions and operators that can be applied to subprograms ($\mathit{arity}(f) > 0$). The full program space of a synthesizer with a component set $\mathcal{C}$ is then every program using the components in $\mathcal{C}$. We denote this program set $(\!|\mathcal{C}|\!)$. This set is infinite and highly redundant, e.g., contains both `x + 1` and `1 + x + 0`, and in our mutating domain, both `arr.sort()` and `arr.sort().sort()`.

Classical Observational Equivalence. In a problem domain with no side-effects, observational equivalence unifies programs with the same evaluation results on the example inputs. This is expressed in the equivalence relation $\equiv_{\vec{\mathcal{E}}}$ that deems two programs p_1, p_2 (observationally) equivalent if $\forall \iota \to \omega \in \vec{\mathcal{E}}.[\![p_1]\!](\iota) = [\![p_2]\!](\iota)$. The OE-reduced space $(\!|\mathcal{C}|\!)^{OE}$ is then defined using $\equiv_{\vec{\mathcal{E}}}$. This idea was first suggested by Udupa et al. [59] and Albarghouthi et al. [2] and is in wide use today.

A classical bottom-up enumeration (CBE), keeps a bank of programs, and the vector of observed results is used to label the program's equivalence class. Only one program with each vector is added to the bank, meaning any observationally equivalent programs are discarded.

A program space with mutations. In the presence of mutations, the space of possible programs no longer comprises only programs evaluated in the initial states $\vec{I}$. As we saw in Sec. 2, composition in this space requires subprograms that are evaluated in a modified state.

The full program space when $\mathcal{F}$ includes mutations is constructed inductively, building larger valid triples from smaller ones via applications of Eval, and the Frame rule when necessary. We denote the possible values for each $x \in \mathcal{V}$ as $\mathcal{K}_x$. We define the full space of programs $(\!|\mathcal{C}|\!)$:

▶ **Definition 2** (Full program space). *We define the full program space* $(\!|\mathcal{C}|\!)$ *as* $\bigcup_{i\geq 0}(\!|\mathcal{C}|\!)_{(i)}$ *where* $(\!|\mathcal{C}|\!)_{(i)}$ *is defined inductively as follows:*

$$(\!|\mathcal{C}|\!)_{(0)} = \{\{e\vec{m}p\}l\{e\vec{m}p; \overrightarrow{v@\bot}\} | l \in lits, v \in \mathcal{K} \text{ is the value of } l\} \cup$$
$$\{\{\vec{\sigma}\}x\{\vec{\sigma}; \langle v_i @ x\rangle_i\} | x \in \mathcal{V}, \vec{\sigma} = \langle x \mapsto v_i\rangle_i, \vec{v} = \mathcal{K}_x \times \cdots \times \mathcal{K}_x\}$$
$$(\!|\mathcal{C}|\!)_{(n)} = (\!|\mathcal{C}|\!)_{(n-1)} \cup \big\{\{\vec{P}\}f(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\} | f \in \mathcal{F}, arity(f) = k, \{\vec{\sigma}_i\}p_i\{\vec{\sigma}'_i; r_i\} \in (\!|\mathcal{C}|\!)_{(n-1)},$$
$$\exists \vec{R}_1,\ldots,\vec{R}_k.\textsc{Eval} \text{ of } f \text{ on } \{\vec{\sigma}_i * \vec{R}_i\}p_i\{\vec{\sigma}'_i * \vec{R}_i; \vec{r}_i\} \text{ is } \{\vec{P}\}f(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\}\big\}$$

Finally, given $(\!|\mathcal{C}|\!)$, we can define the OE-reduced space that SObEq will enumerate.

3.4 SObEq: observational equivalence with side effects

As we showed in Sec. 2, classical OE's $\equiv_{\vec{\mathcal{E}}}$ is no longer useful for the program space in Def. 2. We need our equivalence relation to encompass every aspect of the goal φ (Def. 1).

The generalized formalization of OE [45] defined two properties for equivalence, *interchangeability*: two equivalent programs can be used interchangeably within a larger program, and *consistency*: two equivalent programs will either both satisfy or both not satisfy the specification.

To satisfy consistency we must include everything that participates in Def. 1. The *value* returned by the expression, and the postcondition, which includes any constrained effect. We also need the state on which the program will be evaluated to check whether it describes $\vec{I}$.

As we saw in Sec. 2, to satisfy interchangeability we need the *location* of the value as well: while the literal $[2,3,1]$ and a variable `x` whose valuation is $[2,3,1]$ return the same value, they are not interchangeable under a larger program like `?.sort()`, where they would lead to different postconditions. Moreover, to be able to swap a program in, its postcondition must be the same.

Combined, then, our equivalence relation simply compares the triple excluding the program:

▶ **Definition 3** (SObEq equivalence relation). *The equivalence relation for* $\vec{\Phi} = \langle(\iota_i \to \omega_i, q_i)\rangle_i$ *is*

$$\{\vec{\sigma}_1\}p_1\{\vec{\sigma}'_1;\vec{r}_1\} \equiv_{\vec{\Phi}} \{\vec{\sigma}_2\}p_2\{\vec{\sigma}'_2;\vec{r}_2\} \iff \vec{\sigma}_1 = \vec{\sigma}_2 \wedge \vec{\sigma}'_1 = \vec{\sigma}'_2 \wedge \vec{r}_1 = \vec{r}_2$$

Or in other words, equivalence does not look at the program, only at its specifications.

Subsumption of triples. Finally, we consider the generality ordering of triples and (implicitly) of their equivalence classes, which will let us find more general programs.

In Sec. 2, we considered the two triples $tr_4 = \{\texttt{n} \mapsto 5\}\texttt{n - n}\{\texttt{n} \mapsto 5; 0@\bot\}$ and $tr_5 = \{emp\}\texttt{0}\{emp; 0@\bot\}$. While tr_4 and tr_5 are not observationally equivalent under Def. 3, as $emp \neq \texttt{n} \mapsto 5$, tr_5 is *more general* than tr_4: it produces the same result ($0@\bot$) and the same effect ($\emptyset$) from more concrete states. If tr_5 can absorb tr_4, we can arrive at a more compact representation of the space. To this end, we formally define this generality: the partial order of programs that differ by only a more general pre- and postcondition.

▶ **Definition 4** (Generality ordering of triples). *Given a specification $\vec{\Phi}$, we define*

$$\{\vec{P}_1\}p_1\{\vec{Q}_1; \vec{r}_1\} \sqsubseteq_{\vec{\Phi}} \{\vec{P}_2\}p_2\{\vec{Q}_2; \vec{r}_2\} \iff \exists \vec{R}.\{\vec{P}_1\}p_1\{\vec{Q}_1; \vec{r}_1\} \equiv_{\vec{\Phi}} \{\vec{P}_2 * \vec{R}\}p_2\{\vec{Q}_2 * \vec{R}; \vec{r}_2\}$$

I.e., the more general way to achieve the same result and effect is considered larger.

While $\sqsubseteq_{\vec{\Phi}}$ is clearly not an equivalence relation (the existence of a frame axiom is not symmetrical) so it cannot be used to partition the space into classes, we notice that replacing a triple with one that is larger according to $\sqsubseteq_{\vec{\Phi}}$ does preserve the two properties of an OE-relation, *interchangeability* and *consistency*, in one direction. Originally [45], *interchangeability* states that swapping two equivalent programs within a larger program will yield an equivalent larger program, and *consistency* states that two equivalent programs either both satisfy a given specification φ or neither does. Under $\sqsubseteq$, however, a slightly weaker – but still useful – version holds.

Interchangeability under $\sqsubseteq_{\vec{\Phi}}$ means that given $f \in \mathcal{F}, arity(f) = k$, k triples $tr_j = \{\vec{P}_j\}p_j\{\vec{Q}_j; \vec{r}_j\}$, $1 \leq j \leq k$, and k assertions $\vec{R}_j$ s.t. Eval can be applied on f and $\{\vec{P}_j * \vec{R}_j\}p_j\{\vec{Q}_j * \vec{R}_j; \vec{r}_j\}$ to infer $\{\vec{P}\}f(p_1, \ldots, p_k)\{\vec{Q}; \vec{r}\}$, and a triple $tr'_i = \{\vec{P}'_i\}p'_i\{\vec{Q}'_i; \vec{r'}_i\}$ s.t. $tr_i \sqsubseteq_{\vec{\Phi}} tr'_i$ for some $1 \leq i \leq k$, then:

1. There exists $\vec{R'}_i$ s.t. Eval can still be applied on f when replacing $\{\vec{P}_i * \vec{R}_i\}p_i\{\vec{Q}_i * \vec{R}_i; \vec{r}_i\}$ with $\{\vec{P'}_i * \vec{R}'_i\}p'_i\{\vec{Q'}_i * \vec{R}'_i; \vec{r'}_i\}$ to infer $\{\vec{P'}\}f(p_1, \ldots, p'_i, \ldots, p_k)\{\vec{Q'}; \vec{r'}\}$, and
2. $\{\vec{P}\}f(p_1, \ldots, p_k)\{\vec{Q}; \vec{r}\} \sqsubseteq_{\vec{\Phi}} \{\vec{P'}\}f(p_1, \ldots, p'_i, \ldots, p_k)\{\vec{Q'}; \vec{r'}\}$

Consistency under $\sqsubseteq_{\vec{\Phi}}$ means that for goal $\mathcal{G}$, if $\{\vec{P}_1\}p_1\{\vec{Q}_1; \vec{r}_1\} \sqsubseteq_{\vec{\Phi}} \{\vec{P}_2\}p_2\{\vec{Q}_2; \vec{r}_2\}$, then if there exists $\vec{R}_1$ s.t. $\{\vec{P}_1 * \vec{R}_1\}p_1\{\vec{Q}_1 * \vec{R}_1; \vec{r}_1\}$ matches $\mathcal{G}$, then there exists $\vec{R}_2$ s.t. $\{\vec{P}_2 * \vec{R}_2\}p_2\{\vec{Q}_2 * \vec{R}_2; \vec{r}_2\}$ matches $\mathcal{G}$. It's important to notice that the $\Leftrightarrow$ of the equivalence version of *consistency* is now $\Rightarrow$: if $\{\vec{P}_1\}p_1\{\vec{Q}_1; \vec{r}_1\}$ is a solution, so is $\{\vec{P}_2\}p_2\{\vec{Q}_2; \vec{r}_2\}$, but since $\{\vec{P}_2\}p_2\{\vec{Q}_2; \vec{r}_2\}$ is more general $\vec{P}_2$ might describe $\vec{I}$ even if $\vec{P}_1$ did not.

▶ **Lemma 5.** Interchangeability *and* consistency *are preserved under* $\sqsubseteq$.

The proof of Lemma 5 is found in the extended version of the paper.

With the program space $(\!|\mathcal{C}|\!)$ and the SObEq equivalence relation and generality ordering, we are ready to search for a program, i.e., to construct the OE-reduced space via enumeration.

4 The SObEq Enumeration

In this section, we describe the construction of the OE-reduced space. We first describe the structure of the space, then the enumeration, and finally how the enumeration can find equivalence classes that, by virtue of their generality, subsume other equivalence classes.

Classical Bottom-Up Enumeration. A bottom-up enumeration consists of applying each $f \in \mathcal{F}$ to k-tuples of child programs from the bank of previously seen programs, where $\mathit{arity}(f) = k$. Each newly enumerated $f(p_1, \ldots, p_k)$ is evaluated, and if an equivalence class labeled by its observed values is not yet represented in the bank, it is added to the bank. Enumeration is done according to some ordering, e.g., height, number of AST nodes, or probability score [7]. This is usually facilitated by the structure of the bank to allow for easier retrieval of relevant child-programs.

The precondition bank. In our setting, we want to bank discovered triples according to their precondition: a bank section $\vec{a}$ contains all discovered programs whose precondition is $\vec{a}$.

There are two reasons for this: first, when composing a k-tuple of arguments to f, finding a sequence Eval is enabled for can be done constructively, without considering all possible k-tuples. For a sequence of length $k-1$ with postcondition assertion $\vec{Q}_{k-1}$, we can preemptively rule out assertion vectors $\vec{a}$ with different values assigned to a variable appearing in both $\vec{Q}_{k-1}$ and $\vec{a}$, and will never form a valid sequence. In Algorithm 1 we call this construction CollectChildren.

Second, and even more importantly, this lets us track what assertions have been discovered by the enumeration. If the enumeration produces a triple $\{\vec{a}\}p\{\vec{a}';\vec{r}\}$ where $\vec{a}$ or $\vec{a}'$ have not yet been seen by the enumeration, then a new bank will be initialized for $\vec{a}$ (or $\vec{a}'$, resp.).

Initializing a new bank. A new assertion $\vec{a}$ can be discovered by the enumeration in one of two ways, as a pre- or a postcondition. For example, the derivation in Fig. 3 combines $\{\mathtt{n} \mapsto 5\}\mathtt{n\ -\ 1}\{\mathtt{n} \mapsto 5; 4\}$ with $\{\mathtt{arr} \mapsto \mathit{arr}_{\mathit{sort}}\}\mathtt{arr}\{\mathtt{arr} \mapsto \mathit{arr}_{\mathit{sort}}; \mathit{arr}_{\mathit{sort}}\}$ into $\{\mathtt{arr} \mapsto \mathit{arr}_{\mathit{sort}} * \mathtt{n} \mapsto 5\}\mathtt{arr[n-1]}\{\mathtt{arr} \mapsto \mathit{arr}_{\mathit{sort}} * \mathtt{n} \mapsto 5; 100\}$. Both the compound pre- and postcondition might be new for the enumeration, in which case they do not have a section in the bank.

If $\vec{a}$ is discovered as a precondition, we naturally add a section of the bank to bank the newly enumerated program. If $\vec{a}$ was discovered as a postcondition, then it may be a result of an effect, which means some of its atomic heaplets may not be in the bank yet. In this case, we create a section for $\vec{a}$ in the bank, but also add its atomic heaplets to the bank if they are new: we construct $\{a_x\}x\{a_x; r_x\}$ where $a_x = \langle x \mapsto a_i(x)\rangle_i$ and $r_x = \langle a_i(x)\rangle_i$ for each x in $\vec{a}$, and add it to the bank if a_x is not yet in the bank. In Algorithm 1, this is collectively denoted as InitBank($\vec{a}$).

At the beginning of the enumeration, the bank is trivially initialized with the empty assertion $\vec{emp}$ for literals and with $\langle x \mapsto \iota_i(x)\rangle_i$ for each variable $x \in \mathcal{V}$.

Enumerating terms. The main enumeration is not very different from the CBE enumeration: the enumerator selects $f \in \mathcal{F}$ with $\mathit{arity}(f) = k$, and composes as arguments for it all k-tuples of triples $\{\vec{P}_i\}p_i\{\vec{Q}_i;\vec{r}_i\}$ that Eval can be applied to from triples already in the bank.

The application of Eval then constructs a new triple $\{\vec{P}\}f(p_1, \ldots, p_k)\{\vec{Q};\vec{r}\}$ that can then be *observed*: we use its pre- and postcondition assertions and its results as the label for its equivalence class. This label is searched for in the bank for precondition $\vec{P}$. If the bank for $\vec{P}$ has an equivalent program, $\{\vec{P}\}f(p_1, \ldots, p_k)\{\vec{Q};\vec{r}\}$ is discarded, and if no equivalent program was previously seen, $\{\vec{P}\}f(p_1, \ldots, p_k)\{\vec{Q};\vec{r}\}$ is added to the bank for $\vec{P}$.

We also have at our disposal $\sqsubseteq_{\vec{\Phi}}$ (Def. 4), which means we reduce the space even further.

Algorithm 1 The SObEq enumeration.

Input : Set of specifications $\vec{\Phi} = \langle(\iota_i \to \omega_i, q_i)\rangle_i$, variables $\mathcal{V}$ specified in example inputs
Output : Program p composed of components $\mathcal{C}$ s.t. p that satisfies $\vec{\Phi}$
Globals: Precondition bank *bank*, mapping from assertion to an OE bank

```
1  initialAsserts := {emp} ∪ {⟨x ↦ ι_i(x)⟩_i | x ∈ V}
2  foreach ā ∈ initialAsserts do InitBank(ā);
3  repeat // until timeout
4      foreach f ∈ F, arity(f) = k do
5          foreach {σ̄_1 * R̄_1}p_1{σ̄'_1 * R̄_1; r̄_1} ··· {σ̄_k * R̄_k}p_k{σ̄'_k * R̄_k; r̄_k} ∈
             CollectChildren(bank, k) do
6              {σ̄}p{Q̄; r̄} := Eval (f, {σ̄_1 * R̄_1}p_1{σ̄'_1 * R̄_1; r̄_1} ··· {σ̄_k * R̄_k}p_k{σ̄'_k * R̄_k; r̄_k})
7              if ∃R̄.{σ̄ * R̄}p{Q̄ * R̄; r̄} matches G_Φ̄ then return p;
8              if ∀ā ∈ keys(bank).∃R̄.(ā * R̄ = σ̄ ∧ R̄ ⊆ Q̄) ⇒ (ā, Q̄ \ R̄, r̄) ∉ bank(ā) then
9                  if σ̄ ∉ keys(bank) then InitBank(σ̄);
10                 bank(σ̄) += {σ̄}p{Q̄; r̄}
11             end
12             if Q̄ ∉ keys(bank) then InitBank(Q̄);
13         end
14     end
15 until timeout;
```

Using more general triples. We want to use the $\sqsubseteq_{\vec{\Phi}}$ relation to let more general triples subsume less general ones in the course of enumeration, even when they are not strictly equivalent. However, this can happen in one of two cases: (i) the less general triple being enumerated after the more general triple is in the bank, and (ii) a more general triple being enumerated later.

Handling (i) is simple enough: when enumerating $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$, even if there is no representative for the equivalence class of $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$, the enumerator checks whether some $\{\vec{\sigma}_2\}p_2\{\vec{\sigma}'_2;\vec{r_2}\}$ s.t. $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \sqsubseteq_{\vec{\Phi}} \{\vec{\sigma}_2\}p_2\{\vec{\sigma}'_2;\vec{r_2}\}$ already exists in the bank, and if it does, discards $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$. This is efficiently done by changing the assertions to more general ones and searching the bank.

Handling (ii), however, is a larger feat. *In theory*, once we enumerate the more generic triple $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$, we can replace all less-general triples $\{\vec{\sigma}_2\}p_2\{\vec{\sigma}'_2;\vec{r_2}\}$ with $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$, propagating this change to any compositions that use p_2. This would be prohibitively costly on its own before the implications of Lemma 5: this replacement can weaken the pre- and postconditions of compositions, changing their place in the bank. This cascade might even weaken the precondition a triple that did not satisfy $\vec{\Phi}$ enough to describe $\vec{I}$ and become a solution. As such, we elect to only incorporate (i) into SObEq.

4.1 The enumeration algorithm

We can now compose the complete SObEq enumeration, seen in Algorithm 1.

Lines 1–2 of the algorithm initialize the enumeration by calling InitBank, described above, to create the basic programs for $\vec{emp}$ and for each of the variables' initial values.

Line 4 selects the next component for application; components with an arity of 0 are handled exclusively by InitBank. Next, on line 5 CollectChildren constructs all k-tuples of children that Eval is enabled for. Since it is a part of the construction process, CollectChildren returns each k triples already updated with the frame axioms $\vec{R}_1, \ldots, \vec{R}_k$ to a unified composed heap.

Line 6 then applies Eval to construct the new triple. If the new triple is a solution to the task, it will be returned by line 7. If not, line 8 checks whether an equivalent or subsuming program already exists in the bank via its equivalence class label: if some existing key in the bank can be made equivalent to the current program via a frame axiom. For example, the program $\{\mathtt{n} \mapsto 5\}n - n\{\mathtt{n} \mapsto 5; 0@\bot\}$ will be enumerated, and will then be tested twice: for $R = emp$, $(\mathtt{n} \mapsto 5, \mathtt{n} \mapsto 5, 0@\bot)$ will be searched for in the bank, and also for $R = n \mapsto 5$, $(emp, emp, 0@\bot)$ will be searched and found (for the program `0`), so $\{\mathtt{n} \mapsto 5\}n - n\{\mathtt{n} \mapsto 5; 0@\bot\}$ is discarded. A precondition $\vec{a}$ in the bank is deemed relevant if there exists a frame axiom $\vec{R}$ s.t. $\vec{R}$ both matches $\vec{a}$ to $\vec{\sigma}$ and all values in $\vec{R}$ are unchanged in $\vec{Q}$ (i.e., $\vec{R}$ truly is a frame). Every $\vec{a}$ in the bank lookup uses the relevant equivalence class label, with $\vec{a}$ as its precondition and subtracting the frame axiom from its postcondition assertion. If no triple is found, the new triple is added to the bank. If either the pre- or postcondition assertion do not exist in the bank, InitBank is called for them to generate triples for their variables.

SObEq as a base enumerator. Many synthesizers [16, 45, 36, 5] use a base OE enumeration of expressions to synthesize additional language constructs, e.g., conditionals, higher-order functions, and assignments. These techniques are parameterizable in their base enumerator, and so could leverage SObEq to expand their target language.

Correctness of SObEq. We show that Algorithm 1 is *correct*: that if a solution p exists in the unreduced space, SObEq's enumeration will find a solution observationally equivalent to p or more general than p.

▶ **Theorem 6** (Correctness of SObEq). *If a valid solution $\{\vec{\sigma}\}p\{\vec{\sigma}'; \vec{r}\}$ to goal $\mathcal{G}_{\vec{\Phi}}$ exists in $(\!|\mathcal{C}|\!)$, then there exists $\{\vec{\sigma}''\}p'\{\vec{\sigma}'''; \vec{r}'\}$ that matches $\mathcal{G}_{\vec{\Phi}}$ in the SObEq-reduced space, and it is reachable by SObEq's enumeration.*

In the extended version of the paper we prove Theorem 6 in two steps: first we define the (unreduced) space reachable via enumeration and show that any solution will still be in that space, then we apply the reduction of $\equiv_{\vec{\Phi}}$ to the space, defining the *SObEq-reduced program space* $(\!|\mathcal{C}|\!)^S$, and prove that any solution will have an observationally equivalent or more general under $\sqsubseteq_{\vec{\Phi}}$ solution in $(\!|\mathcal{C}|\!)^S$.

5 Implementation

We implemented a SObEq synthesizer for JavaScript programs in 3,700 lines of Scala code. The set of functions and operators $\mathcal{F}$ contains 54 components for integers, strings, arrays, and sets, including nine mutating operations: postfix increment for integers, the array functions `sort`, `reverse`, `splice`, `push`, `pop`, `shift`, and the set functions `add` and `delete`. JavaScript strings are immutable, so $\mathcal{F}$ includes no mutating components for strings. $\mathcal{F}$ also includes a sequence operator for any two expressions. We allow a task to provide additional string constants for *lits*, common in benchmarks from the literature [4, 7] or interaction models [17, 16].

Preserving immutability. Effect constraints in $\vec{\Phi}$ can specify that a variable has not changed in the postcondition of the solution. But effect constraints cannot distinguish between, e.g., `x` and `--(++x)` because we only reason about the pre- and postcondition, not the path between them. This is generally desirable as it marks `--(++x)` as redundant, but this will only happen when applying `--`. The user may want `x` to *always* be immutable.

This can be enforced by the enumeration. We add to our task specifications the ability to mark a variable as `immut`. Once the user specifies `x` as immutable, there is no reason to construct `++x` at all: a triple $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$ where $\langle\sigma_i(x)\rangle_i \neq \langle\sigma'_i(x)\rangle_i$ is discarded immediately, and $\vec{\sigma}'$ is not initialized in the bank, since any programs with the precondition $\vec{\sigma}'$ assume a mutation of `x`.

Goals without an output. Under CHSL it is possible to express a goal that has no specified output, only a specified effect. This is desirable when, e.g., specifying that a value is inserted into a list, but what the insertion returns is irrelevant. If a benchmark has no specified outputs, i.e., $\vec{\Phi} = \langle(\iota_i, q_i)\rangle_i$, we create the synthesis goal $\mathcal{G}_{\vec{\Phi}} = \{\vec{P}\}_\{\langle q_i * _\rangle_i;_\}$. The lack of an output does not change anything in how programs are enumerated and the goal is matched in the same way.

6 Experimental Evaluation

We evaluate SOBEQ on our implementation. All our experiments used a server with Intel Xeon Gold 6338 2 GHz[1] and 128GB of RAM, with the JVM maximum heap size set to 110 GB.

Research Questions. We aim to answer the following research questions:

RQ1 How does SOBEQ compare to the state-of-the-art in syntax-guided synthesis of heap mutations?

RQ2 What is the overhead of SOBEQ compared to a classical OE enumeration?

RQ3 How effective is SOBEQ at pruning the space of programs?

RQ4 How does SOBEQ compare to enumerating with concrete states?

6.1 Benchmarks and baselines

We evaluated SOBEQ on a set of 63 benchmarks, curated from previous synthesizers [52, 7, 16, 22], programming exercises on LEETCODE.COM, and questions on STACKOVERFLOW.COM. We discarded benchmarks that use types other than strings, arrays and sets, i.e., that would be inherently unsupported by our implementation. As SOBEQ supports a straight-line sequence of expressions, we also discarded benchmarks from Shi et al. [52] and Ferdowsifard et al. [16] if their solution used assignments to intermediate variables, loops, or conditionals.

We split the benchmarks into two sets: ▷ ***May* Mutate**: 45 benchmarks that have a non-mutating solution in $\mathcal{C}$. ▷ ***Must* Mutate**: 18 benchmarks that require mutation: benchmarks with a non-empty effect constraint or cannot be solved without mutation in $\mathcal{C}$. Of these, five are benchmarks from *May* with a constrained effect, ruling out non-mutating solutions. For RQ2, we also create ▷ ***Pure***: a version of *May* where all variables are marked as `immut`.

On average, *May* benchmarks have 2.8 examples (min 1, max 4) and 1.4 variables (min 1, max 3). *Must* benchmarks have 2.7 examples (min 1, max 5), 2.3 variables (min 1, max 4), and 1.2 variables specified in the effect (min 0, max 2). Five *Must* benchmarks have no specified outputs. More information about our benchmarks is provided in the extended version of the paper.

[1] This processor has 32 cores/64 threads, but our implementation is single-threaded.

Selecting a baseline. We considered several synthesizers as baselines for SOBEQ, ruling out synthesizers built for a single example [22], synthesizers that mutate by assigning synthesized pure expressions [16, 17, 59], and tools hinging on Java's rich type system [15, 61]. FRANGEL [52] is the current state of the art in synthesis of general programs with heap mutations: it is specified with input-output examples, it has the same ability as SOBEQ to declare a variable as immutable, and its stochastic search carries no additional requirements for specification other than the ability to evaluate – just as SOBEQ does. This makes it the most suitable baseline.

FrAngel as a baseline. FRANGEL targets Java collection implementations, which have many more methods than their JavaScript counterparts. This means FRANGEL and SOBEQ can yield solutions that are entirely incomparable. To ensure an apples-to-apples comparison, we created new Java classes that mimic JavaScript types in Java, recreating the same $\mathcal{C}$ used by SOBEQ's implementation. We then defined our benchmark set over these types.

In addition, in order to force FRANGEL to create a straight-line sequence of expression statements as SOBEQ does, we also removed from its enumeration loops, conditionals, and assignments. Notice that removing choices from its random sampling only improves FRANGEL's odds of finding a solution; before this additional change, FRANGEL's run times were considerably worse.

FRANGEL also has the option to set inputs as immutable, which let us encode `immut` variables, and an option to set a target value for an input, which we used for encoding effect constraints in the benchmark. This means FRANGEL can run on every benchmark in SOBEQ's suite.

Selecting timeouts. We consider two timeouts for our runs. First, we set our timeout at 60 minutes, extending the original 30 minute timeout used to evaluate FRANGEL. However, since we aim SOBEQ to be used in interactive systems, and by other OE-based systems as the base enumerator, we also consider results at an *interactive timeout* or a timeout needed for the synthesizer to run in a user-facing tool and not create a *disruptive interruption* [43]. Building on recent interactive synthesizers, we choose 7 seconds as our interactive timeout.

6.2 RQ1: Comparison to FrAngel

We compare SOBEQ to state of the art FRANGEL, using an equivalent component set as detailed in Sec. 6.1. FRANGEL's algorithm is stochastic, so we ran FRANGEL five times on each of the 63 benchmarks. We consider a benchmark *solved* if the synthesizer returns a solution that is correct on all examples. While some benchmarks from the literature contained a gold standard solution, we only used those as part of the analysis of result quality. We compare the quality of SOBEQ's solutions to FRANGEL on three counts: the size of the solution, is it *overfitted*, and does it contain *spurious operations*. These are shown in Fig. 5. In addition, we compare the run times of the two tools at both an interactive 7s and a one-hour timeout.

Size of the solution. While the size of solutions is not, in itself, a good proxy for quality, it is telling in combination with overfitting and spurious operations. SOBEQ postprocessed solutions had an average of 6.2 AST nodes at an interactive timeout and 6.9 at a 1-hour timeout (max 14, 25, resp.) where *Must* benchmarks were generally smaller – 5.7 nodes at the interactive timeout and 6.1 in an hour – compared to *May* benchmarks – 6.4 and 7.1 nodes.

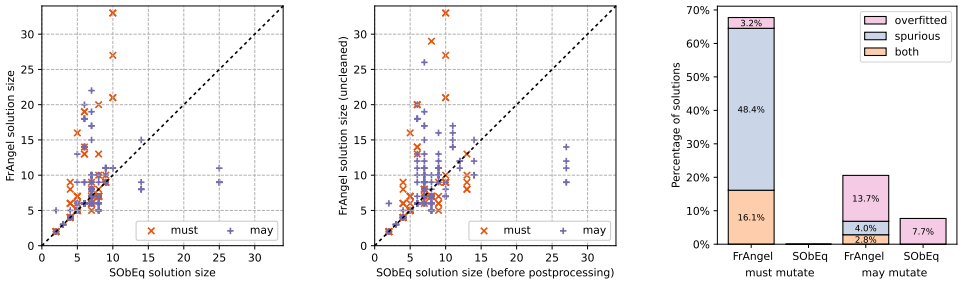


Figure 5 RQ1: Comparing SObEq's solutions to FrAngel's on size (left, center) and quality (right). Solution sizes are per-benchmark: each SObEq solution has up to five corresponding FrAngel values. Above the line means SObEq is smaller. We consider both the final results (left) and solutions before postprocessing/cleaning steps (center). Solution quality (right) indicates the percentage of solutions with overfitted or spurious operations, determined by manual inspection (lower is better).

The left and center of Fig. 5 show the AST size for each SObEq solution compared to each of FrAngel's solutions for the same benchmark. Overall, FrAngel's solutions are much larger than SObEq's. There are two benchmarks where SObEq's solution is considerably larger than FrAngel's: *Is All Positive?* (originally from FrAngel), both SObEq's and FrAngel's solutions were overfitted. In *Get first name from name with comma* (originally from Probe [7]), FrAngel returned two overfitted solutions similar to SObEq's, both appearing closer to the diagonal line, and three times found a solution closer to the gold standard solution, which is shorter, though never the gold standard solution itself. (The two identical solutions and three identical solutions overlap in Fig. 5.) An interesting trend in SObEq's solutions was that the enumeration order favored extracting postfix increment to a sequence. E.g., instead of `buffer[idx++]++, idx` SObEq returns the larger but functionally identical `buffer[idx]++, idx++, idx`.

Overfitted solutions. We define overfitted solutions as solutions that do not generalize beyond the provided examples. This was determined by manual inspection of both SObEq and FrAngel solutions, and with reference to task descriptions and gold standard solutions, if they exist.

Two SObEq solutions were overfitted, and one additional benchmark was close to a gold standard solution for the task, but since all its inputs were of length 4, it replaced `str.length` in the gold standard solution with `4`. Altogether, 3 of SObEq's 54 solutions were overfitted (6%). Of 279 FrAngel solutions (up to five per benchmark, which sometimes differ) 47 were overfitted (17%). As the right of Fig. 5 shows, both tools were more prone to overfitting in *May* benchmarks.

We bring several examples of FrAngel's overfitted solutions here.

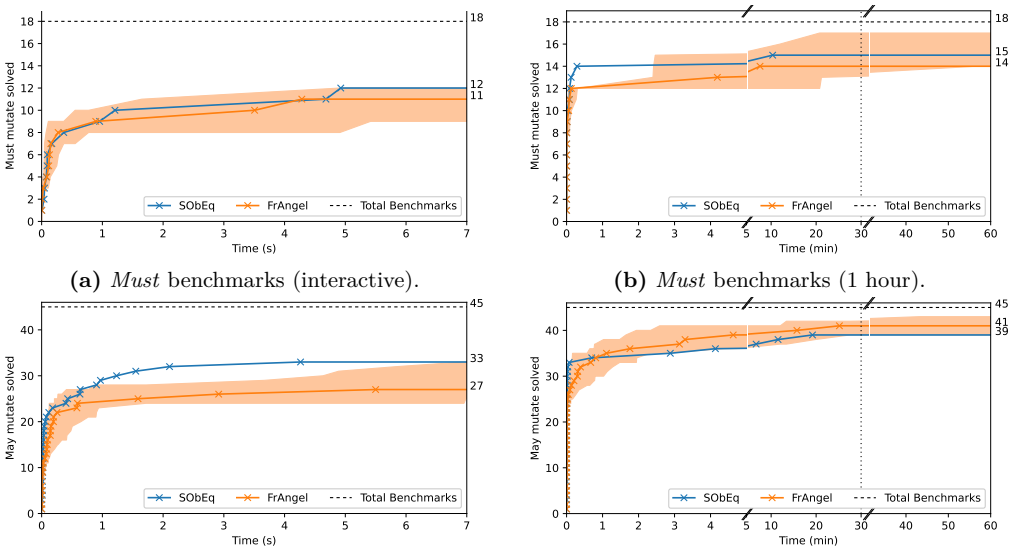
For example in the *Max and Min* benchmark, a simplified version of the example in Sec. 1, all four solutions (one run timed out) are overfitted and *different*. Two of them are:

```
return arr.join(arr.join(Str("").replaceAll(Str(""), Str("␣"))
  .concat(arr.join(Str("")).indexOf(arr.slice(Int(0), Int(1)).join(Str(""));
```

where the result number (max plus min) is generated by `indexOf`, and

```
arr.reverse();
return arr.join(arr.sort().join(Str("")).replaceAll(Str(""), Str("␣")).replace(
  arr.join(arr.join(Str("")).replaceAll(Str(""), Str("␣")), Str("␣")
).length();
```

where it is generated by the length of a string.



(a) *Must* benchmarks (interactive). **(b)** *Must* benchmarks (1 hour).

(c) *May* benchmarks (interactive). **(d)** *May* benchmarks (1 hour).

Figure 6 RQ1: The number of benchmarks SObEq and FrAngel solve over time. FrAngel is stochastic: median run for each benchmark shown by line with best and worst runs as range.

We also consider FrAngel's results on the three benchmarks that SObEq overfitted on: For the *Is All Positive?* benchmark mentioned above for its large solutions, both SObEq and FrAngel overfit to the examples in two different ways. FrAngel overfit all give solutions, returning `true` only for arrays where the smallest element is 1, and SObEq looked for a `'-'` in an overfitted location in the printed array. For *get from name with comma*, four of five FrAngel solutions were *not* overfit. The fifth is:

```
return _arg_0.split(Str(",")).sort().slice(Int(1).subtract(_arg_1),_arg_1).join(Str(""));
```

where the input string is split into words but then sorted before extracting the `1-_arg_1`th word. This happens to be true because all example inputs have a decreasing lexicographic ordering of the words. In *Negative Index Of*, where SObEq string lengths to `4`, three of FrAngel's solutions are not overfitted, but two rely on string `<` comparisons to between variables and constants.

Spurious operations. We define spurious operations as terms in the AST that do not contribute to reaching the target values: e.g., sorting an already-sorted list, mutating a temporary value that is then discarded, etc. Of the 279 FrAngel solutions, 37 had spurious operations (13%). We bring a few examples of these here.

One of the solutions FrAngel found for the *buffer-add* benchmark was:

```
arr.splice(beg++, 1, toadd).reverse();
return end + 1;
```

Here, `splice` removes a sub-array from `arr`, replaces it with `toadd`, and returns it. The solution then calls `reverse()` on this temporary value, which is immediately discarded.

Likewise, in three of the five solutions to *rotate queue*:

```
mod.equals(inp.push(inp.shift()));
```

where input variable `mod` is compared to the result of the target program (shifting the first element out of the array and pushing it back in at the end).

For *set add-delete*, which asks for an in-place mutation of a the set `seen`, one solution is:

```
seen.delete(s2).compareTo(seen.add(s1).has(null));
```

where both `has` and `compareTo` are useless, computing a result that is not returned.

And, of course, this problem is not disparate from overfitting; the only successful run of FrAngel for *fill fibonacci window* found this solution:

```
out.reverse().spliceReplace(
  Int(1),
  out.concat(out).length().subtract(
    Str("␣").concat(out.join(Str(""))).lastIndexOf((new IntArray(Int(1))).join(Str("")))
) ).shift();
```

where `reverse().spliceReplace(1,_)` is a consistent, if odd, way to remove the first element and add something after the second element by swapping the elements and then replacing the (now) second one. The second argument to `splice`, which is expected to compute the next fibonacci element, overfits to the examples, and `shift`ing on the discarded return value of `splice` is spurious.

SObEq, in contrast, had no spurious operations. Some are avoided thanks to OE: FrAngel can draw programs that perform round trip operations like reversing multiple times (3 of 5 solutions to the example from Sec. 1 do this) or sort a sorted list. Enumerating with OE discards these, as they return the heap to a state it was already at and return the same value.

Other spurious operations are avoided because of the implicit bias of a bottom-up search toward small programs: the program `inp.push(inp.shift())` already satisfies the input, so a bottom-up search would not try to compose it into a larger program adding on spurious operations. Likewise, the enumeration would try to compose `arr.splice(beg++, 1, toadd)` to `end + 1` before trying it with the larger `arr.splice(beg++, 1, toadd).reverse()`.

Runtimes. We compared SObEq's runtimes to FrAngel's best, worst and median performance on each benchmark in Fig. 6. In an interactive timeout, SObEq outperforms FrAngel for the majority of the 7 seconds. At the 7s limit, SObEq solves 45 benchmarks, 7 benchmarks more than FrAngel's median run, whereas FrAngel solves between 34 and 45 benchmarks. We also notice the main disadvantage of FrAngel, which is the variance introduced by the stochastic algorithm: a difference of 30% between its best and worst cases. At the 1-hour timeout, SObEq solves 54 benchmarks, compared to FrAngel's 53–60 (median 55).

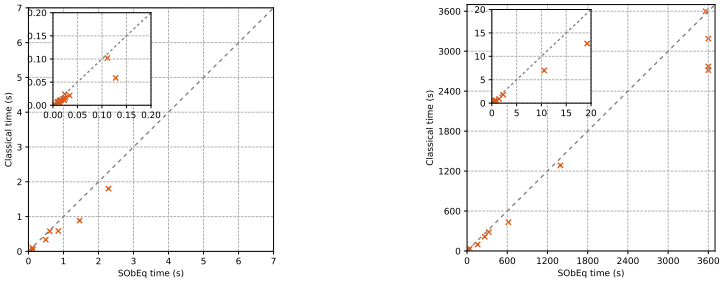
Fig. 6 separates benchmarks into sets to ensure performance is not at the expense of one set of benchmarks over the other, important for a general-purpose synthesizer. At an interactive timeout, SObEq performs on *Must* benchmarks as well as FrAngel's best run, both solving 12 of 18 benchmarks, and solves one more than FrAngel's median run of 11. For *May* benchmarks, SObEq solves 33 of 45, while FrAngel solves between 25 and 33 (median 27). After an hour, SObEq outperforms FrAngel's median run for *Must* benchmarks, solving 15 benchmarks to FrAngel's 14 and is within FrAngel's range for both benchmark sets.

To answer RQ1, **SObEq returns higher quality solutions than FrAngel at comparable runtimes, without the nondeterministic variance in both time and expected solution.**

6.3 RQ2: Overhead of SObEq

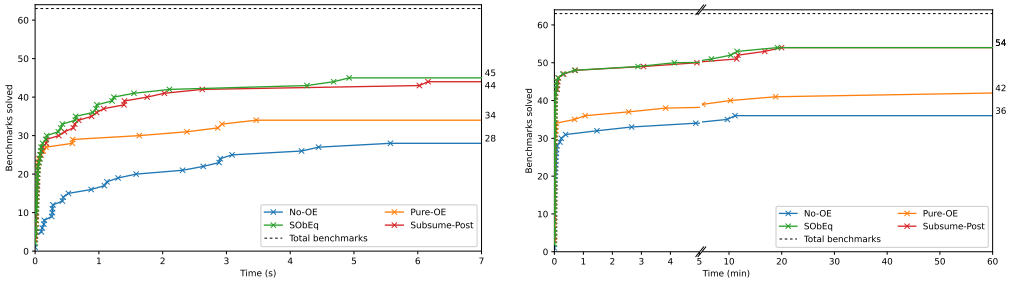
A more general algorithm inevitably carries an overhead on a (now) special case compared to specialized algorithms for that case. This is often because of additional work by an algorithm making fewer assumptions, but in OE this can also take the form of a more refined equivalence relation, which unifies fewer programs creating a bigger space to explore. We wish to measure this overhead in order to quantify the cost of the changes made to support mutations in SObEq.

If $\mathcal{C}$ has no mutating components, SObEq would not discover assertions outside the initial preconditions and their compositions. Such a search space is directly comparable to a classical bottom-up enumeration (CBE). This lets us measure the cost of the more



(a) Interactive timeout. **(b)** 1-hour timeout.

Figure 7 RQ2: Comparison with CBE: time to termination for each *Pure* benchmark with SObEq compared to a classical bottom-up enumeration. Points at (timeout,timeout) not shown.



(a) Interactive timeout. **(b)** 1-hour timeout.

Figure 8 RQ3: Run times of disabling different pruning mechanisms within SObEq: any OE, OE outside the initial state, and discarding programs according to $\sqsubseteq$.

fine-grained equivalence classes caused by using results, not just values, in equivalence class labels, and the additional overhead of creating a compound heap for sub-programs using different variables.

To measure the overhead, we restricted our $\mathcal{C}$ to only non-mutating components, and compared its performance on *Pure* benchmarks to an enumerator that uses only the initial *concrete* state and uses the classical value-based equivalence relation for an OE-reduction. This is equivalent classical OE with a height-based enumeration. By removing the mutating components for this evaluation, we avoid the correctness issues in classical OE, while controlling for performance differences caused by having different grammars.

In this restricted context CBE outperforms SObEq for most benchmarks (time comparison shown in Fig. 7). In an interactive timeout, CBE solved the benchmarks an average of 1.52 times faster (median 1.5). CBE was also able to solve one additional benchmark, *Max and Min*, within the 7s timeout whereas SObEq took 10.5s, and it is therefore not shown in Fig. 7. The biggest difference, where CBE was 2x faster or more (up to a maximum of 2.16 times faster), were all in benchmarks with runtimes of under 130ms for both enumerators.

In the full hour, CBE solved the benchmarks an average of 1.48 times faster (median 1.5), and finished with a two-benchmark lead: there are three benchmarks CBE solved that SObEq did not, and one benchmark that SObEq solved and CBE did not.

For RQ2, **we find that the overhead of SObEq's more fine-grained equivalence classes and creating compound heaps is not negligible compared to CBE.**

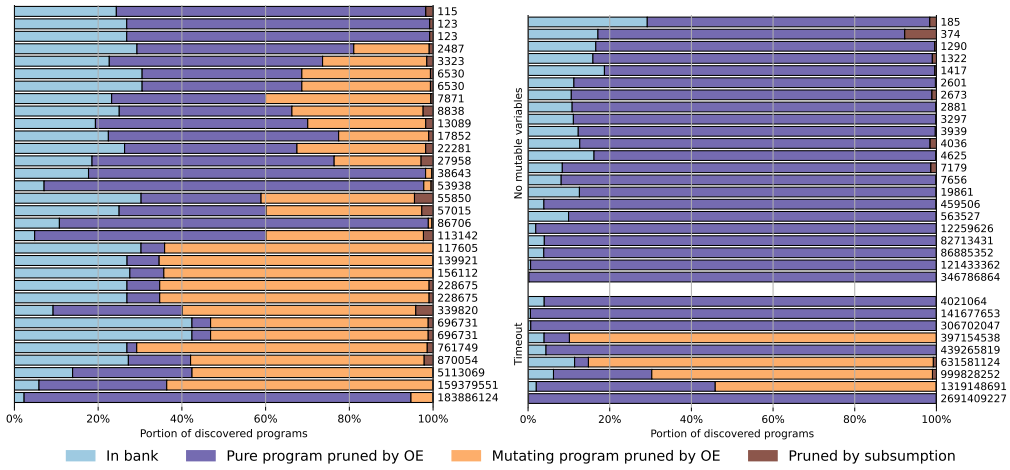


Figure 9 Handling of programs seen in the space of each benchmark (percentages plotted, no. programs in text). Benchmarks where mutation cannot occur and timeouts plotted separately.

6.4 RQ3: SObEq's effectiveness in pruning

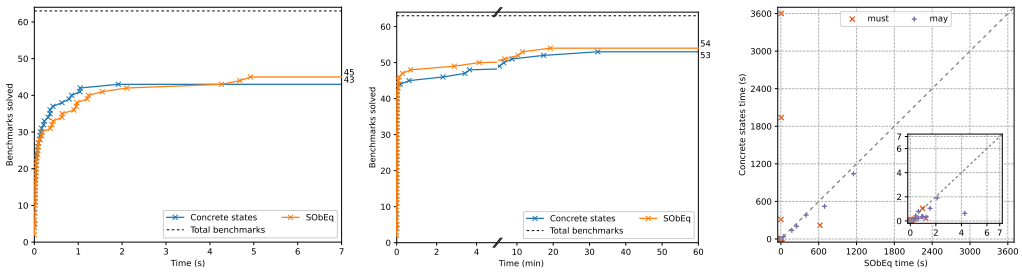
To test SObEq's effectiveness at pruning its space we compare SObEq to two ablation enumerations: 1. *No-OE*: an enumeration that discards no programs, i.e, enumerates $(\!|\mathcal{C}|\!)^{reach}$. 2. *Pure-OE*: an enumeration that only performs OE on pure programs, i.e., fully enumerates the space outside preconditions that cover the initial state. Additionally, to measure the effect of subsumption, we also add 3. *Subsume-Post*: an enumeration with the SObEq OE-reduction, but no subsumption during enumeration, only as postprocessing of results. The results are shown in Fig. 8.

It is, of course, unsurprising that *No-OE* performs poorly: with no reduction of the space, few programs are feasible, and this version solves only 28 benchmarks within 7s, diverging completely after 10 minutes. *Pure-OE* is an interesting case: testing whether it should prune a program required computing its effect, an expensive operation that leads *Pure-OE* to be considerably slower. At the 1-hour limit, *Subsume-Post* does as well as SObEq. However, it lags a little behind SObEq for most of its run, having solved one fewer benchmark at 7s.

In summary, **both the application of OE to the entire space and the application of subsumption help SObEq's performance, particularly in an interactive timeout.**

We also looked at what happens to the discovered programs across runs of SObEq, shown in Fig. 9, specifically to gauge how many programs are discarded and how many are subsumed by a more general program that already exists in the bank. We separate out runs that time out and benchmarks where all input variables are strings, i.e., no mutation is *possible*. In the latter, the number of mutating programs pruned is trivially 0, but subsumption still takes place.

On (weighted) average, only 3% of programs seen by SObEq are added to the bank (med 14%, min 0.1%, max 42% of the programs). 66% on average are pure programs pruned by equivalence (med 72%, min 2%, max 99.98%), and another 30% are mutating programs pruned by equivalence (med 1.5%, min 0%, max 90%). Finally, an average 0.2% of the programs are pruned by subsumption (med 0.4%, min 0.00002%, max 8%). **While the ratios vary, each of the pruning methods in SObEq is significant for some of the benchmarks.**



(a) Interactive timeout. **(b)** 1-hour timeout. **(c)** Per-benchmark comparison.

Figure 10 RQ4: Comparing an enumeration with preconditions to an enumeration with concrete states. Points at (timeout,timeout) not shown in (c).

6.5 RQ4: Enumerating with preconditions vs. concrete states

Finally, we examine the effect of the *local reasoning* in SObEq's enumeration. Composing partial heaps takes effort, but how costly is it, and does it allow more benchmarks to be solved? To test this, we created a version of SObEq where instead of footprints, preconditions are always *full* concrete states. We compare this to SObEq's enumeration in Fig. 10.

For most benchmarks, both enumerations perform similarly with concrete states slightly faster. We expected to see this as an effect of the overhead of composition, but it is also caused by pure programs being banked in one main state, which changes the enumeration order favorably for *May* benchmarks. If we consider only *Must* benchmarks, SObEq overtakes the concrete state enumeration after 1.5s and stays ahead for the entire 1-hour run.

Three benchmarks are noteworthy in this experiment. The concrete state enumeration times out on our overview example (benchmark *Max and Min with n*) while SObEq solves it in under 5s. Two more, *partition k largest* and *get range loopy while*, SObEq solves in 90ms and 4.6s, resp., and enumerating with concrete states takes 5min and 32min, resp. All three share the need to use multiple variables *after* one is mutated, which means subprograms already enumerated with the unmutated variable do not make a valid sequence and need to be re-enumerated.

To conclude, **SObEq's local reasoning is faster for benchmarks that need to mutate, and is not significantly slower for other benchmarks.**

7 Related work

Syntax- and Component-Guided Synthesis. Syntax-Guided Synthesis (SyGuS) [3] is a form of the synthesis problem providing a specification along with a grammar defining its program space. It can also describe a form of synthesis where programs are constructed by syntax rules. The latter is essentially identical to Component-Based Synthesis, where the synthesizer is provided with a list of functions and operations spanning the program space. Both of these formulations are tackled by many synthesis strategies: enumerative [35, 52, 7, 14, 25, 22, 28, 54, 9, 62, 34], SMT [30, 6, 50, 1] and SAT [56, 58] based, by traversing constructed representations of the space such as VSAs [31, 33, 63], version spaces [53] and Petri-nets [15, 24].

Proof-directed synthesizers. A specific form of enumerative synthesis is one that attempts to find a proof for a proof goal, where the proof dictates a program. This idea is not new [57], and some deductive type-driven synthesizers describe their search as a proof

search [24, 42]. Other works' goals are more expressive than types: refinement types [48], relational specifications [32], select/update operators [41], Separation Logic, and even natural language [10]. Importantly, proof searches are overwhelmingly *deductive*, searching the proof space top-down. To our knowledge, SObEq is the first synthesis proof search that enumerates the proof space bottom-up.

Synthesis with Separation Logic. Separation Logic proofs are a popular vehicle for proof-directed synthesis. They have been successfully used for parallelizing non-parallel code [8], but are most often used to synthesize heap-manipulating programs. SuSLik [49] performs a deductive search with inferences about the heap to generate programs performing pointer operations. Cypress [29] extends this with the ability to synthesize recursive auxiliaries, RoboSuSLik [11] extends it with read-only borrows, and RusSOL [20] applies it to Rust programs, leveraging Rust's type system to simplify the required specification from formal assertions to functional annotations of Rust functions. All of these accept assertions as their specification, hinge on manually-crafted inference rules, and employ solvers. SObEq, in contrast, works on input-output examples and relies only the generic Eval rule in conjunction with an interpreter.

Observational equivalence. Observational Equivalence [59, 2] is a popular way to reduce the size of the space when enumerating bottom-up. Originally OE works on examples, but RESL [45] generalized it to other specifications on programs as well; this generalization also introduced the notions of *interchangeability* and *consistency* that SObEq's correctness relies on. SyMetric [18] relaxes OE to observational similarity by choosing representatives via clustering instead of equivalence, then repairing the resulting program. While this compacts the space like SObEq's subsumption, correctness of SyMetric's solutions hinges on the success of its repair step.

Many of the synthesizers that use OE only synthesize pure programs [46, 7, 45, 2, 60, 40] (we include [45] in this list, despite its misuse of JavaScript's `Array.sort`), others employ synthesis with OE to produce pure expressions that are then used in assignments or other larger mutating expressions [37, 38, 59, 17, 16], creating a top-level program with memory mutations.

Arborist [36] and Lens [47] relax the purity assumption in interesting ways. Arborist tackles the specific problem of dependent loops (e.g., *fold*) where the composition of smaller programs discovers new inputs and changes the equivalence classes. Arborist uses finite tree automata to unify programs without discarding them, which allows it to iteratively break up equivalence classes. Arborist finds a solution if its iterative refinement terminates. Lens take advantage of its target domain (straight line assembly code) to treat operations as having no arguments, making sequences the only composition. In this representation, OE values are full register valuations, and synthesis of state-mutating programs reduces to the OE enumeration used for pure programs. Both hinge on the ability to represent equivalence classes *without* discarding any programs using, implementing OE without a bank, which is only tractable under very strict assumptions on the set of components and on values.

Synthesis with heap mutations. The problem of synthesizing mutating snippets has been worked around in many ways. SL proof-directed approaches assume a limited target language of mutations available to the proof. Sketch[56], Transit [59], SnipPy [17], Simpl [55] and Cozy [37, 38] assume expressions are pure, then use them in specific mutating contexts validated by other means (e.g., SMT). SyPet [15] and EdSynth [61] rely on types for getting programs, and only then evaluate them. Likewise, FrAngel [52] draws random programs, then evaluates them.

LooPy's [16] problem statement bounds the intermediate mutations, and is to enumerate independent paths between these states. Similarly Lens [47] reduces all registers to 3-bit and constructs the full space of intermediate states. Cobalt [41] also separates pure expressions from mutations using provided API annotations. Toshokan [27] encodes the specifications for mutating methods as two versions of the function, one for changes to the state and the other the for result, akin to the two parts of SObEq's postcondition.

CodeHint [22] is, in some ways, the closest to our approach: it defines expressions as equivalent if they have the same value and side effects in the *current state*, or, in our terminology, an equivalence relation on both before- and after-states. How current states are used in enumeration is not defined in the paper, but their implementation uses programs enumerated on the initial state to populate their bank for new after-states, replicating the problem discussed in Sec. 2.1. Moreover, CodeHint's focus is its interactivity, and its interaction is designed around a single example, with any additional information only filtering an initial list of programs, not informing the OE enumeration. This is why we ruled it out as a baseline in Sec. 6.1.

SObEq sheds assumptions that expressions are pure and that intermediate states are known in advance. Moreover, SObEq has the formal guarantee in Theorem 6, unlike the two most general existing solutions: FrAngel, which randomly draws programs, and CodeHint, which performs OE only on its first example and mutated states in an unprincipled manner.

Other side effects like file or database writes have been a challenge in program synthesis. RbSyn [25] addresses database writes via a type-guided top-down enumeration from a syntax including *effect holes* that can be filled with annotated mutating methods; expressions not within effect holes have a purity assumption, limiting how effects are composed, unlike SObEq that can use mutating operations anywhere. CodeHint allows users to enable effects on the file system in their enumeration, even though they cannot be undone like memory mutations are.

References

1 Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample guided inductive synthesis modulo theories. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 270–288, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-96145-3_15.

2 Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *International conference on computer aided verification*, pages 934–950. Springer, 2013. doi:10.1007/978-3-642-39799-8_67.

3 Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. *Dependable Software Systems Engineering*, 40:1–25, 2015. doi:10.3233/978-1-61499-495-4-1.

4 Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2017: Results and analysis. *arXiv preprint*, 2017. arXiv:1711.11438.

5 Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I 23*, pages 319–336. Springer, 2017. doi:10.1007/978-3-662-54577-5_18.

6 Haniel Barbosa, Andrew Reynolds, Daniel Larraz, and Cesare Tinelli. Extending enumerative function synthesis via SMT-driven classification. In *2019 Formal Methods in Computer Aided Design (FMCAD)*, pages 212–220. IEEE, 2019. doi:10.23919/FMCAD.2019.8894267.

7 Shraddha Barke, Hila Peleg, and Nadia Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020. doi:10.1145/3428295.

8 Matko Botinčan, Mike Dodds, and Suresh Jagannathan. Proof-directed parallelization synthesis by separation logic. *ACM Trans. Program. Lang. Syst.*, 35(2), July 2013. doi:10.1145/2491522.2491525.

9 José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. FlashFill++: Scaling programming by example by cutting to the chase. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571226.

10 João Costa Seco, Jonathan Aldrich, Luís Carvalho, Bernardo Toninho, and Carla Ferreira. Derivations with holes for concept-based program synthesis. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 63–79, 2022. doi:10.1145/3563835.3567658.

11 Andreea Costea, Amy Zhu, Nadia Polikarpova, and Ilya Sergey. Concise read-only specifications for better synthesis of programs with pointers. In Peter Müller, editor, *Programming Languages and Systems*, pages 141–168, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-44914-8_6.

12 Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *Proceedings of the 2020 CHI conference on human factors in computing systems*, pages 1–12, 2020. doi:10.1145/3313831.3376442.

13 Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices*, 53(4):420–435, 2018. doi:10.1145/3192366.3192382.

14 Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. *ACM SIGPLAN Notices*, 52(6):422–436, 2017.

15 Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas Reps. Component-based synthesis for complex APIs. In *Proceedings of the 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2017*, 2017.

16 Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. LooPy: Interactive program synthesis with control structures. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA), October 2021. doi:10.1145/3485530.

17 Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. Small-step live programming by example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pages 614–626. Association for Computing Machinery, 2020. doi:10.1145/3379337.3415869.

18 Jack Feser, Işıl Dillig, and Armando Solar-Lezama. Inductive program synthesis guided by observational program similarity. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):912–940, 2023. doi:10.1145/3622830.

19 John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices*, 50(6):229–239, 2015. doi:10.1145/2737924.2737977.

20 Jonáš Fiala, Shachar Itzhaky, Peter Müller, Nadia Polikarpova, and Ilya Sergey. Leveraging Rust types for program synthesis. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi:10.1145/3591278.

21 Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2016.

22 Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. CodeHint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 653–663, 2014.

23 Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1926385.1926423.

24 Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. Program synthesis by type-guided abstraction refinement. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28, 2019. doi:10.1145/3371080.

25 Sankha Narayan Guria, Jeffrey S Foster, and David Van Horn. RbSyn: type-and effect-guided program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 344–358, 2021. doi:10.1145/3453483.3454048.

26 Sankha Narayan Guria, Jeffrey S Foster, and David Van Horn. Absynthe: Abstract interpretation-guided synthesis. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1584–1607, 2023. doi:10.1145/3591285.

27 Kangjing Huang and Xiaokang Qiu. Bootstrapping library-based synthesis. In *International Static Analysis Symposium*, pages 272–298. Springer, 2022. doi:10.1007/978-3-031-22308-2_13.

28 Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. Reconciling enumerative and deductive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 1159–1174, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3385412.3386027.

29 Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. Cyclic program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 944–959, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454087.

30 Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 215–224. IEEE, 2010. doi:10.1145/1806799.1806833.

31 Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, and Zhenjiang Hu. Question selection for interactive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1143–1158, 2020. doi:10.1145/3385412.3386025.

32 Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 407–426, 2013. doi:10.1145/2509136.2509555.

33 Tessa A Lau, Pedro M Domingos, and Daniel S Weld. Version space algebra and its application to programming by demonstration. In *ICML*, pages 527–534, 2000.

34 Woosuk Lee. Combining the top-down propagation and bottom-up enumeration for inductive program synthesis. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–28, 2021. doi:10.1145/3434335.

35 Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. *ACM SIGPLAN Notices*, 53(4):436–449, 2018. doi:10.1145/3192366.3192410.

36 Xiang Li, Xiangyu Zhou, Rui Dong, Yihong Zhang, and Xinyu Wang. Efficient bottom-up synthesis for programs with local variables. *Proceedings of the ACM on Programming Languages*, 8(POPL), 2024. doi:10.1145/3632894.

37 Calvin Loncaric, Michael D Ernst, and Emina Torlak. Generalized data structure synthesis. In *Proceedings of the 40th International Conference on Software Engineering*, pages 958–968, 2018. doi:10.1145/3180155.3180211.

38 Calvin Loncaric, Emina Torlak, and Michael D. Ernst. Fast synthesis of fast collections. In *PLDI 2016: Proceedings of the ACM SIGPLAN 2016 Conference on Programming Language Design and Implementation*, pages 355–368, Santa Barbara, CA, USA, June 2016. doi:10.1145/2908080.2908122.

39 David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the API jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 48–61, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1065010.1065018.

40 Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. Bottom-up synthesis of recursive functional programs using angelic execution. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–29, 2022. doi:10.1145/3498682.

41 Ashish Mishra and Suresh Jagannathan. Specification-guided component-based synthesis from effectful libraries. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):616–645, 2022. doi:10.1145/3563310.

42 Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 619–630, 2015. doi:10.1145/2737924.2738007.

43 Antti Oulasvirta and Pertti Saariluoma. Surviving task interruptions: Investigating the implications of long-term working memory theory. *International Journal of Human-Computer Studies*, 64(10):941–961, 2006. doi:10.1016/J.IJHCS.2006.04.006.

44 Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic: 15th International Workshop, CSL 2001 10th Annual Conference of the EACSL Paris, France, September 10–13, 2001, Proceedings 15*, pages 1–19. Springer, 2001. doi:10.1007/3-540-44802-0_1.

45 Hila Peleg, Roi Gabai, Shachar Itzhaky, and Eran Yahav. Programming with a read-eval-synth loop. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), November 2020. doi:10.1145/3428227.

46 Hila Peleg and Nadia Polikarpova. Perfect is the enemy of good: Best-effort program synthesis. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

47 Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–310, 2016.

48 Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016. doi:10.1145/2908080.2908093.

49 Nadia Polikarpova and Ilya Sergey. Structuring the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290385.

50 Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. cvc4sy: smart and fast term enumeration for syntax-guided synthesis. In *International Conference on Computer Aided Verification*, pages 74–83. Springer, 2019. doi:10.1007/978-3-030-25543-5_5.

51 John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002. doi:10.1109/LICS.2002.1029817.

52 Kensen Shi, Jacob Steinhardt, and Percy Liang. FrAngel: component-based synthesis with control structures. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019. doi:10.1145/3290386.

53 Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. Syntax-guided synthesis of datalog programs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 515–527, 2018. doi:10.1145/3236024.3236034.

54 Calvin Smith and Aws Albarghouthi. MapReduce program synthesis. *Acm Sigplan Notices*, 51(6):326–340, 2016. doi:10.1145/2908080.2908102.

55 Sunbeom So and Hakjoo Oh. Synthesizing imperative programs from examples guided by static analysis. In *International Static Analysis Symposium*, pages 364–381. Springer, 2017. doi:10.1007/978-3-319-66706-5_18.

56 Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 404–415, 2006.

57 Jamie Stark and Andrew Ireland. Towards automatic imperative program synthesis through proof planning. In *14th IEEE International Conference on Automated Software Engineering*, pages 44–51. IEEE, 1999. doi:10.1109/ASE.1999.802091.

58 Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina S Păsăreanu. SyRust: automatic testing of Rust libraries with semantic-aware program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 899–913, 2021.

59 Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. TRANSIT: Specifying protocols with concolic snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 287–296, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2491956.2462174.

60 Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 452–466. ACM, 2017.

61 Zijiang Yang, Jinru Hua, Kaiyuan Wang, and Sarfraz Khurshid. EdSynth: Synthesizing api sequences with conditionals and loops. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 161–171. IEEE, 2018. doi:10.1109/ICST.2018.00025.

62 Yongho Yoon, Woosuk Lee, and Kwangkeun Yi. Inductive program synthesis via iterative forward-backward abstract interpretation. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1657–1681, 2023. doi:10.1145/3591288.

63 Yongwei Yuan, Arjun Radhakrishna, and Roopsha Samanta. Trace-guided inductive synthesis of recursive functional programs. *Proceedings of the ACM on Programming Languages*, 7(PLDI):860–883, 2023. doi:10.1145/3591255.

Contract Usage and Evolution in Android Mobile Applications

David R. Ferreira
Faculty of Engineering, University of Porto, Portugal

Alexandra Mendes
INESC TEC, Faculty of Engineering, University of Porto, Portugal

João F. Ferreira
INESC-ID & IST, University of Lisbon, Portugal

Carolina Carreira
Carnegie Mellon University, Pittsburgh, PA, USA
INESC-ID & IST, University of Lisbon, Portugal

Abstract

Contracts and assertions are effective methods to enhance software quality by enforcing preconditions, postconditions, and invariants. Previous research has demonstrated the value of contracts in traditional software development. However, the adoption and impact of contracts in the context of mobile app development, particularly of Android apps, remain unexplored.

To address this, we present the first large-scale empirical study on the use of contracts in Android apps, written in Java or Kotlin. We consider contract elements divided into five categories: conditional runtime exceptions, APIs, annotations, assertions, and other. We analyzed 2,390 Android apps from the F-Droid repository and processed 52,977 KLOC to determine 1) how and to what extent contracts are used, 2) which language features are used to denote contracts, 3) how contract usage evolves from the first to the last version, and 4) whether contracts are used safely in the context of program evolution and inheritance. Our findings include: 1) although most apps do not specify contracts, annotation-based approaches are the most popular; 2) apps that use contracts continue to use them in later versions, but the number of methods increases at a higher rate than the number of contracts; and 3) there are potentially unsafe specification changes when apps evolve and in subtyping relationships, which indicates a lack of specification stability. Finally, we present a qualitative study that gathers challenges faced by practitioners when using contracts and that validates our recommendations.

2012 ACM Subject Classification Software and its engineering → System description languages; Software and its engineering → Software development techniques; Software and its engineering → Software verification and validation

Keywords and phrases Contracts, Design by Contract, DbC, Android, Java, Kotlin

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.11

Supplementary Material *Software*: `https://github.com/sr-lab/contracts-android`

Funding This work was supported by Fundação para a Ciência e a Tecnologia (FCT): João F. Ferreira by projects UIDB/50021/2020 (DOI: 10.54499/UIDB/50021/2020) and the "InfraGov" project, with ref. n. 2024.07411.IACDC (DOI: 10.54499/2024.07411.IACDC), funded by the "Plano de Recuperação e Resiliência (PRR)" under the investment "RE-C05-i08 - Ciência Mais Digital", measure "RE-C05-i08.M04" (in accordance with the FCT Notice No. 04/C05 i08/2024), framed within the financing agreement signed between the "Estrutura de Missão Recuperar Portugal (EMRP)" and the FCT as an intermediary beneficiary; Carolina Carreira by the project VeriFixer, with reference 2023.15557.PEX (DOI: 10.54499/2023.15557.PEX). Alexandra Mendes was financed by National Funds through the Portuguese funding agency, FCT, within project LA/P/0063/202043 (DOI: 10.54499/LA/P/0063/2020).

© David R. Ferreira, Alexandra Mendes, João F. Ferreira, and Carolina Carreira;
licensed under Creative Commons License CC-BY 4.0

Acknowledgements We thank the anonymous reviewers for their valuable feedback, which helped improve the quality of this paper. We also thank André Salgado, who contributed to a preliminary version of the study and the tool.

1 Introduction

Building reliable mobile applications is a growing concern, as they are increasingly used in critical domains such as health, finance, and government. There are now more mobile phones than people in the world[1] with more than 2 million apps available in the App Store and Google Play [37]. Additionally, data from 2024 shows that Android is the most used platform (47%), followed by Windows (26%), and then iOS (18%) [35]. Therefore, faults in mobile apps, and particularly in Android apps, can impact a very large number of users. In addition, with an increasing number of apps in critical areas such as health and finance, faults can have a huge negative impact. It is thus important to use software reliability techniques when developing mobile applications.

One of these techniques is Design by Contract (DbC) [26], under which software systems are seen as components that interact amongst themselves based on precisely defined specifications of client-supplier obligations (*contracts*). Suppliers expect that certain conditions are met by the client before using the component (*preconditions*), maintain certain properties from entry to exit (*invariants)*, and guarantee that certain properties are met upon exit (*postconditions*). These contracts are written as *assertions* in the code. Currently, there are assertion capabilities in most programming languages, but assertions are not universally used.

Current efforts in academia and industry show that DbC [27] is an active topic of interest to the software industry, with companies such as Amazon Web Services and Consensys investing largely in the development of tools such as Dafny [25]. Additionally, the creation of tools like Verus [23] for correctness verification in Rust, further underline its importance. Such tools use DbC in the specifications used for formal verification.

DbC can help identify failures [4], improve code understanding [16], and improve testing efforts [36]. This has led to a number of empirical studies on the use of contracts in a variety of contexts [10, 33, 15, 9, 22, 21, 12, 13]. However, *there are no previous studies on the presence and usage of contracts in Android applications nor any study that includes the Kotlin language*.

We present the first large-scale empirical study of contract usage in Android mobile apps written in Java or Kotlin. Our goal is to understand 1) how and to what extent contracts are used, 2) which language features are used to denote contracts, 3) how contract usage evolves from the first to the last version, and 4) whether contracts are used safely in the context of program evolution and inheritance. Information on how practitioners use contracts can help create and improve tools and libraries by researchers and tool builders [33]. Also, empirical evidence about the benefits of contracts can encourage their adoption by practitioners and the establishment of DbC as a software design standard [36].

In summary, the contributions of this paper are:

- The first large-scale empirical study about contract usage and evolution in Android apps, resulting in a list of findings and recommendations for practitioners, researchers, and tool builders. No previous studies consider Kotlin.

[1] `https://www.weforum.org/agenda/2023/04/charted-there-are-more-phones-than-people-in-the-world/` (last accessed on 01 April 2025)

- A list of language features, tools, and libraries to represent contracts in Android applications.
- A dataset of 1,767 Java and 623 Kotlin Android apps, together with scripts that can be used to build large-scale datasets of Android apps.
- An updated and extended version of Dietrich et al.'s tool [13], which can now analyze Kotlin code and can be used to investigate additional Android-specific contracts.
- A user study that validates our recommendations and contributes with further suggestions from practitioners for increasing contract usage.

Even though we update and extend Dietrich et al.'s tool [13], our work *is not* a replication of their study. Our study differs from theirs by focusing on Android apps and not on Java apps only. Due to the focus on Android, our study considers Kotlin in addition to Java, as since 2019, Kotlin is the preferred language for Android app developers[2]. Further, Kotlin is now used by over 60% of Android professional developers[3].

As mentioned above, similar studies to ours have been conducted for different ecosystems, because investigating how developers use contracts can inform future developments that make DbC more effective in practice, thus increasing software reliability.

Data & Artifact Availability. To support our study, an artifact was developed to automatically collect contracts from Android applications and to produce the necessary empirical data. The artifact is written in Python and Java, and includes an extension of the tool proposed by Dietrich et al. [13]. All the code and datasets are publicly available: `https://github.com/sr-lab/contracts-android`

2 Contracts in Android Applications

Our notion of contract follows from the theory of *design by contract* [26], where preconditions, postconditions, and invariants are used to document (and specify) state changes that might occur in a program. Pre and postconditions are associated with individual methods and constrain their input and output values. On the other hand, invariants are associated with classes and properties and constrain all the public methods in a given class. Preconditions represent the expectations of the contract, and postconditions represent its guarantees. Invariants represent the conditions that the contract maintains.

Contrary to the Eiffel language, conceived by Bertrand Meyer in 1985, neither Java nor Kotlin provide a native and standardized approach for contract specification [10]. Still, developers can take advantage of language features and libraries to specify preconditions, postconditions, and class invariants in both languages. For example, they can use constructs provided by the programming language, such as the Java `assert` keyword introduced in Java 1.4; they can use conditional runtime exceptions such as Java `IllegalArgumentException`; they can use annotations such as the AndroidX annotations `@NonNull` and `@Nullable`; and they can use specialized libraries such as Google Guava's Preconditions API.[4]

To facilitate the comparison with previous studies, we group these constructs into the five categories proposed by Dietrich et al. [13]: conditional runtime exceptions (CREs), APIs, annotations, assertions, and other. The main difference is that, since we focus on Android

[2] https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred (last accessed on 01 April 2025)
[3] `https://developer.android.com/kotlin` (last accessed on 01 April 2025)
[4] `https://guava.dev/releases/snapshot-jre/api/docs/com/google/common/base/Preconditions.html` (last accessed on 01 April 2025)

Table 1 Contract elements considered in this study.

category	examples

CREs (74 constructs)	`IllegalArgumentException`
	`EmptyStackException`
	`SecurityException`
	`UnsupportedOperationException`
	`AccessControlException`
	`IndexOutOfBoundsException`
	`NullPointerException`

APIs (31 constructs)	`org.apache.commons.lang.Validate.*`
	`org.apache.commons.lang3.Validate.*`
	`com.google.common.base.Preconditions.*`
	`org.springframework.util.Assert.*`

Assertions (6 constructs)	`assert` (Java) `assert` (Kotlin) `check(), checkNotNull()` (Kotlin) `require(), requireNotNull()` (Kotlin)

Annotations (136 constructs)	`org.jetbrains.annotations.*`
	`org.intellij.lang.annotations.*`
	`edu.umd.cs.findbugs.annotations.*`
	`android.annotation.*`
	`androidx.annotation.*`
	`javax.annotation.*` (JSR305)

Other (1 construct)	`@ExperimentalContracts` (Kotlin)

applications, we include contract elements that are specifically used by Android developers (e.g., Android annotations and specific Android runtime exceptions). To search for relevant contract elements, we used two main additional sources: the Android API Reference[5] and the Kotlin Standard Lib API[6]. Table 1 summarizes the classification and provides some examples; we consider a total of 248 constructs. Below, we briefly describe each category. More details are included in the Supplementary Material [17].

2.1 CREs

An exception can be used to signal, at runtime, a contract violation. Bloch [6] suggests the use of runtime exceptions to indicate programming errors, as the great majority indicates precondition violations. However, it is important to note that the exception itself does not represent a contract; it needs to be associated with a previous check (e.g., an exception thrown inside an *if-else block*) to be considered so. Java and Kotlin offer many exceptions that can be used for this purpose, such as the *IllegalArgumentException*. The *android.util* package offers additional exceptions that we are also interested in analyzing, such as the case of the *AndroidRuntimeException*. Additionally, we are interested in a particular exception, the *UnsupportedOperationException*, which, according to the Java documentation, is thrown to indicate that the requested operation is not supported. As Dietrich et al. argue, this is the strongest possible precondition and can not be satisfied by any client [13].

The following code shows an example of a precondition. An *IllegalArgumentException* is thrown when the contract *shoppingCart.isEmpty()* is violated. The method *proceed-*

[5] `https://developer.android.com/reference` (last accessed on 01 April 2025)
[6] `https://kotlinlang.org/api/core/kotlin-stdlib` (last accessed on 01 April 2025)

WithCheckout can only perform its task when the *shoppingCart* has at least one item.

```
1      public void proceedWithCheckout(List<Item> shoppingCart)  {
2          if (shoppingCart.isEmpty()) {
3            throw new IllegalArgumentException();
4          }
5          ...
6      }
```

We consider a total of 74 CREs (while Dietrich et al. [13] consider six). We show some examples in Table 1 but, due to lack of space, the full list is in the Supplementary Material [17].

2.2 APIs

APIs consist of wrappers around conditional exceptions and other basic constructs. This contributes to a simpler and explicit representation of contracts. We are interested in the four APIs listed in Table 1. For example, the *Apache Commons* offers the *Validate*[7] class that, according to the official documentation, "assists in validating arguments", suggesting a precondition usage. The methods provided by the *Validate* class are simply wrapping exceptions that we have already considered in the CREs. The same libraries do not offer any equal approach to specify postconditions, which suggests a preference from tool builders towards preconditions. Nevertheless, and against the guidelines, practitioners can still use any of those API's methods to check postconditions.

In the following example, that makes use of an API, a precondition *items list is not empty* is declared. In other words, the method *addToShoppingCart* guarantees that if the client fulfills its obligation to provide a non-empty list of items, it will be able to perform its job correctly.

```
1      import org.apache.commons.lang3.Validate
2
3      fun addToShoppingCart(items: List<Item>): List<Item>  {
4          Validate.notEmpty(items)
5          shoppingCart.addAll(items)
6          return shoppingCart
7      }
```

2.3 Assertions

Assertions were introduced in Java 1.4 and are specified through the *assert* reserved keyword. It helps practitioners verify conditions that must be true during runtime. JVM throws an *AssertionError* if the condition is false. However, JVM disables assertion validation by default, requiring it to be explicitly enabled. This means that practitioners may assume that contracts specified through assertions will be validated at runtime when in fact the assertions are disabled. This leads to an incorrect, and potentially dangerous, assumption. Having that in mind, assertions can still easily be used to check preconditions and postconditions.

In the following example, the contract associated with the *addToShoppingCart* method defines two preconditions – the list of items to add to the shopping cart must have a size of

[7] `https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/Validate.html` (last accessed on 01 April 2025)

greater than zero and *smaller or equal to ten* – and a postcondition – the items added to the shopping cart *will be present in the shopping cart list*.

```
1      public List<Item> addToShoppingCart(List<Item> items){
2        assert !items.isEmpty();
3        assert items.size() <= 10;
4        shoppingCartItems.addAll(items);
5        assert shoppingCartItems.containsAll(items);
6        return shoppingCartItems;
7      }
```

Kotlin also has its own *assert*. However, contrary to the Java version, *assert* in Kotlin is a function and not a reserved word. This means that any class can define a method with the name *assert*, which makes it harder for an automated analysis tool to distinguish between Kotlin's assert or a developer's custom method. Additionally, contrary to Java, Kotlin always executes the assert expression and only uses the *-ea* JVM flag to decide whether to throw the exception. Kotlin also offers other methods: *check()*, *checkNotNull()*, *require()*, and *requireNotNull()*. Although these throw an *IllegalArgumentException* or an *IllegalStateException* instead of an *AssertionError*, we added them to the assertions category because of their syntactic similarities.

The following code uses Kotlin's methods to specify the same pre and postconditions as in the previous Java example.

```
1      fun addToShoppingCart(items: List<Item>): List<Item> {
2          assert(items.isNotEmpty())
3          require(items.size <= 10)
4          shoppingCartItems.addAll(items)
5          check(shoppingCartItems.containsAll(items))
6          return shoppingCartItems
7      }
```

2.4 Annotations

Annotations are metadata added to the program providing information that can be used at compile time or runtime to perform further actions. Java provides many annotations through the *java.lang* package. Table 1 lists the annotation packages we are particularly interested in studying. No previous studies consider the *android.annotation* and the *androidx.annotation*.

The annotation-based approach is particularly interesting for two reasons. First, many annotations can be associated with the method's arguments (preconditions), the method's return values (postconditions), or the class properties (invariants). Second, since annotations are usually added to the method's signature or to the class property, there is a greater separation between the contract specification and the service's implementation. This means that annotations, like in the Eiffel's approach, do not increase the complexity of the method's implementation, contrary to what happens with CREs, APIs, and assertion-based approaches.

The code shown below uses annotations from the *javax.validation.constraints.** packages to specify contracts. The method states that it can only *return a list with a minimum size of 1* (postcondition), if the *item identifier is not null* and the *quantity is greater or equal to one* (preconditions). Also, the class property *items* is associated with a class invariant that states that the *shopping cart can only contain ten items at maximum*. This example shows that adding contracts through annotations does not require adding extra checks to

the implementation, contributing to cleaner code.

```
1      import javax.validation.constraints.*
2      class ShoppingCart {
3          @Size(max=10)
4          private val items: List<Item> = mutableListOf()
5
6          @Size(min=1) fun addItem(@NotNull itemUUID: String, @Min(1)
               quantity: Int): List<Item> {
7              ...
8          }
9      }
```

2.5 Other

We consider Kotlin Contracts[8], an experimental feature introduced in Kotlin 1.3 that allows developers to state a method's behavior to the compiler explicitly. As the following example shows, they also provide useful information to the compiler: the call to *split* in line 4 causes no error, because the contract specified in line 10 guarantees that *birthdate* is not null.

```
1  @ExperimentalContracts
2  fun sendBirthdayMessage(birthdate: String?) {
3   birthdateIsValid(birthdate)
4   val birthdaySplit = birthdate.split("/")
5    ...
6  }
7
8  @ExperimentalContracts
9  fun birthdateIsValid(birthdate: String?) {
10  contract{returns() implies (birthdate != null)}
11  if (birthdate == null) {
12   throw IllegalArgumentException()
13  } ... }
```

3 Related Work

This section presents related work on the usage of contracts, assertions, and annotations by practitioners.

3.1 DbC and Contract Usage

It is widely supported that DbC contributes to improving software reliability [28, 39, 19]. The advantages commonly mentioned are that DbC (i) improves code understanding [16, 29, 39, 34], (ii) helps identify bugs earlier and diagnose failures [39, 4, 9, 13, 33], and (iii) contributes to better tests [39, 4, 33, 3, 36]. Some studies demonstrated that DbC requires fewer project person-to-hour resources [8, 36], but could not confirm an impact on quality. Moreover, DbC contributes to less time spent on writing tests [36]. Blom et al. [7] suggest that DbC results in fewer errors and decreases development time. In another study, Zhou et al. [42] show that DbC increased reliability in software components. In a study on C# projects using Code Contracts, Schiller et al. [33] found a high percentage of contracts related to

[8] `https://github.com/Kotlin/KEEP/blob/master/proposals/kotlin-contracts.md` (last accessed on 01 April 2025)

null checking and suggest the importance of creating design patterns alongside tools and libraries. Estler et al. [15] analyzed 21 Eiffel, C#, and Java projects known to be equipped with contracts. Most contracts are null checks, with preconditions being typically larger than postconditions. The authors concluded that the average number of clauses per specification is stable over time and that the method's implementation changes more frequently than its specification. However, they warned that strengthening contracts may be more frequent than weakening, indicating some unsafe evolution of contracts. Lastly, Dietrich et al. [13] investigated 176 popular Java projects in the Maven repository and found that the majority of programs do not use contracts significantly. They found that CREs are the most commonly used category, followed by asserts. The dominance of preconditions over postconditions in contracts is consistent with other studies [10, 33]. They found that projects that use contracts maintain or even expand their usage over time. Similarly to Estler et al. [15], the authors reported some unsafe evolution of contracts, which can happen when a method strengthens its preconditions or weakens its postconditions. They also found many violations of the Liskov Substitution Principle (LSP), with prevalence in the annotations. The LSP states that objects of a superclass should be replaceable with objects of a subclass without altering the correctness of the program. According to this principle, a sub-type can only weaken preconditions or strengthen postconditions and class-invariants from its parent [1]. A sub-type should behave in a way that does not violate the expectations set by its super-type. This ensures that any code that works with the super-type can work with the sub-type without requiring modifications or encountering unexpected behavior. The authors caution that their dataset mainly includes libraries, which may explain the low usage of annotations. This study is the one most related to the work presented here, as it also studies contracts in Java. However, our study differs from Dietrich et al.'s [13] in that, not only we consider more constructs, we also focus on Android apps and we study both Java and Kotlin.

3.2 Assertion Usage

Kudrjavets et al. [22] studied two Microsoft Corporation components, written mainly in C and C++, and found that increased assert density led to a decrease in fault density, and that using asserts was more effective for fault detection than some static analysis tools. Kochhar and Lo [21] studied a dataset of 185 Apache Java projects available on GitHub and found that adding asserts contributes to fewer defects, especially when many developers are involved. This agrees with reports from Kudrjavets et al. [22] but it is not supported by Counsell et al. [12], who analyzed two industrial Java systems and found no evidence that asserts were related to the number of defects. Kochhar and Lo [21] also concluded that developers with more ownership and experience use asserts more often, which shows that more advanced programmers see it as a valuable practice. In line with other previously mentioned studies for contracts [33, 15], most uses are related to null-checking.

3.3 Annotation Usage

There is a general understanding that the use of annotations among practitioners is growing [40, 18]. Yu et al. [40] conducted a study on 1,094 GitHub open-source projects and found a median value of 1.707 annotations per project, with some developers overusing them. The authors argue the need for better training and tools to help derive better annotations. Other authors made a similar claim for contracts [33]. Additionally, developers with higher ownership use annotations more often, which agrees with the findings by Kochhar and Lo [21] related to assertion usage. Grazia and Pradel [18] investigated the evolution of type

annotations, some of which can act as contracts, in 9,655 Python projects. The authors reported that although type annotations usage is increasing, less than 10% of potential elements are being annotated. This contradicts the (general) annotations overuse reported by Yu et al. [40]. More importantly, the study found that once added, 90.1% of type annotations are never updated. This indicates that specifications are more stable than implementations, which is desirable. A similar finding was reported by Estler et al. [15] related to the stability of contracts while the program evolves. Also relevant is that most type annotations were associated with parameter and return types, rather than with variable types. Finally, the authors found that adding type annotations increased the number of detected type errors. This motivates the general use of these features to improve software reliability.

4 Study Design

In this section, we present the design of our study, including the research questions, how the dataset of Android apps is created, the classification used for contracts, and the methodologies used to study contract usage and evolution.

4.1 Research Questions

In this study, we aim to answer the following research questions:

- **RQ1**. [**Contract Usage**] How and to what extent are contracts used in Android applications?
- **RQ2**. [**First-To-Last Version Evolution**] How does contract usage evolve in an application from the first to the last version?
- **RQ3**. [**Safety**] Are contracts used safely in the context of program evolution and inheritance?

4.2 Dataset

The dataset used is composed of real-world apps obtained from F-droid,[9] an alternative app store listing over 4,000 free and open-source projects. The fact that it has a large number of open-source apps on a wide range of domains, makes F-Droid a good option. Moreover, F-Droid is normally used in research studies on Android apps [11, 41]. Apart from native Android apps written in Java or Kotlin, F-Droid's catalog also contains projects that use hybrid frameworks (e.g., React Native) that we exclude from our dataset.

We started by downloading the *F-Droid index*, which is a list of URLs for each project available in the catalog. Next, this list is *filtered* based on the following criteria: **1)** The application source code is hosted in GitHub; **2)** The application source code is either Java or Kotlin; **3)** The GitHub project is not archived; **4)** The GitHub project has had a commit since 2018. These inclusion criteria ensure that the project's source code is easily accessible (through GitHub), is written mainly in Java or Kotlin (the languages we are interested in studying), while also guaranteeing that the project is active and relevant. We retrieve *two versions* for each of the filtered projects, which is a required step for the *First-to-Last Version* evolution study. We do this by storing a list of the URLs pointing to two GitHub versions: we first try to retrieve the oldest and the most recent *release*; if there are not enough releases, we try to retrieve the oldest and the most recent *tag*; finally, if there are not enough tags, we

[9] `https://f-droid.org` (last accessed on 01 April 2025)

Table 2 Dataset metrics.

metric	Java	Kotlin	Both
projects	1,767	623	2,390
compilation units	208,479	129,490	337,969
classes	305,749	265,410	571,159
methods (all)	2,113,620	632,416	2,746,036
constructors (all)	208,949	100,534	309,483
methods (public, protected, internal)	1,801,171	506,647	2,307,818
constructors (public, protected, internal)	187,789	99,221	287,010
KLOC including comments	40,635	12,341	52,977

just keep the most recent commit of the repository. If there are no releases nor tags, we only consider one version (excluding it from the *First-to-Last Version* evolution study). Although our script resolved most of the versioning schemes found, some projects required manual handling to determine which version was the first and the last. Throughout the paper we refer to the most recent version as *last* or *second* version. Finally, we clone all the projects contained in the versions list. *Every file that is neither a Java nor a Kotlin file is removed* from the dataset, which helps to decrease its size.

4.2.1 Dataset metrics

From the initial list of 4,070 projects in the F-Droid index retrieved on May 21, 2023, we got 3,215 hosted in GitHub, 3,141 non-duplicated URLs, and 2,390 projects after filtering by the inclusion criteria. Out of these, 1,767 are Java applications and 623 are Kotlin applications. For 1,802 applications we were able to retrieve two versions to be used in the *first-to-last version evolution* study. This means that for 588 applications it was only possible to retrieve one version (these are applications for which there are no GitHub releases nor tags). While these applications are still evaluated in the context of the *usage* and *LSP* studies, they are not considered for the *first-to-last version evolution* study.

Table 2 presents additional metrics about the dataset size. As the table shows, the dataset is imbalanced, with more Java apps. The dataset includes 208,479 Java and 129,490 Kotlin compilation units and, therefore, Java represents 61.7% of the overall number of compilation units. This imbalance requires caution when trying to read this work's results from the perspective of comparing Java against Kotlin's use of contracts. Furthermore, the dataset includes 571,159 classes, 2,746,036 methods, and 309,483 constructors. We did not consider *private* methods, because those are not used directly by a client, and a contract is a bond between a supplier and a client. In total, we analyzed 2,594,828 *public*, *protected*, and *internal* methods and constructors.

In terms of diversity, the dataset includes apps from various domains, such as gaming, communication, multimedia, security, health, and productivity.

4.3 Data Collection and Analysis

Here, we describe the analysis tool and the studies conducted to answer our research questions: the usage study, the *first-to-last version* evolution study, and the Liskov Substitution Principle study.

4.3.1 Analysis Tool

Our analysis tool is an extension of the tool created by Dietrich et al. [13], which was used in their study on the usage of contracts in Java apps. We extended the tool to support Kotlin and more constructs focused on Android apps. Additionally, the framework suffered considerable refactoring and organization to ease its comprehension and maintainability. The main effort was to add support for Kotlin. The original tool used the JavaParser[10] library to perform AST analysis of Java code. Since this library is not able to parse Kotlin source code, we integrated JetBrains's Kotlin compiler[11] to perform this task. This required us to implement new versions of the tool's extractors and visitors classes using the methods provided by the new library to be able to identify contract patterns in Kotlin. We also updated the JavaParser library to support newer Java versions.

The tool is divided into three parts: 1) *usage*, which extracts the list of contracts present in each program and produces statistics about their use; 2) *inheritance*, which identifies contracts in overridden methods and validates whether they violate the Liskov Substitution Principle; and 3) *first-to-last version evolution*, which analyses how identified contracts evolve in later versions of the application. The following sections describe how each component contributes to answering our research questions.

4.3.2 Usage Study

The usage study is divided in two main steps: 1) identifying contract occurrences and 2) producing statistics about those results. Our tool uses the JavaParser and JetBrains's Kotlin compiler libraries to perform AST analysis. This analysis is done against a set of extractors to identify occurrences of our defined constructs. Each category requires different approaches for their identification:

- *CREs*. During the AST analysis, we look for the pattern:

  ```
  if (<condition>) { throw new <exception> (<args>) }
  ```

  When this pattern is found, we check whether the exception belongs to the list of CREs considered (see Section 2). In line with Java's good practices, we assume that CREs are used with preconditions.
- *APIs*. Firstly, we check whether the file contains an import declaration to any API package considered. If any is found, all call expressions in that file are analyzed to determine if they are invoking any of the methods provided by the API. As stated before, we assume the analyzed APIs to be associated with preconditions.
- *Assertions*. Identifying Java asserts is straightforward since the JavaParser provides a visitor method for this particular statement. The complexity lies in identifying Kotlin asserts, which is not a reserved keyword. To handle this challenge, when analyzing a file, we first search for any method declaration and any import statement that has a name equal to one of the following expressions: *assert*, *require*, *requireNotNull*, *check*, and *checkNotNull*. Next, we identify whether the class invokes any method with one of those names. Suppose a class contains a method declaration or import statement, as well as an invocation using the name of one of these expressions. In that case, we consider it an ambiguous situation, and therefore, we do not consider it an assert instance. If the class

[10] `https://javaparser.org` (last accessed on 01 April 2025)
[11] `https://github.com/JetBrains/kotlin` (last accessed on 01 April 2025)

invokes one of those methods but does not declare/import any method with that same name, we consider it an assert. We do not classify assertions either as preconditions or postconditions.

- *Annotations*. We check if the source code file contains an import statement to one of the packages listed in Table 1. If that is the case, we check every annotation in that file to see if it matches any of those provided by the imported package. We also identify the artifact to which the annotation is associated as follows: 1) annotations associated with a method's parameters are preconditions; 2) annotations associated with a method are postconditions; and 3) annotations associated with a field are class invariants.
- *Others*. This category only includes the investigation of the experimental *Kotlin Contracts*. To identify occurrences of this construct, we look for the pattern `contract {returns (<condition>) implies (<condition>)}`.

Our tool creates a *JSON* file for program version that stores the identified contracts, including 1) the file path, 2) the associated condition, 3) the method or property name, 4) the type of artifact (method or property), 5) the line number, and 6) the contract type. In the second step of the *usage* study, all the *JSON* files are analyzed to produce statistics about the identified contracts, including the frequency of each category (API, annotation, assertion, etc.), class (preconditions, postconditions, and class invariants), and construct (java assert, Guava API, *androidx* annotations, etc.). For each category, we also compute the Gini coefficient and the list of programs with more contracts.

4.3.3 First-to-Last Version Evolution

We focus on the initial and final GitHub versions of each project as these represent critical moments in the development: the initial introduction of the DbC constructs and the culmination of the development process. This allows us to check if there were any significant changes in the use of contracts.

After identifying a contract in the first version of the app, we check whether, in the later version, the contract still exists, was modified, or removed. We also report cases when a contract is added to an artifact (method or parameter) in the later version of the app (but was not present in the first version). These provide insights into how contracts evolve in an app and whether this evolution poses risks to the client.

As already mentioned, a contract establishes rights and obligations between clients and suppliers. Therefore, when a contract is altered, both parts should be informed and updated accordingly. This is particularly crucial when a *precondition is strengthened* or when a *postcondition is weakened*. In the first case, if the precondition is strengthened and the client does not know it, it can fail to cover its new obligations, and, therefore, the supplier is not bound to keep its part of the contract. In the latter case, if the postcondition is weakened, the client may still be making assumptions that the supplier does not ensure anymore. An example is shown in Listing 1, where the annotation *@NonNull* was added to the *toolbar* parameter in the last version. This is the case of a *precondition strengthening*: in the first version, the method accepted a null *toolbar*, but now it requires it to be not null. Therefore, if the client is not updated, it will fail to cover its new obligation.

Similarly to Dietrich et al. [13], we create *diff records* from the contracts present in the two versions of a program's method and then classify them according to the *evolution patterns* listed in Table 3.

4.3.4 Liskov Substitution Principle Study

When a method is overridden in a subclass, that class can specify new contracts added to the ones inherited from the superclass method. In this case, proper handling of contracts should

Listing 1 Example of a precondition strengthened using the annotation `@NonNull`, taken from the project Retro Music Player, a music player for Android (in class ToolbarContentTintHelper).

```
    public static void setToolbarContentColorBasedOnToolbarColor(
        @NonNull Context context,
-       Toolbar toolbar,
+       @NonNull Toolbar toolbar,
        @Nullable Menu menu,
        int toolbarColor,
        final @ColorInt int menuWidgetColor
```

Table 3 Classification of the diff records produced during the evolution and LSP study.

Classification	Description	Risk
PreconditionsStrengthened	A precondition was added to a method or a clause to an existing precondition with the '&' or '&&' operators.	Potential risk
PreconditionsWeakened	A precondition was removed from a method, or a clause was added to an existing precondition with the '|' or '||' operators.	No risk.
PostconditionsStrengthened	A postcondition was added to a method or a clause to an existing postcondition with the '&' or '&&' operators.	No risk.
PostconditionsWeakened	A postcondition was removed from a method, or a clause was added to an existing postcondition with the '|' or '||' operators.	Potential risk.
MinorChange	Contract elements are the same, but in different order; or removal of a Nullable postcondition, which is not considered as a significant weakening [13].	No risk.

follow the Liskov Substitution Principle (LSP), which states that the subclass method must accept all input that is valid to the superclass method and meet all guarantees made by the superclass method. In other words, a subclass method can only *weaken preconditions* and *strengthen postconditions*.

To detect those occurrences, we list all methods in each program-version pair associated with their respective class. We also identify the class' parents. Then, similarly to the *first-to-last version* evolution study, diff records are created between the subclass and the superclass methods. These records are classified based on the evolution patterns outlined in Table 3, following the categories and descriptions proposed by Dietrich et al. [13].

5 Results

In this section, we present the results of our empirical study, as well as the main findings. As mentioned earlier, the dataset contains an imbalanced distribution of compilation units, with 61.7% written in Java and 38.3% in Kotlin. This imbalance should be considered when interpreting the findings, particularly in the context of comparing contract usage between Java and Kotlin.

5.1 RQ1: Contract Usage

Table 4 shows the number of contracts found per category, considering all versions (columns 2 and 3) and considering only the latest version of each app (columns 4 and 5). The table also identifies the number of apps containing at least one contract for that category (columns 6 and 7). The most obvious conclusion is that, in both languages, annotation-based contracts are the most popular category. More specifically, considering both languages in the last version, annotations represent 85.2% of the contracts found, followed by CRE with 11.1%, and then assertions with 2.9%. The results show similar tendencies between Java and Kotlin, and the only difference is that while Java's second most popular category is CREs, in Kotlin, it is assertions. This relatively high percentage of the assertion category in Kotlin is explained by our inclusion of the four language's standard library methods listed in Section 2, where *require()* alone counts 901 total occurrences.

Finding 1: Most contracts are annotation-based, accounting for 86.21% in Java and 79.29% in Kotlin of the total number of contracts found.

This distribution in categories' popularity significantly differs from the findings of Dietrich et al. [13], who reported that the most common category was CREs and found surprisingly low use of annotations. This may be explained by the fact that, while our dataset is formed mostly by user-focused Android applications, Dietrich et al.'s dataset was mainly Java libraries. In Table 6, we can also see that most annotations found belong to the *androidx.annotation.** package that the authors did not consider since it is Android-specific. Nevertheless, the high number of annotation-based contracts found is in line with literature that supports its increasing popularity [40, 18].

From Table 4, we also verify that the usage of *APIs* is low in both languages, and it is even more residual in Kotlin applications, where only nine instances were found in the latest versions. Skepticism around adding third-party dependencies to projects, which may lead to maintainability and support issues in the future, may explain this finding [5, 38].

Table 4 Number of contracts found in the dataset by category.

	contracts (all ver.)		contracts (2nd ver.)		applications	
Category	Java	Kotlin	Java	Kotlin	Java	Kotlin
API	1,813	10	1,125	9	24	4
annotation	194,448	26,849	115,861	17,490	1,227	547
assertion	3,525	3,868	2,217	2,370	325	234
CRE	26,076	3,374	15,195	2,187	787	288
other	–	1	–	1	–	1

Table 5 Gini coefficient by category.

Category	**Java**	**Kotlin**
assertion	0.70	0.71
API	0.80	0.37
annotation	0.87	0.76
CRE	0.77	0.67
others	–	1.00

Finding 2: The use of APIs to specify contracts is rare.

Table 6 shows the frequency of each construct. We highlight that the high number of annotations found is leveraged mostly by the *androidx.annotation.** package. In APIs, the *Guava* library constitutes most of the usage. We were not expecting to see any usage of *Spring Framework Asserts* since this library was designed to be used in the *Spring* framework, but we still found one occurrence. At the same time, we found no occurrences of the now deprecated *FindBugs* annotations. Additionally, we identified a single occurrence of *Kotlin Contracts*, which may depict the practitioner's distrust of using a feature still in an experimental phase.

We now consider Table 5, which presents each category's computed *Gini coefficient*. The *Gini coefficient* measures the inequality among the values of a frequency distribution. In other words, a *Gini coefficient* of 0 indicates perfect equality, where all apps have the same number of contracts. In contrast, a *Gini coefficient* of 1 means that a single program has all the contracts. We observe that all coefficients in the table are high, except for Kotlin's API usage. This means that although some apps use contracts intensively, the majority does not use them significantly. This aligns with the results found by Dietrich et al. [13]. This conclusion can also be seen in Table 7, where the five projects that use more contracts per category are listed. The table shows the number of contract elements used and the application's category. We find that a small group of projects own a large percentage of the overall use in each category. It is clearly visible from the *assertion* and *CRE* categories that the numbers quickly decrease through the first to the fifth application showing the unbalanced usage between applications. F-Droid does not provide statistics, such as downloads, but the categories shown provide an indication of their purpose (with over half of these applications belonging to the category *Internet*).

Finding 3: Although there are some applications that use contracts intensively, the majority do not use them significantly.

Lastly, Table 8 presents the frequency of each contract type. Once again, we have distinct results for Java and Kotlin. In Java, we found 64.80% of the *classified* instances in the

Table 6 Number of contracts found in the dataset by construct and category.

		contracts (all ver.)		contracts (2nd ver.)	
Construct	Category	Java	Kotlin	Java	Kotlin
cond. runtime exc.	CRE	25,565	3,232	14,887	2,071
unsupp. op. exc.	CRE	511	142	308	116
java assert	assertion	3,525	–	2,217	–
kotlin assert	assertion	–	3,868	–	2,370
guava precond.	API	1,798	10	1,121	9
commons validate	API	14	0	3	0
spring assert	API	1	0	1	0
JSR303, JSR349	annotation	0	0	0	0
JSR305	annotation	4,195	20	2,133	13
findbugs	annotation	0	0	0	0
jetbrains	annotation	2,310	138	1,596	98
android	annotation	12,003	5,704	7,013	3,414
androidx	annotation	175,940	20,987	105,119	13,965
kotlin contracts	others	–	1	–	1

Table 7 Top five applications using contracts (second versions only) by category.

Category	**Applications**
assertion	K1rakishou-Kuroba-Experimental (378; Internet), a-pavlov-jed2k (314; Internet), abhijitvalluri-fitnotifications (143; Connectivity), thundernest-k-9 (114; Internet), mozilla-mobile-firefox-android-klar (95; Internet)
CRE	redfish64-TinyTravelTracker (1,036; Navigation), nikita36078-J2ME-Loader (690; Games), abhijitvalluri-fitnotifications (561; Connectivity), lz233-unvcode-android (561; Writing), cmeng-git-atalk-android (447; Internet)
API	wbaumann-SmartReceiptsLibrary (534; Money), alexcustos-linkasanote (318; Internet), BrandroidTools-OpenExplorer (69; System), snikket-im-snikket-android (60; Internet), oshepherd-Impeller (33; Internet)
annotation	MuntashirAkon-AppManager (5,957; System), Forkgram-TelegramAndroid (5,552; Internet), Telegram-FOSS-Team-Telegram-FOSS (5,549; Internet), MarcusWolschon-osmeditor4android (4,393; Navigation), NekoX-Dev-NekoX (4,032; Internet)
other	zhanghai-MaterialFiles (1; System)

Table 8 Number of contracts found in the dataset by type.

	contracts (all ver.)		contracts (2nd ver.)		applications	
Type	Java	Kotlin	Java	Kotlin	Java	Kotlin
precond.	145,961	9,323	85,627	5,810	1,132	355
postcond.	49,694	11,669	30,224	7,632	925	438
invariants	26,623	9,217	16,280	6,221	677	359
unclassified	3,584	3,893	2,267	2,394	279	202

Table 9 The top 10 most frequent constructs per type in the last versions of Java applications.

Preconditions	Postconditions
AndroidXNonNull (45,399)	AndroidXNonNull (12,943)
AndroidXNullable (18,236)	AndroidXNullable (6,945)
IllegalArgumentException (7,663)	AndroidSuppressLint (3,125)
IllegalStateException (3,232)	AndroidTargetApi (1,243)
NullPointerException (2,230)	AndroidXRequiresApi (760)
GuavaPreconditionNotNull (1,021)	AndroidXWorkerThread (568)
AndroidXStringRes (1,008)	AndroidXCheckResult (474)
JSR305NonNull (860)	AndroidXCallSuper (421)
IndexOutOfBoundsException (656)	AndroidXKeep (398)
JetBrainsNotNull (612)	AndroidXUiThread (347)

last versions to be preconditions, 22.87% postconditions, and only 12.32% class invariants. These results align with other studies on contracts [10, 33, 13] that show a clear preference towards preconditions. However, results for Kotlin are different: considering last versions, we found 38.81% to be postconditions, 31.64% class invariants, and 29.55% preconditions. This suggests that Kotlin developers tend to favor postconditions, while preconditions come at the last position. According to the classification described in Section 4.3.2, only annotations are classified as postconditions or class invariants. This means that in Kotlin, there is a higher number of annotations associated with methods' return values and class properties than with the methods' parameters.

Finding 4: Java and Kotlin practitioners display different tendencies when it comes to the contract type. In Java, there is a preference towards preconditions, while in Kotlin, postconditions are the most frequent type.

Although we can not provide a reason for this finding with certainty, analysing the most frequent constructs for pre and postconditions in both languages can give us some hints.

Tables 9 and 10 show the top 10 most frequent constructs per type in the last versions of Java and Kotlin apps, respectively. Comparing the two tables reveals distinct behavior patterns: for Kotlin, none of the top ten constructs relates to null-checking; however, for Java's instances reported in Table 9, 84.48% of preconditions and 73.05% of postconditions are associated with null-checking. In this number, we are not considering potential *IllegalArgumentException* and *IllegalStateException* that could be associated with null-checking since this would require analyzing the condition in the *if-statement*. This suggests a lack of expressiveness in the contracts specified by Java practitioners, with most being associated with null-checking, consistent with prior studies [33, 15].

This contrast in null-checking contracts between Java and Kotlin is easily explained by the languages' different takes on nullability. In Kotlin, regular types are non-nullable

Table 10 The top 10 most frequent constructs per type in the last versions of Kotlin applications.

Preconditions	Postconditions
AndroidXStringRes (1,162)	AndroidSuppressLint (2,289)
IllegalStateException (772)	AndroidXVisibleForTesting (1,663)
IllegalArgumentException (748)	AndroidXRequiresApi (738)
AndroidXColorInt (532)	AndroidXWorkerThread (638)
AndroidXDrawableRes (435)	AndroidXMainThread (442)
AndroidXAttrRes (255)	AndroidXCallSuper (323)
AndroidXColorRes (199)	AndroidXColorInt (244)
AndroidXIdRes (187)	AndroidTargetApi (205)
ProviderMismatchException (177)	AndroidXUiThread (196)
UnsupportedOperationException (116)	AndroidXAnyThread (184)

Table 11 Contract elements by type in both versions.

		contracts (1st vers.)		contracts (2nd vers.)	
Type	category	Java	Kotlin	Java	Kotlin
cond. runtime exc.	CRE	10,678	1,161	14,887	2,071
unsupp. op. exc.	CRE	203	26	308	116
java assert	assertion	1,308	–	2,217	–
kotlin assert	assertion	–	1,498	–	2,370
guava precond.	API	677	1	1,121	9
commons validate	API	11	0	3	0
spring assert	API	0	0	1	0
JSR303, JSR349	annotation	0	0	0	0
JSR305	annotation	2,062	7	2,133	13
findbugs	annotation	0	0	0	0
jetbrains	annotation	714	40	1,596	98
android	annotation	4,990	2,290	7,013	3,414
androidx	annotation	70,821	7,022	105,119	13,965
kotlin contracts	others	–	0	–	1

by default; therefore, in most cases, practitioners do not have the need for constructs like *AndroidXNonNull* or *JSR305NonNull*. On the other hand, it is interesting to observe that relaxing this constraint to allow nullable types is not a common practice since we found no meaningful use of constraints like *AndroidXNullable* and similar in Kotlin.

Finding 5: In Java applications, at least 80.85% of preconditions, 63.84% of postconditions, and 62.73% of class invariants are related to null-checking. In the case of Kotlin, we found only about 3.18% of preconditions, 7.17% of postconditions, and 0.66% of class invariants to be performing null-checking.

5.2 RQ2: First-to-Last Version Evolution

Table 11 presents the number of contracts in both versions by category. The *Type* column presents all types that are supported. In general, for most cases, the number of contracts in each category increased from the first to the last version. The only category where the number decreased was the *Apache's Commons Validate* for Java.

We computed some metrics to understand how the increase in the program's size relates to the number of contracts (see Table 12). These include the average and median values

Table 12 Average and median number of methods, contracts, and their ratio for the two versions.

	1st version		2nd version	
Metric	Median	Average	Median	Average
methods count	288	925.175	334	1039.360
contracts count	8	72.567	7	86.807
contract-to-method ratio	0.038	0.072	0.030	0.064

for the number of methods, the number of contracts, and the ratio between both (for both versions). The table shows that there is an average increase of about 114.185 methods per program. This is expected since the program's size tends to increase from the first to the second version. However, a more interesting insight comes from the contracts count. Although the average number of contracts per program increased, its median value decreased. This means that the dataset includes outliers with a significant rise in contract usage that considerably affected the average value. To confirm this data, we computed the ratio between the number of contracts and the number of methods for each version of a program. Then, we computed the difference between the second and the first version's ratio for each program. The average of these differences is -0.0077, and the median is -0.0012. Although the values are very small, we conclude that the number of methods increases significantly more than the number of contracts.

Finding 6: Apps that use contracts continue to use them in later versions. Moreover, the total and average numbers of contracts increase, but its median decreases by a small factor. Also, the number of methods increases at a higher rate than the number of contracts.

Similarly to our study, Dietrich et al. [13] also found that the median value of the ratio does not change much. Still, while we observed a decline between the two versions (from 0.038 to 0.030), they reported an increase (from 0.021 to 0.023). This means that although both studies show general stability related to contracts usage, contrary to their study, we were not able to find a positive correlation between the increase in the number of methods and in the number of contracts.

5.3 RQ3: Safety

To address whether practitioners tend to misuse contracts in either program evolution or inheritance contexts, we build *diff records* to be classified according to *evolution patterns*. Some of these *evolution patterns* are associated with a potential risk that may lead to client breaks, namely when *preconditions are strengthened* or *postconditions are weakened*. This process was described in more detail in Sections 4.3.3 and 4.3.4. It is important to note that the analysis tool cannot precisely capture all contract changes due to the variety of constructs we are analyzing and the complexity of their semantics. This can potentially lead to under-reporting. Another factor that may contribute to under-reporting is file path changes between versions, which may lead to no evolution patterns being detected. Even so, Table 13 still provides valuable insights into the safety of contract usage and evolution. The table shows the frequency of each *evolution pattern* in the context of *program evolution* (third column). We see that many contracts remain unchanged and that most changes are not critical. However, most of the changes that occur can lead to potential breaks, with *precondition strengthening* being over three and a half times more prevalent than *postcondition weakening*.

Listing 2 Example of a postcondition weakened using a Jetbrains annotation, taken from the project mGerrit, a Gerrit client for Android (in class SyncProcessor).

```
1 -   @NotNull
2     public Intent getIntent() { return intent; }
```

Table 13 Contract evolution in the context of program evolution and inheritance.

Contract Evolution	Critical	Evolution (#)	Inheritance (#)
unchanged	no	28,723	207
minor change	no	61	5
preconditions weakened	no	688	5
postconditions strengthened	no	1,035	76
preconditions strengthened	yes	1,963	284
postconditions weakened	yes	552	1
unclassified	?	858	159

An example of a precondition strengthening using an annotation and taken from our dataset was already shown in Listing 1. The code is from the class `ToolbarContentTintHelper` in project Retro Music Player,[12] a music player for Android. Adding `@NonNull` to the toolbar parameter strengthens the precondition by explicitly requiring callers to pass a non-null `Toolbar` instance, potentially breaking clients that previously relied on more permissive behavior. Listing 2 shows an example of a postcondition weakening. The code is taken from class `SyncProcessor` in project mGerrit,[13] a Gerrit client for Android. The postcondition is weakened because the `@NotNull` annotation promises a non-null `Intent`, but if `intent` is ever null, this contract is violated – potentially leading to runtime errors like NullPointerException in callers that rely on the non-null guarantee.

Finding 7: There are instances of unsafe contract changes while the program evolves, particularly cases of preconditions strengthening.

Finally, Table 13 also presents the results found for *evolution patterns* in the context of *inheritance* (fourth column). We observe that the *precondition strengthening* pattern makes up almost 50% of classified instances. We also note that from the classified instances, most parts are related to contract changes which means a lack of stability in specifications. Both in the *evolution* and the *inheritance* study, we found lower occurrences of *postcondition weakening* when compared to the other classifications. Also, compared to the reports from Dietrich et al.'s study [13], our results indicate a greater ratio of *precondition strengthening* per preconditions found.

Finding 8: There are instances of unsafe contract changes in an overriding context that violate the Liskov Substitution Principle, particularly cases of preconditions strengthening.

[12] `https://github.com/RetroMusicPlayer/RetroMusicPlayer` (last accessed on 01 April 2025)
[13] `https://github.com/JBirdVegas/external_jbirdvegas_mGerrit` (last accessed on 01 April 2025)

6 Discussion

In this section, we answer the research questions listed in Section 4.1, we discuss the practical implications of our findings, and we outline threats to the validity of our work.

6.1 Answers to Research Questions

Based on our findings, we answer the research questions posed in Section 4.1 as follows:

RQ1 [Contract Usage]. *How and to what extent are contracts used in Android applications?* Contracts are concentrated in a small number of apps. When applications use contracts, annotation-based approaches are the most frequent, with the *androidx.annotation* package being the most popular. The use of APIs to specify contracts is rare. While in Java, 64.80% of the classified instances are preconditions, Kotlin programs display a more equally distributed selection with 22.87% postconditions at the top. We also found that more than 60% of the classified contracts in Java are related to null-checking, while in Kotlin that number is smaller than 8%.

RQ2 [First-to-Last Version Evolution]. *How does contract usage evolve in an application from the first to the last version?* Applications that use contracts continue to use them in later versions. When comparing the number of contracts in both versions, the average number of contracts increases. This is caused by some outliers that increase its usage substantially, driving up the average. In fact, the median value decreases. Furthermore, the contract-to-method ratio decreases between versions – an average decrease of -0.0077 and a median decrease of -0.0012. Although by a residual factor, we observed that the number of contracts declines as programs grow.

RQ3 [Safety]. *Are contracts used safely in the context of program evolution and inheritance?* Contract changes are frequent and can lead to potential breaks, with *preconditions strengthening* being the most classified pattern. These results show a potentially unsafe use of contracts that may lead to client breaks and violate the Liskov Substitution Principle.

6.2 Practical Implications & Recommendations

Our findings lead to the following practical implications and recommendations.

Recommendation 1. Due to the fragmentation of technologies and approaches to specifying contracts, both Java and Kotlin standard libraries should be equipped with constructs to specify contracts and with proper official documentation.

Recommendation 2. It would be desirable to have libraries that standardize contract specifications in Java and Kotlin. Our results suggest that these libraries should be built around annotation-based contracts, given its popularity among practitioners. An annotation-based approach, where specifications are added to the program as metadata, is similar to Eiffel's approach, where the assertions do not obfuscate the method's implementation. This recommendation also applies to tool builders: given that the current use of APIs in Android development appears to be relatively low, analysis tools for Android that leverage contracts should prioritize annotations.

Recommendation 3. New tools to aid practitioners writing contracts would be valuable. For example, the integration into IDEs of contract suggestion features supported by tools for invariant inference, such as Daikon [14], could help increase practitioners' use of contracts. Another contribution could be IDE and continuous integration plugins to detect contract violations in the context of program evolution and inheritance.

Recommendation 4. Our findings show that Kotlin's default non-nullable types reduce the need to explicitly write some contracts, highlighting the significance of language design features that enable safety by default. These findings are relevant for the design of programming languages and can serve as motivation for practitioners when selecting programming languages for new projects.

6.3 User Study

To evaluate the recommendations we derived from our findings, and to gather challenges faced by practitioners when using contracts, we conducted a qualitative survey study with 16 practitioners. In particular, we are interested in answering the following research questions (SRQs):

- **SRQ1**. [**Challenges**] What are the main challenges that users face when using contracts?
- **SRQ2**. [**Recommendations**] What do users recommend to increase contracts' adoption?

6.3.1 Methodology

To answer our RQs, we designed a qualitative survey study.

6.3.1.1 Recruitment

To improve the external validity, we allowed the participation of all kinds of software developers, but we recorded their experience with Android development. We recruited participants through Discord, LinkedIn, and our network (e.g., past students and colleagues). We also used snowball sampling by asking our contacts to distribute the study to their professional network. Our survey was implemented on Qualtrics and shared online. To prevent bots, all participants had to complete a reCAPTCHA challenge[14]. Per our inclusion criteria, participants were required to be at least 18 years old, in the United States, fluent in English, and possess some programming experience to ensure familiarity with basic software development concepts. All participants that we were able to recruit and who met the eligibility criteria were included in the final sample. Before deploying the study, we piloted it with five participants, iterating the survey between participants.

6.3.1.2 Survey Description

We begin our survey by showing participants the consent form. If they agree, we show the first section of our study, where we ask participants about their programming background and years of programming experience. Then, to ensure all participants are aware of the concept of DbC, we provide a short description and an example (see Figure 1). Participants are then asked about their confidence in understanding the definition of a contract, followed by questions regarding their frequency of contract use.

[14] reCAPTCHA is a security service provided by Google that protects websites from fraud and abuse by distinguishing human users from automated software.

Design by Contract is a technique in which software systems are seen as components that interact amongst themselves based on precisely defined specifications of client-supplier obligations (contracts).
Contracts are specifications of an agreement between the client and the supplier of a component, where the supplier expects that certain conditions are met by the client before using the component (preconditions), maintains certain properties from entry to the component to exit (invariants), and guarantees that certain properties are met upon exit (postconditions). These contracts can be written as assertions directly into the code.
For example, a way of enforcing a precondition in Java using exceptions might be:

```
public void proceedWithCheckout ( List < Item> shoppingCart ) {
if ( shoppingCart.isEmpty () ) {
throw new IllegalArgumentException () ;
}
...
}
```

Other examples include annotations such as `@NonNull` , which can be used to express preconditions. In Java and Kotlin, the assert keyword can be used to enforce the validity of a condition (for example, an invariant). APIs such as `org.apache.commons.lang.Validate.*` or `com.google.common.base.Preconditions.*` are also used to denote contracts. Finally, Kotlin offers features such as `@ExperimentalContracts` that allow the developer to state a method's behavior to the compiler explicitly.

Figure 1 Explanation shown to participants about DbC.

Here, the survey is split into two parts. For those who never use contracts, a follow-up section asks for the reasons for not using contracts. Participants who use contracts are asked to describe their reasons for using contracts and any challenges they have encountered. This is followed by the recommendations section. It begins by asking participants to suggest ways to improve the adoption of contracts. Following this, participants are presented with the following recommendations to improve contracts, obtained from the findings of our empirical study:

- Extend Java and Kotlin standard libraries with specialized constructs to specify contracts and with proper official documentation.
- Have libraries that standardize contract specifications in Java and Kotlin.
- Integrate into IDEs contract suggestion features supported by tools that automatically generate assertions and contracts.
- IDE and continuous integration plugins to automatically detect contract violations.

Participants were asked to rank these recommendations in terms of importance. Finally, the survey concludes with a demographic section.

The recommendations presented to participants in the user study were derived from our empirical findings but reformulated in a more concise and direct way. Presenting the recommendations exactly as shown in Section 6.2, which includes both context and the recommendation itself, was deemed too verbose for the user study.

6.3.1.3 Ethical Considerations

The study was approved by the IRB of Carnegie Mellon University. The participants did not receive payment upon survey completion. All participants were shown a consent form before filling in the survey. We did not collect any personally identifiable data.

6.3.1.4 Demographics

We recruited two participants for the initial pilot and three more for the follow-up pilot. For the finished survey, we recruited 23 participants. Of those 23, seven were ineligible or did not pass our screening questions (e.g., by not having programming experience). The remaining 16 participants sample is composed of individuals aged between 18 and 44 years, with most (nine participants) in the 25-34 age bracket. Gender representation includes male, female, non-binary/third gender, and one participant preferring not to disclose their gender. Educational backgrounds are high, with most participants holding graduate or

professional degrees and a smaller portion possessing bachelor's degrees. The sample is primarily White or Caucasian, with one Asian participant and one preferring not to disclose their race. Programming experience among the participants is diverse, with Python being the most commonly used language, followed by Java, JavaScript, C++, Rust, TypeScript, Go, C, Kotlin, and Dafny. All participants had some programming experience, with five participants having 1–3 years, three with 4–6 years, another five with 7–10 years, and finally, two with over 10 years of experience. Only one had less than one year of experience. Regarding experience with Android development, about half of the participants, 9 out of 16, had no years of experience. A subset had some experience, with one participant having between 1–3 years and another 7–10 years. The remaining five participants had less than one year of experience with Android software development.

6.3.1.5 Analysis

We used descriptive statistics to analyze the survey data from the closed-answer questions. For the qualitative responses, we developed three distinct codebooks tailored to different aspects of the dataset: 1) the reasons behind participants' use or non-use of contracts, 2) the challenges encountered while using contracts, and 3) the recommendations offered by participants to enhance the adoption of contracts. We used emergent coding techniques to develop the codebooks. We followed an iterative process to code the qualitative data. One of the researchers began by creating the first versions of the three codebooks. After this, two researchers independently double-coded all the answers, refined the codebook, recoded the answers again, and finally met to discuss any disagreements and reach a consensus.

6.3.2 Results

Among our survey participants, all, except one, reported using contracts in their programming practices, citing various reasons that underscore the multifaceted benefits of this approach. The participant who said they did not use contracts attributed their decision to the informal nature of their programming work, mainly prototyping and scripting. A significant majority, 11, highlighted the role of contracts in enhancing code quality and reliability. They mention that they use contracts to assert postconditions, verify preconditions, detect bugs, and identify edge-case bugs. This ensures that the code behaves as expected across compile-time and runtime scenarios. Four participants mentioned the importance of contracts as a documentation tool for improving code clarity. Three responses said that they used contracts in software design to manage expectations for software behavior. Lastly, two participants pointed out the operational benefits of contracts in enhancing the development process. They mentioned how contracts facilitate "sanity checks" (Participant 10) and ensure compliance with requirements.

6.3.2.1 SRQ1: Challenges

This subsection addresses SRQ1 and explores users' main challenges when using contracts in software development.

Participants provided diverse answers when questioned about their challenges when using contracts. In Table 14, we describe the codes and their respective frequency in participants' answers. The most cited obstacle was *Maintenance and Flexibility*, mentioned by three participants. This code highlights the sometimes complicated tasks of maintaining and updating contracts in complex projects. Participant 12 mentioned, "*if the implementation changes, we need to update the contract, and so, it can become complex to know which contracts*

Table 14 Codebook for participants' challenges when using contracts.

#	Code	Description
3	Maintenance and Flexibility	Problem with maintenance of contracts when implementations change, and the perceived lack of flexibility with contracts.
2	Specification and Expressiveness	Challenges in defining specifications and on the balance between contract expressiveness and automatic verification capabilities.
2	Cognitive Overload and Integration	Increased cognitive load due to managing both code and contracts, and integrating contracts into existing codebases.
2	Loop Invariants and Abstraction Levels	Specific challenges in formulating loop invariants and choosing the appropriate level of abstraction.
2	Enforcement Challenges	Challenges related to effectively enforcing contracts within the development process.
1	Security Concerns	Potential security risks.
1	Learning Curve and Documentation	Initial learning curve, difficulty in understanding contract libraries and navigating the documentation.

need to be updated". Challenges like *Specification and Expressiveness, Cognitive Overload, Loop Invariants*, and *Enforcement* were each present in the answers of two participants. And, finally, *Security Risks* and *Learning Curve and Documentation* were mentioned as challenges by one participant each.

6.3.2.2 SRQ2: Recommendations

This subsection addresses SRQ2 and users' recommendations to improve the adoption and usage of contracts in software development.

As mentioned before, we showed participants four recommendations obtained from our empirical study. Overall, participants seem to value all recommendations previously identified, as most classify them as "Very Important" and "Somewhat Important". The recommendation that participants seem to value the most is "IDE and continuous integration plugins to automatically detect contract violations" with 14 saying it is "Very Important" for them and two "Somewhat Important". This recommendation is closely followed by the one that suggests integrating contracts into IDEs ("Integrate into IDEs contract suggestion features supported by tools that automatically generate assertions and contracts") with 11 saying it is "Very Important" for them, and five "Somewhat Important." The remaining suggestions are to extend standard libraries with specialized constructs to specify contracts and with proper official documentation; these were also valued by participants, but one participant showed some uncertainty and indicated they were "Not sure" and classified it as "Not Important at All". Our results suggest that participants view the recommendations identified in our empirical work as valuable and support our insights.

Before asking participants to rank the previously identified recommendations, we asked them to suggest ways to improve the adoption of DbC. The codebook with the frequency of each code can be seen in Table 15. Participants' answers were diverse and seemed to also validate our results. The most frequent code in participants' suggestions is *Tool Support and Integration*: in total, seven participants suggested developing tools and IDE integrations that assist in creating, verifying, and managing contracts. This code validates our findings as it is similar to the recommendations that we derived from our empirical study. The second most frequent codes were the ones related to providing educational materials, templates, user-friendly interfaces, and robust error handling for users. The codes *Educational Resources and Training, Error Handling and Debugging Support, and User Interface and Templates*

Table 15 Codebook for participants' suggestions, including the frequency and description of each code.

#	Code	Description
7	Tool Support and Integration	Developing tools and IDE integrations that assist in creating, verifying, and managing contracts.
3	Educational Resources and Training	Providing more educational materials, examples, and training on DbC principles and benefits.
3	Error Handling and Debugging Support	Ensuring error recovery mechanisms and developing tools to simplify debugging processes related to contract violations.
2	Standards and Guidelines	Establishing standards or guidelines for how contracts should be defined, including preconditions and postconditions.
2	Incremental Adoption Strategies	Encouraging incremental adoption of DbC to make it easier for developers to integrate into their workflows.
2	User Interface and Templates	Providing user interfaces and templates to facilitate the writing of contracts and automatic code generation/repair.
2	NLP and AI	Utilizing NLP and AI for contract code suggestions.
2	Specification / Code Repair	Providing the ability to repair code based on changes to specifications (contracts) or update specifications based on code changes.
1	Programming Language Support	Enhancing programming language features to support contracts more effectively.
1	Automatic Verification and Testing	Improving automatic verification of contracts with less human effort and generating tests from contracts.
1	Real-Time Feedback and Metrics	Integrating real-time feedback and metrics within IDEs to provide indicators of code quality and contract coverage..

were each found three times in participants' answers. These recommendations suggest that participants need resources that support them in the practical implementation of contracts. Participant 9 directly says that they think that a way to improve the adoption of contracts is to "*always make sure there is a way to recover from the exceptions thrown whenever the assert (Python) statement is used.*" *Standards and Guidelines, Incremental Adoption Strategies, Natural Language Processing and AI, and Specification / Code Repair* were each mentioned twice. Particularly, *Natural Language Processing and AI* in similar ways by two participants, with Participant 7 saying "*I think AI contract code suggestions would reduce the barrier to entry and cost of writing the code.*" Finally, *Programming Language Support, Automatic Verification and Testing, and Real-Time Feedback and Metrics* were mentioned once, reinforcing that participants desire more automatic implementations of contracts and more feedback from their application.

Overall, our results suggest a clear direction – developers seem to desire improved tool support and integration of DbC in the development process. Our results highlight the need for future work on contracts and validate the findings of our empirical study.

6.4 Threats to Validity

Internal Validity. The accuracy of our results depends on the quality and correctness of the artifact, and there may exist bugs in the code. To mitigate this, we extensively tested the tool. In addition, all code and datasets used are publicly available for other researchers and potential users to check the validity of the results. Regarding the user study, one potential threat is the Hawthorne effect, where participants may alter their behaviour because they are aware they are being observed. To mitigate this risk, we ensured that participation was confidential and that responses could not be linked to individuals.

External Validity. The projects that we selected might not be an accurate representation of other, more popular, Android app stores. We mitigated this by using F-Droid, a collection of open-source applications commonly used in other research studies. We also mitigated this risk by analysing *all* the projects that satisfy the inclusion criteria, leading to a substantial dataset (51 MLoC) with applications of different types. Regarding the user study, one potential threat arises from the fact that about half of the participants lacked prior experience with Android development. As a result, the findings may not fully generalize.

Conclusion Validity. We might have missed language constructs that could be used to specify contracts. To mitigate this, we followed an established taxonomy [13] that we adapted and extended by systematically searching forums and the official Android documentation. The full list of constructs is available in the Supplementary Material [17]. Also, all our code is easily open to extension. Another risk comes from our dataset being imbalanced (with more Java than Kotlin applications). We mitigate this by explicitly discussing this imbalance when presenting results that might be affected by it.

7 Conclusions

Empirical evidence about contract usage can help the software engineering community create or improve existing libraries and tools to increase DbC adoption. This also helps to understand DbC's current practices better, helping practitioners discover and decide between different approaches. Researchers can also use our contributions to conduct additional studies.

Future work includes large-scale studies with practitioners to understand the challenges faced when specifying contracts, the use of annotations to improve Android analysis tools [24, 32, 30, 31], and the development of tools that can help increase the adoption of DbC [20, 43, 2].

References

1 Y. A.Feldman, O. Barzilay, and S. Tyszberowicz. Jose: aspects for design by contract. In *Fourth IEEE International Conference on Software Engineering and Formal Methods*, Los Alamitos, CA, USA, 2006.

2 Shibbir Ahmed, Sayem Mohammad Imtiaz, Samantha Syeda Khairunnesa, Breno Dantas Cruz, and Hridesh Rajan. Design by contract for deep learning apis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 94–106, 2023. doi:10.1145/3611643.3616247.

3 A. Algarni and K. Magel. Toward design-by-contract based generative tool for object-oriented system. In *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS). Proceedings*, pages 168–73, Piscataway, NJ, USA, 2018.

4 M. Aniche. *Effective Software Testing. A Developer's Guide*. Manning, Shelter Islands, 2022.

5 M. Backes, S. Bugiel, and E. Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 356–367. Association for Computing Machinery, 2016.

6 Joshua Bloch. *Effective java*. Addison-Wesley Professional, 2nd edition, 2008.

7 M. Blom, E. J. Nordby, and A. Brunstrom. On the relation between design contracts and errors: a software development strategy. In *Proceedings Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 110–117, 2002.

8 M. Blom, E.J. Nordby, and A. Brunstrom. An experimental evaluation of programming by contract. In *Proceedings Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 118–127, 2002.

9 C. Casalnuovo, P. Devanbu, A. Oliveira, V. Filkov, and B. Ray. Assert use in github projects. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE). Proceedings*, volume 1, pages 755–66, Los Alamitos, CA, USA, 2015.

10 P. Chalin. *Are practitioners writing contracts?*, pages 100–113. Springer, Berlin, Germany, 2006. doi:10.1007/11916246_5.

11 Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. Storydroid: Automated generation of storyboard for android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 596–607. IEEE, 2019. doi:10.1109/ICSE.2019.00070.

12 S. Counsell, T. Hall, T. Shippey, D. Bowes, A. Tahir, and S. MacDonell. Assert use and defectiveness in industrial code. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops*, pages 20–23, October 2017.

13 J. Dietrich, D. J. Pearce, K. Jezek, and P. Brada. Contracts in the wild: A study of java programs. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:29, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPICS.ECOOP.2017.9.

14 Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007. doi:10.1016/J.SCICO.2007.01.015.

15 H.-C. Estler, C. A. Furia, M. Nordio, M. Piccioni, and B. Meyer. Contracts in practice. In *FM 2014: Formal Methods. 19th International Symposium. Proceedings: LNCS 8442*, pages 230–46, Cham, Switzerland, 2014.

16 G. Fairbanks. Better code reviews with design by contract. *IEEE Software*, 36(6):53–6, 2019.

17 David R. Ferreira, Alexandra Mendes, João F. Ferreira, and Carolina Carreira. Contract usage and evolution in Android mobile applications (supplementary material), 2025. Available online at: https://archimendes.com/publication/2025/ecoop/ecoop25-AndroidContracts-SupplementaryMaterial.pdf.

18 L. Di Grazia and M. Pradel. The evolution of type annotations in python: An empirical study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 209–220, New York, NY, USA, 2022. Association for Computing Machinery.

19 B. Hollunder, M. Herrmann, and A. Hülzenbecher. Design by contract for web services: Architecture, guidelines, and mappings. In *International Journal on Advances in Software*, volume 5, 2012.

20 Marieke Huisman and Raúl E Monti. Teaching design by contract using snap! In *The Logic of Software. A Tasting Menu of Formal Methods: Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*, pages 243–263. Springer, 2022. doi:10.1007/978-3-031-08166-8_12.

21 P. Kochhar and D. Lo. Revisiting assert use in github projects. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 298–307, 2017.

22 Gunnar Kudrjavets, Nachiappan Nagappan, and Thomas Ball. Assessing the relationship between software assertions and faults: An empirical investigation. In *2006 17th International Symposium on Software Reliability Engineering*, pages 204–212, 2006. doi:10.1109/ISSRE.2006.14.

23 Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, et al. Verus: A practical

foundation for systems verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 438–454, 2024.

24 Olivier Le Goaer and Julien Hertout. Ecocode: A sonarqube plugin to remove energy smells from android projects. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–4, 2022.

25 K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pages 348–370. Springer, 2010. doi:10.1007/978-3-642-17511-4_20.

26 B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

27 Bertrand Meyer. Programming as contracting. *Advances in Object-Oriented Software Engineering*, pages 1–15, 1988.

28 P. V. R. Murthy. Design by contract methodology. In *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 482–8, Piscataway, NJ, USA, 2018.

29 A. Naumchev. Seamless object-oriented requirements. In *2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON). Proceedings*, Piscataway, NJ, USA, 2019.

30 Ricardo B Pereira, João F. Ferreira, Alexandra Mendes, and Rui Abreu. Extending Ecoandroid with automated detection of resource leaks. In *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, pages 17–27, 2022. doi:10.1145/3524613.3527815.

31 Ana Ribeiro, João F. Ferreira, and Alexandra Mendes. Ecoandroid: An Android studio plugin for developing energy-efficient Java mobile applications. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 62–69. IEEE, 2021. doi:10.1109/QRS54544.2021.00017.

32 Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. Jucify: A step towards android code unification for enhanced static analysis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1232–1244, 2022. doi:10.1145/3510003.3512766.

33 T. W. Schiller, K. Donohue, F. Coward, and M. D. Ernst. Case studies and tools for contract specifications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 596–607, New York, NY, USA, 2014. Association for Computing Machinery.

34 C. Silva, S. Guerin, R. Mazo, and J. Champeau. Contract-based design patterns: a design by contract approach to specify security patterns. In *ARES 2020: Proceedings of the 15th International Conference on Availability, Reliability and Security*, New York, NY, USA, 2020.

35 StatCounter Global Stats. Operating system market share worldwide, 2024. [Online; accessed 01-April-2025]. URL: https://gs.statcounter.com/os-market-share#monthly-202411-202412-bar.

36 J. Tantivongsathaporn and D. Stearns. An experience with design by contract. In *2006 13th Asia Pacific Software Engineering Conference (APSEC'06)*, pages 327–33, Piscataway, NJ, USA, 2006.

37 K. Tao and P. Edmunds. Mobile apps and global markets. *Theoretical Economics Letters*, 08:1510–1524, January 2018.

38 Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu. An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45, 2020.

39 Y. Wei, C.A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *2011 33rd International Conference on Software Engineering (ICSE 2011)*, pages 191–200, Piscataway, NJ, USA, 2011.

40 Z. Yu, C. Bai, L. Seinturier, and M. Monperrus. Characterizing the usage, evolution and impact of java annotations in practice. *IEEE Transactions on Software Engineering*, 47(5):969–986, 2021. doi:10.1109/TSE.2019.2910516.

41 Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun Chen. Studying the characteristics of logging practices in mobile apps: a case study on f-droid. *Empirical Software Engineering*, 24:3394–3434, 2019. doi:10.1007/S10664-019-09687-9.

42 Y. Zhou, P. Pelliccione, J. Haraldsson, and M. Islam. Improving robustness of autosar software components with design by contract: A study within volvo ab. In *Software Engineering for Resilient Systems. 9th International Workshop, SERENE 2017. Proceedings: LNCS 10479*, pages 151–68, Cham, Switzerland, 2017.

43 Álvaro Silva, Alexandra Mendes, and João F. Ferreira. Leveraging large language models to boost Dafny's developers productivity. In *International Conference on Formal Methods in Software Engineering (FormaliSE)*, 2024. arXiv:2401.00963.

Automatic Goal Clone Detection in Rocq

Ali Ghanbari
Auburn University, Al, USA

Abstract

Proof engineering in Rocq is a labor-intensive process, and as proof developments grow in size, redundancy and maintainability become challenges. One such redundancy is *goal cloning*, *i.e.*, proving α-equivalent goals multiple times, leading to wasted effort and bloated proof scripts. In this paper, we introduce clone-finder, a novel technique for detecting goal clones in Rocq proofs. By leveraging the formal notion of α-equivalence for Gallina terms, clone-finder systematically identifies duplicated proof goals across large Rocq codebases. We evaluate clone-finder on 40 real-world Rocq projects from the CoqGym dataset. Our results reveal that each project contains an average of 27.73 instances of goal clone. We observed that the clones can be categorized as either *exact goal duplication*, *generalization*, or *α-equivalent goals with different proofs*, each signifying varying levels duplicate effort. Our findings highlight significant untapped potential for proof reuse in Rocq-based formal verification projects, paving the way for future improvements in automated proof engineering.

2012 ACM Subject Classification Software and its engineering → Software maintenance tools; Software and its engineering → Formal software verification; Theory of computation → Type theory

Keywords and phrases Clone Detection, Goal, Proof, Rocq, Gallina

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.12

Supplementary Material *Software (Source code)*: `https://aub.ie/cfsrc`
archived at `swh:1:dir:3c32ef30be8294f864b7be5fd7109522fe5a2eb1`

Acknowledgements The author thanks Anonymous ECOOP 2025 Reviewers for their insightful and constructive feedback that significantly improved this paper.

1 Introduction

Rocq [20][1] is a popular interactive theorem prover based on type theory. Rocq has proved to be a viable tool for constructing certified software, as demonstrated by projects like CompCert [27], Iris [22], ConCert [2], and Fiat Cryptography [13].

Proof engineering [33] in Rocq is a labor-intensive task [6, 15, 14]. As proof developments grow in size, redundancy and maintainability become major challenges. An important, yet relatively overlooked, issue in proof engineering research is *goal cloning* – proving of α-equivalent goals more than once. Such a duplication of effort not only results in waste of manpower, but also inflates proof scripts, increases maintenance costs, and reduces opportunities for reuse. While Rocq offers mechanisms for structuring proofs, avoiding redundancy remains a challenge. Large-scale verification projects, such as seL4 [26] and Verisoft [1], have highlighted the necessity of tools that support automated detection of redundant proof efforts [6]. While there is tool support for code clone detection [18, 40] in Isabelle/HOL [30], to the best of our knowledge, detection of redundant proof efforts in Rocq proofs is not explored yet.

The problem that we set out to solve is as follows. *Given a set of Rocq proofs, find the set of α-equivalent goals making up the proofs.* "Goal" is a Rocq terminology for proof obligation; each proof comprises sub-proofs for a set of goals that are written as Gallina [20]

[1] The Coq development community recently renamed the system to The Rocq Prover. In this paper, we use Coq and Rocq interchangeably.

© Ali Ghanbari;
licensed under Creative Commons License CC-BY 4.0

terms. We want to automatically identify *goal clones*, *i.e.*, α-equivalent goals. Such goals, and their corresponding sub-proofs, can be factored out as independent lemmas and reused, instead of proving them from scratch each time they are observed. To solve this, we introduce clone-finder, a novel technique that leverages the formal definition of α-equivalence of Gallina terms to systematically identify duplicated proof goals across large Rocq codebases. This enables proof engineers to refactor their developments by extracting common subproofs as reusable lemmas, thereby reducing redundant proof effort and improving maintainability. Although the problem statement is simple, and the notion of α-equivalence of Gallina terms has a solid foundation in the well-established field of type theory, finding α-equivalent Gallina terms in practice is not trivial. This is mainly because the variables mean different things in different contexts and any comparison must first universally quantify the terms with the free variables defined in the context before performing α-equivalence check (see §3 for more details on the proposed approach).

Our prototype Python implementation of clone-finder interacts with Rocq through Coq-LSP [9] and its Python interface, CoqPyt [7]. We evaluate clone-finder on 40 real-world Rocq projects from the CoqGym dataset [42]. Our benchmark contains a diverse set of Rocq projects, representative of the real-world Rocq projects: the project are of different sizes, ranging from 53 to 29,260 lines of code, and they are from various domains – from set theory to distributed system verification to geometry (see §5.1 for more details on benchmark preparation process). We ran clone-finder on our benchmark of Rocq projects, while measuring the number of goal clones that it detects as well as its runtime overhead. We observed that in 20 out of 40 projects, clone-finder detected at least one goal clone. On average clone-finder finds 27.73 instances of goal clones per project. Manual review of the reported clones revealed three common patterns of goal cloning: (1) exact goal duplication with identical or near-identical proofs, *e.g.*, structurally identical proofs with different variable names; (2) α-equivalent goals with generalized proofs, where one proof subsumes the other; and (3) α-equivalent goals with entirely different proofs. Each of these instances signifies extra spent proof effort that could be saved and repurposed for other tasks by factoring out the duplicated goals as independent lemmas. These findings suggest significant untapped potential for proof reuse in Rocq-based formal verification projects.

Another observation that we make is that clone-finder is a light-weight system: over 94.71% of its running time is from using Coq-LSP through CoqPyt. An average 45.31 seconds of run time, for a fresh run, and support for incremental analysis, which could significantly reduce the run time, make clone-finder a practical tool for daily proof engineering. Mitigating the overhead of interfacing Coq-LSP is a matter of investing engineering effort so that CoqPyt does not reload common dependencies of the proof scripts within a workspace every time we load a script from that workspace. Another possibility is to implement clone-finder directly as a Rocq plugin using Rocq plugin library and Coq-Elpi [8]. We save these as a future extension of this work.

Beyond detecting proof redundancy, our work contributes an open-source enhancement to CoqPyt, which enables analyzing real-world Rocq projects with it. We will make both clone-finder and our enhancement publicly available to foster further research in automated proof engineering. To summarize, this paper makes the following contributions.

- **Technique:** a simple technique, named clone-finder, based on the theoretically elegant notion of α-equivalence is proposed for detecting goal clones in Rocq scripts. This technique can be used alongside lemma extraction tool-set [32, 35] to rewrite clones as independent lemmas to increase reuse and mitigate waste of proof efforts. Additionally, the support for incremental analysis would allow using clone-finder in a background thread in a Rocq IDE to detect clones as the proof engineer writes the proofs.

- **Evaluation:** We evaluate clone-finder using a set of real-world Rocq projects. We report that real-world Rocq projects are likely to contain large numbers of clones with either (1) identical; (2) generalized; or (3) entirely different pairs of proofs. These clones indicate waste of manpower in proof engineering that could potentially be repurposed for use in other aspects of system specification and verification.
- **Open-Source Contribution:** While implementing clone-finder, which interacts with the Python interface for Coq-LSP, named CoqPyt, we realized that CoqPyt does not allow passing physical-to-logical path mapping, which is essential in working with real-world Rocq projects. We have added this feature to the framework and will make a pull request about it. We have also made clone-finder publicly available [17]. This source code contains a Python-based parser for Gallina that can be used in future research projects.

The remainder of this paper is organized as follows. We provide background on Rocq and code clone detection in §2. In §3 we present the clone-finder approach in detail, and in §4 we discuss the implementation of our prototype. We present our experimental evaluation in §5, followed by an analysis of threats to validity in §6. We review related work in §7 and conclude the paper, while sketching future research directions, in §8.

2 Background

In this section, we briefly introduce Rocq, Gallina, Calculus of Inductive Constructions, and code clone detection. We also provide references for further reading on these topics.

2.1 Rocq, Gallina, and Calculus of Inductive Constructions

Rocq [20], formerly known as Coq, is an interactive theorem prover. At its core, lies a specification language, named Gallina [20]. Gallina is an elegant functional programming language that was originally built upon the calculus of inductive constructions with definitions [4, 31, 39]. This calculus still constitutes a substantial subset of Gallina today. Proofs in Rocq are usually written in a tactic language, named Ltac [12, 20], which is a domain-specific language for building Gallina terms in a type-driven fashion that serve as proofs. The type of these terms are also Gallina terms, that serve as theorem statements and goals.

Rocq is a state-based interactive proof assistant. Theorem proving in Rocq starts with the theorem itself as the initial goal. The proof engineer repeatedly applies Ltac tactics to decompose the goal into a list of subgoals. The proof is complete when there are no sub-goals left. At each step, Rocq displays the local definitions as available hypotheses that can be used in theorem proving. A successful Rocq proof implicitly amounts to creating a proof tree whose root is the original theorem and whose nodes are the goals. Figure 1 shows a simple example. All goals share the same global context, but have exclusive local contexts, *e.g.*, hypotheses a, b : nat in Figure 1. The edges of the proof tree are annotated with tactics. The proof for a goal in the proof tree can be obtained by conducting a depth-first search [10] on the node corresponding to the goal and accumulating the tactics that are visited along the way.

2.2 Code Clone Detection

Code clone detection is an active area of research in software engineering aimed at identifying duplicated code snippets within software systems [28, 36, 29, 41]. Code duplication is widespread in software development. It is sometimes introduced deliberately to accelerate prototyping, reuse existing functionality, or meet tight deadlines, and other times it arises inadvertently. However, excessive cloning can degrade code quality, increase maintenance overhead, and propagate defects across multiple locations in the codebase [23, 24].

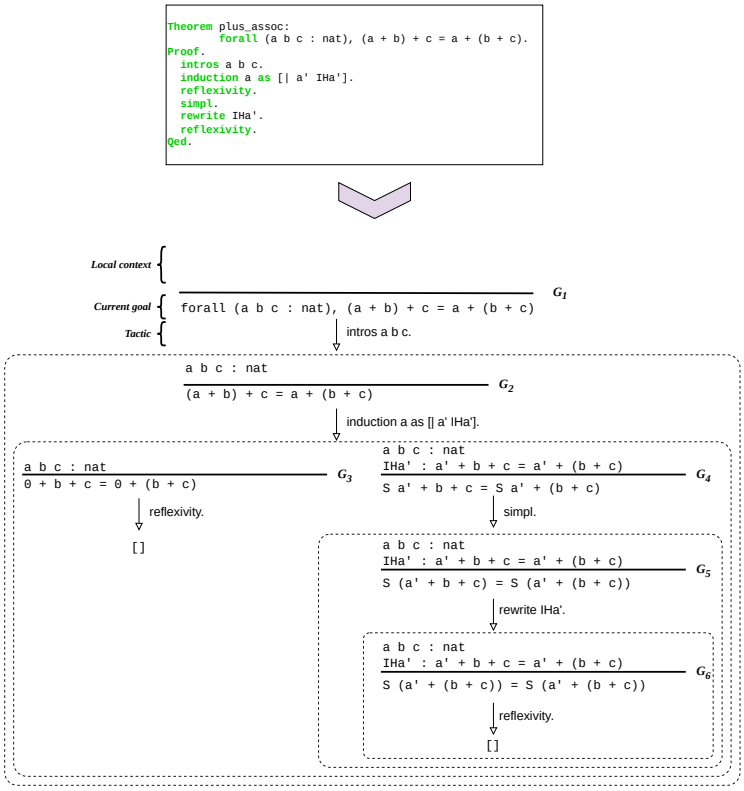


Figure 1 A simple Rocq theorem and its proof in Ltac. The proof process starts with the theorem statement as the current goal. Each valid Ltac command transforms the state of the proof assistant, prompting it to generate zero or more new goals. This amounts to creating a proof tree, where nodes corresponds to goals and the edges are labeled with the Ltac command that generated the child nodes. A depth-first search on each node of the tree to accumulate the tactics at the edges constitute the sub-proof for the goal at each node. This figure also illustrates how Algorithm 1 constructs proof trees by executing proofs line by line, while tracking the goals that are created and eliminated as a result of running each tactic.

Code clones refer to code snippets that exhibit similarity based on a defined metric. The literature categorizes code clones into four primary types [36, 29]: (1) exact clones; (2) renamed clones; (3) near-miss clones; and (4) semantic clones.

While the code clone detection technique introduced in this paper falls under the categories of type 1 and type 2 clones (our technique finds α-equivalent goals in a project), the type of code that we study in this paper are fundamentally different from that of studied in the literature. We are finding clones in functional programs, rather than imperative programs, so the existing techniques for code clone detection [28, 29, 41] could not be used directly.

3 The clone-finder Approach

At the highest level, clone-finder takes, as input, the base path for a Rocq project, Rocq options, and the minimum proof size threshold, and outputs a list of clones with proofs at least as large as the provided threshold. The proof size is defined in terms of lines of Ltac code. The operations of clone-finder can be summarized as (1) constructing proof trees

$$
\begin{array}{rcl}
t & ::= & s \mid x \mid t_1\ t_2 \mid \texttt{forall}\ (x : t_1),\ t_2 \\
& & \mid \texttt{fun}\ (x : t_1)\ \Rightarrow\ t_2 \mid \texttt{let}\ x : t_1 \coloneqq t_2\ \texttt{in}\ t_3 \\
& & \mid \texttt{fix}\ y\ (x : t_1) : t_2 \coloneqq t_3 \\
& & \mid \texttt{match}\ t_1\ \texttt{return}\ t_2\ \texttt{with}\ \ [\mid x_1\ \ldots\ x_n\ \Rightarrow t_3]^*\ \texttt{end}
\end{array}
$$

$$
\begin{array}{rrcl}
\text{(terms)} & t, t_1, t_2, t_3 & \in & \mathbb{T} \\
\text{(sorts)} & s & \in & \{\texttt{Set}, \texttt{Type}\} \\
\text{(variables)} & x, y, x_1, \ldots, x_n & \in & \mathbb{V}
\end{array}
$$

Figure 2 Abstract syntax of a core subset of Gallina terms.

(described in Algorithm 1); (2) flattening, deduplicating, and generalization of the proof trees to construct the list of pairs of goals and their corresponding proofs (described in Algorithms 2 and 3); (3) finding pairs of α-equivalent generalized goals and reporting the output (described in Algorithm 4). We now describe each of these steps in more details.

The enabling tool of clone-finder are the notions of free variables and α-equivalence. To define these notions more precisely, we use a core subset of Gallina subsumed by the calculus of inductive constructions with definitions. Figure 2 shows this subset of Gallina. Note that while inductive type definitions are part of the syntax of the calculus of inductive constructions with definitions, we have not included them in our core language, because this is how Rocq operates: after an inductive type definition is type checked, the type name, its constructors, as well as induction principles and recursor, are added as constants into the global type environment and type definition themselves do not appear in goals. The following definitions define notions of free variables, substitution, and α-equivalence. Readers familiar with these standard notions may skip the following three definitions.

▶ **Definition 1** (Free Variables). *The set of free variables of a Gallina term t, denoted $FV(t)$, is defined recursively as follows.*

- $FV(s) = \emptyset$, *for all sorts* s,
- $FV(x) = \{x\}$, *for all* $x \in \mathbb{V}$,
- $FV(t_1\ t_2) = FV(t_1) \cup FV(t_2)$ *for all* $t_1, t_2 \in \mathbb{T}$,
- $FV(\texttt{forall}\ (x : t_1),\ t_2) = FV(t_1) \cup (FV(t_2) - \{x\})$, *for* $x \in \mathbb{V}$ *and* $t_1, t_2 \in \mathbb{T}$,
- $FV(\texttt{fun}\ (x : t_1)\ \Rightarrow\ t_2) = FV(t_1) \cup (FV(t_2) - \{x\})$, *for* $x \in \mathbb{V}$ *and* $t_1, t_2 \in \mathbb{T}$,
- $FV(\texttt{let}\ x : t_1 \coloneqq t_2\ \texttt{in}\ t_3) = FV(t_1) \cup FV(t_2) \cup (FV(t_3) - \{x\})$, *for* $x \in \mathbb{V}$ *and* $t_1, t_2, t_3 \in \mathbb{T}$,
- $FV(\texttt{fix}\ y\ (x : t_1) : t_2 \coloneqq t_3) = FV(t_1) \cup FV(t_2) \cup (FV(t_3) - \{x, y\})$,
- $FV(\texttt{match}\ t_1\ \texttt{return}\ t_2\ \texttt{with}\ |C_1| \ldots |C_k\ \texttt{end}) = FV(t_1) \cup FV(t_2) \cup FV_C(C_1) \cup \cdots \cup FV_C(C_k)$, *for* $k \geq 0$,

where s is a sort, $x, y \in \mathbb{V}$, and $t_1, t_2, t_3 \in \mathbb{T}$. The helper function FV_C is defined to be $FV_C(x_1\ \ldots\ x_n\ \Rightarrow t_3) = FV(t_3) - \{x_1, \ldots, x_n\}$, wherein $n \geq 1$ and $x_1, \ldots, x_n \in \mathbb{V}$.

All the non-free variables occurring in a Gallina term are referred to as *bound variables*. Two Gallina terms are said to be α-equivalent, if we can systematically rename bound variables in one term to make it syntactically equal to another. To define this notion precisely, we first need to define variable replacement, which is used for variable renaming in the α-equivalence check.

▶ **Definition 2** (Substitution). *Given a variable x and a term u, $t^{x\to u}$ denotes the substitution of x with u in t, and is defined as follows.*

- $s^{x\to u} = s$, *for all sorts* s,
- $x_0^{x\to u} = u$, *if* $x_0 = x$,
- $x_0^{x\to u} = x_0$, *for all* $x_0 \in \mathbb{V}$ *such that* $x_0 \neq x$,
- $(t_1\ t_2)^{x\to u} = t_1^{x\to u}\ t_2^{x\to u}$, *for all* $t_1, t_2 \in \mathbb{T}$,
- $(\texttt{forall}\ (x_0 : t_1),\ t_2)^{x\to u} = \texttt{forall}\ (x_0 : t_1^{x\to u}),\ t_2^{x\to u}$, *for all* $x_0 \in \mathbb{V}$ *and* $t_1, t_2 \in \mathbb{T}$ *such that* $x_0 \neq x$ *and* $x_0 \notin FV(u)$,
- $(\texttt{forall}\ (x_0 : t_1),\ t_2)^{x\to u} = \texttt{forall}\ (x_0 : t_1^{x\to u}),\ t_2$, *if* $x_0 \notin FV(u)$,
- $(\texttt{fun}\ (x_0 : t_1)\ \Rightarrow\ t_2)^{x\to u} = \texttt{fun}\ (x_0 : t_1^{x\to u})\ \Rightarrow\ t_2^{x\to u}$, *for all* $x_0 \in \mathbb{V}$ *and* $t_1, t_2 \in \mathbb{T}$ *such that* $x_0 \neq x$ *and* $x_0 \notin FV(u)$,
- $(\texttt{fun}\ (x_0 : t_1)\ \Rightarrow\ t_2)^{x\to u} = \texttt{fun}\ (x_0 : t_1^{x\to u})\ \Rightarrow\ t_2$, *for all* $t_1, t_2 \in \mathbb{T}$ *such that* $x_0 \in FV(u)$,
- $(\texttt{let}\ x_0 : t_1 \coloneqq t_2\ \texttt{in}\ t_3)^{x\to u} = \texttt{let}\ x_0 : t_1^{x\to u} \coloneqq t_2^{x\to u}\ \texttt{in}\ t_3^{x\to u}$, *for all* $x_0 \in \mathbb{V}$ *and* $t_1, t_2, t_3 \in \mathbb{T}$ *such that* $x_0 \neq x$ *and* $x_0 \notin FV(u)$,
- $(\texttt{let}\ x_0 : t_1 \coloneqq t_2\ \texttt{in}\ t_3)^{x\to u} = \texttt{let}\ x_0 : t_1^{x\to u} \coloneqq t_2^{x\to u}\ \texttt{in}\ t_3$, *for all* $t_1, t_2, t_3 \in \mathbb{T}$ *such that* $x_0 \in FV(u)$,
- $(\texttt{fix}\ y\ (x_0 : t_1) : t_2 \coloneqq t_3)^{x\to u} = \texttt{fix}\ y\ (x_0 : t_1^{x\to u}) : t_2^{x\to u} \coloneqq t_3^{x\to u}$, *for all* $y, x_0 \in \mathbb{V}$ *and* $t_1, t_2, t_3 \in \mathbb{T}$ *such that* $y \neq x$, $x_0 \neq x$, $x_0 \notin FV(u)$, *and* $y \notin FV(u)$,
- $(\texttt{fix}\ y\ (x_0 : t_1) : t_2 \coloneqq t_3)^{x\to u} = \texttt{fix}\ y\ (x_0 : t_1^{x\to u}) : t_2^{x\to u} \coloneqq t_3$, *for all* $t_1, t_2, t_3 \in \mathbb{T}$ *such that* $x_0 \in FV(u)$ *or* $y \in FV(u)$,
- $(\texttt{match}\ t_1\ \texttt{return}\ t_2\ \texttt{with}\ |C_1| \ldots |C_k\ \texttt{end})^{x\to u} = \texttt{match}\ t_1^{x\to u}\ \texttt{return}\ t_2^{x\to u}\ \texttt{with}\ |C_1^{x\to u}| \ldots |C_k^{x\to u}\ \texttt{end}$, *where* $C_i^{x\to u}$ *is defined as* $(x_1\ \ldots\ x_n\ \Rightarrow t_3)^{x\to u} = (x_1\ \ldots\ x_n\ \Rightarrow t_3^{x\to u})$, *if* $x \notin \{x_1, ..., x_n\}$ *and* $\{x_1, ..., x_n\} \cap FV(u) = \emptyset$, *and* $(x_1\ \ldots\ x_n\ \Rightarrow t_3)^{x\to u} = x_1\ \ldots\ x_n\ \Rightarrow t_3$ *if* $\{x_1, ..., x_n\} \cap FV(u) \neq \emptyset$.

We now define the notion of α-equivalence as follows.

▶ **Definition 3** (α-equivalence). *The following axiom and rules of inference prove judgments of the form $t_1 =_\alpha t_2$, where t_1 and t_2 are Gallina terms.*

- ***(Renaming-Forall)**:* $(\texttt{forall}\ (x : t_1),\ t_2) =_\alpha (\texttt{forall}\ (y : t_1'),\ t_2')$, *if* $t_1 =_\alpha t_1'$, $y \notin FV(t_2)$, y *is not bound in* t_2, *and* $t_2^{x\to y} =_\alpha t_2'$,
- ***(Renaming-Fun)**:* $(\texttt{fun}\ (x : t_1)\ \Rightarrow\ t_2) =_\alpha (\texttt{fun}\ (y : t_1')\ \Rightarrow\ t_2')$, *if* $t_1 =_\alpha t_1'$, $y \notin FV(t_2)$, y *is not bound in* t_2, *and* $t_2^{x\to y} =_\alpha t_2'$,
- ***(Renaming-Let)**:* $(\texttt{let}\ x : t_1 \coloneqq t_2\ \texttt{in}\ t_3) =_\alpha (\texttt{let}\ y : t_1' \coloneqq t_2'\ \texttt{in}\ t_3')$, *if* $t_1 =_\alpha t_1'$, $t_2 =_\alpha t_2'$, $y \notin FV(t_3)$, y *is not bound in* t_3, *and* $t_3^{x\to y} =_\alpha t_3'$,
- ***(Renaming-Fix)**:* $(\texttt{fix}\ y\ (x : t_1) : t_2 \coloneqq t_3) =_\alpha \texttt{fix}\ y'\ (x' : t_1') : t_2' \coloneqq t_3'$, *if* $t_1 =_\alpha t_1'$, $t_2 =_\alpha t_2'$, $y' \notin FV(t_3)$, y' *is not bound in* t_3, $x' \notin FV(t_3)$, x' *is not bound in* t_3, *and* $t_3^{y\to y'\ x\to x'} =_\alpha t_3'$,
- ***(Renaming-Match)**:* $(\texttt{match}\ t_1\ \texttt{return}\ t_2\ \texttt{with}\ |\ C_1\ |\ \ldots\ |\ C_k\ \texttt{end})$ $=_\alpha (\texttt{match}\ t_1'\ \texttt{return}\ t_2'\ \texttt{with}\ |\ C_1'\ |\ \ldots\ |\ C_k'\ \texttt{end})$, *if* $t_1 =_\alpha t_1'$, $t_2 =_\alpha t_2'$, *and for all* $1 \leq i \leq k$, *we have* $C_i \equiv_\alpha C_i'$, *where* $\equiv_\alpha$ *is defined as:* $(x_1\ \ldots\ x_n\ \Rightarrow t_3) \equiv_\alpha (x_1'\ \ldots\ x_n'\ \Rightarrow t_3')$ *if* $x_1'\ \ldots\ x_n' \notin FV(t_3)$, *variables* $x_1'\ \ldots\ x_n'$ *are not bound in* t_3, *and* $t_3^{x_1\to x_1'\ \ldots\ x_n\to x_n'} =_\alpha t_3'$.
- ***(Compatibility-1)**: If* $t_1 =_\alpha t_2$, *then* $t_1\ t =_\alpha t_2\ t$, $t\ t_1 =_\alpha t\ t_2$, $(\texttt{forall}\ (x : t_1), t) =_\alpha (\texttt{forall}\ (x : t_2), t)$, *and* $(\texttt{fun}\ (x : t_1) \Rightarrow t) =_\alpha (\texttt{fun}\ (x : t_2) \Rightarrow t)$, *for all term* t,
- ***(Compatibility-2)**: If* $t_1 =_\alpha t_2$, *then* $(\texttt{match}\ t_1\ \texttt{return}\ t\ \texttt{with}\ |\ C_1\ |\ \ldots\ |\ C_k\ \texttt{end}) =_\alpha (\texttt{match}\ t_2\ \texttt{return}\ t\ \texttt{with}\ |\ C_1\ |\ \ldots\ |\ C_k\ \texttt{end})$ *and* $(\texttt{match}\ t\ \texttt{return}\ t_1\ \texttt{with}\ |\ C_1\ |\ \ldots\ |\ C_k\ \texttt{end}) =_\alpha (\texttt{match}\ t\ \texttt{return}\ t_2\ \texttt{with}\ |\ C_1\ |\ \ldots\ |\ C_k\ \texttt{end})$, *for all term* t *and match clauses* C_1, ..., C_k.

Algorithm 1 Constructing proof trees from Rocq proofs.

Require: List of Rocq source files, coq_files, and `coqc` flags, stored in variable "flags"
Ensure: Returning a mapping from ⟨File, Theorem⟩ to ProofTree

```
tree_map ← ∅
for all file ∈ coq_files do                          ▷ Construct a proof tree for each theorem in each file
    pf ← connect_to_coq_lsp(file, flags)             ▷ Connect to Coq-LSP: an iterable proof file object is returned
    for all step ∈ pf do                             ▷ Sweep the current file step-by-step
        step.execute()
        if step.in_proof() then                      ▷ Have stepped into the proof of a theorem?
            thm ← step.get_theorem()
            tree ← ProofTree()                       ▷ Initialize a new proof tree
            tree_map[⟨file, thm⟩] ← tree             ▷ Store the proof tree
            prev ← ∅                                 ▷ Track goals at the last step
            while not step.can_close() do            ▷ Iterate through the proof steps until no more goals remain
                step.execute()
                curr ← ∅                             ▷ Store current proof state
                for all g ∈ pf.goals() do
                    ctx ← g.get_hypotheses()         ▷ Extract local and in-file context (hypotheses)
                    goal ← g.get_type()              ▷ Extract the goal type (i.e., goal statement)
                    curr ← curr ∪ {⟨ctx, goal⟩}      ▷ Store current goals
                end for
                if prev ≠ ∅ then
                    gone ← {t ∈ prev | t ∉ curr}     ▷ Goals that disappeared
                    new ← {t ∈ curr | t ∉ prev}      ▷ Newly introduced goals
                    for all t ∈ gone do
                        ctx ← t[0]
                        goal ← t[1]
                        children ← {c[1] | c ∈ new}  ▷ Children are new goals
                        tac ← step.get_tactic()      ▷ Get the applied tactic
                        tree.AddNode(goal, ctx, children, tac)
                                                     ▷ Add node to proof tree and connect it to its children
                    end for
                end if
                prev ← curr                          ▷ Update tracking state
            end while
        end if
    end for
    close(pf)                                        ▷ Close the proof file and the Coq-LSP connection
end for
return tree_map                                      ▷ Return the constructed proof tree map
```

- ***(Compatibility-3)***: *If* $t_1 =_\alpha t_2$, *then* $(\texttt{let}\ x : t_1 := t\ \texttt{in}\ u) =_\alpha (\texttt{let}\ x : t_2 := t\ \texttt{in}\ u)$, $(\texttt{let}\ x : t := t_1\ \texttt{in}\ u) =_\alpha (\texttt{let}\ x : t := t_2\ \texttt{in}\ u)$, $(\texttt{fix}\ y(x : t_1) : t \Rightarrow u) =_\alpha (\texttt{fix}\ y(x : t_2) : t \Rightarrow u)$, *and* $(\texttt{fix}\ y(x : t) : t_1 \Rightarrow u) =_\alpha (\texttt{fix}\ y(x : t) : t_2 \Rightarrow u)$, *for all terms* t *and* u,
- ***(Compatibility-4)***: *If* $t_1 =_\alpha t_2$, *then* $(\texttt{forall}\ (x : t), t_1) =_\alpha (\texttt{forall}\ (z : t), t_2)$, $(\texttt{fun}\ (x : t) \Rightarrow t_1) =_\alpha (\texttt{fun}\ (z : t) \Rightarrow t_2)$, *for all term* t *and variable* z,
- ***(Compatibility-5)***: *If* $t_1 =_\alpha t_2$, *then* $(\texttt{let}\ x : t := u\ \texttt{in}\ t_1) =_\alpha (\texttt{let}\ z : t := u\ \texttt{in}\ t_2)$, $(\texttt{fix}\ y(x : t) : u \Rightarrow t_1) =_\alpha (\texttt{fix}\ z(x : t) : u \Rightarrow t_2)$, *and* $(\texttt{fix}\ y(x : t) : u \Rightarrow t_1) =_\alpha (\texttt{fix}\ y(z : t) : u \Rightarrow t_2)$, *and* $(\texttt{match}\ t\ \texttt{return}\ u\ \texttt{with} \mid C_1 \mid \ldots \mid (x_1\ \ldots\ x\ \ldots\ x_n) \Rightarrow t_1\ \ldots \mid C_k\ \texttt{end}) =_\alpha (\texttt{match}\ t\ \texttt{return}\ u\ \texttt{with} \mid C_1 \mid \ldots \mid (x_1\ \ldots\ z\ \ldots\ x_n) \Rightarrow t_2\ \ldots \mid C_k\ \texttt{end})$, *for all terms* t *and* u, *and variable* z,
- ***(Reflexivity)***: $t =_\alpha t$, *for all term* t,
- ***(Symmetry)***: *If* $t_1 =_\alpha t_2$ *then* $t_2 =_\alpha t_1$,
- ***(Transitivity)***: *If* $t_1 =_\alpha t_2$ *and* $t_2 =_\alpha t_3$ *then* $t_1 =_\alpha t_3$.

Renaming rules are the foundations of α*-equivalence. Compatibility rules have the effect of extending* α*-equivalence from subterms to bigger terms. Reflexivity, symmetry, and transitivity make* α*-equivalence an equivalence relation. If we can derive* $t_1 =_\alpha t_2$, *we say that the terms* t_1 *and* t_2 *are* α*-equivalent.*

Armed with these definitions, we are now in a position to define our algorithms making up clone-finder. The first step is to extract goals and construct proof trees, which is described in Algorithm 1. This step is performed to find the proof corresponding to each goal in a Rocq project. clone-finder invokes this algorithm by first finding all the `.v` files in the target

Algorithm 2 Flatten and remove redundant goals.

Require: pt_map, a mapping from ⟨File, Theorem⟩ to ProofTree, *i.e.*, Algorithm 1's output
Ensure: List of goals and their generalizations, where redundant goals have been removed

```
1: g_list ← []                                              ▷ Initialize list for storing goals
2: for all (f, t, pt) ∈ pt_map do                           ▷ Iterate over proof tree map
3:     for all g ∈ pt.GetNodes() do                         ▷ Extract goals from proof tree
4:         ctx ← g.GetContext()
5:         g_gen ← Generalize(ctx, g)
6:         p ← g.GetProof()
7:         g_list.append(⟨g, g_gen, p, f, t⟩)
8:     end for
9: end for
10: red_g ← {}                                              ▷ Initialize set for redundant goals
11: for i = 0 to |g_list| − 1 do                            ▷ Iterate through goals list
12:     for j = 0 to |g_list| − 1 do                        ▷ Compare each goal with others
13:         if i ≠ j and prodBody(g_list[i][1], g_list[j][1]) then   ▷ Check if one goal is part of another
14:             red_g.add(g_list[j][1])
15:         end if
16:     end for
17: end for
18: g_list ← [g ∈ g_list | g[1] ∉ red_g]                    ▷ Remove redundant goals
19: return g_list
```

Rocq project directory. This list, together with `coqc` flags are passed to the algorithm. In summary, this algorithm runs each Rocq script in the given project step by step to construct proof trees using a tree data structure. Proof tree construction is done as follows. After executing a tactic, the current goal disappears, and a set of new goals might be created, these goals are regarded as the children of the current goal in the proof tree. We can identify the edges of the tree by tracking how goals are created during the proof execution. For example, if the list of goals changes from $[A, B]$ to $[C, D, B]$, we know that node A has two children C and D. Algorithm 1 creates nodes that point to the children node with the help of a dictionary-based data structure that maps current goal to the child goals and other information about the current goals such as the tactic that transforms it to the child goals and the context right before the execution of the tactic. Figure 1 provides a simple example of a Rocq proof and its corresponding proof tree.

More precisely, Algorithm 1 constructs a mapping of Rocq scripts to their corresponding proof trees, capturing the hierarchical structure of goals and tactics used to prove them. It iterates over each Rocq source file (line 2), establishing a connection to Coq-LSP to analyze proof steps (lines 3 and 4). When a proof of a theorem is encountered (line 6), the algorithm initializes a proof tree data structure and adds it to the resulting map, mapping file name and the the current theorem name pairs to the newly created proof tree (lines 7-9). This proof tree object, gets populated as follows. The algorithm iterates through proof steps until the proof is complete (line 11). At each step, it extracts the current goals and the local, and in-file, contexts (lines 13–18), detects goals that have disappeared or emerged (lines 20–21), and records the proof structure by adding nodes to the proof tree (lines 22–29). Finally, after processing all steps, the proof file is closed (line 35), and the mapping of file names and theorem names to proof trees is returned (line 31).

The next step in clone-finder's pipeline is to flatten the proof trees by turning the proof tree map generated by Algorithm 1 into a list of tuples of a goal, its generalization (*i.e.*, universal quantification of its free variables), its proof, its containing file name, and its containing theorem name. At this step, clone-finder also deduplicates the goals by removing the tuples whose generalization is a body of another tuple's generalization. Algorithm 2 formalizes this process. This algorithm processes a proof tree map by flattening its structure and eliminating redundant goals. First, it initializes an empty list g_list (line 1) to store extracted goals and their associated information. Then, it iterates through the proof tree

Algorithm 3 Generalization of a Goal.

Require: ctx: A map from variables to terms
Require: goal: The goal to be generalized
Ensure: A generalized goal with universally quantified variables
1: fvs ← $\{v \in FV(\text{goal}) \mid v \in \text{ctx}\}$ ▷ Find free variables bound in the context
2: rev_top_order ← reverse(topo_sort(ctx, fvs)) ▷ Sort dependencies in reverse topological order
3: **for** each $v \in$ rev_top_order **do**
4: goal ← "`forall` ("v" : "ctx[v]", "goal")" ▷ Prepend universal quantifiers
5: **end for**
6: **return** goal

map (pt_map) and extracts goals from each proof tree (lines 2–3). The method GetNodes of proof tree, returns the set of nodes, *i.e.*, the goals, in the proof tree. For each goal, it retrieves its context, generalizes it, and pairs it with its proof, file, and theorem name before appending this information to g_list (lines 4–7). The method GetProof, receives a node, *i.e.*, a goal, as input, conducts a depth-first search rooted at that node, while accumulating the tactics along each edge that it visits. The list of tactics ultimately returned by the method constitutes the proof for that goal. Another function used by the algorithm is the Generalize function, which is calculated according to Algorithm 3 and will be described shortly. Next, a set red_g is initialized (line 10) to keep track of redundant goals. The algorithm then performs a pairwise comparison of the goals (lines 11–13), marking a goal as redundant if its generalized form appears as a subterm within another goal using the function prodBody (line 13). The definition below, provides a more formal description of the prodBody function.

▶ **Definition 4** (prodBody Function). *Given Gallina terms t and t', prodBody(t, t') returns true if t is a product type, aka, universally quantified term, with body t'', i.e., $t =$ `forall` $(x : t_0)$, t'' for some term t_0, such that either $t' = t''$ or prodBody(t'', t') is true.*

Finally, the redundant goals are removed from g_list (line 18), ensuring that only distinct, non-nested goals remain.

The Generalize function transforms a given goal into a universally quantified form by identifying and universally quantifying all free variables in the goal that are present in the context (ctx). Algorithm 3 defines this function. This function first collects the relevant free variables (fvs) and determines their dependency order using topo_sort (Line 2), ensuring that variables are introduced in a correct dependency-aware sequence. The topological sorting in reverse order ensures that if variable v1 depends on v2, then v2 is quantified before v1, preventing type dependency issues. The function then iterates through the reverse of this topological order (Line 2) and prepends universal quantifiers (`forall`) to the goal in the correct order, making sure all dependent variables appear before their use. This process ensures that dependently typed terms remain well-formed while abstracting away unnecessary local dependencies.

The function topo_sort constructs a topological ordering [10] of variables in ctx that are also present in fvs. It first builds a directed acyclic graph wherein nodes are variables and edges represent dependency relationships between the variables. More precisely, assume that we have $v_1 : t_1$ and $v_2 : t_2$ in local context, where t_1 is a Gallina term such that $v_2 \in FV(t_1)$. Intuitively, this means that v_1 needs v_2 in order for it to be defined, because its type uses v_2, so we create a directed edge from the node corresponding to v_1 to that of v_2 to denote this dependency. The function then conducts a topological sorting on this graph by conducting a depth-first search on each node. The output of topo_sort function is a list of variables that if reversed will present the variables in a valid dependency order. So, if v_1 depends on v_2 in the graph, we will universally quantify v_2 before v_1, thereby creating a well-formed dependently typed term.

Algorithm 4 Reporting clones.

Require: goals_list: List of deduplicated and generalized goals, *i.e.*, Algorithm 2's output
Ensure: List of alpha equivalent goals
1: output $\leftarrow \emptyset$
2: **for** $i = 0$ to $|\text{goals_list}| - 1$ **do**
3: **for** $j = i + 1$ to $|\text{goals_list}| - 1$ **do**
4: **if** goals_list$[i][1]$ $=_\alpha$ goals_list$[j][1]$ **then**
5: output.append(goals_list$[i]$ $\cdot$ goals_list$[j]$)
6: ▷ Concatenate goals_list[i] and goals_list[j] to generate output pairs
7: **end if**
8: **end for**
9: **end for**
10: **return** output

The last step in clone-finder's pipeline involves finding pairs of generalized goals that are α-equivalent. This process is formalized in Algorithm 4. The algorithm simply picks pairs of tuples from the output of Algorithm 2 and checks if the compartments corresponding to the generalized goals are α-equivalent. If so, the tuples, that contain various information, such as the containing theorem name, file name, and proofs, are concatenated to form the output of clone-finder.

4 Implementation

We have implemented clone-finder in 4,969 lines of Python code. As we have given high-level explanation in §3, clone-finder interacts with Coq-LSP to open the Rocq files in a project and execute them step by step, extracting goals, as well as local context entries. In this implementation of clone-finder, for the sake of efficiency, we only consider local contexts, which could introduce unsoundness and false positives. We have implemented a Gallina parser to parse the goals and find α-equivalent goals, as outlined in §3. Currently, clone-finder offers a command-line interface though which it receives target project's base directory, Rocq command-line options, *i.e.*, `-R/-Q` for mapping physical paths to logical paths, and the minimum size of the proofs. clone-finder interfaces Coq-LSP through a recent Python framework, named CoqPyt [7]. We observed that CoqPy does not allow passing `-R/-Q` options to Coq-LSP, which is essential for running real-world projects that we have studied in this paper. Therefore, we extended the API provided by the framework so that they receive `-R/-Q` options and pass them to Coq-LSP. We shall publish this open-source contribution.

Minimum size of proofs parameter of clone-finder dictates the tool to ignore clones with proofs under a certain number of lines of Ltac code. This parameter defaults to 5, *i.e.*, only clones with proofs equal to 5 lines of Ltac commands or more will be reported. In §5, we study clone-finder's performance with this default value. Studying clone-finder with different parameter sizes is the subject of a future extension of this work. We would also like to emphasize that clone-finder could be implemented using Coq-Elpi [8]. Our Python implementation of Gallina parser calls for disabling the notations that might have been used in the proof scripts. Specifically, in all our experiments, we invoked clone-finder by first prepending each proof script with the directive `Set Printing All` to disable the notations. This is essential for our parser to work. Despite this limitation, we opted for using Python and directly parsing Gallina expressions, as we believe that this approach would make this infrastructure available for a wider research community and foster research in automated theorem proving.

5 Experiments

In this section, we report our experimental results for running clone-finder on a number of real-world Rocq projects. All our experiments are conducted on a Dell workstation with a 48-core Intel Xeon Gold CPU @ 2.3 GHz and 512 GB of RAM, running Ubuntu 22.04.5 LTS.

5.1 Benchmark Projects

Our benchmark suite consists of real-world Rocq projects from the CoqGym dataset [42], which is a widely used dataset in the literature [42, 38, 37, 16, 15, 14]. CoqGym comprises 135 Rocq projects obtained from GitHub. We observed that the projects were compatible with 5 different versions of Rocq. Older projects were compatible with Coq version 8.5, 8.9.0, or 8.12.0. 46 projects were compatible with version 8.16.0, and 36 projects with version 8.20.0, *i.e.*, the latest version of Rocq as of conducting our experiments. Coq 8.16.0 is the first version of Rocq that includes Coq-LSP, so we had to use version 8.16.0 or newer. We chose Coq 8.16.0, because it was compatible with most number of projects. Unfortunately, the older version of Coq-LSP, shipped with Coq 8.16.0, occasionally crashed with a "segmentation fault" error, when we used it through CoqPyt, so we ended up with 40 working Rocq projects.

Columns "Project Name" and "Project Size" of Table 1 list all the projects in our benchmark and their sizes. We have reported the project sizes both in terms of number of `.v` files in the project (column "#Files") and total lines of code (column "LoC"). Our benchmark contains a diverse set of Rocq projects, representative of the real-world Rocq projects. The projects are of varying sizes from 53 to 29,260 lines of code, and they are from various domains: from set theory to distributed system verification to geometry.

5.2 Results

We ran clone-finder on each of our benchmark projects. The last four columns of Table 1 report our experimental results in terms of number clones, *i.e.*, the number of α-equivalent goal pairs with minimum proof size of 5 lines of Ltac code, as well as the time, in seconds. We make three time measurements: (1) under the column "Build," we report the time for running Rocq on all the project files to build and generate ".vo" files; (2) under the column "clone-finder Total," we report the end-to-end time for running clone-finder, including the time Rocq needs to load and type-check the files; and (2) under the column "Analysis," we report the time clone-finder needs to parse the goals, generalize and deduplicate them, and find α-equivalent ones. Studying clones with different proof sizes is left as a future work.

In 20 out of 40 projects, clone-finder finds at least one clone. The average number of goal clones per project is 27.73 (with median being 0.5). Larger projects tend to have more instances of redundancy in them. By inspecting the clones and their proofs, we observed 3 major types of proofs: (1) identical or near identical proofs, where the proofs are almost the same, except the variable names have changed; (2) generalized proofs, where one proof proves a more general form of the goal; (3) entirely different proofs, where new proofs have been invented in each case the goal is observed.

Figure 3 shows an example of a pair of α-equivalent goals and their identical proofs. We do not have information about how the authors have written these proofs. Specifically, we are not sure whether the author have copied and pasted the proofs or they have rewritten the proofs in each case. Either way, we believe that this kind of redundancy is the least expensive one, as no new theorem or a generalization thereof needs to be created.

Table 1 Benchmark of Rocq projects and the result of applying clone-finder on each project.

Project Name	Project Size		# Clones	Time (s)		
	# Files	LoC		Build	clone-finder Total	Analysis
AlmostFull	11	3,625	102	8.563	1962.643	27.661
Angles	6	3,633	25	4.356	79.379	50.821
Autosubst	8	1,553	0	3.484	104.820	0.511
bbv	16	9,027	6	13.460	3137.301	93.807
CDF	13	8,558	33	14.127	3502.113	21.409
Chapar	10	20,371	447	303.367	2896.301	223.292
Checker	2	141	0	0.950	11.309	0.016
ConstructiveGeometry	7	1,453	0	2.123	21.533	0.204
Coqoban	2	6,398	0	2.286	16.486	0.947
DepMap	6	1,259	0	6.701	1538.239	5.151
DomainTheory	6	1,223	1	3.472	96.321	5.981
ExtLib	118	10,304	3	29.696	2860.590	9.163
FreeGroups	1	654	0	1.166	71.283	0.785
FunctionsInZFC	1	6,294	0	1.414	18.383	0.001
Game	15	1,345	1	2.950	101.722	1.158
Groups	1	306	0	0.850	8.788	0.089
GroupTheory	12	1,661	2	3.793	843.374	1.891
Hedges	1	3,265	8	2.775	164.673	32.239
HighSchoolGeometry	72	29,260	218	185.082	29773.385	1062.286
Huffman	22	6,791	9	15.160	3112.586	29.449
IZF	9	2,419	3	2.207	23.045	1.169
JML	30	20,009	27	23.594	5292.184	48.636
Lambda	12	2,122	0	5.559	1374.476	1.899
LibHyps	8	2,957	0	2.013	1111.587	0.000
mathcomp.bigenough	1	122	0	1.272	185.391	0.000
mathcomp.finmap	2	5,029	0	19.181	1204.847	3.433
mathcomp.zify	4	1,583	0	23.888	3616.383	0.025
MiniML	1	1,425	3	1.958	24.486	6.717
NotationGallery	1	53	0	0.120	93.005	0.000
OtwayRees	21	1,401	16	5.850	119.969	8.015
parseque	14	833	0	4.811	749.475	0.000
Ramsey	1	172	0	0.603	50.528	0.000
RecordUpdate	3	131	0	0.947	11.400	0.000
RSA	5	1,654	0	3.991	1010.601	13.068
Schroeder	5	640	0	1.503	76.689	0.573
Stalmarck	38	14,108	78	30.861	7819.962	134.503
Subst	20	7,506	82	11.001	132.391	9.668
WeakUpTo	10	2,334	7	3.433	26.097	2.284
ZFC	11	4,336	38	4.538	47.417	15.437
ZF	15	7,977	0	5.015	42.641	0.000

Figure 4 shows an example of the second type of redundancy, wherein one goal is in fact a generalization of the other. In the first case, the local variable `eH` is bound to a specific value of type `U`, while in the second case, the author proves the same theorem for all values of type `U`. In such cases, it would be possible to factor out the more general goal as an independent lemma, which could also be useful for future developments of the system.

```
(*Goal 1:*) step_star (init ?x) ?x0 ?x1 /\ In ?x2 (messages ?x1)
(*Proof 1:*)
    split_all.
    eassumption.
    subv s1.
    apply in_app_iff.
    right.
    apply in_eq.

(*Goal 2:*) step_star (init ?x) ?x0 ?x1 /\ In ?x2 (messages ?x1)
(*Proof 2:*)
    split_all.
    eassumption.
    subv s1.
    apply in_app_iff.
    right.
    apply in_eq.
```

Figure 3 Duplicate proofs example: Example of two goals from Rocq project Chapar where α-equivalent goals show up in two different theorems `KVSAlg1CauseObl.algrec_step` and `KVSAlg1CauseObl.cause_rec` and identical proofs have been used.

Lastly, Figure 5 shows an example of the third kind of redundancy, *i.e.*, entirely different proofs have been proposed for α-equivalent goals. This could have educational value, *e.g.*, proving same goals in different ways, or the authors have proved these goals at different times, or different people have proved the goals, and a refactoring opportunity was missed.

Another observation that we can make from Table 1 concerns the time measurements. We observe that, on average, it takes 1,833.345 seconds for clone-finder to analyze a Rocq project in our benchmark to find clones. We further observed that 94.71% of this end-to-end time is from using CoqPyt/Coq-LSP to execute the Rocq files step by step, as the average time required to find clones after type checking the files is only 45.31 seconds (with median being 2.091 seconds). This means that, compared to normal build (normal build time is reported under "Build" column in Table 1), on average, there is a 122.4X increase in run time. This significant overhead is because CoqPyt loads Rocq script files individually, rather than loading the entire project as a batch, which calls for loading and checking all the dependencies of the script file each time we load Rocq script. We are still investigating if this issue can be resolved by only modifying CoqPyt so that its `ProofFile` class can accept a Rocq project to load, and cache the common dependencies between the files in the project and avoid loading and checking them from scratch, or if a modification to Coq-LSP is also needed. Another way to mitigate this overhead would be through incorporating clone-finder as a Rocq plugin that directly retrieves the AST for goals and contexts from Rocq platform itself. We save these engineering enhancements as a future extension of this work. The current version of clone-finder tool stores partial results in file and avoids invoking Coq-LSP whenever the cached results are already available.

6 Threats to Validity

Like all empirical studies, our findings are subject to threats to validity. In this section we discuss some of these threats and how we mitigate them.

We had to limit our experiments to the set of Rocq projects that were compatible with Coq 8.16.0, which slashed the size of our benchmark by 86 projects. Additionally, the Coq-LSP version shipped with Coq 8.16.0 occasionally crashed, leading to a reduced set of 40 working Rocq projects out of the 46 projects compatible with Coq 8.16.0. Despite these cuts in the size of the benchmark, the Rocq projects are of varying sizes and domains, giving us some level of confidence about their representativeness. Nevertheless, this reduction could introduce bias by excluding certain types of projects or proofs from the analysis, potentially affecting the representativeness of the results. To mitigate this threat, we will make clone-finder open-source so that research community can study it on larger available Rocq dataset.

```
(*Goal 1:*) In U H eH -> Setsubgroup U H Gr
(*Proof 1:*)
    intro H'.
    apply T_1_6_2 with (witness := witness); auto with sets.
    red in |- *; intros a b H'0 H'1; try assumption.
    lapply (h2 a (inv b)); [ intro H'5; lapply H'5; [ intro H'6; generalize H'6; clear H'5 | clear H'5 ] | idtac ];
    auto with sets.
    rewrite <- (inv_involution' b); auto with sets.
    red in |- *; intros a H'0; try assumption.
    rewrite <- (G2b' (inv a)).
    apply h2; auto with sets.
    rewrite <- (G3b' witness); auto with sets.

(*Goal 2:*) forall x : U, In U H x -> Setsubgroup U H Gr
(*Proof 2:*)
    intros witness inH.
    apply T_1_6_2 with (witness := witness); trivial.
    red in |- *.
    intros a H'.
    cut (exists n : nat, inv a = phi a n).
    intro H'0; elim H'0; intros n E; rewrite E; clear H'0.
    apply positive_powers; trivial.
    cut (inv a = phi a m).
    intro H'0; rewrite H'0.
    exists m; trivial.
    symmetry in |- *.
    apply powers_repeat with (n := r); trivial.
    apply H_included_in_G; auto.
```

Figure 4 More general proof reinvented: Example of two goals from Rocq project GroupTheory where α-equivalent goals show up in two different theorems `T_1_6_3` and `T_1_6_4`. The second proof is a more general form of the first one, and the proofs seem to be reinvented for each goal.

clone-finder looks for α-equivalent goals in Rocq projects. In other words, in this paper, we only consider syntactic clones. Our results are not generalizable to the other major type of clones. Specifically, the projects in our benchmark could contain semantic clones, *e.g.*, closely related goals that can be generalized into a single more general goal, or even universally quantified goal pairs with non-dependent quantified variables that are different only in the order of their quantified variables. We save this interesting topic for a future work (see §8).

The basic idea behind Algorithm 1 is inspired from [42]. Similar algorithm has been used in other works as well [32, 35]. This style of constructing proof trees has the drawback of reporting inaccurate results when facing with compound tactics, tactics that resolves more than one goal, and focus-shifting tactics. Some authors [35] have explicitly mentioned that such commands would break the algorithm, while others [42] have desugared or otherwise filtered out the proofs containing such tactics. In this work, we opted for not filtering out proofs, as in our application, missing a few clone cases or reporting inaccurate proof boundaries for a goal is not harmful.

Last but not least, the primary metrics used to evaluate clone-finder are the number of clones detected and the time taken for analysis. While these metrics provide significant insights into the performance of the proposed technique, they may not fully capture the practical significance or impact of the detected clones. Additional metrics, such as the effort required to refactor or eliminate the clones will be studied in a future work.

7 Related Work

The works by Ghanouchi [18] and Vinan and Hupel [40] that introduce two approaches for clone detection in Isabelle/HOL theories are closest to clone-finder. Ghanouchi [18] applies fuzzy token matching to detect clones in Isabelle/HOL theory files. This technique extends previous token-based approaches by considering near-miss clones through similarity-based token matching. Vinan and Hupel [40], on the other hand, adapt the ConQAT framework, a well-known clone detection tool, to detect proof clones in Isabelle/HOL theories. This technique extracts structured information from Isabelle document markup and performs syntactic clone detection using ConQAT's text-based similarity analysis. Unlike these two techniques, clone-finder is not intended to find clones in proof scripts themselves. Instead,

```
(*Goal 1:*) orthogonal (vec H O) (vec H C)
(*Proof 1:*)
    apply ortho_sym.
    rewrite H11.
    VReplace (vec H A) (mult_PP (-1) (vec A H)).
    rewrite H12.
    VReplace (mult_PP k (mult_PP (-1) (mult_PP k' (vec A B)))) (mult_PP (- (k * k')) (vec A B)).
    auto with geo.

(*Goal 2:*) orthogonal (vec H O) (vec H A)
(*Proof 2:*)
    apply ortho_sym.
    replace (vec H A) with (mult_PP (- k) (vec A B)).
    auto with geo.
    VReplace (vec H A) (mult_PP (-1) (vec A H)).
    rewrite H8; Ringvec.
```

Figure 5 Reinvented proofs example: Example of two goals from Rocq project HighschoolGeometry where α-equivalent goals show up in two different theorems `intersection_cercle_droite` and `intersection2_cercle_droite`, different proofs seem to be reinvented.

it is designed to detect goals that could be addressed once and for all. In other words clone-finder leverages notions of α-equivalence of goals, and proof irrelevance [4], to detect redundant proof efforts. One major benefit of clone-finder's clone detection approach, driven by α-equivalence notion, is that it can be sound and complete, but the aforementioned similarity-based heuristics may result in false positives/negatives.

Goal clone detection aligns with *proof reuse* in proof engineering. "Large-scale proof development may involve redundant efforts that can be time-consuming. Proof reuse addresses this by repurposing existing proofs as much as possible, minimizing the amount of redundant work that proof engineers must do" [33]. Existing proof reuse techniques mainly focus on proof generalization [11, 19, 5, 3, 34, 21]. To the best of our knowledge, clone-finder is the first technique on the topic of goal clone detection that realizes proof reuse by identifying α-equivalent goals that can be extracted as independent lemmas and reused. As such lemma extraction tools for Rocq, *e.g.*, company-coq [32] and CoqPIE [35], can be used alongside clone-finder.

Finding α-equivalent goals in Rocq proofs is directly related to the software engineering problem Type 1, *i.e.*, exact clones, and Type 2, rename clones, clone detection. There is a rich body of knowledge developed for clone detection for various procedural programming languages [28, 36, 29, 41] and even Java bytecode [25, 43]. Being language dependent, none of these techniques can be directly applied to goal clone detection. Additionally, functional programming defines α-equivalence, which is a theoretically sound notion of syntactic clones, *i.e.*, Type 1 and Type 2 combined, that can be leveraged in the context of functional programming, in general, and Rocq, in particular, without resorting to potentially inaccurate heuristics proposed in the past.

8 Conclusion and Future Work

We design, implement, and evaluate a technique and a tool, named clone-finder, for detecting goal clones, *i.e.*, instances of α-equivalent goals that are proved multiple times within a Rocq project. Our evaluation of clone-finder on 40 real-world Rocq projects reveals that half of the projects (20 out of 40) contained at least one goal clone. The detected redundancies fell into three primary categories: (1) exact goal duplication with identical or near-identical proofs, *e.g.*, structurally identical proofs with different variable names; (2) α-equivalent goals with generalized proofs, where one proof subsumes the other; and (3) α-equivalent goals with entirely different proofs. These findings highlight a significant opportunity for improving proof maintainability and reusability in large-scale Rocq developments.

Performance analysis of clone-finder provided empirical evidence that it is a lightweight system, with an average runtime of 45.31 seconds per project, making it a practical tool for daily proof engineering tasks. Additionally, in the process of implementing clone-finder, we contributed an enhancement to CoqPyt, allowing for handling of physical-to-logical path mappings in Rocq projects.

Our results suggest that redundancy in proof engineering is a widespread issue and that automation could help mitigate wasted effort. Moving forward, we envision extending clone-finder to detect semantic clones with the help of large language models, integrating it directly as a Rocq plugin for improved efficiency, and exploring automated lemma extraction to further reduce redundant proof efforts. We are also planning to incorporate an η-equivalence detection mechanism in clone-finder.

References

1 Eyad Alkassar, Mark A. Hillebrand, Dirk C. Leinenbach, Norbert W. Schirmer, Artem Starostin, and Alexandra Tsyban. Balancing the load. *Journal of Automated Reasoning*, 42(2):389–454, April 2009. doi:10.1007/s10817-009-9123-z.

2 Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. Concert: a smart contract certification framework in coq. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 215–228. ACM, 2020. doi:10.1145/3372885.3373829.

3 Gilles Barthe and Olivier Pons. Type isomorphisms and proof reuse in dependent type theory. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2030 of *Lecture Notes in Computer Science*, pages 57–71. Springer, 2001. doi:10.1007/3-540-45315-6_4.

4 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. doi:10.1007/978-3-662-07964-5.

5 Alex J. Best. Automatically generalizing theorems using typeclasses (short paper). In Jasmin Blanchette, James H. Davenport, Peter Koepke, Michael Kohlhase, Andrea Kohlhase, Adam Naumowicz, Dennis Müller, Yasmine Sharoda, and Claudio Sacerdoti Coen, editors, *Joint Proceedings of the FMM, FVPS, MathUI,NatFoM, and OpenMath Workshops, Doctoral Program, and Work in Progress at the Conference on Intelligent Computer Mathematics 2021 co-located with the 14th Conference on Intelligent Computer Mathematics (CICM 2021), Virtual Event, Timisoara, Romania, July 26 - 31, 2021*, volume 3377 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2021. URL: https://ceur-ws.org/Vol-3377/fmm12.pdf.

6 Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski. Challenges and experiences in managing large-scale proofs. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings*, volume 7362 of *Lecture Notes in Computer Science*, pages 32–48. Springer, 2012. doi:10.1007/978-3-642-31374-5_3.

7 Pedro Carrott, Nuno Saavedra, Kyle Thompson, Sorin Lerner, João F. Ferreira, and Emily First. Coqpyt: Proof navigation in python in the era of llms. In Marcelo d'Amorim, editor, *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, pages 637–641. ACM, 2024. doi:10.1145/3663529.3663814.

8 Coq-Elpi Team. Coq-Elpi: Coq plugin embedding elpi, 2025. Accessed: 04/22. URL: `https://github.com/LPCIC/coq-elpi`.

9 Coq-LSP Team. Coq-LSP: Visual Studio Code Extension and Language Server Protocol for Rocq / Coq, 2025. Accessed: 04/22. URL: `https://github.com/ejgallego/coq-lsp`.

10 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

11 Régis Curien. *Outils pour la preuve par analogie*. Theses, Université Henri Poincaré - Nancy 1, January 1995. URL: `https://hal.univ-lorraine.fr/tel-01748604`.

12 David Delahaye. A tactic language for the system coq. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000. `doi:10.1007/3-540-44404-1_7`.

13 Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic: With proofs, without compromises. *ACM SIGOPS Oper. Syst. Rev.*, 54(1):23–30, 2020. `doi:10.1145/3421473.3421477`.

14 Emily First and Yuriy Brun. Diversity-driven automated formal verification. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1–13. ACM, 2022. `doi:10.1145/3510003.3510138`.

15 Emily First, Yuriy Brun, and Arjun Guha. Tactok: semantics-aware proof synthesis. *Proc. ACM Program. Lang.*, 4(OOPSLA):231:1–231:31, 2020. `doi:10.1145/3428299`.

16 Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In Satish Chandra, Kelly Blincoe, and Paolo Tonella, editors, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1229–1241. ACM, 2023. `doi:10.1145/3611643.3616243`.

17 Ali Ghanbari. Automatic Goal Clone Detection in Rocq, 2025. Accessed: 04/22. URL: `https://github.com/ali-ghanbari/clone-finder`.

18 Seifeddine Ghanouchi. Code clone detection in isabelle using token stream similarity. Technical report, Technical University of Munich, 2023. Bachelor's Thesis.

19 Robert W Hasker and Uday S Reddy. Generalization at higher types. In *Proceedings of the Workshop on the λProlog Programming Language*, pages 257–271, 1992.

20 Inria. The coq proof assistant documentation, 2025. Accessed: 04/22. URL: `https://rocq-prover.org/doc/V9.0.0/refman/index.html`.

21 Einar Broch Johnsen and Christoph Lüth. Theorem reuse by proof term transformation. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics, 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004, Proceedings*, volume 3223 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2004. `doi:10.1007/978-3-540-30142-4_12`.

22 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018. `doi:10.1017/S0956796818000151`.

23 Elmar Jürgens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 485–495. IEEE, 2009. `doi:10.1109/ICSE.2009.5070547`.

24 Cory Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empir. Softw. Eng.*, 13(6):645–692, 2008. `doi:10.1007/S10664-008-9076-6`.

25 Iman Keivanloo, Chanchal K. Roy, and Juergen Rilling. Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning. In James R. Cordy, Katsuro Inoue, Rainer Koschke, Jens Krinke, and Chanchal K. Roy, editors, *Proceeding of the 6th International Workshop on Software Clones, IWSC 2012, Zurich, Switzerland, June 4, 2012*, pages 36–42. IEEE Computer Society, 2012. `doi:10.1109/IWSC.2012.6227864`.

26 Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010. doi:10.1145/1743546.1743574.

27 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.

28 Micheline Bénédicte Moumoula, Abdoul Kader Kaboré, Jacques Klein, and Tegawendé F. Bissyandé. Large language models for cross-language code clone detection. *CoRR*, abs/2408.04430, 2024. doi:10.48550/arXiv.2408.04430.

29 Morteza Zakeri Nasrabadi, Saeed Parsa, Mohammad Ramezani, Chanchal Roy, and Masoud Ekhtiarzadeh. A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. *J. Syst. Softw.*, 204:111796, 2023. doi:10.1016/J.JSS.2023.111796.

30 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-45949-9.

31 Christine Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993. doi:10.1007/BFB0037116.

32 Clément Pit-Claudel and Pierre Courtieu. Company-coq: Taking proof general one step closer to a real ide. In *CoqPL'16: The Second International Workshop on Coq for PL*, January 2016. doi:10.5281/zenodo.44331.

33 Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. QED at large: A survey of engineering of formally verified software. *CoRR*, abs/2003.06458, 2020. arXiv:2003.06458.

34 Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Ornaments for proof reuse in coq. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 26:1–26:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.ITP.2019.26.

35 Kenneth Roe and Scott F. Smith. Coqpie: An IDE aimed at improving proof development productivity - (rough diamond). In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, volume 9807 of *Lecture Notes in Computer Science*, pages 491–499. Springer, 2016. doi:10.1007/978-3-319-43144-4_32.

36 Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of computing TR*, 541(115):64–68, 2007.

37 Alex Sanchez-Stern, Yousef Alhessi, Lawrence K. Saul, and Sorin Lerner. Generating correctness proofs with neural networks. In Koushik Sen and Mayur Naik, editors, *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2020, London, UK, June 15, 2020*, pages 1–10. ACM, 2020. doi:10.1145/3394450.3397466.

38 Alex Sanchez-Stern, Emily First, Timothy Zhou, Zhanna Kaufman, Yuriy Brun, and Talia Ringer. Passport: Improving automated formal verification using identifiers. *ACM Trans. Program. Lang. Syst.*, 45(2):12:1–12:30, 2023. doi:10.1145/3593374.

39 Paula Severi and Erik Poll. Pure type systems with definitions. In Anil Nerode and Yuri V. Matiyasevich, editors, *Logical Foundations of Computer Science, Third International Symposium, LFCS'94, St. Petersburg, Russia, July 11-14, 1994, Proceedings*, volume 813 of *Lecture Notes in Computer Science*, pages 316–328. Springer, 1994. doi:10.1007/3-540-58140-5_30.

40 Dominik Vinan and Lars Hupel. Clone detection in isabelle theories. Technical report, Technical University of Munich, 2016.

41 Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 87–98. ACM, 2016. doi:10.1145/2970276.2970326.

42 Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 6984–6994. PMLR, 2019. URL: http://proceedings.mlr.press/v97/yang19a.html.

43 Dongjin Yu, Jie Wang, Qing Wu, Jiazha Yang, Jiaojiao Wang, Wei Yang, and Wei Yan. Detecting java code clones with multi-granularities based on bytecode. In Sorel Reisman, Sheikh Iqbal Ahamed, Claudio Demartini, Thomas M. Conte, Ling Liu, William R. Claycomb, Motonori Nakamura, Edmundo Tovar, Stelvio Cimato, Chung-Horng Lung, Hiroki Takakura, Ji-Jiang Yang, Toyokazu Akiyama, Zhiyong Zhang, and Md. Kamrul Hasan, editors, *41st IEEE Annual Computer Software and Applications Conference, COMPSAC 2017, Turin, Italy, July 4-8, 2017. Volume 1*, pages 317–326. IEEE Computer Society, 2017. doi:10.1109/COMPSAC.2017.104.

FuzzFlesh: Randomised Testing of Decompilers via Control Flow Graph-Based Program Generation

Amber Gorzynski
Imperial College London, UK

Alastair F. Donaldson
Imperial College London, UK

Abstract

Decompilation is the process of translating compiled code into high-level code. Control flow recovery is a challenging part of the process. "Misdecompilations" can occur, whereby the decompiled code does not accurately represent the semantics of the compiled code, despite it being syntactically valid. This is problematic because it can mislead users who are trying to reason about the program.

We present CFG-based program generation: a novel approach to randomised testing that aims to improve the control flow recovery of decompilers. CFG-based program generation involves randomly generating control flow graphs (CFGs) and paths through each graph. Inspired by prior work in the domain of GPU computing, (CFG, path) pairs are "fleshed" into test programs. Each program is decompiled and recompiled. The test oracle verifies whether the actual runtime path through the graph matches the expected path. Any difference in the execution paths after recompilation indicates a possible misdecompilation. A key benefit of this approach is that it is largely independent of the source and target languages in question because it is focused on control flow. The approach is therefore applicable to numerous decompilation settings. The trade-off resulting from the focus on control flow is that misdecompilation bugs that do not relate to control flow (e.g. bugs that involve specific arithmetic operations) are out of scope.

We have implemented this approach in FuzzFlesh, an open-source randomised testing tool. FuzzFlesh can be easily configured to target a variety of low-level languages and decompiler toolchains because most of the CFG and path generation process is language-independent. At present, FuzzFlesh supports testing decompilation of Java bytecode, .NET assembly and x86 machine code. In addition to program generation, FuzzFlesh also includes an automated test-case reducer that operates on the CFG rather than the low-level program, which means that it can be applied to any of the target languages.

We present a large experimental campaign applying FuzzFlesh to a variety of decompilers, leading to the discovery of 12 previously-unknown bugs across two language formats, six of which have been fixed. We present experiments comparing our generic FuzzFlesh tool to two state-of-the-art decompiler testing tools targeted at specific languages. As expected, the coverage our generic FuzzFlesh tool achieves on a given decompiler is lower than the coverage achieved by a tool specifically designed for the input format of that decompiler. However, due to its focus on control flow, FuzzFlesh is able to cover sections of control flow recovery code that the targeted tools cannot reach, and identify control flow related bugs that the targeted tools miss.

2012 ACM Subject Classification Security and privacy → Software reverse engineering; Software and its engineering → Software testing and debugging; Software and its engineering → Compilers

Keywords and phrases Decompiler, Reverse Engineering, Control Flow, Software Testing, Fuzzing

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.13

Funding This work was supported by EPSRC grant EP/R006865/1 and gift funding from Google.

1 Introduction

In this work we present a new randomised testing approach for automatically finding *misdecompilation* bugs in decompilers.

© Amber Gorzynski and Alastair F. Donaldson;
licensed under Creative Commons License CC-BY 4.0

Decompilers aim to reconstruct source code corresponding to compiled code. This can be useful for detecting security vulnerabilities in binaries [1, 17, 54, 28, 60, 48], source code recovery [18], or to enable program analysis in cases where the source code is unavailable [45, 55, 52, 20, 24].

Decompilation is a difficult process because (a) some information is lost during compilation and (b) restoring high-level code structure is technically difficult, particularly when the code has been heavily altered through compiler optimisations. Control flow recovery is especially challenging because there are often multiple valid ways to structure a given control flow graph (CFG) using high-level constructs (such as `while` or `if/else`) and the process of deciding which structure to use can be a source of errors. In addition, attempts by the decompiler to make the output code more readable (for example by eliminating as many `goto` statements as possible) rely on complex algorithms [6, 65, 25, 4].

In this paper, we define *misdecompilation* as any semantic deviation between the original low-level code and its decompiled counterpart. Misdecompilations mislead analysts, wasting their time. They are more serious than decompiler crashes or failures to decompile because there is no indication to the user that the output is incorrect. The consequences of misdecompilations can be severe: misunderstandings arising from a misdecompilation could cause a security analyst to overlook a vulnerability or mis-classify a piece of malware. For example, Votipka et al. [59] find that malware engineers prefer working with decompiled code for readability, but must switch to (harder-to-read) disassembly when they suspect a misdecompilation has occurred. Source code recovery can also be complicated by the presence of misdecompilation bugs [18].

In recognition of the importance of reliable decompilers, recent work has used randomised testing to search for decompiler bugs in a proactive manner [41, 71, 43]. However, these only target one language format each and do not focus specifically on control flow.

We build on Klimis et al. [34] to develop a novel decompiler fuzzing approach called CFG-based program generation. It has two key features:

CFG-based program generation targets control flow recovery. The approach aims to thoroughly test the control flow recovery components of decompilers. Beyond control flow, our programs are designed to use limited low-level language features, so that the only source of complexity is the control flow. This restriction to a limited set of features is important, because in practice each decompiler has an informal "scope", which is the set of programs it *should* be able to decompile. Programs that make intricate use of specialised low-level language features risk being outside of the scope of a decompiler, which may not support the reconstruction of these features. These programs are not useful as test cases because (a) they will not be recompilable, and therefore cannot detect misdecompilation bugs, and (b) will not be interesting as crash- or failure-inducing test cases to developers, who generally focus on in-scope supported language features. Prior work has devoted considerable attention to addressing the problem of recompilability, for example through post-processing the decompiled output [41, 71]. By producing programs that are syntactically simple, our approach sidesteps this problem.

The trade-off associated with our restriction to the control flow-related parts of low-level languages is that the programs generated by our approach are unlikely to trigger bugs related to detailed language-specific features (such as bugs in the handling of arithmetic operations).

CFG-based program generation is easily retargeted to various low-level languages. All mainstream programming languages have some notion of control flow. CFGs generated by our approach can therefore be "fleshed" into many different target languages, without the

need to provide full grammars to the tool or perform extensive language-specific engineering. We are not aware of any other decompiler testing techniques that can be easily retargeted to work with distinct low-level languages. Prior decompiler testing tools have been highly customised for particular languages, requiring a great deal of expertise to build and typically using existing specialised program generators for the target language [43]. Our approach drastically reduces this manual effort with respect to the testing of control flow recovery.

In more detail, CFG-based program generation involves randomly generating CFGs according to graph generation algorithms. Random paths through each CFG are also generated; these represent the expected runtime execution path. Each (CFG, path) pair is "fleshed" into a program that (a) guarantees that a particular path through the program should be traversed, and (b) checks that this path really *is* followed when the program executes. This check forms the test oracle: each program contains a self-checking assertion that fails if there is any difference between the actual and expected path.

We have implemented this approach in FuzzFlesh, a new open source decompiler testing tool. Figure 1 illustrates how we use FuzzFlesh to test decompilers. First, we **generate** a CFG and path, which we **flesh** into a low-level program (for example a Java bytecode class). We use the decompiler under test to **decompile** the program. We then **recompile** the program. Finally, we **execute** the recompiled program and assert that the actual runtime path matches the expected path. Any differences indicate a possible misdecompilation.

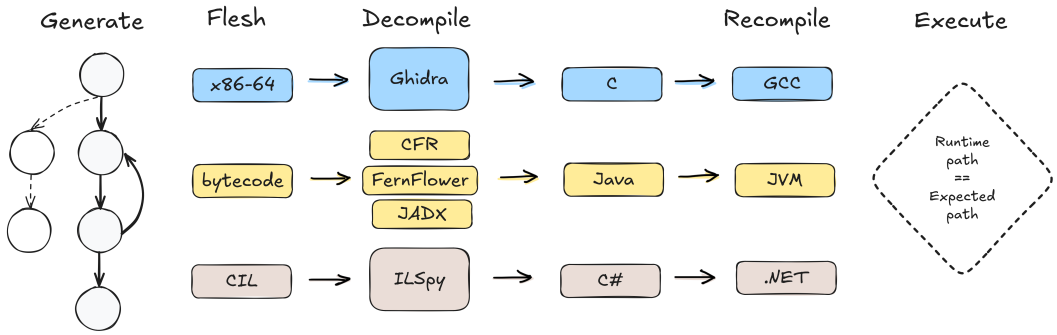


Figure 1 Workflow of CFG-based program generation as implemented by FuzzFlesh.

FuzzFlesh also features a test case reducer that operates on the CFG and path to automatically reduce bug-triggering programs to a manageable size. The reduction process retains the "expected vs. actual path" test oracle, which assists in pinpointing the source of the misdecompilation.

Most of the program generation and reduction toolchain required can be implemented in a language-independent way, which means that it can be easily extended to multiple languages. Based on our experience implementing support for four languages, we estimate that implementing an additional language would only take approximately 8 hours for an experienced programmer (see Section 6.1). This is in contrast to other program generators used by decompiler testing tools, which require extensive language-specific engineering to produce test cases. For example, JavaFuzzer [66] and Csmith [68] are used by other tools and can only produce Java and C programs respectively. Such custom program generation tools require a large amount of effort to produce for each language. FuzzFlesh is therefore particularly useful for quickly testing a range of decompiler toolchains across different languages.

Currently, FuzzFlesh supports the generation of low-level programs in three different formats: Java bytecode, x86-64 machine code, and .NET assembly. We have used FuzzFlesh to test three Java decompilers (CFR, FernFlower, JADX), one binary decompiler (Ghidra), and one C# decompiler (ILSpy). We compare the performance of FuzzFlesh to a binary decompiler testing tool (DecFuzzer) and a Java decompiler testing tool (JD-Tester).

Our evaluation aims to answer the following research questions about CFG-based program generation, in the context of the FuzzFlesh tool:

RQ1 How effective is FuzzFlesh at identifying decompiler bugs?

RQ2 How does the efficacy of FuzzFlesh vary across languages and decompiler toolchains?

RQ3 How thoroughly is FuzzFlesh able to test the control flow recovery components of decompilers in comparison to other decompiler testing approaches?

RQ4 To what extent can FuzzFlesh identify bugs that are different from those identified by other decompiler testing approaches?

Our results show that FuzzFlesh is effective: it has found 12 previously-unknown bugs in four decompilers across two languages, of which eight have been confirmed (with six already fixed) in response to our bug reports. Ten of the bugs are directly related to control flow recovery. Six of the total bugs are misdecompilations, while the remaining six are refusals to decompile, whereby the decompiler refuses to produce a program in the target language due to an internal failure.[1] FuzzFlesh achieves reasonable code coverage of decompilers despite the limited syntax and language features used within its programs. As expected, due to its focus on control flow, the coverage FuzzFlesh achieves on any particular decompiler is lower than that achieved by language-specific decompiler testing tools. Nevertheless, FuzzFlesh is able to cover code that other tools cannot reach. In particular, FuzzFlesh identifies bugs within control flow recovery code that is not covered by other tools.

In summary, our main contributions are:

- A novel program generation technique that specifically targets control flow recovery in decompilers by generating CFG data structures. The programs are designed to be syntactically simple and therefore easily recompilable.
- An implementation of this technique, FuzzFlesh, which comprises: a CFG and path generating component, three different program "fleshing" modules that target x86, Java bytecode, and .NET assembly, a program reducer that operates on the CFG and path data structures, and a test harness for automating the decompiler testing workflow.
- A large-scale testing campaign that identified 12 bugs by applying FuzzFlesh to five decompilers covering three distinct (low, high) language pairs: CFR, FernFlower, JADX, Ghidra, and ILSpy.
- An evaluation of FuzzFlesh featuring a comparison to two other recent decompiler testing approaches, DecFuzzer and JD-Tester, showing that FuzzFlesh offers complementary bug-finding abilities.

Availability. FuzzFlesh is open-source and available on GitHub [21]. An artifact is also published along with this paper [22]. This includes a copy of FuzzFlesh and all scripts and instructions required to reproduce our results.

[1] See Section 2.2 for further detail on decompiler problem categorisation.

2 Background

In this section we explain the necessary prerequisites for understanding the CFG-based program generation approach. Further related work is described in Section 7.

2.1 Decompilation

Decompilers are available for many languages [5, 30, 50, 29]. In general, decompilers are "best effort": they may detect and reject low-level programs that are overly complex. Even when decompilation is attempted, decompilers typically cannot produce code that is syntactically identical to the original source code [16, 64]. Nevertheless, in order to be useful, decompilers must be free from serious errors. In particular, when a decompiler *does* deem a given low-level program in scope for decompilation, the high-level program that the decompiler emits *must* respect the semantics of the low-level program.

Decompilation involves several phases that reverse the compilation process. Implementations vary, but the general approach is as follows [13]. First, the executable binary is lifted to an intermediate representation (IR) using a process similar to disassembly. Next, the CFG, variables, and types are recovered from the IR. Further phases of recovery may take place, notably control flow structure recovery that recovers high-level syntactic control flow structures from the CFG (e.g. `if/else`, `for`, `while` and `do-while` in C-like languages). Other optimisations such as dead code elimination and variable renaming may be performed. Finally, code generation outputs a decompiled program in the source code language.

All phases of decompilation are challenging because information is irretrievably lost during compilation, including variable names (unless debug information has been inserted by the compiler), types, and high-level programming abstractions. Control flow structure recovery is particularly important and has been much researched. Early structuring algorithms attempt to match node configurations against a codified set of high-level schema [6]. However, this technique struggles when it encounters unexpected configurations. As a fallback, a decompiler may handle such cases by using `goto` statements. However, the use of `goto` can lead to code that is difficult to read, and many high-level languages (notably Java) do not feature an unrestricted `goto` statement or similar. Alternative structuring algorithms can be used to attempt to eliminate [65, 25] or minimise [4] the need for `goto` statements, but correctly implementing these algorithms is technically challenging. Control flow recovery becomes difficult when the program's control flow graph features *irreducible* structures [2] – cycles in the graph that have multiple entry points – but the target language does not support arbitrary `goto` statements, because irreducible control flow does not map naturally to structured programming language constructs.

2.2 Decompiler bug categories

Two categories of decompiler problems are unequivocally bugs:

- **Decompiler crash:** The decompiler fails to produce a program in the target language, and crashes, hangs, or otherwise fails ungracefully.
- **Misdecompilation:** The decompiler produces a program in the target language that can be successfully recompiled, but it is semantically inconsistent with the original program.

Two other categories of problems are sometimes, but not always, indicative of bugs:

- **Refusal to decompile:** The decompiler refuses to produce a program in the target language, failing gracefully with an error message. This may be indicative of a bug in the decompiler, but alternatively may be because the input program does not fall within the scope of programs the decompiler aims to handle.

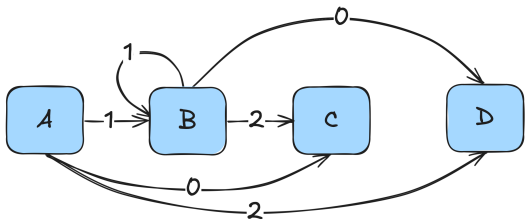


Figure 2 Example CFG generated by FuzzFlesh with directions `[0]` and expected output `[A,C]`.

- **Recompilation failure:** The decompiler produces a program in the target language that cannot be successfully recompiled, due to syntax or typing problems. Whether such problems constitute decompiler bugs depends on whether the decompiler has been designed to produce code that is expected to be directly recompilable.

Our approach aims to find misdecompilation bugs, which are of prime interest because they may mislead users. As a by-product, our approach has the potential to find decompiler crash bugs, which may be frustrating for users but are less serious than misdecompilations. It can also find "refusal to decompile" issues that may be useful to decompiler developers in understanding and assessing the limits of their tools.

While some works address the detection of recompilation failures [71], this is not our focus because many decompilers do not aim to produce code that is directly recompilable. A distinction exists between *semantically correct* and *recompilable* code: the decompiler output may be semantically correct in that it is an accurate representation of the original code, but it may need significant formatting adjustments in order to recompile without errors. For example, Ghidra [50] sometimes excludes function declarations or necessary `#include` statements. This does not pose a problem for the typical user who visually analyses the decompiled output. We therefore do not include this category of failure because it conflates bugs and expected limitations of the decompiler.[2]

3 CFG-based Program Generation and the FuzzFlesh Tool

We now describe our CFG-based program generation approach in detail and explain how we have put it into practice in the FuzzFlesh tool.

We start with an overview of how the approach works using an example graph that was able to identify a misdecompilation bug in CFR [5], a tool that decompiles Java bytecode into Java source code. We then provide further details of each component of the approach.

CFG-based program generation involves randomly generating CFG skeleton data structures and then "fleshing" them into programs in a target low-level language. Figure 2 shows an example CFG that might be generated. Each node represents a basic block in the low-level program that will ultimately be generated, and each edge represents a jump in the control flow between two nodes. The outgoing edges of a node are numbered with consecutive integers starting from 0. A CFG has a single entry point, in this case node A.

A path through the CFG is also randomly chosen, and a corresponding sequence of edge labels is determined; we call this sequence the *directions* that must be taken for the path to be followed. For example, suppose the path $[A, C]$ through the CFG of Figure 2 was chosen. The directions sequence [0] would force this path to be taken.

[2] We do automatically apply minor fixes to programs to support recompilability, which are discussed in Section 4. We do not report these issues as bugs.

```
test_program() {
    int[] directions = {0};
    int[] actual_output;

    int dir_index = 0;
    int out_index = 0;


block_A:
  actual_output[out_index++] = 0;
  switch(directions[dir_index++]) {
    case 0: goto block_C;
    case 1: goto block_B;
    case 2: goto block_D;
  }
```

```
block_B:
  actual_output[out_index++] = 1;
  switch(directions[dir_index++]) {
    case 0: goto block_D;
    case 1: goto block_B;
    case 2 : goto block_C;
  }

block_C:
  actual_output[out_index++] = 2;
  return;

block_D:
  actual_output[out_index++] = 3;
  return;
}
```

Figure 3 Example pseudocode corresponding to the CFG in Figure 2.

From a (CFG, path) pair, (G, p), a concrete low-level program and program input are synthesised, where the program input is the sequence of directions associated with p. The program is designed to (a) consume the directions and follow the corresponding control flow path, and (b) output the sequence of nodes that is traversed during execution. Therefore, on execution, the program should output the path p.

The synthesised low-level program is then decompiled and recompiled, and executed on the input, and a check is performed to confirm that the program output does indeed indicate that the required path was followed. If not, a misdecompilation has occurred and can be investigated.[3] The comparison of the expected and actual path forms the test oracle, which is self-contained within every test case.

Returning to the example of Figure 2, Figure 3 shows C-like pseudocode illustrating how the graph can be fleshed into a program. The actual output is recorded at each node in the `actual_output` array. The `directions` array is read from every time that a conditional jump is encountered. The target language of the flesher depends on the toolchain under test.

When configured to target Java, our FuzzFlesh tool generates a textual Java bytecode program for this example, which is converted to a `class` file using the Jasmin assembler [47]. FuzzFlesh uses CFR to decompile the program, uses `javac` to recompile it, and executes it on the input [0], which should force the path $[A, C]$ to be followed. The actual output indicated that the path $[A, B, C]$ had been followed: the bytecode program had been misdecompiled. We have reported this issue to the CFR developers.

An attractive feature of CFG-based program generation is that it is easy to retarget for various low-level languages. For example, the graph- and path-generation parts of our FuzzFlesh implementation are entirely language-independent. The only language-specific work that is required is the straightforward logic to flesh a (CFG, path) pair into a low-level program. The tool is thereby able to test a range of decompilers for various languages.

In practice, FuzzFlesh generates larger and more complex CFGs than the one shown in Figure 2. Once a potential bug is identified, FuzzFlesh reduces the program to a manageable size by performing a series of operations on the CFG data structure and then re-fleshing it into a (smaller) program.

[3] Is is theoretically possible that the problem could actually be a compiler bug introduced during the recompilation process, but this has not occurred in practice in our experience.

We now provide further detail on how we have realised each component of CFG-based program generation in FuzzFlesh. Section 3.1 and Section 3.2 set out the CFG and path generation approaches. Section 3.3 describes how these CFGs and paths are fleshed into programs. Section 3.4 explains the test oracle used to identify potential misdecompilations. Finally, Section 3.5 describes the CFG-based program reducer.

3.1 Graph generation

The aim of the graph generator is to create a CFG that can be used as the basis for a program. It generates graphs using a set of user-configurable input parameters including a random seed. Each graph is randomly generated from scratch; there is no initial corpus of graphs. The output is a directed graph data structure that contains nodes and directed edges. The generated graphs must be a valid basis for programs, which is achieved by ensuring that they meet a set of conditions. First, there must exist a single entry point to the graph; second, there must exist at least one path from the entry node to an *exit node*: a node with no successors. All target languages (x86-64, Java bytecode, and .NET assembly) allow irreducible control flow (see Section 2.1), which means we do not impose further restrictions on the shape of the graphs, although other target languages that do not support arbitrary control flow (such as WebAssembly) would require additional checks. This issue is discussed further in Section 4.2. Disconnected graphs are valid programs, providing there is only one entry point in the graph. The disconnected region can be considered "dead" code that the decompiler may eliminate.

The graph generator in FuzzFlesh grows a graph of a parameterised target size using a breadth-first algorithm adapted from Krapivsky and Redner [35]. Starting with the root, the generation algorithm adds a random number of successors nodes. It continues to add a random number of successor nodes to each existing node until the target graph size is reached. This constructs a wide tree graph that contains a path from the entry node to at least one exit node, but does not have any jumps beyond immediate child nodes. To enhance the graph shape, we add edges to a randomly chosen number of nodes in the graph. A proportion of these additional edges jump backwards to a direct ancestor of the current node, which introduce cycles into the CFG so that important program constructs such as loops are represented. Finally, additional exit nodes are optionally added to a randomly selected sample of nodes in the graph. This reduces the likelihood that a path will become stuck in a part of the graph that has no naturally-occurring exit nodes due to the back edge annotations.

We also implemented an alternative graph-generation algorithm adapted from Erdős and Rényi [19]. This graph generator creates a set of nodes according to a graph size parameter and randomly connects them via directed edges. A single node is randomly selected as the root node. In addition, exit nodes are randomly added. In practice we found that this method generates graphs with many disconnected components and short paths, and we found it to be ineffective at triggering bugs in practice, so do not discuss it further.

FuzzFlesh has been designed in a modular fashion so that other graph generation approaches would be straightforward to integrate. For example, one could mine existing programs for interesting control flow graphs and using these as the basis for fuzzing; this is an area for future work.

This stage of the toolchain is entirely language-independent.

3.2 Path generation

The FuzzFlesh path generator performs a random traversal of each generated CFG. It records the sequence of nodes visited on the path, and the sequence of directions that will force this path to be followed. Multiple paths can be generated for each graph, and they can be configured by a maximum length input parameter. Once this length is reached, the path generation algorithm identifies the shortest path to an exit node and adds this to the path. FuzzFlesh does not attempt to cover all paths through the CFG. Instead, a user-configurable number of paths through each CFG are selected at random.

This stage of the toolchain is also language-independent: paths are a data structure containing the path and the directions array.

3.3 Program fleshing

At this stage, FuzzFlesh becomes language-specific. The inputs to the flesher component are a graph data structure and a path. The output is a program in a particular low-level language, ready to be decompiled. By construction, the output program will write the IDs of each node visited at runtime to an actual output array. The contents of this output array can be compared with the expected output from the path to provide the test oracle.

Program fleshing involves traversing the CFG and fleshing each node based on its characteristics. As described in Section 3.1, the CFG is constructed as an abstract graph (i.e. not in a low-level language format) and fleshing provides a direct mapping between CFG nodes and appropriate low-level language operations. First, the start of the program is fleshed with the necessary code to begin the program, including the declaration of the function and data structures for the directions and output arrays. Next, the flesher traverses the CFG and fleshes each node with code that records the node ID in the output array so that the actual path taken through the program at runtime is recorded. In addition, the flesher checks the number of children that each node has and fleshes it with an appropriate basic control instruction such as `goto`, `if/else`, or `switch`. Exit nodes with no children contain a return instruction. We do not include more complex language features in order to ensure that the resulting program is simple enough to be decompiled and subsequently recompiled.

During the fleshing stage, the directions array corresponding to the expected path can be configured to be statically unknown or known. This provides respectively less or more information about the program to the decompiler under test.

If the directions are statically unknown, then the directions array is passed to the test case function at runtime as a parameter from the wrapper program. In this case, the test case is fleshed to accept the directions array and read directions from it, but these directions are not known at decompile-time. This creates a program in which every node in the graph is potentially dynamically reachable at runtime, given suitable inputs.

Alternatively, the directions array can be hard-coded into the function during the fleshing step. In this case, the directions are known to the decompiler.

These approaches may result in different decompilation heuristics. The overall impact on the decompiler is unclear: providing more information to the decompiler by hard-coding the directions could make the program easier to decompile. However, as discussed in Section 4.1, in some cases we generate programs in a high-level language such as C and use a compiler to generate the low-level program (rather than directly generating the low-level program). In this setting, statically-known directions can be used to perform optimisations at compile-time. The resulting optimised binary may have a more complicated and less easily recoverable

structure. Alternatively, compiler optimisations may have greatly simplified the program, which could result in more straightforward decompilation. To maximise diversity during testing, we generate the low-level programs using a range of compiler optimisation levels.

3.4 Test oracle evaluation

Each program generated by FuzzFlesh contains a built-in self-checking test oracle in the form of the expected output. A test harness performs the testing process and evaluates the test oracle to determine whether the test has failed. The testing process for decompilers is as follows:

1. **Decompile:** We use the decompiler under test to decompile the program to a target source code. At this stage, the decompiler may crash or fail and throw an exception if it encounters code that it is unable to process.
2. **Recompile:** If the decompilation was able to produce a source program, we recompile the resulting program.
3. **Execute:** Finally, we execute the de- and recompiled program. If the actual output recorded by the program at runtime differs from the expected output, then a misdecompilation bug is flagged for investigation.

3.5 Test case reduction

Once a possible bug is found, FuzzFlesh reduces the bug-triggering test case to a manageable size in order to pinpoint the source of the error. We have designed a custom automated test case reducer to make the simplification and triage of bug-triggering test cases easier. Most of the reduction is done at the language-independent graph and path level, which means that it can be reused across multiple languages. The reducer performs the following types of operation on a (CFG, path) pair:

- Merge nodes: Two nodes n_1 and n_2 are merged together into a single node n' so that the total number of nodes in the graph is reduced by one. Any inwards edge to n_1 or n_2 becomes an inward edge to n', and similarly for any outwards edge. If either of the nodes was on the path, then the path and associated directions are also updated. Node merge operations must ensure that at least one exit node is retained so that the program does not become non-terminating.
- Remove edges: Edges are removed from the graph. If an edge is on the path, then the path and directions are also updated.

After each operation, the (CFG, path) pair are re-fleshed into a program. If compiling, decompiling, and recompiling still trigger the same bug, then the program is saved and reduced further. Otherwise, the updated program is discarded and a different reduction operation is attempted.

4 Decompilers under test

We test three Java decompilers: CFR [5], FernFlower [30], and JADX [32], the popular binary decompiler Ghidra [50], and the C# decompiler ILSpy [29]. We selected these decompilers because they are relatively well established and currently maintained, and they have been reviewed in previous studies [41, 43, 71]. Although it is considered in prior work we did not include the decompiler RetDec because it describes itself as in “limited maintenance mode” and only has one commit in the past six months [3]. We also do not consider the angr decompiler [57] because we found that extensive engineering effort is required to allow outputs from angr to be recompiled.

Table 1 summarises the key characteristics of these decompilers, which we now discuss in more detail.

Table 1 Decompilers under test.

Name	Input	Target	CFG recovery
CFR	Java bytecode	Java	Pattern-matching
FernFlower	Java bytecode	Java	Tarjan's algorithm
JADX	Java bytecode	Java	Pattern-matching
Ghidra	Binary	C	Tarjan's algorithm
ILSpy	.NET assembly	C#	Pattern-matching

4.1 Binary decompilers

Ghidra [50] is an open-source reverse engineering tool that allows users to perform a range of analyses on compiled code. We specifically test the decompiler configuration that decompiles x86-64 machine code to C. Rather than directly generating machine code, FuzzFlesh instead fleshes graphs into C programs, which are then compiled to machine code using gcc with a randomly chosen optimisation level. Ghidra disassembles and decompiles these programs to C code. It uses Tarjan's algorithm for identifying strongly connected graph components [58] for control flow recovery. The output programs can typically be re-compiled and executed. Some minimal adjustments are occasionally required. For example, Ghidra output often includes a function call to `__stack_chk_fail()` to indicate that there may be an issue with the decompilation process, but it does not define this function and so the program fails to recompile. This adjustment is easily automatable: we simply insert a line into the decompiled program: `void __stack_chk_fail(){ return; }`, which allows the program to recompile and execute without errors or warnings. Based on our experience, Ghidra tends to insert such calls frequently even within programs that it otherwise appears to have decompiled correctly, which indicates a degree of over-caution. In the case that the `__stack_chk_fail()` call relates to a genuine problem with the decompiled code, our test oracle should identify the decompilation as incorrect since the program output will be altered.

4.2 Java decompilers

FernFlower [30], CFR [5], and JADX [32] are open-source decompilers that decompile Java bytecode class files to Java source code. CFR and JADX use pattern-matching control flow recovery algorithms that are supplemented by heuristics (for example in the handling of switch statements). FernFlower uses some pattern-matching for code generation but, similarly to Ghidra, also makes use of Tarjan's algorithm (see Section 4.1). It was the first Java decompiler that specifically aimed to be able to deobfuscate code, and it includes modules specifically for restructuring control flow that is not representable in Java.

FuzzFlesh generates test programs in textual Java bytecode, which are converted to binary Java class files using Jasmin [47]. We generate Java bytecode rather than Java because it allows more expressive control flow. Specifically, Java does not allow irreducible control flow [23, Section 2.2.6] (see Section 2.1). However, Java bytecode *does* allow irreducible control flow; the bytecode instruction set [51] includes an unrestricted `goto` command that can be used to transfer control to anywhere in the program. This difference in restrictions allows compilers that generate bytecode to perform optimisations that introduce irreducible control flow.

It is possible that some Java bytecode programs do not have a corresponding representation in Java due to differences in restrictions between the languages. In this case, it is expected that a bytecode decompiler will gracefully refuse to decompile the program. Nevertheless, since decompilers can be used for security purposes to inspect potentially malicious class files, it is desirable for bytecode decompilers to be as capable as possible in decompiling atypical bytecode. At a minimum, the decompiler should provide an informative error message stating which part of the bytecode it was unable to process.

4.3 C# decompilers

ILSpy [29] is an open source .NET decompiler that decompiles programs expressed in the .NET intermediate representation, Common Intermediate Language (CIL) to C#. Unlike the Java decompilers, it does not have to convert irreducible control flow to a reducible graph because C# supports irreducible control flow via a `goto` statement. FuzzFlesh has two modes for targeting CIL: one in which CIL code is generated directly, and another where C# programs are generated, and then compiled into CIL using the .NET C# compiler csc.

C# programs are typically compiled to CIL, which is then JIT-compiled using the .NET runtime on Windows and Linux or the mono runtime on Mac OS. The intermediate CIL programs are not usually optimised ahead of runtime, and there are no options to do so in .NET or mono. ILSpy decompiles these programs and returns a corresponding C# program.

ILSpy does not apply algorithms to remove or reduce `goto` statements. This means that the decompiler output is often very long and consists mainly of `goto`s rather than other control flow structures.

5 Evaluation: Bugs found

In this section we answer research questions **RQ1** and **RQ2** set out in Section 1 and recapped below. In Section 6 we further evaluate FuzzFlesh in comparison to other decompiler testing tools to answer the remaining research questions.

RQ1 How effective is FuzzFlesh at identifying decompiler bugs? (Section 5.1)

RQ2 How does the efficacy of FuzzFlesh vary across languages and decompiler toolchains? (Section 5.2)

5.1 RQ1: Effectiveness of FuzzFlesh for bug-finding

We tested five decompilers, which are described in detail in Section 4 and summarised in Table 1. FuzzFlesh identified 12 bugs in total, comprising six misdecompilation bugs and six decompiler failures. Details of these bugs are provided in Table 2. The bugs were found across a total of four decompilers, which target two different low- to high-level language pairs. At the time of writing, eight bugs have been confirmed and six fixed by developers. We label each bug as “M” to denote a misdecompilation, or “R” to denote a refusal to decompile that we deemed to be likely to be in-scope for the decompiler in question, in line with our definitions of bug types in Section 2.

Most bugs are related to control flow restructuring. The bugs can be grouped into themes: cycle bugs, switch bugs, and Java-specific irreducible control flow bugs.

5.1.1 Cycles

At a low level, all program cycles are implemented via branch instructions. The control flow recovery algorithm must translate these to high-level looping constructs, potentially resorting to the use of `goto` statements if these are supported by the target high-level language. As

Table 2 List of decompilation bugs found (GitHub issues are hyperlinked to the bug ID) Type M = misdecompilation, Type R = refusal to decompile.

ID	Decompiler	Type	Description	Fixed
G1	Ghidra	M	Missing jump table entry	✓
G2	Ghidra	M	Incorrect counter increment	✓
G3	Ghidra	M	Counter array incorrectly sized	
G4	Ghidra	M	Locals instead of array	
G5	Ghidra	R	Incorrect switch decompilation	✓
C1	CFR	M	Missing switch case	
C2	CFR	R	Nested switch	✓
C3	CFR	R	Irreducible control flow cannot be restructured	✓
F1	FernFlower	M	Incorrect infinite loop	
F2	FernFlower	R	Irreducible control flow cannot be restructured	
J1	JADX	R	Code restructure failure	✓
J2	JADX	R	Unreachable block	

discussed in Section 2, some decompilers use advanced recovery algorithms that aim to minimise or completely avoid the use of `goto`, but can be hard to implement correctly. FuzzFlesh found several bugs related to incorrect decompilation of cycles.

As an example, Figure 4 shows a CFG and path that generated by FuzzFlesh that triggered a bug in FernFlower after being fleshed into Java bytecode; this is issue F1 in Table 2. The path is shown by solid edges, while other off-path edges are shown by dotted lines. Figure 5 shows the CFG and path corresponding to the Java program that was emitted by FernFlower. The decompiler output contains an infinite loop: FernFlower attempted to form the code into a `while` structure but did not correctly determine how the termination should occur.

Other cycle-related bugs included misdecompilations of loop-control variables by Ghidra.

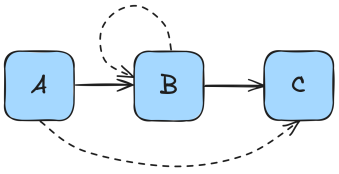


Figure 4 CFG and path of original program.

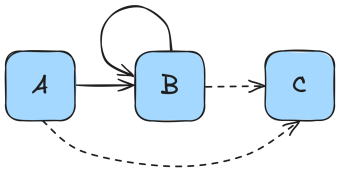


Figure 5 CFG and path of program decompiled by FernFlower containing an infinite loop.

5.1.2 Switch statements

Three bugs were related to the decompilation of switch statements. Figures 6a and 6b show an example of a switch misdecompilation by Ghidra. The original program contains a jump table with five distinct instruction addresses (lines 3–7 of Figure 6a). However, the decompiled code only contains four switch cases (lines 3–6 of Figure 6b): there should be another case for the value 0. The problem was caused by an issue in the algorithm that reconstructs loops, which did not correctly handle the existence of a self-edge. This has been fixed by the Ghidra developers in response to our bug report [36].

```
switchD_001000b9::switchdataD_0010018c

0010018c d4 fe ff ff     uint     FFFFFED4h
00100190 98 ff ff ff     uint     FFFFFF98h
00100194 db ff ff ff     uint     FFFFFFDBh
00100198 51 ff ff ff     uint     FFFFFE51h
0010019c 76 ff ff ff     uint     FFFFFE76h
```

(a) Original jump table.

```
void test(int* p_1, int* p_2) {
switch(p_2[local_2]){
    case 1: goto block_D; ...
    case 2: goto block_E; ...
    case 3: goto block_I; ...
    case 4: goto block_F; ...
}
```

(b) Decompiled switch statement.

Figure 6 Illustration of a bug in Ghidra related to switch statements (issue G1 of Table 2). The jump table contains five entries, but only four of these are captured by switch cases in the decompiled code.

5.1.3 Java-specific irreducible control flow

As discussed in Section 4.2, Java bytecode allows irreducible control flow while Java does not. This means that decompilers must structure any irreducible control flow to produce valid Java. Irreducible control flow can be converted to reducible control flow using several techniques [8]. We found a number of bugs relating to this conversion process. These bugs are particularly interesting because they are unlikely to be found by program generation approaches that do not specifically aim to create unusual control flow structures.

An example of an irreducible CFG produced by FuzzFlesh is shown in Figure 7. This formed the basis for a program that caused a decompiler failure in FernFlower, which we reported in issue F2. Feedback in the discussion thread for the issue indicates that FernFlower should be able to handle simple cases of irreducible control flow such as this one. Indeed, the removal of any edge, node or statement from the CFG resulted in a program that FernFlower was able to decompile, even those still containing irreducible control flow.

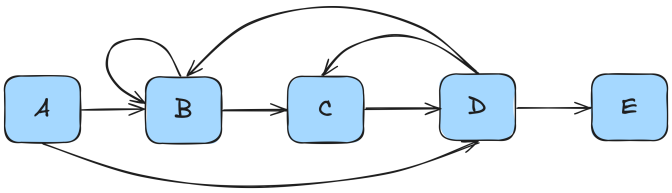


Figure 7 CFG that triggered a decompiler failure bug in FernFlower (issue F2 of Table 2).

5.2 RQ2: Efficacy of FuzzFlesh across languages and decompiler toolchains

FuzzFlesh found bugs in binary and Java decompilers but was not successful at finding bugs in C# decompilers. This is likely to be because CIL is fairly high level (unlike machine code), C# allows unrestricted control flow (unlike Java), and ILSpy does not appear to attempt to minimise the number of `goto` statements in the output code.

FuzzFlesh is not able to test decompilers whose output requires extensive engineering effort to recompile, for example angr [57].

6 Evaluation: Comparison to other approaches

In this section we answer research questions **RQ3** and **RQ4** set out in Section 1 and recapped below by evaluating the efficacy of FuzzFlesh and its ability to exercise the control flow recovery components of decompilers in comparison to other decompiler testing approaches.

RQ3 How thoroughly is FuzzFlesh able to test the control flow recovery components of decompilers in comparison to other decompiler testing approaches? (Section 6.2)

RQ4 To what extent can FuzzFlesh identify bugs that are different from those identified by other decompiler testing approaches? (Section 6.3)

6.1 Overview of decompiler testing approaches

We compare FuzzFlesh to two other tools[4] that aim to test decompilers:

- **DecFuzzer** [41], a binary decompiler fuzzer that targets open-source decompilers Ghidra and RetDec and commercial decompilers IDA-Pro and JEB3. It uses Csmith to generate an initial corpus of programs for de- and recompilation. Only a subset of the Csmith grammar is used to improve the likelihood of recompilability. Mutations are applied to any programs that successfully recompile, designed to yield programs that are equivalent modulo inputs [37]. Comparing results between original and mutated programs then provides a metamorphic test oracle [11].
- **JD-Tester** [43], a Java decompiler fuzzer that targets three Java decompilers: CFR, FernFlower, and JADX. JD-Tester uses two existing Java program generators: JavaFuzzer and Hephaestus. Each program is compiled to a class file, decompiled by the decompiler under test, and recompiled. The output of the original and recompiled program are compared to find decompilation bugs.

Table 3 Overview of decompiler testing tools.

Tool	Generator	Decompilation scenarios			Control flow bugs
		bin to C	class to Java	CIL to C#	
FuzzFlesh	FuzzFlesh	✓	✓	✓	✓
DecFuzzer	Csmith	✓			
JD-Tester	JavaFuzzer / Hephaestus		✓		✓

Table 3 summarises the characteristics of each testing approach, and highlights two key features of FuzzFlesh. First, FuzzFlesh covers multiple (low-level, high-level) language pairs, while the other tools only cover a single language pair. We are not aware of any decompiler testing tool that is able to target multiple languages. This is because of the simple nature of program generation in FuzzFlesh. For example, FuzzFlesh devotes only 86 lines of code to Java-specific program generation, while JavaFuzzer consists of 3,412 lines and Csmith contains around 40,000 lines of code.[5] This feature means that it is easy to add an additional language pair to FuzzFlesh. Most of the program generation logic is language-independent, which means it does not need to be repeated for each new language.

[4] A recent paper evaluates binary decompilers using Dsmith [7], a fuzzer based on Csmith. The artifact was released in September 2024; due to time constraints we have not included it in our evaluation.

[5] Lines of code were measured using the cloc tool.

Second, Fuzzflesh targets control flow recovery. DecFuzzer and JD-Tester programs do contain complex control flow, however DecFuzzer is not able to find any control-flow related bugs. A detailed comparison of the nature of bugs found by each tool is given in Section 6.3.

6.2 RQ3: Comparison of thoroughness of control flow recovery testing

Code coverage measures the proportion of the decompiler source code that is executed ("covered") during testing. This provides a high-level indication of the extent to which a testing tool is exercising the decompiler. However, coverage is limited as an evaluation metric because (a) it does not provide any information on the relative importance of different lines, and (b) it is possible to execute code that contains a bug without actually triggering the bug. We therefore also include a comparison of actual bugs in Section 6.3.

Table 4 summarises the aggregate line coverage of each decompiler by FuzzFlesh, JD-Tester and DecFuzzer. The coverage is calculated by running each tool in fuzzing mode 8 hours on a 16-core 128GB RAM machine running Ubuntu 22.04. The "path known" and "path unknown" configurations of FuzzFlesh refer to whether the directions array that guides the path through the CFG are hard-coded into the program or passed at runtime respectively (see Section 3.3). The "JavaFuzzer" and "Hephaestus" configurations of JD-tester refer to the underlying Java program generator used (see Section 6.1). We report bytecode instruction coverage for Java and line coverage for C throughout; branch coverage follows a similar pattern but is lower for all tools (not reported here but available in the accompanying artifact).

We summarise our main findings, before giving more specific detail on the experimental setup and results for binary decompilers and Java decompilers in Sections 6.2.1 and 6.2.2, respectively.

Overall, coverage of each decompiler is highest for the language-specific testing tools JD-Tester and DecFuzzer. This is expected because they are far more syntactically expressive than the FuzzFlesh programs. Nevertheless, despite its limited syntax, FuzzFlesh achieves reasonable overall coverage of up to 36% of Ghidra, 31% of CFR, 42% of FernFlower, and 37% of JADX. The different FuzzFlesh generation modes generally achieve similar levels of overall coverage, as do the different program generators used by JD-Tester.

As part of these experiments we also computed coverage data at checkpoints of 2 and 4 hours of each 8 hour fuzzing run, and found that coverage had already plateaued after 2 hours for all tools.

Table 4 Instruction coverage comparison for an 8 hour fuzzing run (%).

Decompiler	Testing tool				
	FuzzFlesh path known	FuzzFlesh path unknown	DecFuzzer	JD-Tester JavaFuzzer	JD-tester Hephaestus
Ghidra	36	31	39	–	–
CFR	31	30	–	53	42
FernFlower	42	41	–	55	48
JADX	37	36	–	50	44

6.2.1 Binary decompiler coverage

Ghidra is implemented in a combination of Java and C++. We restrict our coverage comparison to the decompiler component, which is implemented entirely in C++. We use gcov and gcovr to analyse coverage.

DecFuzzer generates programs using Csmith. These are compiled to binaries, decompiled using Ghidra, and recompiled. Successfully recompiled programs are used as seeds for equivalence modulo inputs testing. DecFuzzer aims to improve the likelihood of successful recompilation by restricting Csmith to produce only one function (besides `main()`), limiting the complexity of expressions, and excluding arrays, structs, unions, pointers and various other language features. The restricted Csmith command in full is:

```
csmith -no-arrays -no-structs -no-unions -no-safe-math -no-pointers
-no-longlong -max-funcs 1 -max-expr-complexity 5
```

As discussed further in Section 6.3.2, despite these restrictions DecFuzzer's use of Csmith does not lead to programs that can be successfully recompiled after decompilation by recent versions of Ghidra.

In addition to overall coverage, we compare the unique coverage of each testing tool. Table 5 shows that FuzzFlesh is able to cover around 12,700 lines within Ghidra that DecFuzzer programs are not able to cover. In particular, FuzzFlesh covers parts of the control flow restructuring algorithms (within `block.cc`) that DecFuzzer does not cover.

Table 5 Unique line coverage comparison for the binary decompiler Ghidra.

	Lines covered by				
Decompiler	FuzzFlesh only	DecFuzzer only	Both	Neither	Total lines
Ghidra	12,700	6,300	12,800	39,800	71,600

6.2.2 Java decompiler coverage

We use the Jacoco code coverage tool to measure coverage of the bytecode-to-Java decompilers, which are implemented in Java. Fuzzing mode involves program generation, compilation, decompilation, recompilation, and test oracle evaluation. Neither of these tools requires an initial seed corpus for test generation. We then gather the programs that were generated by each testing tool and decompile them using versions of CFR, FernFlower, and JADX that are instrumented with Jacoco. The motivation for separating the coverage measurement from the previous step is that coverage instrumentation can slow program execution, which could lead to underestimating the coverage achieved in a given time budget.

In addition to overall coverage, we also analyse the unique coverage of each tool. Although JD-Tester achieves higher overall coverage, FuzzFlesh is able to cover some parts of each decompiler's code that are not covered by JD-tester. Table 6 shows the number of instructions that are covered by only FuzzFlesh or only JD-Tester. The instructions covered by FuzzFlesh that are not covered by JD-Tester mainly relate to control-flow recovery. For example, FuzzFlesh covers instructions in the following classes that are not covered by JD-Tester (non-exhaustive list):

- CFR: SwitchReplacer, LoopIdentifier, JumpsIntoDoRewriter, and JoinBlocks
- FernFlower: IrreducibleCFGDeobfuscator, ControlFlowGraph, SwitchStatement, JumpInstruction, and DomHelper (which calculates node dominance),
- JADX: BlockSplitter, BlockProcessor, FixMultiEntryLoops, SwitchInsn, and LoopRegionVisitor

Table 6 Unique bytecode instruction coverage comparison for Java decompilers.

Decompiler	Instructions covered by				Total instructions
	FuzzFlesh only	JD-Tester only	Both	Neither	
CFR	800	35,300	51,000	79,000	166,100
FernFlower	4,700	15,900	34,900	38,300	93,800
JADX	1,300	17,100	43,400	60,400	122,200

6.3 RQ4: The kinds of bugs found by FuzzFlesh vs. other approaches

While code coverage provides a useful overview of the potential capabilities of decompiler testing tools, it is only a proxy for the ideal fuzzer performance measurement: the potential number of "useful" bugs that can be found [33]. In this section we evaluate the bug-finding ability of FuzzFlesh in comparison to DecFuzzer and JD-Tester.

6.3.1 Bugs in all decompilers

FuzzFlesh is the only decompiler testing tool that we are aware of that is able to find bugs in decompilers that target different language-pairs. In this respect it has a higher bug-finding potential than other tools, since it can be easily extended to other languages in the future, while DecFuzzer and JD-Tester both rely on language-specific program generators.

6.3.2 Bugs in binary decompilers

Problems with recompilability within DecFuzzer mean that we cannot perform a head-to-head comparison of bug-finding ability. No DecFuzzer programs are recompilable using the latest Ghidra plugin, which means that they are not able to identify misdecompilations. DecFuzzer runs on Radare2, which is a reverse-engineering framework that uses a Ghidra plugin for decompilation. However, multiple compiler errors occur when trying to recompile the decompiled programs within DecFuzzer's workflow. The errors include trivial issues such as "unknown type name uint" but also more complex issues such as the incorrect number of arguments to functions. Modifying DecFuzzer to support recompilability would likely require a significant amount of engineering work to address the latest Ghidra output format. This illustrates the difficulty of ensuring recompilability. We contacted the authors of DecFuzzer, and they shared with us the commit used for Radare2 and their best estimate of the versions of the Ghidra plugin, r2ghidra, and other tools that were used, however some of the original artifacts from the paper (such as the contemporary binaries) are no longer available. It has therefore not been possible to reconstruct a working version of the tool to perform a direct bug-finding comparison due to incompatibilities between the historic Ghidra versions and current binaries. The simple syntax used by FuzzFlesh means that we are less likely to encounter this type of recompilation issues in the future, because we expect that future versions of all tools will continue to support basic syntax.

Considering the original bugs reported to have been found by DecFuzzer [41] and the actual bugs found by FuzzFlesh: FuzzFlesh mostly identifies control flow related bugs, while DecFuzzer mainly identified bugs relating to type recovery and optimisations, and was not able to find any bugs within the control flow recovery module of Ghidra or any other binary decompiler. FuzzFlesh found a total of 6 bugs in Ghidra, at least three of which are confirmed by developers to be directly related to control flow recovery. While the sample size is small, this indicates that FuzzFlesh is likely to be capable of finding control flow related bugs that DecFuzzer would not identify.

6.3.3 Bugs in Java decompilers

JD-Tester is able to find control flow related bugs. This is because (a) the programs generated by JD-Tester include control flow structures, and (b) decompiling bytecode to Java is easier than decompiling machine code (for example because instructions and data are separate). This means that more programs are recompilable than in the case of binary decompiler testing, which enables the detection of control flow misdecompilations. However, some of the bugs identified by FuzzFlesh are unlikely to be discoverable by JD-Tester due to the complex control flow employed. To compare the tools directly, we evaluate whether JD-Tester is able to find the control flow related bugs that FuzzFlesh identified in Java decompilers. We restrict the comparison to bugs that have been fixed.

To do this, we rely on the notion of "correcting commit" used in an empirical study of compiler testing techniques [9]: the commit in which a bug is fixed (so that the bug manifests before the commit but not after). This is essential because otherwise, especially in the case of misdecompilation bugs, it is not possible to directly identify which underlying bug is triggered by a bug-inducing test program. Our evaluation works as follows:

- We identify correcting commits associated with two CFR bugs and one JADX bug.
- We run JD-Tester on a preceding commit (i.e. one for which we know that the bug is present) for 2 hours and identify any failing test cases. These are potential bug-triggering programs that could identify the bug.
- We run the failing test cases on the correcting commit. If any test now passes, then we infer that its original failure was caused by the bug. Such a test identifies the bug.

We found that none of the JD-Tester programs that failed on the preceding commit passed on the correcting commit, indicating that none of the potential bug-triggering test programs produced by JD-Tester were caused by these specific bugs. Although the time budget is limited, in contrast FuzzFlesh was able to identify these bugs almost immediately. These results suggest that JD-Tester is not able to identify some bugs that are found by FuzzFlesh. This evaluation is based on a very small sample size of the two correcting commits (recall that we focus here only on bugs that have been fixed), however the coverage analysis provides supporting evidence that JD-Tester may not be able to trigger bugs in some parts of the decompiler code. While neither of the FernFlower bugs have been fixed, one relates to irreducible control flow restructuring. This would certainly not be detectable by JD-Tester because its programs do not contain irreducible control flow. Similarly, the Jadx bug would not be detectable by JD-Tester because it related to irreducible control flow restructuring.

To summarise, in prior work JD-Tester has been shown to perform well at finding a range of bugs in Java decompilers, including some control flow bugs [41]. However, FuzzFlesh was able to identify some previously-unknown and potentially significant bugs that JD-Tester appears unable to find.

7 Related Work

Decompiler testing. Historically, the main evaluation metrics used to assess decompilers were focused on readability, for example code compactness and complexity (measured by cyclic complexity and the number of `goto`s in a decompiled program). In recent years there has been an increasing focus on correctness within decompiler research [41].

As discussed in Section 6, as far as we are aware FuzzFlesh is the first random program generator that is specifically targeted at decompilers. JD-Tester [43] uses a differential testing approach with the pre-existing program generator JavaFuzzer [66]. It finds 62 bugs in three

Java decompilers: CFR, FernFlower, and JADX. A vast majority of the bugs reported in this study are decompilation failures and crashes, most of which have not been fixed by developers. This highlights the importance of focusing on misdecompilations that are more likely to present a serious problem to users.

DecFuzzer [41] evaluates leading commercial and open-source decompilers using the existing program generator Csmith [68] to generate a corpus of programs. It uses equivalence modulo inputs testing [37] to compare the performance of four binary decompilers (Ghidra [50], RetDec [3], IDA-Pro [27] and Jeb3 [53]). They find 13 bugs, however they report that this required "extensive manual effort" from expert security analysts.

We recently became aware of D-Helix [71], which proposes a general framework for decompiler testing that is intended to assist developers with decompilation tasks, and identifies 25 bugs across binary decompilers Ghidra and angr. D-Helix detects semantic discrepancies between the binary input and the decompiled output by performing symbolic execution on both the original binary and a recompiled decompiled output. It also includes a debugging tool that helps to pinpoint the source of the discrepancy. In future work it would be interesting to compare FuzzFlesh experimentally with D-Helix.

Compiler testing. We have used techniques from the wider compiler fuzzing domain. In recent years there has been a surge of interest in randomised compiler testing techniques, for languages such as C [68, 42, 37], Java / Java bytecode [12, 69, 62], various GPU programming languages [38, 14, 15], Rust [56, 61], Verilog [26], as well as techniques for testing machine learning compilers [39, 40, 44], and techniques that use large language models for randomised compiler testing [67].

Our approach is directly inspired work on testing compilers for SPIR-V, a low level language for GPU programming [34], in which skeleton control flow graphs generated by the Alloy tool [31] are fleshed into executable SPIR-V programs. Our approach is somewhat related to a compiler testing method called *skeletal program enumeration* where the control flow structure of an existing program is used as the basis for generating a large set of programs that populate this structure with different uses of program variables [70]. Other program generation approaches include creating full programs from scratch [68, 42] and mutating existing full programs [37, 15, 49, 38]. A comprehensive review of current state-of-the-art compiler testing approaches is given by Chen et al. [10]. Recently, Large Language Models (LLMs) have been used to generate programs in a range of languages based on user prompts [63]. However, these programs are only designed to detect crash bugs due to the difficulty of ensuring that LLM-generated programs are free from undefined behaviour (UB). We are not aware of any other program generator targeting miscompilations or misdecompilations that is designed to be easily extensible to other languages.

Strategies for constructing test oracles include differential testing [46, 38, 68, 42], where the output of similar compilation systems are compared, and metamorphic testing [11, 37, 49, 38, 15], which specifies how changes in the input source code should affect the output code produced by the compiler.

8 Conclusion and future work

CFG-based program generation is a novel decompiler testing approach that we have used to identify 12 bugs across four decompilers covering two languages. Our implementation, FuzzFlesh, demonstrates the feasibility of sharing a language-independent random CFG and path generator across several different languages with a small amount of incremental programming time.

Areas for further work include alternative CFG and path generation algorithms. For example, mining a corpus of graph structures from real-world programs and mutating them. This would ensure that realistic control flow structures are included in test cases. In addition, more sophisticated path generation algorithms could be created to ensure that each generated CFG is well-explored. For example, repeating path cycles to ensure that loop-like behaviour is included in the test programs.

CFG-based program generation also has the potential to be applied to find bugs in general-purpose compilers. The self-checking nature of the test cases mean that only a single compiler is required, and the ease of adding new languages mean that the approach could be used for newly established compiler toolchains. It is most likely to be useful for testing the control flow optimisation passes of compilers. However, our current implementation is unlikely to produce programs that will detect bugs in mainstream compilers without further work. Decompiler control flow recovery is challenging because it involves imposing structure onto spaghetti code: the more bizarre the control flow, the more challenging decompilation is. Compiling complex control flow has different challenges: it is permissible for a compiler to produce spaghetti code as output, and if the control flow of the source program is too complex then it is possible that the compiler will not attempt to optimise it. Therefore, further work in this area could explore graph generation approaches that may be more likely to find bugs in compilers. For example, graphs could be generated in a way that will be sufficiently complex to thoroughly exercise control flow optimisation compiler passes, but not so complex that compilers will not attempt to perform optimisation.

References

1 Jaime Acosta and Daniel Krych. Hands-on cybersecurity studies: Uncovering and decoding malware communications-malware analysis with Ghidra. Technical Report ARL-TR-9129, Computational and Information Sciences Directorate, DEVCOM Army Research Laboratory, 2020. Accessed: 2025-04-25. URL: `https://apps.dtic.mil/sti/trecms/pdf/AD1119396.pdf`.

2 Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

3 Avast. RetDec: A retargetable machine-code decompiler based on LLVM, 2022. Accessed: 2025-04-25. URL: `https://github.com/avast/retdec`.

4 Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. Ahoy SAILR! there is no need to DREAM of C: A Compiler-Aware structuring algorithm for binary decompilation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 361–378, Philadelphia, PA, August 2024. USENIX Association. Accessed: 2025-04-25. URL: `https://www.usenix.org/system/files/usenixsecurity24-basque.pdf`.

5 Lee Benfield. CFR - another Java decompiler, 2025. Accessed: 2025-04-25. URL: `https://www.benf.org/other/cfr/`.

6 David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 353–368, Washington, D.C., August 2013. USENIX Association. Accessed: 2025-04-25. URL: `https://www.usenix.org/system/files/conference/usenixsecurity13/sec13-paper_schwartz.pdf`.

7 Ying Cao, Runze Zhang, Ruigang Liang, and Kai Chen. Evaluating the effectiveness of decompilers. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, pages 491–502, New York, NY, USA, 2024. Association for Computing Machinery. `doi:10.1145/3650212.3652144`.

8 Larry Carter, Jeanne Ferrante, and Clark D. Thomborson. Folklore confirmed: reducible flow graphs are exponentially larger. In Alex Aiken and Greg Morrisett, editors, *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, pages 106–114. ACM, 2003. doi:10.1145/604131.604141.

9 Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 180–190. ACM, 2016. doi:10.1145/2884781.2884878.

10 Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *ACM Comput. Surv.*, 53(1):4:1–4:36, 2021. doi:10.1145/3363562.

11 T.Y. Chen, S.C. Cheung, and S.M. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, The Hong Kong University of Science and Technology, 1998. Accessed: 2025-04-25. URL: https://www.cse.ust.hk/faculty/scc/publ/CS98-01-metamorphictesting.pdf.

12 Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of JVM implementations. In Chandra Krintz and Emery D. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 85–99. ACM, 2016. doi:10.1145/2908080.2908095.

13 Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Softw. Pract. Exp.*, 25(7):811–829, 1995. doi:10.1002/SPE.4380250706.

14 Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.*, 1(OOPSLA):93:1–93:29, 2017. doi:10.1145/3133917.

15 Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpinski. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 1017–1032. ACM, 2021. doi:10.1145/3453483.3454092.

16 Luke Dramko, Jeremy Lacomis, Edward J. Schwartz, Bogdan Vasilescu, and Claire Le Goues. A taxonomy of C decompiler fidelity issues. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024. Accessed: 2025-04-25. URL: https://www.usenix.org/system/files/usenixsecurity24-dramko.pdf.

17 Lukás Durfina, Jakub Kroustek, and Petr Zemek. PsybOt malware: A step-by-step decompilation case study. In Ralf Lämmel, Rocco Oliveto, and Romain Robbes, editors, *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*, pages 449–456. IEEE Computer Society, 2013. doi:10.1109/WCRE.2013.6671321.

18 Mike Van Emmerik and Trent Waddington. Using a decompiler for real-world source recovery. In *11th Working Conference on Reverse Engineering, WCRE 2004, Delft, The Netherlands, November 8-12, 2004*, pages 27–36. IEEE Computer Society, 2004. doi:10.1109/WCRE.2004.42.

19 P Erdös and A Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

20 Miguel Gómez-Zamalloa, Elvira Albert, and Germán Puebla. Decompilation of Java bytecode to Prolog by partial evaluation. *Inf. Softw. Technol.*, 51(10):1409–1427, 2009. doi:10.1016/J.INFSOF.2009.04.010.

21 Amber Gorzynski. FuzzFlesh, 2024. URL: https://github.com/ambergorzynski/control_flow_fleshing.

22 Amber Gorzynski. FuzzFlesh artifact, 2024. URL: https://github.com/ambergorzynski/control_flow_fleshing/tree/ecoop-2025.

23 James Gosling and Henry McGilton. The Java Language Environment, 1996. Accessed: 2025-04-25. URL: https://www.oracle.com/java/technologies/simple-familiar.html.

24 Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: thorough, declarative decompilation of smart contracts. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 1176–1186. IEEE / ACM, 2019. doi:10.1109/ICSE.2019.00120.

25 Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. A comb for decompiled C code. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20, pages 637–651, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3320269.3384766.

26 Yann Herklotz and John Wickerson. Finding and understanding bugs in FPGA synthesis tools. In Stephen Neuendorffer and Lesley Shannon, editors, *FPGA '20: The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, February 23-25, 2020*, pages 277–287. ACM, 2020. doi:10.1145/3373087.3375310.

27 Hex-Rays. IDA Pro, 2025. Accessed: 2025-04-25. URL: https://hex-rays.com/ida-pro/.

28 Joshua Homan. Deobfuscating Python bytecode, 2016. Accessed: 2025-04-25. URL: https://cloud.google.com/blog/topics/threat-intelligence/deobfuscating-python.

29 ICSharpCode. ILSpy, 2025. Accessed: 2025-04-25. URL: https://github.com/icsharpcode/ILSpy.

30 IntelliJ. Fernflower, 2025. Accessed: 2025-04-25. URL: https://github.com/JetBrains/intellij-community/blob/master/plugins/java-decompiler/engine/src/org/jetbrains/java/decompiler.

31 Daniel Jackson. Alloy: a language and tool for exploring software designs. *Commun. ACM*, 62(9):66–76, 2019. doi:10.1145/3338843.

32 JADX Authors. JADX, 2025. Accessed: 2025-04-25. URL: https://github.com/skylot/jadx.

33 George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2123–2138. ACM, 2018. doi:10.1145/3243734.3243804.

34 Vasileios Klimis, Jack Clark, Alan Baker, David Neto, John Wickerson, and Alastair F. Donaldson. Taking back control in an intermediate representation for GPU computing. *Proc. ACM Program. Lang.*, 7(POPL):1740–1769, 2023. doi:10.1145/3571253.

35 P. L. Krapivsky and S. Redner. Organization of growing random networks. *Physical Review E*, 63(6), May 2001. doi:10.1103/physreve.63.066123.

36 Ryan Kurtz. Git commit fixing a bug found by fuzzflesh in Ghidra, 2024. Accessed: 2025-04-25. URL: https://github.com/NationalSecurityAgency/ghidra/commit/6ede2b498f75e29c13fe64d3ce7549e90e17aa2b.

37 Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 216–226. ACM, 2014. doi:10.1145/2594291.2594334.

38 Christopher Lidbury and Alastair F. Donaldson. Dynamic race detection for C++11. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 443–457. ACM, 2017. doi:10.1145/3009837.3009857.

39 Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. NNSmith: Generating diverse and valid test cases for deep learning compilers. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 530–543. ACM, 2023. doi:10.1145/3575693.3575707.

40 Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. Coverage-guided tensor compiler fuzzing with joint IR-pass mutation. *Proc. ACM Program. Lang.*, 6(OOPSLA1):1–26, 2022. doi:10.1145/3527317.

41 Zhibo Liu and Shuai Wang. How far we have come: Testing decompilation correctness of C decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, pages 475–487, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3395363.3397370.

42 Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for C and C++ compilers with yarpgen. *Proc. ACM Program. Lang.*, 4(OOPSLA):196:1–196:25, 2020. doi:10.1145/3428264.

43 Yifei Lu, Weidong Hou, Minxue Pan, Xuandong Li, and Zhendong Su. Understanding and finding Java decompiler bugs. *Proc. ACM Program. Lang.*, 8(OOPSLA1), April 2024. doi:10.1145/3649860.

44 Haoyang Ma, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. Fuzzing deep learning compilers with HirGen. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 248–260. ACM, 2023. doi:10.1145/3597926.3598053.

45 Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. The convergence of source code and binary vulnerability discovery - A case study. In Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako, editors, *ASIA CCS '22: ACM Asia Conference on Computer and Communications Security, Nagasaki, Japan, 30 May 2022 - 3 June 2022*, pages 602–615. ACM, 2022. doi:10.1145/3488932.3497764.

46 William M. McKeeman. Differential testing for software. *Digit. Tech. J.*, 10(1):100–107, 1998. Accessed: 2025-04-25. URL: http://www.dtjcd.vmsresource.org.uk/pdfs/dtj_v10-01_1998.pdf.

47 Jonathan Meyner and Daniel Reynaud. Jasmin, 2004. Accessed: 2025-04-25. URL: https://jasmin.sourceforge.net/.

48 Omid Mirzaei, Roman Vasilenko, Engin Kirda, Long Lu, and Amin Kharraz. SCRUTINIZER: detecting code reuse in malware via decompilation and machine learning. In Leyla Bilge, Lorenzo Cavallaro, Giancarlo Pellegrino, and Nuno Neves, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 18th International Conference, DIMVA 2021, Virtual Event, July 14-16, 2021, Proceedings*, volume 12756 of *Lecture Notes in Computer Science*, pages 130–150. Springer, 2021. doi:10.1007/978-3-030-80825-9_7.

49 Kazuhiro Nakamura and Nagisa Ishiura. Random testing of C compilers based on test program generation by equivalence transformation. In *2016 IEEE Asia Pacific Conference on Circuits and Systems, APCCAS 2016, Jeju, South Korea, October 25-28, 2016*, pages 676–679. IEEE, 2016. doi:10.1109/APCCAS.2016.7804063.

50 National Security Agency. Ghidra - reverse software engineering framework, 2025. Accessed: 2025-04-25. URL: https://ghidra-sre.org/.

51 Oracle. The Java virtual machine instruction set, 2025. Accessed: 2025-04-25. URL: https://docs.oracle.com/javase/specs/jvms/se24/html/jvms-6.html.

52 Jihee Park, Sungho Lee, Jaemin Hong, and Sukyoung Ryu. Static analysis of JNI programs via binary decompilation. *IEEE Trans. Software Eng.*, 49(5):3089–3105, 2023. doi:10.1109/TSE.2023.3241639.

53 PNF Software. JEB3, 2013. Accessed: 2025-04-25. URL: https://www.pnfsoftware.com/blog/category/jeb3/.

54 Bernardo Quintero. From assistant to analyst: The power of Gemini 1.5 Pro for malware analysis, 2024. Accessed: 2025-04-25. URL: https://cloud.google.com/blog/topics/threat-intelligence/gemini-for-malware-analysis.

55 Pemma Reiter, Hui Jun Tay, Westley Weimer, Adam Doupé, Ruoyu Wang, and Stephanie Forrest. Automatically mitigating vulnerabilities in x86 binary programs via partially recompilable decompilation. *CoRR*, abs/2202.12336, 2022. arXiv:2202.12336.

56 Mayank Sharma, Pingshi Yu, and Alastair F. Donaldson. RustSmith: Random differential compiler testing for rust. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 1483–1486. ACM, 2023. doi:10.1145/3597926.3604919.

57 Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 138–157. IEEE Computer Society, 2016. doi:10.1109/SP.2016.17.

58 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. doi:10.1137/0201010.

59 Daniel Votipka, Seth M. Rabin, Kristopher K. Micinski, Jeffrey S. Foster, and Michelle L. Mazurek. An observational investigation of reverse engineers' processes. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1875–1892. USENIX Association, 2020. Accessed: 2025-04-25. URL: https://www.usenix.org/system/files/sec20-votipka-observational.pdf.

60 Pei Wang, Qinkun Bao, Li Wang, Shuai Wang, Zhaofeng Chen, Tao Wei, and Dinghao Wu. Software protection on the go: a large-scale empirical study on mobile app obfuscation. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 26–36. ACM, 2018. doi:10.1145/3180155.3180169.

61 Qian Wang and Ralf Jung. Rustlantis: Randomized differential testing of the Rust compiler. *Proc. ACM Program. Lang.*, 8(OOPSLA2):1955–1981, 2024. doi:10.1145/3689780.

62 Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. JITfuzz: Coverage-guided fuzzing for JVM just-in-time compilers. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 56–68. IEEE, 2023. doi:10.1109/ICSE48619.2023.00017.

63 Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4All: Universal fuzzing with large language models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 126:1–126:13. ACM, 2024. doi:10.1145/3597503.3639121.

64 Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping Johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 158–177. IEEE Computer Society, 2016. doi:10.1109/SP.2016.18.

65 Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015. Accessed: 2025-04-25. URL: https://www.ndss-symposium.org/wp-content/uploads/2017/09/11_4_2.pdf.

66 Andrey Yakovlev, Mohammad R. Haghighat, and Dmitry Khukhro. AzulSystems/JavaFuzzer, 2018. Accessed: 2025-04-25. URL: https://github.com/AzulSystems/JavaFuzzer.

67 Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. Whitefox: White-box compiler fuzzing empowered by large language models. *Proc. ACM Program. Lang.*, 8(OOPSLA2):709–735, 2024. doi:10.1145/3689736.

68 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011. doi:10.1145/1993498.1993532.

69 Zhiqiang Zang, Nathan Wiatrek, Milos Gligoric, and August Shi. Compiler testing using template java programs. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 23:1–23:13. ACM, 2022. doi:10.1145/3551349.3556958.

70 Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 347–361. ACM, 2017. doi:10.1145/3062341.3062379.

71 Muqi Zou, Arslan Khan, Ruoyu Wu, Han Gao, Antonio Bianchi, and Dave (Jing) Tian. D-Helix: A generic decompiler testing framework using symbolic differentiation. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024. Accessed: 2025-04-25. URL: https://www.usenix.org/system/files/usenixsecurity24-zou.pdf.

IsaBIL: A Framework for Verifying (In)correctness of Binaries in Isabelle/HOL

Matt Griffin
University of Surrey, Guildford, UK
Imperial College London, UK

Brijesh Dongol
University of Surrey, Guildford, UK

Azalea Raad
Imperial College London, UK

Abstract

This paper presents IsaBIL, a binary analysis framework in Isabelle/HOL that is based on the widely used Binary Analysis Platform (BAP). Specifically, in IsaBIL, we formalise BAP's intermediate language, called BIL and integrate it with Hoare logic (to enable proofs of *correctness*) as well as incorrectness logic (to enable proofs of *incorrectness*). IsaBIL inherits the full flexibility of BAP, allowing us to verify binaries for a wide range of languages (C, C++, Rust), toolchains (LLVM, Ghidra) and target architectures (x86, RISC-V), and can also be used when the source code for a binary is unavailable.

To make verification tractable, we develop a number of big-step rules that combine BIL's existing small-step rules at different levels of abstraction to support reuse. We develop high-level reasoning rules for RISC-V instructions (our main target architecture) to further optimise verification. Additionally, we develop Isabelle proof tactics that exploit common patterns in C binaries for RISC-V to discharge large numbers of proof goals (often in the 100s) automatically. IsaBIL includes an Isabelle/ML based parser for BIL programs, allowing one to *automatically* generate the associated Isabelle/HOL program locale from a BAP output. Taken together, IsaBIL provides a highly flexible proof environment for program binaries. As examples, we prove correctness of key examples from the Joint Strike Fighter coding standards and the MITRE database.

2012 ACM Subject Classification Theory of computation → Program reasoning

Keywords and phrases Binary Analysis Platform, Isabelle/HOL, Hoare Logic, Incorrectness Logic

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.14

Related Version *Extended Version*: `https://arxiv.org/abs/2504.16775` [21]

Supplementary Material *Other (Isabelle/HOL)*: `https://doi.org/10.5281/zenodo.15261359` [20]

Funding *Brijesh Dongol*: Supported by VeTSS and EPSRC grants EP/X037142/1, EP/X015149/1, EP/V038915/1, and EP/R025134/2.
Azalea Raad: Supported by the UKRI Future Leaders Fellowship MR/V024299/1, by the EPSRC grant EP/X037029/1 and by VeTSS.

1 Introduction

Analysing binary code is highly challenging and is essential for decompilation and verification when one does not have access to the source code (e.g., proprietary code). It is also a key tool for finding security vulnerabilities and exploits and is often the only way to (dis)prove properties about programs [46]. Many tools for binary analysis such as angr [52], LIEF [49] and Manticore [36] lift machine code to higher-level, human-readable representations. These tools have been developed for different goals covering different languages (Rust, C++), hardware architectures (e.g., x86, RISC-V), types of binaries (ELF, smart contracts) and security analysis (malware analysis, reverse engineering).

© Matt Griffin, Brijesh Dongol, and Azalea Raad;
licensed under Creative Commons License CC-BY 4.0

The *Binary Analysis Platform* (BAP) [8] provides a generic approach to binary analysis. BAP natively supports several languages (C, Python and Rust) and hardware architectures (x86, Arm, RISC-V, MIPS) and additionally can be integrated with toolchains such as LLVM and Ghidra to analyse programs on less common architectures such as Atmel AVR [2]. BAP is well-supported and widely used with an active user community. The platform can analyse binaries derived from a particular source language such as C/C++ and Rust, as well as closed-source functions. BAP provides a disassembler that lifts binaries into a unified intermediate language called *BIL (Binary Instruction Language)*, thus enabling architecture-agnostic analysis. While BAP also provides a set of static analysis tools, they do not provide mechanisms for formal verification, e.g., logics for (dis)proving correctness of programs. Recent works have covered subsets of BAP (called AIR) in UCLID5 [10] and use an extended semantics to enable reasoning about an information flow hyper-property known as *trace-property observational determinism* [10].

We present IsaBIL, a verification framework for low-level binaries encoded using the Isabelle/HOL theorem prover. IsaBIL provides a *deep embedding* of BIL programs into Isabelle/HOL, providing a *direct interface* between BAP and Isabelle/HOL without using any additional tools. We leverage the fact that BAP can generate BIL programs as abstract data types, providing a convenient hook for IsaBIL's ML-based parser.[1] Since BAP can lift binaries into BIL both with and without the source code being available, so can IsaBIL. Moreover, IsaBIL allows the generated BIL program to be connected to *any* operational semantics. We build on the semantics described by [8] (see § 3) to enable bit-precise analysis. Our approach generates an Isabelle *locale* [27], which provides an extensible interface that can be instantiated to different scenarios (see Figure 2). We demonstrate this by both proving (using Hoare logic [23]) and disproving (using Incorrectness logic [42]) properties for a number of key examples (see Table 1).

Table 1 IsaBIL examples and their sizes.

Test	Size (LoC)			
	C	**RISC-V**	**BIL**	**Isabelle**
AV Rule 17	13	34	80	94
AV Rule 19	10	16	36	64
AV Rule 20	18	52	118	393
AV Rule 21	16	45	105	210
AV Rule 23	11	35	81	90
AV Rule 24	15	40	107	74
AV Rule 25	24	99	233	578
DF (good)	27	19	44	366
DF (bad)	27	19	43	171
read_data (bad)	74	131	296	474
sec_recv (bad)	38	122	272	752
Total	**273**	**612**	**1415**	**3266**

Scalability. Our approach is *compositional* in that we can analyse and (dis)prove properties about a *code fragment*, say a function f, *in isolation* and then *reuse* these properties (without reproving them) in bigger contexts where f is used. To show this, we use our technique to detect (using incorrectness logic) a *double-free vulnerability* in the cURL library (used primarily for transferring data over the internet). As we demonstrate in § 7, although this

[1] Note that this is a significant advancement over the encoding by Griffin and Dongol [19], which used an external Python tool to translate BAP outputs to Isabelle/HOL.

Listing 1 Double Free (Bad).

```
void bad() {
    int *p = malloc(42);
    if (MyTrue) {
        free(p);
        // ...
    }
    free(p);
}
```

Listing 2 Double Free (Good).

```
void good() {
    int *p = malloc(42);
    if (MyTrue) {
        free(p);
        return;
    }
    free(p);
}
```

vulnerability is "hidden" in an internal function (not exposed in a header file) and is not visible to client applications, it is accessible by a call chain (comprising four separate functions) that can be triggered from an external function, i.e. it is accessible from within a client application and thus this vulnerability can be easily exploited. Indeed, this vulnerability has been previously documented as CVE-2016-8619.

Contributions. Our main contributions are as follows. **(1)** We develop IsaBIL, a flexible, semi-automated tool for verifying binaries (lifted to BIL) within the Isabelle/HOL theorem prover. Our mechanisation is complete with respect to the BIL language and semantics. During the mechanisation, we uncovered and fixed several inconsistencies in the official BAP documentation (see §3). **(2)** We developed IsaBIL as a highly flexible and extensible system (see Figure 2) using Isabelle *locales* [3, 27]. We incorporate logics for (in)correctness, thus obtain methods for both proving and disproving properties. To the best of our knowledge, this is the first application of Incorrectness logic [42] in the analysis of low-level binaries. **(3)** We extend IsaBIL with a number of automation techniques, including reusable high-level big-step theorems, proof tactics that automatically discharge large numbers of proof goals, and architecture-specific proof optimisations. We also include a native parser for BIL_{ADT} programs, written in Isabelle/ML, to *automatically* transpile BAP outputs to an Isabelle/HOL locale suitable for verification. **(4)** We apply our methods to a number of examples (see Table 1), including key tests from the Joint Strike Fighter (JSF) C++ Coding Standards [32]. **(5)** We show the scalability of IsaBIL by detecting a double-free vulnerability in a large example (253 assembly LoC) in the cURL library.

2 Motivation and Workflow

In this section, we motivate our work and provide an overview of our overall approach using two running examples. We give the C program and their equivalent BIL in §2.1. We present our workflow and an overview of the Isabelle/HOL mechanisation in §2.2.

2.1 Two C programs and their BIL representations

We motivate our analysis using two variations of a double-free vulnerability (listed as CWE-415 in the MITRE database[2]). A double-free error occurs when a program calls `free` twice with the same argument. This may cause two later calls to `malloc` to return the same pointer, potentially giving an attacker control over the data written to memory.

The occurrence of the double-free vulnerability is straightforward to see in Listing 1, but of course, in a real program, the vulnerability may be much more difficult to identify. Listing 2 presents an alternative program that contains two occurrences of `free(p)` with no intervening `malloc`. However, Listing 2 does not contain a double-free vulnerability since the true branch returns from the method call before the second `free(p)` is executed.

[2] `https://cwe.mitre.org/data/definitions/415.html`

Listing 3 Pretty printed BIL of Listing 1.

```
1  sub bad(bad_result)
2
3  X10 := 0x2A
4  call @malloc with return 5
5  mem := mem with [X8 - 0x18, el]:u64 <- X10
6
6  X15 := mem[X3 - 0x7C8, el]:u32
7  when (X15 = 0) goto 13
8  goto 10
9
10 X10 := mem[X8 - 0x18, el]:u64
11 call @free with return 13
12
13 X10 := mem[X8 - 0x18, el]:u64
14 call @free with return 16
15
16 call X1 with noreturn
```

Listing 4 Pretty printed BIL of Listing 2.

```
1  sub good(good_result)
2
3  X10 := 0x2A
4  call @malloc with return 5
5  mem := mem with [X8 - 0x18, el]:u64 <- X10
6
6  X15 := mem[X3 - 0x7C8, el]:u32
7  when (X15 = 0) goto 13
8  goto 10
9
10 X10 := mem[X8 - 0x18, el]:u64
11 call @free with return 16
12
13 X10 := mem[X8 - 0x18, el]:u64
14 call @free with return 16
15
16 call X1 with noreturn
```

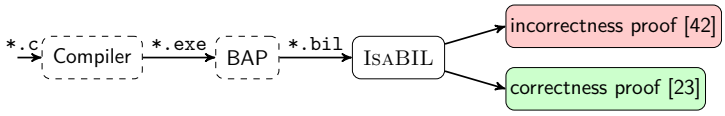


Figure 1 Translation of C source programs to an Isabelle/HOL proof.

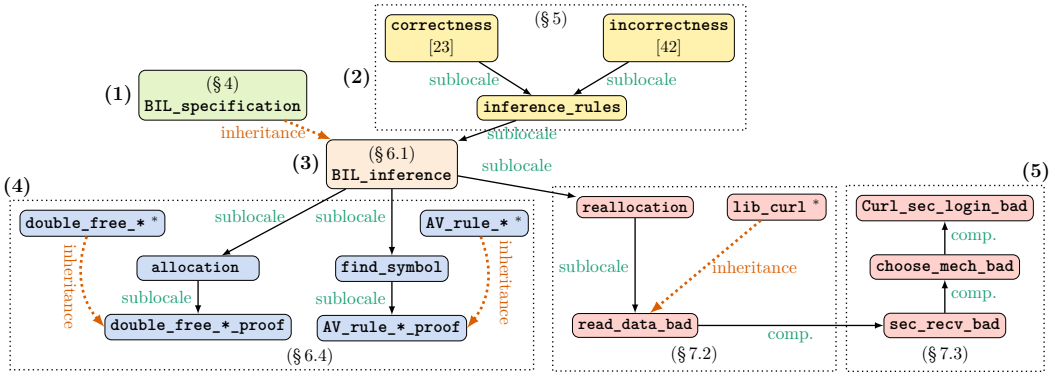


Figure 2 Structure of IsaBIL locales: theories indicated by * are auto-generated from a given `*.bil` file following the translation in Figure 1.

The focus of our work is to analyse such programs *without* relying on any high-level language semantics. Instead, we aim to develop a technique for reasoning about programs (e.g., those in Listings 1 and 2) at a lower level of abstraction by first compiling the programs then, lifting the resulting binaries to an intermediate representation (BIL). Although such analysis contains more detail, the analysis reflects the behaviour of the *actual* executable, thus is more accurate (e.g., does not require any additional compiler correctness assumptions).

2.2 Workflow and Mechanisation

Our overall workflow is given in Figure 1, where the dashed steps represent existing work. The first step to proving properties about our example program is to use BAP to generate a BIL representation of a binary executable (see Figure 1). For our example program in Listings 1 and 2, the BIL equivalent (expressed in BIR format) are given in Listings 3 and 4, respectively. BIR (Binary Intermediate Representation) is a structured refinement of BIL that significantly enhances human readability by presenting sequences of BIL as a

single instruction. Take for example the BIR instruction `call @free with return 18` on line 16 in Listings 3 and 4. This represents two BIL statements, a jump to the address of the symbol "free", and the assignment of the return value 18 to the return pointer. Note that IsaBIL takes as input BAP's *abstract data type* format (see § 4), which we refer to as BIL_{ADT}, as opposed to the human-readable forms shown in Listings 3 and 4. This makes the BIL instructions simpler to parse when generating the corresponding Isabelle/HOL object (discussed in more detail below).

Extensions to Isabelle/HOL's logic [15, 5, 26, 18] often make use of Isabelle/HOL's *locale* system [27, 3], which provides convenient modules for building parametric theories. Locales are aimed at supporting the use of local assumptions and definitions for collections of similar theories, defined in terms of fixed variables and definitions. We refer to the fixed variables as parameters which must satisfy the local assumptions. For a locale $\mathcal{L}$, we give its definition as $\mathcal{L} = (p_1, p_2, \dots)$ where p_1, p_2 are the parameters of the locale. Each parameter may be any type derived from a set or a relation. From these variables, assumptions and definitions we can derive general theorems within the context of a locale. These theorems can be exported to the current proof context by *instantiation*, in which we provide values to one or all of the parameters and verify the fixed assumptions of these parameters. This effectively gives us the general theorems contained within each locale for free for a specific instantiation. Locales can be extended through *inheritance* or *interpretation*, enabling reusability and specialisation by introducing and refining a locale's parameters. Inheritance operates hierarchically, directly propagating the context of a parent locale to a child locale. This transfer includes general theorems, assumptions, definitions and fixed variables. Interpretation is compositional, associating a locale or definition with another locale. During interpretation, the assumptions of the parent locale must be satisfied as part of the mapping process. A locale that interprets another locale is referred to as a sublocale, denoted $\mathcal{L} \subseteq \mathcal{L}'$, where locale $\mathcal{L}'$ is the parent of $\mathcal{L}$. For IsaBIL, they provide a uniform system for representing both the program, and the underlying reasoning framework.

The overall structure of the IsaBIL development is given in Figure 2, and comprises four main parts (which highlight our four main contributions). Overall, we have: **(1)** a locale (see § 4), representing a complete formalisation of the BIL specification from § 3, **(2)** a set of locales (see § 5) that encode Hoare Logic (to verify correctness) and Incorrectness Logic (as developed by O'Hearn), **(3)** a locale (see § 6.1) that combines formalisation of the BIL specification with (in)correctness logics, **(4)** a set of locales (see § 6.4) that apply the resulting verification framework to prove correctness of a number of examples from the literature and **(5)** a set of locales (see § 7.2 and 7.3) that demonstrate scalability of the approach. In particular, § 7.2 demonstrates the use of subroutines to enable one to focus on key functions of interest, even when starting with large BIL files (> 100k LoC). Then § 7.3 describes how verified subroutines can be used as subcomponents to prove (in)correctness of a larger system.

Our framework is highly flexible and extensible. For instance, the **`BIL_inference`** locale combines the **`BIL_specification`** and **`inference_rules`** locales. In case one wishes to change the specification (for a different operational semantics), or use a different set of inference rules (to perform a type of analysis), one can simply change the specification and/or inference rule locales. Similarly, a locale can be extended with additional semantic features: our proofs extend the **`BIL_inference`** locale with an **`allocation`** locale (that models memory allocation) and a **`find_symbol`** locale (that tracks a symbol table) to enable reasoning about different program features.

Our encoding of BIL (§ 4) is the *first* full mechanisation of BIL in Isabelle/HOL. This mechanisation is a standalone contribution and can be used by others outside of our reasoning framework. Our encoding uncovers some inconsistencies in the official specification, which are addressed by the mechanisation (see § 3). Moreover, given that BIL is an intermediate language for many different architectures, future versions could be further optimised to a particular architecture. As an example, we present optimisations for RISC-V (see § 6.2.3), which is the architecture that we focus on.

To maximise flexibility, we provide an encoding of the inference rules parameterised by an operational semantics, i.e., an *instance* of the correctness and incorrectness rules can be generated for *any* operational semantics. This instance automatically inherits all of the rules of Hoare and Incorrectness logic that we prove generically in the `correctness` and `incorrectness` locales (§ 5).

The BIL semantics from § 4 and the inference rules from § 5 are combined to form a new locale, `BIL_inference`, which provides facilities to verify BIL programs. Then, as discussed above, to verify particular types of examples, we extend the locales with models of allocation and symbols to verify particular sets of examples. These proofs represent the *first* proofs of both correctness and incorrectness for BIL programs in Isabelle/HOL. As we shall see in § 3, the BIL semantics is highly detailed, with the reductions related to even single load command potentially splitting into a large number of *small-step* transitions. This has the potential to increase the verification complexity making the proofs of even small programs intractable. To address this, we use BIL's big-step semantics (§ 6.1) and introduce a number of high-level lemmas that allow one to discharge such proofs generically. These lemmas are reusable across all of the examples that we verify.

Our proofs cover key examples from the JSF coding standards [32] and the CWE database[2]. This task is greatly simplified using IsaBIL's parser, written in Isabelle/ML [50], that *automatically generates* a locale corresponding to a BIL program written in BIL_{ADT} format. Note that Isabelle isolates Isabelle/ML programming, and hence our parser cannot interfere with soundness of Isabelle's core logic engine. The generated locale is then combined with `BIL_inference` to provide a context in which (in)correctness proofs can be carried out.

3 BIL Syntax and Semantics

Before proceeding with the presentation of our proofs, it is essential to acquire a basic understanding of BIL. This section serves as a succinct introduction to BIL, emphasizing its type system, memory model and operational semantics. We encourage interested readers to explore the BIL manual [8].

3.1 Syntax

The basic syntax of BIL programs is given in Figure 3.

Statements. A BIL statement may assign an expression e to a variable var, transfer control to a given address e, interrupt the CPU with a given interrupt num, be an instruction with unknown semantics, a while loop, or an if-then-else conditional (with an optional else clause). Note that each BIL statement (including a compound statement) corresponds to a single machine instruction, thus side effects such as setting a status register must be captured in the statement definition (see § 3.4).

Statements, where str denotes a string (of type $string$) and $i \in \mathbb{Z}$

$$\begin{aligned} stmt \ni s ::= {} & var := e \mid \textbf{jmp}\ e \mid \textbf{cpuexn}(i) \mid \textbf{special}(str) \mid \textbf{while}\ (e)\ seq \\ & \mid \textbf{if}\ (e)\ seq \mid \textbf{if}\ (e)\ seq_1\ \textbf{else}\ seq_2 \\ bil \ni seq ::= {} & stmt^* \end{aligned}$$

Expressions, where id denotes a variable name literal (of type $string$)

$$\begin{aligned} exp \ni e ::= {} & v \mid var \mid e_1[e_2, ed] : sz \mid e_1\ \textbf{with}\ [e_2, ed] : sz \leftarrow e_3 \mid e_1\ bop\ e_2 \mid uop\ e \\ & \mid cast : sz[e] \mid \textbf{let}\ var = e_1\ \textbf{in}\ e_2 \mid \textbf{ite}\ e_1\ e_2\ e_3 \mid e_1\ @\ e_2 \mid \textbf{extract}: sz_1 : sz_2[e] \\ bop ::= {} & aop \mid lop \qquad var ::= id : type \qquad endian \ni ed ::= \textbf{el} \mid \textbf{be} \\ cast ::= {} & \textbf{low} \mid \textbf{high} \mid \textbf{signed} \mid \textbf{unsigned} \end{aligned}$$

Types and Values, where $sz, nat \in \mathbb{N}$

$$\begin{aligned} type \ni t ::= {} & \textbf{imm}\langle sz\rangle \mid \textbf{mem}\langle sz_{addr},\ sz_{val}\rangle \qquad\qquad word \ni w ::= nat :: sz \\ value \ni v ::= {} & w \mid v[w \leftarrow v',\ sz_{val}] \mid \textbf{unknown}[str] : t \end{aligned}$$

Figure 3 BIL syntax and types.

Expressions. Memory loads and stores are represented at the expression level as $e_1[e_2, ed] : sz$ for loads, and e_1 **with** $[e_2, ed] : sz \leftarrow e_3$ for stores. In these expressions, the evaluation of e_1 represents the target storage being accessed, while the evaluation of e_2 specifies the address. Since a memory operation can span multiple addresses, both the endianess (ed) and the size (sz) of the operation are provided. For stores, the evaluated value of e_3 is written to memory at the given address. Binary operations now differentiate arithmetic (aop) and logical (lop) operations. Expressions in BIL also include casts, let bindings, if-then-else expressions, concatenations and extractions. The extraction operation, denoted as $\textbf{extract}: sz_1 : sz_2[e]$, first evaluates the expression e to a word w and then extracts a slice from w using BIL's existing extraction mechanism. Specifically, $\textbf{ext}\ w \backsim \textbf{hi} : sz_1 \backsim \textbf{lo} : sz_2$ represents the bits of w from bit sz_1 to bit sz_2. Thus, for example, $\textbf{ext}\ 01001011 \backsim \textbf{hi} : 5 \backsim \textbf{lo} : 2$ is 0010. Expressions are side-effect free and are evaluated wrt a state $\Delta : var \to val$ mapping variables to values.

Types and Values. The highly expressible nature of BIL is due to its type system, which defines two distinct types for its values, the irreducible subset of a BIL expression. These types are $\textbf{imm}\langle sz\rangle$ (describing an *immediate type* of size sz) and $\textbf{mem}\langle sz_{addr},\ sz_{val}\rangle$ (describing a *storage type* with address size sz_{addr} and value size sz_{val}). A value can be bound to one of the three possibilities. **(1)** A machine word (w). **(2)** An abstract storage ($v[w \leftarrow v',\ sz_{val}]$), which allows us to define a memory as a chain of mappings, where $[w \leftarrow v', sz_{val}]$ defines a single mapping from an address (w) to a value (v') of size sz_{val}. Here, v could either be the root of the chain (in which case it is $\textbf{unknown}[str] : t$) or another storage. Technically, v could also be a word, but we disallow this by introducing a typing context (see below). Similarly, v' could be a storage according to the grammar, but this possibility is also eliminated by the typing context. When v or v' is an unknown value, we require that type of v is **mem** and the type of v' is **imm**. **(3)** An "unknown" value ($\textbf{unknown}[str] : t$), which is obtained from the evaluation of a BIL expression holding some string information str of some type t.

The type of a value can be obtained using the function `type`, which is define as follows:

$$\begin{aligned} \texttt{type}(nat :: sz) &= \textbf{imm}\langle sz\rangle \\ \texttt{type}(v[(nat :: sz_{addr}) \leftarrow v',\ sz_{val}]) &= \textbf{mem}\langle sz_{addr},\ sz_{val}\rangle \\ \texttt{type}(\textbf{unknown}[str] : t) &= t \end{aligned}$$

In [21] we show the lifted BIL equivalent of a RISC-V program using the syntax presented in this section.

3.2 Typing Rules and Typing Context

BIL's type system facilitates the need for typing rules to ensure type correctness. At the lowest level, the rules ensure that types are correctly defined with the predicate t **is ok**. The type of variables must be tracked during symbolic execution to ensure that they do not change and only values of the correct type can be assigned. This is achieved via a *typing context*, defined as $\Gamma ::= var^*$, where type correctness of Γ is expressed by overloading the predicate Γ **is ok**. The rules for t **is ok** and Γ **is ok** are given below:

$$\frac{sz > 0}{\textbf{imm}\langle sz\rangle \textbf{ is ok}}\ (\textsc{twf_imm}) \qquad \frac{sz_{addr} > 0 \qquad sz_{val} > 0}{\textbf{mem}\langle sz_{addr},\ sz_{val}\rangle \textbf{ is ok}}\ (\textsc{twf_mem}) \qquad \frac{}{[] \textbf{ is ok}}\ (\textsc{tg_nil})$$

$$\frac{str \notin dom(\Gamma) \qquad t \textbf{ is ok} \qquad \Gamma \textbf{ is ok}}{(str : t) \# \Gamma \textbf{ is ok}}\ (\textsc{tg_cons})$$

These are extended to expressions, then to the level of statements (see [8] for details). As part of our formalisation, we discover a missing typing rule for empty sequences, which would otherwise be needed to ensure the sequencing rules are type correct. We have created a pull request in the BAP repositories for this issue (see `https://github.com/BinaryAnalysisPlatform/bap/pull/1588`), which has now been merged.

3.3 Expression Semantics

Expression evaluation requires the repeated application of small-step semantics rules until the expression is reduced to a value. An expression step in BIL is formally expressed with the $\Delta \vdash e \rightsquigarrow e'$, where multiple steps can be reduced using the reflexive transitive closure $\Delta \vdash e \rightsquigarrow^* e'$. Recall that loads and stores occur at the expression level, and are sensitive to endian orderings. We provide the load rules here, and refer the interested reader to the BIL manual [8] for further details.

A store instruction (semantics not shown) targets a single unit of addressable memory, typically 8-bits in size. When a storage operation exceeds this size, it is converted into a sequence of 8-bit stores, organised in big-endian order. The reduction rules for load operations is given in Figure 4. First, all sub-expressions of a memory object are reduced to values using the rules LOAD_STEP_ADDR and LOAD_STEP_MEM. Then, the resulting object is recursively deconstructed using the LOAD_WORD_EL and LOAD_WORD_BE rules, depending on the endian. This process continues until one of LOAD_BYTE, LOAD_UN_MEM or LOAD_UN_ADDR is used. LOAD_BYTE reduces the expression to the value, v', when the memory object is a storage of an immediate (known) value, w. LOAD_UN_MEM and LOAD_UN_ADDR both reduce the expression to an unknown value when the memory or address being loaded is unknown, respectively.

▶ **Example 1.** Consider the instruction, `X10 := mem[X8-0x18, el]:u64`, from line 12 in Listing 3, which performs a little-endian load of a 64-bit word from address `X8-0x18` in the variable `mem`. We first reduce the address expression, `X8-0x18`, to a value using LOAD_STEP_ADDR, next we apply LOAD_STEP_MEM to read the value stored in `mem`. Next,

$$\frac{(var, v) \in \Delta}{\Delta \vdash var \rightsquigarrow v}\ (\textsc{var_in})$$

$$\frac{\Delta \vdash e_2 \rightsquigarrow e_2'}{\Delta \vdash e_1[e_2, ed] : sz \rightsquigarrow e_1[e_2', ed] : sz}\ (\textsc{load_step_addr})$$

$$\frac{\Delta \vdash e_1 \rightsquigarrow e_1'}{\Delta \vdash e_1[v_2, ed] : sz \rightsquigarrow e_1'[v_2, ed] : sz}\ (\textsc{load_step_mem})$$

$$\frac{}{\Delta \vdash v[w \leftarrow v',\ sz][w, ed] : sz \rightsquigarrow v'}\ (\textsc{load_byte})$$

$$\frac{w_1 \neq w_2}{\Delta \vdash v[w_1 \leftarrow v',\ sz][w_2, ed] : sz \rightsquigarrow v[w_2, ed] : sz}\ (\textsc{load_byte_from_next})$$

$$\frac{}{\Delta \vdash (\mathbf{unknown}[str] : t)[v, ed] : sz \rightsquigarrow \mathbf{unknown}[str] : \mathbf{imm}\langle sz\rangle}\ (\textsc{load_un_mem})$$

$$\frac{}{\Delta \vdash v[w_1 \leftarrow v',\ sz'][\mathbf{unknown}[str] : t, ed] : sz \rightsquigarrow \mathbf{unknown}[str] : \mathbf{imm}\langle sz\rangle}\ (\textsc{load_un_addr})$$

$$\frac{sz > sz_{mem} \qquad \mathtt{succ}(w) = w' \qquad \mathtt{type}(v) = \mathbf{mem}\langle sz_{addr},\ sz_{mem}\rangle}{\Delta \vdash v[w, \mathbf{be}] : sz \rightsquigarrow v[w, \mathbf{be}] : sz_{mem} \mathbin{@} v[w', \mathbf{be}] : (sz - sz_{mem})}\ (\textsc{load_word_be})$$

$$\frac{sz > sz_{mem} \qquad \mathtt{succ}(w) = w' \qquad \mathtt{type}(v) = \mathbf{mem}\langle sz_{addr},\ sz_{mem}\rangle}{\Delta \vdash v[w, \mathbf{el}] : sz \rightsquigarrow v[w', \mathbf{el}] : (sz - sz_{mem}) \mathbin{@} v[w, \mathbf{el}] : sz_{mem}}\ (\textsc{load_word_el})$$

Figure 4 Reduction rules for Load.

we look up the value of `mem` in Δ in VAR_IN, which reduces `mem` to the corresponding storage. Next, we apply LOAD_WORD_EL to break down the 64-bit load into 8 separate loads in little-endian format. Each of the 8 byte-sized loads are individually read from memory using LOAD_BYTE and LOAD_BYTE_FROM_NEXT. Finally, we apply BIL's concatenation rule (not shown) to join each byte into a single 64-bit word.

IsaBIL Extensions. The previous sections present a summary of BIL's semantics from the BIL manual [8] with some minor corrections. In this section, we present extensions to BIL's big-step expression evaluation semantics. The reduction of expressions, e.g., in load and store instructions, results in a significant proof burden. To make verification tractable, IsaBIL defines additional big-step expression evaluation rules which are derived from a combination of BIL's existing small-step and big-step expression semantics. For example, the load expression in Example 1 requires the repeated application of over 150 small-step rules. Splitting the 64-bit read into individual reads in Example 1 (from LOAD_WORD_EL onwards) and the concatenation of the result can be combined into a single big-step rule. We first provide some syntactic sugar for a storage whose size is a multiple of 8.

The definition uses the function $\mathtt{succ}(w)$, which retrieves the successor of the word w such that $\mathtt{succ}(w) ::= w + 1$. Furthermore, we use $\mathtt{storage}_{en}\ (v, w, v', N)$ to refer to a contiguous storage of the value v' of size $N \in \{8i \mid i \in \mathbb{N}\}$ (i.e., N is a multiple of 8) at address w on the storage v:

$$\mathtt{storage}_{\mathbf{el}}\ (v, w, v', N) \doteq \begin{cases} \mathtt{storage}_{\mathbf{el}} \begin{pmatrix} v[w \leftarrow \mathbf{ext}\ v' \backsim \mathbf{hi} : 7 \backsim \mathbf{lo} : 0,\ 8], \\ \mathtt{succ}(w), \mathbf{ext}\ v' \backsim \mathbf{hi} : N - 1 \backsim \mathbf{lo} : 8, N - 8 \end{pmatrix} & \textbf{if } \boldsymbol{N > 8} \\ v[w \leftarrow v',\ 8] & \textbf{otherwise} \end{cases}$$

$$\mathtt{storage}_{\mathbf{be}}\ (v, w, v', N) \doteq \begin{cases} \mathtt{storage}_{\mathbf{be}} \begin{pmatrix} v[w \leftarrow \mathbf{ext}\ v' \backsim \mathbf{hi} : N - 1 \backsim \mathbf{lo} : N - 8,\ 8], \\ \mathtt{succ}(w), \mathbf{ext}\ v' \backsim \mathbf{hi} : N - 9 \backsim \mathbf{lo} : 0, N - 8 \end{pmatrix} & \textbf{if } \boldsymbol{N > 8} \\ v[w \leftarrow v',\ 8] & \textbf{otherwise} \end{cases}$$

For example, suppose B_1 and B_2 are two 8-bit words. We have:

$$\mathtt{storage}_{\mathbf{el}}\ (v, w, B_1B_2, 16) = v[w \leftarrow B_2, 8][\mathtt{succ}(w) \leftarrow B_1, 8]$$
$$\mathtt{storage}_{\mathbf{be}}\ (v, w, B_1B_2, 16) = v[w \leftarrow B_1, 8][\mathtt{succ}(w) \leftarrow B_2, 8]$$

We give a subset of these rules for little endian 64-bit load and store operations below.

$$\frac{}{\Delta \vdash \texttt{storage}_{\mathbf{el}}\,(v, w_2, w_1, 64)\,[w_2, 64] : \mathbf{el} \rightsquigarrow^* w_1}\ (\textsc{refl_load_el_word64})$$

$$\frac{}{\Delta \vdash v\ \mathbf{with}\ [w_2, \mathbf{el}] : 64 \leftarrow w_1 \rightsquigarrow^* \texttt{storage}_{\mathbf{el}}\,(v, w_2, w_1, 64)}\ (\textsc{refl_store_el_word64})$$

▶ **Example 2.** Consider the load from Example 1. The first few steps proceed as before. However, instead of applying LOAD_WORD_EL, we can directly apply REFL_LOAD_EL_WORD64 to obtain the corresponding value $\Delta \vdash (\texttt{storage}_{\mathbf{el}}\,(v, w, v', 64))[w, el] : u64 \rightsquigarrow^* v'$.

All big-step rules from this section, such as REFL_LOAD_EL_WORD64 have been verified in Isabelle wrt to the small step semantics.

3.4 Statement semantics

In this section, we describe the operational semantics for evaluating statements. We also provide a correction to the sequencing rule from the manual [4]. Although each statement induces local control flow, in BIL, they are assumed to execute to completion in a single step.

The semantics of a statement is defined by $(\Delta,\ pc) \vdash seq \rightsquigarrow (\Delta',\ pc')$, which executes the *bil* statement *seq* from the variable store Δ to generate a new variable store Δ' and program counter pc'. The MOVE statement (see below) modifies Δ by generating a new variable binding, and JMP (not shown) affects program counter.

Example rules for MOVE (aka assignment) and IF_TRUE (for branching on a true guard are given below). In the move rule, the given expression e is evaluated to a value using the big-step semantics, and the value for the given variable var is updated in the variable state. In the IF_TRUE rule, the guard is evaluated (to *true*) then the statement seq_1 is executed. The sequencing rules describe execution of a list of statements to completion. The sequencing rules in manual [4] do not provide a means to reduce multiple statements in a single transition[3].

$$\frac{\Delta \vdash e \rightsquigarrow^* v}{(\Delta,\ pc) \vdash var := e \rightsquigarrow (\Delta(var \mapsto v),\ pc)}\ (\textsc{move})$$

$$\frac{\Delta \vdash e \rightsquigarrow^* true \qquad (\Delta, pc) \vdash seq_1 \rightsquigarrow (\Delta', pc')}{(\Delta,\ pc) \vdash \mathbf{if}\ (e)\ seq_1\ \mathbf{else}\ seq_2 \rightsquigarrow (\Delta',\ pc')}\ (\textsc{if_true})$$

$$\frac{(\Delta_1,\ pc_1) \vdash s_1 \rightsquigarrow (\Delta_2,\ pc_2) \qquad (\Delta, pc_2) \vdash s_2 \ldots s_n \rightsquigarrow (\Delta_3, pc_3)}{(\Delta_1, pc_1) \vdash s_1 s_2 \ldots s_n \rightsquigarrow (\Delta_3, pc_3)}\ (\textsc{seq_rec})$$

$$\frac{}{(\Delta, pc) \vdash \varepsilon \rightsquigarrow (\Delta, pc)}\ (\textsc{seq_nil})$$

▶ **Example 3.** Consider the load expression from Example 1, which appears on the right-hand side of the assignment statement given below:

$$(\Delta, (12 :: 64) + (1 :: 64)) \vdash \{\texttt{X10 := mem[X8-0x18, el]:u64}\} \rightsquigarrow (\Delta(X10 \mapsto v'), (13 :: 64))$$

To apply the statement semantics, we start by deconstructing it using SEQ_REC, which results in two sub-steps. For the first, we apply MOVE and the steps in Example 1 to reduce the expression `mem[X8-0x18,el]:u64` to a value, v'. Since the original sequence only contained a single statement, our recursive sequence (i.e., the second premise of SEQ_REC) is empty, which can be trivially reduced using SEQ_NIL.

[3] We have created a Git pull request in the BAP repositories for this issue (`https://github.com/BinaryAnalysisPlatform/bil/pull/12`.

3.5 BIL Step Relation

A machine instruction in BIL is represented by a named tuple, $insn = (\!|\ \mathbf{addr},\ \mathbf{size},\ \mathbf{code}\ |\!)$ (implemented later as an Isabelle `record`). Here **addr**, refers to the instruction's address in memory; **size**, the size of the instruction; and **code**, the semantics of the program represented by BIL statements. Instructions operate over the BIL machine state, (Δ, pc, mem), where Δ is the variable store, pc is the program counter and mem is a variable that denotes the memory. To decode the current instruction from a machine state, we assume an uninterpreted $\mapsto$ function, referred to as the *decode predicate*, which maps a machine state, (Δ, pc, mem), to the corresponding instruction, $(\!|\ \mathbf{addr},\ \mathbf{size},\ \mathbf{code}\ |\!)$. The BIL step relation $(\Delta, pc, mem) \rightsquigarrow (\Delta', pc', mem')$ defines the execution of a single BIL statement:

$$\frac{(\Delta, pc, mem) \mapsto (\!|\ \mathbf{addr} = pc, \mathbf{size} = z, \mathbf{code} = bil\ |\!) \qquad (\Delta, pc + z) \vdash bil \rightsquigarrow (\Delta', pc')}{(\Delta, pc, mem) \rightsquigarrow (\Delta', pc', mem)}\ (\textsc{step_prog})$$

▶ **Example 4.** Consider the move statement from Example 3, which resides at the program address 12 and being of one-byte size. This is expressed as the BIL instruction below:

$$(\!|\ \mathbf{addr} = 12 :: 64, \mathbf{size} = 1 :: 64, \mathbf{code} = \{\texttt{X10 := mem[X8-0x18, el]:u64}\}\ |\!)$$

The instruction is obtained by decoding $(\mapsto)$ a machine state of the form $(\Delta, 12 :: 64, mem)$ where $X8, mem \in \texttt{dom}(\Delta)$. By using decode we can structure a step proof for the machine state as follows:

$$(\Delta, 12 :: 64, mem) \rightsquigarrow (\Delta(X10 \mapsto v'), 13 :: 64, mem)$$

By applying the STEP_PROG rule for programs, we are left with a reduction $(\Delta, (12 :: 64) + (1 :: 64)) \vdash \{\texttt{X10 := mem[X8-0x18, el]:u64}\} \rightsquigarrow (\Delta(X10 \mapsto v'), (13 :: 64))$, which we derived in Example 3.

4 Mechanisation

In this section, we describe the IsaBIL mechanisation, which utilises extensible *locales* to maximize reusability. In § 4.1, we describe the generic `BIL_specification` locale that formalises BIL's syntax and semantics, and in § 4.2 we describe the IsaBIL translation tool that automatically generates locales from BIL programs.

4.1 The `BIL_specification` Locale

IsaBIL's `BIL_specification` locale (see Figure 2) provides a complete mechanisation of BIL's syntax and semantics[4]. We encode BIL's operational rules (§ 3.5) as Isabelle's built-in *inductive predicates*, which allow one to formalise the structure of the syntax from Figure 3.

The `BIL_specification` $= (\mapsto)$ locale provides a single parameter: the *decode* predicate $(\mapsto)$, which will be uniquely instantiated for each binary that we wish to verify.

We provide rules for both *introduction* (which are used to introduce compound statements such as **if-then-else** to the proof goal) as well as *elimination* (which are used to eliminate compound statements from the proof assumptions) directly within the `BIL_specification` locale. For example, introduction and elimination rules for the STEP_PROG rule in § 3.5 which formalises a single step of the program are given below:

[4] As outlined in § 3 and provided in full in [21].

```
lemma step_progI[intro]:
assumes ‹(Δ, pc, mem) ↦
            ⦇addr = pc, size = sz, code =
     seq⦈›
    and ‹(Δ, pc + sz) ⊢ seq ⇝ (Δ', pc')›
  shows ‹(Δ, pc, mem) ⇝ (Δ', pc', mem)›
proof ...
```

```
lemma step_progE[elim]:
assumes ‹(Δ, pc, mem) ⇝ (Δ', pc', mem)›
    and ‹(Δ, pc, mem) ↦
           ⦇addr = pc, size = sz, code =
     seq⦈›
obtains ‹(Δ, pc + sz) ⊢ seq ⇝ (Δ', pc')›
proof ...
```

We prove introduction and elimination lemmas in this way for all of the rules in the official BIL specification [6] with respect to our changes (§ 3). In total, the `BIL_specification` locale consists of over 500 lemmas and rules.

4.2 The Program Locale

Recall that our workflow (Figure 1) proceeds by *Step 1*: compiling the given program; *Step 2*: feeding the resulting assembly into BAP to generate the corresponding BIL_{ADT}; and *Step 3*: using IsaBIL to automatically generate an Isabelle/HOL locale for the given BIL_{ADT} input. Note that if a binary is distributed without source code, then Step 1 could be skipped and the given assembly could be fed directly into BAP to generate the BIL_{ADT}.

The locale for a program prog is generated using IsaBIL's translation tool, which is written in Isabelle/ML and is invoked using the custom Isabelle/HOL commands `BIL` or `BIL_file` for inline or external BIL in BIL_{ADT} format, respectively. Both commands take as an input, a name for the locale and either a BIL string (for `BIL`) or a filename (for `BIL_file`). Within this locale, we automatically generate a decode predicate, $\mapsto_{\text{prog}}$, that maps each machine state (Δ, pc, mem) to an instruction.

Note that, internally, Isabelle maintains the BIL_{ADT} format, however, we choose to represent it using `class` locales [50, 3, 41] to maintain human-readable syntax. For instance, without these syntax classes, expressing a direct jump to the fixed program address 3076 would require writing `RAX := Val(Imm(Word(3076,64)))`. By instantiating the word syntax (*nat* :: *sz*, see Figure 3), we can simplify the expression to the syntax `RAX := (3076 :: 64)`.

In addition to $\mapsto_{\text{prog}}$, a program's locale defines *Step 1*: an *address set*, `addr_set` : $\mathbb{P}(word)$ (defining the set of addresses that have corresponding instructions) and *Step 2*: a *symbol table*, `sym_table` : $string \rightarrow word$ (mapping the binary's symbols, e.g. main, memcpy, free, as string literals to raw addresses).

Both `addr_set` and `sym_table` capture additional information about a binary that can be used later in a proof. For example, the `addr_set` is useful for determining whether the program counter points to a valid address and is particularly important for validating the correctness of indirect jumps. Many binary proofs verify that execution from some entry point, such as the `main` function, is either correct or incorrect with respect to some property. Symbols typically represent entry points to functions and although the address of an entrypoint may differ between binaries, the symbol will remain constant. Hence, referring to positions within the binary using symbol names, stored in the `sym_table` rather than program addresses, is often more convenient. Further details regarding the extraction of the `addr_set` and `sym_table` from a binary can be found in § 6.3.

▶ **Example 5.** Consider the `Binary_A` program below, written in BIL_{ADT}, which corresponds to the BIL instruction `X8 := X2 + 32`. We assume the program has been compiled for RISC-V; for readability, BIL_{ADT} provides the original assembly at Example 5. The program locale corresponding to Example 5 starts on Example 5, which fixes the corresponding decode predicate.

```
BIL ‹
105dc: <test>
105dc: addi s0, sp, 32
(Move(Var("X8",Imm(64)),PLUS(Var("X2",Imm(64)),Int(32,64))))
› defining Binary_A

locale Binary_A
    fixes decode :: ‹(var ⇒ val option) × word × var ⇒ insn ⇒ bool›
    assumes decode_105dc: ‹(Δ, 0x105dc :: 64, mem) ↦prog
                ⦇ addr = 0x105dc :: 64, size = 4, bil = [X8 := X2 + (32 ::
     64)] ⦈›
begin
definition addr_set :: ‹word set› where ‹addr_set = {(0x105dc :: 64)}›
definition sym_table :: ‹str ⇀ word› where ‹sym_table = ["test" ↦ (0x105dc
    :: 64)]›
end
```

Externally linked binaries. Binaries often include links to external code, typically in the form of function calls. Accurately modeling these external calls is crucial for understanding the behavior of the original binary. IsaBIL provides two methods for handling externally linked code, depending on the availability of that external code. If the external binary is available, an Isabelle/HOL locale can be generated for the external binary, which is then combined with the existing binary locale through inheritance. If the external binary is not available, we can make assumptions about its behavior and define an approximate implementation using locale assumptions. We give an example for both cases in [21].

5 (In)correctness

In this section, we present the IsaBIL encoding of Hoare and Incorrectness logic (see Figure 2). Our locale-based encoding combines both into a single proof system **`inference_rules`** locale, where we verify the AGREEMENT and DENIAL lemmas (see [42]).

Hoare and O'Hearn triples define (in)correctness of a command c over a pre-state σ satisfying the predicate P, resulting in a post-state τ satisfying the predicate Q. In the locale, we assume the existence of a big-step transition relation $(c, \sigma) \Rightarrow \tau$ defining the execution of c from state σ to termination, resulting in state τ. Thus, we have the well-known under- and over-approximating rules:

$$\text{(HOARE)}\ \frac{\forall \sigma, \tau.\ ((c,\sigma) \Rightarrow \tau) \implies (P(\sigma) \implies Q(\tau))}{\{P\}c\{Q\}} \qquad \text{(OHEARN)}\ \frac{\forall \tau.\ Q(\tau) \implies \exists \sigma.\ P(\sigma) \wedge ((c,\sigma) \Rightarrow \tau)}{[P]\,c\,[Q]}$$

Note that our definition of correctness follows partial correctness, meaning termination is not guaranteed. We encode Hoare and Incorrectness triples as locales **`correctness`** $= (\Rightarrow)$ and **`incorrectness`** $= (\Rightarrow)$, respectively, both of which are parameterised by $\Rightarrow$. Within the **`correctness`** locale, we verify standard Hoare logic rules, e.g.,

$$\text{(POST__CONJ__CORR)}\ \frac{\{P\}c\{Q_1\} \quad \{P\}c\{Q_2\}}{\{P\}c\{Q_1 \wedge Q_2\}} \qquad \text{(PRE__DISJ__CORR)}\ \frac{\{P_1\}c\{Q\} \quad \{P_2\}c\{Q\}}{\{P_1 \vee P_2\}c\{Q\}} \qquad \text{(STRENGTHEN__WEAKEN__CORR)}\ \frac{P' \implies P \quad \{P\}c\{Q\} \quad Q \implies Q'}{\{P'\}c\{Q'\}}$$

Similarly, within the **`incorrectness`** locale, we verify rules for Incorrectness logic, e.g.,

$$\text{(POST__DISJ__INCORR)}\ \frac{[P]\,c\,[Q_1] \quad [P]\,c\,[Q_2]}{[P]\,c\,[Q_1 \vee Q_2]} \qquad \text{(PRE__DISJ__INCORR)}\ \frac{[P_1]\,c\,[Q] \quad [P_2]\,c\,[Q]}{[P_1 \vee P_2]\,c\,[Q]} \qquad \text{(STRENGTHEN__WEAKEN__INCORR)}\ \frac{P \implies P' \quad [P]\,c\,[Q] \quad Q' \implies Q}{[P']\,c\,[Q']}$$

We then combine these to form a locale **inference_rules** $= (\Rightarrow)$, again parameterised by $\Rightarrow$. Within this combined locale, we are able to prove the AGREEMENT and DENIAL rules [42]:

$$\text{(AGREEMENT)}\quad \frac{[U]\,c\,[U'] \qquad U \implies O \qquad \{O\}c\{O'\}}{U' \implies O'} \qquad \text{(DENIAL)}\quad \frac{[U]\,c\,[U'] \qquad U \implies O \qquad \neg(U' \implies O')}{\neg(\{O\}c\{O'\})}$$

For proofs of incorrectness, it is typically not enough to simply state that if $[P]\,c\,[Q]$ holds for an error state Q then the program c is incorrect. A program cannot be incorrect if there is no valid state that satisfies Q and as such, we must ensure that Q is *non-trivial*, i.e., $\exists\sigma.\ Q(\sigma)$. Proving that Q is non-trivial can be difficult, as Q is often an over-approximation of the actual set of post-states (defined by some predicate Q') reachable from P. Therefore, proofs of incorrectness usually take the form $\exists Q'.\ [P]\,c\,[Q'] \wedge (\forall\sigma.\ Q'(\sigma) \implies Q(\sigma)) \wedge (\exists\sigma.\ Q'(\sigma))$. Note that this does not necessarily imply that $[P]\,c\,[Q] \wedge (\exists\sigma.\ Q(\sigma))$ since the implication is in the wrong direction to apply the STRENGTHEN_WEAKEN_INCORR rule.

6 Automation and Examples

In this section, we bring together the **BIL_specification** and the **inference_rules** locales to create a new locale, **BIL_inference** (see § 6.1), providing a proof environment for BIL programs. This in turn enables us to develop a large number of highly reusable proof automation procedures for verifying both correctness and incorrectness of BIL programs (see § 6.2). The final major component of IsaBIL automation is the BIL_{ADT} parser that automatically generates Isabelle/HOL locales for BIL programs (§ 6.3), providing a smooth and seamless pipeline from BAP to Isabelle/HOL proofs. We offer an overview of IsaBIL, discussing correctness and incorrectness proofs for a handful of examples.

6.1 The BIL_inference locale

BIL_inference interfaces with the **BIL_specification** and **inference_rules** locales (see Figure 2) and is later extended with instances of state models (e.g., **allocation** and **find_symbol**) to enable verification of generated BIL programs. Therefore, it is one of IsaBIL's most complex components.

Our first task is to define a big-step relation, $\xRightarrow[BIL]{}$, required by **inference_rules**, using the BIL step relation, $(\Delta, pc, mem) \rightsquigarrow (\Delta', pc', mem')$, defined in § 3.5. One option for defining $\xRightarrow[BIL]{}$ is to simply take the transitive closure of $\rightsquigarrow$. However, this would mean that the pre/postconditions that we define in our example would be predicates over the full variable store Δ, which is heavy-handed.

An alternative (which is the approach we take) is to define another transition relation $\xrightarrow[BIL]{}$ that abstracts $\rightsquigarrow$, and define $\xRightarrow[BIL]{}$ as the transitive closure of $\xrightarrow[BIL]{}$. As we shall see, this vastly increases the flexibility of our approach, which becomes **(1)** extensible, since $\xrightarrow[BIL]{}$ can be used to cover additional program features such as memory allocation (see § 6.4.1), and **(2)** more efficient since the components of Δ that are not needed for the proof can be ignored.Thus, assuming $\mathcal{C} = (\Delta, pc, mem)$, we have:

$$\text{(PROGRAM_BIG_STEP)}\quad \frac{\mathcal{C} = (_, pc, _) \qquad pc \in \texttt{addr_set} \qquad \mathcal{C} \rightsquigarrow \mathcal{C}' \qquad (\mathcal{C}, \sigma) \xrightarrow[BIL]{} \sigma' \qquad (\mathcal{C}', \sigma') \xRightarrow[BIL]{} \tau}{(\mathcal{C}, \sigma) \xRightarrow[BIL]{} \tau}(\mapsto)$$

$$\text{(PROGRAM_BIG_STEP_LAST)}\quad \frac{\mathcal{C} = (_, pc, _) \qquad \mathcal{C}' = (_, pc', _) \qquad pc \in \texttt{addr_set} \qquad pc' \notin \texttt{addr_set} \qquad \mathcal{C} \rightsquigarrow \mathcal{C}' \qquad (\mathcal{C}, \sigma) \xrightarrow[BIL]{} \tau}{(\mathcal{C}, \sigma) \xRightarrow[BIL]{} \tau}(\mapsto)$$

Here, we assume `addr_set` is the set of addresses that contain program instructions (see § 4.2), which will be later instantiated by a program's locale. A program may take a step as long as the current program counter corresponds to a program instruction. Note that the precise state model (i.e., type of σ, σ' and τ) that one uses in PROGRAM_BIG_STEP and PROGRAM_BIG_STEP_LAST will depend on the verification task at hand. If required, one could also take $\sigma = \Delta$ allowing the proofs to inspect the full variable state (at a low level of abstraction).

Overall, we obtain a locale **`BIL_inference`** $= (\mapsto, \xrightarrow[BIL]{}, \texttt{addr_set})$. This locale extends the **`BIL_specification`** (from § 4) from which we inherit the decode predicate, $\mapsto$. Then, we use $\xrightarrow[BIL]{}$ and `addr_set` to define $\Longrightarrow_{BIL}$ as above. This allows us to interpret the **`inference_rules`** as sublocale, instantiating the parameter $\Rightarrow$ to $\Longrightarrow_{BIL}$.

Within **`BIL_inference`**, we can prove several rules of Hoare and Incorrectness logic in terms of the BIL semantics. These proofs are trivial at this level, but are nevertheless critical for proof automation since they can be used by all of our examples [21].

Stuckness and Undefined Behaviour. A BIL program is considered stuck if it reaches a state where no transition rules apply. Programs that become stuck are deemed correct, as they do not terminate.

The most common cause of stuckness in BIL is an attempt to decode an instruction at an address that does not exist within the program (see § 3.5). This typically occurs when a jump targets a non-existent address, resulting in a post-state with no valid instruction to execute. However, the big step semantics treat this as valid program termination rather than an explicit error. If the prover wishes to ensure that the program terminates with an expected PC they may assert its value in an (in)correctness triple's post-state Q.

Stuckness may also arise due to type errors, such as assigning a variable a value of an incompatible type. These types of stuckness are prevented by ensuring BIL statements are type correct, which can be achieved using IsaBIL's automated type checker (see [21]). In our proofs, we verify type correctness when necessary to prevent stuckness. More details, along with an example, can be found in the BIL manual [6].

Many traditional sources of stuckness in program semantics, such as dereferencing invalid memory or reading from uninitialized registers, result in undefined behaviour (UB) for BIL. In BIL, UB is represented using the **unknown**[str] : t value. For example, if an uninitialized register is used:

```
1     mv a6, a5      # UB if a5 was never initialized
```

Since the value of `a5` (a 64 bit register) is unspecified, **unknown**[*str*] : **imm**⟨64⟩ will be set to `a6`. Evaluating BIL semantics that contain unknown values (such as `a6` + 5) will propagate further unknowns unless avoided by control flow structures such as if-then-else. However, this does not necessarily lead to stuckness, except in cases where the target of a dynamic jump is unknown (i.e. **jmp unknown**[*str*] : **imm**⟨64⟩). In such cases, the program cannot take a step as it does not know which program address to jump to. In (in)correctness logic, unknowns that do not lead to stuckness may still prevent the discharge of the post-state (Q). Unknown values in BIL can be resolved by explicitly specifying constraints on the program and pre-state as required.

6.2 Proof Automation

Instruction set architectures (ISAs), despite their large scale, are inherently finite, and compilers employ common patterns for optimization. Additionally, developers leverage reusable components such as *functions* and *gadgets* to perform common tasks. Consequently,

binaries often exhibit a significant degree of repetition. Once identified, this repetition can be verified without the need for human input. IsaBIL exploits this repetition to alleviate the burden associated with proof construction and verification. In this section, we outline the proof automation techniques employed by IsaBIL.

6.2.1 Eisbach

IsaBIL heavily leverages Eisbach [33], an Isabelle tactic framework for proof automation, to discharge proof goals automatically. We create Eisbach methods for symbolic execution (`sexc`) and type checking (`typec`) which utilise IsaBIL's *introduction* and *elimination* rules to solve both big- and small-step BIL statements, expressions and variables as well as proving type correctness. These methods employ a best-effort approach. If a case that cannot be solved automatically occurs, the methods will backtrack to a safe state and the partially completed proof context will be handed back to the human prover, whereby corrections or manual inputs can be made before resuming the automation (see [21] for details).

6.2.2 Human Readability

In assembly languages, registers are typically assigned specific names and types. However, in BIL, registers are represented as generic *variables* of type *var*. These variables are parameterised with names and types. For example, the RISC-V registers `R0-31` are represented by the variables ("R0-31" : **imm**⟨64⟩). While this representation offers flexibility, it can be verbose. It is common knowledge for RISC-V developers that R0-31 are registers capable of storing 64-bit words. To enhance program readability, we define the set of registers for `riscv64` [55] as the 64bit registers R0-31 : **imm**⟨64⟩. Additionally, we introduce syntax abbreviations for the 64-bit x86 architecture.

6.2.3 Architecture-Specific Proof Optimisations (for RISC-V)

Whilst IsaBIL provides sufficient granularity to handle many architectures out of the box, the speed and efficiency of proofs can be improved by tailoring optimizations to specific hardware architectures. This section describes the proof optimizations undertaken for RISC-V.

Instructions. Modern architectures comprise of large instruction sets. Proofs of programs can be optimised by verifying high-level lemmas corresponding to program steps of the most commonly used instructions directly in the `BIL_inference` locale. For example, IsaBIL provides step rules for the 32-bit (4-byte) RISC-V instructions `auipc`, `jalr` and `ld`.

$$\texttt{auipc}\ rd,\ imm = (\!|\ \mathbf{addr} = pc, \mathbf{size} = 4 :: 64, \mathbf{code} = \{rd := pc + imm\}\ |\!)$$

The `auipc` (Add Upper Immediate to Program Counter) instruction facilitates the computation of an absolute address. It achieves this by adding the immediate value *imm* to *pc*, storing the result in the destination register *rd*. `auipc` is commonly used to create jump targets, such as those for external functions.

$$\texttt{jalr}\ rd,\ rs1,\ \mathit{offset} = (\!|\ \mathbf{addr} = pc, \mathbf{size} = 4 :: 64, \mathbf{code} = \{rd := pc + 4;\ \mathbf{jmp}\ rs1 + \mathit{offset}\}\ |\!)$$

The `jalr` (Jump And Link Register) instruction performs an unconditional jump. It sets *pc* to the address stored in the source register $rs1$ with an optional immediate value *offset*. The address of the subsequent instruction is then stored in the destination register *rd*. This instruction is used for function calls, where it is important to preserve the return address.

$$\texttt{ld}\ rd,\ \mathit{offset}(rs1) = (\!|\ \mathbf{addr} = pc, \mathbf{size} = 4 :: 64, \mathbf{code} = \{rd := mem[rs1 + \mathit{offset}, \mathbf{el}] : 64\}\ |\!)$$

```
00000000000104a0 <malloc@plt>:
   104a0: 00002e17            auipc  t3,0x2
   104a4: b78e3e03            ld     t3,-1160(t3) # 12018 <malloc@GLIBC_2
    .27>
   104a8: 000e0367            jalr   t1,t3
   104ac: 00000013            nop

00000000000104b0 <free@plt>:
   104b0: 00002e17            auipc  t3,0x2
   104b4: b70e3e03            ld     t3,-1168(t3) # 12020 <free@GLIBC_2
    .27>
   104b8: 000e0367            jalr   t1,t3
   104bc: 00000013            nop
```

Figure 5 PLT stubs for free and malloc.

The `ld` (Load Double) instruction is used to load a 64-bit word, referred to as a double word, from memory. First, `ld` computes an address by adding the contents of the source register $rs1$ to the immediate *offset*. The value at this address in memory is then stored in the destination register rd. `ld` is the primary means of retrieving data of this size from memory.

Representing the instructions `auipc`, `jalr` and `ld` as the high-level program step lemmas (given below) allows the prover to otherwise skip the lengthy reduction for a program step. The rules for these instructions are provided below:

$$\frac{(\Delta, pc, mem) \mapsto \texttt{auipc}\ rd,\ imm}{(\Delta, pc, mem) \rightsquigarrow (\Delta(rd \mapsto pc + imm), pc + 4), mem)}\ (\text{AUIPC_LEMMA})$$

$$\frac{(\Delta, pc, mem) \mapsto \texttt{jalr}\ rd,\ rs1,\ offset \quad (rs1, addr) \in \Delta}{(\Delta, pc, mem) \rightsquigarrow (\Delta(rd \mapsto pc + 4), addr + offset, mem)}\ (\text{JALR_LEMMA})$$

$$\frac{(\Delta, pc, mem) \mapsto \texttt{ld}\ rd,\ offset(rs1) \quad (mem, v) \in \Delta \quad (rs1, addr) \in \Delta \quad \Delta \vdash v[addr + offset, \mathbf{el}] : 64 \rightsquigarrow^* w}{(\Delta, pc, mem) \rightsquigarrow (\Delta(rd \mapsto w), pc + 4), mem)}\ (\text{LD_LEMMA})$$

Further semantics for `addi`, `sd` and `ret` RISC-V instructions are provided in [21].

Execution. We also prove high-level big-step semantics rules for common gadgets in binary executables. For example, the *procedure linkage table* (PLT) acts as an intermediary between the current program and shared libraries, including C's standard library. The PLT facilitates indirect calls to external functions, whose locations are unknown until runtime when they are resolved by the dynamic loader. In the dump of the RISC-V binary for Listing 1 given in § 6.2.3, we can observe the presence of stubs within the PLT, representing entries for each external call. Specifically, in § 6.2.3, the stubs correspond to the functions `free` and `malloc` that are called by the program.

By leveraging the consistent pattern observed in PLT entries, we can construct universal big-step rules for PLT stubs, as demonstrated below.

$$\frac{\begin{array}{c}(\Delta_1, pc, mem) \mapsto \texttt{auipc}\ t3,\ 0x2 \qquad (\Delta_2, pc + 4, mem) \mapsto \texttt{ld}\ t3,\ offset(t3) \\ (\Delta_3, pc + 8, mem) \mapsto \texttt{jalr}\ t1,\ t3,\ 0 \qquad \{pc, pc + 4, pc + 8\} \subseteq \texttt{addr_set} \\ \Delta_2 = \Delta_1(t3 \mapsto pc + 0x2) \qquad \Delta_3 = \Delta_2(t3 \mapsto w) \qquad \Delta_4 = \Delta_3(t1 \mapsto pc + 12) \\ ((\Delta_1, pc, mem), \sigma_1) \xrightarrow[BIL]{} \sigma_2 \qquad ((\Delta_2, pc + 4, mem), \sigma_2) \xrightarrow[BIL]{} \sigma_3 \qquad ((\Delta_3, pc + 8, mem), \sigma_3) \xrightarrow[BIL]{} \sigma_4 \\ ((\Delta_4, w, mem), \sigma_4) \xRightarrow[BIL]{} \tau \qquad (mem, v) \in \Delta_1 \qquad \Delta \vdash v[pc + \texttt{0x2} + offset, \mathbf{el}] : 64 \rightsquigarrow^* w\end{array}}{((\Delta_1, pc, mem), \sigma_1) \xRightarrow[BIL]{} \tau}\ (\text{PLT_STUB_LEMMA})$$

Now, if we encounter a PLT stub in a proof at a lower level, we can discharge the proof obligation efficiently using PLT_STUB_LEMMA. This lemma significantly reduces the proof effort required. Without PLT_STUB_LEMMA, one must apply over 150 rules to achieve the same result. Furthermore, by applying this approach to stack allocations and deallocations, we eliminate the need for approximately 50 rules in the case of allocation and 250 rules in the case of deallocation. Further details on this are provided in [21].

6.3 BIL to IsaBIL transpiler

A key component of our IsaBIL framework is a verified BIL parser that transpiles BIL_{ADT} programs into an Isabelle/HOL locale to enable verification.

Parsing occurs in two distinct phases. The first phase involves the translation of BIL_{ADT} into BIL_{ML}, a format suitable for manipulation within Isabelle/ML. This intermediate step enables analysis tasks in Isabelle/ML, such as calculating the size of instructions. The second phase involves translating BIL_{ML} into an Isabelle/HOL representation, which is the starting point for verification. Parsing can be invoked directly on a BIL_{ADT} string with `BIL` or alternately on a file containing BIL_{ADT} with `BIL_file`. An overview of the parsing process is given in Figure 6.

During the first phase, the parser iterates over the instructions in a BIL_{ADT} input. It captures and stores all program addresses in `addr_set`. Moreover, the parser associates symbols with a program address in `sym_table`. Since the syntax of BIL_{ADT} resembles a tree structure, it is intuitive to convert it into an Abstract Syntax Tree (AST) using a lexer. This AST is processed by a Recursive Descent Parser (RDP), which traverses each node in the tree depth-first, matching the node's value to a corresponding parsing function. For example, if the parser encounters a `Var` node, it will attempt to parse the first child as a `string` and the second child as a BIL type. This process transforms the input into BIL_{ML}, a structured data type closely resembling BIL_{ADT} which can be manipulated within Isabelle/ML.

The second phase defines a locale within the current proof context with a fixed decode predicate. For each BIL_{ML} instruction, an assumption is added to the locale, stating how a program with any variable state and memory, but with the instructions program address, decodes to an Isabelle/HOL representation of said instruction. To obtain the Isabelle/HOL representation, an RDP translator recursively converts each BIL_{ML} instruction to Isabelle/HOL terms defined in the **`BIL_specification`** locale (see § 4).

The first phase of the parser harnesses Isabelle/HOL's *code generation*, where specifications for both the lexer and parser were defined within Isabelle/HOL and subsequently translated into ML. BIL_{ML} for example, is a direct ML representation of the BIL specification from the IsaBIL framework. Using *code reflection*, we import the generated ML code back into Isabelle/HOL as a plugin. Consequently, the lexer and parser are formally verified under the Isabelle/HOL framework. In contrast, the second phase employs non-code-generated ML. This decision stems from its necessity to interface with Isabelle/HOL's core ML library, which cannot be achieved via code-generated ML. To bridge this gap, we introduce a minimal layer of unverified ML code that lifts the BIL representation obtained from the verified parser into Isabelle/HOL's proof system. Below, as an example, we present a snippet of the translation code responsible for converting BIL_{ML}'s binary operations to IsaBIL's binary operations:

```
mk_exp (AstParser.BinOp (e1,bop,e2)) = @{term "BinOp"} $ mk_exp e1 $
    mk_bop bop $ mk_exp e2
```

Note that Isabelle isolates Isabelle/ML programming, and hence our parser cannot interfere with the soundness of Isabelle's core logic engine.

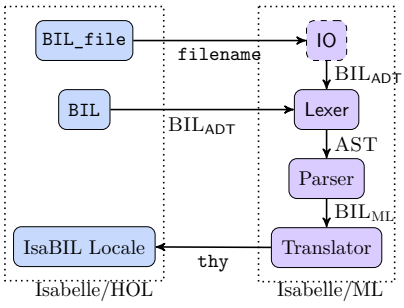


Figure 6 Overview of the BIL parser, entry-points via the `BIL` or `BIL_file` commands.

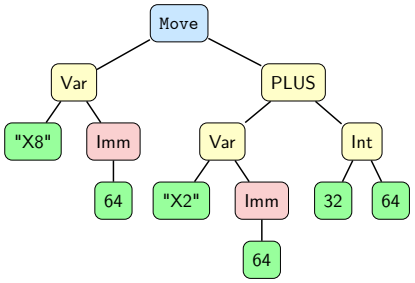


Figure 7 AST representation of the BIL_{ADT} statement `Move(Var("X8", Imm(64)), PLUS(Var("X2", Imm(64)), Int(32, 64)))`.

Table 2 Overview of theories and lemma for each of IsaBIL's components.

Component	Lemmas
Correctness	13
Incorrectness	3
Inference Rules	2
BIL Specification (including syntax)	594
BIL Inference	13
RISC-V optimisations	74
Alloc Model	6
Find Symbol	0
Double Free (total for both examples)	77
AV Rules (total for all seven examples)	187

Table 3 AV rules with description.

AVR	Description (the following shall not be used)
17	The error indicator errno
19	`<locale.h>` and the setlocale function[5]
20	The `setjmp` macro and the `longjmp` functions
21	The signal handling facilities of `<signal.h>`
23	The library functions `atof`, `atoi` and `atol` from library `<stdlib.h>`
24	The library functions `abort`, `exit`, `getenv` and system from library `<stdlib.h>`
25	The time handling functions of library `<time.h>`

6.4 Example Proofs

The majority of the development consists of general tactics and lemmas which can be reused across multiple developments. We give the number of lemmas for each component in Table 2.

All of the sublocales and inherited locales can reuse the proofs of the higher-level (more abstract) locales. Thus, for instance, the 594 lemmas of the **`BIL_specification`** locale are applicable to **`BIL_inference`** and all of the examples, which improves re-usability. The RISC-V optimisations help improve performance and are reusable across all examples. The theories for **`double_free_*`** and **`find_symbol`** are general and apply to all corresponding examples. Finally, each example comprises a step lemma for each line of code. Using the earlier optimisations, solving them is trivial and only requires running the automated tactics.

To motivate IsaBIL, we employ a combination of correctness and incorrectness proofs, focusing on illustrative examples frequently referenced in BAP literature [8]. These examples represent patterns commonly found within much larger programs/libraries.

6.4.1 Double Free

We detail the (in)correctness proofs for CWE-415: Double Free outlined in §2. To classify this vulnerability, we require a memory allocation model, which is not part of the BIL semantics (§3). Memory allocation in C is reflected as a PLT stub in the binary. In particular, we are only required to track the pointer (memory address) that is allocated (and later freed). To this end, we develop a locale, **`allocation`**, that extends the BIL semantics (see Figure 2) with an abstract allocation model inspired by earlier works [34, 31].

$$\frac{\texttt{is_alloc}(\mathcal{C}) \quad w = \texttt{next_ptr}(\omega_{\mathcal{A}}) \quad sz = \texttt{get_sz}(\mathcal{C}) \quad \omega'_{\mathcal{A}} = \omega_{\mathcal{A}}.[\texttt{alloc}(w, sz)]}{(\mathcal{C}, \omega_{\mathcal{A}}) \underset{\mathcal{A}}{\longrightarrow} \omega'_{\mathcal{A}}}\ (\text{ALLOC}) \qquad \frac{\texttt{is_free}(\mathcal{C}) \quad w = \texttt{get_ptr}(\mathcal{C}) \quad \omega'_{\mathcal{A}} = \omega_{\mathcal{A}}.[\texttt{free}(w)]}{(\mathcal{C}, \omega_{\mathcal{A}}) \underset{\mathcal{A}}{\longrightarrow} \omega'_{\mathcal{A}}}\ (\text{FREE}) \qquad \frac{\neg\texttt{is_free}(\mathcal{C}) \quad \neg\texttt{is_alloc}(\mathcal{C})}{(\mathcal{C}, \omega_{\mathcal{A}}) \underset{\mathcal{A}}{\longrightarrow} \omega_{\mathcal{A}}}\ (\text{SKIP})$$

Figure 8 Allocation semantics.

The `allocation` locale. We define `allocation` by instantiating the small-step transition relation $\underset{BIL}{\longrightarrow}$ of `BIL_specification` (see §6.1). We start by defining two memory operations:

$$memop ::= \texttt{alloc}(w, sz) \mid \texttt{free}(w)$$

where w is the memory being allocated or freed and sz is the size. Additionally, we assume an abstract allocation function `next_ptr` : $memop^* \rightarrow word$ that serves the purpose of selecting suitable memory addresses for allocation and retrieves the next address from the allocator based on the given history (sequence) of allocations and deallocations. For each allocation, we use `next_ptr` to obtain the next available memory address. Consequently, in the state of `allocation`, we must track the sequence of *memop* operations, which is provided to `next_ptr` as an input whenever an allocation occurs. For the Double Free example, no other variables need to be tracked, hence we simply assume the state to be this sequence, which we denote $\omega_{\mathcal{A}} : memop^*$.

Recall that by using the $\mapsto$ predicate (§ 3.5), the BIL instruction corresponding to $\mathcal{C} = (\Delta, pc, mem)$ can be uniquely identified. We use predicates `is_free` : $prog \rightarrow bool$ and `is_alloc` : $prog \rightarrow bool$, which hold when the next instruction to be executed in the given $\mathcal{C} \in prog$ corresponds to a free and allocation instruction, respectively. We assume that an instruction that frees memory cannot simultaneously perform an allocation and vice versa, i.e., $\neg\texttt{is_free}(\mathcal{C}) \vee \neg\texttt{is_alloc}(\mathcal{C})$. When the instruction for $\mathcal{C}$ is an allocation, we utilise the function `get_sz` : $prog \rightarrow sz$ to obtain the size allocated and when the instruction for $\mathcal{C}$ is a deallocation, we utilise the function `get_ptr` : $prog \rightarrow word$ to obtain the address that is freed. The size and location can be determined using the $\mapsto$ function on the given $\mathcal{C}$. Thus, we define a small-step transition relation, $\underset{\mathcal{A}}{\longrightarrow}$, over allocations in Figure 8.

We encode this *allocation model* as the locale `allocation`, fixing the functions `next_ptr`, `is_free`, `is_alloc`, `get_sz` and `get_ptr`. This locale is derived as an *interpretation* of the locale `BIL_inference` retaining the decode predicate $\mapsto$ and address set `addr_set` as abstract, but overriding $\underset{BIL}{\longrightarrow} = \underset{\mathcal{A}}{\longrightarrow}$. Thus, we have the signature `allocation` = ($\mapsto$, `addr_set`, `next_ptr`, `is_free`, `is_alloc`, `get_sz`, `get_ptr`), where parameters `next_ptr`, `is_free`, `is_alloc`, `get_sz` and `get_ptr` are used to derive $\underset{\mathcal{A}}{\longrightarrow}$ within the locale.

The `double_free_bad` and `double_free_good` locales. These locales correspond to the double free program and are *auto-generated* from the corresponding BIL programs (shown in Listings 3 and 4). This step is trivial when using IsaBIL's `BIL_file` command, which we have developed using Isabelle/ML. In particular, we invoke the Isabelle/HOL commands:

```
BIL_file ‹double-free-bad.bil.adt›  defining double_free_bad
BIL_file ‹double-free-good.bil.adt› defining double_free_good
```

where `double-free-bad.bil.adt` is a file containing the BIL for Listing 1 in BIL_{ADT} format and `double_free_bad` is the name of the locale to be generated. The good version is similar.

The `double_free_*_proof` locales. We combine the locales generated from the BIL programs with our allocator locale (**`allocation`**) to produce new locales **`double_free_bad_proof`** and **`double_free_good_proof`**. Recall from § 4.2 that for each locale generated from a BIL_{ADT} program, we have access to a symbol table `sym_table` mapping strings (i.e., external function calls) to the program address at which the symbol appears. Note that when a BIL program calls a function, it typically stores the return address, then jumps to the function. The symbol corresponding to the program counter after the jump contains the name of the external function, which is stored in the symbol table. In RISC-V convention, the first argument of a function call is stored in the register `X10`. To allocate memory, function `malloc` is called with the size as the first argument whereas to deallocate a pointer, function `free` is called with the pointer as the first argument. Therefore, we can obtain the size allocated by a call to `malloc` and pointer freed by a call to `free` by retrieving the value stored in register `X10`.

We therefore instantiate the predicates `is_free`, `is_alloc`, `get_sz` and `get_ptr` from the **`allocation`** locale as follows:

$$\texttt{is_free}((_, pc, _)) = (pc = \texttt{sym_table}(\text{“}\texttt{free}\text{”})) \qquad \texttt{get_ptr}((\Delta, _, _)) = \Delta(\texttt{X10})$$
$$\texttt{is_alloc}((_, pc, _)) = (pc = \texttt{sym_table}(\text{“}\texttt{malloc}\text{”})) \qquad \texttt{get_sz}((\Delta, _, _)) = \Delta(\texttt{X10})$$

The exact implementation of `next_free` occurs at a lower level of abstraction, and depends on the allocation model. At this level, we provide the main requirement axiomatically, i.e., we require

$$w = \texttt{next_ptr}(\omega_{\mathcal{A}}) \implies (\forall i.\ \omega_{\mathcal{A}}(i) = \texttt{alloc}(w, sz) \implies \exists j.\ j > i \wedge \omega_{\mathcal{A}}(i) = \texttt{free}(w))$$

This presents us with all the components necessary to prove (in)correctness of our examples. The following predicate captures the double-free over the memory state defined above:

$$\texttt{double_free_vuln} = \lambda\omega_{\mathcal{A}}.\ \exists i, j, w.\ i < j \wedge \omega_{\mathcal{A}}[i] = \texttt{free}(w) \wedge \omega_{\mathcal{A}}[j] = \texttt{free}(w) \wedge (\forall k.\ i < k < j \implies \omega_{\mathcal{A}}[k] \neq \texttt{alloc}(w, sz))$$

Note that the predicate captures *only* the double-free vulnerability and does not formalise other pointer misuse vulnerabilities, e.g., use-after-free or free-before-alloc.

▶ **Theorem 6.** *Let $\mathcal{C}_{bad}$ and $\mathcal{C}_{good}$ be the programs corresponding to* ***`double_free_bad`*** *and* ***`double_free_good`****, respectively. Both of the following hold*

1. $\exists Q.\ [\neg\texttt{double_free_vuln}]\ \mathcal{C}_{bad}\ [Q] \wedge (\forall \sigma.\ Q(\sigma) \Longrightarrow \texttt{double_free_vuln}(\sigma)) \wedge (\exists \sigma.\ Q(\sigma))$
2. $\{\neg\texttt{double_free_vuln}\}\ \mathcal{C}_{good}\ \{\neg\texttt{double_free_vuln}\}$

Note that in $\mathcal{C}_{bad}$, the memory error does not lead to early termination via the allocation semantics in Figure 8. The program continues executing as normal, but generates a history that can be shown to contain a double-free vulnerability at termination.

6.4.2 AV rules

In a similar manner, we prove incorrectness of the AV rules from Table 3. Details of these proofs are provided in [21].

7 Case Study: Double-Free Vulnerability in cURL (CVE-2016-8619)

cURL is an open-source library primarily used for transferring data over the internet using various protocols, including TLS. It has a large user base, supporting multiple operating systems and architectures (including RISC-V), and thus ensuring the security of cURL is critically important.

```
1  static __attribute__ ((noinline)) CURLcode read_data(struct connectdata *conn,
2                                                       curl_socket_t fd,
3                                                       struct krb5buffer *buf)
4  {
5    int len;
6    void* tmp = NULL;
7    CURLcode result;
8
9    result = socket_read(fd, &len, sizeof(len)); // result = 0 iff socket_read successful
10   if(result)
11     return result;
12
13   len = ntohl(len);
14   tmp = realloc(buf->data, len);
15
16   if (len) {
17     len = ntohl(len);
18     tmp = realloc(buf->data, len);
19   }
20
21   if(tmp == NULL)
22     return CURLE_OUT_OF_MEMORY;
23
24   buf->data = tmp;
25   result = socket_read(fd, buf->data, len);
26   if(result)
27     return result;
28   buf->size = conn->mech->decode(conn->app_data, buf->data, len,
29                                  conn->data_prot, conn);
30   buf->index = 0;
31   return CURLE_OK;
32 }
```

Figure 9 The `read_data` function of the cURL library in `security.c`. The snippet highlighted in red shows the vulnerable code in version 7.50.3, and that in green shows the corrected vulnerability in version 7.51.0.

Unfortunately, however, cURL has been known to contain vulnerabilities. Consider the function `read_data` in Figure 9, which underpins the Kerberos authentication protocol in cURL (prior to version 7.51.0), where § 7 reads the length of an incoming data packet from a socket. If this succeeds, `socket_read` will return a 0, causing the if statement on § 7 to fall through. Then, on § 7 the call to `ntohl` converts the data length (`len`) from network byte order (big-endian) to host byte order (big- or little-endian). Note that this method does not sanitise its input. Next, `realloc` on § 7 resizes the `data` buffer in `krb5buffer`, and returns either a new pointer to the resized memory, or `NULL` if an error occurs. The rest of the method then reads and decodes the remaining data from the socket.

The caller is responsible for managing the `krb5buffer` pointer by allocating it before and freeing it after calling `read_data`. Here we assume `malloc` is called directly before `read_data` and free is called directly after, as follows: `malloc(buf, sz); read_data(conn, fd, buf); free(buf);`

If receiving data from the socket on § 7 yields a length of 0 (`len` = 0), the call to `realloc` with `len` = 0 results in *zero reallocation*, leading to *undefined behaviour*[6]. Most implementations of `realloc` will return NULL and free the pointer following a *zero reallocation*. As such, since the `krb5buffer` buffer is already freed before `realloc` returns and the caller is expected to free it after the function returns, this will lead to a *double-free vulnerability*, identified by the vulnerability enumeration CVE-2016-8619.

[6] `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2464.pdf`

$$\frac{\texttt{is_realloc}(\mathcal{C}) \quad w = \texttt{get_ptr}(\mathcal{C}) \quad sz = \texttt{get_sz}(\mathcal{C}) \quad \omega'_{\mathcal{A}} = \omega_{\mathcal{A}}.[\texttt{alloc}(w, sz)]}{(\mathcal{C}, \omega_{\mathcal{A}}) \xrightarrow[\mathcal{R}]{} \omega'_{\mathcal{A}}} \text{(REALLOC)}$$

$$\frac{\texttt{is_realloc}(\mathcal{C}) \quad w = \texttt{get_ptr}(\mathcal{C}) \quad sz = \texttt{get_sz}(\mathcal{C}) \quad sz = 0 \quad \omega'_{\mathcal{A}} = \omega_{\mathcal{A}}.[\texttt{free}(w)]}{(\mathcal{C}, \omega_{\mathcal{A}}) \xrightarrow[\mathcal{R}]{} \omega'_{\mathcal{A}}} \text{(ZERO_REALLOC)}$$

$$\frac{\neg\texttt{is_free}(\mathcal{C}) \quad \neg\texttt{is_alloc}(\mathcal{C}) \quad \neg\texttt{is_realloc}(\mathcal{C})}{(\mathcal{C}, \omega_{\mathcal{A}}) \xrightarrow[\mathcal{R}]{} \omega_{\mathcal{A}}} \text{(SKIP)}$$

Figure 10 Reallocation semantics.

This vulnerability is resolved in version 7.51.0 of cURL by preempting the zero-allocation that leads to it: as highlighted in green in Figure 9, placing a guard around the call to `realloc`[7] ensures that it is not called with a zero length. As such, if `len` $= 0$, then the call to `realloc` is skipped, `TMP` remains `NULL` and returns 0. This ensures consistent behaviour across all C implementations.

7.1 Scalability of the Parser

We cross compile cURL 7.50.3 for RISC-V on Linux using the GNU RISC-V compiler with the default options. We modify the source code for `read_data` by adding the `noinline` attribute, which ensures the function is not inlined. Note that `noinline` is not a requirement for verification, but facilitates compositional reasoning (see § 7.3).

Using the generated BIL directly creates a scalability challenge. The assembly corresponding to version 7.50.3 contains *63,592 RISC-V instructions*, which equates to *127,023 LoC in BIL* with *63,592 BIL instructions*. Naively generating the corresponding IsaBIL locales (see § 6.3) takes approximately 14 minutes, which is impractical for verification. Of this, the lexer takes 4s, the parser takes 2s and the translator takes approximately 13 minutes. The wait time in the translator arises from instantiating the Isabelle/HOL locale in the proof context, an operation which cannot be avoided.

To address this, we extend the `BIL_file` command with a *new option*, `with_subroutines`, to focus only on the subroutines of interest (and their dependencies). For example, to generate the IsaBIL locale for `read_data` in version 7.50.3, we use the following command:

```
BIL_file <libcurl.7.50.3.bil.adt> defining read_data_7_50_3
  with_subroutines read_data and ...
```

where "..." contains functions that `read_data` calls (e.g., `ntohl`). Running `BIL_file` now takes only 6s, with 131 instructions to verify. We next show how we detect this vulnerability in version 7.50.3 using an incorrectness proof.

7.2 Incorrectness of the `read_data` Subroutine

`read_data` contains a double-free vulnerability. However, unlike our prior example (§ 6.4.1), the vulnerability arises from a reallocation of memory. Thus, we first describe the construction of a **`reallocation`** locale, which reuses components from the **`allocation`** locale. Then, we prove the incorrectness of the `read_data` function in version 7.50.3.

The `reallocation` locale. We define **`reallocation`** by instantiating the small-step transition relation $\xrightarrow[BIL]{}$ of **`BIL_specification`** (see § 6.1). Whilst we do not extend **`allocation`** from § 6.4.1 directly, as it already defines small step semantics ($\xrightarrow[\mathcal{A}]{}$) that are incompatible

[7] https://github.com/curl/curl/commit/3d6460edeee21d7d790ec570d0887bed1f4366dd

with reallocation, we can borrow: (1) the definition of *memop*, (2) our abstract allocator `next_ptr`, (3) our memory trace $\omega_{\mathcal{A}}$, (4) the getter for the size `get_sz`, (5) the getter for the pointer to free `get_ptr`, (6) the predicate that denotes a program is freeing memory `is_free` and (7) the predicate that denotes a program is allocating memory `is_alloc`. These concepts are explained in detail in § 6.4.1.

We use the predicate `is_realloc` to determine whether the next instruction to be executed in the given $\mathcal{C}$ corresponds to a `realloc` instruction. We assume that an instruction that frees or allocates memory cannot simultaneously perform an reallocation and vice versa, i.e., $\neg\texttt{is_free}(\mathcal{C}) \vee \neg\texttt{is_alloc}(\mathcal{C}) \vee \neg\texttt{is_realloc}(\mathcal{C})$.

When a program reallocates memory it has two parameters, a pointer to the memory to be resized, and the new size of that memory. We utilise the existing functions `get_sz` to obtain the size reallocated and `get_ptr` to obtain the address that is reallocated. The size and location can be determined using the $\mapsto$ function on the given $\mathcal{C}$. The size and location are then used to add an $\texttt{alloc}(w, sz)$ to the $\omega_{\mathcal{A}}$. However, if the size of a reallocation is 0, this represents a zero-reallocation, and hence a $\texttt{free}(w)$ is added to the $\omega_{\mathcal{A}}$. This results in a small-step transition relation, $\underset{\mathcal{R}}{\longrightarrow}$, over reallocations as given in Figure 10. Note that we retain the ALLOC and FREE rules from Figure 8, and adapt the SKIP to encompass is reallocations. Finally, we obtain the signature

$$\texttt{reallocation} = (\mapsto, \texttt{addr_set}, \texttt{next_ptr}, \texttt{is_free}, \texttt{is_alloc}, \texttt{is_realloc}, \texttt{get_sz}, \texttt{get_ptr})$$

where the parameters `next_ptr`, `is_free`, `is_realloc`, `is_alloc`, `get_sz` and `get_ptr` are used to derive $\underset{\mathcal{R}}{\longrightarrow}$ within the locale.

The `read_data` proof locale. We combine the locale generated for cURL binaries to produce a new locale `read_data_7_50_3_proof`. Reallocations are handled by calls to a PLT stub for `realloc` so we can proceed in the same way as `malloc`/`free`. In § 6.4.1, we discussed how the first argument of a subroutine call is stored in the register `X10`. Reallocations take two arguments: (1) a pointer to memory, and (2) a size. The second argument (2) is stored in the return register (`X11`), thus we instantiate the parameters of **`reallocation`** as follows for both locales:

$$\texttt{is_realloc}(_, pc, _) = (pc = \texttt{sym_table}(\text{“}\texttt{realloc}\text{”}))$$

$$\texttt{get_sz}(\Delta, pc, _) = \begin{cases} \Delta(\texttt{X11}) & \textbf{if } pc = \texttt{sym_table}(\text{“}\texttt{realloc}\text{”}) \\ \Delta(\texttt{X10}) & \textbf{otherwise} \end{cases}$$

The remaining definitions of `get_ptr`, `is_free` and `is_alloc` match **`allocation`** (see §6.4.1).

▶ **Theorem 7.** *Let $RD_{7.50.3}$ be the program corresponding to* **`read_data_7_50_3`**. *Then*

$$\exists Q.\ [\neg\texttt{double_free_vuln}]\ RD_{7.50.3}\ [Q] \wedge (\forall \sigma.\ Q(\sigma) \implies \texttt{double_free_vuln}(\sigma)) \wedge (\exists \sigma.\ Q(\sigma)).$$

7.3 Compositionality

The proof in Figure 9 covers the `read_data` subroutine in the cURL library. However, this subroutine is an *internal* function (i.e. not defined in a header file), making the vulnerability difficult to exploit directly in practice. Nevertheless, as shown in the call graph in Figure 11, there is a path from an *external* function (defined in a header file), namely `Curl_sec_login`,

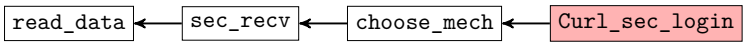


Figure 11 A partial call graph in cURL demonstrating a path from an external function (highlighted), `Curl_sec_login`, to the internal function `read_data`.

to `read_data`. That is, a call to `Curl_sec_login` includes a call to `choose_mech`, which in turn includes a call to `sec_recv`, which itself calls `read_data`. As such, the vulnerability in `read_data` *could potentially* be exploited by a call to `Curl_sec_login`.

Note that this may not always be the case, as freeing the data buffer pointer is left up to the caller, not the callee, and thus further incorrect behaviour to by caller (in this case a function above `read_data` in the call graph) could rectify this vulnerability by forgetting to call free, or perhaps reallocating the pointer again before freeing it. Thus, we should aim to establish that each of these methods are (in)correct formally.

We do this in a *compositional* fashion through *proof reuse*: we reuse the (in)correctness specifications we prove at lower levels of the call graph in higher levels. For instance, when proving that `sec_recv` is (in)correct, we reuse the specification of `read_data` in Theorem 7 as a lemma. Indeed, this is precisely why we used the `noinline` attribute when compiling `read_data`: this allows us to reuse its specification; otherwise each call to `read_data` would be inlined, preventing us from proof reuse.

In Theorem 8 below, we prove incorrectness specifications for the $SR_{7.50.3}$, $CM_{7.50.3}$ and $CSL_{7.50.3}$ functions, showing that each of these functions do indeed exhibit a double-free vulnerability. Our proof is compositional in that the proof for each function reuses the specification of the function below it, as discussed above (e.g. the proof of $SR_{7.50.3}$ reuses the incorrectness specification of `read_data` in Theorem 7 as a lemma).

▶ **Theorem 8.** *Let* $C \in \{SR_{7.50.3}, CM_{7.50.3}, CSL_{7.50.3}\}$*, be a program corresponding to the 7.50.3 versions of* `sec_recv`*,* `choose_mech` *and* `Curl_sec_login`*. Then:*

$$\exists Q.\ [\neg\texttt{double_free_vuln}]\ C\ [Q] \wedge (\forall \sigma.\ Q(\sigma) \implies \texttt{double_free_vuln}(\sigma)) \wedge (\exists \sigma.\ Q(\sigma)).$$

Theorem 8 demonstrates a scalable proof for the incorrectness specifications of the functions in Figure 11. We can apply similar techniques to verify their corresponding correctness specifications for version 7.51.0; we have elided these proofs as they are similar.

8 Conclusions and Related Work

Program analysis in the absence of source code techniques typically focus on a combination of hardware (e.g., x86, ARM), intermediary representations, e.g., LLVM-IR, high-level language models (e.g., C/C++). Our work adds to the growing literature on low-level verification using Hoare logic, and is the first to also apply incorrectness logic to verify the absence and presence of bugs in binaries. While many works in the literature focus on particular architectures, e.g., RISC-V, x86 or ARM, our work is on the generic BAP platform, and hence inherits its generality. BAP is a production ready system with 1000s of users but lack a verification component that IsaBIL provides. We have developed a complete formalisation of BIL, and developed the first (in)correctness logics for BIL. Our formalisation of the BIL specification uncovered bugs in that specification, which have been reported to the developers and corrected. We provide automation to enable better proof scalability, and demonstrate compositional proofs.

The importance of binary-level analysis has meant that there is increasing work on theorem provers being applied to verify program binaries.

Verified machine code. Closest to our work is the PICINÆ framework [22], which lifts BIL to Coq for verifying properties like termination and functional correctness. Unlike IsaBIL, its lifter is unverified and adds an extra step via a custom intermediate representation (IR) with unclear semantics. The work is preliminary, focusing on small examples like memset and string comparisons.

Early machine-code verification includes Myreen et al.[37, 38], who used Hoare logic on high-level Arm models. Similarly, Jensen et al. [25] developed separation logic for verifying low-level x86 code via abstractions, with the framework being encoded in Coq. Later work on *decompilation into logic* [39, 40] targets x86, ARM, and PowerPC by translating binaries into tail-recursive functions in HOL4 [24], avoiding conventional lifters like BAP. Verification is done via Hoare triples over these functions, shifting trust to mechanised ISA models [13, 30, 16, 17]. However, these models were not originally designed for decompilation and may not reflect modern ISAs.

The Islaris framework [44] builds on Sail-based Armv8-A and RISC-V semantics [47, 48, 1], using higher-order separation logic in Coq with automatic Sail-to-Coq translation. Unlike Islaris, which focuses on reasoning directly over ISA models, our approach targets lifted assembly to enable a unifying, architecture-agnostic proof layer. Our automation (§ 6) is reusable across architectures supported by BAP. MiniSail [54] encodes a subset of Sail in proof assistants like Isabelle/HOL, enabling reasoning over lifted Sail models similar to IsaBIL. While useful for verifying Sail's type soundness, it lacks IsaBIL's automation and higher-level reasoning capabilities.

Verified compilation. Notable compilation correctness efforts include CakeML [28] and CompCert [7], targeting ML and C, respectively. Sewell et al.[45] verify seL4 C code down to gcc-generated binaries. Bedrock [11] uses an intermediate representation and strongest postcondition reasoning in Coq for C macros, unlike IsaBIL, which allows verification of binaries that may not be generated by a compiler. Overall, these approaches verify functional correctness at the source level, requiring compiler models and assumptions about architecture and optimisation levels. Moreoever, they are unsuitable when source code is unavailable, whereas IsaBIL, via BAP, operates directly on binaries.

Verified lifting. Verbeek et al [51] have formalised and integrated a subset of the x86-64 instruction set (i.e., the 64-bit subset of x86) within Isabelle/HOL and used it to verify so called *sanity properties* (e.g., the integrity of a return address) for a large codebase. However, the focus of this verification is correctness of the lifting, as opposed to properties about the program itself (e.g., termination, functionality). IsaBIL employs type verification, however, this method only verifies that the lifted BIL is correct according to its specification, and not the specification of the assembly it was lifted from. Lam and Coughlin [29] developed a verified BAP lifter for ARMv8. BAP is already accompanied by an extensive set of validation tools that uses trace files to verify the lifter's accuracy [8], that are checked "per instruction". We consider formal verification of BAP lifters to be future work.

Verified instrumentation. Armor [56] presents a checker for memory safety and control flow integrity of Arm binaries, which instruments a check before particular operations, then uses HOL to prove that the modified binary is correct. Similarly, RockSalt [35] is a

[8] `https://github.com/BinaryAnalysisPlatform/bap-veri`

Coq-based checker for a subset of x86 that is used to verify software-based fault isolation implementations. RockSalt is designed to check specific sandbox policies, e.g., that the code will only read/write data from specified contiguous segments in memory. Unlike IsaBIL, which has the full flexibility of Hoare and O'Hearn's logics for proving (in)correctness, both Armor and RockSalt focus on a subset of properties.

Security analysis. An overview of security properties, exploits and tools for binary analysis has been given by Shoshitaishvili et al [46]. Above, we have focussed mainly on safety, though also describe how BAP can be extended to reason about transient execution vulnerabilities (e.g., as exploited by Spectre and Meltdown) [53, 10, 19]. Cheang et al [10] cover model checking (using UCLID5), while Wang et al [53] develop a taint tracking tool based on BAP to check correctness of Spectre mitigations. Griffin and Dongol [19] consider proofs of speculative execution over BAP outputs, but only cover a subset of the BIL language called AIR. They apply Hoare-style proofs to verify a hyperproperty [12] known as TPOD [10] directly over AIR without any of the modularity offered by IsaBIL. Nevertheless, it would be interesting to extend our work to cover proofs of (in)correctness of speculative execution [9, 14].

(Incorrectness) separation logic. In IsaBIL we do not currently have support for separation logic (SL) or its incorrectness analogue, ISL [43]. While the absence of (I)SL support does not impede the scalability of our approach, it is an interesting direction of future work to pursue. Doing so, however, is non-trivial as the memory representation in BIL is incompatible with that of (I)SL. As described in § 3.1, memory in BIL is represented as a "history" of updates (mutations) that must be "replayed" to ascertain its contents. For instance, the BIL memory $[x \leftarrow 1][y \leftarrow 2][x \leftarrow 3]$ describes *any* memory obtainable after applying (replaying) the listed updates in order, namely by first updating location x to hold 1, then updating y to 2 and finally re-updating x to 3. While this confers the existence and the final values of locations x and y, it has no bearing on the existence or the values of other memory locations.

Representing the memory in BIL as described above is a design choice we *inherited* from BIL, and it is not immediately conducive to "separation" as in (I)SL. As such, to add support for (I)SL in BIL we would have to fundamentally change the memory representation in BIL, an undertaking which is far from trivial. While we believe this a worthy direction of work in the future, doing so is beyond the scope of our work here.

References

1 Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for armv8-a, risc-v, and CHERI-MIPS. *Proc. ACM Program. Lang.*, 3(POPL):71:1–71:31, 2019. doi:10.1145/3290384.

2 Atmel. Avr instruction set manual, 2016. URL: https://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf.

3 Clemens Ballarin. Locales and locale expressions in isabelle/isar. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *LNCS*, pages 34–50. Springer, 2003. doi:10.1007/978-3-540-24849-1_3.

4 BAP. Bap toolkit examples, 2019. URL: https://github.com/BinaryAnalysisPlatform/bap-toolkit/blob/master/av-rule-21/descr.

5 Thomas Bauereiß, Armando Pesenti Gritti, Andrei Popescu, and Franco Raimondi. Cosmedis: A distributed social media platform with formally verified confidentiality guarantees. In *SP*, pages 729–748. IEEE Computer Society, 2017. doi:10.1109/SP.2017.24.

6 A formal specification for BIL: BIL instruction language (v0.3), 2018. URL: https://github.com/BinaryAnalysisPlatform/bil/releases.

7 Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *LNCS*, pages 460–475. Springer, 2006. doi:10.1007/11813040_31.

8 David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, volume 6806 of *LNCS*, pages 463–469. Springer, 2011. doi:10.1007/978-3-642-22110-1_37.

9 Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new spectre era. In Alastair F. Donaldson and Emina Torlak, editors, *PLDI*, pages 913–926. ACM, 2020. doi:10.1145/3385412.3385970.

10 Kevin Cheang, Cameron Rasmussen, Sanjit A. Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In *CSF*, pages 288–303. IEEE, 2019. doi:10.1109/CSF.2019.00027.

11 Adam Chlipala. The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. In Greg Morrisett and Tarmo Uustalu, editors, *ICFP*, pages 391–402. ACM, 2013. doi:10.1145/2500365.2500592.

12 M. R. Clarkson and F. B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, 2010. doi:10.3233/JCS-2009-0393.

13 Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. In Franz Baader, editor, *CADE*, volume 2741 of *Lecture Notes in Computer Science*, pages 106–120. Springer, 2003. doi:10.1007/978-3-540-45085-6_9.

14 Brijesh Dongol, Matt Griffin, Andrei Popescu, and Jamie Wright. Relative security: Formally modeling and (dis)proving resilience against semantic optimization vulnerabilities. In *CSF*, pages 403–418. IEEE, 2024. doi:10.1109/CSF61375.2024.00027.

15 Chelsea Edmonds and Lawrence C. Paulson. Formal probabilistic methods for combinatorial structures using the lovász local lemma. In Amin Timany, Dmitriy Traytel, Brigitte Pientka, and Sandrine Blazy, editors, *CPP*, pages 132–146. ACM, 2024. doi:10.1145/3636501.3636946.

16 Anthony C. J. Fox. Formal specification and verification of ARM6. In David A. Basin and Burkhart Wolff, editors, *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 25–40. Springer, 2003. doi:10.1007/10930755_2.

17 Anthony C. J. Fox and Magnus O. Myreen. A trustworthy monadic formalization of the armv7 instruction set architecture. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2010. doi:10.1007/978-3-642-14052-5_18.

18 Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying strong eventual consistency in distributed systems. *CoRR*, abs/1707.01747, 2017. arXiv:1707.01747.

19 Matt Griffin and Brijesh Dongol. Verifying secure speculation in Isabelle/HOL. In Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan, editors, *FM*, volume 13047 of *LNCS*, pages 43–60. Springer, 2021. doi:10.1007/978-3-030-90870-6_3.

20 Matt Griffin, Brijesh Dongol, and Azalea Raad. Isabelle/HOL files for "IsaBIL: A framework for verifying (in)correctness of binaries in Isabelle/HOL", 2025. doi:10.5281/zenodo.15261359.

21 Matt Griffin, Brijesh Dongol, and Azalea Raad. IsaBIL: A framework for verifying (in)correctness of binaries in Isabelle/HOL (Extended version), 2025. arXiv:2504.16775.

22 Kevin W. Hamlen, Dakota Fisher, and Gilmore R. Lundquist. Source-free machine-checked validation of native code in coq. In *FEAST*, FEAST'19, pages 25–30, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3338502.3359759.

23 Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. doi:10.1145/363235.363259.

24 Graham Hutton. Book review: Introduction to HOL: A theorem proving environment for higher order logic by mike gordon and tom melham (eds.), cambridge university press, 1993, ISBN 0-521-44189-7. *J. Funct. Program.*, 4(4):557–559, 1994. doi:10.1017/S0956796800001180.

25 Jonas Braband Jensen, Nick Benton, and Andrew Kennedy. High-level separation logic for low-level code. In Roberto Giacobazzi and Radhia Cousot, editors, *POPL*, pages 301–314. ACM, 2013. doi:10.1145/2429069.2429105.

26 Florian Kammüller and Manfred Kerber. Investigating airplane safety and security against insider threats using logical modeling. In *SP*, pages 304–313. IEEE Computer Society, 2016. doi:10.1109/SPW.2016.47.

27 Florian Kammüller, Markus Wenzel, and Lawrence C Paulson. Locales a sectioning concept for isabelle. In *TPHOLs*, pages 149–165. Springer, 1999. doi:10.1007/3-540-48256-3_11.

28 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *POPL*, pages 179–192. ACM, 2014. doi:10.1145/2535838.2535841.

29 Kait Lam and Nicholas Coughlin. Lift-off: Trustworthy armv8 semantics from formal specifications. In Alexander Nadel and Kristin Yvonne Rozier, editors, *FMCAD*, pages 274–283. IEEE, 2023. doi:10.34727/2023/ISBN.978-3-85448-060-0_36.

30 Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 42–54. ACM, 2006. doi:10.1145/1111037.1111042.

31 Xavier Leroy and Sandrine Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *J. Autom. Reason.*, 41(1):1–31, 2008. doi:10.1007/s10817-008-9099-0.

32 Lockheed Martin. Joint strike fighter, air vehicle, c++ coding standard, 2005.

33 Daniel Matichuk, Toby C. Murray, and Makarius Wenzel. Eisbach: A proof method language for isabelle. *J. Autom. Reason.*, 56(3):261–282, 2016. doi:10.1007/s10817-015-9360-2.

34 Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring C semantics and pointer provenance. *Proc. ACM Program. Lang.*, 3(POPL):67:1–67:32, 2019. doi:10.1145/3290380.

35 Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. Rocksalt: better, faster, stronger SFI for the x86. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *PLDI*, pages 395–404. ACM, 2012. doi:10.1145/2254064.2254111.

36 Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *ASE*, pages 1186–1189. IEEE, 2019. doi:10.1109/ASE.2019.00133.

37 Magnus O. Myreen, Anthony C. J. Fox, and Michael J. C. Gordon. Hoare logic for ARM machine code. In Farhad Arbab and Marjan Sirjani, editors, *FSEN*, volume 4767 of *LNCS*, pages 272–286. Springer, 2007. doi:10.1007/978-3-540-75698-9_18.

38 Magnus O. Myreen and Michael J. C. Gordon. Hoare logic for realistically modelled machine code. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *LNCS*, pages 568–582. Springer, 2007. doi:10.1007/978-3-540-71209-1_44.

39 Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. Machine-code verification for multiple architectures - an application of decompilation into logic. In Alessandro Cimatti and Robert B. Jones, editors, *FMCAD*, pages 1–8. IEEE, 2008. doi:10.1109/FMCAD.2008.ECP.24.

40 Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. Decompilation into logic - improved. In Gianpiero Cabodi and Satnam Singh, editors, *FMCAD*, pages 78–81. IEEE, 2012. URL: https://ieeexplore.ieee.org/document/6462558/.

41 Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002. doi:10.1007/3-540-45949-9.

42 Peter W. O'Hearn. Incorrectness logic. *POPL*, 4(POPL):1–32, 2019. doi:10.1145/3371078.

43 Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. Local reasoning about the presence of bugs: Incorrectness separation logic. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 225–252, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-53291-8_14.

44 Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. Islaris: verification of machine code against authoritative ISA semantics. In Ranjit Jhala and Isil Dillig, editors, *PLDI*, pages 825–840. ACM, 2022. doi:10.1145/3519939.3523434.

45 Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In Hans-Juergen Boehm and Cormac Flanagan, editors, *PLDI*, pages 471–482. ACM, 2013. doi:10.1145/2491956.2462183.

46 Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE SP*, 2016.

47 Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. Relaxed virtual memory in armv8-a. In Ilya Sergey, editor, *ESOP*, volume 13240 of *LNCS*, pages 143–173. Springer, 2022. doi:10.1007/978-3-030-99336-8_6.

48 Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. Armv8-a system semantics: Instruction fetch in relaxed architectures. In Peter Müller, editor, *ESOP*, volume 12075 of *LNCS*, pages 626–655. Springer, 2020. doi:10.1007/978-3-030-44914-8_23.

49 Romain Thomas. Lief - library to instrument executable formats. https://lief.quarkslab.com, April 2017.

50 Christian Urban. The Isabelle cookbook: A gentle tutorial for programming Isabelle, 2019.

51 Freek Verbeek, Joshua A. Bockenek, Zhoulai Fu, and Binoy Ravindran. Formally verified lifting of c-compiled x86-64 binaries. In Ranjit Jhala and Isil Dillig, editors, *PLDI*, pages 934–949. ACM, 2022. doi:10.1145/3519939.3523702.

52 Fish Wang and Yan Shoshitaishvili. Angr - the next generation of binary analysis. In *IEEE SecDev*, pages 8–9. IEEE Computer Society, 2017. doi:10.1109/SecDev.2017.14.

53 Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Trans. Software Eng.*, 47(11):2504–2519, 2021. doi:10.1109/TSE.2019.2953709.

54 Mark Wassell, Alasdair Armstrong, Neel Krishnaswami, and Peter Sewell. Mechanised metatheory for the sail isa specification, 2018.

55 Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The RISC-V instruction set manual, volume i: User-level isa, version 2.0. Technical Report UCB/EECS-2014-54, University of California, Berkeley, 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html.

56 Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. Armor: fully verified software fault isolation. In Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and Sebastian Fischmeister, editors, *EMSOFT*, pages 289–298. ACM, 2011. doi:10.1145/2038642.2038687.

Spegion: Implicit and Non-Lexical Regions with Sized Allocations

Jack Hughes
School of Computing, University of Kent, Canterbury, UK

Michael Vollmer
School of Computing, University of Kent, Canterbury, UK

Mark Batty
School of Computing, University of Kent, Canterbury, UK

Abstract

Region based memory management is a powerful tool designed with the goal of ensuring memory safety statically. The region calculus of Tofte and Talpin is a well known example of a region based system, which uses regions to manage memory in a stack-like fashion. However, the region calculus is lexically scoped and requires explicit annotation of memory regions, which can be cumbersome for the programmer. Other systems have addressed non-lexical regions, but these approaches typically require the use of a substructural type system to track the lifetimes of regions. We present Spegion, a language with implicit non-lexical regions, which provides these same memory safety guarantees for programs that go beyond using memory allocation in a stack-like manner. We are able to achieve this with a concise syntax, and without the use of substructural types, relying instead on an effect system to enforce constraints on region allocation and deallocation. These regions may be divided into sub-regions, i.e., **Sp**littable r**Egion**s, allowing fine grained control over memory allocation. Furthermore, Spegion permits *sized* allocations, where each value has an associated size which is used to ensure that regions are not over-allocated into. We present a type system for Spegion and prove it is type safe with respect to a small-step operational semantics.

2012 ACM Subject Classification Software and its engineering → Allocation / deallocation strategies; Theory of computation → Type theory

Keywords and phrases Regions, Type Systems, Effect Systems, Programming Languages, Memory

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.15

Related Version *Full Version*: `https://arxiv.org/abs/2506.02182`

Funding This work is supported by the EPSRC under grants *EP/X021173/1* and *EP/V000470/1*.

1 Introduction

Writing error-free code is hard. Writing error-free systems code is even harder. Possibly the most common source of bugs in systems programming code is the traditional approach of manual memory management. Use-after-free bugs, where a program attempts to access memory that has been deallocated, are pervasive in C code, and can lead to undefined behaviour, crashes, and security vulnerabilities. Nevertheless, manual memory management is still widely used in systems code as it provides the programmer with fine-grained control over memory usage. A vast swathe of research has been dedicated to the problem of memory safety in systems programming languages. One approach is to use a garbage collector to automatically manage memory. However, garbage collectors can introduce unpredictable pauses in the execution of a program, which is unacceptable in systems programming where performance is critical. Another approach is to ensure memory safety statically using a type system, one of the most well-studied verification tools in computer science.

© Jack Hughes, Michael Vollmer, and Mark Batty;
licensed under Creative Commons License CC-BY 4.0

The region calculus of Tofte and Talpin [29] is a type system designed for ensuring memory safety in ML [27] through the use of memory *regions*. Memory is divided into regions which are allocated in a stack-like manner, using the language's **letregion** construct. This allows deallocation to be done in a single operation by popping the topmost region from the stack:

letregion ρ **in** e

The memory region ρ is created and pushed onto the region stack. After the evaluation of **letregion**, ρ is popped from the stack, deallocating any memory allocated during the evaluation of e. Region based memory management has since been applied to languages other than ML, such as Java [6,9] and Prolog [21]. However, these region calculi are lexically scoped, restricting the programmer from expressing more complex memory usage patterns that may arise in code. Lexically-scoped regions are tied to the static structure of the program and are deallocated in a stack-like manner. This limits flexibility in memory management, as regions cannot be split or partially deallocated. Consider the following C program, for example: Here

```
struct task {int n; struct task* next;};

struct task* build_tasks () {
    struct task* t      = malloc(sizeof(struct task));
    t->next             = malloc(sizeof(struct task));
    t->next->next       = malloc(sizeof(struct task));
    t->next->next->n    = 1;
    t->next->next->next = NULL;
    return t;
  }

int main () {
  struct task* t_head = build_tasks();
  struct task* t_last = t_head->next->next;
  free(t_head->next);
  free(t_head);
  return t_last->n;
}
```

Figure 1 Non-lexically-scoped memory use.

we have a linked list of tasks, where each task has a pointer to the next task. The function `build_tasks` allocates three tasks and links them together. The main function then frees the first two tasks, leaving the third task dangling. Traditional region calculi are unable to express this pattern as the regions are allocated and de-allocated non-lexically.

There has been prior work exploring the safe use of non-lexically scoped regions, though this work has often focused on inferring region annotations for functional programs, or involved the use of substructural types (requiring that code is written to use memory linearly or avoid aliasing, for example), or imposed constraints on the runtime of programs like requiring reference counting. This research is covered in detail in Section 6.2.

We present an approach to non-lexically scoped region which allows for the same memory safety guarantees as prior systems, without the need for linear or substructural types and with a concise syntax. We refer to these as *implicit* regions, as they are not first-class values in the language's syntax and are not tied to the lexical structure of the program. Our system, named Spegion, removes the need for first class regions in the language's source syntax (with the exception of polymorphic constructs). Instead, when allocating a piece of data, the programmer needs only to provide an existing value in the region they wish to allocate the new value into. For example,

let $x =$ **newrgn in**
  v **at** x
  freergn x;

The **newrgn** construct returns a null pointer for a new region, which is then used to allocate v into that region. The type system then ensures that the region associated with x is live when v is allocated into it. Eliminating first-class regions from the syntax provides natural compatibility with C-like languages, where regions are not first-class values. Spegion sidesteps the need to retrofit complex region syntax onto C by using a special syntactic form for allocation (v **at** x) to determine which region to allocate a value into. Moreover, implicit regions eliminate the need for substructural type constraints, a common source of complexity in region type systems and a feature which C does not support.

Our non-lexical approach to regions in Spegion also permits the splitting of regions into sub-regions, allowing unused memory to be reclaimed without deallocating the entire region. This allows the typing of patterns which are difficult to express, even in C.

Furthermore, our system introduces the notion of *sized* regions. Each value in our language has an associated size, which is an abstract representation of the amount of memory that the value occupies. We use this information to ensure that regions are not over-allocated into, as each region has an associated maximum size constraint. These constraints are provided explicitly by the programmer as annotations on region creation and allocation sites, giving them fine-grained control over memory usage in their program. In the previous example, we allocated v into the region associated with x. This allocation carries the implicit constraint that the region of x must be of an arbitrary size. A more fine-grained allocation is possible using size annotations:

let $x =$ **newrgn** [10] **in**
  v [5] **at** x

Here, 10 is a size representing 10 abstract units of memory. The region associated with x has a size of 5 and we can only allocate into this region if the total size of allocations into the region does not exceed 10. In the above code, we allocate 5 units of memory into the region. as indicated by the size annotation on the allocation site. Contrarily, the following code would be invalid:

let $x =$ **newrgn** [10] **in**
  v [10] **at** x;
  v [5] **at** x

Since the sum of allocations (15) is greater than the region's bound size. This program is thus rejected by the type checker.

Spegion's size annotations are abstract units of memory, but the principle can be applied to concrete memory sizes. In performance critical applications, such as embedded systems, where the programmer has clear static constraints on the size of data structures, the programmer can use these size annotations to ensure that the program does not exceed the available memory.

Size annotations are an optional feature of Spegion, as unannotated regions and allocations default to being of unbounded size. These size constraints on allocations can be enforced using an effect system, and present an opportunity for further research into type-level reasoning about memory usage.

1.1 Contributions

This paper makes the following contributions:

- We introduce Spegion, a novel core calculus and formal semantics for a language featuring implicit, non-lexical regions with explicit size constraints. Unlike traditional region calculi which are lexically scoped and require first-class region syntax, Spegion removes first-class regions from the source language syntax (except for polymorphic constructs), enabling natural compatibility with C-like languages.
- We develop a type system that leverages effects to ensure safe allocation into implicit regions, eliminating the need for substructural or linear types commonly required in prior non-lexical region systems such as Cyclone. This approach supports flexible memory management patterns including region splitting and partial deallocation, which are difficult to express in lexically scoped systems.
- We extend the concept of region-based memory management by introducing sized regions, where each region and allocation carries an abstract size annotation. This allows static enforcement of memory usage bounds via an effect system, providing fine-grained control over memory consumption and enabling static reasoning about memory usage in performance-critical applications.
- We provide a formal proof of type safety for Spegion, building on and adapting the syntactic proof techniques from Helsen and Thiemann's work on the region calculus.
- We demonstrate the expressiveness and practical applicability of Spegion through a series of illustrative examples, including a sketch of an extension combining refinement types with region sizes to further enhance static guarantees about memory usage.

1.2 Overview

The rest of the paper is structured as follows. Section 2 introduces the syntax and typing rules of Spegion. In Section 3 we present the dynamic semantics of the language as reduction rules. In Section 4 we present the type safety properties of the language in the form of progress and preservation theorems. Section 5 explores the potential applications of Spegion in several domains, including a sketch of how refinement types can be combined with our system to provide powerful static guarantees about memory usage. We also discuss the application of our system in the context of memory safety in C-like languages. Section 6 discusses related work, and Section 7 concludes.

2 Static Semantics

We define a core calculus for Spegion, drawing from the region calculus of [29] and the presentation by [17]. The syntax of types is given by:

$$\begin{aligned} \tau &::= \quad \alpha \mid \text{Int} \mid \text{Unit} \mid \text{Bool} \mid \text{Ref}\ \tau \mid \mu \xrightarrow{\varphi} \mu \mid \forall\{\alpha, \rho, \epsilon\}.\mu \xrightarrow{\varphi} \mu \\ \mu &::= \quad (\tau, \rho) \end{aligned} \tag{types}$$

Types τ are either type variables α, integers, units, booleans, references to other types, function types, or polymorphic types. A program expression in our calculus is assigned a "type-with-place", i.e., a pair of a type and a region, denoted by μ.

Function types are written as $\mu_1 \xrightarrow{\varphi} \mu_2$. Above the function arrow is an *arrow effect*, i.e. the latent effect that happens on application of the function, as in [29]. Polymorphic functions are typed by type schemes $\forall\{\alpha, \rho, \epsilon\}.\mu$ which bind type α, region ρ, and effect ϵ variables. We refer to these as kind-annotated type variables, however, in the syntax of our calculus we typically omit the kind annotations for brevity.

Kinds are given by the grammar:

$$\kappa ::= \quad \text{Type} \mid \kappa \to \kappa \mid \text{Region} \mid \text{Effect} \mid \text{Size} \qquad \text{(kinds)}$$

That is a kind is either a type, a function kind, a region, an effect, or a size. Kinding rules for types are given in Figure 2. These rules are mostly straightforward and ensure that type, regions, effects, and sizes are well-structured.

$$\frac{\text{Type} = K(\alpha)}{K \vdash \alpha : \text{Type}}\ \kappa\text{-VAR} \qquad \frac{K, \alpha : \text{Type}, \rho : \text{Region}, \epsilon : \text{Effect} \vdash \tau : \text{Type}}{K \vdash \forall\{\alpha, \rho, \epsilon\}.\tau : \text{Type}}\ \kappa\text{-FORALL} \qquad \frac{\text{Region} = K(\rho)}{K \vdash \rho : \text{Region}}\ \kappa\text{-REG}$$

$$\frac{}{K \vdash (\to) : \text{Type} \to \text{Effect} \to \text{Type} \to \text{Type}}\ \kappa\text{-}\to \qquad \frac{K \vdash \kappa_1 : \text{Type} \qquad K \vdash \tau : \text{Type}}{K \vdash (\kappa_1 \to \kappa_2)\ \tau : \kappa_2}\ \kappa\text{-APP}_1$$

$$\frac{K \vdash \kappa_1 : \text{Effect} \qquad K \vdash \varphi : \text{Effect}}{K \vdash (\kappa_1 \to \kappa_2)\ \varphi : \kappa_2}\ \kappa\text{-APP}_2 \qquad \frac{}{K \vdash \text{Int} : \text{Type}}\ \kappa\text{-INT} \qquad \frac{}{K \vdash \text{Unit} : \text{Type}}\ \kappa\text{-UNIT}$$

$$\frac{K \vdash \tau : \text{Type}}{K \vdash \text{Ref}\ \tau : \text{Type}}\ \kappa\text{-REF} \qquad \frac{}{K \vdash \text{Bool} : \text{Type}}\ \kappa\text{-BOOL} \qquad \frac{}{K \vdash s : \text{Size}}\ \kappa\text{-SIZE}$$

$$\frac{K \vdash \kappa_1 : \text{Size} \qquad K \vdash \kappa_2 : \text{Size}}{K \vdash \kappa_1\ op\ \kappa_2 : \text{Size}}\ \kappa\text{-OP} \qquad \frac{K \vdash \tau : \text{Type} \qquad K \vdash \rho : \text{Region}}{K \vdash (\tau, \rho) : \text{Type}}\ \kappa\text{-TyWithPlace}$$

$$\frac{}{K \vdash_\kappa \{\bot\} : \text{Effect}}\ \kappa\text{-BOT} \qquad \frac{K \vdash \varphi_1 : \text{Effect} \quad K \vdash \varphi_2 : \text{Effect}}{K \vdash \varphi_1 \times \varphi_2 : \text{Effect}}\ \kappa\text{-}\times \qquad \frac{K \vdash \varphi_1 : \text{Effect} \quad K \vdash \varphi_2 : \text{Effect}}{K \vdash \varphi_1 \sqcup \varphi_2 : \text{Effect}}\ \kappa\text{-}\sqcup$$

$$\frac{K \vdash s : \text{Size} \qquad K \vdash \rho : \text{Region}}{K \vdash \{\textbf{alloc}\ s\ \rho\} : \text{Effect}}\ \kappa\text{-ALLOC} \qquad \frac{K \vdash \rho : \text{Region} \qquad K \vdash s : \text{Size}}{K \vdash \{\textbf{fresh}\ \rho\ s\} : \text{Effect}}\ \kappa\text{-FRESH}$$

$$\frac{K \vdash \rho : \text{Region} \qquad K \vdash s : \text{Size} \qquad K \vdash \rho' : \text{Region}}{K \vdash \{\textbf{split}\ \rho\ s\ \rho'\} : \text{Effect}}\ \kappa\text{-SPLIT}$$

$$\frac{K \vdash \rho : \text{Region}}{K \vdash \{\textbf{free}\ \rho\} : \text{Effect}}\ \kappa\text{-FREE} \qquad \frac{\text{Effect} = K(\epsilon)}{K \vdash \{\epsilon\} : \text{Effect}}\ \kappa\text{-}\epsilon \qquad \frac{K \vdash \epsilon : \text{Effect} \qquad K \vdash \varphi : \text{Effect}}{K \vdash \{\textbf{rec}\ \epsilon\ \varphi\} : \text{Effect}}\ \kappa\text{-REC}$$

Figure 2 Kinding rules in Spegion.

2.1 Effects and Sizes

An effect φ describes the sequence of actions that are performed on a region when an expression is evaluated. Many expressions in the calculus convey some effect, which form part of the expression's typing judgement. The following grammar defines these actions:

$$\begin{aligned}\varphi ::= \quad & \varphi \times \varphi \mid \{\bot\} \mid \{\textbf{fresh } \rho\ s\} \mid \{\textbf{free } \rho\} \mid \{\textbf{split } \rho\ s\ \rho'\} \mid \{\textbf{alloc } s\ \rho\} \mid \varphi \sqcup \varphi \\ \mid \quad & \{\epsilon\} \mid \{\textbf{rec } \epsilon\ \varphi\}\end{aligned} \tag{effects}$$

Effects are composed using the $\times$ operator, which describes the sequential composition of two effects. This operator also ensures constraints on memory usage are respected when two effects are composed. The $\{\bot\}$ effect describes an empty effect, i.e., no actions are performed, and is used in the typing of expressions which are pure computations. The creation of a fresh region is denoted by $\{\textbf{fresh } \rho\ s\}$, where ρ is a fresh region name and s is the region's *size*.

A size is an abstract unit of memory. For simplicity, we consider sizes to be natural numbers, extended with a special size ω which represents an unknown size. Sizes thus comprise an extended natural numbers preordered semiring $(\overline{\mathbb{N}}, +, \cdot, \dot{-}, 0, 1, \sqsubseteq)$ where $\overline{\mathbb{N}} = \mathbb{N} \cup \{\omega\}$, such that $\forall n \in \overline{\mathbb{N}}. n + \omega = \omega = \omega + n = \omega$. The preorder $\sqsubseteq$ is the standard preorder on $\mathbb{N}$ with ω as the greatest element. The semiring is also equipped with a truncated subtraction (or *monus*) operation $\dot{-}$, given by Definition 2.1:

▶ **Definition 2.1** (Monus ($\dot{-}$)). For any two numbers $n, n' \in \overline{\mathbb{N}}$, $\dot{-}$ is defined:

$$n \dot{-} n' = \begin{cases} n - n' & n, n' \sqsubset \omega \wedge n \sqsupseteq n', \\ 0 & n, n' \sqsubset \omega \wedge n \sqsubset n' \\ n & n' = 0 \\ 0 & n \sqsubset n' \wedge n' = \omega, \\ \omega & \text{if } n = \omega \end{cases}$$

The typing rules do not make use of subtraction, however, it is required during evaluation.

The effect $\{\textbf{free } \rho\}$ describes the freeing of a region ρ, which deallocates all memory in the region. The creation of sub-regions is denoted by $\{\textbf{split } \rho\ s\ \rho'\}$, where ρ' is a fresh region name for a sub-region of ρ with size s. Allocation of a value into a region is described by $\{\textbf{alloc } s\ \rho\}$, where s is the size of the value being allocated, and ρ is the region it is being allocated into. If statements introduce branching into our calculus, with the different cases potentially having different effects. This is captured by the $\{\varphi_1 \sqcup \varphi_2\}$ effect, where φ_1 denotes the effect of the true branch and φ_2 the effect of the false branch. Recursion is described in effects through the $\{\epsilon\}$ and $\{\textbf{rec } \epsilon\ \varphi\}$ effects. The former denotes the usage of a recursive function definition, while the latter captures the latent effect of this function upon application. We explain further when we discuss the typing rules for recursive functions.

2.2 Syntax and Typing

The term syntax of our calculus comprises the λ-calculus, with the addition of references, sequential composition, polymorphic recursion, as well as constructs for region manipulation. These include expressions for creating, freeing, and splitting regions, as well as primitives for allocating data into regions and copying data between regions. The full syntax is given by the following grammar, which divides constructs between values v and expressions e:

$$\begin{aligned} v ::= \quad & n \mid \text{true} \mid \text{false} \mid \Lambda\{\alpha, \rho, \epsilon\}.\lambda x.e \mid \lambda x.e \mid () \mid l_\rho \\ e ::= \quad & x \mid l_\rho \mid v\ [s]\ \textbf{at}\ e \mid e\ e \mid \textbf{ref}\ e \mid\ !e \mid e := e \mid e;\ e \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \\ \mid \quad & \textbf{let}\ x = e\ \textbf{in}\ e \mid e\ @\ \mu \mid \textbf{let}\ f = \textbf{fix}(f, (\Lambda\{\alpha, \rho, \epsilon\}.\lambda x.e_1)\ [s]\ \textbf{at}\ e_2)\ \textbf{in}\ e_3 \\ \mid \quad & \textbf{newrgn}\ [s] \mid \textbf{freergn}\ e \mid \textbf{split}\ [s]\ e \mid \textbf{copy}\ e\ \textbf{into}\ e \end{aligned} \tag{terms}$$

The syntax of values v includes integers n, booleans true and false, polymorphic functions $\Lambda\{\alpha, \rho, \epsilon\}.\lambda x.e$, functions $\lambda x.e$, unit $()$, and locations l_ρ. Expressions e include variables x, locations l_ρ, value allocations $v\ [s]\ \textbf{at}\ e$, function application $e\ e$, reference creation $\textbf{ref}\ e$,

$$\frac{}{K \mid \Gamma \mid \Sigma \vdash c : \mathrm{Int}}\ \textsc{t-int} \qquad \frac{}{K \mid \Gamma \mid \Sigma \vdash () : \mathrm{Unit}}\ \textsc{t-unit}$$

$$\frac{}{K \mid \Gamma \mid \Sigma \vdash \mathrm{true} : \mathrm{Bool}}\ \textsc{t-true} \qquad \frac{}{K \mid \Gamma \mid \Sigma \vdash \mathrm{false} : \mathrm{Bool}}\ \textsc{t-false}$$

$$\frac{\tau = \Sigma(l_\rho) \qquad K \vdash \tau : \mathrm{Type}}{K \mid \Gamma \mid \Sigma \vdash l_\rho : \tau}\ \textsc{t-loc} \qquad \frac{K \vdash \mu_1 : \mathrm{Type} \quad K \mid \Gamma, x : \mu_1 \mid \Sigma \vdash e : \mu_2 \mid \varphi}{K \mid \Gamma \mid \Sigma \vdash \lambda x.e : \mu_1 \xrightarrow{\varphi} \mu_2}\ \textsc{t-}\lambda$$

$$\frac{K, \alpha : \mathrm{Type}, \rho : \mathrm{Region}, \epsilon : \mathrm{Effect} \mid \Gamma, x : \mu_1 \mid \Sigma \vdash e : \mu_2 \mid \varphi}{K \mid \Gamma \mid \Sigma \vdash \Lambda\{\alpha, \rho, \epsilon\}.\lambda x.e : \forall\{\alpha : \mathrm{Type}, \rho : \mathrm{Region}, \epsilon : \mathrm{Effect}\}.\mu_1 \xrightarrow{\varphi} \mu_2}\ \textsc{t-}\Lambda$$

Figure 3 Value typing rules in Spegion.

dereferencing $!e$, assignment $e := e$, sequential composition $e;\ e$, conditionals **if** e **then** e **else** e, let-binding **let** $x = e$ **in** e, type applications $e\ @\ \mu$, and recursive function definitions **let** $f =$ **fix**$(f, (\Lambda\{\alpha, \rho, \epsilon\}.\lambda x.e_1)\ [s]$ **at** $e_2)$ **in** e_3. Region manipulation constructs include creating a new region **newrgn** $[s]$, freeing a region **freergn** e, splitting a sub-region from another region **split** $[s]\ e$, and copying data between regions **copy** e **into** e.

Expressions are typed by the judgement:

$$K \mid \Gamma \mid \Sigma \vdash e : (\tau, \rho) \mid \varphi$$

assigning an expression e the type τ in region ρ with effect φ. A context of kind-annotated type variables K provides a mapping from type variables to kinds, given by:

$$K ::= \quad \emptyset \mid K, \alpha : \mathrm{Type} \mid K, \rho : \mathrm{Region} \mid K, \epsilon : \mathrm{Effect} \qquad \text{(kinding contexts)}$$

That is, a context may be empty $\emptyset$, extended with a type variable $\alpha : \mathrm{Type}$, a region variable $\rho : \mathrm{Region}$, or an effect variable $\epsilon : \mathrm{Effect}$. Free variable contexts Γ are given by:

$$\Gamma ::= \quad \emptyset \mid \Gamma, x : \mu \qquad \text{(free variable contexts)}$$

A store typing context Σ is a mapping from locations to types, defined:

$$\Sigma ::= \quad \emptyset \mid \Sigma, l_\rho : \tau \qquad \text{(store typing)}$$

That is l_ρ is a location in region ρ with type τ. Figure 4 provides typing rules for the expressions. Values, which are non-computational objects, are typed by a separate judgement

$$K \mid \Gamma \mid \Sigma \vdash v : \tau$$

This judgement is similar to that of expressions but does not include an effect. Figure 3 gives the typing rules for values, relating terms to types.

The rules for integers (T-INT), unit (T-UNIT), and bools (T-TRUE, T-FALSE) are standard. For (T-λ), the type of the argument x is added to the variable context Γ and the expression e is typed. The type of the function is then a function type from the type of the argument to the type of the body. The effect of the function is the latent effect of the body. A location l_ρ is typed by (T-LOC), where the location's type τ is looked up in the store typing Σ. Finally, polymorphic functions are typed by (T-Λ), which also adds the kind-annotated type, region, and effect variables to the kind variable context K.

$$\text{T-USE-VAL}\ \frac{K \mid \Gamma \mid \Sigma \vdash l_\rho : \tau}{K \mid \Gamma \mid \Sigma \vdash l_\rho : (\tau, \rho) \mid \{\bot\}} \qquad \frac{K \mid \Gamma \mid \Sigma \vdash e : (\tau', \rho) \mid \varphi \qquad K \mid \Gamma \mid \Sigma \vdash v : \tau \qquad K \vdash s : \text{Size}}{K \mid \Gamma \mid \Sigma \vdash v\ [s]\ \textbf{at}\ e : (\tau, \rho) \mid \varphi \times \{\textbf{alloc}\ s\ \rho\}}\ \text{T-VAL}$$

$$\frac{s \sqsupseteq 1 \qquad K, \rho : \text{Region} \vdash s : \text{Size}}{K \mid \Gamma \mid \Sigma \vdash \textbf{newrgn}\ [s] : (\text{Unit}, \rho) \mid \{\textbf{fresh}\ \rho\ s\} \times \{\textbf{alloc}\ 1\ \rho\}}\ \text{T-NEWRGN}$$

$$\frac{K \mid \Gamma \mid \Sigma \vdash e : (\tau, \rho) \mid \varphi}{K \mid \Gamma \mid \Sigma \vdash \textbf{freergn}\ e : (\text{Unit}, \rho_{\text{glob}}) \mid \varphi \times \{\textbf{free}\ \rho\}}\ \text{T-FREERGN}$$

$$\frac{K, \rho' : \text{Region} \mid \Gamma \mid \Sigma \vdash e : (\tau, \rho) \mid \varphi \qquad s \sqsupseteq 1 \qquad K \vdash s : \text{Size}}{K \mid \Gamma \mid \Sigma \vdash \textbf{split}\ [s]\ e : (\text{Unit}, \rho') \mid \varphi \times \{\textbf{split}\ \rho\ s\ \rho'\} \times \{\textbf{alloc}\ 1\ \rho'\}}\ \text{T-SPLIT}$$

$$\frac{K \mid \Gamma \mid \Sigma \vdash e_1 : (\tau, \rho) \mid \varphi_1 \qquad K \mid \Gamma \mid \Sigma \vdash e_2 : (\tau', \rho') \mid \varphi_2}{K \mid \Gamma \mid \Sigma \vdash \textbf{copy}\ e_1\ \textbf{into}\ e_2 : (\tau, \rho') \mid \varphi_1 \times \varphi_2 \times \{\textbf{alloc}\ 1\ \rho'\}}\ \text{T-COPY}$$

$$\text{T-VAR}\ \frac{K \vdash \mu : \text{Type}}{K \mid \Gamma, x : \mu \mid \Sigma \vdash x : \mu \mid \{\bot\}} \qquad \text{T-LET}\ \frac{K \mid \Gamma \mid \Sigma \vdash e_1 : \mu_1 \mid \varphi_1 \qquad K \mid \Gamma, x : \mu_1 \mid \Sigma \vdash e_2 : \mu_2 \mid \varphi_2}{K \mid \Gamma \mid \Sigma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \mu_2 \mid \varphi_1 \times \varphi_2}$$

$$\frac{K \mid \Gamma \mid \Sigma \vdash e_1 : (\text{Bool}, \rho) \mid \varphi_1 \qquad K \mid \Gamma \mid \Sigma \vdash e_2 : \mu_2 \mid \varphi_2 \qquad K \mid \Gamma \mid \Sigma \vdash e_3 : \mu_2 \mid \varphi_3}{K \mid \Gamma \mid \Sigma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \mu_2 \mid \varphi_1 \times (\varphi_2 \sqcup \varphi_3)}\ \text{T-IF}$$

$$\text{T-REF}\ \frac{K \mid \Gamma \mid \Sigma \vdash e : (\tau, \rho) \mid \varphi}{K \mid \Gamma \mid \Sigma \vdash \textbf{ref}\ e : (\text{Ref}\ \tau, \rho) \mid \varphi \times \{\textbf{alloc}\ 1\ \rho\}} \qquad \text{T-DEREF}\ \frac{K \mid \Gamma \mid \Sigma \vdash e : (\text{Ref}\ \tau, \rho) \mid \varphi}{K \mid \Gamma \mid \Sigma \vdash !e : (\tau, \rho) \mid \varphi}$$

$$\text{T-ASSIGN}\ \frac{K \mid \Gamma \mid \Sigma \vdash e_1 : (\text{Ref}\ \tau, \rho') \mid \varphi_1 \qquad K \mid \Gamma \mid \Sigma \vdash e_2 : \mu \mid \varphi_2}{K \mid \Gamma \mid \Sigma \vdash e_1 := e_2 : (\text{Unit}, \rho_{\text{glob}}) \mid \varphi_1 \times \varphi_2} \qquad \text{T-SEQ}\ \frac{K \mid \Gamma \mid \Sigma \vdash e_1 : (\text{Unit}, \rho) \mid \varphi_1 \qquad K \mid \Gamma \mid \Sigma \vdash e_2 : \mu \mid \varphi_2}{K \mid \Gamma \mid \Sigma \vdash e_1; e_2 : \mu \mid \varphi_1 \times \varphi_2}$$

$$\frac{K \mid \Gamma \mid \Sigma \vdash e_1 : (\mu_1 \xrightarrow{\varphi} \mu_2, \rho) \mid \varphi_1 \qquad K \mid \Gamma \mid \Sigma \vdash e_2 : \mu_1 \mid \varphi_2}{K \mid \Gamma \mid \Sigma \vdash e_1\ e_2 : \mu_2 \mid \varphi_1 \times \varphi_2 \times \varphi}\ \text{T-APP}$$

$$\frac{K \mid \Gamma \mid \Sigma \vdash e : \forall\{\alpha, \rho, \epsilon : \text{Effect}\}.\mu_1 \xrightarrow{\varphi} \mu_2 \mid \{\bot\} \qquad K \vdash (\tau, \rho') : \text{Type}}{K \mid \Gamma \mid \Sigma \vdash e\ @\ (\tau, \rho', \varphi) : [\alpha \mapsto \tau, \rho \mapsto \rho'](\mu_1 \xrightarrow{\varphi} \mu_2) \mid \{\bot\}}\ \text{T-TYAPP}$$

$$\text{T-FIX}\ \frac{\begin{array}{c}\mu_f = (\forall\{\alpha, \rho, \epsilon\}.(\alpha, \rho) \xrightarrow{\{\epsilon\}} \mu_1, \rho_f) \\ K \mid \Gamma, f : \mu_f \mid \Sigma \vdash (\Lambda\{\alpha, \rho, \epsilon\}.\lambda x.e_1)\ [s]\ \textbf{at}\ e_2 : (\forall\{\alpha, \rho, \epsilon\}.(\alpha, \rho) \xrightarrow{\varphi} \mu_1, \rho_f) \mid \varphi_1 \\ \varphi' = [\{\epsilon\} \mapsto \{\textbf{rec}\ \epsilon\ \varphi\}]\varphi \qquad K \mid \Gamma, f : (\forall\{\alpha, \rho, \epsilon\}.(\alpha, \rho) \xrightarrow{\varphi'} \mu_1, \rho_f) \mid \Sigma \vdash e_3 : \mu_2 \mid \varphi_2\end{array}}{K \mid \Gamma \mid \Sigma \vdash \textbf{let}\ f = \textbf{fix}(f, (\Lambda\{\alpha, \rho, \epsilon\}.\lambda x.e_1)\ [s]\ \textbf{at}\ e_2)\ \textbf{in}\ e_3 : \mu_2 \mid \varphi_1 \times \varphi_2}$$

Figure 4 Expression typing rules in Spegion.

In the typing of expressions, the rule for variables (T-VAR) is straightforward, if the variable is present in the free variable context Γ with type μ then $x : \mu$. The rule for locations (T-USE-VAL) is also straightforward, if the location l_ρ is present in the store typing Σ with type τ then $l_\rho : (\tau, \rho)$.

Creating a new region is handled by the (T-NEWRGN) rule. The **newrgn** construct allocates a new region of size s and has type (Unit, ρ) where ρ is the freshly allocated region. This behaviour is captured in the rule's effect $\{\textbf{fresh}\ \rho\ s\}$, which binds a fresh region name ρ to the size s. Evaluating this expression returns a unit value, which acts as a pointer into the region, allowing this region to be referenced in the subsequent program. This eliminates the need for syntactic region variables in all non-polymorphic code. The allocation of this null value is described in the second part of the effect $\{\textbf{alloc}\ 1\ \rho\}$, i.e., allocate a value of size 1 into ρ (since a value cannot be 0-sized). Thus, we have the additional constraint $s \sqsupseteq 1$ ensuring regions must be able to allocate at least 1 unit of memory.

The rule for freeing a region (T-FREERGN) deletes the region ρ associated with the expression e, freeing its associated memory locations. This yields an effect comprised of the effect of typing e (φ) followed by the effect $\{\textbf{free}\ \rho\}$. Evaluating a **freergn** expression returns a pointer to value of type Unit in the global region ρ_{glob} which is always available.

A sub-region may be split from a region using the (T-SPLIT) rule. As with **freergn**, the region to be split is associated with the sub-expression e. The programmer-defined size s determines the size of the new sub-region, represented by the effect $\{\textbf{split}\ \rho\ s\ \rho'\}$. As in (T-NEWRGN), a null pointer into the new region is created (thus the rule also has an **alloc** effect), and the overall expression has the type (Unit, ρ'), where ρ' is the fresh sub-region.

Copying data between regions is handled by the (T-COPY) rule. The behaviour of **copy** can also be simulated in terms of (T-ASSIGN) and (T-VAL). However, we include the **copy** construct as a useful primitive. The expression **copy** e_1 **into** e_2 copies the location obtained from evaluating e_1 from its region ρ into the region ρ' associated with e_2. The effect of this operation is the composition of the effects obtained from typing e_1 and e_2, followed by the effect $\{\textbf{alloc}\ 1\ \rho'\}$. Locations have a size of 1, thus this effect conveys the creation of a new location in ρ' pointing to location value.

Allocating a value into a region is handled by the T-VAL rule. A programmer is required to annotate the allocation of a value with a size, which is used in the effect $\{\textbf{alloc}\ s\ \rho\}$. Unlike [29], where the region is explicitly passed as an argument to the allocation primitive, our rule allocates the value into the region associated with the sub-expression e. For example, consider the following trivial program:

$$\textbf{let}\ x = \textbf{newrgn}\ [3]\ \textbf{in}\ ()\ [2]\ \textbf{at}\ x$$

which creates a new region of size 3 and allocates a unit value of size 2 into it. This program has the type (Int, ρ) with the effect:

$$\{\textbf{fresh}\ \rho\ 3\} \times \{\textbf{alloc}\ 1\ \rho\} \times \{\textbf{alloc}\ 2\ \rho\}$$

conveying the creation and allocation into a region without ever giving it an explicit name in the program. This allows our calculus to model region-based memory management in languages where these first-class regions are not present in the source language.

Typing rules for let-binding (T-LET), creating a reference (T-REF), dereferencing (T-DEREF), assigning to a reference (T-ASSIGN), and sequential composition (T-SEQ) thus handle the sequencing of the effects of their sub-expressions, but are otherwise straightforward. Similarly, application (T-APP) sequences the effects of the sub-expressions, followed by the function type's latent effect. If statements (T-IF) are typed using a join effect $\{\varphi_1 \sqcup \varphi_2\}$, which is the effect of the true branch φ_1 and the false branch φ_2 combined.

Recursive function definitions are typed by the (T-FIX) rule. The body of the definition is typed with an effect variable in place of the functions actual latent effect. In the rule's third premise, this variable is then substituted for the recursive effect $\{\textbf{rec}\ \epsilon\ \varphi\}$ yielding φ'. In the typing of e_3, f is then bound with the same function type as before but with this new latent effect φ'. This recursive effect conveys that the the effect φ may be repeated an unbounded number of times. This is akin to the notion of "Kleene star effects" [23, 24] and iterable sequential effects [15].

A consequence of recursion is that regions which appear freely in e_1 must be capable of allocating an unbounded amount of memory. This constraint is enforced at the point where the recursive variable effect is composed with another effect in the typing of the function body (via $\times$), since the presence of a ϵ in φ indicates that the function recurses. We provide the definition of valid effect composition in detail in Section 2.3.

Recursive variable usage thus requires that the variable's type scheme be instantiated via the (T-TYAPP) rule, which instantiates the type scheme and region at the type and region provided by the programmer. This is the only rule in the syntax where a region name is required, which can be obtained using a primitive operator regionOf(e).

2.3 Effect Composition

The typing rules above define the individual effects of expressions. However, taken alone an effect does not provide a complete picture of the memory usage of a program. For example, the effect of an allocation is the $\{\textbf{alloc}\ s\ \rho\}$ effect preceded by the effect of the expression which identifies the region being allocated into. However, at the point of typing this allocation, there is no way of knowing if the allocation described by this effect is valid – this can only be known when the effect is composed with the effect which describes the creation of ρ. We describe below the behaviour of the $\times$ operator, which composes two effects. Effects in our calculus are sequential, thus the effect $\varphi \times \varphi'$ describes the effect of φ followed by the effect of φ'. Figure 5 provides the rules for valid effect compositions.

In the ($\times$-$\bot$) and ($\times$-FRESH) rules, composing an effect with $\{\bot\}$ or $\{\textbf{fresh}\ \rho\ s\}$ is always valid. In the latter case we thus assume that SPEGION always has the ability to allocate new regions. An effect which frees a region $\{\textbf{free}\ \rho\}$ is valid only if the region is not already freed in the preceding effect ($\times$-FREE).

Effects which handle allocation must consider three cases. The first ($\times$-FRESHALLOC) is when the entire lifetime of the region up until the point of the current allocation is described in the preceding effects. In this case, the size of the new allocation effect s' must not exceed the current total size of allocations in the ρ. This is enforced by the constraint $\exists s''.s'' + s' + \text{sumAllocs}(\rho, \varphi) \sqsubseteq s$, i.e., there exists some leftover space s'' after adding the size of the new allocation s' to the total size of allocations in ρ. Definition 2.2 defines the function calculating this total size, mapping a region and effect to a size.

▶ **Definition 2.2** (Total Size of Allocations)**.**

$$\text{sumAllocs}(\rho, \varphi) = \begin{cases} s + \text{sumAllocs}(\rho, \varphi') & \varphi = \varphi' \times \{\textbf{alloc}\ s\ \rho\} \\ s + \text{sumAllocs}(\rho, \varphi') & \varphi = \varphi' \times \{\textbf{split}\ \rho\ s\ \rho'\} \wedge \\ s + s' & \varphi = \varphi' \times \varphi'' \wedge \\ & s\ = \text{sumAllocs}(\rho, \varphi') \wedge \\ & s' = \text{sumAllocs}(\rho, \varphi'') \\ \max(s, s') & \varphi = \varphi' \sqcup \varphi'' \wedge \\ & s\ = \text{sumAllocs}(\rho, \varphi') \wedge \\ & s' = \text{sumAllocs}(\rho, \varphi'') \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{}{\varphi \times \{\bot\}}\ \times\text{-}\bot \qquad \frac{}{\varphi \times \{\mathbf{fresh}\ \rho\ s\}}\ \times\text{-Fresh} \qquad \frac{\{\mathbf{free}\ \rho\} \notin \varphi}{\varphi \times \{\mathbf{free}\ \rho\}}\ \times\text{-Free}$$

$$\times\text{-FreshAlloc}\ \frac{\exists s''.s' + s'' + \mathrm{sumAllocs}(\rho, \varphi) \sqsubseteq s}{\{\mathbf{fresh}\ \rho\ s\} \times \varphi \times \{\mathbf{alloc}\ s'\ \rho\}} \qquad \times\text{-Alloc}\ \frac{\{\mathbf{free}\ \rho\} \notin \varphi \qquad \{\mathbf{fresh}\ \rho\ s'\} \notin \varphi}{\varphi \times \{\mathbf{alloc}\ s\ \rho\}}$$

$$\times\text{-SplitAlloc}\ \frac{\exists s''.s' + s'' + \mathrm{sumAllocs}(\rho, \varphi) \sqsubseteq s}{\{\mathbf{split}\ \rho\ s\ \rho'\} \times \varphi \times \{\mathbf{alloc}\ s'\ \rho\}} \qquad \times\text{-FreshSplit}\ \frac{\exists s''.s' + s'' + \mathrm{sumAllocs}(\rho, \varphi) \sqsubseteq s}{\{\mathbf{fresh}\ \rho\ s\} \times \varphi \times \{\mathbf{split}\ \rho\ s'\ \rho'\}}$$

$$\times\text{-Split}\ \frac{\{\mathbf{free}\ \rho\} \notin \varphi}{\varphi \times \{\mathbf{split}\ \rho\ s\ \rho'\}} \qquad \times\text{-SplitSplit}\ \frac{\exists s''.s' + s'' + \mathrm{sumAllocs}(\rho, \varphi) \sqsubseteq s}{\{\mathbf{split}\ \rho\ s\ \rho'\} \times \varphi \times \{\mathbf{split}\ \rho'\ s'\ \rho''\}} \qquad \times\text{-}\sqcup\ \frac{\varphi_1 \times \varphi_2 \qquad \varphi_1 \times \varphi_3}{\varphi_1 \times (\varphi_2 \sqcup \varphi_3)}$$

$$\times\text{-VarR}\ \frac{\forall(\rho \times s) \in \mathrm{freeAllocs}(\varphi).s \sqsupseteq \omega}{\varphi \times \{\epsilon\}} \qquad \times\text{-VarL}\ \frac{\forall(\rho \times s) \in \mathrm{freeAllocs}(\varphi).s \sqsupseteq \omega}{\{\epsilon\} \times \varphi} \qquad \times\text{-Rec}\ \frac{\times[\epsilon \mapsto \{\bot\}]\varphi'}{\varphi \times \{\mathbf{rec}\ \epsilon\ \varphi'\}}$$

Figure 5 Effect composition rules in Spegion ($\times$).

A similar case applies when allocating into a sub-region where the entire lifetime of the region is described in the preceding effects ($\times$-SplitAlloc).

The second case for allocation ($\times$-Alloc) is when the region is not created in the preceding effect, i.e. $\varphi \times \{\mathbf{alloc}\ s\ \rho\}$ where $\{\mathbf{fresh}\ s'\ \rho\} \notin \varphi$. This situation arises when combining two effects in the typing of a sub-term of the expression where the region is created. In this case, the current total size of allocations in ρ is unknown, thus the permissibility of the current allocation effect is unknown. Thus, the allocation is simply accepted as valid with the knowledge that $\varphi \times \{\mathbf{alloc}\ s\ \rho\}$ will eventually be composed with some effect describing the preceding lifetime of ρ, at which point it will be validated by the previous case.

The rules for composing split effects $\{\mathbf{split}\ \rho\ s\ \rho'\}$ behave in a similar manner as allocation ($\times$-FreshSplit), ($\times$-SplitSplit), and ($\times$-Split). The first two rules handle the cases where the entire lifetime of the region is described in the preceding effects, whilst the last rule handles the case where this information is not available.

Composing a join effect $\varphi_2 \sqcup \varphi_3$ with another effect φ_1 is straightforward ($\times$-$\sqcup$). Since both branches of the program must be well-typed, the effect φ_1 is simply composed with both effects. If this is valid then so the composition of the join: $\varphi_1 \times \{\varphi_2 \sqcup \varphi_3\}$.

Effect variables ϵ occur as part of an effect when a recursive function definition is typed. Composition of these effects are handled by the ($\times$-VarR) and ($\times$-VarL) rules. A constraint of recursive function definitions is that regions which occur freely of the function body must be created with an unbounded size ω since it is not possible to know statically how many times the function will recurse. Regions which are created inside the function body itself may be given any size. The addition of a lightweight refinement types would allow us to express and type programs which allocate recursively into a free region under some constraints. We consider this idea in more depth in Section 5.2.

The free regions of the effects $\varphi \times \{\epsilon\}$ and $\{\epsilon\} \times \varphi$ are collected by the function freeAllocs, given by Definition 2.3. This function maps an effect to the set of allocations (and splits) of free regions which occur in the effect. A region is free if there is no $\{\mathbf{fresh}\ s\ \rho\}$ effect in φ.

This set takes the form of pairs of regions and sizes, where $\bot$ is used to indicate that the region was freed inside φ. Since regions of unbounded size are represented by size ω, the only sized allocations that are permitted must also be those of ω. This is enforced by the constraint $\forall(\rho \times s) \in \text{freeAllocs}(\varphi).s \sqsupseteq \omega$.

▶ **Definition 2.3** (Free Region Allocations of an Effect)**.** Given an effect φ, the allocations of regions which are free in φ is given by:

$$\text{freeAllocs}(\varphi) = \begin{cases} \{(\rho \times s)\} & \varphi = \{\textbf{alloc}\ s\ \rho\} \\ \{(\rho \times s)\} \cup \text{freeAllocs}(\varphi') & \varphi = \varphi' \times \{\textbf{alloc}\ s\ \rho\} \wedge \\ & \{\textbf{fresh}\ \rho\ s'\} \notin \varphi' \\ \{(\rho \times s)\} & \varphi = \{\textbf{split}\ \rho\ s\ \rho'\} \\ \{(\rho \times s)\} \cup \text{freeAllocs}(\varphi') & \varphi = \varphi' \times \{\textbf{split}\ \rho\ s\ \rho'\} \wedge \\ & \{\textbf{fresh}\ \rho\ s'\} \notin \varphi' \\ \{(\rho \times 0)\} & \varphi = \{\textbf{free}\ \rho\} \\ \{(\rho \times 0)\} \cup \text{freeAllocs}(\varphi') & \varphi = \varphi' \times \{\textbf{free}\ \rho\} \wedge \\ & \{\textbf{fresh}\ \rho\ s\} \notin \varphi' \\ \text{freeAllocs}(\varphi') & \varphi = \{\textbf{rec}\ \epsilon\ \varphi'\} \\ \text{freeAllocs}(\varphi') \cup \text{freeAllocs}(\varphi'') & \varphi = \varphi' \times \varphi'' \\ \text{freeAllocs}(\varphi') \cup \text{freeAllocs}(\varphi'') & \varphi = \varphi' \sqcup \varphi'' \\ \emptyset & \text{otherwise} \end{cases}$$

Finally, the usage of a recursive function definition is handled via the $\{\textbf{rec}\ \epsilon\ \varphi''\}$ effect. This rule ($\times$-Rec) is straightforward, since the burden of ensuring that free regions in φ'' are unbounded is placed on the ($\times$-VarR) and ($\times$-VarL) rules. The latent effect of the recursive function definition φ'' is simply composed with the preceding effect φ. Effect variables in φ'' are substituted for $\{\bot\}$ since there is no need to recheck the validity of $\{\epsilon\}$ effects.

Effect combination is a partial function, since the combination of some effects is not valid. For example, the combination of two effects which free the same region is invalid. In this case, the function $\times$ is undefined. Undefined compositions are rejected during typing.

We conclude this section with an example of a Spegion program and its typing:

```
let x = newrgn [5] in
in
   (λz.newrgn [5];
      newrgn [5]) [1] at x
```

which defines a function that creates two fresh regions, and is typed as:

$$((\text{Unit}, \rho')\ \xrightarrow{\{\textbf{fresh}\ \rho_1\ 5\} \times \{\textbf{fresh}\ \rho_2\ 5\}}\ (\text{Unit}, \rho_2), \rho) \mid \{\textbf{alloc}\ 1\ \rho\}$$

3 Dynamic Semantics

The relation $\langle e \mid \sigma \rangle \to \langle e' \mid \sigma' \rangle$ means that e reduces to e' in a single step. Unlike the calculus of [17], our reduction rules include an explicit store to enable the use of references. This store is dual layered, with an outer store σ mapping region names ρ to pairs of inner stores $\sigma_\rho^{in\cdot}$, and maximum sizes s:

$$\sigma ::= \quad \emptyset \mid \rho \mapsto (\sigma_\rho^{in\cdot}, s) \qquad \text{(outer store)}$$

This maximum size which is specified by the programmer as an annotation when the store is allocated initially via a **newrgn** expression. Inner stores thus map locations to values:

$$\sigma_\rho^{in\cdot} ::= \quad \emptyset \mid l \mapsto v \qquad \text{(inner store)}$$

Using our store typing context Σ which maps locations to types, we define the following judgement for well-typed stores:

$$K \mid \Gamma \mid \Sigma \vdash \sigma$$

This judgement asserts that all values in the store are coherent with the store typing, and that the total size allocated for each region does not exceed the region's specified maximum size. Figure 6 gives the typing rules for stores.

$$\frac{}{K \mid \Gamma \mid \Sigma \vdash \emptyset}\ \textsc{st-outer-empty} \qquad \frac{\text{currentSize}(\sigma(\rho)) \sqsubseteq s \qquad K \mid \Gamma \mid \Sigma \vdash \sigma \qquad K \mid \Gamma \mid \Sigma \vdash \sigma_\rho^{in\cdot}}{K \mid \Gamma \mid \Sigma \vdash \sigma, \rho \mapsto (\sigma_\rho^{in\cdot}, s)}\ \textsc{st-outer-region}$$

Figure 6 Typing rules of σ.

From these rules a store is well-typed if it is empty (ST-OUTER-EMPTY) or (ST-OUTER-REGION) if each of its inner stores are well-typed, and the total size of values allocated in a region does not exceed the region's maximum size s.

The currentSize function is given by Definition 3.1, mapping inner stores to the total size of values allocated in that inner store, the dynamic counterpart to Definition 2.2 in Section 2:

▶ **Definition 3.1** (Current Size).

$$\text{currentSize}(\sigma_\rho^{in\cdot}) = \begin{cases} \text{currentSize}(\emptyset, s) & = 0 \\ \text{currentSize}((\sigma_\rho^{in\cdot\prime}, l_\rho \mapsto v), s) & = \text{sizeOf}(v) \\ & \quad + \text{currentSize}(\sigma_\rho^{in\cdot\prime}, s) \end{cases}$$

The function sizeOf(v) maps values v to sizes, given by Definition 3.2:

▶ **Definition 3.2** (Value Size). Given a value v, sizeOf(v) is defined:

$$\text{sizeOf}(v) = \begin{cases} 1 + |\text{freeLocs}(e)| & v = (\lambda x.e) \\ 1 + |\text{freeLocs}(e)| & v = (\Lambda\{\alpha, \rho, \epsilon\}.\lambda x.e) \\ 1 & \text{otherwise} \end{cases}$$

where freeLocs(e) returns a list of locations in an expression e.

A separate judgement types the inner store:

$$K \mid \Gamma \mid \Sigma \vdash \sigma_\rho^{in\cdot}$$

asserting that all values in the inner store are coherent with the store typing. The typing rules for inner stores are straightforward and are given in Figure 7.

$$\frac{}{K \mid \Gamma \mid \Sigma \vdash \emptyset}\ \textsc{st-inner-empty} \qquad \frac{\tau = \Sigma(l_\rho) \qquad K \mid \Gamma \mid \Sigma \vdash \sigma_\rho^{in\cdot} \qquad K \mid \Gamma \mid \Sigma \vdash v : \tau}{K \mid \Gamma \mid \Sigma \vdash \sigma_\rho^{in\cdot}, l_\rho \mapsto v}\ \textsc{st-inner-loc}$$

Figure 7 Typing rules of $\sigma_\rho^{in\cdot}$.

$$\textsc{e-newrgn}\ \frac{\rho = \text{freshRegion}() \qquad l_\rho = \text{freshLoc}(\rho)}{\langle \textbf{newrgn}\ [s] \mid \sigma \rangle \longrightarrow \langle l_\rho \mid \sigma, \rho \mapsto (l_\rho \mapsto (), s) \rangle}$$

$$\frac{}{\langle \textbf{freergn}\ l_\rho \mid \sigma \rangle \longrightarrow \langle l^1_{\rho_{\text{glob}}} \mid \sigma \setminus \rho \rangle}\ \textsc{e-freergnL}$$

$$\frac{\rho' = \text{freshRegion}() \qquad (\sigma^{in.}_\rho, s_a) = \sigma(\rho) \qquad l_{\rho'} = \text{freshLoc}(\rho')}{\langle \textbf{split}\ [s]\ l_\rho \mid \sigma \rangle \longrightarrow \langle l'_\rho \mid [\rho \mapsto (\sigma^{in.}_\rho, s_a \dot{-} s)]\sigma, \rho' \mapsto (l_{\rho'} \mapsto (), s) \rangle}\ \textsc{e-splitL}$$

$$\frac{\begin{array}{c}(\sigma^{in.}_\rho, s_a) = \sigma(\rho) \qquad (\sigma^{in.}_{\rho'}, s'_a) = \sigma(\rho') \\ v = \sigma^{in.}_\rho(l_\rho) \qquad l'_{\rho'} = \text{freshLoc}(\rho')\end{array}}{\langle \textbf{copy}\ l_\rho\ \textbf{into}\ l_{\rho'} \mid \sigma \rangle \longrightarrow \langle l'_{\rho'} \mid [\rho' \mapsto ((\sigma^{in.}_{\rho'}, l'_{\rho'} \mapsto l_\rho), s'_a)]\sigma \rangle}\ \textsc{e-copyL}$$

$$\frac{\begin{array}{c}(\sigma^{in.}_\rho, s_a) = \sigma(\rho) \qquad s_v = \text{sizeOf}(v) \\ s_v \sqsubseteq s \qquad l'_\rho = \text{freshLoc}(\rho)\end{array}}{\langle v\ [s]\ \textbf{at}\ l_\rho \mid \sigma \rangle \longrightarrow \langle l'_\rho \mid [\rho \mapsto ((\sigma^{in.}_\rho, l'_\rho \mapsto v), s_a)]\sigma \rangle}\ \textsc{e-valL}$$

$$\frac{(\sigma^{in.}_\rho, s_a) = \sigma(\rho) \qquad (\lambda x.e) = \sigma^\rho_{in.}(l_\rho)}{\langle l_\rho\ l'_{\rho'} \mid \sigma \rangle \longrightarrow \langle [x \mapsto l'_{\rho'}]e \mid \sigma \rangle}\ \textsc{e-appL}$$

$$\frac{}{\langle \textbf{let}\ x : \mu = l_\rho\ \textbf{in}\ e_2 \mid \sigma \rangle \longrightarrow \langle [x \mapsto l_\rho]e_2 \mid \sigma \rangle}\ \textsc{e-letL}$$

$$\textsc{e-ifFalse}\ \frac{\sigma^{in.}_\rho = \sigma(\rho) \qquad \text{false} = \sigma^{in.}_\rho(l_\rho)}{\langle \textbf{if}\ l_\rho\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \mid \sigma \rangle \longrightarrow \langle e_3 \mid \sigma \rangle} \qquad \textsc{e-ifTrue}\ \frac{\sigma^{in.}_\rho = \sigma(\rho) \qquad \text{true} = \sigma^{in.}_\rho(l_\rho)}{\langle \textbf{if}\ l_\rho\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \mid \sigma \rangle \longrightarrow \langle e_2 \mid \sigma \rangle}$$

$$\frac{(\sigma^{in.}_\rho, s_a) = \sigma(\rho) \qquad l'_\rho = \text{freshLoc}(\rho)}{\langle \textbf{ref}\ l_\rho \mid \sigma \rangle \longrightarrow \langle l'_\rho \mid [\rho \mapsto ((\sigma^{in.}_\rho, l'_\rho \mapsto l_\rho), s_a)]\sigma \rangle}\ \textsc{e-refL}$$

$$\frac{(\sigma^{in.}_\rho, s_a) = \sigma(\rho) \qquad l'_\rho = \sigma^\rho_{in.}(l_\rho)}{\langle !l_\rho \mid \sigma \rangle \longrightarrow \langle l'_\rho \mid \sigma \rangle}\ \textsc{e-derefL}$$

$$\textsc{e-assignL}\ \frac{(\sigma^{in.}_\rho, s_a) = \sigma(\rho)}{\langle l_\rho := l_{\rho'} \mid \sigma \rangle \longrightarrow \langle l^1_{\rho_{\text{glob}}} \mid [\rho \mapsto ((\sigma^{in.}_\rho, l_\rho \mapsto l_{\rho'}), s_a)]\sigma \rangle} \qquad \textsc{e-seqNext}\ \frac{}{\langle l_\rho; e_2 \mid \sigma \rangle \longrightarrow \langle e_2 \mid \sigma \rangle}$$

$$\textsc{e-tyApp}\ \frac{}{\langle e\ @\ (\tau, \rho') \mid \sigma \rangle \longrightarrow \langle e \mid \sigma \rangle} \qquad \textsc{e-fixL}\ \frac{\sigma^{in.}_\rho = \sigma(\rho) \qquad \Lambda\{\alpha, \rho, \epsilon\}.\lambda x.e_1 = \sigma^{in.}_\rho(l_\rho)}{\langle \textbf{let}\ f = (f, l_\rho)\ \textbf{in}\ e_3 \mid \sigma \rangle \longrightarrow \langle [x \mapsto l_\rho]e_3 \mid \sigma \rangle}$$

Figure 8 Evaluation rules of Spegion.

Figure 8 gives the reduction rules for Spegion. For brevity, we omit congruence rules which are standard. The complete set of rules can be found in the appendix.

The (E-NEWRGN) rule allocates a new region in the store, with the specified size s, and allocates and steps to a new location that points to a unit value in the fresh region. The (E-FREERGNL) rule simply frees the region ρ from the store and steps to a pointer to the unit value in the global region $l^1_{\rho_{\text{glob}}}$.

Splitting a sub-region off from another region is handled by the (E-SPLITL) rule. The rule makes use of the size semiring's truncated subtraction operation, given by Definition 2.1, to calculate the new reduced maximum size of the parent region ρ. As in (E-NEWRGN), the rule then allocates a new region ρ', then allocates and steps to a location in ρ' pointing to a unit value to act as a pointer to the region.

The (E-COPYL) rule copies a location from one region to another. A fresh location $l'_{\rho'}$ is allocated in the region being copied to (ρ') pointing to the location being copied from ρ (l_ρ).

The reduction rule for allocation (E-VALL) compares the programmer annotated size against the actual size of the value being allocated (calculated via 3.2). If the annotated size is greater than or equal to the actual size, the value is allocated into the region at a fresh location l'_ρ in the store. Type safety guarantees that ρ has enough space to allocate a value of the annotated size. If the annotated size is less than the actual size, the program gets stuck. Type safety also ensures that such a situation should never arise.

The reduction rules for application (E-APPL), let binding (E-LETL), if statements (E-IFFALSE), (E-IFTRUE), referencing (E-REFL), dereferencing (E-DEREFL), assignment (E-ASSIGNL), and sequential composition (E-SEQNEXT) are standard.

Type application (E-BIGAPP) is handled by the rule of the same name, which is a no-op, since it is a purely static construct. Recursive functions are handled by the (E-FIXL) rule, which substitutes the location l_ρ, where the recursive function $\Lambda\{\alpha, \rho, \epsilon\}.\lambda x.e_1$ is stored, for the variable f in e_3.

4 Properties of the Type System

Type safety is guaranteed by Theorems 1 and 2. Theorem 1 states the property of *progress*, i.e, a well-typed closed term is either a value, or can be further reduced. Theorem 2 states the property of type *preservation* (or *subject reduction*), which states that if a term is well-typed and can take a step of evaluation, then the resulting term is also well-typed. The proofs are based on the syntactic type soundness proofs of Helsen and Thiemann [17], and can be found in the extended version of this paper.

▶ **Theorem 1** (Progress). *If*

$$K \mid \Gamma \mid \Sigma \vdash e : (\tau, \rho) \mid \varphi$$

and

$$K \mid \Gamma \mid \Sigma \vdash \sigma$$

then either

i *e is a value or*
ii *e has the form (x) (a variable), with $x \in fv(e)$ or*
iii *there exists an e' such that $\langle e \mid \sigma \rangle \longrightarrow \langle e' \mid \sigma' \rangle$*

▶ **Theorem 2** (Preservation). *If*

$$K \mid \Gamma \mid \Sigma \vdash e : \mu \mid \varphi$$
$$K \mid \Gamma \mid \Sigma \vdash \sigma$$
$$\langle e \mid \sigma \rangle \longrightarrow \langle e' \mid \sigma' \rangle$$

for some Σ' such that

$$\Sigma' \supseteq \Sigma$$

we have that

$$K \mid \Gamma \mid \Sigma' \vdash e' : \mu \mid \varphi'$$
$$K \mid \Gamma \mid \Sigma' \vdash \sigma'$$

where

$$K \vdash \varphi' \sqsubseteq \varphi : \mathit{Effect}$$

From our assumption of store typing correctness, we assert that e is a sub-derivation of some larger derivation e″ for which correctness with regard to region size holds, i.e., given

$$K \mid \Gamma \mid \Sigma \vdash \sigma$$

we have that:

$$\forall \rho \in \sigma . \forall l_\rho \in \sigma^{in \cdot}_\rho . l_\rho \in dom(\Sigma) \wedge sizeOf(\Sigma(l_\rho)) \sqsupseteq sizeOf(\sigma^{in \cdot}_\rho(l_\rho))$$

Finally, we assume the existence of a global region parametrising the calculus which cannot be freed, as well as a location inside this global region of type Unit:

$$\forall \Sigma . l^1_{\rho_{glob}} : \mathit{Unit} \in \Sigma$$

Preservation states that if an expression e is well-typed and has effect φ, and a reduction step can be made, then the resulting expression e' is also well-typed with an updated store typing Σ' and an effect φ' such that $\varphi' \sqsubseteq \varphi$.

$$\frac{K \vdash \varphi_1 \equiv \varphi_2 : \text{Effect}}{K \vdash \varphi_1 \sqsubseteq \varphi_2 : \text{Effect}}\ \text{SB-}\equiv$$

$$\frac{K \vdash \varphi : \text{Effect}}{K \vdash \{\bot\} \sqsubseteq \varphi}\ \text{SB-}\bot$$

$$\frac{K \vdash \varphi_1 \sqsubseteq \varphi_3 : \text{Effect} \qquad K \vdash \varphi_2 \sqsubseteq \varphi_3 : \text{Effect}}{K \vdash \varphi_1 \times \varphi_2 \sqsubseteq \varphi_3 : \text{Effect}}\ \text{SB-}\times_{above}$$

$$\frac{K \vdash \varphi_1 \sqsubseteq \varphi_2 : \text{Effect} \qquad K \vdash \varphi_3 : \text{Effect}}{K \vdash \varphi_1 \sqsubseteq \varphi_2 \times \varphi_3 : \text{Effect}}\ \text{SB-}\times^1_{below}$$

$$\frac{K \vdash \varphi_2 : \text{Effect} \qquad K \vdash \varphi_1 \sqsubseteq \varphi_3 : \text{Effect}}{K \vdash \varphi_1 \sqsubseteq \varphi_2 \times \varphi_3 : \text{Effect}}\ \text{SB-}\times^2_{below}$$

$$\frac{K \vdash \varphi_1 \sqsubseteq \varphi_3 : \text{Effect} \qquad K \vdash \varphi_2 \sqsubseteq \varphi_3 : \text{Effect}}{K \vdash \varphi_1 \sqcup \varphi_2 \sqsubseteq \varphi_3 : \text{Effect}}\ \text{SB-}\sqcup_{above}$$

$$\frac{K \vdash \varphi_1 \sqsubseteq \varphi_2 : \text{Effect} \qquad K \vdash \varphi_3 : \text{Effect}}{K \vdash \varphi_1 \sqsubseteq \varphi_2 \sqcup \varphi_3 : \text{Effect}}\ \text{SB-}\sqcup^1_{below}$$

$$\frac{K \vdash \varphi_2 : \text{Effect} \qquad K \vdash \varphi_1 \sqsubseteq \varphi_3 : \text{Effect}}{K \vdash \varphi_1 \sqsubseteq \varphi_2 \sqcup \varphi_3 : \text{Effect}}\ \text{SB-}\sqcup^2_{below}$$

Figure 9 Subsumption rules for Spegion ($\sqsubseteq$).

The relation $\sqsubseteq$ defines effect *subsumption*. Effect subsumption is used to relate the effects of intermediate terms as a program is evaluated, since evaluation does not preserve syntactic effects. For example, an allocation expression has an effect which contains an allocation effect $\{\mathbf{alloc}\ s\ \rho\}$. However, the reduction step for allocation expressions produces a location, and locations are typed with an empty effect $\{\bot\}$. To the programmer, these intermediate effects are not of interest – the effect of importance is the overall effect of the program. In preservation, however, we ensure that the intermediate effects are related to this overall effect via the subsumption judgement:

$$K \vdash \varphi_1 \sqsubseteq \varphi_2 : \text{Effect}$$

These rules are given in Figure 9. These rules are standard, however, note that we include

$$\frac{K \vdash \varphi : \text{Effect}}{K \vdash \varphi \equiv \varphi : \text{Effect}}\ \equiv\text{-REFL} \qquad \frac{K \vdash \varphi_1 : \text{Effect} \quad K \vdash \varphi_2 : \text{Effect} \quad K \vdash \varphi_1 \equiv \varphi_2 : \text{Effect}}{K \vdash \varphi_2 \equiv \varphi_1 : \text{Effect}}\ \equiv\text{-SYM}$$

$$\frac{K \vdash \varphi_1 \equiv \varphi_2 : \text{Effect} \quad K \vdash \varphi_2 \equiv \varphi_3 : \text{Effect}}{K \vdash \varphi_1 \equiv \varphi_3 : \text{Effect}}\ \equiv\text{-TR}$$

$$\frac{K \vdash \varphi_1 \equiv \varphi_1' : \text{Effect} \quad K \vdash \varphi_2 \equiv \varphi_2' : \text{Effect}}{K \vdash \varphi_1 \times \varphi_2 \equiv \varphi_1' \times \varphi_2' : \text{Effect}}\ \equiv\text{-}\times\text{-CONG}$$

$$\frac{K \vdash \varphi_1 : \text{Effect} \quad K \vdash \varphi_2 : \text{Effect} \quad K \vdash \varphi_3 : \text{Effect}}{K \vdash \varphi_1 \times (\varphi_2 \times \varphi_3) \equiv (\varphi_1 \times \varphi_2) \times \varphi_3 : \text{Effect}}\ \equiv\text{-}\times\text{-ASSOC}$$

$$\frac{K \vdash \varphi_1 : \text{Effect} \quad K \vdash \varphi_2 : \text{Effect} \quad K \vdash \varphi_3 : \text{Effect}}{K \vdash \varphi_1 \sqcup (\varphi_2 \sqcup \varphi_3) \equiv (\varphi_1 \sqcup \varphi_2) \sqcup \varphi_3 : \text{Effect}}\ \equiv\text{-}\sqcup\text{-ASSOC}$$

$$\frac{K \vdash \varphi_1 \equiv \varphi_1' : \text{Effect} \quad K \vdash \varphi_2 \equiv \varphi_2' : \text{Effect}}{K \vdash \varphi_1 \sqcup \varphi_2 \equiv \varphi_1' \sqcup \varphi_2' : \text{Effect}}\ \equiv\text{-}\sqcup\text{-CONG}$$

Figure 10 Effect equality rules for Spegion ($\equiv$).

kinding of effects and we include rules for the join operator $\sqcup$. The (SB-$\equiv$) rule requires a judgement of effect equivalence:

$$K \vdash \varphi_1 \equiv \varphi_2 : \text{Effect}$$

which is given by the rules in Figure 10. Again, these rules are completely standard.

5 Applications

5.1 System Code Examples

Various systems programming idioms are expressible in the language. Below, we present examples first in C and then in Spegion. In each case, the translation is shallow: region-size annotations required by our language can be inferred from the C program. In one example, our language exceeds what can be expressed in C. In all cases, the type system statically recognises memory safety errors.

Use After Free. In the C code below, a struct with two members is allocated, written to, freed, and then erroneously read from. This is *use after free*, a failure of memory safety.

```
struct mine {int a; int b;};

int free_help(int* p) { free(p); }

int main () {
  struct mine *mp = malloc(sizeof(struct mine));
  mp->a = 0;
  mp->b = 1;
  int *bp = &(mp->b);
  free_help(mp);
  return *bp;    // uaf
}
```

The translation of this code to our language is direct: the size of the new region is taken from the call to `malloc` in the C code.

let $r =$ **newrgn** $[2]$ **in**
let $(x, y) = (\mathbf{ref}\ 0, \mathbf{ref}\ 0)\ [2]$ **at** r **in**
let $x := 0\ [1]$ **at** ρ_{glob} **in**
let $y := 1\ [1]$ **at** ρ_{glob} **in**
let $bp = y$ **in**
freergn r;
let $b = !bp$

This program does not pass the static semantics because on the last line, the type effects record that region r was previously freed, and in typing the final line, the T-REF rule requires the region of bp to be live.

Recursion. Below, a function allocates, calls itself recursively, and then frees its prior allocation. There is no memory safety error: despite being nested by the recursion, all of the allocations are matched by frees.

```
int alloc_rec_free (int n) {
  if (n) {
    int *p = malloc(sizeof(int));
    int  v = alloc_rec_free (n - 1);
    free(p);
    return v;
  } else
    return 0;
}

int main () {
  return alloc_rec_free(10);
}
```

The translation uses the **fix** constructor. Again the translation is shallow: none of the additional annotations in our language require any analysis to concoct. Indeed the function's type scheme is empty because it conveys no change to the live regions to its context.

let arf $=$ **fix**(arf, $(\Lambda\emptyset.\lambda n.$
  if $(n == 0\ [1]$ **at** $\rho_{\text{glob}})$ **then**
    let $r =$ **newrgn** $[1]$ **in**

$$
\begin{array}{l}
\quad\quad \textbf{let } p = \textbf{ref } 0\ [1]\ \textbf{at } r\ \textbf{in} \\
\quad\quad \textbf{let } v = (\mathrm{arf}\ @\ \emptyset)\ (n-1\ [1]\ \textbf{at } \rho_{\mathrm{glob}})\ \textbf{in} \\
\quad\quad \textbf{freergn } p; \\
\quad\quad v \\
\quad \textbf{else} \\
\quad\quad 0\ [1]\ \textbf{at } \rho_{\mathrm{glob}} \\
)\ [1]\ \textbf{at } \rho_{\mathrm{glob}})\ \textbf{in} \\
(\mathrm{arf}\ @\ \emptyset)\ (10\ [1]\ \textbf{at } \rho_{\mathrm{glob}})
\end{array}
$$

The static semantics accepts this program and it is memory safe.

Modelling Loops. Loops can be translated to recursive functions. The C code below is analogous to the code above but it uses a loop. The loop body allocates, it records the value of the index and then frees the prior allocation.

```
int main () {
  int v;
  for (int n = 10; n > 0; n--) {
    int *p = malloc(sizeof(int));
    v = n;
    free(p);
  }
  return v;
}
```

The loop is translated into a recursive function whose argument serves as the loop index.

$$
\begin{array}{l}
\textbf{let } \mathrm{loop} = \textbf{fix}(\mathrm{loop}, (\Lambda\emptyset.\lambda n. \\
\quad \textbf{if } (n > 0\ [1]\ \textbf{at } \rho_{\mathrm{glob}})\ \textbf{then} \\
\quad\quad \textbf{let } r = \textbf{newrgn } [1]\ \textbf{in} \\
\quad\quad \textbf{let } p = \textbf{ref } (0\ [1]\ \textbf{at } \rho_{\mathrm{glob}})\ [1]\ \textbf{at } r\ \textbf{in} \\
\quad\quad \textbf{let } v := n\ \textbf{in} \\
\quad\quad \textbf{freergn } r; \\
\quad\quad (\mathrm{loop}\ @\ \emptyset)\ (n-1\ [1]\ \textbf{at } \rho_{\mathrm{glob}}) \\
\quad \textbf{else } () \\
)\ [1]\ \textbf{at } \rho_{\mathrm{glob}})\ \textbf{in} \\
(\mathrm{loop}\ @\ \emptyset)\ (10\ [1]\ \textbf{at } \rho_{\mathrm{glob}}); \\
v
\end{array}
$$

Pointer Arithmetic and Finite Buffers. The C code below produces and consumes packets, serialising them in a buffer. Spegion tracks pointer arithmetic and will catch patterns of use that exhaust the size of the buffer.

```
struct packet {int len; int payload[];};

struct packet * produce (struct packet *p, int data) {
  p->len = sizeof(int);  p->payload[0] = data;
  return p->payload + p->len;
}
```

```
struct packet * consume (struct packet *p, int* sum) {
  *sum += p->payload[0];
  return p->payload + p->len;
}

int main () {
  int sum = 0;
  void *buf = malloc(1000);
  struct packet *pp = buf, *cp = buf;
  pp = produce(pp, 1);  pp = produce(pp, 2);
  cp = consume(cp, &sum);  cp = consume(cp, &sum);
  return sum;
}
```

`main` function translates into the following Spegion. Statically, the loads and stores of calls to produce and consume are checked against the bounds of the region of buf. The return pointers type check, even if they escape the region of buf, matching idiomatic use of pointer arithmetic in C.

let r1 = **newrgn in**
let sum = **ref** (0 **at** r1) **in**
let buf = **newrgn** [1000] **in**
let init = **ref** (initPacket [1] **at** buf) **in**
let (pp, cp) =!init **in**
let pp = produce(pp, 1) **in**
let pp = produce(pp, 2) **in**
let cp = consume(cp, sum) **in**
let cp = consume(cp, sum) **in**
!sum

Non-Lexically-Scoped Lifetimes. The C code of Figure 1 allocates a linked list of three tasks in list order, takes a reference to the last task, and then frees the first two tasks. Spegion can represent and statically type programs like this. The translation is direct:

let re = **newrgn** [1] **in**
let end = 0 [1] **at** re **in**

let r_0 = **newrgn** [2] **in**
let t_0 = (0 [1] **at** ρ_{glob}, end) [2] **at** r_0 **in**
let $(n_0, \text{next}_0) = t_0$ **in**

let r_1 = **newrgn** [2] **in**
let t_1 = (0 [1] **at** ρ_{glob}, end) [2] **at** r_1 **in**
let $(n_1, \text{next}_1) = t_1$ **in**
let $\text{next}_0 = t_1$ **in**

let r_2 = **newrgn** [2] **in**
let t_2 = (0 [1] **at** ρ_{glob}, end) [2] **at** r_2 **in**
let $(n_2, \text{next}_2) = t_2$ **in**
let $\text{next}_1 = t_2$ **in**

let $n_2 = 1$ [1] **at** ρ_{glob} **in**

freergn t_0;

freergn t_1;

let $(n, _) = !t_2$ **in**

n

Splitting. Our language represents idioms that C cannot represent, in particular *splitting*. In this idiom, the programmer creates a region. Subsequently they split the region into a part that is freed, and a part that is kept and used.

C cannot represent this idiom, but the C code below demonstrates the idea. The `alloc2` function mirrors the initial allocation, but to match the limitations of C it is implemented with an indirection and two allocations matching the subsequent split. The first part of the allocation made by `alloc2` is freed, and the second part is used.

```
struct indirect {int* p1; int* p2;};

struct indirect * alloc2 () {
  struct indirect *p = malloc(sizeof(struct indirect));
  p->p1 = malloc(sizeof(int));
  p->p2 = malloc(sizeof(int));
  return p;
}

int main () {
  struct indirect *pr = alloc2();
  *(pr->p2) = 1;
  free(pr->p1);
  return *(pr->p2);
}
```

Our language can represent this programming idiom. The region r is created in one call to **newrgn**, and then split into two parts. These parts can be used as a pair, but they can also be independently freed.

let $r =$ **newrgn** [2] **in**

let $r_1 =$ **split** [1] **in**

let $v_1 = (0$ [1] **at** $\rho_{\text{glob}})$ [1] **at** r_1 **in**

let $v_2 = (0$ [1] **at** $\rho_{\text{glob}})$ [1] **at** r **in**

let $(p_1, p_2) = (v_1, v_2)$ [2] **at** ρ_{glob} **in**

let $p_2 := 1$ [1] **at** ρ_{glob}**in**

freergn p_1;

$!p_2$

5.2 Refinement Types

Refinement types give the programmer the ability to use *predicates* in the type as a means of restricting the values described by the type [11, 22]. For example, the type:

$n : \text{Int} \to \text{Int}\ [n' \mid n' \leq n]$

describes a function from integers to integers which returns a value that must be less than or equal to the input. This is a simple example of a refinement type, but they can be used to encode many forms of specification.

A natural extension of our work would be to incorporate refinement types into our type system, allowing the programmer to reason about region sizes in the types of their programs. In this section we sketch out a simple refinement system based on the approach of Jhala and Vazou [18] that could be used in tandem with SPEGION's region sizes and sized allocations. We define a new sort of *predicates* which can be used in the type system to restrict the values described by a type. This section is mostly a sketch of the ideas, and we leave the full development of the system to future work. We begin with the syntax of predicates:

$$p := \quad x \mid n \mid \text{true} \mid \text{false} \mid p + p \mid p \cdot p \mid \neg p \mid \textbf{if } p \textbf{ then } p \textbf{ else } p \mid f(\overline{p}) \qquad \text{(predicates)}$$

Predicates are drawn from quantifier-free fragment of linear arithmetic and uninterpreted functions [3]. That is, a predicate p may be a variable x, a natural number constant n, boolean constants true and false, the sum or product of two predicates, the negation of a predicate, a conditional, or the application of an uninterpreted function f to a list of predicates. For this section, we extend SPEGION with type annotations and a new list datatype:

$$\text{List } (\tau, \rho) = \text{Nil} : \text{List } (\tau, \rho) \mid \text{Cons} : (\tau, \rho) \xrightarrow{\{\bot\}} \text{List } (\tau, \rho) \xrightarrow{\{\bot\}} \text{List } (\tau, \rho)$$

That is, a list is a value which is either empty or a pair of a value of type (τ, ρ) and a reference to another list. For simplicity the payload of the list must be stored in the same region as the list itself. One could envision a more complex type where the payload and indeed the tail of the list could be stored in separate regions. With the above type we can construct a function which takes an integer n and creates a list in a region of arbitrary size that has length n:

```
let r = newrgn
  in
  let buildList = fix(buildList, Λ{α, ρ, ϵ}λn.
    if n == 0
      then
        Nil at r
      else
        Cons (n, buildList @(Int, regionOf(r)) n − 1)) at r
in ...
```

With refinement types we could constrain this function to only return lists where the size of the list is less than or equal to the size of the region:

```
let r = newrgn [5]
in
  let buildList : ((n : Int, regionOf(r)) -φ-> (List (Int, regionOf(r))) [1 + (2 · n) < 5]) =
    fix(buildList, Λ{α, ρ, ϵ}λn.
      if n == 0
        then
          Nil [1] at r
        else
          Cons (n at [1], buildList @(Int, regionOf(r)) n − 1)) [1] at r
  in ...
```

where $\varphi = \{\textbf{alloc } 1 \text{ regionOf}(r)\} \sqcup (\{\epsilon\} \times \{\textbf{alloc } 1 \text{ regionOf}(r)\})$. Note we remove superfluous $\{\bot\}$ effects from the type annotation for brevity. Thus if we try to apply buildList to the integer 10, this application will fail as the size of the list produced is greater than the

size of the region associated with r. This is just a taste of what the interaction between refinement types and region sizes can offer. We believe that adding refinements to Spegion is a natural extension of the language, with few changes required to the core calculus.

6 Related Work

Key areas of related work for this research relate to the Rust programming language, and effect and region types in programming language design.

6.1 Rust

Rust is a popular systems programming language that aims to statically enforce memory safety, and it has also been the subject of significant academic research. Notable formal work includes Rust Belt [19] and Oxide [31]. The key concepts involved in Rust's approach to enforcing memory safety are *ownership* and *lifetimes*, with the latter corresponding to regions. As a key component of enforcing safety guarantees, Rust often limits or prohibits the introduction of *aliases* of pointers to mutable data, giving it a similar flavour to the substructural type system approaches detailed later in this section. This is, in part, because Rust is also concerned with preventing data races in concurrent code.

The type system might be extended to the concurrent context by reconciling the sequence of effects with the orderings imposed by a declarative concurrency semantics, like those of sequential consistency, ARMv8, or C++, but this is left to future work. Even in sequential code, however, there are difficulties involved in aliasing pointers and (for example) building cyclic data structures.

The following program is rejected by the Rust compiler:

```
fn main() {
    let mut a: i32 = 10;
    let b: &i32 = &a;

    {
        let c: &mut i32 = &mut a;
        *c = 20;
    }
    // use a,b ...
}
```

The compiler will point out `a` being borrowed multiple times (immutably by `b` and mutably by `c`). This program can be represented in our calculus with no issue, as our approach allows arbitray aliasing of mutable references. In general, Rust's expressive type system supports many patterns of memory usage, and we believe there is potential in combining ideas from Rust and from our effect-based approach.

6.2 Effect Types and Region Calculi

There is extensive prior work on using effects and regions in programming language design. Starting in the 1980s, Lucassen and Gifford, among others, explored the development of polymorphic effect type systems [14, 20]. They made use of a *kind* system consisting of types, which describe what sort of value an expression may evaluate to; effects, which describe what side effects may be performed when evaluating an expression; and regions, which describe groups of values in memory that are related. Building upon the idea of associating types with regions, Tofte and Talpin introduced the region calculus and region-based memory management [28, 29], which provided a structured way to control memory in higher order programs by associating types with parameters to statically track their lifetimes.

Region-based memory management formed the basis of ML-Kit, a compiler for Standard ML that used *region inference* to infer region annotations of source programs and statically determine memory usage [4, 25]. Rather than requiring programmers to manually annotate their programs with region/lifetime parameters, the compiler determines them automatically. Subsequent work investigated improving region inference, as it was prone to potentially producing programs that leak memory or unnecessarily extend the lifetime of objects [1, 26]. ML-Kit was augmented with a runtime garbage collector in order to collect memory left around inside regions which had become garbage but had not yet been de-allocated because it lived inside a still-alive region [16].

Existing papers have proven type safety theorems for variations of the region calculus. Helsen and Thiemann give an elegant syntactic proof of type safety for a region calculus with polymorphic types and recursive let-bindings, which was a primary source for the formalism in this paper [17]. They represent region de-allocation by introducing a special "dead" region identifier, and when a region goes out of scope they substitute this special identifier for the now-dead reagion. This also means they do not need to have a phase distinction between handles and region identifiers. However, they do not have a store in their operational semantics, which leads to a straightforward proof but unfortunately does not allow for mutable data. Subsequent work extended this approach with an explicit store, but had to remove polymorphic types [7]. Prior work also investigated proof of soundness for a fragment of the region calculus via a translation into a target language [2].

One of the major issues with using region-based memory management as proposed in the original region calculus is that the lexical scoping of regions means they are required to follow a stack or last-in-first-out (LIFO) ordering. This was observed by Aiken et al., demonstrating that extending the traditional region calculus by removing the LIFO restriction dramatically reduces memory usage (sometimes asymptotically) in functional programs [1]. Like our approach, they de-couple allocation and de-allocation, though their approach is for functional programs without mutable memory. They are also focused on region annotation inference in order to statically determine memory lifetimes of ML programs, and so they do not consider bounding or splitting regions and cannot track fine-grained effect details like an allocation consuming a specific amount of memory in a region.

Relatedly, Boudol presented a monomorphic variant of the region calculus that permitted early de-allocation [5]. A type system tracks both negative de-allocation effects and positive effects such as reads and writes, ensuring that the positive effects on a region cannot occur after the negative effect.

Further work explored adding explicit region allocation and de-allocation to a C-like language, where the safety of region de-allocation relied on keeping track of a dynamic *reference count* for objects in the heap [12, 13].

A prominent line of research that addressed the problem of LIFO lifetimes of regions is the use of *substructural types* in region type systems [10]. The Cyclone programming language, and related work on the calcluls of capabilities, allowed for type-safe manual memory management using a combination of type system features including substuctural types, allowing for both LIFO regions as well as first-class dynamic regions and unique pointers [8, 10]. This style of type system requires that programmers thread resources linearly through programs and avoid creating aliases. In contrast, a primary goal of our system is to avoid imposing this requirement of linearity in programs, and allowing for the aliasing of pointers that occurs in idiomatic imperative code.

Region types have also been used in programming language design for reasons beyond memory management. For example, in Gibbon, region types are used in combination with an effect type system to type programs that create and manipulate serialised data, where each region contains a serialised data structure [30].

7 Conclusion

We have proposed a type system for region-based memory management that allows implicit regions with non-lexical scoping and explicit size constraints. We have proven type safety for this system, and shown several directions in which this system can be extended. Based on this type system, we are currently developing an implementation of Spegion in the form of both an interpreter for Spegion as well as a static analysis tool targeting the C language.

References

1 Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: improving region-based analysis of higher-order languages. *SIGPLAN Not.*, 30(6):174–185, June 1995. doi:10.1145/223428.207137.

2 Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Region analysis and the polymorphic lambda calculus. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, LICS '99, page 88, USA, 1999. IEEE Computer Society.

3 Clark W. Barrett, Aaron Stump, and Cesare Tinelli. The smt-lib standard version 2.0, 2010. URL: https://smt-lib.org/papers/smt-lib-reference-v2.0-r10.12.21.pdf.

4 Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 171–183, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/237721.237771.

5 Gérard Boudol. Typing safe deallocation. In Sophia Drossopoulou, editor, *Programming Languages and Systems*, pages 116–130, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-78739-6_10.

6 Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time java. *SIGPLAN Not.*, 38(5):324–337, May 2003. doi:10.1145/780822.781168.

7 Christiano Calcagno, Simon Helsen, and Peter Thiemann. Syntactic type soundness results for the region calculus. *Inf. Comput.*, 173(2):199–221, March 2002. doi:10.1006/inco.2001.3112.

8 Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 262–275, New York, NY, USA, 1999. Association for Computing Machinery. doi:10.1145/292540.292564.

9 Morgan Deters and Ron K. Cytron. Automated discovery of scoped memory regions for real-time java. *SIGPLAN Not.*, 38(2 supplement):25–35, June 2002. doi:10.1145/773039.512433.

10 Matthew Fluet, Greg Morrisett, and Amal Ahmed. Linear regions are all you need. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP'06, pages 7–21, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11693024_2.

11 Tim Freeman and Frank Pfenning. Refinement types for ml. *SIGPLAN Not.*, 26(6):268–277, May 1991. doi:10.1145/113446.113468.

12 David Gay and Alex Aiken. Memory management with explicit regions. *SIGPLAN Not.*, 33(5):313–323, May 1998. doi:10.1145/277652.277748.

13 David Gay and Alex Aiken. Language support for regions. *SIGPLAN Not.*, 36(5):70–80, May 2001. doi:10.1145/381694.378815.

14 David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 28–38, New York, NY, USA, 1986. Association for Computing Machinery. doi:10.1145/319838.319848.

15 Colin S. Gordon. Polymorphic iterable sequential effect systems. *ACM Trans. Program. Lang. Syst.*, 43(1), April 2021. doi:10.1145/3450272.

16 Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. *SIGPLAN Not.*, 37(5):141–152, May 2002. doi:10.1145/543552.512547.

17 Simon Helsen and Peter Thiemann. Syntactic type soundness for the region calculus. *Electronic Notes in Theoretical Computer Science*, 41(3):1–19, 2001. HOOTS 2000, 4th International Workshop on Higher Order Operational Techniques in Semantics (Satellite to PLI 2000). doi:10.1016/S1571-0661(04)80870-3.

18 Ranjit Jhala and Niki Vazou. Refinement types: A tutorial. *Found. Trends Program. Lang.*, 6(3–4):159–317, October 2021. doi:10.1561/2500000032.

19 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158154.

20 J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/73560.73564.

21 Henning Makholm. A region-based memory manager for prolog. *SIGPLAN Not.*, 36(1):25–34, October 2000. doi:10.1145/362426.362434.

22 Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. *SIGPLAN Not.*, 38(9):213–225, August 2003. doi:10.1145/944746.944725.

23 Alan Mycroft, Dominic Orchard, and Tomas Petricek. Effect systems revisited–control-flow algebra and semantics. In *Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays on Semantics, Logics, and Calculi - Volume 9560*, pages 1–32, Berlin, Heidelberg, 2015. Springer-Verlag. doi:10.1007/978-3-319-27810-0_1.

24 Dominic Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. *SIGPLAN Not.*, 51(1):568–581, January 2016. doi:10.1145/2914770.2837634.

25 Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):724–767, July 1998. doi:10.1145/291891.291894.

26 Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher Order Symbol. Comput.*, 17(3):245–265, September 2004. doi:10.1023/B:LISP.0000029446.78563.a4.

27 Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, and Peter Sestoft. Programming with regions in the ml kit (for version 4), October 2001.

28 Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 188–201, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/174675.177855.

29 Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997. doi:10.1006/inco.1996.2613.

30 Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. Local: a language for programs operating on serialized data. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 48–62, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3314221.3314631.

31 Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. Oxide: The essence of rust, 2021. arXiv:1903.00982.

A Lightweight Method for Generating Multi-Tier JIT Compilation Virtual Machine in a Meta-Tracing Compiler Framework

Yusuke Izawa
Tokyo Metropolitan University, Japan

Hidehiko Masuhara
Institute of Science Tokyo, Japan

Carl Friedrich Bolz-Tereick
Heinrich-Heine-Universität Düsseldorf, Germany

Abstract

Meta-compiler frameworks, such as RPython and Graal/Truffle, generate high-performance virtual machines (VMs) from interpreter definitions. Although they generate VMs with high-quality just-in-time (JIT) compilers, they still lack an important feature that dedicated VMs (i.e., VMs that are developed for specific languages) have, namely *multi-tier compilation*. Multi-tier compilation uses light-weight compilers at early stages and highly optimizing compilers at later stages in order to balance between compilation overheads and code quality.

We propose a novel approach to enabling multi-tier compilation in the VMs generated by a meta-compiler framework. Instead of extending the JIT compiler backend of the framework, our approach drives an existing (heavyweight) compiler backend in the framework to quickly generate unoptimized native code by merely embedding directives and compile-time operations into interpreter definitions.

As a validation of the approach, we developed 2SOM, a Simple Object Machine with a two-tier JIT compiler based on RPython. 2SOM first applies the tier-1 threaded code generator that is generated by our proposed technique, then, to the loops that exceed a threshold, applies the tier-2 tracing JIT compiler that is generated by the original RPython framework. Our performance evaluation that runs a program with a realistic workload showed that 2SOM improved, when compared against an RPython-based VM, warm-up performance by 15%, with merely a 5% reduction in peak performance.

2012 ACM Subject Classification Software and its engineering → Just-in-time compilers; Software and its engineering → Interpreters

Keywords and phrases virtual machine, JIT compiler, multi-tier JIT compiler, meta-tracing JIT compiler, RPython

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.16

Related Version *Full Version*: `http://arxiv.org/abs/2504.17460` [24]

Supplementary Material *Software (Source Code)*: `https://doi.org/10.5281/zenodo.15286979`
Software (Source Code): `https://doi.org/10.5281/zenodo.15286954`
Software (Source Code): `https://doi.org/10.5281/zenodo.15287001`
Software (ECOOP 2025 Artifact Evaluation approved artifact):
`https://doi.org/10.4230/DARTS.11.2.16`

Funding This research was supported by KAKENHI grant nos. 25K21180, 24K23866, 23H03368, and 23K28058, and JST ACT-X grant no. JPMJAX2003.

Acknowledgements We would like to thank Stefan Marr for his valuable advice on the measurement methodology and the evaluation results of 2SOM. We are also grateful to the anonymous reviewers for their insightful comments, which helped improve this manuscript.

© Yusuke Izawa, Hidehiko Masuhara, and Carl Friedrich Bolz-Tereick;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

A *meta-compiler framework* [7, 38] is a system that generates high-performance virtual machines (VMs) from interpreter definitions. Traditionally, VMs were manually implemented by VM developers from scratch, including interpreters, just-in-time (JIT) compilers, and garbage collectors. However, by using a meta-compiler framework, it is possible to generate a VM of comparable quality to traditional VMs simply by defining the interpreter. Currently, there are two frameworks in this domain: RPython [6] and Graal/Truffle [39]. These frameworks have demonstrated their practical effectiveness by generating high-performance VMs such as PyPy [36], Pycket [33], TruffleSqueak [31], and TruffleRuby [38].

Given the importance of multi-tier JIT compilation VMs, efforts have been made with Graal/Truffle to enable a meta-compiler framework to perform multi-tier JIT compilation [35]. The multi-tier JIT compilation VM has multiple execution tiers using different compilation methods or optimization levels to balance code quality and compilation time [1]. In Graal/Truffle's methodology, a single JIT backend is used, and first-tier and second-tier compilations are achieved by gradually enabling or disabling optimizations. However, it has not yet achieved the incorporation of tiers with fundamentally different compilation methods, as has been done in traditional VMs.

Furthermore, as with traditional VMs, enabling a meta-compiler framework to generate a multi-tier JIT compilation VM requires a significant development effort. In traditional VMs, developing a multi-tier JIT compilation VM is more complex than simply implementing multiple compilers. At the least, a VM developer develops multiple compilers with different compilation behaviors – a compiler for rapid compilation speed but generating code with moderate quality, and a compiler for generating quality code – plus a profiler to gather runtime information for switching between different compilers, and a mechanism to transition between code generated by different compilers. Therefore, a framework developer needs to implement these components at the level of a meta-compiler framework.

To address the implementation cost, we propose a novel methodology that enables a meta-compiler framework to generate a multi-tier JIT compilation VM without requiring the development of new JIT compilation backends. Our methodology treats interpreter definitions not only as semantic specifications of the source language but also as specifications of compilation strategies of the meta-compiler. Under our methodology, a VM developer provides interpreter definitions for each compilation tier. Then, RPython can automatically generate the necessary components for multi-tier JIT compilation.

Specifically, the proposed methodology:

(Multiple compilers:) generates a lightweight compiler for rapid code generation during the warm-up phase of a source program, and a heavyweight compiler for optimizing the hot spots of a source program,

(Profiling mechanism:) generates profiling code fragments that dynamically identify hot spots and trigger tier transitions, and

(Transition mechanism:) switches between code generated by different compilers.

To validate our methodology, we implemented a two-level Simple Object Machine, which we call 2SOM for short, on top of the RPython framework. 2SOM is an extended version of PySOM [27] that is an implementation of Simple Object Machine [19] by RPython. 2SOM has two JIT compilers:

(Tier-1:) A threaded code generator, which quickly generates subroutine-threaded code for frequently invoked methods. The threaded code generator is generated by using the RPython's mechanism based on the seminal ideas of Izawa et al. [25] along with several techniques proposed in this paper to make it practical.

(Tier-2:) A tracing JIT compiler, which is generated by RPython [6, 16, 17] and compiles traces with aggressive optimizations to frequently executed loops in the threaded code.

Our approach effectively enables multi-tier JIT compilation for meta-compiler frameworks. Although this paper demonstrates an application to SOM, the proposed approach should also be applicable to other RPython-based language implementations.

To evaluate the effectiveness of the proposed approach, we analyzed the performance of 2SOM with respect to (1) improvement of the warm-up performance for a realistic workload, (2) degradation of peak performance, and (3) quality of the threaded code generator. Our evaluation demonstrates that (1) two-level compilation runs about 15% faster than tracing JIT-only compilation, (2) the degradation is up to about 5% compared to tracing JIT-only compilation, and (3) our contribution improves the code quality of about 10% compared to interpreter execution.

This paper makes the following contributions:

- A novel approach for creating a multi-tier JIT compilation VM using the RPython framework by extending interpreter definitions.
- The optimization of meta-compiler–based threaded code generation [25], addressing practical issues and proposing solutions for integration with meta-tracing JIT compiler frameworks.
- The implementation of 2SOM, the first VM with two JIT compilers – a threaded code generator and a tracing compiler – generated entirely from interpreter definitions.
- A methodology for synthesizing large-scale application programs to evaluate the warm-up performance of the JIT-compiler-based VMs.
- The improvement of warm-up execution speed in 2SOM for synthesized application programs.

The remainder of this paper is organized as follows. Section 2 introduces the techniques underlying our proposal, including JIT and multi-tier JIT compilation, meta-compilation, meta-tracing compilation, and meta-compiler–based threaded code generation. Section 3 discusses the challenges of realizing a multi-tier JIT compilation VM within a meta-compiler framework. Section 4 presents the architecture and technical details of 2SOM. Section 5 identifies practical problems in meta-compiler-based threaded code generation and proposes solutions. Section 6 evaluates 2SOM's performance and discusses the results. Section 7 sets the results into the context of related work, and Section 8 concludes the paper.

2 Background

In this section, we demonstrate the overview and implementation challenges in a multi-tier JIT compiler. Subsequently, we present an overview of the meta-compiler framework. Finally, we introduce meta-compiler–based threaded code generation, a promising technique for lightweight compilation. Finally, we provide an overview of the Simple Object Machine that we used to implement our proposal.

2.1 JIT and Multi-Tier JIT Compilation

In general, most VMs have interpreters as their first execution tier. This is because the interpreter is easy to implement and can perform profiling tasks such as type information collection. For many languages, a VM developer can likely develop a lightweight JIT compiler with moderate effort. A lightweight JIT compiler generates code quickly, but the performance of the generated code is moderate. To achieve the desired performance, heavyweight JIT compilers are implemented. The code that they generate is highly optimized but compilation consumes markedly more time than lightweight JIT compilation.

Most traditional VMs such as Java and JavaScript VMs [32, 26, 13, 18, 34] use multi-tier JIT compilation [1]. A multi-tier JIT compilation VM involves a hierarchy of JIT compilers, each offering different optimization tiers. As the optimization tiers grow in number, compilation time also increases, and the resulting code achieves higher performance. Most multi-tier JIT compilation VMs include at least two lightweight and heavyweight compilers.

Multi-tier JIT compilation dynamically profiles the runtime information of a source program to select the most appropriate compiler for compilation and execution. Additionally, it transitions execution to higher optimization levels when beneficial, thereby enhancing the overall JIT performance.

The early execution phase, evaluated by *warm-up performance*, and the computationally intensive phase, evaluated by *peak performance*, are crucial for improving the VM's overall performance. The lightweight compiler is typically used to accelerate warm-up, whereas the heavyweight compiler focuses on maximizing peak performance. Below, we discuss the relationship between VM execution, warm-up performance, peak performance, and multi-tier JIT compilation in detail.

2.1.1 Warm-up and Peak Performance in Multi-Tier JIT Compilation

Warm-up performance refers to the efficiency during the phase required for a VM to transition from its initial execution. This phase typically involves interpretation or lightweight compilation to a state where the optimized machine code generated by the JIT compiler is actively used. This transition is critical for interactive or short-lived applications, for which execution speed during the early stages of program execution exerts a significant impact on user experience. For long-running server applications, however, warm-up performance is less critical than peak performance.

Peak performance, conversely, represents the maximum level of processing power and resource utilization that a VM can achieve under optimal conditions. This phase is usually reached after the warm-up phase, when the VM is executing fully optimized code. This is particularly important for long-running applications, such as server processes, for which sustained high performance is essential. The C2 compiler in HotSpot JVM [32] is designed to maximize peak performance by leveraging the profiling data gathered during the warm-up phase to generate highly optimized machine code. However, achieving peak performance comes with trade-offs including the overhead of profiling, the complexity of code transitions, and potential inefficiencies caused by suboptimal profiling parameters or compilation strategies.

In the case of HotSpot JVM, the C1 compiler [26] generates moderately optimized code quickly, allowing the application to execute at a reasonable speed while gathering profiling information. This profiling data is then used by the C2 compiler [32] to produce highly optimized code, improving performance over time. However, the extra overhead introduced by tiered compilation, such when as profiling and transitioning between different levels of compiled code, can undermine warm-up performance if not carefully managed.

Multi-tier JIT compilation technique ensures that frequently executed code is greatly optimized and achieves faster peak performance. On the other hand, the code parts that are not frequently executed benefit from quicker compilation, leading to the improvement of warm-up performance. Correspondingly, overall execution performance is improved by tailoring the level of optimization to the usage frequency of each part of the code.

2.2 Meta-Compilation

While interpreters are easy to implement, understand, and extend, implementing a JIT compiler is an error-prone task and requires engineering work-hours summing, at times, to dozens of years.

Meta-compilation reduces the amount of work in implementing an individual JIT compiler by making the JIT compiler reusable and language-independent. A meta-compiler compiles the source program together with the interpreter that runs it. By using both the source code and the interpreter for compilation, meta-compilers can produce highly optimized code that is equivalent to state-of-the-art JIT compilers.

In general, a meta-compiler compiles a source program together with an interpreter that executes the source program. The process involves the following steps:

1. **Hot Spot Identification**: identifying hot spots by executing the interpreter,
2. **Behavior Extraction**: extracting the selected hot spot's behavior, and
3. **Native Code Translation**: translating this behavior into native code.

The current meta-compilation system differs from the method of extracting the hot spot's behavior. The meta-tracing JIT compilation system in RPython [6] traces the interpreter as it executes the hot spot, whereas the self-optimizing interpreter in Graal/Truffle [39] applies partial evaluation to it.

Next, we introduce RPython, which we use as the basis of our proposal.

2.2.1 RPython: A Meta-Tracing Compiler Framework

While traditional JIT compilers compile a frequently executed method as their compilation unit, tracing JIT compilers [2, 16, 3] compile the execution path of a source program called *trace*. Typically, tracing JIT compilers compile a loop in a source program.

The RPython framework [36, 6] uses tracing JIT compilation for its meta-compilation technique. A meta-tracing JIT compiler doesn't trace a source program directly but instead traces the executing interpreter. The RPython framework provides annotations to indicate which part of the interpreter can be a compilation target. For instance, to find a loop in a source program, an annotation is inserted where the back edge of the loop occurs in the interpreter. The meta-tracing compiler starts to trace there when a profiling counter on a back edge exceeds a threshold. Then, tracing continues until it reaches the inserted annotation again. The resulting trace is compiled into machine code, and the subsequent executions of the compiled source program are run in the generated machine code.

To further improve performance, VM developers can use annotations to convey more information to the meta-tracing JIT compiler. For instance, VM developers can specify that certain variables in the interpreter implementation should be constant, allowing for more aggressive optimizations. In addition, by specifying a function as side-effect–free, the meta-tracing JIT compiler can replace the function call with a resulting constant if all of the passed arguments are constant. If not, when the meta-tracing JIT compiler sees the same operation on the same arguments again later, it reuses the result from the previous operation's result. In addition, if inlining a particular function defined in an interpreter would make the performance worse, the developers can avoid inlining that function by specifying an annotation to leave a call instruction on it.

2.3 Meta-Compiler–Based Threaded Code Generation

Threaded code [4, 12, 9, 10] is a form of machine code generated from a bytecode-based source program and its interpreter. Among various types of threading (which include direct threading [4] and indirect threading [9, 10], and subroutine threading [9, 10]) compiles a sequence of bytecode instructions into corresponding call instructions to their respective handlers. Threaded code generation is a lightweight compilation technique, simplifying the process and eliminating the overhead of bytecode fetching and dispatching [14].

Meta-compiler–based threaded code generation [25] is a compilation technique for lightweight compilation that generates subroutine-threaded code using a meta-tracing JIT compiler. Unlike other threaded code generators that develop a compiler from scratch, this approach leverages the interpreter definitions used in the compilation of the meta-tracing JIT compiler. Specifically, it makes use of the annotation mechanisms provided by the RPython framework. For instance, annotations are added to prevent inlining within all handler functions in the interpreter definitions, allowing call instructions to bytecode handlers to be aligned sequentially in the resulting trace. Additionally, while tracing JIT compilers trace only one side of a branch; this technique introduces mechanisms to trace all possible paths in the source program in one shot to compile the entire method. In this way, a new lightweight compilation technique is achieved without creating an entirely new compilation pipeline within the meta-tracing compiler.

The workflow for threaded code generation is illustrated in Figure 2. First, the threaded code generator traces `strange_add` by executing its bytecode, as shown in Listing 2. During this process, it records call instructions to the corresponding handler functions in the trace. When it encounters a branch instruction, such as `JUMP_IF L1`, it first traces the false branch. Upon reaching `RET`, a dummy instruction, `cut_here(return)`, is temporarily added to the trace. The technique then returns to trace the true branch, stopping at `RET` again. The resulting trace, shown in Listing 3, includes only calls to handlers, along with guards, labels, and pseudo instructions such as `cut_here`.

Next, the technique splits the trace into two parts, `Trace A` and `Trace B`, replacing the dummy instructions with actual RPython intermediate representations (e.g., `finish()`). The split traces are shown in Listing 4. Finally, it stitches `Trace B` back into `Trace A` at the guard point, `guard_false(i2)`. This stitched trace is then compiled into the final assembly code, as displayed in Listing 5.

Meta-compiler–based threaded code generation demonstrates promise as a foundation for multi-tier JIT compilation in a meta-compiler framework, because it simplifies the integration of a lightweight JIT compiler within the RPython framework. However, meta-compiler–based threaded code generation was limited to *offline* code generation, and the generated code was not optimized sufficiently. This paper, therefore, proposes optimizations to enable practical code generation (discussed in Section 5).

2.4 Simple Object Machine

Simple Object Machine (SOM) is a dynamically typed language similar to Smalltalk. It is designed as a subset of Smalltalk for the purpose of teaching and researching the implementation of a VM [19]. SOM supports class inheritance, closures, and non-local returns. The SOM family comprises many VM implementations written in C/C++, Rust, Java, Python, and Smalltalk.

Our proposed two-level SOM is based on PySOM. PySOM is a SOM VM that has AST and bytecode interpreters written in RPython. We are going to adopt the bytecode interpreter to implement 2SOM.

Listing (1) Example SOM program.

```
C = (
  calc: n = (
    | x |
    # for ...
    1 to: n do: [ |i|
      x :=
        "Addition: call strange_add"
        x + strange_add: i m: n.
    ] )

  "Newly added method"
  strange_add: n m: m = (
    [ n % 42 == 0 ]
      ifTrue:  [ ^ (m - 42) ]
      ifFalse: [ ^ (n + m) ] )

  "Entry point of this program"
  run = (
    calc: 10000.
  ) )
```

Listing (2) Pseudo bytecode format corresponding to the program shown in Listing 1.

```
calc:
  DUP      # n
  CONST 1 # i
L0:
  CALL "strange_add"
  # ... Same to Figure 2 ...
  RET
strange_add:
  DUP         # n
  CONST 42
  MOD         # n % 42
  JUMP_IF L1
# false branch
  DUP         # n
  DUP         # m
  ADD         # n + m
  RET
# true branch
L1:
  CONST 42
  DUP         # n
  SUB         # n - 42
  RET
```

Figure 1 Example source program and corresponding bytecode representation in 2SOM.

Listing (3) Temporal trace from `strange_add` traced by the threaded code generation.

```
# p0 means a red var.
# in interpreter
label(strange_add)
call(DUP, p0)
call(CONST,p0,  42)
i2 = call(MOD, p0)
guard_false(i2)
# false branch
call(DUP, p0)
call(DUP, p0)
call(ADD)
cut_here(return)
# true branch
call(CONST, p0, 42)
call(DUP, p0)
call(SUB, p0)
finish(p0)
```

Listing (4) Traces after reconstructing the control flow.

```
# Trace A
label(strange_add)
call(DUP, p0)
call(CONST, p0, 42)
i2 = call(MOD)
# go to L1 if failed
guard_false(i2) [L1]
# false branch
call(DUP, p0)
call(DUP, p0)
call(ADD, p0)
finish()

# Trace B
# true branch
label(L1)
call(CONST, p0, 42)
call(DUP, p0)
call(SUB, p0)
finish()
```

Listing (5) Pseudo-assembly code compiled from Listing 4.

```
strange_add:
  push p0
  call DUP
  push 42
  push p0
  call CONST
  push p0
  call MOD
  jnz L1
  push p0
  call DUP
  push p0
  call DUP
  push p0
  call ADD
  ret
L1:
  push 42
  push p0
  call CONST
  push p0
  call DUP
  push p0
  call SUB
  ret
```

Figure 2 Compilation steps of the meta-compiler–based threaded code generation.

3 Problem

While multi-tier JIT is used to improve performance in widely used VMs, implementing a multi-tier JIT compilation VM is harder than implementing a single-tier JIT compilation VM. The multi-tier option requires at least the following components of VM developers:

(Multiple compiler implementations:) Both lightweight and heavyweight compilers are required, each generating code at different optimization levels, balancing code generation time and execution efficiency.

(Profiling mechanism:) A profiling mechanism is necessary to dynamically determine which compiler to invoke. The profiler measures the execution frequency of methods or loops in an interpreter or in code generated by a lightweight compiler. Profiling data are collected to trigger higher optimization levels for frequently executed code.

(Transition mechanism from code to code generated by different compilers:) A mechanism is required to transition between code generated from different compilers, enabling seamless transitions between different compilation tiers.

Due to these complexities, it is not trivial to generate a multi-tier JIT compilation VM using a meta-compiler framework. Thus, a naïve approach may unnecessarily increase the implementation costs. The primary problem lies in the absence of a mechanism for integrating multiple compilers into such frameworks. One potential approach is to separately implement both lightweight and heavyweight compilers within the framework, coupled with a profiler to trigger transitions and manage the control flow between code from different compilers. While this approach appears feasible, it incurs significant implementation costs comparable to those already associated with traditional VMs featuring multi-tier JIT compilation.

4 Two-Level Compilation by Integrating Meta-Compiler–Based Threaded Code Generation

To address the problems outlined in Section 3, we propose a novel methodology for generating a multi-tier JIT compilation VM with minimal implementation costs. This methodology leverages the RPython framework to generate a multi-tier JIT compiler VM directly from interpreter definitions, reducing the need to implement several necessary components for multi-tier JIT compilation from scratch. As a proof of concept, we generate 2SOM, a two-level multi-tier JIT compiler VM for Simple Object Machine [19], using RPython. We will begin by showing the architecture and behavior of 2SOM in Section 4.1 and then describe how we realize it in Section 4.2.

4.1 Architecture and Behavior of 2SOM

The architecture of 2SOM, illustrated in Figure 3, uses a single meta-tracing JIT compiler for multi-tier JIT compilation to simplify the implementation of multiple compilers. In this architecture, the interpreter controls whether lightweight or heavyweight compilation is applied. The interpreter for lightweight compilation, which is called the *lightweight interpreter* here, generates threaded code [25] in collaboration with the meta-tracing JIT compiler to improve warm-up performance. Conversely, the interpreter for heavyweight compilation (the *heavyweight interpreter*) generates trace-based, highly optimized code for peak performance. To manage runtime integration, 2SOM incorporates a profiler and switcher. The profiler identifies hot spots in code compiled by a lightweight JIT compiler, and the switcher transitions execution to heavyweight compilation when necessary, ensuring

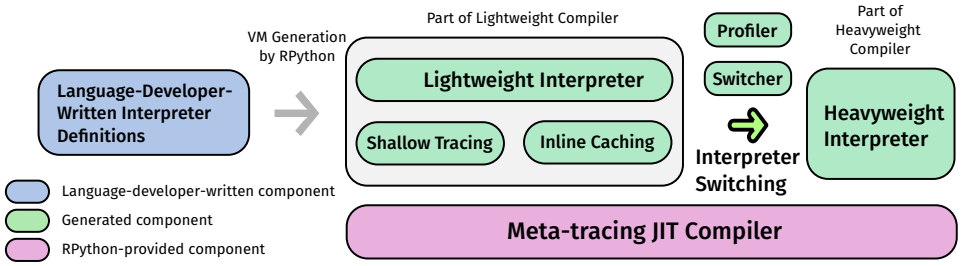


Figure 3 Architecture of 2SOM.

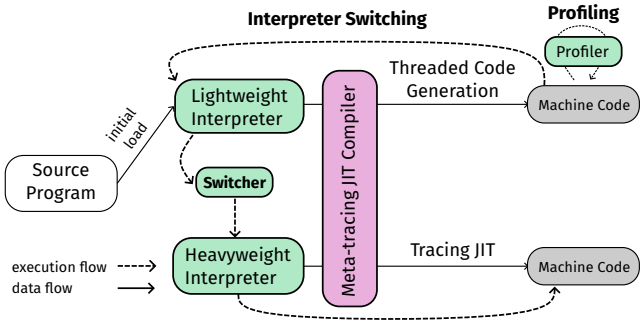


Figure 4 Execution flow of 2SOM.

Listing (6) Obtained trace from the lightweight compilation tier.

```
# Trace A
label(strange_add)
call(DUP)
call(CONST, 42)
i2 = call(MOD)
# go to L1 if failed
guard_false(i2) [L1]
# false branch
call(DUP)
call(DUP)
call(ADD)
finish()

# Trace B
# true branch
label(L1)
call(CONST, 42)
call(DUP)
call(SUB)
finish()
```

Listing (7) Obtained trace from the lightweight compilation tier.

```
# p0 means a red var. in interpreter
label(loop)
i0 = getarrayitem(p0, 0) # get n
i1 = getarrayitem(p0, 1) # get i
i2 = getarrayitem(p0, 2) # get x
# inlining strange_add
i3 = int_mod(i0, 42)     # n % 42
guard_false(i3)          # n % 42 == 0
i4 = int_add(i1, i0)     # ^(n + m)
# inlining finished
i5 = int_add(i2, i4)     # x + strange..
i6 = int_add(i1, 1)      # incr i
i6 = int_le(i6, i0)
guard_true(i6)
setarrayitem(p0, 1, i6) # set to x
setarrayitem(p0, 2, i5) # set to i
jump(loop)
```

Figure 5 Obtained traces from 2SOM when executing the program shown in Figure 1.

efficient optimization and seamless control flow. The profiler and switcher are implemented as interpreter definitions for the sake of reducing the need to develop complex components from scratch.

Next, we illustrate how 2SOM's multi-tier JIT compilation works, using the program shown in Figure 5 and the traces presented in Figure 5. An overview of 2SOM's execution flow is provided in Figure 4.

As depicted in Figure 4, the execution of 2SOM begins in the lightweight interpreter with the `run` function, which serves as the program's entry point. This function calls `calc:` with an argument of 10,000. The lightweight interpreter then processes the `calc:` function, starting with `initialization: x`. Subsequently, the `initialization` function is compiled into a subroutine-threaded code by the lightweight compiler. Subroutine-threaded code replaces bytecode instructions with direct calls to operation handlers, ensuring efficient execution with minimal overhead compared to heavyweight compilation. The trace obtained from this process is shown in Listing 6.

After initialization, the `calc` function enters a loop that iterates from 1 to `n`. Within this loop, the result of `strange_add: i m: n` is repeatedly added to `x`. The `strange_add: i m: n` function is initially compiled by the lightweight compiler. This loop involves frequent backward jumps, which are monitored by an embedded profiler in the code compiled by the lightweight interpreter, as depicted by Figure 4. Once the frequency of these backward jumps surpass a predefined threshold, the profiler identifies the loop as a "hot" spot suitable for more aggressive optimization.

At this stage, the interpreter switcher transfers control from the lightweight interpreter to the heavyweight interpreter. The interpreter switcher recovers the frame used by the lightweight interpreter and passes it to the heavyweight interpreter for continued execution. The heavyweight interpreter performs trace-based compilation, applying advanced techniques such as loop unrolling and inlining. The tracing JIT compiler then compiles the entire loop, inlining and optimizing the `strange_add` function. Hence, the remainder of the loop executes using highly optimized machine code generated by the heavyweight compilation tier, thereby improving efficiency while maintaining correctness. The corresponding trace is presented in Listing 7.

4.2 Technical Details

In this section, we describe the technical details of realizing 2SOM. We describe how each of the three elements essential for realizing two-level JIT compilation in 2SOM is achieved, specifically using interpreter definitions and the RPython framework.

4.2.1 Multiple Compiler Implementations

To enable multi-tier JIT compilation without the complexity of implementing multiple distinct compilers, we introduce meta-compiler–based threaded code generation [25] into RPython. Consequently, RPython can generate both generator and compiler from the two interpreter definitions with different specifications of compiler strategies. In this setup, the meta-compiler-based threaded code generation works as a lightweight compiler, while the tracing JIT compiler serves as the heavyweight compiler.

Using this technique minimizes the development and maintenance overhead associated with creating and supporting multiple compilers. Additionally, VM developers can leverage the interpreter annotations and hints provided by RPython to further customize the compilation process, tailoring optimization levels to the unique requirements of their language.

To illustrate this technique, we provide examples of implementing a heavyweight compiler, which will be followed later by a lightweight compiler example.

For the heavyweight compiler, we use the original interpreter definition originally provided by the RPython framework. In this interpreter, a `jitdriver` is created to configure the constant variables (greens) and varying variables (reds) in the dispatching loop. Also, hints are used to guide the JIT compiler in optimizing frequently executed code paths. The hint

Listing 8 Lightweight interpreter for threaded code generation.

```
# greens: names of constant
# reds: varying variables (reds)
threadeddriver = JitDriver(
  greens=['bytecodes','pc'],
  reds=['frame'],
  threaded_code_gen=True)

def threaded_interpret(frame):
  # entry point of lightweight compilation
  threadedriver.can_enter_jit(frame=frame,
   bytecodes=frame.bytecodes, pc=frame.pc)

  while True:
    # indicate dispatch loop to
    # the meta-tracing compiler
    threadedriver.jit_merge_point(
      frame=frame,
      bytecodes=frame.bytecodes,
      pc=frame.pc)

    opcode = frame.bytecodes[frame.pc]
    if opcode == ADD:
      handler_add(frame)
    elif opcode == SUB:
      handler_sub(frame)
    # ... other handlers ...
    frame.pc += 1

@enable_threaded_code
def handler_add(frame):
  w_y = frame.pop()
  w_x = frame.pop()
  frame.push(w_x.add(w_y))

# ... other handler functions ...
```

Listing 9 Heavyweight interpreter for tracing compilation.

```
tracingdriver = JitDriver(
  greens=['bytecodes','pc'], reds=['frame'])

def tracing_interpret(frame):
  while True:
    # indicate dispatch loop to
    # meta-tracing compiler
    tracingdriver.jit_merge_point(
      frame=frame,
      bytecodes=frame.bytecodes,
      pc=frame.pc)
    opcode = frame.bytecodes[frame.pc]

    # handle back-ward jump
    if opcode == JUMP_BACKWARD:
      # entry point of heavyweight
      # compilation
      tracingdriver.can_enter_jit(
        frame=frame,
        bytecodes=frame.bytecodes,
        pc=frame.pc)

    elif bytecode == ADD:
      handler_add(frame)
    elif bytecode == SUB:
      handler_sub(frame)
    # ... other handlers ...
    frame.pc += 1

def handler_add(frame):
  w_y = frame.pop()
  w_x = frame.pop()
  frame.push(w_x.add(w_y))

# ... other handler functions ...
```

`jit_merge_point` tells the meta-tracing JIT compiler the part of interpreter-dispatching loop, and the `can_enter_jit` informs an entry point of the JIT compilation to the meta-tracing JIT compiler.

The heavyweight interpreter is defined as displayed by Listing 9. The interpreter guides the meta-tracing JIT compiler to perform heavyweight compilation. To indicate the place of the dispatching loop, the `jit_merge_point` hint is placed just after `while True: ...` To compile the loop of a source program, `can_enter_jit` hint is placed at the handler for `JUMP_BACKWARD`, which corresponds to the back-edge jump in a source program.

For the lightweight compiler, we define an interpreter that enables the meta-tracing JIT compiler to perform meta-compiler-based threaded code generation. The major differences from the heavyweight interpreter are that all handlers are annotated with `enable_threaded_code` and the placement of `can_enter_jit`. The `enable_threaded_code` annotation serves as an instruction to retain calls to handlers in the trace. Moreover, to compile the method body, `can_enter_jit` is placed at the very beginning of the method call–that is, just before the interpreter is invoked and enters the dispatching loop.

The lightweight interpreter is defined as shown in Listing 8. The handlers `handler_add` and `handler_sub` are decorated with the annotation `enable_threaded_code`, and the placement of `can_enter_jit` is moved to just before the entry of the dispatching loop.

Listing 10 Embedded profiler in the interpreter

```
HOT_THRESHOLD = 1000

def threaded_interpret(frame):
  while True:
    opcode = frame.bytecodes[frame.pc]

    # backward jump should be
    # compiled by heavyweight compiler
    if opcode == JUMP_BACKWARD:
      # increment the counting value
      count = frame.counts.get(frame.pc,0)+1
      frame.counts[frame.pc] = count

      if count == HOT_THRESHOLD:
        # Trigger heavyweight compilation
        raise ContinueInTier2(frame)

    # ... Execute other opcodes ...
    frame.pc += 1
```

Listing 11 Interpreter switcher that dispatches from lightweight to heavyweight interpreters

```
def interpet_switcher(bytecode):
  frame = Frame(bytecode)
  while True:
    try:
      # Start interpretation first
      # in lightweight interpreter
      w_x = threaded_interpret(frame)
    except ContinueInTier2 as e:
      # Catch thrown ContinueTier2
      # from a generated threaded code,
      # and Continue to execute in
      # heavyweight interpreter
      w_x = tracing_interpret(e.frame)
    return w_x
```

4.2.2 Profiling Mechanism

Integrating different compilers requires a mechanism to decide when to trigger heavyweight compilation during executing code generated by a lightweight JIT compiler. We achieve this by generating code fragments that can profile runtime information in the code compiled by a lightweight JIT compiler. These generated profilers are responsible for monitoring the execution frequency of code spots and triggering recompilation at higher optimization levels when necessary.

The profiler works using a simple frequency counter that increments each time a particular code spot, such as a method or loop, is executed. When the counter exceeds a predefined threshold, the profiler determines that the code spot is "hot" and would benefit from additional optimization. At this point, the profiler triggers the heavyweight tracing JIT compilation for that spot.

Under our methodology, this profiling mechanism provides a flexible method to manage multiple compilers generated by RPython. VM developers can adjust the thresholds and profiling strategies to suit the dynamic behavior of their language in their interpreter definitions.

The example definition of the profiler is depicted in Listing 10. This profiler monitors the execution of specific opcodes. In this definition, backward jumps – typically indicating loops – are identified as key candidates for heavyweight compilation. When a backward jump opcode (`JUMP_BACKWARD`) is encountered, the profiler increases a counter tied to the program counter (`frame.pc`) stored in the `frame.counts` dictionary, which tracks execution frequencies for each opcode.

If this counter exceeds the predefined `HOT_THRESHOLD` (set to 1000 in this case), the profiler marks the code spot as "hot" and suitable for further optimization. At this point, the profiler triggers heavyweight compilation by using a global jump. Because RPython does not provide global jumping, we use an exception mechanism instead. This exception, called `ContinueTier2`, halts the lightweight interpreter and switches control to the heavyweight interpreter. To make the transition work correctly, the exception contains an execution context. The exception is used in the transition mechanism described in Section 4.2.3.

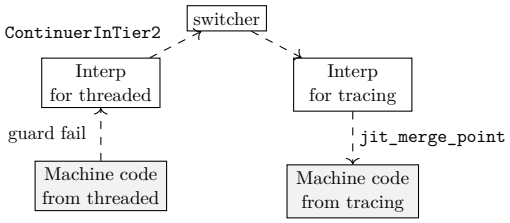


Figure 6 Overview of interpreter switching technique.

4.2.3 Transition Mechanism Between Generated Code

Finally, we need a mechanism to effect transitions from code generated by a lightweight JIT compiler to that generated by a heavyweight JIT compiler. This is achieved through the *interpreter switching* technique, which redirects the execution in code compiled by the lightweight compiler to the code compiled by the heavyweight compiler.

We overview the technique in Figure 6. When the profiler, the function of which is explained in Section 4.2.2, detects a hot spot suitable for tracing JIT compilation, a guard fail occurs. Then, a global jump from code generated by a lightweight JIT compiler to the lightweight interpreter happens. Next, the lightweight interpreter throws an exception to move to the interpreter switcher. The switcher catches it and resumes the execution context. Next, it launches the heavyweight interpreter with the resumed execution context. If the running program is already compiled by the heavyweight compiler, the control goes to the compiled machine code via `jit_merge_point`. Otherwise, the program is interpreted and will be compiled via `can_enter_jit` in the heavyweight interpreter.

The interpreter switcher is implemented as described in Listing 11. The `interpret_switcher` function handles this transition. It starts with the lightweight interpreter and, upon catching a `ContinueTier2` exception thrown from `threaded_interpret`, passes the execution context (`e.frame`) to the heavyweight interpreter `tracing_interpret`. Both interpreters return a result, `w_x`, to the caller, ensuring consistent execution flow.

Together with the profiling and interpreter switching technique explained in sections 4.2.2 and 4.2.3, we can maintain correctness while optimizing performance. By leveraging the interpreter for managing transitions, the framework provides a basis for multi-tier JIT compilation.

However, this mechanism can introduce significant runtime overhead because the control bypasses the interpreter when it goes to other code. We evaluate the overhead by measuring the peak performance of 2SOM's two-level compilation in section 6.

5 Improving Meta-Compiler–Based Threaded Code Generation

In 2SOM, we use meta-compiler–based threaded code generation [25] for lightweight compilation. However, its integration with a meta-tracing JIT compiler in 2SOM has exposed problems, as the earlier implementation [25] focused solely on code generation. Therefore, we analyzed the technical problems that arise when introducing meta-compiler–based threaded code generation to the meta-tracing compiler framework and introduced a new technique and optimization to enhance its efficiency and compatibility within the meta-tracing JIT compiler.

Listing 12 Example of SOM program that has a side effect.

```
"Compute the sum of arr"
strange_sum_arr: arr index: i
  sum: n = (
  [ i <= arr length ]
    ifTrue: [
     sum_array: arr
      index: (i + 1)
      sum: (n + (arr at: i)).
    ]
    ifFalse: [
     "Clear all the elements"
     arr clear. ^ n  ] )

"Entry point"
run = (
  | arr |
  "Create an array where"
  "all elements are 1"
  arr :=
    Array new: 30 putAll: 1.
  strage_sum_arr: arr
    index: 1 sum: 0. )
```

Listing 13 Simplified bytecode from Listing 12.

```
strange_sum_arr:
 DUP0 # arr
      # arr length
 CALL(length, frame)
 DUP2 # i
 LE   # i <= arr length
 JUMP_IF_FALSE L1
 DUP1 # i
 CONST 1
 ADD  # i + 1
 DUP1 # arr
 CALL(at, frame)
      # arr at: i
 DUP2 # n
 ADD  # n + (arr at: i)
 JUMP strange_sum_arr
L1:
 DUP0 # arr
 CALL(clear, frame)
 DUP2 # n
 RET
```

Listing 14 Handler function that interprets `CALL` instruction.

```
def threaded_interpret(frame):
  ...
  while True:
    ...
    if opcode == CALL:
      handler_CALL(frame)
```

```
    # ... XXX ...

@dont_look_inside
def handler_CALL(frame):
  method = frame.bytecodes[frame.pc+1]
  newframe = crate_frame(frame, method)
  w_x = threaded_interpret(newframe)
  frame.push(w_x)
```

5.1 Runtime Problem of the Meta-Compiler-Based Threaded Code Generation

Meta-compiler–based threaded code generation exhibits unsatisfactory runtime execution and performance. In particular, the following technical problems occur when we use it in 2SOM.

1. If the definition of the lightweight interpreter has side effects, tracing the lightweight interpreter leads to an incorrect state due to tracing both conditional branches in a row.
2. Function calls always become indirect in the compiled code.

The first problem occurs because meta-compiler–based threaded code generation and tracing JIT compilation use different compilation scopes (method-based for the threaded code generation and trace-based for tracing compilation) but rely on the same compilation engine. For example, when performing threaded code generation, as described in the previous section, branches in the code are linearized into a single sequence during tracing. This means that even branches that would not be executed during actual program runs are included in the trace. If these unnecessary branches are executed during tracing, it can lead to an invalid runtime state.

Consider a simple `if-then-else` statement where the `then` branch is always taken during execution. During threaded code generation, both branches are traced at compilation time as described in Section 2. Although the meta-tracing JIT compiler executes the source

program with the interpreter at compilation, tracing the `else` branch can modify variables or states, causing inconsistencies. For example, the program in Listing 12[1] computes the sum of an array: the `then` clause (receiver of `ifTrue`) adds each value to `n`, while the `else` clause (receiver of `ifFalse`) returns `n`. However, executing the `clear` function in the `else` clause, which sets the array to zeros, corrupts the runtime state if na"ively traced by the meta-compiler–based threaded code generation.

The second problem arises because the meta-tracing JIT compiler could not identify the target of a handler function wrapped in a call instruction during tracing, when the handler represented a method invocation. In general, calling a dynamically bound method takes longer time than calling a statically bound method. This is because a system needs to look up a correct method by using the runtime class of a callee method. The same kind of problem occurs in threaded code generation.

For example, consider a handler function `handler_CALL` that dynamically dispatches a method call as illustrated in Listing 14. During tracing, the meta-tracing JIT compiler encounters the call instruction for `handler_CALL` but cannot identify the callee method compiled at tracing. Without knowing whether the target method is compiled, the compiler cannot optimize the method call, which leads to worse performance.

In meta-compiler–based threaded code generation, this could result in repeated interpretation or inefficient indirect calls, even if the invoked method is a commonly used one that would benefit from inlining or specialized optimizations. This lack of optimization is particularly problematic for workloads with frequent method invocations, as it leads to missed performance improvements.

5.2 Our Solutions

In this section, we show the solutions, namely, shallow tracing and direct calls by inline caching to the first and second problems.

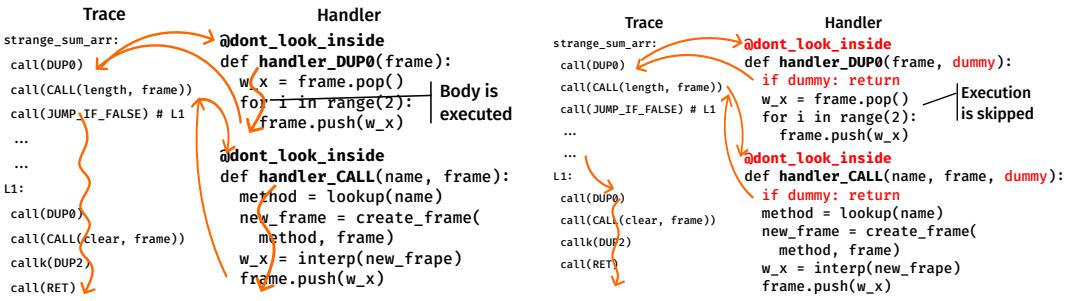
5.2.1 Shallow Tracing

For the first problem, we propose the *shallow tracing* technique. Shallow tracing prevents side effects during tracing by introducing lightweight annotations within the interpreter. These annotations guide the tracing process to leave only call instructions to handler functions, while the actual execution of these handlers is avoided. In contrast, during the interpretation and execution of generated machine code, the annotations have no effect, ensuring seamless runtime execution.

The fundamental reason for needing shallow tracing lies in the fact that, while the `dont_look_inside` annotation retains the call instruction to the handler function, it still executes the function's body, using the interpreter's state. This is not a problem in standard tracing compilation. However, in method-based threaded code generation, both branches of a conditional are traced. If side effects occur during the tracing of one branch, they propagate to the tracing of the other branch, corrupting the interpreter's state.

Shallow tracing suppresses the execution of the body that would otherwise occur under `dont_look_inside`. This technique relies on the following two key annotations and the dummy flag mechanism:

[1] Its simplified bytecode appears in Listing 13.



(a) Flow of tracing before applying shallow tracing. The left-hand side and right-hand side show the trace and the handler functions, respectively.

(b) Flow of tracing after applying shallow tracing. The left-hand side and right-hand side show the trace and the handler functions, respectively.

Figure 7 The problem with naïvely integrating meta-compiler-based threaded code generation with a meta-tracing JIT compiler (left-hand side), and our solution to it (right-hand-side). In each side, orange lines are the flow of tracing. In naïve integration, the body of the handler function is executed during tracing, which can lead to side effects that corrupt the interpreter's state. To resolve this, we introduce shallow tracing, a technique that prevents the execution of the handler function's body during tracing, thereby avoiding such side effects.

- `we_are_jitted`: Checks whether the execution is currently within a traced and compiled context, and
- `dont_look_inside`: Ensures that only call instructions to annotated functions are traced without inlining their bodies of annotated functions, and
- `dummy` flag mechanism: the dummy flag is inserted into every handler. It is turned into `True` inside the then branch conditioned with `we_are_jitted`. When this flag is on, the dummy return is triggered. When this trace is compiled and executed, the dummy flag is turned into `False`, allowing the handler's body to execute correctly.

Its overall execution flows are compared in Figure 7: the left-hand side illustrates the trace without shallow tracing, while the right-hand side shows the execution flow with shallow tracing applied. Shallow tracing introduces a *dummy flag* and a *dummy return* for each handler (except for control-flow–related handlers like `JUMP_IF_FALSE`). Also, the implementation in an interpreter and the resulting trace are shown in Listings 15 and 16, respectively.

5.2.1.1 Optimizing Shallow Tracing

Manually implementing the code in Listing 15 is complex and time-consuming for VM developers. Naïve insertion of a dummy flag into handlers also risks performance overhead. To address this, we automate the interpreter generation at VM generation time, while optimizing shallow-tracing code by eliminating redundant `dummy` flag checks.

We introduce a new annotation, `enable_threaded_code`, to automate this process. Annotated handlers are automatically marked with `dont_look_inside` and a dummy flag, and their call sites are wrapped with `we_are_jitted`, leveraging the RPython VM generation process [36]. Developers only need to annotate each handler function, as shown in Listing 17, and the annotated interpreter is transformed into the structure in Listing 18.

To mitigate the dummy flag overhead, we apply a two-step optimization. First, for each annotated handler, we generate both a stub handler (with a dummy check) and a real handler (without the check) under the same name. Second, during shallow tracing, only stub handlers

Listing 15 Naïve version of an interpreter that supports shallow tracing.

```
def threaded_interpret(frame):
  while True:
    opcode = frame.bytecodes[frame.pc]
    if opcode == ADD:
      # returns true while tracing and
      # executing machine code
      if we_are_jitted():
        # set 'dummy' True
        # to avoid side-effects
        handler_add(frame, dummy=True)
      else:
        # set 'dummy' False
        # execute body
        handler_add(frame, dummy=False)

    elif opcode == SUB:
      if we_are_jitted():
        handler_sub(frame, dummy=True)
      else:
        handler_sub(frame, dummy=False)
    # ... other handlers ...
    frame.pc += 1

# leaves only handler call but
# executes the body
@dont_look_inside
def handler_add(frame, dummy):
  # immediately return when dummy
  # is true to avoid side-effects
  if dummy: return
  w_y, w_x = frame.pop(), frame.pop()
  frame.push(w_x.add(w_y))

@dont_look_inside
def handler_sub(frame, dummy):
  if dummy: return
  # ... subtract two elements in
  # frame.stack ...
```

Listing 16 Naïve version of a generated trace by shallow tracing.

```
label(strange_sum_arr)
call(handler_DUP, p0, True)
call(handler_CALL, "length", p0, True)
call(handler_DUP2, p0, True)
i1 = call(handler_LE, p0, True)
# i1 is initialized with a default value (0)
guard_false(i1)
call(handler_DUP1, p0, True)
call(handler_CONST, p0, 1, True)
call(handler_ADD, p0, True)
call(handler_ADD, p0, "at", True)
call(handler_DUP2, p0, True)
call(handler_ADD, p0, True)
jump(strange_sum_arr)

label(L1)
call(handler_DUP0, p0, True)
call(handler_CALL, "clear", p0, True)
finish()
```

are traced to build the instruction-to-handler mapping; after tracing, they are replaced with real handlers at compile time. Listings 19 and 20 show the traces before and after this optimization.

5.2.2 Direct Calls with Inline Caching

Inline caching [11] is a well-known runtime optimization technique used to accelerate method dispatch in object-oriented programming languages such as Smalltalk-80, SELF, and Java. It optimizes method calls by caching method lookups directly at the call site, avoiding redundant type resolution when the same types are repeatedly encountered. If the cached type matches the runtime type of the receiver, a fast path is taken; otherwise, the system falls back to a slow path for dynamic method resolution [20].

In the absence of inline caching in meta-compiler–based threaded code generation, method calls to compiled code are executed as indirect calls. Such calls bypass `jit_merge_point`, which can lead to significant performance degradation when methods are invoked repeatedly. Direct calls with inline caching alleviate this by converting indirect calls into direct calls, thereby improving execution speed. Figure 8 provides an overview of this transformation at runtime, revealing the transition from indirect to direct calls enabled by inline caching.

Listing 17 Annotated version of the interpreter design for shallow tracing.

```
def threaded_interpret(frame):
  while True:
    opcode = frame.bytecodes[frame.pc]
    if opcode == ADD:
      handler_add(frame)
    elif opcode == SUB:
      handler_sub(frame)
    frame.pc += 1

@enable_threaded_code
def handler_ADD(frame):
  w_y = frame.pop()
  w_x = frame.pop()
  frame.push(w_x.add(w_y))

@enable_threaded_code
def handler_SUB(frame):
  # subtract two elements in frame.stack
```

Listing 18 Generated stub and original handlers.

```
def threaded_interpret(frame):
  while True:
    opcode = frame.bytecodes[frame.pc]
    if opcode == ADD:
      if we_are_jitted():
        stub_ADD(frame, True)
      else:
        ADD(frame)
    elif opcode == SUB:
      if we_are_jitted():
        stub_SUB(frame, True)
      else:
        SUB(frame)
    frame.pc += 1

@dont_look_inside
def stub_ADD(frame, dummy):
  if dummy: return
  ADD(frame)

@dont_look_inside
def stub_SUB(frame, dummy):
  if dummy: return
  SUB(frame)

def ADD(frame):
  # ... add two elements ...

def SUB(frame):
  # ... subtract two elements ...
```

Listing 19 Before applying the handler optimization in shallow tracing.

```
label(strange_sum_arr)
call(stub_DUP, p0, True)
call(stub_CALL, "length", p0, True)
call(stub_DUP2, p0, True)
i1 = call(stub_LE, p0, True)
# i1 is initialized with a default value (0)
guard_false(i1)
call(stub_DUP1, p0, True)
call(stub_CONST, p0, 1, True)
call(stub_ADD, p0, True)
call(stub_CALL, p0, "at", True)
call(stub_DUP2, p0, True)
call(stub_ADD, p0, True)
jump(strange_sum_arr)

label(L1)
call(handler_DUP0, p0, True)
call(handler_CALL, "clear", p0, True)
finish()
```

Listing 20 After applying the handler optimization in shallow tracing.

```
label(strange_sum_arr)
call(DUP, p0)
call(CALL, "length", p0)
call(DUP2, p0)
i1 = call(LE, p0)
# i1 is initialized with a default value (0)
guard_false(i1)
call(DUP1, p0)
call(CONST, p0, 1)
call(ADD, p0)
call(CALL, p0, "at")
call(DUP2, p0)
call(ADD, p0)
jump(strange_sum_arr)

label(L1)
call(handler_DUP0, p0)
call(handler_CALL, "clear", p0)
finish()
```

To integrate inline caching efficiently, we leverage the annotation mechanism of the RPython framework instead of implementing it from scratch. Inline caching is realized by instrumenting the interpreter with runtime type collection, validation, and direct call conversion. During interpretation, runtime type information – such as the receiver's type and the associated method – is collected. This collected type information is then validated at runtime, enabling the conversion of indirect calls into direct calls.

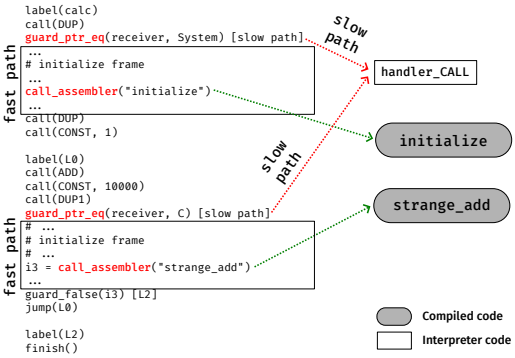


Figure 8 Overview of direct calls with inline caching.

Listing 21 Interpreter definition instrumented for inline caching

```
def threaded_interpret(frame):
  while True:
    jit_merge_point(..)
    ...
    if opcode == CALL:
      # check whether the runtime type of
      # this method equals to the
      # recorded runtime type
      method = frame.stack[frame.pc]
      if check_type(method, frame.pc):
        # if it matches, calling the
        # method with call_assembler
        call_assembler(interp(frame))
      else:
        # otherwise, call handler_CALL
        handler_CALL(frame)

def handler_CALL(frame):
  method = pop(stack)
  if not we_are_jitted():
     # during interpreting (not jitted),
     # record runtime method types at pc
     record_type(pc, method.type)
  r = threaded_interpret(frame)
  frame.push(r)
```

At runtime, our inline caching implementation works as follows:

- Fast Path: If the runtime type matches the cached type, the system directly calls the compiled method using the `call_assembler` instruction.
- Slow Path: If the type validation fails, the system falls back to the `handler_CALL` function, which dynamically resolves the method and records type information for future optimizations.

The implementation uses the RPython annotation mechanism. Two annotations were originally developed:

- `record_type`: records the runtime type of a method during interpretation.
- `check_type`: validates the runtime type against the cached type.
- `call_assembler`: performs the direct call to the compiled method.

Listing 21 illustrates how these annotations are used in the interpreter:

Based on the implementation, the compilation with direct calls with inline caching works as follows:

1. During interpretation, the `handler_CALL` function records the runtime type of the receiver using the `record_type` annotation.
2. During tracing, the `CALL` handler, `check_type` validates the runtime type of the receiver. If the validation succeeds, the `call_assembler` instruction is invoked to perform a direct call to the compiled method.
3. During compilation, `check_type` is transformed into a guard instruction that ensures the receiver type matches the recorded type. Similarly, `call_assembler` is converted directly into an assembler call instruction in the resulting trace.

6 Evaluation

In this section, we address the three questions introduced at the beginning of this paper by evaluating the performance of the two-level compilation implemented in 2SOM. We begin by describing the benchmarking programs and the execution environment used in the evaluation. Each research question is then addressed by analyzing both the warmup performance and peak performance of two-level compilation and meta-compiler–based threaded code generation using 2SOM. Since the measurement methods differ for each case, they are described in separate sections of their own.

6.1 Methodology

Throughout this evaluation, we basically use the SOM benchmarks [28], an extended version of the Are We Fast Yet benchmark [30], which includes programs such as PageRank and graph search algorithms.

To answer Q1 and Q2 in particular, we should analyze the performance of two-level JIT using real-world large-scale programs. However, SOM benchmark suite does not have such programs. Thus, we synthesize a program that has workloads similar to real-world large-scale programs.

First, we examined the relationship when methods in the DaCapo benchmark [5] were sorted in descending order based on the number of method calls. Using the least squares method to analyze this relationship, we found that the R^2 value exceeded 0.98, indicating a high correlation. Therefore, we aimed to approximate the ranking correlation between DaCapo's method call counts and the method order using the SOM benchmark suite.

We generated 20 variants of *synthesized benchmark program* using the following procedure: (1) place each program SOM's benchmark suite in order, (2) manually tune the number of internal iterations for each program so that the correlation between DaCapo's method call counts and method ranks is as close as possible. (3) Finally, from the program created in steps (1) and (2), generate the remaining 19 variants by randomly shuffling the program execution order.The details are described and discussed in Appendix A of the extended version [24].

In Q1, we measure the elapsed time obtained from the first iteration of each program and repeat this measurement 2,000 times. We calculate their medians, averages, and variances.

We evaluate the warm-up and peak performances against six execution modes:

- Interpreter-only execution: execution using 2SOM's interpreter without any form of compilation.
- Threaded code generation: meta-compiler–based threaded code generation implemented in 2SOM, offering a lightweight approach to improving execution efficiency.
- Tracing JIT: execution utilizing RPython's meta-tracing compiler.
- Tracing JIT with high threshold: A variation of the tracing compilation where the compilation threshold is increased by a factor of three. This configuration is designed to investigate the fundamental differences between a higher-threshold tracing and two-level JIT compilations.
- Two-level JIT: A combined strategy that combines both threaded code generation and tracing compilation, leveraging the strengths of both techniques for improved execution performance.
- TrufleSOM: execution using TruffleSOM, which is another SOM built with the Truffle framework [39], to compare the performance of 2SOM with a different language implementation. Note that multi-tier JIT compilation [35] is enabled.

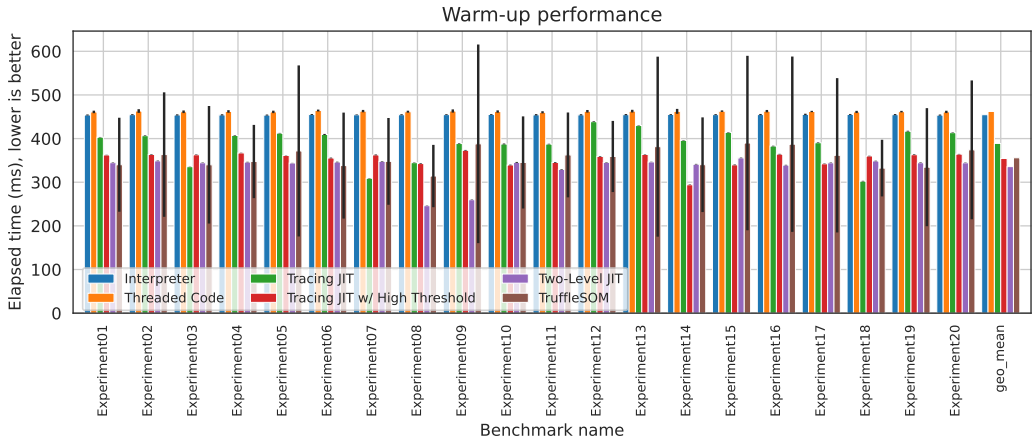


Figure 9 Warm-up performance (execution time of the first iteration).

In Q2, we evaluate the peak performance of 2SOM. We use the synthesized programs used in Q1 in addition to the SOM microbenchmark programs. Each program is executed 2,000 times, and the average elapsed times are calculated based on the last half of the iterations to omit the effect of JIT compilation. We compare the following execution models: tracing JIT, tracing JIT with a higher threshold, and two-level JIT.

In Q3 (performance of threaded code), we assess the improvements introduced by meta-compiler–based threaded code generation by utilizing the SOM microbenchmarks. Specifically, we measure the peak performance of these benchmarks to evaluate the effectiveness of meta-compiler–based threaded code generation enhancements.

The 2SOM codebase is available at Zenodo [21]. To build 2SOM, we use GCC version 14.2.1 and our customized PyPy, which is available at Zenodo [22]. For measuring time, we use `monotonic_clock` by our RPython extension [23]. For TruffleSOM, we use revision e9d8032 of the repository hosted on GitHub[2], along with GraalVM Community Edition version 23.0.2.

All evaluations are conducted on a system running Ubuntu 22.04.1 LTS, equipped with a quad-core Intel Core i7-6700 CPU and 32 GB of RAM. To ensure reliable and consistent measurements, we use ReBench [29], a benchmarking framework designed to manage the evaluation process and minimize measurement noise.

6.2 Q1: Does Two-Level Compilation Improve Warm-Up Performance?

In this section, we answer Q1 by evaluating the warm-up performance of two-level JIT in 2SOM. The results of these measurements are presented in Figure 9, which shows the elapsed times with error bars displaying their variances.

As shown in Figure 9, two-level compilation demonstrated the best warm-up performance; it executed approximately 15% faster than tracing JIT alone. Furthermore, even when compared to tracing compilation with a threshold increased by a factor of three and TruffleSOM, two-level compilation outperformed by approximately 5%.

[2] `https://github.com/SOM-st/TruffleSOM`

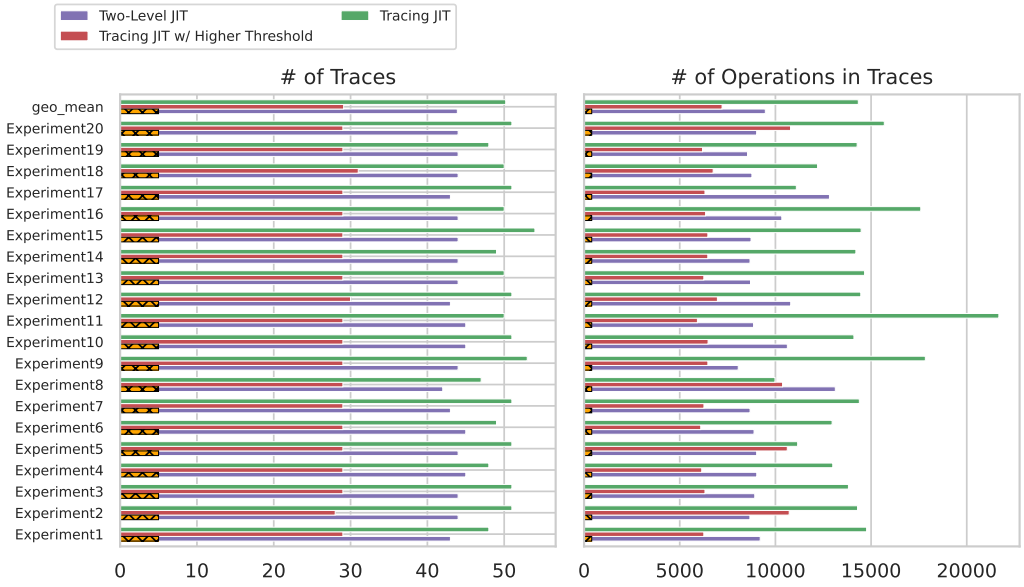


Figure 10 Number of operations and traces in the following strategies: two-level JIT, tracing JIT with high threshold, and tracing JIT in the evaluation of warm-up performance. The orange stitched bar inside the two-level JIT shows the number of operations and traces generated from the threaded code generation.

To statistically validate whether there were differences in the data for each pair – two-level and tracing JITs, as well as two-level JIT and tracing JIT with a high threshold – we conducted the Wilcoxon signed-rank test. The results revealed that for the two-level and tracing JITs pair, $p = 0.0314\%$, and for the two-level JIT and tracing JIT with a higher threshold pair, $p = 2.151\%$. In both cases, the null hypothesis was rejected.

These findings indicate that two-level JIT improves warm-up performance compared to RPython's single-tier compilation. Furthermore, they suggest that two-level JIT enhances warm-up performance more effectively than optimizing compilation strategies by limiting the compilation target scope, such as adjusting thresholds to reduce compilation time.

In addition, we break down the traces generated from these strategies: two-level JIT, tracing JIT with high threshold, and tracing JIT. The results are shown in Figure 10.

The first breakdown (the left-hand side of Figure 10: number of traces) shows that two-level JIT generates fewer traces than tracing JIT but more than tracing JIT with a high threshold.

The second breakdown (the right-hand side of Figure 10: number of operations in traces) shows that two-level JIT generates fewer operations in traces than tracing JIT but slightly more than tracing JIT with a high threshold. In addition, the ratio of traces generated from threaded code overall number of traces are approximately 10%. These results stem from the characteristic of the lightweight compiler that it generates compact traces during early execution phases.

These results demonstrate the ability of the two-level JIT to balance warm-up speed and trace generation; lightweight compilation using threaded code can improve early performance. Therefore, two-level JIT effectively works interpreter execution, threaded code generation, and tracing together.

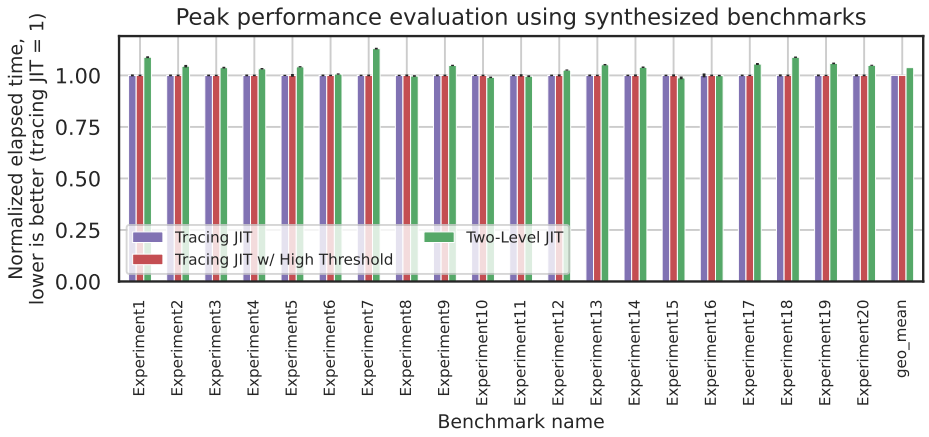


Figure 11 Peak performance results: Normalized elapsed time (lower is better) across synthesized programs for different JIT compilation strategies: (1) Tracing JIT, (2) Tracing JIT with a high threshold, and (3) Two-Level JIT. Results are normalized to the baseline Tracing JIT.

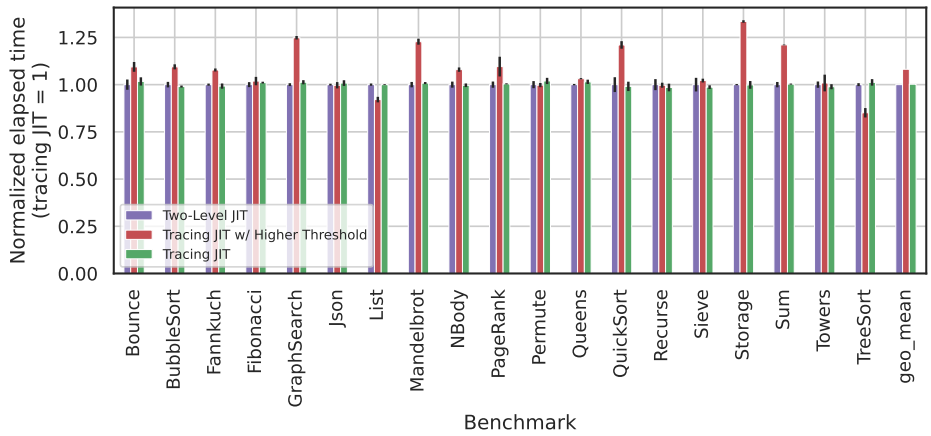


Figure 12 Peak performance results: Normalized elapsed time (lower is better) for SOM benchmarks, comparing three JIT compilation strategies: (1) Two-Level JIT, (2) Tracing JIT with a higher threshold, and (3) Tracing JIT. The results are normalized to the baseline Tracing JIT.

6.3 Q2: How Good is the Peak Performance of Two-Level Compilation?

In this section, we answer Q2 by describing the peak performance of two-level JIT in 2SOM. We measure the peak performance of synthesized programs and the SOM microbenchmark programs, and these results are shown in Figures 11 and 12.

As shown in Figure 11, the peak performance of two-level JIT is slower than approximately 3% against tracing JIT, and approximately 5% slower than tracing JIT with high threshold. In addition, as shown in Figure 12, the peak performance of two-level JIT is slower than approximately 7% against tracing compilation and approximately 10% slower than tracing JIT with high threshold.

This result indicates that introducing the meta-compiler–based threaded code generation to a meta-tracing JIT compiler does not significantly reduce the peak performance of meta-tracing JIT compilation. In particular, the introduction of interpreter switching does not result in a significant performance degradation in peak performance.

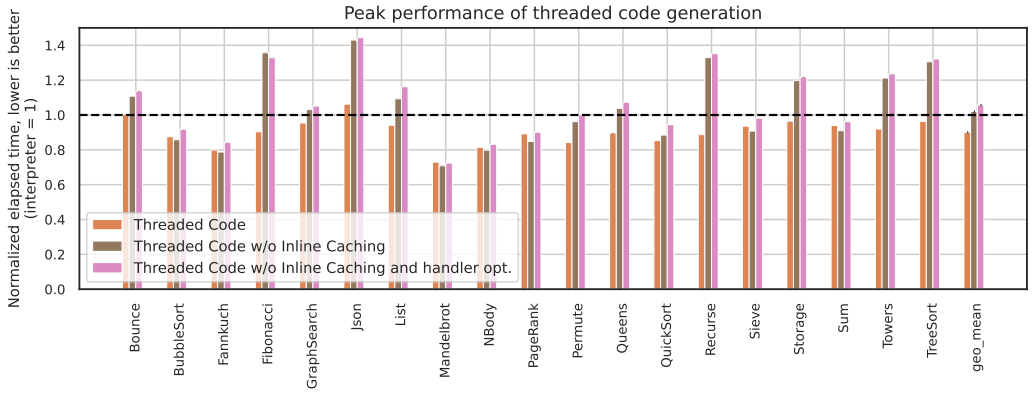


Figure 13 Comparison of peak performance for threaded code generation under different optimization configurations: (1) all optimizations enabled, (2) without inline caching, and (3) without both inline caching and shallow tracing handler optimization. Each data is normalized to interpreter-only execution.

6.4 Q3: How Much is the Improvement of Threaded Code Generation?

In this section, we answer Q3 by describing the peak performance of the meta-compiler–based threaded code generation. To evaluate the impact of the optimizations implemented in threaded code generation, we measured its peak performance. Specifically, we compare three configurations: one with all optimization options enabled, one with only the direct calls with inline caching optimization disabled, and another with both direct calls with inline caching and shallow tracing handler optimization disabled. This comparison allowed us to assess the performance improvements contributed by each optimization individually. The result is shown in Figure 13.

As shown in Figure 13, the fully optimized configuration (Threaded Code, orange-colored bar) consistently achieves the best performance across all benchmarks, running about 10% faster than the interpreter. Disabling direct calls with inline caching (brown-colored bar) degrades performance by about 10% on average, highlighting its importance in optimizing frequent function calls and dynamic dispatch. Disabling both direct calls with inline caching and shallow tracing handler optimization (pink-colored bar) further degrades performance by about 5%, highlighting their complementary impact on efficiency. Overall, the results underscore the need for these optimizations to achieve optimal runtime performance.

This result shows that our implemented optimization improves the performance of the meta-compiler–based threaded code generation. In particular, direct calls with inline caching significantly improve the performance in function call-intensive programs such as Fibonacci, Json, Recurse, Storage, Towers, and TreeSort. However, further optimization is needed to improve the warm-up performance more. For example, we believe that inline caching implemented this time can be optimized more efficiently by using polymorphic inline caching.

6.5 Threats to Validity

Since this evaluation was conducted on a subset of Smalltalk, the results may differ when it is performed on a larger language implementation such as PyPy and RSqueak [15].

7 Related Work

There exist several approaches to building a lightweight compiler in a meta-compiler framework. Copy-and-patch compilation [40] is a meta-compilation technique that performs template-based compilation. The template is generated from a bytecode instruction or an AST node of a high-level language by the MetaVar compiler. Although they build a new meta-compiler to generate a new baseline compiler, our approach is built on an existing meta-tracing compiler. In addition, although they use a parameterized binary code for generating a template, we use an interpreter definition with annotations to compile a source program.

There are also other methodologies for realizing a multi-tier JIT compilation VM using a meta-compiler framework. Graal/Truffle [38] has introduced a first-tier execution in addition to the second-tier execution that performs heavyweight compilation [35]. The approach is straightforward: they control the level of optimization in their meta-compilation system to enable multi-tier execution. While their methodology requires major modifications to the meta-compiler, our methodology is adding multiple tiers with different compilation methods in RPython.

Furthermore, other meta-circular VMs provide multi-tier JIT compilation, such as Jikes RVM [1] and Maxine VM [37]. These two VMs have lightweight and heavyweight compilers. In particular, they have original code generators for lightweight compilation: Jikes RVM's lightweight compiler translates Java bytecode into native code by simulating Java's operand stack. Maxine VM's lightweight compiler translates Java functions into native code using templates. In contrast, our lightweight compilation reuses the compiler of heavyweight compilation. To control the code quality and compilation time, we slightly modify the definition of an interpreter provided to the meta-tracing compiler.

8 Conclusion and Future Work

This paper introduces a lightweight methodology that enables a meta-compiler framework to generate a multi-tier JIT compilation VM. By treating interpreter definitions as specifications not only for language semantics but also for compilation strategies, we enable RPython to generate key components for multi-tier JIT compilation: multiple compilers, a profiler, and a transition mechanism.

To demonstrate our approach, we created a two-tier JIT compilation version of the Simple Object Machine called 2SOM. 2SOM incorporates two JIT compilers: a tier-1 threaded code generator and a tier-2 tracing JIT compiler. This integration was achieved by addressing practical challenges in combining meta-compiler-based threaded code generation [25] with RPython.

Our evaluation shows that the proposed methodology significantly improves performance. Warm-up performance was improved by approximately 15%, while peak performance degradation was limited to approximately 5% compared to tracing JIT-only systems. Furthermore, enhancements to the threaded code generator improved execution speed by 10% compared to interpreter-based execution. These results demonstrated the potential of our methodology to simplify the development of multi-tier JIT compilers in other RPython-based language implementations.

Despite its advantages, our approach has certain limitations. First, it cannot add a tier that requires different low-level code generation and optimization mechanisms (e.g., register allocation or instruction-level scheduling) because it does not change the processes of RPython after generated traces. Second, it cannot add a tier that requires manipulating intermediate

code. This limitation arises because the approach relies on running the interpreter to obtain traces of the intermediate code (RPython traces). The design does not assume modifications to these traces.

Future work is not only to solve these limitations, but also to apply our method to larger language systems than 2SOM and validate the effectiveness of multi-tier JIT compilation VMs using more realistic data.

One potential application is PyPy. Although we have conducted experiments using synthesized programs to evaluate the effectiveness of multi-tier JIT compilation VMs. However, we think that using a larger benchmark set would enable more detailed analysis. To achieve this in PyPy, two specific challenges must be addressed: a) How to reduce the effort required to annotate PyPy's interpreter, which consists of approximately 200,000 lines of code. b) How to handle cases where handlers in the PyPy interpreter may throw exceptions, given that our current tier-1 compiler does not support tracing handlers that can raise RPython-level exceptions. c) How to develop a lightweight compiler that can generate higher-quality code than threaded code generation, enabling the multi-tier JIT compilation VM to be effective for a broader range of workloads.

For challenge a), it will be necessary to allow at least two interpreters to be generated from a single interpreter. One idea is to improve PyPy's translation flow, which generates VMs, to automate the generation of interpreters.

For challenge b), we believe that adopting the *zero-cost exception mechanism* [8] introduced in Python 3.11–now being integrated into PyPy–will allow us to trace handlers in the PyPy interpreter. Unlike earlier versions of Python where try statements compiled to `SETUP_FINALLY` (incurring overhead even without exceptions), Python 3.11 uses `NOP` instructions for the non-exception path. This change should let our threaded code generation avoid triggering RPython-level exceptions and work smoothly in PyPy with this mechanism enabled.

For challenge c), we believe that a solution can be achieved through further refinement of meta-compiler–based threaded code generation. In general, interpreters implement the behavior of handler functions by calling various helper functions within those handlers. The current threaded code generator only leaves call instructions to the handler functions in the trace, but more efficient lightweight compilation could be achieved by inlining non-critical helper functions and leaving only those helper functions that are time-consuming to compile as call instructions.

References

1 Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2000, pages 47–65. Association for Computing Machinery, 2000. doi:10.1145/353171.353175.

2 Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000. doi:10.1145/349299.349303.

3 Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. SPUR: A Trace-based JIT Compiler for CIL. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA 2010, pages 708–725. Association for Computing Machinery, 2010. doi:10.1145/1869459.1869517.

4 James R. Bell. Threaded code. *Commun. ACM*, 16(6):370–372, 1973. doi:10.1145/362248.362270.

5 Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA 2006, pages 169–190. Association for Computing Machinery, 2006. doi:10.1145/1167473.1167488.

6 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS 2009, pages 18–25, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1565824.1565827.

7 Carl Friedrich Bolz and Laurence Tratt. The Impact of Meta-tracing on VM Design and Implementation. *Science of Computer Programming*, 98:408–421, 2015. Special Issue on Advances in Dynamic Languages. doi:10.1016/j.scico.2013.02.001.

8 CPython. Zero cost exception handling. URL: https://github.com/python/cpython/blob/main/InternalDocs/exception_handling.md.

9 Charles Curley. Life in the FastForth Lane. *Forth Dimensions*, 14(4):6–12, 1993. URL: https://www.forth.org/fd/curley2.html.

10 Charles Curley. Optimizing in a BSR/JSR Threaded Forth. *Forth Dimensions*, 14(5):21–26, 1993.

11 L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. Association for Computing Machinery. doi:10.1145/800017.800542.

12 Robert B.K. Dewar. Indirect Threaded Code. *Communications of the ACM*, 18(6):330–331, June 1975. doi:10.1145/360825.360849.

13 Eclipse Foundation. Open J9: Unleash the power of java, 2017. URL: https://www.eclipse.org/openj9/.

14 M. Anton Ertl and David Gregg. The Structure and Performance of Efficient Interpreters. *Journal of Instruction-level Parallelism*, 5, November 2003. URL: http://www.jilp.org/vol5/v5paper12.pdf.

15 Tim Felgentreff, Tobias Pape, Patrick Rein, and Robert Hirschfeld. How to Build a High-Performance VM for Squeak/Smalltalk in Your Spare Time: An Experience Report of Using the RPython Toolchain. In *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies*, IWST '16, pages 21:1–21:10. Association for Computing Machinery, 2016. RSqueak,RPython. doi:10.1145/2991041.2991062.

16 Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-Based Just-in-Time Type Specialization for Dynamic Languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2009, pages 465–478. Association for Computing Machinery, 2009. doi:10.1145/1542476.1542528.

17 Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: An Effective JIT Compiler for Resource-Constrained Devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, pages 144–153. Association for Computing Machinery, 2006. doi:10.1145/1134760.1134780.

18 Google. Google's High-performance Open Source JavaScript and WebAssembly Engine, 2015. URL: https://v8.dev/.

19 Michael Haupt, Robert Hirschfeld, Tobias Pape, Gregor Gabrysiak, Stefan Marr, Arne Bergmann, Arvid Heise, Matthias Kleine, and Robert Krahn. The som family: Virtual machines for teaching and research. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE 2010, pages 18–22. Association for Computing Machinery, 2010. doi:10.1145/1822090.1822098.

20 Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In Pierre America, editor, *ECOOP'91 European Conference on Object-Oriented Programming*, pages 21–38. Springer Berlin Heidelberg, 1991. doi:10.1007/BFB0057013.

21 Yusuke Izawa. 2SOM: A Two-Level Simple Object Machine, April 2025. Version 1.0. doi:10.5281/zenodo.15286979.

22 Yusuke Izawa. Customized PyPy for Threaded Code Generation, April 2025. Version 1.0. doi:10.5281/zenodo.15287001.

23 Yusuke Izawa. RPython Extension that Enables a Monotonic Clock, April 2025. Version 1.0. doi:10.5281/zenodo.15286954.

24 Yusuke Izawa, Hidehiko Masuhara, and Carl Friedrich Bolz-Tereick. A Lightweight Method for Generating Multi-Tier JIT Compilation Virtual Machine in a Meta-Tracing JIT Compiler Framework (Extended Version). arXiv, 2025. doi:10.48550/arXiv.2504.17460.

25 Yusuke Izawa, Hidehiko Masuhara, Carl Friedrich Bolz-Tereick, and Youyou Cong. Threaded Code Generation with a Meta-Tracing JIT Compiler. *Journal of Object Technology*, 21(2):a1, 2022. doi:10.5381/jot.2022.21.2.a1.

26 Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™ Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1), 2008-05. doi:10.1145/1369396.1370017.

27 Stefan Marr. PySOM: A Simple Object Machime Smalltalk implemented in Python, 2013. URL: https://github.com/smarr/PySOM.

28 Stefan Marr. Som Benchmarks, 2013. URL: https://github.com/SOM-st/SOM/tree/9c04914f800dc3ccbfaa1dc8fcf78cc5714549a4/Examples/Benchmarks.

29 Stefan Marr. ReBench: Execute and Document Benchmarks Reproducibly, August 2018. Version 1.0. doi:10.5281/zenodo.1311762.

30 Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-language compiler benchmarking: Are we fast yet? In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, pages 120–131. Association for Computing Machinery, 2016. doi:10.1145/2989225.2989232.

31 Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. Graalsqueak: Toward a smalltalk-based tooling platform for polyglot programming. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2019, pages 14–26. ACM, 2019. doi:10.1145/3357390.3361024.

32 Michael Paleczny, Christopher Vick, and Cliff Click. The Java Hotspot™ Server Compiler. In *Proceedings of the 2001 Symposium on Java™ Virtual Machine Research and Technology Symposium - Volume 1*, JVM 2001, page 1. USENIX Association, 2001.

33 Tobias Pape, Tim Felgentreff, Robert Hirschfeld, Anton Gulenko, and Carl Friedrich Bolz. Language-independent storage strategies for tracing-jit-based virtual machines. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, pages 104–113, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2816707.2816716.

34 Fillip Pizlo. Speculation in JavaScriptCore, 2020. URL: https://webkit.org/blog/10308/speculation-in-javascriptcore/.

35 Aleksandar Prokopec. Multi-Tier Compilation in GraalVM, 2021. URL: https://medium.com/graalvm/multi-tier-compilation-in-graalvm-5fbc65f92402.

36 Armin Rigo and Samuele Pedroni. PyPy's Approach to Virtual Machine Construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA 2006, pages 944–953. Association for Computing Machinery, 2006. doi:10.1145/1176617.1176753.

37 Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.*, 9(4), January 2013. doi:10.1145/2400682.2400689.

38 Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 662–676. Association for Computing Machinery, 2017. doi:10.1145/3062341.3062381.

39 Thomas Würthinger, Andreas Wöundefined, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS 2012, pages 73–82. Association for Computing Machinery, 2012. doi:10.1145/2384577.2384587.

40 Haoran Xu and Fredrik Kjolstad. Copy-and-Patch Compilation: A Fast Compilation Algorithm for High-Level Languages and Bytecode. *Proc. ACM Program. Lang.*, 5(OOPSLA), 2021. doi:10.1145/3485513.

Contrasting Deadlock-Free Session Processes

Juan C. Jaramillo
Unversity of Groningen, The Netherlands

Jorge A. Pérez
Unversity of Groningen, The Netherlands

Abstract

Deadlock freedom is a crucial property for message-passing programs. Over the years, several different type systems for concurrent processes that ensure deadlock freedom have been proposed; this diversity raises the question of how they compare. We address this question, considering two type systems not covered in prior work: Kokke *et al.*'s HCP, a type system based on a linear logic with *hypersequents*, and Padovani's *priority-based* type system for asynchronous processes, dubbed P. Their distinctive features make formal comparisons relevant and challenging. Our findings are two-fold: (1) the hypersequent setting does not drastically change the class of deadlock-free processes induced by linear logic, and (2) we relate the classes of deadlock-free processes induced by HCP and P. We prove that our results hold under both synchronous and asynchronous communication. Our results provide new insights into the essential mechanisms involved in statically avoiding deadlocks in concurrency.

2012 ACM Subject Classification Theory of computation → Type structures; Theory of computation → Program analysis; Theory of computation → Process calculi

Keywords and phrases session types, process calculi, deadlock freedom

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.17

Related Version *Full Version*: `https://arxiv.org/abs/2504.15845` [19]

Funding *Juan C. Jaramillo*: Ministry of Science of Colombia (Minciencias).

Acknowledgements We are grateful to Channa Dias Perera for initial discussions on the topics of this paper. We also thank Andreia Mordido and the anonymous reviewers for their helpful suggestions.

1 Introduction

The *deadlock freedom* property (DF, in the following) is a crucial guarantee for message-passing programs. Informally, DF ensures that processes "never get stuck" – an essential requirement in concurrent and distributed software systems, where even a single component that becomes permanently blocked while waiting for a message can compromise reliability. As illustration, consider a toy concurrent program in the language of Gay and Vasconcelos [14]:

$$\langle \mathsf{let}\,(x_1, z) = \mathsf{receive}\,x_1\ \mathsf{in}\ \mathsf{send}\ \mathtt{42}\,y_1\rangle \parallel \langle \mathsf{let}\,(y_2, k) = \mathsf{receive}\,y_2\ \mathsf{in}\ \mathsf{send}\ \text{`}\mathtt{hello}\text{'}\,x_2\rangle$$

We have two concurrent threads, each running an expression with two actions in sequence. This program is evaluated under a context that binds together x_1 and x_2 and declares them as endpoints of the same linear channel (same for y_1 and y_2); we omit it for readability. Given a linear channel x, "$\mathsf{send}\ \mathtt{v}\,x$" sends value $\mathtt{v}$ along x and returns the continuation of x, whereas "$\mathsf{receive}\,x$" returns a pair with the continuation of x and a received value. The threads are thus engaged in separate but intertwined protocols (one along x_i, the other along y_i). Each thread should receive on a channel and then send a value on the other. However, the output matching the input in one thread is blocked in the other – the program is stuck. The key challenge in enforcing DF is thus performing the non-local analysis needed to avoid *circular dependencies* between threads, which are often less apparent than in this program.

© Juan C. Jaramillo and Jorge A. Pérez;
licensed under Creative Commons License CC-BY 4.0

An expressive model for concurrent processes, the π-calculus offers a convenient basis for developing analysis techniques for message-passing programs in different paradigms [32]. Several type systems that enforce (forms of) DF have been proposed over the years (see, e.g., [21, 11, 5, 27, 7, 34, 1]). They avoid circular dependencies in processes using different insights, which raises the question of how they compare. In this paper, we rigorously compare some representative type systems for DF to better understand their underlying mechanisms.

A key motivation for studying type systems for concurrent processes is their suitability for developing advanced analysis techniques at an appropriate level of abstraction – these typed frameworks have been directly incorporated into languages such as Scala [33], OCaml [28, 17], and Rust [24]. Also, concurrent processes offer an adequate representation level for verifying liveness properties in real Go programs [25, 26]. Moreover, type systems for the π-calculus are the basis for static analyzers (such as Kobayashi's TyPiCal [20]) and can be ported to enforce DF for higher-order concurrent programs [29]. Type systems based on the Curry-Howard correspondence between session types and linear logic ("propositions-as-sessions" [3, 37]) enforce DF; they entail topological invariants on processes, key to ensure deadlock and memory leak freedom in concurrent programs [18]. Further understanding the essential mechanisms that enforce DF in logic-based session types is a central topic in our work.

Prior work by Dardha and Pérez [9] contrasts type systems for DF by relating the classes of processes they induce. They compare a *priority-based* type system by Kobayashi [21] and a type system derived from "propositions-as-sessions". Writing $\mathbb{K}$ and $\mathbb{L}$ to denote the respective classes of processes, a result in [9] is that $\mathbb{L} \subsetneq \mathbb{K}$: there exist deadlock-free processes that cannot be typed by logic-based type systems. The class $\mathbb{K} \setminus \mathbb{L}$ includes, for instance, process networks organized in circular topologies (e.g., the dinning philosophers). It also includes process P_0 below, in which two processes interact along independent protocols (*sessions*):

$$P_0 \triangleq (\nu\, x_1x_2)(\nu\, y_1y_2)(x_1(z).\overline{y_1}\langle \texttt{42}\rangle.\mathbf{0} \mid \overline{x_2}\langle \texttt{hello}\rangle.y_2(k).\mathbf{0})$$

Above, "$(\nu\, x_1x_2)$" declares a new session with linear endpoints x_1 and x_2, and "$x(y).Q$" (resp. "$\overline{x}\langle v\rangle.Q$") denotes an input (resp. output of value v) along x with continuation Q. P_0 is the deadlock-free variant of the program given above, now neatly specified as a process. In $\mathbb{L}$, processes can share at most one session in parallel, whereas processes in P_0 share two: the session "x_1x_2" for exchanging "`hello`" and the session "y_1y_2" for exchanging "`42`". In [9], both compared classes rely on a reduction semantics; accordingly, DF concerns the absence of stable states with *pending reduction steps* on linear names/channels.

Here we extend the work in [9] by considering type systems not studied there: (i) Kokke *et al.*'s HCP [23] and (ii) Padovani's priority-based type system [27], here dubbed P. Their distinctive features make formal comparisons relevant and challenging, as we discuss next.

Derived from "propositions-as-sessions", HCP is based on a presentation of classical linear logic (CLL) with *hypersequents*. HCP has been studied in several works, which have deepened into its theory and applications [22, 12, 31]. In HCP, the semantics of processes is given by a Labeled Transition System (LTS), rather than by a reduction semantics. This unlocks the use of well-established analysis techniques, notably observational equivalences, to reason about typed processes. The (typed) LTS for HCP includes the usual rules for observing actions and synchronizations, here dubbed *regular transitions*, but also rules that allow for *delayed actions* (i.e., observable actions due to prefixes not at top-level) and *self-synchronizations* (i.e., internal actions that do not require the interaction of two parallel processes). Remarkably, the bisimilarity induced by the LTS coincides, in a fully-abstract way, with a *denotational equivalence* over processes – this gives a strong justification for the design of the LTS. Typing ensures *progress*: well-typed processes not equivalent to $\mathbf{0}$ always have a transition step.

An intriguing observation is that there are HCP processes that are stuck in a reduction-based setting but have synchronizations under HCP's LTS. We discuss two interesting examples, using "$x().Q$" and "$\overline{x}\langle\rangle.Q$" to denote empty inputs and outputs, respectively:

- Process $P_1 \triangleq (\nu\, xx')\overline{x'}\langle\rangle.x().\mathbf{0}$ is self-synchronizing: it does not have any reductions, whereas $P_1 \xrightarrow{\tau} \mathbf{0}$, as in HCP synchronizations do not need a parallel context to be enabled.
- Process $P_2 \triangleq (\nu\, xx')(\nu\, yy')(\overline{y}\langle\rangle.x().\mathbf{0} \mid \overline{x'}\langle\rangle.y'().\mathbf{0})$ is arguably the paradigmatic example of a deadlock, with two independent sessions unable to reduce. In HCP, thanks to delayed actions, we have both $P_2 \xrightarrow{\tau} (\nu\, yy')(\overline{y}\langle\rangle.\mathbf{0} \mid y'().\mathbf{0})$ and $P_2 \xrightarrow{\tau} (\nu\, xx')(\overline{x}\langle\rangle.\mathbf{0} \mid x'().\mathbf{0})$.

This observation shows that comparisons such as [9] should accommodate other semantics for processes, beyond reductions, which entail other definitions of DF and induce new deadlock-free processes (such as P_1 and P_2). This begs a first question:

(Q1) What is the status of delayed actions and self-synchronizations, as typable in HCP, with respect to DF? Do they induce a distinct class of deadlock-free processes?

Considering Padovani's P means addressing the case of processes with *asynchronous communication*, widely used in real programs. In the π-calculus, asynchronous communication is obtained by defining outputs as non-blocking actions; only inputs can block the execution of a process [2, 16]. Asynchrony and DF are then closely related: because an asynchronous setting involves the least amount of blocking operations, it offers the most elementary scenario for investigating DF [35]. Now, the comparisons in [9] are limited to synchronous processes; we then observe that they should be extended to address asynchronous communication, too. Let us write $\mathbb{H}$ and $\mathbb{P}$ to denote the class of processes induced by HCP and P, respectively, and $\mathbb{C}$ to denote the class of processes induced by CP (a fragment of HCP without hypersequents and with a restricted form of parallel composition). We consider the second question:

(Q2) What is the status of $\mathbb{H}$ and $\mathbb{C}$ with respect to asynchronous processes in $\mathbb{P}$?

Interestingly, our two questions are intertwined: HCP and P are related due to asynchrony. Our insight is that delayed actions in HCP already implement an asynchronous semantics. To see this, consider the process $P_3 \triangleq (\nu\, xx')(\nu\, yy')(\overline{y}\langle\rangle.\overline{x}\langle\rangle.\mathbf{0} \mid x'().y'().\mathbf{0})$, which is not stuck: the output on y is not really blocking, as delayed actions allow to anticipate the output (and synchronization) on x. In other words, P_3 is morally the same as $(\nu\, xx')(\nu\, yy')(\overline{y}\langle\rangle.\mathbf{0} \mid \overline{x}\langle\rangle.\mathbf{0} \mid x'().y'().\mathbf{0})$. We thus conclude that the (typed) operational semantics for HCP admits many synchronous-looking processes (such as P_3) that actually behave in an asynchronous way; this naturally invites the question of how $\mathbb{H}$ relates to $\mathbb{P}$.

Contributions and Outline. This paper offers conceptual and technical contributions. From a conceptual view, we reflect upon the significance of formally comparing different type systems for concurrency, while uniformly presenting various notions in the literature. On the technical side, we address **(Q1)** and **(Q2)**, thus extending and complementing [9]. A key discovery is identifying *commuting conversions* and *disentanglement* as key notions for characterizing deadlock-free processes in HCP – as we will see, these notions define sound *process optimizations* that respect causality and increase parallelism. As our results hold also in an asynchronous setting, not treated in prior work, our findings are most related to abstractions and mechanisms used in practical concurrent programming. More in detail:

§ 2 We introduce SP, a session π-calculus that we use as baseline for comparisons, as well as CP and HCP, the type systems resulting from the Curry-Howard correspondence with CLL and CLL with hypersequents, respectively.

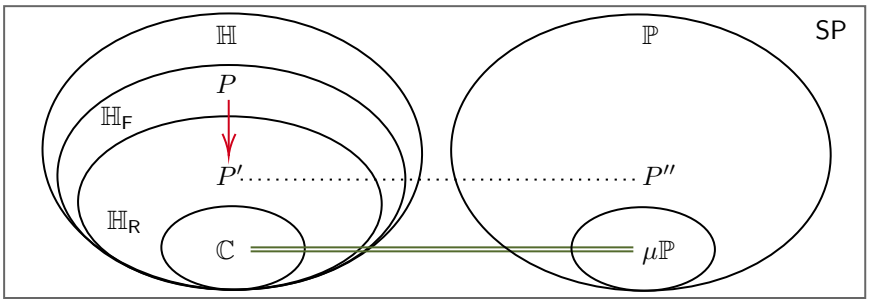


Figure 1 Overview of main results. Above, $\mathbb{H}_\mathsf{F}$, $\mathbb{H}_\mathsf{R}$, $\mathbb{P}$, $\mathbb{C}$, and $\mu\mathbb{P}$ stand for classes of deadlock-free SP processes. The red arrow represents optimizations (commuting conversions and disentanglement) that relate $\mathbb{H}_\mathsf{F}$ and $\mathbb{H}_\mathsf{R}$. The green lines denote the correspondence $\mathbb{C} = \mu\mathbb{P}$. The dotted line means that P' and P'' are the representatives of P in $\mathbb{H}_\mathsf{R}$ and $\mathbb{P}$, up to an encoding.

§ 3 We define $\mathbb{C}$ and $\mathbb{H}$: the classes of deadlock-free SP processes induced by CP and HCP, respectively. We identify two relevant sub-classes of $\mathbb{H}$:
$\mathbb{H}_\mathsf{F}$**:** Processes that use the Full LTS.
$\mathbb{H}_\mathsf{R}$**:** Processes that *only* use Regular transitions (no delayed actions/self-synchronizations).
In our main result (Corollary 53), we prove that for every process $P \in \mathbb{H}_\mathsf{F}$, there exists an observationally equivalent process $P' \in \mathbb{H}_\mathsf{R}$. Enabled by process optimizations induced via commuting conversions and disentanglement, this result shows that delayed actions and self-synchronizations are *inessential* when it comes to DF, addressing **(Q1)**.

§ 4 To address **(Q2)**, we define $\mathbb{P}$: the class of deadlock free SP processes induced by P, and its sub-class $\mu\mathbb{P}$ ("micro $\mathbb{P}$"), which is inspired by CP. We prove that $\mathbb{C} = \mu\mathbb{P}$ (Corollary 74), thus casting Dardha and Pérez's key finding into the asynchronous setting. Our main results are: Theorem 76, which precisely relates the classes $\mathbb{H}$, $\mathbb{H}_\mathsf{F}$, $\mathbb{H}_\mathsf{R}$, and $\mathbb{P}$; and Corollary 77, which strengthens Corollary 53 by considering DF in $\mathbb{P}$. Moreover, we introduce SP^a, an asynchronous variant of SP, and its corresponding classes of processes. We show how to transfer Theorem 76 and Corollary 77 from SP to SP^a (Corollary 88).

Figure 1 offers a graphical description of the classes of processes and some of our main results (Corollaries 53 and 74 and Theorem 76). For simplicity, the figure does not depict several encodings needed to bridge syntactic differences between the various classes of processes / type systems under consideration; Figure 12 (Page 26) presents a detailed version. In § 5 we collect some closing remarks. The extended version of the paper [19] contains omitted proofs.

2 Session Processes and Their Type Systems

We present SP, our reference language of session processes. As observed by Dardha and Pérez [9], this is a convenient framework for formal comparisons, because typing in SP ensures communication safety but not DF. We also present CP and HCP, the two typed languages based on "propositions-as-sessions", and recall useful associated results.

2.1 Session Processes (SP)

SP is a core session-typed π-calculus, based on the language defined by Vasconcelos [36]. We assume a base set of *variables*, ranged over by $x, y, \ldots$, which denote *channels* (or *names*): they can be seen as *endpoints* on which interaction takes place. Processes interact to exchange values $v, v', \ldots$; for simplicity, values correspond to variables, i.e., $v ::= x$. We write $\widetilde{x}$ to denote a finite sequence of variables. The syntax of *processes* is as follows:

$$P \quad ::= \quad \mathbf{0} \mid \overline{x}\langle v\rangle.P \mid x(y).P \mid P_1 \mid P_2 \mid (\nu\, xy)P \mid \overline{x}\langle\,\rangle.P \mid x().P$$

$$(\nu\, xy)(\overline{x}\langle v\rangle.P \mid y(z).Q \mid S) \longrightarrow (\nu\, xy)(P \mid Q\{^{v}/_{z}\} \mid S) \qquad \text{R-Com}$$

$$(\nu\, xy)(\overline{x}\langle\,\rangle.P \mid y().Q) \longrightarrow P \mid Q \qquad \text{R-EmptyCom}$$

$$\text{R-Par}\ \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \qquad \text{R-Res}\ \frac{P \longrightarrow P'}{(\nu\, xy)P \longrightarrow (\nu\, xy)P'} \qquad \text{R-Struct}\ \frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}$$

$$P \mid Q \equiv Q \mid P \qquad P \mid \mathbf{0} \equiv P$$

$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad (\nu\, xy)P \equiv (\nu\, yx)P$$

$$(\nu\, xy)P \mid Q \equiv (\nu\, xy)(P \mid Q) \ \text{ if } x, y \notin \mathtt{fv}(Q) \qquad (\nu\, xy)(\nu\, wz)P \equiv (\nu\, wz)(\nu\, xy)P$$

Figure 2 SP: Reduction semantics (top) and structural congruence (bottom).

Process $\mathbf{0}$ denotes inaction. Process $\overline{x}\langle v\rangle.P$ sends v along x and continues as P. Process $x(y).P$ receives v along x and continues as $P\{^{v}/_{y}\}$, i.e., the process resulting from the capture-avoiding substitution of y by v in P. Process $P_1 \mid P_2$ denotes the parallel execution of P_1 and P_2. Process $(\nu\, xy)P$ declares x and y as *co-variables*, restricting their scope to P. That is, restriction declares co-variables as dual ends of a channel. Finally, $\overline{x}\langle\,\rangle.P$ and $x().P$ denote empty output and input (close and wait), which represent the closing of session x.

As usual, $x(y).P$ binds y in P and $(\nu\, xy)P$ binds x, y in P. The sets of free and bound variables of P, denoted $\mathtt{fv}(P)$ and $\mathtt{bv}(P)$, are defined accordingly. The semantics of processes is given using a reduction relation, defined next.

▶ **Definition 1** (SP: Reduction)**.** Reduction *for* SP*, denoted* $\longrightarrow$*, is defined by the rules in Figure 2 (top). It relies on* structural congruence*, denoted* $\equiv$*, the smallest congruence on processes that satisfies the axioms in Figure 2 (bottom).*

We now define session types, which express protocols associated to channels:

▶ **Definition 2** (SP: Session Types and Typing Environments)**.** *The syntax of session types and typing environments is inductively defined as follows:*

$$T, S \ ::= \ ?S.T \mid\ !S.T \mid \mathsf{end}_? \mid \mathsf{end}_! \qquad \Gamma \ ::= \ \emptyset \mid \Gamma, x : T$$

The type $!S.T$ (resp. $?S.T$) is assigned to a channel able to send (resp. receive) a value of type S and to continue as T. For convenience, we have two different types for channels with terminated protocols, denoted $\mathsf{end}_?$ and $\mathsf{end}_!$. Duality on types ensures that the endpoints of a channel agree on the protocol they describe:

▶ **Definition 3** (SP: Duality)**.** *The dual of* S*, denoted* $\overline{S}$*, is defined as follows:*

$$?S.T \triangleq\ !S.\overline{T} \qquad \overline{!S.T} \triangleq ?S.\overline{T} \qquad \overline{\mathsf{end}_?} \triangleq \mathsf{end}_! \qquad \overline{\mathsf{end}_!} \triangleq \mathsf{end}_?$$

▶ **Definition 4** (SP: Typing Judgment and Typing Rules)**.** *The judgment* $\Gamma \vdash_{\mathrm{S}} P$ *describes a well-typed process under the typing environment* Γ*. The typing rules for* SP *are shown in Figure 3. We shall write* $P \in \mathsf{SP}$ *if* $\Gamma \vdash_{\mathrm{S}} P$*, for some* Γ*.*

We now state the properties of well-typed SP processes, whose proofs follow [36]:

$$\textsc{Inact}\ \frac{}{\cdot \vdash_{\mathrm{S}} \mathbf{0}} \qquad \textsc{EInput}\ \frac{\Gamma \vdash_{\mathrm{S}} P}{\Gamma, x : \mathsf{end}_? \vdash_{\mathrm{S}} x().P} \qquad \textsc{EOutput}\ \frac{\Gamma \vdash_{\mathrm{S}} P}{\Gamma, x : \mathsf{end}_! \vdash_{\mathrm{S}} \overline{x}\langle\rangle.P} \qquad \textsc{Par}\ \frac{\Gamma_1 \vdash_{\mathrm{S}} P \qquad \Gamma_2 \vdash_{\mathrm{S}} Q}{\Gamma_1, \Gamma_2 \vdash_{\mathrm{S}} P \mid Q}$$

$$\textsc{Res}\ \frac{\Gamma, x : T, y : \overline{T} \vdash_{\mathrm{S}} P}{\Gamma \vdash_{\mathrm{S}} (\nu\, xy)P} \qquad \textsc{Out}\ \frac{\Gamma, x : U \vdash_{\mathrm{S}} P}{\Gamma, x : !T.U, v : T \vdash_{\mathrm{S}} \overline{x}\langle v\rangle.P} \qquad \textsc{In}\ \frac{\Gamma, y : T, x : U \vdash_{\mathrm{S}} P}{\Gamma, x : ?T.U \vdash_{\mathrm{S}} x(y).P}$$

Figure 3 SP: Typing Rules for Processes.

▶ **Theorem 5** (SP: Preservation [36]). *Suppose* $\Gamma \vdash_{\mathrm{S}} P$. *If* $P \equiv Q$ *then* $\Gamma \vdash_{\mathrm{S}} Q$. *Also, if* $P \longrightarrow Q$ *then* $\Gamma \vdash_{\mathrm{S}} Q$.

We will be interested in *well-formed processes*, defined using two auxiliary notions. First, we say that P is *prefixed at* x, written $\mathtt{Pr}(P) = x$, if P has one of the following forms: $x().P'$, $\overline{x}\langle\rangle.P$, $x(z).P'$, or $\overline{x}\langle v\rangle.P'$. Second, we say a process $(\nu\, xy)(P \mid Q)$ is a *redex* if: (1) $P = x().P'$ and $Q = \overline{y}\langle\rangle.Q'$, or (2) $P = x(z).P'$ and $Q = \overline{y}\langle v\rangle.Q'$. We may now define:

▶ **Definition 6** (SP: Well-formedness). *A process is well-formed if for each of its structural congruent processes of the form* $(\nu\, x_1y_1)\dots(\nu\, x_ny_n)(P_1 \mid \cdots \mid P_m)$ *where* $\mathtt{Pr}(P_i) = x_k$ *and* $\mathtt{Pr}(P_j) = y_k$, *then* $(\nu\, x_ky_k)(P_i \mid P_j)$ *is a redex.*

▶ **Theorem 7** (Safety [36]). *Typable processes are well-formed.*

As a consequence of Theorems 5 and 7 we obtain the main guarantee of SP: *typable processes only reduce to well-formed processes.*

▶ **Corollary 8** ([36]). *If* $\Gamma \vdash_{\mathrm{S}} P$ *and* $P \longrightarrow^* Q$, *then* Q *is well-formed.*

Even though typability ensures well-formedness, which excludes error processes such as $(\nu\, xy)(\overline{x}\langle v\rangle.P \mid y().Q)$ and $(\nu\, xy)(x().P \mid y().Q)$, typability in SP also admits well-typed process that exhibit unwanted behaviors, in particular deadlocks. Before illustrating this fact, we give the first definition of deadlock freedom that we will encounter, which follows [9]. We write $(\nu\, \widetilde{xy})Q$ to stand for $(\nu\, x_1y_1)\cdots(\nu\, x_ny_n)Q$, for some $n \geq 0$.

▶ **Definition 9** (SP: Deadlock Freedom). *A process* $P \in \mathsf{SP}$ *is deadlock free, written* $\mathcal{D}_{\mathbf{S}}(P)$, *if the following condition holds: whenever* $P \longrightarrow^* P'$ *and one of the following holds:* (1) $P' \equiv (\nu\, \widetilde{xy})(\overline{x_i}\langle v\rangle.Q_1 \mid Q_2)$, (2) $P' \equiv (\nu\, \widetilde{xy})(x_i(z).Q_1 \mid Q_2)$, (3) $P' \equiv (\nu\, \widetilde{xy})(x_i().Q_1 \mid Q_2)$, *or* (4) $P' \equiv (\nu\, \widetilde{xy})(\overline{x_i}\langle\rangle.Q_1 \mid Q_2)$ *(with* $x_i \in \widetilde{x}$ *in all cases), then* $P' \longrightarrow R$, *for some* R.

Hence, $\mathcal{D}_{\mathbf{S}}(P)$ is read as "P is deadlock-free in SP". Later on, we will encounter analogous notations for HCP and P; they will be denoted $\mathcal{D}_{\mathbf{H}}(P)$ and $\mathcal{D}_{\mathbf{P}}(P)$, respectively.

▶ **Example 10** (SP: A Typable but Deadlocked Process). Consider the process P_{10} defined as

$$P_{10} \triangleq (\nu\, x_1y_1)(\nu\, x_2y_2)(\nu\, v_1k_1)(\nu\, v_2k_2)(x_1(z_1).x_2(z_2).C_1 \mid \overline{y_2}\langle v_2\rangle.\overline{y_1}\langle v_1\rangle.C_2)$$

where $C_1 \triangleq x_1().x_2().z_1().z_2().\mathbf{0}$ and $C_2 \triangleq \overline{y_1}\langle\rangle.\overline{y_2}\langle\rangle.\overline{k_1}\langle\rangle.\overline{k_2}\langle\rangle.\mathbf{0}$. Similar to P_2 (§ 1), the structure of actions in process P_{10} induces a circular dependency that prevents synchronizations: we have that $\neg\mathcal{D}_{\mathbf{S}}(P_{10})$ and yet P_{10} is well-typed with the empty environment.

$$\text{C-}\mathbf{1}\ \frac{}{x[\,].\mathbf{0} \vdash_{\mathsf{C}} x : \mathbf{1}} \qquad \text{C-}\bot\ \frac{P \vdash_{\mathsf{C}} \Gamma}{x().P \vdash_{\mathsf{C}} \Gamma, x : \bot} \qquad \text{C-Mix}_0\ \frac{}{\mathbf{0} \vdash_{\mathsf{C}} \cdot} \qquad \text{C-}\otimes\ \frac{P \vdash_{\mathsf{C}} \Gamma, y : A \qquad Q \vdash_{\mathsf{C}} \Delta, x : B}{x[y].(P \mid Q) \vdash_{\mathsf{C}} \Gamma, \Delta, x : A \otimes B}$$

$$\text{C-}\parr\ \frac{P \vdash_{\mathsf{C}} \Gamma, y : A, x : B}{x(y).P \vdash_{\mathsf{C}} \Gamma, x : A \parr B} \qquad \text{C-Cut}\ \frac{P \vdash_{\mathsf{C}} \Gamma, x : A \qquad Q \vdash_{\mathsf{C}} \Delta, y : A^{\bot}}{(\nu xy)(P \mid Q) \vdash_{\mathsf{C}} \Gamma, \Delta} \qquad \text{C-Id}\ \frac{}{[x \leftrightarrow y] \vdash_{\mathsf{C}} x : A, y : A^{\bot}}$$

Figure 4 CP: Typing rules for processes.

2.2 CP and HCP

The languages CP and HCP use the same syntax. Here we consider HCP as presented by Kokke *et al.* [23] (other variants have been studied in, e.g., [13]). Assuming an infinite set of *names* $(x, y, z, \dots)$, the set of *processes* $(P, Q, \dots)$ in CP and HCP is defined as follows:

$$P, Q ::= \mathbf{0} \mid (\nu xy)P \mid P \mid Q \mid [x \leftrightarrow y] \mid x[y].P \mid x(y).P \mid x[\,].P \mid x().P$$

Most constructs are similar to those in SP; differences are the forwarder process $[x \leftrightarrow y]$, which equates x and y, and process $x[y].P$, which denotes the output of the private name y along x, with continuation P. Also, the empty output is now denoted $x[\,].P$. In $x[y].P$ and $x(y).P$, the name y is bound in P. We write $P\{x/y\}$ to denote the capture-avoiding substitution of y for x in P. We write $\mathsf{fn}(P)$ to denote the free names of P. We use π to range over prefixes: $x[y], x(y), x(), x[\,]$.

The types assigned to names correspond to formulas of classical linear logic (CLL):

$$A, B ::= \mathbf{1} \mid \bot \mid A \otimes B \mid A \parr B$$

The assignment $x : A$ says that x follows the input/output interactions described by A. Assignments $x : A \otimes B$ and $x : A \parr B$ are read as sending and receiving an object of type A along x, with continuation B, respectively. There is a duality in the interpretation of the following pairs: $\otimes$ and $\parr$; $\mathbf{1}$ and $\bot$. Formally, the dual type of A, denoted $A^{\bot}$, is defined as

$$\mathbf{1}^{\bot} \triangleq \bot \qquad \bot^{\bot} \triangleq \mathbf{1} \qquad (A \otimes B)^{\bot} \triangleq A^{\bot} \parr B^{\bot} \qquad (A \parr B)^{\bot} \triangleq A^{\bot} \otimes B^{\bot}$$

Let Γ, Δ range over *environments*, unordered collections of assignments. Given an environment $\Delta = x_1 : A_1, \dots, x_n : A_n$, its domain, written $dom(\Delta)$, is the set $\{x_1, \dots, x_n\}$.

▶ **Definition 11** (CP). *Typing judgements for* CP *are of the form* $P \vdash_{\mathsf{C}} \Gamma$*, with typing rules as given in Figure 4. We shall write* $P \in$ CP *if* $P \vdash_{\mathsf{C}} \Gamma$*, for some* Γ*.*

Observe how typing in CP induces a specific shape for output processes: $x[y].(P \mid Q)$. This is not the case in HCP, where processes are described by *hyperenvironments*, unordered collections of environments: $\mathcal{H}, \mathcal{G} ::= \Gamma_1 \parallel \cdots \parallel \Gamma_n$.

▶ **Definition 12** (HCP). *Typing judgements for* HCP *are of the form* $P \vdash_{\mathsf{H}} \mathcal{G}$*, with typing rules as given in Figure 5. We shall write* $P \in$ HCP *if* $P \vdash_{\mathsf{H}} \mathcal{G}$*, for some* $\mathcal{G}$*.*

CP and HCP differ in process composition, which is key to exclude the circular dependencies that lead to deadlocks: it is realized by Rule C-Cut in CP and Rule H-Cut in HCP. The former requires two premises describing processes that share exactly one session; typing

$$\text{H-Mix}_2\ \frac{P \vdash_{\text{H}} \mathcal{H} \quad Q \vdash_{\text{H}} \mathcal{G}}{P \mid Q \vdash_{\text{H}} \mathcal{H} \parallel \mathcal{G}} \qquad \text{H-Mix}_0\ \frac{}{\mathbf{0} \vdash_{\text{H}} \cdot} \qquad \text{H-ID}\ \frac{}{[x \leftrightarrow y] \vdash_{\text{H}} x : A, y : A^{\perp}} \qquad \text{H-}\bot\ \frac{P \vdash_{\text{H}} \mathcal{H} \parallel \Delta}{x().P \vdash_{\text{H}} \mathcal{H} \parallel \Delta, x : \bot} \qquad \text{H-}\mathbf{1}\ \frac{P \vdash_{\text{H}} \mathcal{H}}{x[].P \vdash_{\text{H}} \mathcal{H} \parallel x : \mathbf{1}}$$

$$\text{H-}\otimes\ \frac{P \vdash_{\text{H}} \mathcal{H} \parallel \Delta, y : A \parallel \Gamma, x : B}{x[y].P \vdash_{\text{H}} \mathcal{H} \parallel \Delta, \Gamma, x : A \otimes B} \qquad \text{H-}\parr\ \frac{P \vdash_{\text{H}} \mathcal{G} \parallel \Delta, y : B, x : A}{x(y).P \vdash_{\text{H}} \mathcal{G} \parallel \Delta, x : A \parr B} \qquad \text{H-Cut}\ \frac{P \vdash_{\text{H}} \mathcal{H} \parallel \Delta, x : A \parallel \Gamma, y : A^{\perp}}{(\nu\, xy)P \vdash_{\text{H}} \mathcal{H} \parallel \Delta, \Gamma}$$

Figure 5 HCP: Typing rules for processes.

involves both composition and restriction and induces tree-like process topologies. The latter involves one premise only, which describes a process in which the separation between x and y is made explicit by "$\parallel$" at the level of hyperenvironments; typing involves restriction only.

Every P typable in CP is also typable in HCP, but the converse does not hold. We have:

▶ **Lemma 13** (Relation Between CP and HCP [23]). *If* $P \vdash_{\text{C}} \Gamma$ *then* $P \vdash_{\text{H}} \Gamma$.

▶ **Example 14** (The converse of Lemma 13 does not hold). To illustrate a process typable in HCP but not in CP, consider $P_{14} \triangleq (\nu\, x_1y_1)(\nu\, x_2y_2)(\nu\, v_1k_1)(\nu\, v_2k_2)(P'_{14} \mid P''_{14})$, where:

$$P'_{14} \triangleq x_1(z_1).x_2(z_2).x_1().x_2().z_1().z_2().\mathbf{0}$$
$$P''_{14} \triangleq y_1[v'_1].([v'_1 \leftrightarrow v_1] \mid y_2[v'_2].([v'_2 \leftrightarrow v_2] \mid C)) \qquad C \triangleq y_1[].\mathbf{0} \mid y_2[].\mathbf{0} \mid k_1[].\mathbf{0} \mid k_2[].\mathbf{0}$$

We have $P_{14} \vdash_{\text{H}} \cdot$. First, $P'_{14} \vdash_{\text{H}} x_1 : \bot \parr \bot, x_2 : \bot \parr \bot$ follows from several applications of Rules H-$\bot$ and H-$\parr$. To type P''_{14}, we first obtain $C \vdash_{\text{H}} y_1 : \mathbf{1} \parallel y_2 : \mathbf{1} \parallel k_1 : \mathbf{1} \parallel k_2 : \mathbf{1}$, crucially using Rule H-Mix$_2$ several times to compose the four independent processes. Because $[v'_2 \leftrightarrow v_2] \vdash_{\text{H}} v'_2 : \mathbf{1}, v_2 : \bot$, by Rules H-Mix$_2$ and H-$\otimes$ we have: $y_2[v'_2].([v'_2 \leftrightarrow v_2] \mid C) \vdash_{\text{H}} y_1 : \mathbf{1} \parallel y_2 : \mathbf{1} \otimes \mathbf{1}, v_2 : \bot \parallel k_1 : \mathbf{1} \parallel k_2 : \mathbf{1}$. We proceed similarly to type the output on y_1 and complete the typing of P''_{14}. Finally, to compose P'_{14} and P''_{14} we apply H-Mix$_2$ and H-Cut.

Now, C cannot be typed in CP: there is no rule similar to Rule H-Mix$_2$, which can compose independent processes. Hence, P_{14} is not typable in CP.

In general, the composition of two processes that share more than one session cannot be typed in CP. Now consider Q_{14}:

$$Q_{14} \triangleq (\nu\, v_2k_2)((\nu\, v_1k_1)((\nu\, x_2y_2)((\nu\, x_1y_1)(P'_{14} \mid y_1[v'_1].([v'_1 \leftrightarrow v_1] \mid y_1[].\mathbf{0}) \mid y_2[v'_2].([v'_2 \leftrightarrow v_2] \mid y_2[].\mathbf{0})) \mid k_1[].\mathbf{0}) \mid k_2[].\mathbf{0})$$

Observe how Q_{14} modifies P_{14} by keeping P'_{14} unchanged and dividing the two outputs in P''_{14} into threads that can be composed separately using Rule C-Cut. We have $Q_{14} \vdash_{\text{C}} \cdot$.

Given a $P \vdash_{\text{H}} \mathcal{G}$, with $\mathcal{G} = \Gamma_1 \parallel \cdots \parallel \Gamma_n$, we are interested in partitioning the free names in P according to the environments $\Gamma_1, \ldots, \Gamma_n$. These *name partitions* describe which names are used by each parallel component of P; they are defined considering all the different hyperenvironments under which P is typable.

▶ **Definition 15** (HCP: Name Partitions). *Given* $\mathcal{G} = \Gamma_1 \parallel \cdots \parallel \Gamma_n$, *we define* $\mathsf{Part}(\mathcal{G}) = \{dom(\Gamma_1), \ldots, dom(\Gamma_n)\}$. *Given* P, *its* name partitions *are* $\mathsf{np}(P) = \{\mathsf{Part}(\mathcal{G}) \mid P \vdash_{\text{H}} \mathcal{G}\}$.

$x *_P y$ *holds whenever there is a* $G \in \mathsf{np}(P)$ *such that* x *and* y *belong to different proper subsets of* G. *Also,* $x \circledast_P y$ *holds whenever* x *and* y *belong to the same proper subset of* G.

▶ **Example 16.** Consider the process $P_{16} \triangleq x().(z[].\mathbf{0} \mid y[].\mathbf{0})$: it can be typed under the hyperenvironments $\mathcal{G} \triangleq x : \bot, z : \mathbf{1} \parallel y : \mathbf{1}$ and $\mathcal{H} \triangleq z : \mathbf{1} \parallel x : \bot, y : \mathbf{1}$. Given this, $\mathsf{np}(P_{16}) = \{\{\{x, z\}, \{y\}\}, \{\{z\}, \{x, y\}\}\}$, and we have, e.g., $y *_{P_{16}} z$ and $x \circledast_{P_{16}} y$.

$$\frac{}{[x \leftrightarrow y] \xrightarrow{[x \leftrightarrow y]} \mathbf{0}}\ [x \leftrightarrow y] \qquad \frac{}{[x \leftrightarrow y] \xrightarrow{[y \leftrightarrow x]} \mathbf{0}}\ [x \leftrightarrow y] \qquad \frac{\pi \in \{x[y], x(y), x(), x[]\}}{\pi.P \xrightarrow{\pi} P}\ \pi$$

$$\frac{P \xrightarrow{l} P' \quad \mathsf{bn}(l) \cap \mathsf{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{l} P' \mid Q}\ \text{PAR}_1 \qquad \frac{Q \xrightarrow{l} Q' \quad \mathsf{bn}(l) \cap \mathsf{fn}(P) = \emptyset}{P \mid Q \xrightarrow{l} P \mid Q'}\ \text{PAR}_2 \qquad \frac{P \xrightarrow{[y \leftrightarrow z]} P'}{(\nu\, xy)P \xrightarrow{\tau} P'\{x/z\}}\ \textsc{AxCut}$$

$$\frac{P \xrightarrow{x[] \mid y()} P'}{(\nu\, xy)P \xrightarrow{\tau} P'}\ \bot\mathbf{1} \qquad \frac{P =_\alpha Q \quad Q \xrightarrow{l} R}{P \xrightarrow{l} R}\ =_\alpha \qquad \frac{P \xrightarrow{l} P' \quad x, y \notin \mathsf{cn}(l) \quad x *_{P'} y}{(\nu\, xy)P \xrightarrow{l} (\nu\, xy)P'}\ \textsc{Res}$$

$$\frac{P \xrightarrow{x[x'] \mid y(y')} P'}{(\nu\, xy)P \xrightarrow{\tau} (\nu\, xy)(\nu\, x'y')P'}\ \otimes\!\!\!\!\otimes \qquad \frac{P \xrightarrow{l} P' \quad Q \xrightarrow{l'} Q' \quad \mathsf{bn}(l) \cap \mathsf{bn}(l') = \emptyset}{P \mid Q \xrightarrow{l \mid l'} P' \mid Q'}\ \text{SYN}$$

$$\frac{P \xrightarrow{l} P' \quad x \notin \mathsf{cn}(l)}{x[].P \xrightarrow{l} x[].P'}\ \pi_1 \qquad \frac{P \xrightarrow{l} P' \quad \mathsf{fn}(P') \neq \emptyset}{x().P \xrightarrow{l} x().P'}\ \pi_2 \qquad \frac{P \xrightarrow{l} P' \quad x, x' \notin \mathsf{cn}(l) \quad x *_{P'} x'}{x[x'].P \xrightarrow{l} x[x'].P'}\ \otimes$$

$$\frac{P \xrightarrow{l} P' \quad x, x' \notin \mathsf{cn}(l) \quad x \circledast_{P'} x'}{x(x').P \xrightarrow{l} x(x').P'}\ \parr \qquad \frac{\pi.P \xrightarrow{l} \pi.P' \quad \mathsf{fn}(\pi) *_{\pi.P} \mathsf{fn}(l)}{\pi.P \xrightarrow{\pi \mid l} P'}\ \mid \pi$$

Figure 6 The LTS for HCP (Definition 17) includes rules for actions and communication (top) and for delayed actions and self-synchronizations (bottom).

The semantics of HCP is given in terms of a Labeled Transition System (LTS):

▶ **Definition 17** (HCP: LTS). *The binary relation on* HCP *processes* $\xrightarrow{l}$ *is defined by the rules in Figure 6, with action labels defined as follows:*

$$l, l' ::= [x \leftrightarrow y] \mid x[y] \mid x(y) \mid x() \mid x[] \mid l \mid l \mid \tau$$

In the following, $\mathsf{fn}(l)$, $\mathsf{bn}(l)$, *and* $\mathsf{cn}(l)$ *denote the free, bound, and channel names of* l, *respectively. They are defined as follows:* $\mathsf{fn}([x \leftrightarrow y]) \triangleq \{x, y\}$, $\mathsf{fn}(\tau) = \mathsf{bn}(\tau) \triangleq \emptyset$, $\mathsf{fn}(x[y]) = \mathsf{fn}(x(y)) = \mathsf{fn}(x()) = \mathsf{fn}(x()) \triangleq x$, *and* $\mathsf{bn}(x[y]) = \mathsf{bn}(x(y)) \triangleq \{y\}$. *Also,* $\mathsf{fn}(l_1 \mid l_2) \triangleq \mathsf{fn}(l_1) \cup \mathsf{fn}(l_2)$, *and* $\mathsf{bn}(l_1 \mid l_2) \triangleq \mathsf{bn}(l_1) \cup \mathsf{bn}(l_2)$. *Moreover,* $\mathsf{cn}(l) \triangleq \mathsf{fn}(l) \cup \mathsf{bn}(l)$.

The rules in Figure 6 are inspired by proof-theoretical transformations on HCP's proof derivations [23]; they can be divided into two classes. The rules above the line are usual formulations for actions, structural rules, and communication. Actions can appear in parallel in a label, and compatible actions in parallel induce a τ-transition. The five rules below the line formalize *delayed actions* and *self-synchronizations*. Intuitively, the first four rules allow to observe an action l in a delayed style; such an action must be independent from the prefix at the top-level (on x). The last rule, $\mid \pi$, implements self-synchronization for a process $\pi.P$: the (top-level) prefix π is considered in parallel with a (delayed) action l emanating from P.

▶ **Example 18.** To illustrate the LTS of HCP, recall the processes P_1 and P_2 from § 1. In HCP, they are denoted as: $P_{18} \triangleq (\nu\, xx')(x[].x'().\mathbf{0})$ and $Q_{18} \triangleq (\nu\, xx')(\nu\, yy')(y[].x().\mathbf{0} \mid x'[].y'().\mathbf{0})$. Consider the following type derivation, parametric on w, z:

$$\Pi_{w,z} = \frac{\dfrac{\mathbf{0} \vdash_{\text{H}}}{w().\mathbf{0} \vdash_{\text{H}} w : \bot}\ \text{H-}\bot}{z[].w().\mathbf{0} \vdash_{\text{H}} w : \bot \parallel z : \mathbf{1}}\ \text{H-}\mathbf{1}$$

Thus, P_{18} is typed as $\Pi_{x',x}$ followed by an application of Rule H-Cut. Also, Q_{18} is typed by $\Pi_{x,y}$ and $\Pi_{y',x'}$ followed by an application of Rules H-Mix$_2$ and H-Cut. Because Rule C-**1** only types empty outputs followed by $\mathbf{0}$, neither P_{18} nor Q_{18} are typable in CP.

By Rules $\mid \pi$ and $\bot\mathbf{1}$ in Figure 6, $P_{18} \xrightarrow{\tau} \mathbf{0}$. Similarly, $y[].x().\mathbf{0} \mid x'[].y'().\mathbf{0} \xrightarrow{x() \mid x'[]} y[].\mathbf{0} \mid y'().\mathbf{0}$ by Rules π, π_1 and SYN. Finally, by Rule $\bot\mathbf{1}$, $Q_{18} \xrightarrow{\tau} (\nu\, yy')(y[].\mathbf{0} \mid y'().\mathbf{0})$.

We now recall some useful notions and results from [23].

▶ **Definition 19** (HCP: Weak LTS and Bisimilarity [23]). *$\stackrel{l}{\Rightarrow}$ is the smallest relation such that:* (1) *$P \stackrel{\tau}{\Rightarrow} P$ for all P;* (2) *and if $P \stackrel{\tau}{\Rightarrow} P'$, $P' \xrightarrow{l} Q'$, and $Q' \stackrel{\tau}{\Rightarrow} Q$, then $P \stackrel{l}{\Rightarrow} Q$.*

A symmetric relation $\mathcal{R}$ is a bisimulation if $P\ \mathcal{R}\ Q$ implies that if $P \stackrel{l}{\Rightarrow} P'$ then $Q \stackrel{l}{\Rightarrow} Q'$ for some Q' such that $P'\ \mathcal{R}\ Q'$. Bisimilarity is the largest relation $\approx_{\mathtt{H}}$ that is a bisimulation.

Well-typed processes satisfy readiness, defined as follows:

▶ **Theorem 20** (HCP: Readiness [23]). *Let $P \vdash_{\mathtt{H}} \Gamma_1 \parallel \cdots \parallel \Gamma_n$. For every $i \in [1,n]$, there exist $x \in dom(\Gamma_i)$, l_x, and P' such that $P \stackrel{l_x}{\Longrightarrow} P'$.*

In HCP, transitions associated to sessions in different partitions can be fired in parallel:

▶ **Lemma 21** ([23]). *If $P \vdash_{\mathtt{H}} \mathcal{G} \parallel \Gamma, x : A \parallel \Delta, y : B$, $P \stackrel{l_x}{\Longrightarrow} P'$, and $P \stackrel{l_y}{\Longrightarrow} P''$, then there exists Q s.t. $P \stackrel{l_x \mid l_y}{\Longrightarrow} Q$ (up to α-renaming).*

▶ **Definition 22** (HCP: Structural Congruence). *Let $\equiv_{\mathtt{H}}$ be the smallest congruence relation generated by the axioms:*

$$P \mid Q \equiv_{\mathtt{H}} Q \mid P \qquad P \mid \mathbf{0} \equiv_{\mathtt{H}} P$$
$$P \mid (Q \mid R) \equiv_{\mathtt{H}} (P \mid Q) \mid R \qquad (\nu\, xy)P \equiv_{\mathtt{H}} (\nu\, yx)P$$
$$(\nu\, xy)P \mid Q \equiv_{\mathtt{H}} (\nu\, xy)(P \mid Q) \quad \text{if } x, y \notin \mathsf{fn}(Q) \qquad (\nu\, xy)(\nu\, wz)P \equiv_{\mathtt{H}} (\nu\, wz)(\nu\, xy)P$$

Because of Readiness (Theorem 20), HCP also satisfies *progress*:

▶ **Corollary 23** (HCP: Progress [23]). *If $P \in$ HCP and $P \not\equiv_{\mathtt{H}} \mathbf{0}$, then $P \xrightarrow{l} P'$ for some l, P'.*

In order to relate with reduction-based semantics (in SP), we define the following:

▶ **Definition 24** (HCP: Reduction). *We write $\rightarrow^*$ for the reflexive, transitive closure of $\xrightarrow{\tau}$.*

▶ **Lemma 25.** *Let $P \in$ HCP s.t. $P \vdash_{\mathtt{H}} \cdot$, then exists P' s.t. $P \rightarrow^* P' \equiv_{\mathtt{H}} \mathbf{0}$.*

Proof sketch. We prove it in two steps: (1) We prove that if $P \not\equiv_{\mathtt{H}} \mathbf{0}$, then there exists P' s.t. $P \xrightarrow{\tau} P'$, and $P' \vdash_{\mathtt{H}} \cdot$. This ensures the existence of a τ-transition in every step. (2) We then prove that $\mathbf{0}$ can be reached in finitely many steps. See [19] for details. ◀

Inspired by Lemma 25 we have the following definition of deadlock freedom for HCP.

▶ **Definition 26** (HCP: Deadlock Freedom). *Let $P \in$ HCP. We say that P is deadlock free, written $\mathcal{D}_{\mathbf{H}}(P)$, iff there exists P' s.t. $P \rightarrow^* P' \equiv_{\mathtt{H}} \mathbf{0}$.*

HCP has a *denotational semantics* that interprets well-typed processes as sets of possible interactions on their free names. Given a process P, its set of denotations is written $\|P\|$. Denotational equivalence, noted $\doteq$, is the typed relation defined as: $P \doteq Q$ iff $\|P\| = \|Q\|$. We shall rely on denotational equivalence; its exact formulation, given in [19], is interesting but not crucial for following our results. We notice that $\equiv_{\mathtt{H}}$ is sound wrt $\doteq$ and that our results are reflected by bisimilarity, as ensured by the following result:

▶ **Theorem 27** (HCP: Full Abstraction [23]). *For well-typed processes: $\approx_{\mathtt{H}} = \doteq$.*

3 The Role of Delayed Actions and Self-Synchronizations in DF

We have seen that SP defines a flexible typing discipline that admits deadlocked and deadlock-free processes. Also, we saw the way in which CP and HCP enforce DF by typing, and that the latter's LTS allows for delayed actions and self-synchronizations. By defining two classes of SP processes, $\mathbb{C}$ and $\mathbb{H}$, which capture typability as in CP and HCP, respectively (Definition 34), we study whether delayed actions and self-synchronizations influence DF.

We proceed as follows. As we have seen, $\mathcal{D}_{\mathbf{H}}(P)$ means that P is deadlock free by considering all sources of behavior, including delayed actions and self-synchronizations; we shall also define $\mathcal{D}^{\mathsf{R}}_{\mathbf{H}}(P)$ to mean that P is deadlock free by considering *regular transitions only*. Our main result is Corollary 53, which ensures: (1) delayed actions and self-synchronizations are *inessential* when it comes to DF, and (2) given an encoding $[\![\cdot]\!] : \mathsf{SP} \to \mathsf{HCP}$, for a process $P \in \mathsf{SP}$ such that $\mathcal{D}_{\mathbf{H}}([\![P]\!])$, there is a P' that *simulates* P, and $\mathcal{D}^{\mathsf{R}}_{\mathbf{H}}([\![P']\!])$.

3.1 From SP to HCP

We start by translating processes in SP into processes in HCP:

▶ **Definition 28.** *Let $P \in \mathsf{SP}$. The encoding $[\![\cdot]\!] : \mathsf{SP} \to \mathsf{HCP}$ is a homomorphism except for the case of free output, defined as $[\![\overline{x}\langle v\rangle.P]\!] \triangleq x[y].([y \leftrightarrow v] \mid [\![P]\!])$. For types, we define:*

$$[\![\mathsf{end}_?]\!] \triangleq \bot \qquad [\![\mathsf{end}_!]\!] \triangleq \mathbf{1} \qquad [\![?T.S]\!] \triangleq [\![T]\!] \parr [\![S]\!] \qquad [\![!T.S]\!] \triangleq [\![T]\!]^{\bot} \otimes [\![S]\!]$$

For typing environments, we define: $[\![\emptyset]\!] \triangleq \emptyset$ and $[\![\Gamma, x : T]\!] \triangleq [\![\Gamma]\!], x : [\![T]\!]$.

Regular Transitions do not depend on self-synchronizations or delayed actions:

▶ **Definition 29** (HCP: Regular Transitions)**.** *A derivation T for $P \xrightarrow{l} Q$ is denoted $T : P \xrightarrow{l} Q$. We define $\mathsf{RT}(T)$ iff T is derived only using rules above the horizontal line in Figure 6.*

▶ **Example 30.** Recall the processes P_{18} and Q_{18} from Example 18, which use self-synchronizations and delayed actions, respectively: we have that $T_1 : P_{18} \xrightarrow{\tau} \mathbf{0}$, and $T_2 : Q_{18} \xrightarrow{\tau} (\nu\, yy')(y[\,].\mathbf{0} \mid y'().\mathbf{0})$. Therefore, $\neg\mathsf{RT}(T_1)$ and $\neg\mathsf{RT}(T_2)$.

The encoding of free output induces transitions involving the forwarder process, not present in the source SP process. We handle the substitutions induced by forwarders in Rule AxCut (Figure 6) as follows:

▶ **Definition 31** (HCP: Ready-for-Substitution)**.** *Let $P \in \mathsf{HCP}$. We say that $P = (\nu\, xy)P'$ is ready for substitution of z for x, written $\mathsf{RS}_{z,x}(P)$, if there exists $z \in \mathsf{fn}(P)$ and P' s.t. $P \xrightarrow{[y \leftrightarrow z]} P'$, and $(\nu\, xy)P \xrightarrow{\tau} P'\{x/z\}$. Otherwise, we write $\neg\mathsf{RS}(P)$.*

We may now define transitions for processes that are ready for a substitution.

▶ **Definition 32** (HCP: Eager-to-Rename)**.** *The relation $\xrightarrow[\mathtt{ER}]{l}$ on HCP processes is defined as:*

$$(P \xrightarrow{l} Q) \wedge \mathsf{RS}_{x,z}(Q) \Rightarrow P \xrightarrow[\mathtt{ER}]{l} Q\{x/z\} \qquad \textit{and} \qquad (P \xrightarrow{l} Q) \wedge \neg\mathsf{RS}(Q) \Rightarrow P \xrightarrow[\mathtt{ER}]{l} Q$$

Intuitively, $\xrightarrow[\mathtt{ER}]{l}$ executes outputs and forwarders in one single step. When $[\![P]\!]$ only relies on regular transitions we have the following operational correspondence result.

▶ **Theorem 33** ($[\![\cdot]\!]$: Operational Correspondence). *Let* $P \in \mathbb{H}$.
1. *If* $T : [\![P]\!] \xrightarrow[\text{ER}]{\tau} R$ *and* $\mathsf{RT}(T)$, *then* $P \longrightarrow Q$ *s.t.* $[\![Q]\!] \equiv_{\text{H}} R$.
2. *If* $P \longrightarrow Q$, *then* $[\![P]\!] \xrightarrow[\text{ER}]{\tau} R$ *s.t.* $R \equiv_{\text{H}} [\![Q]\!]$.

Proof sketch. Part (1) is proven by case analysis in $\xrightarrow{\tau}$, and (2) is proven by induction in the derivation of $\longrightarrow$. See [19] for details. ◀

Following Dardha and Pérez [9], we now define $\mathbb{C}$ and $\mathbb{H}$ as the classes of SP processes that are also typable in CP and HCP, respectively:

▶ **Definition 34** (Classes $\mathbb{C}$ and $\mathbb{H}$). *The classes* $\mathbb{C}$ *and* $\mathbb{H}$ *of* SP *processes are defined as:*

$$\mathbb{C} \triangleq \{P \in \mathsf{SP} \mid \Gamma \vdash_{\text{S}} P \wedge [\![P]\!] \vdash_{\text{C}} [\![\Gamma]\!]\}$$
$$\mathbb{H} \triangleq \{P \in \mathsf{SP} \mid \Gamma \vdash_{\text{S}} P \wedge \Gamma = \Gamma_1, \ldots, \Gamma_n \wedge [\![P]\!] \vdash_{\text{H}} [\![\Gamma_1]\!] \parallel \cdots \parallel [\![\Gamma_n]\!]\}$$

Hence, in writing "$P \in \mathbb{H}$" we are not referring to an HCP process directly, but rather to a SP process P (in the sense of § 2.1) that is related (up to $[\![\cdot]\!]$) to some (typable) HCP process. This is how SP offers a unifying framework for comparing different type systems. As a direct consequence of Lemma 13 and Example 14 we have that $\mathbb{C} \subsetneq \mathbb{H}$.

3.2 A Refined Account of $\mathbb{H}$, Based on Deadlock Freedom

As explained in the previous section, HCP enjoys liveness properties corresponding to the LTS. If we focus only on τ-transitions (cf. Definition 24) we obtain the following results.

We extend Definition 29 (regular transitions) to sequences of transitions, as follows:

▶ **Definition 35** (HCP: Regular Sequence of Transitions). *Assume* $P_1, \ldots, P_n$ *in* HCP *and transitions* $T_i : P_i \xrightarrow{l_i} P_{i+1}$. *We say that a transition sequence* $\rho : P \xrightarrow{l_1} \cdots \xrightarrow{l_n} P_n$ *transitions regularly iff* $\mathsf{RT}(T_i)$ *for all* $i = 1, \ldots, n$. *We write* $\mathsf{RT}(\rho)$.

We are interested in processes whose transitions are regular, thus we refine deadlock freedom in HCP (Definition 26) as follows:

▶ **Definition 36** (HCP: Deadlock Freedom based on Regular Transitions). *Let* $P \in \mathsf{HCP}$. *We say that* P *is deadlock free without delayed actions and self-synchronizations, written* $\mathcal{D}^{\text{R}}_{\mathbf{H}}(P)$, *iff* $\mathcal{D}_{\mathbf{H}}(P)$, $\rho : P \to^* P' \equiv_{\text{H}} \mathbf{0}$, *and* $\mathsf{RT}(\rho)$.

In turn, Definitions 26 and 36 allow us to refine the definition of $\mathbb{H}$ (Definition 34):

▶ **Definition 37** (HCP: Sub-classes of $\mathbb{H}$). *We define the following classes of DF processes:*

$$\mathbb{H}_{\text{F}} \triangleq \{P \in \mathbb{H} \mid \mathcal{D}_{\mathbf{H}}([\![P]\!])\} \qquad \mathbb{H}_{\text{R}} \triangleq \{P \in \mathbb{H} \mid \mathcal{D}^{\text{R}}_{\mathbf{H}}([\![P]\!])\}$$

By Definitions 34 and 37 we have $\mathbb{H}_{\text{R}} \subsetneq \mathbb{H}_{\text{F}} \subsetneq \mathbb{H}$.

▶ **Example 38.** Consider the SP processes:

$$P_{38} \triangleq (\nu\, wz)(\overline{w}\langle\rangle.(\nu\, xy)(\overline{x}\langle\rangle.\mathbf{0} \mid y().z().\mathbf{0})) \quad Q_{38} \triangleq (\nu\, wz)(\overline{w}\langle\rangle.\mathbf{0} \mid (\nu\, xy)(\overline{x}\langle\rangle.\mathbf{0} \mid z().y().\mathbf{0}))$$

We have $P_{38} \in \mathbb{H}_{\text{F}} \setminus \mathbb{H}_{\text{R}}$ and $Q_{38} \in \mathbb{H}_{\text{R}}$. Although $\neg\mathcal{D}_{\mathbf{S}}(P_{38})$, the HCP process $[\![P_{38}]\!]$ has a τ-transition due to a delayed action, and therefore $\mathcal{D}_{\mathbf{H}}([\![P_{38}]\!])$.

▶ **Theorem 39.** *If* $P \in \mathbb{H}_{\text{R}}$, *then* $\mathcal{D}_{\mathbf{S}}(P)$.

Proof. It follows from Theorem 33 and Definitions 9 and 37. ◀

$$
\begin{array}{cc}
x().x'().P \simeq_{cc} x'().x().P & x().x'(y').P \simeq_{cc} x'(y').x().P \\
x(y).x'(y').P \simeq_{cc} x'(y').x(y).P & x().x'[y'].P \simeq_{cc} x'[y'].x().P \\
x[y].x'(y').P \simeq_{cc} x'(y').x[y].P & x[\,].x'(y').P \simeq_{cc} x'(y').x[\,].P \\
x[y].x'[y'].P \simeq_{cc} x'[y'].x[y].P & x[\,].x'[y'].P \simeq_{cc} x'[y'].x[\,].P \\
x[\,].x'[\,].P \simeq_{cc} x'[\,].x[\,].P & x[\,].x'().P \simeq_{cc} x'().x[\,].P \\
(\nu\, xy)(x'().P) \simeq_{cc} x'().(\nu\, xy)(P) & (\nu\, xy)(x'[\,].P) \simeq_{cc} x'[\,].(\nu\, xy)(P) \\
(\nu\, xy)(x'(y').P) \simeq_{cc} x'(y').(\nu\, xy)(P) & (\nu\, xy)(x'[y'].P) \simeq_{cc} x'[y'].(\nu\, xy)(P)
\end{array}
$$

Figure 7 HCP: Commuting Conversions (Definition 40).

From $\mathbb{H}_{\mathsf{F}}$ to $\mathbb{H}_{\mathsf{R}}$ via process optimizations. We now embark to show how to transform processes in $\mathbb{H}_{\mathsf{F}}$ (where self-synchronizations and delayed actions are used) into processes in $\mathbb{H}_{\mathsf{R}}$ (where they are not). Intuitively, we would like to transform, e.g.,

$$(\nu\, wz)(\overline{w}\langle\,\rangle.(\nu\, xy)(\overline{x}\langle\,\rangle.\mathbf{0} \mid y().z().\mathbf{0})) \quad \text{into} \quad (\nu\, wz)(\overline{w}\langle\,\rangle.\mathbf{0} \mid (\nu\, xy)(\overline{x}\langle\,\rangle.\mathbf{0} \mid z().y().\mathbf{0}))$$

The transformation from $\mathbb{H}_{\mathsf{F}}$ to $\mathbb{H}_{\mathsf{R}}$ is formalized by Theorem 47. To achieve it, we leverage two existing ingredients. The *first ingredient* is the notion of *commuting conversions*: sound transformations on processes that involve actions from independent sessions (cf. [10, 30]).

▶ **Definition 40** (HCP: Commuting Conversions)**.** *We define $\simeq_{cc}$ as the smallest congruence on* HCP *processes induced by the equalities in Figure 7.*

The transformations induced by $\simeq_{cc}$ have a proof-theoretical origin, and so they are type-preserving, which ensures that they do not alter the causality relations on processes. We can show that that $\simeq_{cc}$ is sound wrt $\approx_{\mathtt{H}}$, i.e., if $P \simeq_{cc} Q$ then $P \approx_{\mathtt{H}} Q$.

▶ **Example 41.** Recall processes P'_{14}, P''_{14}, and P_{14} from Example 14. We illustrate the optimizations on processes induced by commuting conversions by combining P''_{14} and the process $Q_{41} \triangleq x_2(z_2).x_1(z_1).x_1().x_2().z_1().z_2().\mathbf{0}$, as follows:

$$P_{41} \triangleq (\nu\, x_1y_1)(\nu\, x_2y_2)(\nu\, v_1k_1)(\nu\, v_2k_2)(Q_{41} \mid P''_{14})$$

Note that a τ-transition along the sessions x_1y_1 or x_2y_2 in P_{41} would depend on a delayed action. Now, we can use a commuting conversion to swap the order of the inputs, effectively anticipating the input on x_1: $Q_{41} \simeq_{cc} x_1(z_1).x_2(z_2).x_1().x_2().z_1().z_2().\mathbf{0} = P'_{14}$; therefore, $P_{41} \simeq_{cc} P_{14}$. Clearly, there is a transition sequence $\rho : P_{14} \to^* \equiv_{\mathtt{H}} \mathbf{0}$ s.t. $\mathsf{RT}(\rho)$.

The *second ingredient* is *disentanglement*, which also induces optimizations on processes:

▶ **Definition 42** (HCP: Disentanglement [22])**.** *Disentanglement is described by the smallest relation $\rightsquigarrow$ on (typed) processes closed under the rules in Figure 8. We write $\rightsquigarrow^*$ for the reflexive and transitive closure of $\rightsquigarrow$.*

Each of the process transformations in Figure 8 is justified by an operation on derivations; this way, e.g., for the second rule we have:

$$\dfrac{\dfrac{P \vdash_{\mathtt{H}} \mathcal{G} \parallel \Gamma \qquad Q \vdash_{\mathtt{H}} \mathcal{H}}{P \mid Q \vdash_{\mathtt{H}} \mathcal{G} \parallel \mathcal{H} \parallel \Gamma}\,\mathsf{H}\text{-}\textsc{Mix}_2}{x().(P \mid Q) \vdash_{\mathtt{H}} \mathcal{G} \parallel \mathcal{H} \parallel \Gamma, x : \bot}\,\mathsf{H}\text{-}\bot \quad \rightsquigarrow \quad \dfrac{\dfrac{P \vdash_{\mathtt{H}} \mathcal{G} \parallel \Gamma}{x().P \vdash_{\mathtt{H}} \mathcal{G} \parallel \Gamma, x : \bot}\,\mathsf{H}\text{-}\bot \qquad Q \vdash_{\mathtt{H}} \mathcal{H}}{x().P \mid Q \vdash_{\mathtt{H}} \mathcal{G} \parallel \mathcal{H} \parallel \Gamma, x : \bot}\,\mathsf{H}\text{-}\textsc{Mix}_2$$

$$
\begin{array}{lcl}
x[\,].P \vdash_{\mathtt{H}} \mathcal{G} \parallel x : \mathbf{1} & \rightsquigarrow & x[\,].\mathbf{0} \mid P \vdash_{\mathtt{H}} \mathcal{G} \parallel x : \mathbf{1} \\
x().(P \mid Q) \vdash_{\mathtt{H}} \mathcal{G} \parallel \mathcal{H} \parallel \Gamma, x : \bot & \rightsquigarrow & x().P \mid Q \vdash_{\mathtt{H}} \mathcal{G} \parallel \mathcal{H} \parallel \Gamma, x : \bot \\
(\nu xy)(P \mid Q) \vdash_{\mathtt{H}} \mathcal{G} \parallel \mathcal{H} \parallel \Gamma \parallel \Delta & \rightsquigarrow & (\nu\, xy)P \mid Q \vdash_{\mathtt{H}} \mathcal{G} \parallel \mathcal{H} \parallel \Gamma \parallel \Delta \\
x[y].(P \mid Q) \vdash_{\mathtt{H}} \mathcal{G} \parallel \mathcal{H} \parallel \Gamma, \Delta, x : A \otimes B & \rightsquigarrow & x[y].P \mid Q \vdash_{\mathtt{H}} \mathcal{G} \parallel \mathcal{H} \parallel \Gamma, \Delta, x : A \otimes B \\
x(y).(P \mid Q) \vdash_{\mathtt{H}} \mathcal{G} \parallel \mathcal{H} \parallel \Gamma, \Delta, x : A \parr B & \rightsquigarrow & x(y).P \mid Q \vdash_{\mathtt{H}} \mathcal{G} \parallel \mathcal{H} \parallel \Gamma, \Delta, x : A \otimes B
\end{array}
$$

Figure 8 HCP: Disentanglement (Definition 42). See [19] for details.

This way, disentanglement proceeds by the repeated application of the rules from Figure 8 to the derivation of $P \vdash_{\mathtt{H}} \Gamma_1 \parallel \cdots \parallel \Gamma_n$, which aims at moving downwards the instances of Rule H-Mix$_2$ (cf. Figure 5). If H-Mix$_2$ gets stuck in either H-Cut or H-$\otimes$, then it becomes an instance of C-Cut or C-$\otimes$, respectively (cf. Figure 4).

Disentanglement increases the parallelism in processes. This way, e.g., for P_{14} and Q_{14} (Example 14) we have: $P_{14} \rightsquigarrow^* Q_{14}$. Also, note that Rule H-**1** types $x[\,].P \vdash_{\mathtt{H}} \Gamma \parallel x : \mathbf{1}$, induces a discrepancy: at the process level, the rule places the empty output in sequential composition with P, whereas at the level of typing $x : \mathbf{1}$ is placed in its own partition, enforcing independence from all names in $\mathsf{fn}(P)$. Disentanglement solves this discrepancy. Consider, e.g., process $P_{18} = (\nu\, xx')(x[\,].x'().\mathbf{0})$: we have $P_{18} \rightsquigarrow^* (\nu\, xx')(x[\,].\mathbf{0} \mid x'().\mathbf{0})$.

▶ **Lemma 43** (Disentanglement [23])**.** *If there exists a derivation d of $P \vdash_{\mathtt{H}} \Gamma_1 \parallel \cdots \parallel \Gamma_n$, then there exist derivations $d_1, \ldots, d_n$ of $P_1 \vdash_{\mathtt{C}} \Gamma_1, \ldots, P_n \vdash_{\mathtt{C}} \Gamma_n$ in* CP*, and*

$$
P \vdash_{\mathtt{H}} \Gamma_1 \parallel \cdots \parallel \Gamma_n \quad \rightsquigarrow^* \quad \dfrac{P_1 \vdash_{\mathtt{C}} \Gamma_1 \;\cdots\; P_n \vdash_{\mathtt{C}} \Gamma_n}{P_1 \mid \cdots \mid P_n \vdash_{\mathtt{H}} \Gamma_1 \parallel \cdots \parallel \Gamma_n}\ \textit{H-Mix}_2
$$

where we use the double line to indicate multiple uses of the same rule.

The following result ensures that process observations are invariant under disentanglement. Moreover, since instances of H-Cut are turned into instances of C-Cut, disentangled processes do not feature occurrences of restriction between sessions in sequential composition.

▶ **Lemma 44.** *Suppose $P \in$ HCP. If $P \rightsquigarrow^* P'$ then $P \approx_{\mathtt{H}} P'$.*

Proof. We prove that if $P \rightsquigarrow^* P'$ then $P \triangleq P'$ by induction on the length k of $\rightsquigarrow^*$. For the base case ($k = 1$) we proceed by a case analysis in the last rule applied in the type derivation of $P \vdash_{\mathtt{H}} \mathcal{G}$. The fact that $P \approx_{\mathtt{H}} P'$ follows from Theorem 27. ◀

Given a process P, the rules for disentanglement can be applied in any order; we therefore define a set of disentangled processes:

▶ **Definition 45** (HCP: Disentangled Processes)**.** *Given $P \in$ HCP, the set $\textsc{Dist}(P)$ is defined as follows:*

$$
\textsc{Dist}(P) \triangleq \left\{ P' \in \mathsf{HCP} \;\middle|\; \begin{array}{c} P \rightsquigarrow^* P', \\ P' = P_1 \mid \cdots \mid P_n \vdash_{\mathtt{H}} \Gamma_1 \parallel \cdots \parallel \Gamma_n, \quad P_1 \vdash_{\mathtt{C}} \Gamma_1, \;\cdots\; , P_n \vdash_{\mathtt{C}} \Gamma_n \end{array} \right\}
$$

From Lemma 44 we have that all disentangled processes are observationally equivalent:

▶ **Lemma 46.** *Let $P \in$ HCP. If $Q, R \in \textsc{Dist}(P)$, then $Q \approx_{\mathtt{H}} R$.*

Proof. From Lemma 44, $P \approx_{\mathtt{H}} P'$ for all $P' \in \textsc{Dist}(P)$. By Definition 19, $\approx_{\mathtt{H}}$ is transitive. Hence, for all $Q, R \in \textsc{Dist}(P)$, $Q \approx_{\mathtt{H}} R$. ◀

We use disentanglement to prove that if $\mathcal{D}_{\mathbf{H}}(P)$ then there is a P' s.t. $\mathcal{D}^{\mathsf{R}}_{\mathbf{H}}(P')$, with $P \approx_{\mathtt{H}} P'$.

The following main result establishes that the notions of $\mathcal{D}_{\mathbf{H}}(\cdot)$ (Definition 26) and $\mathcal{D}^{\mathsf{R}}_{\mathbf{H}}(\cdot)$ (Definition 36) are equivalent, up to commuting conversions and disentanglement.

▶ **Theorem 47.** *For all processes $P \in \mathsf{HCP}$ s.t. $\mathcal{D}_{\mathbf{H}}(P)$, then there exist $P', Q \in \mathsf{HCP}$ s.t.:* (1) $P' \in \textit{Dist}(P)$, (2) $P' \simeq_{cc} Q$, (3) $P \approx_{\mathtt{H}} Q$, (4) $\mathcal{D}^{\mathsf{R}}_{\mathbf{H}}(Q)$.

Proof sketch. Parts (1)-(3) follow from Definitions 40 and 45 and Lemma 44. For part (4) we show that self-synchronizations and delayed actions are eliminated by means of disentanglement and commuting conversions, respectively. See [19] for details. ◀

Theorem 47 concerns HCP processes. To lift this result to SP, we need some results.

▶ **Lemma 48.** *Let $P \in \mathbb{H}$. There exists $P' \in \mathbb{H}$ s.t.:* (1) $[\![P]\!] \rightsquigarrow^* [\![P']\!]$. (2) $[\![P]\!] \simeq^*_{cc} [\![P']\!]$.

Proof. The items follow by induction on the length of $\rightsquigarrow^*$ and $\simeq^*_{cc}$, respectively. For (1) in the base cases we proceed by a case analysis in the last rule applied in the type derivation of $[\![P]\!] \vdash_{\mathtt{H}} [\![\Gamma]\!]$; it shows the structure of $[\![P]\!]$ and by Definition 28 we obtain the structure of P. The proof finishes by applying a single step of $\rightsquigarrow$ to the type derivation of $[\![P]\!] \vdash_{\mathtt{H}} [\![\Gamma]\!]$ and replicating it to the derivation of $\Gamma \vdash_{\mathtt{S}} P$. The proof of (2) proceeds similarly. ◀

Lemma 48 ensures that the following definition is sound:

▶ **Definition 49** (Lifting Commuting Conversions and Disentanglement to SP). *Let $P, P' \in \mathbb{H}$. We write $P \rightsquigarrow^* P'$ and $P \simeq^*_{cc} P'$, whenever $[\![P]\!] \rightsquigarrow^* [\![P']\!]$ and $[\![P]\!] \simeq^*_{cc} [\![P']\!]$ hold, respectively.*

Having disentanglement in SP, we aim to prove that the resulting disentangled process *simulates* the original process. To define simulation in SP, we need the following notion:

▶ **Definition 50** (SP: Labelled Reduction Semantics). *Let $P \in \mathsf{SP}$, then $P \overset{\tau:xy}{\Longrightarrow} Q$ whenever:* (1) $P \equiv (\nu\, xy)(\overline{x}\langle v\rangle.P' \mid y(z).Q' \mid R) \longrightarrow (\nu\, xy)(P' \mid Q'\{^v/_z\} \mid R) \equiv Q$, *or* (2) $P \equiv (\nu\, xy)(x().P' \mid \overline{y}\langle\rangle.Q' \mid R) \longrightarrow (\nu\, xy)(P' \mid Q' \mid R) \equiv Q$.

▶ **Definition 51** (SP: Simulation). *A binary relation $\mathbb{S}$ in SP is a (typed) simulation if whenever $(P, Q) \in \mathbb{S}$ then $\Gamma \vdash_{\mathtt{S}} P$, $\Gamma \vdash_{\mathtt{S}} Q$ and $P \overset{\tau:xy}{\Longrightarrow} P'$ implies $Q \overset{\tau:xy}{\Longrightarrow} Q'$ for some Q' s.t. $(P', Q') \in \mathbb{S}$. We write $P \lesssim Q$ whenever $(P, Q) \in \mathbb{S}$ for a (typed) simulation $\mathbb{S}$.*

In the LTS semantics of HCP, the processes P and Q from Theorem 47 are bisimilar; however, in a reduction semantics, any transitions of P that use delayed actions or self-synchronization would be considered as deadlocks, and then Q would only simulate P. Formally:

▶ **Theorem 52.** *The relation $\mathbb{S} = \{(P, R) \mid P \in \mathbb{H}, \text{and } P \rightsquigarrow R\}$ is a simulation.*

Proof sketch. By induction in the structure of P, with a case analysis in $\rightsquigarrow$. See [19] for details. ◀

Finally, Corollary 53, given below, brings Theorem 47 to SP: it shows that even though in HCP both $\mathcal{D}_{\mathbf{H}}(\cdot)$ and $\mathcal{D}^{\mathsf{R}}_{\mathbf{H}}(\cdot)$ are equivalent, in SP those notions are related via a simulation.

▶ **Corollary 53.** *Given $P \in \mathbb{H}_{\mathsf{F}}$, then there exist $Q \in \mathbb{H}$ and $P' \in \mathbb{H}_{\mathsf{R}}$ s.t.:* (1) $[\![Q]\!] \in \textit{Dist}([\![P]\!])$, (2) $[\![Q]\!] \simeq^*_{cc} [\![P']\!]$, (3) $P \lesssim P'$, (4) $[\![P]\!] \approx_{\mathtt{H}} [\![P']\!]$.

Proof. It follows by Lemma 48 and Theorems 47 and 52. ◀

Table 1 Inventory of type systems, classes of processes, and encodings considered in the paper. Shaded cells concern elements used in the case of asynchronous processes (§ 4).

SP	Definition 2	$\mathbb{C}$	Definition 34	$[\![\cdot]\!] : \mathsf{SP} \to \mathsf{(H)CP}$	Definition 28
CP	Definition 11	$\mathbb{H}$	Definition 34		
HCP	Definition 12	$\mathbb{H}_\mathsf{F}$, $\mathbb{H}_\mathsf{R}$	Definition 37		
P	Definition 58	$\mathbb{P}$	Definition 67	$\{\!\![\cdot]\!\!\}^f : \mathsf{SP} \to \mathsf{P}$	Definition 65
μP	Definition 62	$\mu\mathbb{P}$	Definition 67	$\langle\!\langle \cdot \rangle\!\rangle : \mathsf{SP}^\mathsf{a} \to \mathsf{SP}$	Definition 82
SP^a	Definition 78	$\mathbb{C}^\mathsf{a}$, $\mathbb{H}^\mathsf{a}$	Definition 86		

▶ **Example 54** (Corollary 53, At Work)**.** Consider the process P_{54} and its encoding into HCP:

$$P_{54} \triangleq (\nu\, wz)(\overline{w}\langle\rangle.(\nu\, xy)(\overline{x}\langle\rangle.\mathbf{0} \mid (\nu\, ku)(k().\mathbf{0} \mid y().z().\overline{u}\langle\rangle.\mathbf{0})))$$
$$[\![P_{54}]\!] \triangleq (\nu\, wz)(w[].(\nu\, xy)(x[].\mathbf{0} \mid (\nu\, ku)(k().\mathbf{0} \mid y().z().u[].\mathbf{0})))$$

Note that $\neg\mathcal{D}_\mathbf{S}(P_{54})$ but $[\![P_{54}]\!]$ has two τ-transitions:

$$[\![P_{54}]\!] \xrightarrow{\tau} (\nu\, xy)(x[].\mathbf{0} \mid (\nu\, ku)(k().\mathbf{0} \mid y().u[].\mathbf{0})) \to^* \equiv_\mathtt{H} \mathbf{0} \tag{1}$$
$$[\![P_{54}]\!] \xrightarrow{\tau} (\nu\, wz)(w[].(\nu\, ku)(k().\mathbf{0} \mid z().u[].\mathbf{0})) \to^* \equiv_\mathtt{H} \mathbf{0} \tag{2}$$

where the transition in (1) is due to a delayed action on z and a self-synchronization along the session wz, and the transition in (2) is due to a delayed τ-transition along the session xy. Hence, $\mathcal{D}_\mathbf{H}([\![P_{54}]\!])$. Via disentanglement we obtain:

$$[\![P_{54}]\!] \rightsquigarrow (\nu\, wz)(w[].\mathbf{0} \mid (\nu\, xy)(x[].\mathbf{0} \mid (\nu\, ku)(k().\mathbf{0} \mid y().z().u[].\mathbf{0}))) \triangleq R'$$

In R', neither a delayed action nor a self-synchronization are needed, thus $\mathcal{D}^\mathsf{R}_\mathbf{H}(R')$. We could also give priority to the τ-transition along wz; however, because the input on z is not at the top-level, a synchronization on wz still requires a delayed action. This shows the need for commuting conversions. Indeed, by using $\simeq_{cc}$ we obtain:

$$R' \simeq_{cc} (\nu\, wz)(w[].\mathbf{0} \mid (\nu\, xy)(x[].\mathbf{0} \mid (\nu\, ku)(k().\mathbf{0} \mid z().y().u[].\mathbf{0}))) \triangleq R$$

We have that $\mathcal{D}^\mathsf{R}_\mathbf{H}(R)$. Moreover, there is a process P' in SP s.t. $[\![P']\!] = R$:

$$P' \triangleq (\nu\, wz)(\overline{w}\langle\rangle.\mathbf{0} \mid (\nu\, xy)(\overline{x}\langle\rangle.\mathbf{0} \mid (\nu\, ku)(k().\mathbf{0} \mid z().y().\overline{u}\langle\rangle.\mathbf{0})))$$

Notice that $P_{54} \lesssim P'$ follows trivially since $\mathcal{D}_\mathbf{S}(P')$.

4 The Case of Asynchronous Processes

Up to here we have addressed **(Q1)** by considering the relation between $\mathbb{H}_\mathsf{F}$ and $\mathbb{H}_\mathsf{R}$ in HCP. We now turn our attention to **(Q2)** and consider the interplay of DF and *asynchronous processes* as typable by Padovani's type system for asynchronous processes, here dubbed P [27].

We proceed as follows. In § 4.1 we recall Padovani's type system and introduce $\mathbb{P}$, the class of SP processes induced by P, and its sub-class $\mu\mathbb{P}$, derived from the type system μP. In § 4.2, we show that $\mu\mathbb{P} = \mathbb{C}$ (Corollary 74). In § 4.3, we establish a key result: the precise relations between $\mathbb{H}$, $\mathbb{H}_\mathsf{F}$, $\mathbb{H}_\mathsf{R}$, and $\mathbb{P}$ (Theorem 76). Also, we revisit Corollary 53 from the standpoint of P: we prove that the process $P' \in \mathbb{H}_\mathsf{R}$ (obtained via commuting conversions and disentanglement) is also in $\mathbb{P}$ (Corollary 77). In § 4.4 we present SP^a, an asynchronous version of SP, and show how to extend our results for SP to SP^a (Corollary 88).

Table 1 summarizes all technical ingredients, including those introduced in this section.

4.1 The Type System P

We summarize the linear fragment of the type system P by Padovani [27], which streamlines Kobayashi's type system for deadlock freedom [21]. We assume sets of channels $a, b, \ldots$ and variables $x, y, \ldots$; then, names $u, v, \ldots$ are either channels or variables. We find it convenient to use processes with polyadic communication:

$$P ::= u!\langle x_1, \ldots, x_n\rangle \mid u?(x_1, \ldots, x_n).P \mid P \mid Q \mid (\nu\, a)P \mid \mathbf{0}$$

Two salient features are the output $u!\langle x_1, \ldots, x_n\rangle$, which does not have a continuation, and "$(\nu\, a)P$", the standard restriction construct of the π-calculus. Notions of free and bound names are as usual: in the process $u?(x_1, \ldots, x_n).P$, $x_1, \ldots, x_n$ are bound in P; in $(\nu\, a)P$, a is bound in P. We write $\mathsf{fn}(P)$ and $\mathsf{bn}(P)$ for the free and bound names of P, respectively.

The reduction semantics is closed under a structural congruence, denoted $\equiv_{\mathsf{P}}$, which is standardly defined by the following axioms: (i) $P \mid Q \equiv_{\mathsf{P}} Q \mid P$, (ii) $P \mid \mathbf{0} \equiv_{\mathsf{P}} P$, (iii) $P \mid (Q \mid R) \equiv_{\mathsf{P}} (P \mid Q) \mid R$, (iv) $(\nu\, x)P \mid Q \equiv_{\mathsf{P}} (\nu\, x)(P \mid Q)$ (if $x \notin \mathsf{fn}(Q)$), and (v) $(\nu\, x)(\nu\, y) \equiv_{\mathsf{P}} (\nu\, y)(\nu\, x)P$.

Reduction is then defined as

$$a!\langle v_1, \ldots, v_n\rangle \mid a?(x_1, \ldots, x_n).P \longrightarrow_{\mathsf{P}} P\{v_1, \ldots, v_n/x_1, \ldots, x_n\}$$

▶ **Example 55.** Differently from SP, channels in Padovani's framework are used exactly once. Structured communications can of course be still expressed, using continuations. Consider the process $P_{55} \triangleq (\nu\, x)(Q_{55} \mid R_{55})$ with

$$Q_{55} \triangleq (\nu\, a)(\nu\, c)(x!\langle a, c\rangle \mid c!\langle a\rangle) \qquad R_{55} \triangleq x?(z_1, w).w?(z_2).((\nu\, r)z_1!\langle r\rangle \mid z_2?(w).\mathbf{0})$$

Q_{55} sends the two endpoints of a channel: the first is sent along x (together with a continuation channel c), and the other is sent along c. Channel a occurs in both outputs, but these occurrences actually denote different endpoints; this difference will be justified by typing (cf. Example 59). R_{55} receives these endpoints to implement a local exchange of name r along a.

Padovani's type system ensures the *absence of pending communications* in a process with respect to a channel. To formalize this notion, we rely on reduction contexts, defined as $\mathcal{C} ::= [\circ] \mid (\mathcal{C} \mid P) \mid (\nu\, x)\mathcal{C}$. We may now define the following auxiliary predicates:

$$\mathsf{out}(a, P) \;\Leftrightarrow\; P \equiv_{\mathsf{P}} \mathcal{C}[a!\langle e\rangle] \;\wedge\; a \notin \mathsf{bn}(C) \qquad \mathsf{in}(a, P) \;\Leftrightarrow\; P \equiv_{\mathsf{P}} \mathcal{C}[a?(x).Q] \;\wedge\; a \notin \mathsf{bn}(C)$$
$$\mathsf{sync}(a, P) \;\Leftrightarrow\; (\mathsf{in}(a, P) \wedge\; \mathsf{out}(a, P))$$

Intuitively, predicate $\mathsf{in}(a, P)$ holds if a is free in P and there is a subprocess of P that can perform a linear input on a. Predicate $\mathsf{out}(a, P)$ holds if a is free in P and a subprocess of P is waiting to send a value v. Predicate $\mathsf{sync}(a, P)$ denotes a pending input/output on a for which a synchronization on a is immediately available. This way, to formalize a pending input/output for which a synchronization on a is not immediately possible, we define:

$$\mathsf{wait}(a, P) \;\Leftrightarrow\; (\mathsf{in}(a, P) \;\vee\; \mathsf{out}(a, P)) \;\wedge\; \neg\mathsf{sync}(a, P)$$

Thus, ensuring no pending communications in P means ensuring $\neg\mathsf{wait}(a, P)$ for every a.

▶ **Example 56** (Predicates at work)**.** Consider the process $P_{56} \triangleq a?(x).b!\langle v_1\rangle \mid b?(y).a!\langle v_2\rangle$, which has a circular dependency. We have: (1) $\mathsf{out}(a, P_{56})$ and $\mathsf{out}(b, P_{56})$. (2) $\mathsf{in}(a, P_{56})$ and $\mathsf{in}(b, P_{56})$. (3) $\neg\mathsf{sync}(a, P_{56})$ and $\neg\mathsf{sync}(b, P_{56})$. (4) $\mathsf{wait}(a, P_{56})$ and $\mathsf{wait}(b, P_{56})$.

Type System. To ensure absence of pending communications, P relies on *priorities*. Let $p, q, \ldots$ denote subsets of $\{?, !\}$ (*polarities*). Types for channels are defined as follows:

$$t ::= p[t_1, \ldots, t_m]^n$$

where $n \in \mathbb{Z}$ is the priority of a channel, which indicates urgency for its use: the lower the number, the higher the urgency. We write $\#$ for $\{?, !\}$. This way, $\#$ and $\emptyset$ are even polarities.

▶ **Example 57** (Priorities)**.** To illustrate types and priorities, recall again the process $P_{56} = a?(x).b!\langle v_1\rangle \mid b?(y).a!\langle v_2\rangle$. As we have seen, we have $\mathsf{wait}(a, P_{56})$ and $\mathsf{wait}(b, P_{56})$.

Let us assume that x, y, v_1, v_2 are all typed with some type t. By inspecting the left thread of P_{56}, we infer that a and b must be typed as $?[t]^m$ and $![t]^n$, respectively, for some m, n. Then, by inspecting the right thread, we infer that a and b must then be typed with $![t]^m$ and $?[t]^n$, respectively. This way, their types in the composed process are $\#[t]^m$ (for a) and $\#[t]^n$ (for b). Looking again at the left thread, as the input along a is blocking the output along b, we must have $m < n$; in the right thread we have the exact opposite situation, thus $n < m$. These unsolvable inequalities reveal the circular dependency between a and b, which will make P_{56} not typable in the type system P.

We now summarize how typing excludes pending communications. Type environments $\Gamma, \Gamma', \ldots$ are finite maps from channels to types. We write $dom(\Gamma)$ for the domain of Γ. We write Γ, Γ' for the union of Γ and Γ' when $dom(\Gamma) \cap dom(\Gamma') = \emptyset$. We write $+$ to denote a partial composition operator on types defined as: $p[s_1, \ldots, s_m]^n + q[s_1, \ldots, s_m]^n = (p \cup q)[s_1, \ldots, s_m]^n$ if $p \cap q = \emptyset$, and undefined otherwise. For type environments we have: $\Gamma + \Gamma' \triangleq \Gamma, \Gamma'$ if $dom(\Gamma) \cap dom(\Gamma') = \emptyset$, and $(\Gamma, u : t) + (\Gamma', u : s) \triangleq (\Gamma + \Gamma'), u : t + s$. Let $\mathbb{Z}_\top = \mathbb{Z} \cup \{\top\}$, where $n < \top$ for every $n \in \mathbb{Z}$. The functions $|\cdot|$, from types to $\mathbb{Z}_\top$, and $\$^h$ from types to types are defined as follows:

$$|t| \triangleq \begin{cases} n & \text{if } t = p[s]^n \text{and } p \neq \emptyset \\ \top & \text{otherwise} \end{cases} \qquad \$^h t \triangleq \begin{cases} p[s]^{n+h} & \text{if } t = p[s]^n \text{and } p \neq \emptyset \\ t & \text{otherwise} \end{cases}$$

Intuitively, the function $\$^h$ allows us to shift priorities. We say that a type t is *unlimited*, written $\mathsf{un}(t)$, iff $|t| = \top$. We write $\mathsf{un}(\Gamma)$ iff all the types in the range of Γ are unlimited. We are now ready to give the typing rules, considering only processes with biadic communication, for simplicity. The rules for monadic communication are as expected.

▶ **Definition 58** (P)**.** *The typing rules for processes use judgements of the form* $\Gamma \vdash_{\mathrm{P}} P$ *and* $\Gamma \vdash_{\mathrm{P}} \widetilde{e} : \widetilde{t}$ *and are presented in Figure 9. We shall write* $P \in \mathsf{P}$ *if* $\Gamma \vdash_{\mathrm{P}} P$*, for some* Γ*.*

Rule T-In is used to type inputs $u?(x_1, x_2).P$, where u has type $?[t_1, t_2]^n$. The condition $n < |\Gamma|$ ensures that u has been assigned the highest urgency; accordingly, the types of x and y are shifted by n. Rule T-Out, used to type outputs on channels of type $![t_1, t_2]^n$, follows a similar principle: the types of x and y are shifted by n, since they must have lower urgency than u. Rule T-New is used for restricting new channels with full ($\#$) or empty ($\emptyset$) polarity.

▶ **Example 59.** Recall processes Q_{55} and R_{55} from Example 55. Consider types $s' = \$^1![\emptyset[]^4]^2$ and $s = \$^2?[\emptyset[]^4]^1$. Note that $s' + s = \#[\emptyset[]^4]^3$. Process Q_{55} is typed as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{a : s', c : \$^1?[s]^1 \vdash_{\mathrm{P}} a : s', c : \$^1?[s]^1}{a : s', c : \$^1?[s]^1, x :![s', \$^1?[s]^1]^1 \vdash_{\mathrm{P}} x!\langle a, c\rangle}\ \text{T-Out} \quad \dfrac{a : s \vdash_{\mathrm{P}} a : s}{a : s, c :![s]^2 \vdash_{\mathrm{P}} c!\langle a\rangle}\ \text{T-Out}}{a : \#[\emptyset[]^4]^3, x :![s', \$^1?[s]^1]^1, c : \#[s]^2 \vdash_{\mathrm{P}} x!\langle a, c\rangle \mid c!\langle a\rangle}\ \text{T-Par}}{a : \#[\emptyset[]^4]^3, x :![s', \$^1?[s]^1]^1 \vdash_{\mathrm{P}} (\nu\, c)(x!\langle a, c\rangle \mid c!\langle a\rangle)}\ \text{T-New}}{x :![s', \$^1?[s]^1]^1 \vdash_{\mathrm{P}} (\nu\, a)(\nu\, c)(x!\langle a, c\rangle \mid c!\langle a\rangle)}\ \text{T-New}$$

$$\text{T-Idle}\ \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash_{\mathrm{P}} \mathbf{0}} \qquad \text{T-Par}\ \frac{\Gamma_1 \vdash_{\mathrm{P}} P_1 \quad \Gamma_2 \vdash_{\mathrm{P}} P_2}{\Gamma_1 + \Gamma_2 \vdash_{\mathrm{P}} P_1 \mid P_2} \qquad \text{T-out}\ \frac{\Gamma, e_1 : \$^n t_1, e_2 : \$^n t_2 \vdash_{\mathrm{P}} e_1 : \$^n t_1, e_2 : \$^n t_2 \quad \mathsf{un}(\Gamma)}{\Gamma + u :![t_1,t_2]^n \vdash_{\mathrm{P}} u!\langle e_1, e_2\rangle}$$

$$\text{T-In}\ \frac{\Gamma, x : \$^n t_1, y : \$^n t_2 \vdash_{\mathrm{P}} P \qquad n < |\Gamma|}{\Gamma + u :?[t_1,t_2]^n \vdash_{\mathrm{P}} u?(x,y).P} \qquad \text{T-New}\ \frac{\Gamma, a : p[t]^n \vdash_{\mathrm{P}} P \qquad p \text{ even}}{\Gamma \vdash_{\mathrm{P}} (\nu\, a)P}$$

Figure 9 P: Typing rules for asynchronous processes.

$$\text{M-Idle}\ \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash^{\mu}_{\mathrm{P}} \mathbf{0}} \qquad \text{M-Out}\ \frac{\Gamma, e_1 : \$^n t_1, e_2 : \$^n t_2 \vdash^{\mu}_{\mathrm{P}} e_1 : \$^n t_1, e_2 : \$^n t_2 \quad \mathsf{un}(\Gamma)}{\Gamma + u :![t_1,t_2]^n \vdash^{\mu}_{\mathrm{P}} u!\langle e_1, e_2\rangle}$$

$$\text{M-In}\ \frac{\Gamma, x : \$^n t_1, y : \$^n t_2 \vdash^{\mu}_{\mathrm{P}} P \qquad n < |\Gamma|}{\Gamma + u :?[t_1,t_2]^n \vdash^{\mu}_{\mathrm{P}} u?(x,y).P}$$

$$\text{M-New-Par}\ \frac{\Gamma_1, a : p[t]^n \vdash^{\mu}_{\mathrm{P}} P \qquad \Gamma_2, a : q[t]^n \vdash^{\mu}_{\mathrm{P}} Q \qquad dom(\Gamma_1) \cap dom(\Gamma_2) = \emptyset,\ p \cup q \text{ even}}{\Gamma_1 + \Gamma_2 \vdash^{\mu}_{\mathrm{P}} (\nu\, a)(P \mid Q)}$$

Figure 10 μP: Modified typing rules for asynchronous processes.

Similarly we can check that $x :?[s', \$^1?[s]^1]^1 \vdash_{\mathrm{P}} R_{55}$. Note that x is typed with opposite polarities in Q_{55} and R_{55}; thus by applying Rules T-Par and T-New we obtain $\vdash_{\mathrm{P}} P_{55}$.

We may now characterize deadlock freedom. Let us write $\longrightarrow^*_{\mathrm{P}}$ to denote the reflexive, transitive closure of $\longrightarrow_{\mathrm{P}}$. Also, write $P \not\longrightarrow_{\mathrm{P}}$ if there is no Q such that $P \longrightarrow_{\mathrm{P}} Q$. We have:

▶ **Definition 60** (P: Deadlock Freedom [27]). *We say that $P \in \mathsf{P}$ is deadlock free, written $\mathcal{D}_{\mathbf{P}}(P)$, if for every Q such that $P \longrightarrow^*_{\mathrm{P}} (\nu\, \tilde{a})Q \not\longrightarrow_{\mathrm{P}}$ we have $\neg\mathsf{wait}(a, Q)$ for every a.*

We have that processes typed in the empty environment are deadlock free:

▶ **Theorem 61** (P: Deadlock Freedom [27]). *If $\vdash_{\mathrm{P}} P$, then $\mathcal{D}_{\mathbf{P}}(P)$.*

4.2 Relating $\mathbb{P}$ and $\mathbb{C}$

We now consider μP ("micro P"), which is a fragment of P that is similar to CP. We also define the encoding $\{\!|\cdot|\!\}^f : \mathsf{SP} \to \mathsf{P}$, and $\mathbb{P}$ and $\mu\mathbb{P}$ the classes of SP processes that are captured by P and μP, respectively. These are all ingredients for our results in § 4.3.

▶ **Definition 62** (μP). *The typing rules for processes are presented in Figure 10. A judgement $\Gamma \vdash^{\mu}_{\mathrm{P}} P$ denotes that P is well typed in Γ. We shall write $P \in \mu\mathsf{P}$ if $\Gamma \vdash^{\mu}_{\mathrm{P}} P$, for some Γ.*

μP results from P by replacing Rules T-Par and T-New in Figure 9 with Rule M-New-Par, which combines restriction and parallel just as Rule C-Cut in Figure 4.

▶ **Lemma 63.** *If $\Gamma \vdash^{\mu}_{\mathrm{P}} P$ then $\Gamma \vdash_{\mathrm{P}} P$.*

Proof sketch. By induction on the type derivation, with a case analysis in the last rule applied. See [19] for details. ◀

▶ **Corollary 64.** *If* $\vdash^{\mu}_{\mathsf{P}} P$, *then* $\mathcal{D}_{\mathbf{P}}(P)$.

Proof. It follows by Lemma 63 and Theorem 61. ◀

We now define an encoding from SP to P, following a continuation-passing style (cf. [8]). Let f be a function from names to names. We use the following conventions: f_x stands for $f(x)$, and $f, \{x \mapsto c\}$ stands for the function f' s.t. $f'(x) = c$ and $f'(y) = f(y)$ if $x \neq y$.

▶ **Definition 65.** *The (partial) encoding* $\{\![\,\cdot\,]\!\}^f : \mathsf{SP} \to \mathsf{P}$ *is parameterized by a function, ranging over* f, g, *from names to names, and is defined for processes as follows:*

$$
\begin{aligned}
\{\![\overline{x}\langle v\rangle.P]\!\}^f &\triangleq (\nu\, m)(f_x!\langle f_v, m\rangle \mid m?(k).(\nu\, c)(k!\langle c\rangle \mid \{\![P]\!\}^{f,\{x\mapsto c\}})) \\
\{\![x(z).P]\!\}^f &\triangleq f_x?(z,m).(\nu\, a)(m!\langle a\rangle \mid a?(c).\{\![P]\!\}^{f,\{x\mapsto c\}}) \\
\{\![\overline{x}\langle\,\rangle.\mathbf{0}]\!\}^f &\triangleq (\nu\, r) f_x!\langle r\rangle \\
\{\![x().P]\!\}^f &\triangleq f_x?(r).\{\![P]\!\}^f \qquad \textit{(with } r \notin \mathtt{fv}(P) \cup \mathtt{bv}(P)\textit{)} \\
\{\![(\nu\, xy)P]\!\}^f &\triangleq (\nu\, d)\{\![P]\!\}^{f,\{x,y\mapsto d\}}
\end{aligned}
$$

and as a homomorphism for $P \mid Q$ *and* $\mathbf{0}$. *The encoding of session types is as follows:*

$$
\begin{aligned}
\{\![\mathsf{end}_?]\!\} &\triangleq ?[\emptyset[]]^n & \{\![!S.T]\!\} &\triangleq ?[\{\![T]\!\}, ![![\{\![S]\!\}]^i]^j]^n \\
\{\![\mathsf{end}_!]\!\} &\triangleq ![\emptyset[]]^n & \{\![?S.T]\!\} &\triangleq ![\{\![T]\!\}, ![![\{\![\overline{S}]\!\}]^i]^j]^n
\end{aligned}
$$

Typing environments are encoded inductively as $\{\![\emptyset]\!\}^f \triangleq \emptyset$ *and* $\{\![\Gamma, x : T]\!\}^f \triangleq \{\![\Gamma]\!\}^f, f_x : \{\![T]\!\}$. *The values of* i, j *and* n *are calculated when checking in* $\mu\mathsf{P}$ *the typablity of* $\{\![\Gamma]\!\}^f \vdash^{\mu}_{\mathsf{P}} \{\![P]\!\}^f$.

Intuitively, the encoding of processes in Definition 65 collapses the two endpoints of a session into a single name in P. The encoding of $\overline{x}\langle v\rangle.P$ mimics both the sequential structure of a session and synchronous communication. $\{\![\overline{x}\langle v\rangle.P]\!\}^f$ sends the (renamed) value f_v and a new channel m along which an input $m?(k).$ blocks the continuation $\{\![P]\!\}^{f,\{x\mapsto c\}}$, and receives the channel along which the the continuation c of the session x will be sent. Dually, $\{\![x(z).P]\!\}^f$ receives the intended value and the channel m, which sends the channel a, along which the continuation of the session x will be received. Finally, the encoding is partial because $\{\![\overline{x}\langle\,\rangle.P]\!\}^f$ is only defined when $P = \mathbf{0}$: this allows us to characterize the session processes captured in CP, where the empty output has no continuation (see Definition 11).

▶ **Example 66.** Consider the process $P_{66} \triangleq (\nu\, xy)(\overline{x}\langle v\rangle.R \mid y(z).Q)$ in SP. We show $\{\![P_{66}]\!\}^f$ and illustrate its reductions. Let us assume P_{66} is well-typed. Because of linearity of x and y, they only appear in the left and right subprocess, respectively. Let $f = \{x, y \mapsto d\}$ and $g_1 = \{x \mapsto c\}$,$g_2 = \{y \mapsto c'\}$, then we have:

$$
\begin{aligned}
\{\![P_{66}]\!\}^{\emptyset} &= (\nu\, d)(\{\![\overline{x}\langle v\rangle.R]\!\}^f \mid \{\![y(z).Q]\!\}^f) \\
&= (\nu\, d)((\nu\, m)(d!\langle v, m\rangle \mid m?(k).(\nu\, c)(k!\langle c\rangle \mid \{\![R]\!\}^{g_1})) \\
&\qquad\qquad \mid d?(z,m').(\nu\, a)(m'!\langle a\rangle \mid a?(c').\{\![Q]\!\}^{g_2})) \\
&\longrightarrow_{\mathsf{P}} (\nu\, m)(m?(k).(\nu\, c)(k!\langle c\rangle \mid \{\![R]\!\}^{g_1}) \mid (\nu\, a)(m!\langle a\rangle \mid a?(c').\{\![Q]\!\}^{g_2}\{v/z\})) \\
&\longrightarrow_{\mathsf{P}} (\nu\, a)((\nu\, c)(a!\langle c\rangle \mid \{\![R]\!\}^{g_1}) \mid a?(c').\{\![Q]\!\}^{g_2}\{v/z\}) \\
&\longrightarrow_{\mathsf{P}} (\nu\, c)(\{\![R]\!\}^{g_1} \mid \{\![Q]\!\}^{g_2}\{v/z, c/c'\})
\end{aligned}
$$

Our main result concerns the classes of processes associated to P and $\mu\mathsf{P}$, which we define by mirroring the definition of $\mathbb{C}$ and $\mathbb{H}$ (Definition 34):

▶ **Definition 67** ($\mathbb{P}$ and $\mu\mathbb{P}$). *The classes of processes $\mathbb{P}$ and $\mu\mathbb{P}$ are defined as follows:*

$$\mathbb{P} \triangleq \{P \in \mathsf{SP} \mid \Gamma \vdash_{\mathsf{S}} P \wedge \{\!|\Gamma|\!\}^f \vdash_{\mathsf{P}} \{\!|P|\!\}^f\} \qquad \mu\mathbb{P} \triangleq \{P \in \mathsf{SP} \mid \Gamma \vdash_{\mathsf{S}} P \wedge \{\!|\Gamma|\!\}^f \vdash^{\mu}_{\mathsf{P}} \{\!|P|\!\}^f\}$$

The following result establishes an operational correspondence between a process and its encoding, what will allows us to capture deadlock free session processes via the encoding.

▶ **Theorem 68** (Operational Correspondence for $\{\!|\cdot|\!\}^f$). *Let $P \in \mu\mathbb{P}$. We have:*

1. **a.** *If $P \equiv (\nu\,\widetilde{xy})(\overline{x}\langle\rangle.Q_1 \mid y().Q_2 \mid S)$, then $\exists Q \in \mathsf{P}$ such that $\{\!|P|\!\}^f \longrightarrow_{\mathsf{P}} Q$ and $Q \equiv_{\mathsf{P}} \{\!|(\nu\,\widetilde{xy})(Q_1 \mid Q_2 \mid S)|\!\}^f$.*
   b. *If $P \equiv (\nu\,\widetilde{xy})(\overline{x}\langle v\rangle.Q_1 \mid y(z).Q_2 \mid S)$, then $\exists Q \in \mathsf{P}$ such that $\{\!|P|\!\}^f \longrightarrow^3_{\mathsf{P}} Q$ and $Q \equiv_{\mathsf{P}} \{\!|(\nu\,\widetilde{xy})(Q_1 \mid Q_2\{v/z\} \mid S)|\!\}^f$.*
2. *If $\{\!|P|\!\}^f \longrightarrow_{\mathsf{P}} R$, then either:*
   a. *$\exists P' \in \mathsf{SP}$ such that $P \longrightarrow P'$ and $R \equiv_{\mathsf{P}} \{\!|P'|\!\}^f$, or*
   b. *$\exists Q \in \mathsf{P}$, and $P' \in \mathsf{SP}$ such that $R \longrightarrow^2_{\mathsf{P}} Q$, $P \longrightarrow P'$ and $Q \equiv_{\mathsf{P}} \{\!|P'|\!\}^f$.*

Proof. By assumption, $P \in \mu\mathbb{P}$ and so we have that $\Gamma \vdash_{\mathsf{S}} P$, for some Γ, and $\{\!|\Gamma|\!\}^f \vdash^{\mu}_{\mathsf{P}} \{\!|P|\!\}^f$ is defined. Moreover, by Corollary 8 P is well-formed. We consider each item separately. Item 1(b) follows by Definition 65 and the rest of the analysis proceeds as in Example 66; Item 1(a) is similar. Item (2): If $\{\!|P|\!\}^f \longrightarrow_{\mathsf{P}} R$, then $\{\!|P|\!\}^f \equiv_{\mathsf{P}} (\nu\,\tilde{a})(d!\langle\tilde{v}\rangle \mid d?(\tilde{x}).K \mid S)$. By typability of P and Definition 65 one of the two possibilities apply:

- **i** $P \equiv (\nu\,\widetilde{xy})(\overline{x}\langle\rangle.\mathbf{0} \mid y().K' \mid S')$, $\{\!|S'|\!\}^f = S$, $\{\!|K'|\!\}^f = K$ and $f = f', \{x, y \mapsto d\}$,
- **ii** $P \equiv (\nu\,\widetilde{xy})(\overline{x}\langle v\rangle.S_1 \mid y(z).K' \mid S_2)$ and $f = f', \{x, y \mapsto d\}$

In case (i) $\{\!|P|\!\}^f \equiv_{\mathsf{P}} (\nu\,\tilde{a})((\nu\,r)d!\langle r\rangle \mid d?(r').K \mid S) \longrightarrow_{\mathsf{P}} (\nu\,\tilde{a})(K \mid S)$ and $P \equiv (\nu\,\widetilde{xy})(\overline{x}\langle\rangle.\mathbf{0} \mid y().K' \mid S') \longrightarrow (\nu\,\widetilde{xy'})(K' \mid S'')$, which verifies case (2a). In case (ii) we proceed as in Example 66, which verifies case (2b). ◀

As a consequence of Theorem 68 we have the following result, which shows the relation between deadlock-free processes in $\mu\mathbb{P}$.

▶ **Corollary 69.** *Let $P \in \mu\mathbb{P}$. $\mathcal{D}_{\mathbf{S}}(P)$ iff $\mathcal{D}_{\mathbf{P}}(\{\!|P|\!\}^f)$.*

Proof. For each direction we prove its contrapositive. We discuss only the right-to-left direction; the analysis for the left-to-right direction is similar. By Definition 60, there exist R and a s.t. $\{\!|P|\!\}^f \longrightarrow^*_{\mathsf{P}} (\nu\,\tilde{a})R \not\longrightarrow_{\mathsf{P}}$ and $\mathsf{wait}(a, R)$. If $\mathsf{wait}(a, R)$ holds because $\mathsf{in}(a, R)$ then $R \equiv_{\mathsf{P}} (\nu\,\tilde{a})(a?(x).Q \mid S)$. By Theorem 68 and Definition 65 there exists $P' \equiv (\nu\,\widetilde{xy})(z(w).Q' \mid S')$ s.t. $P \longrightarrow^* P'$ and $R \equiv_{\mathsf{P}} \{\!|P'|\!\}^f$. Note that $P' \not\longrightarrow$, since otherwise by Theorem 68(1), $(\nu\,\tilde{a})R \longrightarrow_{\mathsf{P}} R''$ for some process R'', contradicting our assumption, thus $\neg\mathcal{D}_{\mathbf{S}}(P)$. The case when $\mathsf{wait}(a, R)$ holds because $\mathsf{out}(a, R)$ proceeds similarly. ◀

Corollary 69 allows us to detect DF session processes via the encoding.

4.3 Comparing $\mathbb{H}$ and $\mathbb{P}$

From § 3 we infer that a process $P \in \mathbb{H}$, typed as $[\![P]\!] \vdash_{\mathtt{H}} \Gamma_1 \parallel \cdots \parallel \Gamma_n$, can be decomposed into processes $P_1 \vdash_{\mathtt{C}} \Gamma_1, \ldots, P_n \vdash_{\mathtt{C}} \Gamma_n$. Moreover, by Lemma 48 (disentanglement in SP) we infer that processes $P_1, \ldots, P_n$ have a pre-image in SP, therefore $\forall i \in [1, n].\, P_i \in \mathbb{C} \subsetneq \mathbb{H}$.

As an important preliminary step, we first prove that $\mathbb{C} = \mu\mathbb{P}$ (Corollary 74). We split the proof in two separate inclusions. The following lemma is useful to prove that $\mathbb{C} \subseteq \mu\mathbb{P}$.

▶ **Lemma 70.** *Let Γ and P be an environment and a process in P, resp. The following hold:*
1. *If $\Gamma \vdash_{\mathsf{P}} P$, then $\$^n\Gamma \vdash_{\mathsf{P}} P$.*
2. *If $\Gamma \vdash^{\mu}_{\mathsf{P}} P$, then $\$^n\Gamma \vdash^{\mu}_{\mathsf{P}} P$.*
3. *Let $P \in \mathsf{P}$. If $\Gamma \vdash^{\mu}_{\mathsf{P}} P$ and $\mathsf{un}(T)$, then $\Gamma, x : T \vdash^{\mu}_{\mathsf{P}} P$, with x fresh wrt P.*
4. *Let $\Gamma_1, \ldots, \Gamma_n$ be typing environments in SP, then $\{\!|\Gamma_1, \cdots, \Gamma_n|\!\}^f = \{\!|\Gamma_1|\!\}^f + \cdots + \{\!|\Gamma_n|\!\}^f$, for some renaming function f.*
5. *Let T be a session type. Then $\{\!|T|\!\}^f + \$^m\{\!|\overline{T}|\!\}^f = p[T']^n$, for some $m \geq 0$, $n \in \mathbb{Z}$, T' a type in P, and p even.*

Proof. (1) Proven in [27]. (2) Follows from Lemma 63 and item 1. (3) Follows by induction on the type derivation. (Figure 10). (4) Follows immediately by linearity of names in Γ_i, for $i \in [1, n]$. (5) Follows by induction on the structure of T. ◀

▶ **Theorem 71.** $\mathbb{C} \subseteq \mu\mathbb{P}$.

Proof sketch. By induction on the type derivation $\Gamma \vdash_{\mathsf{S}} P$ with a case analysis in the last rule applied. See [19] for details. ◀

To prove the other direction we need the following result.

▶ **Lemma 72.** *Let S and T be session types. We have:* (1) $\{\!|S|\!\}^f + \{\!|T|\!\}^f$ *is defined iff* $T = \overline{S}$. (2) $[\![\overline{S}]\!] = ([\![S]\!])^{\perp}$.

Proof. Item (1) follows by Definition 65 and def. of $+$. Item (2) follows immediately by induction in the structure of S and Definition 28. ◀

▶ **Theorem 73.** $\mu\mathbb{P} \subseteq \mathbb{C}$.

Proof sketch. By induction on the type derivation $\Gamma \vdash_{\mathsf{S}} P$ with a case analysis in the last rule applied. See [19] for details. ◀

▶ **Corollary 74.** $\mathbb{C} = \mu\mathbb{P}$.

Proof. It follows by Theorems 71 and 73. ◀

Main Results. We may now return to considering $\mathbb{H}$. Note that $\mathbb{H} \setminus \mathbb{P} \neq \emptyset$: to see this, consider, e.g., the self-synchronizing process $P_1 \in \mathbb{H}_{\mathsf{F}}$ from Example 18. Using Corollary 74 and disentanglement (Definition 42) we show that given a process $P \in \mathbb{H}_{\mathsf{F}}$, we can find a deadlock-free process $P' \in \mathbb{P}$. Formally, we have:

▶ **Theorem 75.** *Given $P \in \mathbb{H}_{\mathsf{F}}$, $\exists P' \in \mathbb{H}_{\mathsf{R}}$ s.t. $P' \in \mathbb{P}$, $[\![P]\!] \approx_{\mathtt{H}} [\![P']\!]$, and $\mathcal{D}_{\mathbf{P}}(\{\!|P'|\!\}^f)$.*

Proof. By Corollary 53, $[\![P]\!] \rightsquigarrow^* \simeq^*_{cc} [\![P']\!] = P_1 \mid \cdots \mid P_n$, and by Lemma 48 $\forall i \in [1, n]$, there exists $P'_i \in \mathbb{C}$ s.t. $[\![P'_i]\!] = P_i$. From Corollary 74 and Lemma 63 we infer $\{\!|P'_i|\!\}^f \in \mathbb{P}$, $\forall i \in [1, n]$. Moreover, by repeatedly applying Rule T-Par we have that $P' \in \mathbb{P}$. Finally, $\mathcal{D}_{\mathbf{P}}(\{\!|P'|\!\}^f)$ follows from Theorem 39 and Corollary 69. ◀

The following result relates $\mathbb{H}$, its subclasses $\mathbb{H}_{\mathsf{F}}$ and $\mathbb{H}_{\mathsf{R}}$, and $\mathbb{P}$.

▶ **Theorem 76.** *The following hold:* (1) $\mathbb{P} \setminus \mathbb{H} \neq \emptyset$. (2) $\mathbb{H}_{\mathsf{F}} \setminus \mathbb{P} \neq \emptyset$. (3) $\mathbb{P} \cap \mathbb{H}_{\mathsf{R}} \neq \emptyset$.

Proof sketch. We give witnesses for each statement; see [19] for details. ◀

Finally, from Theorem 75 we have the following result, which complements Corollary 53: given $P \in \mathbb{H}_{\mathsf{F}}$, there exists a bisimilar process $P' \in \mathbb{H}_{\mathsf{R}} \cap \mathbb{P}$ via disentanglement. Note that this result applies in particular to processes in $\mathbb{H}_{\mathsf{F}} \setminus \mathbb{P}$.

▶ **Corollary 77.** *Given $P \in \mathbb{H}_{\mathsf{F}}$, there exists $P' \in \mathbb{H}_{\mathsf{R}}$ s.t.* (1) $P \lesssim P'$, (2) $[\![P]\!] \approx_{\mathtt{H}} [\![P']\!]$, *and* (2) $\mathcal{D}_{\mathbf{P}}(\{\!|P'|\!\}^f)$.

$$\text{A-Out}\ \frac{}{v : T, c : S, x : !T.S \vdash^{a}_{S} \overline{x}\langle v, c\rangle} \qquad \text{A-In}\ \frac{\Gamma, y : T, w : S \vdash^{a}_{S} P}{\Gamma, x : ?T.S \vdash^{a}_{S} x(y, w).P} \qquad \text{A-Inact}\ \frac{}{\cdot \vdash^{a}_{S} \mathbf{0}}$$

$$\text{A-EInput}\ \frac{\Gamma \vdash^{a}_{S} P}{\Gamma, x : \mathsf{end}_? \vdash^{a}_{S} x().P} \qquad \text{A-EOutput}\ \frac{}{x : \mathsf{end}_! \vdash^{a}_{S} \overline{x}\langle\rangle} \qquad \text{A-Res}\ \frac{\Gamma, x : T, y : \overline{T} \vdash^{a}_{S} P}{\Gamma \vdash^{a}_{S} (\nu\, xy)P} \qquad \text{A-Par}\ \frac{\Gamma_1 \vdash^{a}_{S} P \qquad \Gamma_2 \vdash^{a}_{S} Q}{\Gamma_1, \Gamma_2 \vdash^{a}_{S} P \mid Q}$$

Figure 11 SP^{a}: Typing Rules for Processes.

4.4 Asynchronous Session Processes

Up to here, we have considered asynchronous communication somewhat indirectly: we have examined the type system for asynchronous processes P, defined its associated classes $\mathbb{P}$ and $\mu\mathbb{P}$, and related them to $\mathbb{C}$ and $\mathbb{H}_{\mathsf{F}}$. To treat asynchronous communication directly, here we consider SP^{a}, an asynchronous variant of SP, and discuss how to transfer our results from §§ 3 and 4.3 from SP to SP^{a} by leveraging an encoding $\langle\!\langle \cdot \rangle\!\rangle : \mathsf{SP}^{a} \to \mathsf{SP}$.

SP^{a}: Asynchronous SP. We define SP^{a} as a variant of SP with asynchronous communication. Considerations about variables, endpoints, and values are as for SP (cf. § 2.1). The syntax of processes is largely as before; only constructs for output and input are modified:

$$P \quad ::= \quad \mathbf{0} \mid x().P \mid \overline{x}\langle\rangle \mid \overline{x}\langle v, c\rangle \mid x(y, w).P \mid P_1 \mid P_2 \mid (\nu\, xy)P$$

Process $\overline{x}\langle v, c\rangle$ sends names v and c along x: the former is a message and the latter is a continuation. Because it has no continuation, this process may be seen as an "output particle" that carries a message. Accordingly, process $x(y, w).P$ receives two values v and c along x and continues as $P\{v/y, c/w\}$, i.e., the process resulting from the capture-avoiding substitution of y by v, and w by c in P. The construct for empty output is also modified. In both $z(x, y).P$ and $(\nu\, xy)P$ x, y are bound in P.

▶ **Definition 78** (SP^{a}). *The typing rules for processes are presented in Figure 11. A judgement $\Gamma \vdash^{a}_{S} P$ denotes that P is well typed in Γ. We shall write $P \in \mathsf{SP}^{a}$ if $\Gamma \vdash^{a}_{S} P$, for some Γ.*

Most typing rules are self-explanatory. Rule A-EOutput types an empty output without continuation. Rule A-Out types the output particle $\overline{x}\langle v, c\rangle$ where $x : !T.S$. Since the output has no continuation, the session name c typed as S serves a a continuation name for x. Dually, Rule A-In types x as $?T.S$ provided that y and w are typed as T and S in P, respectively, thus, w serves as a continuation for x in P.

The reduction semantics for SP^{a} is defined as follows:

$$(\nu\, xy)(\overline{x}\langle v, c\rangle \mid y(z, w).Q \mid S) \longrightarrow (Q\{v/z, c/w\} \mid S) \qquad \text{A-Com}$$

$$(\nu\, xy)(\overline{x}\langle\rangle \mid y().Q \mid S) \longrightarrow Q \mid S \qquad \text{A-EmptyCom}$$

Structural rules and the congruence $\equiv$ are defined as in Figure 2. A labeled reduction semantics $P \xrightarrow{\tau : xy} Q$ for SP^{a} follows easily from Definition 50.

▶ **Example 79.** Consider the process P_{10} from Example 10, which is typable and deadlocked in SP. The following process can be seen as the analogue of P_{10} but in the asynchronous setting of SP^{a}. We define P_{79} as $(\nu\, x_1y_1)(\nu\, x_2y_2)(\nu\, v_1k_1)(\nu\, v_2k_2)P'$, where

$$\begin{aligned} P' \triangleq\ & x_1(z_1,w_1).x_2(z_2,w_2).w_1().w_2().z_1().z_2().\mathbf{0} \\ & \mid (\nu\, c_2c_2')(\overline{y_2}\langle v_2,c_2\rangle \mid \overline{c_2'}\langle\rangle) \mid (\nu\, c_1c_1')(\overline{y_1}\langle v_1,c_1\rangle \mid \overline{c_1'}\langle\rangle) \mid \overline{k_1}\langle\rangle \mid \overline{k_2}\langle\rangle \end{aligned}$$

Note how c_1 and c_2 serve as a continuation for y_1 and y_2, respectively. There exist processes Q_1, Q_2 such that $P_{79} \xrightarrow{\tau:x_1y_1} Q_1 \xrightarrow{\tau:x_2y_2} Q_2 \longrightarrow^* \equiv\ \mathbf{0}$.

We define DF for SP^{a} by adapting Definition 9:

▶ **Definition 80** (SP^{a}: Deadlock Freedom). *A process $P \in \mathsf{SP}^{\mathsf{a}}$ is deadlock free, written $\mathcal{D}_{\mathbf{A}}(P)$, if the following holds: whenever $P \longrightarrow^* P'$ and one of the following holds:* (1) $P' \equiv (\nu\, \tilde{x}\tilde{y})(\overline{x_i}\langle v,c\rangle \mid Q_2)$, (2) $P' \equiv (\nu\, \tilde{x}\tilde{y})(x_i(y).Q_1 \mid Q_2)$, (3) $P' \equiv (\nu\, \tilde{x}\tilde{y})(x_i().Q_1 \mid Q_2)$, *or* (4) $P' \equiv (\nu\, \tilde{x}\tilde{y})(\overline{x_i}\langle\rangle \mid Q_2)$ *(with $x_i \in \tilde{x}$ in all cases) then there exists R such that $P' \longrightarrow R$.*

▶ **Example 81** (Comparing DF in SP and SP^{a}). As just illustrated by means of processes P_{10} (from Example 10) and P_{79}, moving to an asynchronous setting may solve certain deadlocks: we have $\neg\mathcal{D}_{\mathbf{S}}(P_{10})$ and $\mathcal{D}_{\mathbf{A}}(P_{79})$.

On the other hand, moving to an asynchronous setting alone does not resolve the problem. Consider the process $Q_{81} \triangleq (\nu\, x_2y_2)(y_2().\mathbf{0} \mid (\nu\, x_1y_1)(y_1().(\overline{x_2}\langle\rangle \mid \overline{x_1}\langle\rangle)))$. Clearly, Q_{81} is typable in SP^{a}. Also, let Q_{81}^* be the SP variant of Q_{81} with $\mathbf{0}$ as continuation for outputs, i.e., $Q_{81}^* \triangleq (\nu\, x_2y_2)(y_2().\mathbf{0} \mid (\nu\, x_1y_1)(y_1().(\overline{x_2}\langle\rangle.\mathbf{0} \mid \overline{x_1}\langle\rangle.\mathbf{0})))$. We have that Q_{81}^* is typable in SP and that $\neg\mathcal{D}_{\mathbf{S}}(Q_{81}^*)$ and $\neg\mathcal{D}_{\mathbf{A}}(Q_{81})$.

Our goal is to transfer our technical results for SP to the asynchronous setting of SP^{a}. To this end, we define the following encoding:

▶ **Definition 82.** *We define the encoding $\langle\!\langle\, \cdot\, \rangle\!\rangle$ from SP^{a} to SP as follows:*

$$\begin{aligned} \langle\!\langle \overline{x}\langle v,c\rangle \rangle\!\rangle &\triangleq \overline{x}\langle v\rangle.\overline{x}\langle c\rangle.\overline{x}\langle\rangle.\mathbf{0} & \langle\!\langle \mathsf{end}_? \rangle\!\rangle &\triangleq \mathsf{end}_? \\ \langle\!\langle x(y,w).P \rangle\!\rangle &\triangleq x(y).x(w).x().\langle\!\langle P \rangle\!\rangle & \langle\!\langle \mathsf{end}_! \rangle\!\rangle &\triangleq \mathsf{end}_! \\ \langle\!\langle x().P \rangle\!\rangle &\triangleq x().\langle\!\langle P \rangle\!\rangle & \langle\!\langle !T.S \rangle\!\rangle &\triangleq !T.!S.\mathsf{end}_! \\ \langle\!\langle \overline{x}\langle\rangle \rangle\!\rangle &\triangleq \overline{x}\langle\rangle.\mathbf{0} & \langle\!\langle ?T.S \rangle\!\rangle &\triangleq ?T.?S.\mathsf{end}_? \end{aligned}$$

and as an homomorphism for $\mathbf{0}$, $P \mid Q$, and $(\nu\, xy)P$.

▶ **Example 83** ($\langle\!\langle\, \cdot\, \rangle\!\rangle$, At Work). Consider again the process P_{79}. We have seen that $\mathcal{D}_{\mathbf{A}}(P_{79})$. We now check that $\langle\!\langle\, \cdot\, \rangle\!\rangle$ does not add deadlocks by verifying that $\mathcal{D}_{\mathbf{S}}(\langle\!\langle P_{79} \rangle\!\rangle)$:

$$\begin{aligned} \langle\!\langle P_{79} \rangle\!\rangle \triangleq\ & (\nu\, x_1y_1)(\nu\, x_2y_2)(\nu\, v_1k_1)(\nu\, v_2k_2)\langle\!\langle P' \rangle\!\rangle \\ \langle\!\langle P' \rangle\!\rangle \triangleq\ & x_1(z_1).x_1(w_1).x_1().x_2(z_2).x_2(w_2).x_2().w_1().w_2().z_1().z_2().\mathbf{0} \\ & \mid (\nu\, c_2c_2')(\overline{y_2}\langle v_2\rangle.\overline{y_2}\langle c_2\rangle.\overline{y_2}\langle\rangle.\mathbf{0} \mid \overline{c_2'}\langle\rangle.\mathbf{0}) \\ & \mid (\nu\, c_1c_1')(\overline{y_1}\langle v_1\rangle.\overline{y_1}\langle c_1\rangle.\overline{y_1}\langle\rangle.\mathbf{0} \mid \overline{c_1'}\langle\rangle.\mathbf{0}) \mid \overline{k_1}\langle\rangle.\mathbf{0} \mid \overline{k_2}\langle\rangle.\mathbf{0} \end{aligned}$$

One can show that there exist Q_1, Q_2 such that $\langle\!\langle P_{79} \rangle\!\rangle \xrightarrow{\tau:x_1y_1}{}^{3} \equiv Q_1 \xrightarrow{\tau:x_2y_2}{}^{3} \equiv Q_2 \longrightarrow^* \equiv \mathbf{0}$.

By a similar analysis to the one for Theorem 68, we have the following result:

▶ **Theorem 84** ($\langle\!\langle \cdot \rangle\!\rangle$: Operational Correspondence). *Let $P \in \mathsf{SP}^{\mathfrak{a}}$. The following holds:*

1. *if $P \equiv (\nu\, xy)(x(w,z).P \mid \overline{y}\langle v_1, v_2\rangle \mid R)$, then $\exists Q \in \mathsf{SP}$ s.t. $\langle\!\langle P \rangle\!\rangle \longrightarrow^3 Q$ and $Q = \langle\!\langle P\{v_1/w, v_2/z\} \mid R \rangle\!\rangle$.*
2. *If $P \equiv (\nu\, xy)(x().P \mid \overline{y}\langle\rangle.\mathbf{0} \mid R)$, then $\exists Q \in \mathsf{SP}$ s.t. $\langle\!\langle P \rangle\!\rangle \longrightarrow Q$ and $Q = \langle\!\langle P \mid R \rangle\!\rangle$.*
3. *If $\langle\!\langle P \rangle\!\rangle \longrightarrow Q'$, then either: (i) there exists $Q \in \mathsf{SP}$ s.t. $Q' \longrightarrow^2 Q$, $P \longrightarrow P'$ and $\langle\!\langle P' \rangle\!\rangle = Q$, or (ii) $P \longrightarrow P'$ and $\langle\!\langle P' \rangle\!\rangle = Q'$.*

Using Theorem 84 and Definitions 9 and 80 we formalize our claim from Example 83: $\langle\!\langle \cdot \rangle\!\rangle$ does not add deadlocks.

▶ **Theorem 85** (Correspondence DF). *For all $P \in \mathsf{SP}^{\mathfrak{a}}$, $\mathcal{D}_{\mathbf{A}}(P)$ iff $\mathcal{D}_{\mathbf{S}}(\langle\!\langle P \rangle\!\rangle)$.*

Proof sketch. For each direction we prove its contrapositive and proceed by contradiction. See [19] for details. ◀

Using the encoding $\langle\!\langle \cdot \rangle\!\rangle$, and inspired by Theorems 84 and 85, we can define the following classes of asynchronous processes by relying on their synchronous counterparts:

▶ **Definition 86.** *The classes of asynchronous processes are defined as follows:*

$$\mathbb{C}^{\mathfrak{a}} \triangleq \{P \in \mathsf{SP}^{\mathfrak{a}} \mid \langle\!\langle P \rangle\!\rangle \in \mathbb{C}\} \quad \mathbb{P}^{\mathfrak{a}} \triangleq \{P \in \mathsf{SP}^{\mathfrak{a}} \mid \langle\!\langle P \rangle\!\rangle \in \mathbb{P}\} \quad \mu\mathbb{P}^{\mathfrak{a}} \triangleq \{P \in \mathsf{SP}^{\mathfrak{a}} \mid \langle\!\langle P \rangle\!\rangle \in \mu\mathbb{P}\}$$
$$\mathbb{H}^{\mathfrak{a}} \triangleq \{P \in \mathsf{SP}^{\mathfrak{a}} \mid \langle\!\langle P \rangle\!\rangle \in \mathbb{H}\} \quad \mathbb{H}^{\mathfrak{a}}_{\mathsf{F}} \triangleq \{P \in \mathsf{SP}^{\mathfrak{a}} \mid \langle\!\langle P \rangle\!\rangle \in \mathbb{H}_{\mathsf{F}}\} \quad \mathbb{H}^{\mathfrak{a}}_{\mathsf{R}} \triangleq \{P \in \mathsf{SP}^{\mathfrak{a}} \mid \langle\!\langle P \rangle\!\rangle \in \mathbb{H}_{\mathsf{R}}\}$$

Each asynchronous class is contained (up to $\langle\!\langle \cdot \rangle\!\rangle$) in its corresponding synchronous class:

▶ **Lemma 87.** *Let $P \in \mathsf{SP}^{\mathfrak{a}}$. The following hold:* (1) *If $P \in \mathbb{C}^{\mathfrak{a}}$, then $\langle\!\langle P \rangle\!\rangle \in \mathbb{C}$.* (2) *If $P \in \mathbb{H}^{\mathfrak{a}}$, then $\langle\!\langle P \rangle\!\rangle \in \mathbb{H}$.* (3) *If $P \in \mathbb{P}^{\mathfrak{a}}$, then $\langle\!\langle P \rangle\!\rangle \in \mathbb{P}$.* (4) *If $P \in \mu\mathbb{P}^{\mathfrak{a}}$, then $\langle\!\langle P \rangle\!\rangle \in \mu\mathbb{P}$.* (5) *If $P \in \mathbb{H}^{\mathfrak{a}}_{\mathsf{F}}$, then $\langle\!\langle P \rangle\!\rangle \in \mathbb{H}_{\mathsf{F}}$.* (6) *If $P \in \mathbb{H}^{\mathfrak{a}}_{\mathsf{R}}$, then $\langle\!\langle P \rangle\!\rangle \in \mathbb{H}_{\mathsf{R}}$.*

Proof. It follows straightforwardly from Definition 86. ◀

Thus, we can transfer our main results (Corollaries 53, 74, and 77) to $\mathsf{SP}^{\mathfrak{a}}$ via $\langle\!\langle \cdot \rangle\!\rangle$. For instance, Corollary 53 is transferred to $\mathsf{SP}^{\mathfrak{a}}$ as follows, using Lemma 87, and Theorem 84:

▶ **Corollary 88.** *Given $P \in \mathbb{H}^{\mathfrak{a}}_{\mathsf{F}}$, there exist $Q \in \mathbb{H}$ and $P' \in \mathbb{H}^{\mathfrak{a}}_{\mathsf{R}}$ s.t.:* (1) $[\![Q]\!] \in \textsc{Dist}([\![\langle\!\langle P \rangle\!\rangle]\!])$, (2) $[\![Q]\!] \simeq^*_{cc} [\![\langle\!\langle P' \rangle\!\rangle]\!]$, (3) $\langle\!\langle P \rangle\!\rangle \lesssim \langle\!\langle P' \rangle\!\rangle$, (4) $[\![\langle\!\langle P \rangle\!\rangle]\!] \approx_{\mathtt{H}} [\![\langle\!\langle P' \rangle\!\rangle]\!]$.

5 Closing Remarks

Enforcing DF is a relevant issue for message-passing programs, and type systems for concurrent processes offer a convenient setting to develop analysis techniques that detect insidious circular dependencies in programs. Because there exist different type systems that enforce DF, it is natural to wonder how they compare.

We have studied this question in the general and expressive setting of the π-calculus. Our approach aims at establishing the key strengths and limitations of three representative type systems: SP, the reference language for session-typed processes; HCP, the session-typed language based on "propositions-as-sessions" and hypersequents; and P, the priority-based type system for asynchronous processes. Figure 12 gives an overview of our main technical results, refining Figure 1. We briefly discuss their broader significance. Our work can be seen as covering three dimensions of comparison: type systems (SP, HCP, P), process semantics (reduction semantics and LTS), and several definitions of DF (cf. Definitions 9, 26, 36, 60, and 80). Clearly, each type system connects these dimensions in its own way via meta-theoretical results (e.g., type preservation). From this perspective, considering a type

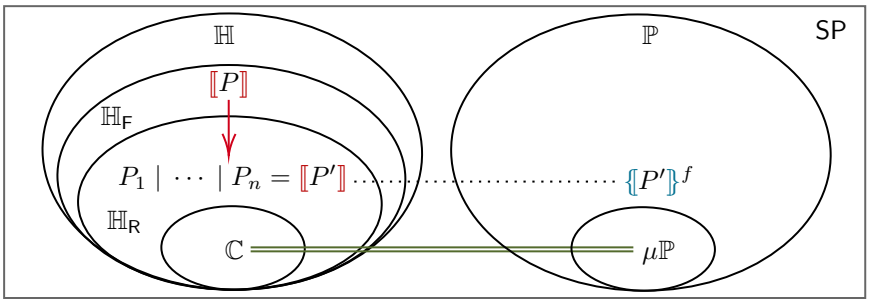


Figure 12 Overview of main results (detailed version). The inclusions between the classes of processes follow from Theorem 76. The red arrow stands for the relation $\rightsquigarrow^* \simeq^*_{cc}$ (disentanglement and commuting conversions) from Corollary 53, where $P_1, \ldots, P_n \in \mathbb{C}$. This arrow, together with $\{\!|P'|\!\}^f \in \mathbb{P}$, represent Corollary 77. The green lines represent Corollary 74 ($\mathbb{C} = \mu\mathbb{P}$).

system such as HCP, which comes with an LTS for typed processes with very distinctive features (delayed actions and self-synchronizations), brings significant value to our results. In contrast, the work of Dardha and Pérez [9] (the most closely related work) considers different type systems for synchronous processes but only under a reduction semantics, which influences the definition(s) of DF to be considered.

Our key discovery is identifying the role that commuting conversions and disentanglement have in the class of deadlock-free processes induced by HCP (Corollary 53). Although studying the practical ramifications of our results goes beyond the scope of this paper, two observations are relevant. First, our results pave the way for developing extensions of the static analyzer TyPiCal [20] that support the mechanized analysis of (session) processes with asynchronous communication. Second, commuting conversions are already important in practical developments, such as execution strategies in *abstract machines* for session-typed programs [4] and definitions of *session subtyping* [6, 15] that enable flexible programming interfaces. An in-depth exploration of these latent connections between our results and tools for the analysis of message-passing programs is left for future work.

It is worth stressing that HCP's LTS enjoys strong logical and denotational justifications; by identifying disentanglement and commuting conversions as key notions for DF, we provide new insights into HCP and its LTS. This way, our work sheds new light into the foundations of HCP, its positioning within the line of work on "propositions-as-sessions", and its expressive power. For instance, it is surprising that processes that share more than one session are typable in HCP, in the sense of Example 14. While Kokke *et al.* [23, 22] have studied disentanglement as a tool to relate HCP and CP, here we uncover its connection with DF. Also, their work on HCP$^-$ (HCP with a reduction semantics but without delayed actions and self-synchronizations) is complementary to our findings. In this sense, our work also offers good motivations for the use of LTSs with regular transitions as the semantics for logic-based session processes (which usually rely on reduction semantics).

It is remarkable that HCP and P are not just two type systems relevant for a formal comparison, but that they are actually connected thanks to asynchrony (an important aspect not covered in Dardha and Pérez's work); the connection is made precise by Theorem 76 and Corollary 77. Here again delayed actions in HCP result to be insightful for an enhanced understanding of logic-based session types. In this line, we observe that De Young *et al.* [10] study asynchrony under "propositions-as-sessions", and in particular explain the role of commuting conversions, which in an asynchronous setting act as as structural congruences (cf. Definition 40). In the synchronous case, they correspond to behavioral equivalences [30].

An interesting point concerns the exact relation between our class $\mathbb{C}$ (Definition 34) and the class $\mathbb{L}$ in Dardha and Pérez's work. There is a key difference between the two: while they consider CP with a typing rule for independent parallel composition (cf. Rule H-MIX$_2$ in Figure 5), we only have Rule C-MIX$_0$ in Figure 4. The reason is technical: it is known that having both MIX$_0$ and MIX$_2$ induces the conflation of the units $\mathbf{1}$ and $\bot$, which conflicts with the distinction made by HCP's denotational semantics (see [19] for details). (In [9], the conflated type is denoted "$\bullet$", and there are no process constructs for session closing.) For the sake of consistency with the denotational semantics, we opted to not include this rule. Therefore, there are processes that are not in our class $\mathbb{C}$ but are in Dardha and Pérez's $\mathbb{L}$.

Having discovered the relation between DF and commuting conversions and disentanglement, an interesting item for future work is revisiting the process transformations proposed by Dardha and Pérez [9] using disentanglement. To focus on the fundamental issues of DF enforcement, we have not covered processes with recursion/sharing; we also plan to consider typed languages with constructs for shared servers and clients.

References

1 Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. Manifest deadlock-freedom for shared session types. In Luís Caires, editor, *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 611–639. Springer, 2019. doi:10.1007/978-3-030-17184-1_22.

2 Gérard Boudol. Asynchrony and the π-calculus (note). Technical report, Rapport de Recherche 1702, INRIA, Sophia-Antipolis, 1992.

3 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2010. doi:10.1007/978-3-642-15375-4_16.

4 Luís Caires and Bernardo Toninho. The session abstract machine. In Stephanie Weirich, editor, *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*, volume 14576 of *Lecture Notes in Computer Science*, pages 206–235. Springer, 2024. doi:10.1007/978-3-031-57262-3_9.

5 Marco Carbone and Søren Debois. A graphical approach to progress for structured communication in web services. In Simon Bliudze, Roberto Bruni, Davide Grohmann, and Alexandra Silva, editors, *Proceedings Third Interaction and Concurrency Experience: Guaranteed Interaction, ICE 2010, Amsterdam, The Netherlands, 10th of June 2010*, volume 38 of *EPTCS*, pages 13–27, 2010. doi:10.4204/EPTCS.38.4.

6 Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. On the preciseness of subtyping in session types. In Olaf Chitil, Andy King, and Olivier Danvy, editors, *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*, pages 135–146. ACM, 2014. doi:10.1145/2643135.2643138.

7 Ornela Dardha and Simon J. Gay. A new linear logic for deadlock-free session-typed processes. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10803 of *Lecture Notes in Computer Science*, pages 91–109. Springer, 2018. doi:10.1007/978-3-319-89366-2_5.

8 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *Proc. of PPDP'12*, pages 139–150. ACM, 2012. doi:10.1145/2370776.2370794.

9 Ornela Dardha and Jorge A. Pérez. Comparing type systems for deadlock freedom. *Journal of Logical and Algebraic Methods in Programming*, 124:100717, 2022. (Preliminary version in the Proceedings of EXPRESS/SOS 2015). doi:10.1016/j.jlamp.2021.100717.

10 Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication. In Patrick Cégielski and Arnaud Durand, editors, *CSL*, volume 16 of *LIPIcs*, pages 228–242. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPICS.CSL.2012.228.

11 Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. On progress for structured communications. In Gilles Barthe and Cédric Fournet, editors, *Trustworthy Global Computing, Third Symposium, TGC 2007, Sophia-Antipolis, France, November 5-6, 2007, Revised Selected Papers*, volume 4912 of *Lecture Notes in Computer Science*, pages 257–275. Springer, 2007. doi:10.1007/978-3-540-78663-4_18.

12 Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. Separating sessions smoothly. In Serge Haddad and Daniele Varacca, editors, *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference*, volume 203 of *LIPIcs*, pages 36:1–36:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.CONCUR.2021.36.

13 Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. Separating sessions smoothly. *Logical Methods in Computer Science*, Volume 19, Issue 3, July 2023. doi:10.46298/lmcs-19(3:3)2023.

14 Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010. doi:10.1017/S0956796809990268.

15 Silvia Ghilezan, Jovanka Pantović, Ivan Prokić, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for asynchronous multiparty sessions. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi:10.1145/3434297.

16 Kohei Honda and Mario Tokoro. On asynchronous communication semantics. In *Object-Based Concurrent Computing*, number 612, in LNCS, 1992.

17 Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. Multiparty Session Programming With Global Protocol Combinators. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:30, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2020.9.

18 Jules Jacobs and Stephanie Balzer. Higher-order leak and deadlock free locks. *Proc. ACM Program. Lang.*, 7(POPL):1027–1057, 2023. doi:10.1145/3571229.

19 Juan C. Jaramillo and Jorge A. Pérez. Contrasting deadlock-free session processes (extended version), 2025. arXiv:2504.15845.

20 Naoki Kobayashi. TyPiCal: Type-based Static Analyzer for the Pi-Calculus. https://www-kb.is.s.u-tokyo.ac.jp/~koba/typical/. Accessed: 2025-03-13.

21 Naoki Kobayashi. A new type system for deadlock-free processes. In *CONCUR'06*, volume 4137 of *LNCS*, pages 233–247, 2006. doi:10.1007/11817949_16.

22 Wen Kokke, Fabrizio Montesi, and Marco Peressotti. Taking linear logic apart. In Thomas Ehrhard, Maribel Fernández, Valeria de Paiva, and Lorenzo Tortora de Falco, editors, *Proceedings Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity-TLLA@FLoC 2018, Oxford, UK, 7-8 July 2018*, volume 292 of *EPTCS*, pages 90–103, 2018. doi:10.4204/EPTCS.292.5.

23 Wen Kokke, Fabrizio Montesi, and Marco Peressotti. Better late than never: A fully-abstract semantics for classical processes. *Proceedings of the ACM on Programming Languages*, 3(POPL), 2019. doi:10.1145/3290337.

24 Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:29, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2022.4.

25 Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off go: liveness and safety for channel-based programming. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 748–761. ACM, 2017. doi:10.1145/3009837.3009847.

26 Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in go using behavioural types. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1137–1148. ACM, 2018. doi:10.1145/3180155.3180157.

27 Luca Padovani. Deadlock and lock freedom in the linear π-calculus. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2603088.2603116.

28 Luca Padovani. A simple library implementation of binary sessions. *J. Funct. Program.*, 27:e4, 2017. doi:10.1017/S0956796816000289.

29 Luca Padovani and Luca Novara. Types for deadlock-free higher-order programs. In Susanne Graf and Mahesh Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*, volume 9039 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2015. doi:10.1007/978-3-319-19195-9_1.

30 Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations and observational equivalences for session-based concurrency. *Information and Computation*, 239:254–302, 2014. doi:10.1016/j.ic.2014.08.001.

31 Zesen Qian, G. A. Kavvos, and Lars Birkedal. Client-server sessions in linear logic. *Proc. ACM Program. Lang.*, 5(ICFP), August 2021. doi:10.1145/3473567.

32 Davide Sangiorgi and David Walker. *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, USA, 2001.

33 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:31, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2017.24.

34 Bernardo Toninho and Nobuko Yoshida. Interconnectability of session-based logical processes. *ACM Trans. Program. Lang. Syst.*, 40(4):17:1–17:42, 2018. doi:10.1145/3242173.

35 Bas van den Heuvel and Jorge A. Pérez. A gentle overview of asynchronous session-based concurrency: Deadlock freedom by typing. In Clément Aubert, Cinzia Di Giusto, Simon Fowler, and Violet Ka I Pun, editors, *Proceedings 17th Interaction and Concurrency Experience, ICE 2024, Groningen, The Netherlands, 21st June 2024*, volume 414 of *EPTCS*, pages 1–20, 2024. doi:10.4204/EPTCS.414.1.

36 Vasco T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, 2012. doi:10.1016/j.ic.2012.05.002.

37 Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014. doi:10.1017/S095679681400001X.

Practical Type-Based Taint Checking and Inference

Nima Karimipour
University of California, Riverside, CA, USA

Kanak Das
University of California, Riverside, CA, USA

Manu Sridharan
University of California, Riverside, CA, USA

Behnaz Hassanshahi
Oracle Labs, Brisbane, Australia

Abstract

Many important security properties can be formulated in terms of flows of tainted data, and improved taint analysis tools to prevent such flows are of critical need. Most existing taint analyses use whole-program static analysis, leading to scalability challenges. Type-based checking is a promising alternative, as it enables modular and incremental checking for fast performance. However, type-based approaches have not been widely adopted in practice, due to challenges with false positives and annotating existing codebases. In this paper, we present a new approach to type-based checking of taint properties that addresses these challenges, based on two key techniques. First, we present a new type-based tainting checker with significantly reduced false positives, via more practical handling of third-party libraries and other language constructs. Second, we present a novel technique to automatically infer tainting type qualifiers for existing code. Our technique supports inference of generic type argument annotations, crucial for tainting properties. We implemented our techniques in a tool TaintTyper and evaluated it on real-world benchmarks. TaintTyper exceeds the recall of a state-of-the-art whole-program taint analyzer, with comparable precision, and 2.93X–22.9X faster checking time. Further, TaintTyper infers annotations comparable to those written by hand, suitable for insertion into source code. TaintTyper is a promising new approach to efficient and practical taint checking.

2012 ACM Subject Classification Software and its engineering → Software verification and validation; Security and privacy → Software security engineering

Keywords and phrases Static analysis, Taint Analysis, Pluggable type systems, Security, Inference

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.18

Related Version *Extended Version*: `https://arxiv.org/abs/2504.18529` [29]

Supplementary Material *Software*: `https://github.com/ucr-riple/TaintTyper`
archived at `swh:1:dir:18769875fdd197811ba90f1008f98cc3d8c47fe0`
Software (ECOOP 2025 Artifact Evaluation approved artifact):
`https://doi.org/10.4230/DARTS.11.2.7`

Funding This research was supported in part by the National Science Foundation under grants CCF-2007024, CCF-2223826, and CCF-2312263, and a gift from Oracle Labs.

1 Introduction

Software security is of critical importance to society, as security vulnerabilities can have severe financial and safety impacts. Many security vulnerabilities can be described as an undesirable flow of *tainted* data. For example, program inputs possibly controlled by an attacker are tainted, and should not flow to sensitive program operations without proper sanitization. *Static taint analysis* aims to automatically discover these dangerous data flows, and many approaches to static taint analysis have been proposed [13, 44, 11, 27, 24, 4, 21, 45, 34, 46, 26].

© Nima Karimipour, Kanak Das, Manu Sridharan, and Behnaz Hassanshahi;
licensed under Creative Commons License CC-BY 4.0

We desire a static taint analysis with the following properties:

- *high precision*, i.e., few false positive reports;
- *high recall*, i.e., few missed vulnerabilities;
- *fast running times*, enabling checking during continuous integration (CI) or even on local builds; and
- *applicability* to existing code bases.

Previous approaches do not satisfy all of these properties. Most existing tools perform whole-program analysis, using inter-procedural alias and dataflow analysis to discover tainted flows. Such approaches are applicable to extant code and can have high precision and recall, but they are difficult to scale to large programs, which can have millions of lines of code [13, 44]. Such analyses can take hours to complete, making them insuitable for CI deployment or local developer runs. Incremental analysis techniques can speed re-analyzing code after small changes [13, 15, 42], but they may miss issues [41] or still consume excessive resources [42].

An alternative approach is type-based checking using type qualifiers [23], as embodied in pluggable type checkers [36, 19]. A type-based approach performs *modular* checking for tainted flows, combining intra-procedural analysis with annotations at method boundaries to capture inter-procedural flows. The Checker Framework [36, 19] includes a type-based tainting checker [2]. Such approaches can have high recall, and they have fast running times, as the use of annotations *completely obviates* the need for inter-procedural analysis. They are also naturally incremental due to modularity, yielding an automatic and significant speedup (nearly 10X in our initial measurements) for small code changes with modern build systems [6]. However, extant type-based approaches lack both precision and applicability. For precision, the previous checker [2] treats third-party libraries and various language constructs too conservatively, leading to excessive false positives. For applicability, running a type-based approach on extant code requires adding annotations, a significant up-front effort that limits adoption.

In this paper, we present a new approach to type-based checking for taint properties that achieves all of our desired properties: high precision and recall, fast running times, and applicability. In contrast to the extant approach [2], we design our checker to prioritize usability and flexibility over full soundness. We handle un-annotated library methods as *polymorphic* by default [27], i.e., only returning tainted data when it is passed in, dramatically reducing the need to annotate such methods. We also introduce more practical handling of a variety of language constructs to reduce false positives. While these techniques can in theory introduce unsoundness, our extensive evaluation showed that the impact on analysis recall was minimal.

We also present a novel technique to automatically infer tainting type qualifiers, achieving applicability to existing code. Our technique extends a recent search-based approach for inferring nullability annotations [30] with several important new features. We develop a novel algorithm for inferring type annotations on generic type arguments, crucial for tainting; this problem was left open in recent work [30, 31]. Our algorithm can also infer polymorphic method annotations when generic types are not present. We introduce an optimized handling of local variable annotations to further improve inference performance.

We implemented our technique in a tool TaintTyper and evaluated it on a range of benchmarks. TaintTyper was able to infer annotations comparable to human-written annotations and suitable for insertion into source code. Further, once inference was completed, TaintTyper had higher recall than a state-of-the-art taint analyzer on real-world benchmarks, with comparable precision, and 2.93X–22.9X faster checking time. Hence, the

evaluation showed that TaintTyper achieved the precision, recall, speed, and applicability goals outlined above. An ablation experiment showed that our new checker and inference features were crucial for TaintTyper's effectiveness.

This paper makes the following key contributions:

- We present a new, more practical type-based tainting checker, with improved handling of third-party libraries and other language constructs.
- We describe a novel inference technique for tainting types. Crucially, the technique handles generic type arguments, and includes an important new optimization to reduce inference time.
- We present a detailed experimental evaluation showing our approach makes type-based taint checking much more practical for real-world projects, significantly improving over the state-of-the-art.

The implementation of TaintTyper is available at `https://github.com/ucr-riple/TaintTyper`.

2 Background

In this section, we provide background on typical approaches to taint analysis (Sec. 2.1) and the type-based approach (Sec. 2.2), comparing them and motivating our techniques.

2.1 Taint Analysis

Taint analysis aims to discover *information flow vulnerabilities* in code [18], an undesirable flow of information from some designated *source* of values to a designated *sink* operation. Our attacker model assumes that the attacker controls inputs originating from untrusted sources and seeks to reach sensitive sinks. In this context, sources include typical mechanisms that receive data from external origins, such as reading from the network, accessing files, or handling user input. Conversely, sinks are operations that have security-critical effects, such as writing to a file, sending data over the network, or executing system or database commands. E.g., for an SQL injection attack, the source is an input from a potential attacker and the sink is a database query, while for a cross-site scripting (XSS) attack, the sink renders output to a web page. Information flow may be *direct*, solely involving data dependencies, or *indirect*, potentially involving control dependence. As with most practical static taint analyses, we consider only direct information flow in this work.

The most common approach to static taint analysis is to perform an inter-procedural (whole-program) dataflow analysis to discover source-to-sink data flows [13, 44, 11, 27, 24, 4, 21, 45, 34, 46, 26]. For languages like Java, inter-procedural analysis requires computing a call graph, typically using whole-program pointer analysis [40]. Tracking of tainted data flows also requires handling of pointer aliasing, to account for flows through object fields and data structures like lists. Precise reasoning about call graphs and pointer aliasing is the key scalability bottleneck for whole-program taint analysis. For example, a recent approach [25] required over 3 hours and 153GB of RAM to precisely analyze the full Java standard library, and real-world programs may grow much larger. Approaches to make such analyses incremental have been proposed [13, 15, 42], which could speed up re-analysis of a code base after a small change. But, such approaches can miss issues [41] or still require significant time or memory [42].

Tainting rules capturing which source-sink flows may be vulnerable are typically provided as input to a taint analysis tool. Such rules may also capture information about *validator* and *sanitizer* operations, whose use may make a source-sink flow safe. Discovering tainting

rules is itself non-trivial and has been a subject of much research [38, 17, 20, 33]. In this work, we focus on the core analysis problems and assume tainting rules have been provided; any improvements in taint rule discovery could be easily combined with our approach.

2.2 Type-Based Taint Analysis

2.2.1 Basics

Type-based taint analysis is built on the ideas of pluggable type systems [36, 19]. A pluggable type system defines a set of *type qualifiers* that refine the built-in types of a language, further restricting the kind of value an expression may evaluate to. We write type qualifiers by prefixing with an `@` character, using Java annotation syntax. For tainting, the main qualifiers are `@Tainted`, for expressions that *may* be influenced by (data-dependent on) a source, and `@Untainted`, for expressions that *must not* be influenced by a source. Tainting rules are provided in the form of `@Tainted` qualifiers on source operations and `@Untainted` qualifiers on sink operations.

A pluggable type system must also define a subtyping relationship $\leq$ between qualifiers. For tainting, we have `@Untainted` $T \leq$ `@Tainted` T for any base type T; `@Untainted` values may safely flow to (possibly) `@Tainted` locations, but not vice-versa. Enforcing this property requires checking for subtype compatibility at all pseudo-assignments in the program, including assignments to variables / fields, parameter passing, and returns. To reduce the number of explicit annotations required, pluggable type systems interpret unqualified types as having a default qualifier. For tainting, the default qualifier is `@Tainted`. The following example illustrates this checking:

```
1 class Ex {
2   @Tainted String f1; @Untainted String f2;
3   void m1(@Untainted String s) {
4     f1 = s; // no error: @Untainted assigned to @Tainted
5   }
6   void m2(@Tainted String t) {
7     m1(t); // error: @Tainted passed to @Untainted
8     f2 = t; // error: @Tainted assigned to @Untainted
9   }
10 }
```

In the code, `@Tainted String f1` could have been written as just `String f1` due to defaulting. Local variables are not subject to defaulting: their types are inferred using flow-sensitive dataflow analysis [36, 19], which also enables handling of taint validators. By default, the Checker Framework's Tainting Checker [2] treats all code, including unannotated third-party libraries, with the same defaulting rules. So, since `@Tainted` is the default qualifier, any value returned by a third-party method is assumed to be tainted. This approach leads to too many false positives in practice; Sec. 4.1 describes our more practical handling of such code.

2.2.2 Method calls

Pluggable type systems must also check that method overriding respects the subtyping rules for the type qualifiers. Return types must be covariant in overriding methods, and parameter types must be contravariant. Consider the following (erroneous) example:

```
1 class Super {
2   @Untainted String foo() { return "hi"; }
3 }
4 class Sub extends Super {
```

```
5   // invalid override!
6   @Tainted String foo() { return source(); }
7 }
```

`Sub.foo()` cannot be allowed to return a `@Tainted String`, since code written to handle `Super` objects may expect `foo()` to return an `@Untainted String`. Without this checking, the following vulnerable code would pass the checker:

```
1 void process(Super s) { sink(s.foo()); }
2 process(new Sub());
```

Given this method override checking, method calls can be handled in the pluggable types approach *without computing a call graph*. At a call c to a method with declared target `P.m`, the checker needs only to check that the parameters passed and return value use at c are consistent with the type of `P.m` itself. If at runtime, an overriding method `Q.m` is invoked at c, override checking ensures that the tainting behavior of `Q.m` is consistent with `P.m`, so no vulnerability is possible. This ability to check calls without a call graph yields a huge scalability benefit for the pluggable types approach.

2.2.3 Data structures / polymorphism

Pluggable type systems use parametric polymorphism to support storing differing types of data in different instances of a data structure. Consider the following example:

```
1 List<@Tainted String> taintedStrs = new ArrayList<>();
2 List<@Untainted String> untaintedStrs = new ArrayList<>();
3 taintedStrs.add(source()); untaintedStrs.add("safe");
4 sink(taintedStrs.get(0)); // error reported
5 sink(untaintedStrs.get(0)); // no error reported
```

Here we have two `ArrayList` instances, one holding possibly-tainted strings and the other holding untainted strings. The taintedness of list contents is captured by adding a taint qualifier to the generic type argument, e.g., `List<@Tainted String>`. With these type declarations, the pluggable type checker reports an error at the first `sink` call while also proving that the second `sink` call is safe.

To achieve similar precision, a whole-program static analysis must use *context sensitivity*: each call to the relevant `ArrayList` methods must be analyzed separately, and `ArrayList`'s internal state must be represented with a context-sensitive heap abstraction [40]. Context sensitivity significantly increases the running time of such analyses. The pluggable types approach uses qualified type arguments to achieve similar precision with much less cost.

`@PolyTaint` annotations can be used for polymorphic methods that do not already use type variables. Consider, e.g., the `parentPath` function in Fig. 1. Here, the return value of a call to `parentPath` should only be considered `@Tainted` if the parameter for that call is `@Tainted`; the `@PolyTaint` annotations capture this property. Again, a standard whole-program analysis can achieve this precision with context sensitivity, but at greater cost to scalability.

It is possible that different objects of a non-generic class vary in terms of whether tainted data is stored in their fields. Such cases can be handled in whole-program analysis using field sensitivity, which models each field of each (abstract) object separately. Typically, no such modeling is used in the pluggable types approach, outside of generic types. We have found that this rarely leads to false positives; qualified generic type arguments and `@PolyTaint` nearly always suffice for good precision in practice.

```
1 Map<String, +@Untainted String> paths = ...;
2 +@PolyTaint String parentPath(+@PolyTaint String path) {
3   return path.substring(0, path.lastIndexOf("/"));
4 }
5 void exec(String name) {
6   String path = paths.get(name);
7   String parent = parentPath(path);
8   sink(Paths.get(parent).toAbsolutePath().toFile());
9 }
10 void sink(@Untainted String t) { ... }
```

Figure 1 Motivating example for inference. Green text indicates where annotations are inserted by TaintTyper.

2.2.4 Benefits

A typechecking approach to tainting scales well since the checking is fully modular: each method can be checked in isolation given only the type signatures of fields it accesses and methods it invokes. Checking is also incremental: after a change, only the code that needs to be re-compiled needs to be re-checked. Beyond improved scalability, a type-based approach to taint checking has various other advantages [36, 19]. Qualifiers serve as machine-checked documentation of tainting properties and invariants, and can also aid in safely performing refactorings. And, errors from the type-based approach can be more understandable, since they only require reasoning about local code and the types of related functions, not an inter-procedural trace. Because our analysis is type-based, it operates over type annotations rather than runtime behavior, meaning the attacker's knowledge of the program's internals does not fundamentally affect the analysis. We assume the attacker cannot modify the program's source code.

Currently, adopting type-based taint checking imposes a significant burden on programmers, as they must manually annotate their own code and any relevant third-party code. Our checking and inference techniques significantly reduce this burden, making the type-based approach more practical.

3 Motivating Example and Approach

Example. Fig. 1 gives a motivating example for our approach. (Disregard the green inferred annotations for the moment.) The `sink` method (line 10) requires an `@Untainted` argument. Assuming the values in the `paths` map cannot be tainted, the `sink` call at line 8 is safe. We shall show how the enhanced checker and inference of TaintTyper can automatically annotate and validate this example.

The first challenge is the use of library methods like `Paths.get`, `toAbsolutePath`, and `toFile` on line 8. Without manual annotation, the previous type-based approach [2] treats these calls as returning a `@Tainted` value by default. Our approach treats most unannotated methods as polymorphic [27]: it assumes they only return `@Tainted` data when a parameter is `@Tainted`. Therefore, by default, TaintTyper determines that the expression passed to `sink` can only be `@Tainted` if the `parent` variable is `@Tainted`, a correct handling for this example.

The `parent` variable is assigned the result of calling `parentPath(path)` (line 7), where `path` is a value in the `paths` map (line 6). As noted above, the values in the `paths` map are not tainted, which is captured by giving `paths` the type `Map<String,@Untainted String>` (line 1); this type makes the `path` variable `@Untainted` at line 6. The `parentPath` function

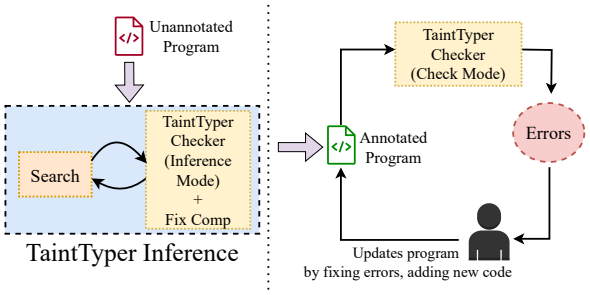


Figure 2 High-level architecture of TaintTyper.

can only return `@Tainted` data if its argument is `@Tainted`, captured with `@PolyTaint` annotations (line 2). With these annotations, the TaintTyper checker reports no warning for this code, as desired.

The example shows that to be effective, TaintTyper must support inferring annotations on generic type arguments and `@PolyTaint` annotations. The generic type argument support is required to eliminate the error, and `@PolyTaint` properly captures the behavior of the `parentPath` method.[1] The inference technique of Sec. 5 can successfully infer the necessary green annotations in Fig. 1.

Our approach. Fig. 2 shows the high-level architecture of TaintTyper. Given an unannotated program, TaintTyper first performs inference to create an annotated version of the program. The inference improves on a recent search-based technique [30] that repeatedly runs a checker to determine the best set of annotations to insert. Here, the checker is our new type-based taint checker, combined with a new fix computation algorithm to enable inference of type argument annotations and `@PolyTaint`. The inference step only needs to run once. Afterward, developers need only run the TaintTyper checker. They can fix the errors initially reported by the checker and also write new code (with annotations) that will be verified by the checker. Since the checker is type based, it runs much faster than a whole-program static analysis, enabling quick turnaround times, less CI resource usage, and even checking on local builds.

4 Practical Type-Based Checking

This section details the new features in TaintTyper's type checker that reduce false positives in real-world code, making the checker practical. We discuss handling of unannotated code, and then new handling of other language constructs.

4.1 Unannotated Code

TaintTyper specially handles interactions with unannotated, unchecked code, typically written by a third-party. Given the prevalence of third-party libraries lacking taint annotations in modern Java programs, such interactions occur commonly. By default, the Checker Framework's Tainting Checker [2] uses `@Tainted` annotation for all code, whether from source or from libraries. So, all unannotated library methods are assumed to have a `@Tainted` return type, yielding too many false positives to be usable.

[1] For this code excerpt, the parameter and return of `parentPath` could be marked `@Untainted`, but this would unnecessarily force all other callers of `parentPath` to pass in `@Untainted` data.

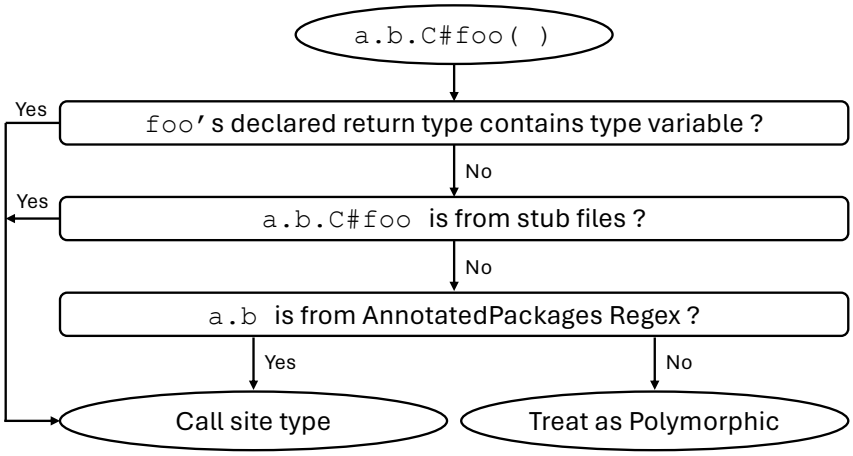


Figure 3 Logic for determining the return type qualifier for a method call, accounting for unannotated code.

TaintTyper adopts a *polymorphic by default* handling of calls to unannotated methods, extending a technique from previous work [27]. This approach treats the return type and all parameter types (including the receiver type) of an unannotated method as if they were annotated as `@PolyTaint` (see Sec. 2.2.3). For calls to such methods, the return value will be treated as `@Tainted` only if at least one of the actual parameters at the call is `@Tainted`. Consider this expression from line 8 from Fig. 1:

```
Paths.get(parent).toAbsolutePath().toFile();
```

The invoked `toFile` method is unannotated, so with our polymorphic treatment, its return value will only be treated as `@Tainted` if the result of the `toAbsolutePath` call is `@Tainted`. Since `toAbsolutePath` is also unannotated, the process recurses, and the taintedness depends on the result of the `Paths.get(parent)` call. Finally, since `Paths.get` is also unannotated, TaintTyper concludes the taintedness of the whole expression depends on whether *parent* is tainted, as desired for this example.

An overview of the logic for determining the return type qualifier for a method call, and when to apply polymorphic defaulting, is given in Fig. 3 (the logic for parameter types is similar). The flowchart handles a call to some method `a.b.C#foo`, where `a.b` is the package containing class `C`. If `foo`'s declared return type contains a generic type variable, we use the standard type checking rule for the call site, even if `foo` is in unannotated code. This exception is important since the return type is at least partially determined by type arguments from the call site. E.g., line 6 of Fig. 1 invokes `Map.get`, whose return type is the type variable `V` for map values. Though `Map.get` is unannotated, TaintTyper applies standard checking, to preserve information from the type argument `@Untainted String` given for this particular `Map` at line 1.

TaintTyper also always uses the standard call site type if `foo`'s type been specified in a stub file [3], as such files allow for externally providing types for library routines like sources and sinks. Otherwise, to determine whether code should be treated as annotated, TaintTyper takes as input an *AnnotatedPackages* regular expression (borrowing from the NullAway nullness checker [12]) which specifies which Java packages should be treated as annotated code. Note that this setting does not distinguish between first-party and third-party code, providing flexibility. E.g., when adopting TaintTyper, source packages

can be set as annotated gradually, initially leaving other source packages as unannotated and unchecked. Only if the `a.b` package is not part of the annotated packages, the polymorphic handling for unannotated code is applied.

The polymorphic-by-default handling of unannotated code is not sound for all cases. For example, if an unannotated method is itself a tainted source, then its return value should be treated as tainted, independent of the argument taintedness. Similar issues arise for unannotated sink methods. As discussed in Sec. 2.1, we view discovery of source and sink methods as a separate problem from the core checking and inference issues we address here. A method may also return tainted data even with untainted arguments, if a setter method of the receiver is invoked earlier with tainted data. When discovered, such cases can be addressed via stub file models [3]. In practice, our evaluation (Sec. 7) showed that this technique did not lead to false negatives in comparison to two state-of-the-art tools on real-world benchmarks, and that it dramatically increased practicality compared to [2].

4.2 Other Constructs

TaintTyper computes a default `@Untainted` type for a variety of language constructs that are treated as `@Tainted` by the previous checker [2]. Enum constants, class literals, lambda expressions, and fields of annotations are always treated as `@Untainted`. A static final field is treated as `@Untainted` if its initializer expression is `@Untainted`. An array initializer expression (e.g., `new String[] {x,y}`) has `@Untainted` contents by default if all the initial array values are `@Untainted`. Similarly, a cast expression is `@Untainted` if the casted expression is `@Untainted`. In some cases, like static final fields, an explicit `@Untainted` annotation could be written, but our defaulting reduces the annotation burden. Note that other cases like lambda expressions cannot be directly annotated, and can only be handled with a warning suppression without our approach.

`java.util.Collection` data structures are widely used in Java programs, making special-case handling useful. A `Collection` is often constructed directly from another `Collection` or array, e.g., `new ArrayList<>(otherList)`. In such cases, if the other `Collection` holds `@Untainted` elements, TaintTyper always treats the new `Collection` as having `@Untainted` elements. The version of [2] we compared with did not handle these cases correctly due to type inference limitations.[2]

The `Collection.toArray(T[])` method, which converts a `Collection` to a `T[]` array, is also frequently used. With the baseline checker, even for a `Collection` of `@Untainted` values, the array contents type of a `toArray` call is `@Tainted` unless the `toArray` argument is explicitly annotated (e.g., `c.toArray(new @Untainted String[0])`). TaintTyper does not require this annotation and treats the result of `toArray` as having `@Untainted` contents if the `Collection` contents are `@Untainted`.

5 Inference

In this section we detail how TaintTyper performs inference. TaintTyper extends a previous search-based inference technique with support for generic types, `@PolyTaint` annotations, and unannotated code. It also introduces a new optimization that significantly improves inference performance.

[2] The inference limitations have been addressed in more recent versions, but we found that these versions introduced new bugs, so we did not update the version that we compared to.

5.1 Baseline Algorithm

Our inference extends the search-based inference approach of Karimipour et al. [30]. The approach aims to infer annotations that minimize the final number of errors reported by a checker, which in [30] was the nullness checker NullAway [12]. By minimizing the final number of errors, this approach infers annotations that *maximize* the amount of code that fully passes the checker. A recent study showed this search-based approach to work better than alternate approaches in practice [28].

The search performs *black-box inference*, in which the effectiveness of annotations in reducing errors is measured by running the checker itself and observing its output. This technique computes a set of annotations that fix some reported checker errors. Then, the checker is re-applied to see if these annotations cause new errors. Consider this example:

```
1 void m1() { m2(source()); m2(source()); }
2 void m2(String t) { sink(t); }
```

Initially, TaintTyper reports an error at line 2, since `t` is `@Tainted` by default and passed to a sink. A local fix for this error is to annotate `t` as `@Untainted`. But, with this fix, TaintTyper reports two new errors on line 1 (since `source`'s return is `@Tainted`), increasing the total number of errors. So, the fix is rejected by the search. The previous work describes optimizations to speed the search process, by evaluating independent annotations simultaneously [30]; we evaluate the effectiveness of these optimizations for taint inference in Sec. 7.2.3.

The previous work only described how to fix errors reported by NullAway. For tainting, computing annotations for fixes is significantly more complex due to the need to support generic types and `@PolyTaint`. Recent work on pluggable type inference left open inference for generic types due to the challenges involved [30, 31]. In Sec. 5.2, we describe our novel technique for computing annotations to fix tainting errors, supporting generic type arguments and `@PolyTaint`.

5.2 Computing Fixes

For our tainting checker, there are two main causes of reported errors: type incompatibility at a (pseudo-)assignment, and incorrect method overriding (see Sec. 2.2). For both cases, fixing the error requires *adjusting the type* of relevant variables or expressions. E.g., for the example of Sec. 5.1, the initial fix was to change the type of parameter `t` to `@Untainted String`. Alg. 1 gives our new technique to compute the annotations to achieve a desired type adjustment. The main procedure FindAnnots takes as arguments an expression e and a desired type for the fix T_f. It either returns a set of annotations that modify e's type to be T_f, or $\perp$ to indicate it has failed to find such a set. FindAnnots relies on various other procedures, some of which we elide for brevity but describe in text. We also only show handling of key representative language constructs for simplicity. We first explain the basic cases for Alg. 1, and then present our novel handling of generic type arguments and `@PolyTaint` annotations.

5.2.1 Basics

For a variable v (which could be a local, parameter, or field), line 3 uses a routine UpdateType (not shown) to update v's declared type directly. UpdateType may fail and return $\perp$; e.g., TaintTyper does not attempt to convert a raw type like `List` to a generic type like `List<@Untainted String>`, as this change is out of scope for our work (other tools can

Algorithm 1 Pseudocode for finding annotations for a fix.

```
1: procedure FindAnnots(e, T_f)
2:     if e is a variable v then
3:         return UpdateType(v, T_f)
4:     else if e is a binary expression e_1 op e_2 then
5:         return FindAnnots(e_1, T_f) ⊎ FindAnnots(e_2, T_f)
6:     else if e is a call e_1.m(e_2, ..., e_n) then
7:         G ← GenericsAnnots(e, T_f)
8:         if G ≠ ⊥ return G
9:         P ← PolyTaintAnnots(e, T_f)
10:        if P ≠ ⊥ return P
11:        m ← InvokedMethod(e)
12:        if TreatAsPolyTainted(m) then
13:            return ⊎_{i∈PTArgs(m)} MakeUntainted(e_i)
14:        else
15:            return UpdateType(ReturnType(m), T_f)
16:        end if
17:    end if
18: end procedure
19: procedure MakeUntainted(e)
20:    T' ← @Untainted TypeOf(e)
21:    return FindAnnots(e, T')
22: end procedure
23: procedure GenericsAnnots(e, T_f)
24:    m ← InvokedMethod(e)
25:    S ← FindTypeSubst(ReturnType(m), T_f)
26:    if S = ⊥ return ⊥
27:    e_r ← ReceiverArg(e)
28:    T' ← ApplySubst(S, ReceiverType(m), TypeOf(e_r))
29:    return FindAnnots(e_r, T')
30: end procedure
```

be applied for such cases [43]). For binary operators with sub-expressions e_1 and e_2, we recursively compute fixes for e_1 and e_2 and then combine them using a special $\uplus$ operator (line 5). $\uplus$ propagates the failure value $\bot$ – it is defined as follows:

$$A \uplus B = \begin{cases} \bot & A = \bot \vee B = \bot, \\ A \cup B & \text{otherwise.} \end{cases}$$

Method calls, handled at lines 6–16, require the most complex handling. We first attempt to handle the call by inferring annotations on generic type arguments (line 7). If that fails, we attempt inference of `@PolyTaint` annotations (line 9). We will explain the generics and `@PolyTaint` cases further shortly. If both generics and `@PolyTaint` inference fail (by returning $\bot$), we fall back to a more direct handling, depending on the method being invoked.

Line 12 checks if the invoked method m should be treated as having `@PolyTaint` annotations. This check returns true if either m has declared `@PolyTaint` annotations or if m is unannotated and our default polymorphic handling applies (see Sec. 4.1). In such cases, to make the call's type untainted, *all* `@PolyTaint` parameters must be made untainted, reflected in line 13 (PTArgs(m) returns the `@PolyTaint` argument positions for m). MakeUntainted (lines 19–22) updates the type of expression e with a top-level `@Untainted` annotation and then recursively calls FindAnnots. Any failure to make an argument untainted is propagated using $\uplus$. Finally, for calls to annotated methods, our base handling is to update the declared return type of the invoked method (line 15).

5.2.2 Generics

For calls to methods involving generic types, our approach aims to find fixes that annotate type arguments instead of directly updating the invoked method's return type. To see why, consider the call `paths.get(name)` on line 6 in Fig. 1. This call invokes `Map.get`, whose return type is generic type variable `V`. Say that FindAnnots aims to make the result of the call `@Untainted`. The baseline technique for method calls (line 15 in Alg. 1) would change `get`'s return type to `@Untainted V`, a valid fix. But, this change would force *all* calls to `Map.get` to return untainted values, preventing *any* `Map` from holding possibly-tainted values, which is impractical.

Instead, our technique tries to find a fix that leverages generic type arguments, as shown in the GenericsAnnots routine of Alg. 1. First (line 25), we call FindTypeSubst (not shown) to find a *substitution* for the type variables in the return type of m that yields the desired type T_f. For the above example, GenericsAnnots is called with $e =$ `paths.get(name)` and $T_f =$ `@Untainted String`. Since `get`'s return type is type variable `V`, line 25 successfully finds a substitution $S =$ `V` $\mapsto$ `@Untainted String` that yields T_f. FindTypeSubst may fail to find a substitution, in which case it returns $\bot$ (line 26). FindTypeSubst works by recursing through type structures, mapping type variables to the desired type arguments; we elide details as they are straightforward.

When we successfully find a substitution S, we then *apply* S to the declared receiver type of the method, and recursively try to find corresponding annotations for the receiver argument of the call (lines 27–29). If S does not cover all type variables in the receiver type, we reuse the generic type arguments from the receiver argument at the call site. For our `paths.get` example, the declared type of the `Map.get` receiver is `Map<K,V>`, but our substitution `V` $\mapsto$ `@Untainted String` does not cover `K`. So, we reuse the `String` type argument for `K` from line 1 of Fig. 1, yielding a recursive call FindAnnots(`paths`, `Map<String,@Untainted String>`) at line 29.

The recursive nature of FindAnnots successfully handles much more complex uses of generic types, e.g.:

```
1 void foo(Map<Integer, +@Untainted String> t) {
2   sink(t.values().iterator().next());
3 }
```

FindAnnots aims to make the return type for the `next` call `@Untainted String`, but it is not immediately evident which generic type argument must be annotated to achieve this. In our algorithm, the generics logic proceeds by recursively trying to make the `iterator` call return `Iterator<@Untainted String>`. This in turn leads to trying to make the `values` call return `Collection<@Untainted String>`, which finally leads to successfully adjusting the type of `t` to `Map<Integer, @Untainted String>`. TaintTyper can also annotate generic type arguments in `extends` clauses of class declarations, and generic methods (where the type variable is scoped to the method instead of the class) are also handled fully.

5.2.3 `@PolyTaint` inference

As noted in Sec. 2.2, `@PolyTaint` can be useful when a method is generic in its tainting behavior but was not declared using generic type variables. Due to the lack of type variables, inference of `@PolyTaint` annotations must *discover* relevant data flow from parameters to return values, which may occur through callee methods. E.g., for the `parentPath` method in Fig. 1, the taintedness of the `path` argument influences the return taintedness via a call to `path.substring`.

Algorithm 2 Pseudocode for inferring polymorphic annotations.

```
1: procedure PolyTaintAnnots(e, T_f)
2:     m ← InvokedMethod(e)
3:     Result ← ∅
4:     F_return ← FindAnnotForReturnStatements(m, T_f)
5:     Worklist ← F_return
6:     Processed ← ∅
7:     while Worklist ≠ ∅ do
8:         F_element ← Worklist.pop()
9:         if F_element is not on a local variable then
10:            Result ← Result ∪ {F_element}
11:            continue
12:        end if
13:        if F_element ∈ Processed then
14:            continue
15:        end if
16:        Processed ← Processed ∪ {F_element}
17:        F_assign ← FindAnnotForAssignments(m, F_element, T_f)
18:        Worklist ← Worklist ∪ F_assign
19:    end while
20:    F_Parameters ← {F | F ∈ Result ∧ F is on parameter of m}
21:    F_NonParameters ← Result \ F_Parameters
22:    if F_Parameters = ∅ then
23:        return MakeUntainted(m)
24:    else
25:        PolyMethodFix ← MakePolyTainted(m, F_Parameters) ∪ F_NonParameters
26:        F_args ← ∅
27:        for arg ∈ PolyMethodFix.args do
28:            F_args ← F_args ∪ FindAnnots(arg, UpdatedTarget(T_f))
29:        end for
30:    end if
31:    return F_args ⊎ PolyMethodFix
32: end procedure
```

The PolyTaintAnnots procedure for inferring `@PolyTaint` annotations (called at line 9 in Alg. 1) is conceptually simple: it works by inserting `@PolyTaint` annotations, observing where such annotations lead to type checking errors, and then recursively inserting more annotations to fix those errors. However, we found that a straightforward implementation based on this strategy was too inefficient, so we introduced two improvements. First, a naïve approach to discovering new type errors is to re-run the full type checker, but leads to many expensive checker runs; instead, we implemented our own limited analysis of data flows relevant to `@PolyTaint` to discover new errors. Second, we bounded the depth of the search into callee methods, giving up and returning $\perp$ if inference required searching further (we found depth five to work best in our experiments).

Alg. 2 gives a simplified view of the PolyTaintAnnots procedure. The algorithm starts by invoking FindAnnotForReturnStatements (not shown), which scans the method body for returned expressions and uses FindAnnots to discover the annotations needed to align each expression's type with T_f. If any of the computed annotations targets a local variable v, PolyTaintAnnots uses FindAnnotForAssignments (not shown) to scan for expressions assigned to v and compute necessary annotations for those expressions (again via FindAnnots). This process iterates using a worklist until all locals are handled. Upon

completing the iterations, if there exists an annotation on a method parameter, the method is identified as polymorphic, and the necessary annotations are computed. The method indirectly invokes FindAnnots from Alg. 1 through calls to FindAnnotForReturnStatements and FindAnnotForAssignments, which may recursively invoke PolyTaintAnnots. A separate depth bound ensures termination, and the algorithm returns $\perp$ if the bound is reached or if any call to FindAnnots returns $\perp$. When a method is determined to be polymorphic, the algorithm calculates the required annotations for its arguments and combines these with the annotations derived for the return type.

5.2.4 Example

We now discuss the overall process of applying inference to our motivating example in Fig. 1. Since the error in the unannotated code is reported at line 8, initially FindAnnots is invoked to try to make the `toFile()` call passed to `sink` have type `@Untainted String`. As discussed in Sec. 4.1, this expression includes nested calls to unannotated code, handled by line 13 in Alg. 1. Eventually, this leads to annotating `parent` as `@Untainted`. This annotation causes a new checker error at line 7, leading inference to use FindAnnots to make the `parentPath(path)` call `@Untainted`. Here, our `@PolyTaint` inference succeeds for `parentPath`, leading to the annotations on line 2. The search then makes `path` `@Untainted`, causing a type error at line 6. Here, FindAnnots makes the `paths.get` call `@Untainted` by updating the generic type of the `paths` field, discussed in detail in Sec. 5.2.2 above. With this change, no new errors are reported, completing inference.

5.2.5 Local Variable Optimization

Our initial inference implementation took an excessive amount of time, nearly 24 hours for larger benchmarks. Most inference time is spent running the checker to detect the impact of annotations on warnings. For tainting, we found that many such checker runs were for annotations on local variables, e.g., the runs for `parent` and `path` in Fig. 1 (discussed above). As an optimization, we enhanced our fix computation to *internally* determine the impact of local variable annotations rather than using the checker. This reasoning requires finding assignments to the relevant locals, and then recursively invoking FindAnnots to make the type of each assignment's right-hand side match the local's new type, similar to the logic shown in Alg. 2 for inferring `@PolyTaint`. With this optimization, two checker calls (for `parent` and `path`) are eliminated in inference for Fig. 1. In Sec. 7.2 we show this optimization has a very significant impact on inference performance.

6 Implementation

TaintTyper includes both type-based taint checking and inference (see Fig. 2). TaintTyper's checker (see Sec. 4) was built using the Checker Framework [36, 19], version 3.42.0. Use of the Checker Framework equips the TaintTyper checker with robust support for flow-sensitive local type inference, checking of generic types, and qualifier polymorphism. For our prototype, we modeled a number of source and sink methods involved in the most common Java vulnerabilities [7]. Our sinks include common methods that write to a file, send data over the network, or execute sensitive system or database commands. We modeled sources that read from the network, the file system, or user input. We also modeled a few relevant well-known sanitizer methods [8]. As noted in Sec. 2, creating more complete databases of sources, sinks, and sanitizers is still a research problem, and TaintTyper can easily benefit from further advances in that area.

TaintTyper's inference implementation uses a modified version of the NullAwayAnnotator tool [30]. Inside the TaintTyper checker, we implemented Alg. 1 to find annotations that can fix an error, and added support for serializing this information in the checker output. Our inference implementation reads in this serialized output to use during its search. The search is similar to that of [30], enhanced with our local variable optimization (Sec. 5.2.5). The search is depth bounded [30], and we used a bound of 15 in our experiments.

7 Evaluation

Here we present an experimental evaluation showing the high effectiveness of TaintTyper in practice.

7.1 Experimental setup and research questions

From previous work we found three benchmark suites will all tainting violations labeled, serving as a ground truth. They are:

- **Securibench Micro** [32], which provides 122 servlets exhibiting potential information-flow vulnerabilities, with the source code annotated with benign or problematic flows.
- **JInfoFlow** [24], a self-contained benchmark of 12 information-flow violations featuring reflection-intensive, event-driven code without dependencies on external libraries.
- **Injection Experiments** [39], comprising 8 Java programs with information-flow violations reported by their tool. While the original tool is no longer available, the dataset remains accessible. In the metadata, the authors label the reports from their tool as true or false positives.

Although these three benchmarks capture various interesting cases, they consist either of toy examples [32, 24] or projects that have not been maintained for years [39]. Consequently, we use them solely as one validation of the impact of the techniques of Sec. 4 on TaintTyper's recall.

For a more realistic evaluation, we examine a suite of actively-developed open-source Java programs used in recent work [10, 14]. We selected only projects for which there were some reported tainting errors. We also strove to include a variety of project sizes and types, ensuring the benchmark reflects real-world scenarios; our suite includes a web framework, content management system, security framework, forum software, and library management system. Due to the lack of a ground truth on these benchmarks, we made considerable manual efforts to ensure the accuracy of our results. This included carefully comparing results against previous static analyzers and manually annotating the code for comparison (further details below). In the end, we prioritized benchmarks that were the most representative and important, ultimately selecting seven projects for our evaluation, as listed in Table 1. We used all benchmarks from [10] for which there were reported tainting errors except for webgoat and opencms, which were not included due to complex build system configurations that TaintTyper cannot yet handle. One such complexity arises when the code relies on generated code, such as when using Project Lombok [1]. If a generated getter method is inferred to return untainted, it must first be delomboked and explicitly annotated. However, when the code is compiled, the generated code is overwritten, causing the loss of information needed to propagate the untainted annotation from the getter to the corresponding field. This is not a fundamental limitation of our approach but rather an implementation challenge that could be addressed with additional engineering effort.

Table 1 Benchmark sizes and inferred annotation counts.

Project	KLoC	Inferred Annotations	
		Total	per kLoC
esapi-java-legacy	18.3	265	14.4
pybbs	9.7	105	10.8
alfresco-core	13.4	81	6.0
alfresco-remote-api	85.8	213	2.5
cxf	47.5	296	6.2
struts	49.3	313	6.3
commons-configuration	20.2	261	12.9

We used CodeQL [4] v2.15.1 and P/Taint [24, 5] as baseline tools for our evaluation. CodeQL is a production-quality security analysis tool, widely used and freely available. P/Taint uses state-of-the-art whole-program analysis techniques, also employed in recent work [13]. We configured both tools to detect issues involving the sources and sinks modeled for TaintTyper (Sec. 6). We considered a variety of other tools for our experiments but found them unsuitable. FlowDroid [11] is a well-known taint analyzer for Android, but it does not support analysis of Java server programs. And, the taint analyses in SonarQube [9], RAPID [21], and CompTaint [13] are not freely available.

Given these benchmarks and tools, we studied seven research questions:

RQ1 Does TaintTyper find the known errors in existing labeled benchmarks?

RQ2 After inference, how do TaintTyper's reported errors compare to those reported by state-of-the-art tools (in terms of precision and recall)?

RQ3 After inference, how does TaintTyper's checking time compare with state-of-the-art tools on real-world benchmarks?

RQ4 Does TaintTyper's inference run in a reasonable amount of time, and are our optimizations effective?

RQ5 Does TaintTyper require a reasonable number of annotations?

RQ6 How do the annotations inferred by TaintTyper compare to manually-written annotations?

RQ7 How is TaintTyper's effectiveness impacted if we disable checker improvements, generic type argument inference, and `@PolyTaint` inference?

We address RQ1–RQ4 in Sec. 7.2. Then, Sec. 7.3 addresses RQ5 and RQ6, and Sec. 7.4 answers RQ7. All experiments were conducted in a Google Cloud instance with an AMD EPYC Milan 3rd Generation 2.45GHz CPU with 32vcpu (16 cores) and 128GB memory.

7.2 Inference effectiveness

7.2.1 Soundness on labeled benchmarks

To evaluate the soundness of TaintTyper, the three labeled benchmark suites were used. For these experiments, we customized the source and sink specifications used by TaintTyper to match what was expected by each benchmark suite.

We first ran TaintTyper's checker without inference, and confirmed that it did not miss *any* labeled vulnerabilities in the benchmarks. We then applied our inference to annotate the benchmarks automatically and re-check them again, to check how the inferred annotations impacted recall.

Table 2 Without inference, TaintTyper reports all the labeled true-positive issues across these three benchmarks. With inferred annotatations, it misses one issue in Securibench Micro, detects all issues in JInfoFlow, and misses 15 issues in Injection Experiments.

SecuriBench Micro		JInfoFlow-bench		Injection Experiments	
Total	136(137)	Total	12	Total	730(745)
aliasing	12	JInfoFlow/basic	2	Snake&Ladder	40
arrays	9	JInfoFlow/ctx	5	MediaPlayer	0
basic	61	JInfoFlow/event	5	EmergencySNRest	0(3)
collections	14			FarmTycoon	5
datastructures	5			Abagail	0
factories	3			JExcelAPI	3(4)
inter	16			Colossus	682(693)
pred	5				
reflection	4				
sanitizers	3(4)				
session	3				
strong updates	1				

Table 3 Precision and recall results across our benchmark suite.

Project	TP	CodeQL		TaintTyper	
		Precision	Recall	Precision	Recall
esapi-java-legacy	18	0.90	0.50	0.95	1.00
pybbs	9	1.00	0.89	1.00	1.00
alfresco-core	2	1.00	0.50	1.00	1.00
alfresco-remote-api	21	0.82	0.43	0.70	1.00
cxf	9	1.00	0.11	0.75	1.00
struts	23	0.39	0.39	0.50	1.00
commons-configuration	11	1.00	0.73	0.69	1.00

For Securibench Micro, the metadata indicated 136 labeled vulnerabilities. However, manual inspection revealed 137 in total, with one unlabeled issue in *Basic26.java* and one missing label in *Basic31.java*. Securibench Micro also makes extensive use of raw types, which appear rarely in modern Java code and are not yet fully handled by TaintTyper; we inserted (unannotated) generic type arguments for these cases. With these fixes, after inference, TaintTyper only missed one true issue out of 137; this was due to an interaction between our polymorphic-by-default library handling and side effects, as discussed in Sec. 4.1.

In inference-annotated JInfoFlow-bench, TaintTyper successfully identifies all 12 labeled vulnerabilities. The Injection Experiments consist of eight Java programs, one of which could not be compiled. We annotated the remaining seven programs (collectively comprising 754 labeled issues) using TaintTyper inference. Eight of the labeled issues occur in files not part of the available benchmark codebase, and one occurs in test code. Upon manually inspecting the inferred annotations and the reported errors from TaintTyper, we found that it discovered 730 of the remaining 745 issues, with the missed issues again due to side-effecting third-party library calls. The number of missed issues across these benchmarks is small, and as shall be shown in Sec. 7.2.2, we saw *no* missed issues for TaintTyper when compared to a production-quality tool on real-world benchmarks. Table 2 summarizes our findings across these benchmarks.

RQ1: TaintTyper's checker identifies all known issues in the labeled benchmarks, and inference reduces recall only slightly.

Table 4 Analysis runtime and inference runtime results across our benchmark suite.

Project	Analysis Runtime			Inference Runtime			
	TaintTyper	CodeQL	P/Taint	ALL	LOC	CORE	NONE
esapi-java-legacy	19s	94s	53m	23m	65m	47m	105m
pybbs	12s	83s	35m	8m	15m	11m	22m
alfresco-core	15s	75s	21m	9m	14m	20m	31m
alfresco-remote-api	69s	202s	167m	134m	311m	391m	711m
cxf	47s	1076s	151m	148m	294m	498m	817m
struts	39s	425s	>48h	343m	629m	1354m	>48h
commons-configuration	25s	114s	209m	105m	168m	299m	432m

7.2.2 Reported errors

Table 3 compares the precision and recall of TaintTyper and CodeQL for our benchmarks, addressing RQ2. For TaintTyper, the errors are computed after inference has run, so the code includes inferred annotations. Computing recall requires knowing the ground truth of all real issues in these benchmarks, which is infeasible to collect. To estimate recall, we use the union of all true positive issues reported by TaintTyper, CodeQL, and P/Taint as our ground truth. (We treat a report as a true positive if the corresponding dataflow is deemed feasible by manual inspection.) This may over-estimate the true recall of all tools, but it provides a good basis for comparing the tools. The "TP" column gives the number of true positive issues for each benchmark.

Comparing error reports between the tools was non-trivial due to their different reporting approaches. For TaintTyper, an error is typically reported as a code location where a `@Tainted` value is being written into an `@Untainted` location. CodeQL reports an error as a data flow trace from a source to a sink, and P/Taint reports an error as a source/sink pair without a trace. Comparing these errors required manually matching each true-positive TaintTyper error to a step in some CodeQL trace, or to some data flow for a P/Taint source/sink pair, and vice versa. CodeQL sometimes treats a formal parameter as a tainted source, without any explicit call passing in tainted data. In these cases, TaintTyper annotates the parameter as `@Untainted`, capturing the fact that tainted data should not be passed in, but does not report an error. We count such cases as equivalent to reporting the error; TaintTyper could easily report errors for such cases if desired.

We exclude the precision and recall of P/Taint from Table 3 as both were very low on our benchmarks. P/Taint found a total of six true positive issues across the benchmarks (all of which were also found by some other tool), and it reported 50 false positives. We carefully checked our P/Taint configuration and confirmed it found expected issues in smaller benchmarks like Securibench Micro [32]. We also consulted with the tool authors, who acknowledged that P/Taint may not handle these benchmarks well (e.g., due to missing framework support). Still, triaging P/Taint results was very useful, to gain further confidence in our ground truth.

TaintTyper finds all true positive issues discovered by CodeQL and P/Taint, leading to a recall of 1 on all benchmarks in Table 3. TaintTyper also finds additional true positives missed by CodeQL, reflected in CodeQL's lower recall numbers. TaintTyper has lower precision than CodeQL on three benchmarks; we suspect this is due to heuristics in CodeQL that are missing in TaintTyper. Still, TaintTyper matches or exceeds CodeQL's precision on the other four benchmarks.

Table 5 Number of annotations inserted manually and by TaintTyper inference, and final error counts with various features disabled; C for checker improvements, G for generics inference, and P for `@PolyTaint` inference.

Project	Manual Annotation Count			Inferred Annotation Count			Error Count			
	Total	TypeArg	PolyTaint	Total	TypeArg	PolyTaint	C off	G off	P off	All Active
esapi-java-legacy	278	20	34	265	13	34	49	16	13	13
pybbs	95	27	4	105	46	4	21	16	2	2
alfresco-core	81	38	0	81	36	0	12	5	2	2
alfresco-remote-api	110	12	10	213	22	7	66	42	27	24
cxf	380	62	37	296	51	42	73	37	20	11
struts	347	50	5	313	47	7	116	54	44	37
commons-configuration	239	10	25	261	12	23	33	17	11	11

RQ2: On our benchmarks, TaintTyper has outstanding recall, with comparable precision to CodeQL.

7.2.3 Performance

Table 4 gives analysis runtimes for TaintTyper, CodeQL, and P/Taint checking. The speedups of TaintTyper's checker over CodeQL are quite significant, ranging from 2.93X–22.9X. And, we see orders-of-magnitude speedups compared to P/Taint, which could not analyze the struts benchmark within a 48-hour time limit. Given that TaintTyper's checking is modular and incremental, we expect even larger speedups over the whole-program analysis approach for larger benchmarks. As a sanity check of the benefits of incremental checking, we manually re-ran TaintTyper checking for five randomly-chosen source files for alfresco-remote-api and struts (the two largest benchmarks), and observed further speedups of 8.7X and 10.9X respectively.

RQ3: TaintTyper's checker runs much faster than the baseline tools, with further incremental speedups.

Table 4 shows inference performance with all optimizations enabled in the ALL column. Fully-optimized inference always ran in less than 6 hours, suitable for an overnight run and acceptable since it only needs to run once (see Fig. 2). The LOC, CORE, and NONE columns respectively show running times with our new local variable optimization only (Sec. 5.2.5), the core optimizations of [30] only (see Sec. 5.1), and no optimizations. The core optimizations of [30] seem to have a lesser impact for tainting inference than they did for nullability; excluding struts (which did not terminate in 48 hours with optimizations disabled), we see a maximum speedup of 2.23X, compared to 17.8X reported in [30]. Our local variables optimization on its own yields speedups of 1.47X–2.78X, sometimes larger than the core optimizations. Fortunately, the techniques are complementary, together yielding the best speedups of 2.75X–5.52X.

RQ4: TaintTyper has acceptable inference performance, aided significantly by our new optimization.

7.3 Assessing annotations

Table 1 shows the number of annotations inferred by TaintTyper for our benchmarks and the corresponding annotation density, addressing RQ4. The number of inferred annotations per KLoC is relatively low, ranging from 2.5-14.4. In comparison, Banerjee et al. [12] reported an average of 13 annotations per KLoC for NullAway, ranging up to 46 annotations, and that checker has been widely adopted.

RQ5: The annotation requirements of TaintTyper are low enough to enable adoption, given the importance of preventing tainting vulnerabilities.

Table 5 compares the number of annotations inferred by TaintTyper to the number of annotations inserted in a separate manual process. Two co-authors added manual annotations, each checking the other's work and coming to consensus for any disagreement. We completed this process, adding 1,530 manual annotations. We limited the manual changes to inserting annotations, prohibiting code changes. This limitation is contrived: a developer would most likely fix bugs and refactor code alongside adding annotations. We chose this methodology to fairly compare with TaintTyper inference, which only inserts annotations.

Table 5 shows that the number of inserted annotations does tend to differ between the two approaches. We inspected the discrepancies in detail, and found that in all cases, TaintTyper's inserted annotations were reasonable; which annotations were "better" was a subjective question. In cases where TaintTyper inserted fewer annotations, one pattern was where a manually-written annotation captured some desired invariant, but TaintTyper eschewed the annotation since it increased the final error count. A second pattern were cases where during manual annotation, an opportunity to use `@PolyTaint` was missed, but TaintTyper made use of `@PolyTaint` to avoid several other `@Untainted` annotations.

For cases where TaintTyper inserted more annotations, a common explanation was again reducing error count; TaintTyper would insert many extra annotations to reduce the final error count by one, but this was not deemed to be worthwhile during manual annotation. In such cases, arguably a better fix would be to restructure the code so fewer annotations would be needed. We could easily add a setting to TaintTyper to limit the number of annotations it inserts to fix a single warning to avoid such cases. In an extended version of the paper [29], we give detailed examples illustrating the above scenarios.

RQ6: TaintTyper's inserted annotations were always acceptable for insertion into source code, and sometimes improved on our manual annotations.

7.4 Ablation

Finally, to answer RQ6, we performed an ablation study to see how many errors are reported by TaintTyper when disabling the checker features of Sec. 4, generics inference (Sec. 5.2.2), and `@PolyTaint` inference (Sec. 5.2.3) individually. The results are shown in Table 5. We see large increases in reported errors with our new checker features disabled (3X-10.5X more errors), and similar impacts with generics support disabled (1.2X-8X more errors), showing the criticality of these features for precision. Further, without the new checker features, 85 of the additional errors were false positives that could not be removed with annotations (see Sec. 4.2); such cases require a warning suppression, causing developer frustration.

Inference of `@PolyTaint` has a less significant impact on final error counts. However, `@PolyTaint` inference is still critical for generating annotations close to what would be manually written, as shown in Table 5. Manual annotations included a total of 115 `@PolyTaint` annotations across our benchmarks, and with inference of such annotations disabled, none of these would be found by TaintTyper. Similarly, Table 5 shows that generic type annotations are used commonly, and hence TaintTyper's support for inferring these annotations is important.

RQ7: Our new checker features, generic type support, and `@PolyTaint` support were all critical to TaintTyper's effectiveness.

7.5 Threats to Validity

Our benchmark choices are a threat to external validity. We described our methodology for choosing benchmarks in Sec. 7.1. While we strove to choose diverse benchmarks, it is possible that TaintTyper will be less effective on a different set of programs. Our choice of tools for comparison (CodeQL and P/Taint) is another threat to external validity; other whole-program static analyzers may perform differently. See Sec. 7.1 for our process in choosing these tools. Paper co-authors performing the manual annotation of benchmarks is a threat to internal validity. We strove to add these annotations disregarding the workings of TaintTyper's inference. Other developers may manually annotate code in different ways, but a user study evaluating the degree of such differences is out of scope for this work. Implementation bugs in TaintTyper may also impact internal validity; but, we have done significant checking of correctness using manual inspection and a suite of regression tests.

8 Related Work

There is a broad literature on taint analysis; here we discuss the most closely related work. In Sec. 2 we contrasted type-based techniques with whole-program approaches [13, 44, 11, 27, 24, 4, 21, 45, 34, 46, 26], and we compared with CodeQL [4] and P/Taint [24] in our evaluation. Recent work by Banerjee et al. [13] describes an approach to incremental taint analysis but does not evaluate its performance. Szabo [42] presents an initial exploration of incrementalizing CodeQL analyses. While their incremental running times were promising, the additional memory usage of their technique was prohibitive. Type-based taint analysis is naturally incremental and does not suffer from these engineering challenges.

The most closely related whole-program approach is that of Huang et al. [27], which partially inspired our work. Their work is also formulated in terms of type-based taint analysis and type inference. Their type system does not support generic types: instead, they apply polymorphic types to fields to achieve a form of field sensitivity. It is unclear how to expose such field polymorphism in terms of standard pluggable types. Their work does not present a technique to persist types into source code to enable checking without inference and is specific to Android applications so we did not include this tool in our evaluations. TaintTyper performs inference for standard Java pluggable types, including generic types, enabling a straightforward integration with standard development workflows. The inference of [27] runs faster than ours since it operates over a single constraint encoding. However, our inference works with an existing checker without requiring a constraint encoding, which simplifies making improvements to the checker like those of Sec. 4.

There are many other approaches to enforcing information flow properties with types. The well-known Jif system [35] and many subsequent works support sophisticated properties like multiple principals. Such features are not necessary to capture the vulnerabilities targeted

by typical taint analyses and this work. Ernst et al. [22] target verification of Android apps, also using pluggable types. Their system aims for soundness for a security-critical military context, leading to an annotation burden of 60 annotations per KLoC, much higher than ours. TaintTyper eschews strict soundness to reduce the annotation burden for usability. We believe TaintTyper's inference technique could be adapted to systems like [22] in the future.

Regarding alternate inference approaches, standard type inference [37, Chapter 22] tries to discover a complete typing for a program in a given type system. Such techniques do not directly apply to our scenario, as a typical program will not be verifiable in our taint type system solely through adding annotations, due to true or false positives; we aim to find a good set of annotations for such untypable programs. Kellogg et al. [31] present a general inference technique for any pluggable type system built on the Checker Framework. However, their technique does not infer polymorphic or generic type annotations, and it may generate many more annotations than what would be written by developer.

Checker Framework Inference [16] uses constraint-based approach to infer types, with applications to domains like a type system for measurement units [47]. We chose a "black box" inference approach [30] for this work since it enabled re-using a checker implementation without reimplementing its logic in a constraint language. The approach of [47] does not support inference of polymorphic method annotations like `@PolyTaint`, and the paper does not discuss in detail its level of support for inferring annotations on generic type arguments.

9 Conclusions

We have presented TaintTyper, a novel approach to type-based taint checking and inference for Java. TaintTyper includes a novel checker that makes the core tainting type system more practical to use, and a novel inference algorithm capable of handling generic types and polymorphic annotations. Our evaluation showed that TaintTyper provided significant scalability advantages over a standard approach, with improved recall and comparable precision, and inferred annotations suitable for direct inclusion in source code. Hence, TaintTyper is a significant step toward practical and widescale type-based taint checking for Java.

References

1 Project Lombok. `https://projectlombok.org`, 2019. Accessed: 2022-04-03.
2 Checker Framework Manual: Tainting Checker. `https://checkerframework.org/manual/#tainting-checker`, 2024. Accessed: 2024-03-13.
3 Checker Framework Manual: Using stub classes. `https://checkerframework.org/manual/#stub`, 2024. Accessed: 2024-03-13.
4 CodeQL. `https://codeql.github.com`, 2024. Accessed: 2024-02-07.
5 Doop – Framework for Java Pointer and Taint Analysis (using P/Taint). `https://github.com/plast-lab/doop`, 2024. Accessed: 2024-07-29.
6 Gradle User Guide – Incremental Java compilation. `https://docs.gradle.org/current/userguide/java_plugin.html#sec:incremental_compile`, 2024. Accessed: 2024-07-29.
7 OWASP Top Ten. `https://owasp.org/www-project-top-ten/`, 2024. Accessed: 2024-04-11.
8 Sanitizer Methods in OWASP Find Security Bugs. `https://github.com/find-sec-bugs/find-sec-bugs/tree/master/findsecbugs-plugin/src/main/resources/safe-encoders/`, 2024. Accessed: 2024-04-11.
9 SonarQube. `https://www.sonarsource.com/products/sonarqube/`, 2024. Accessed: 2024-07-26.

10 Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. Static analysis of java enterprise applications: frameworks and caches, the elephants in the room. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 794–807. ACM, 2020. doi:10.1145/3385412.3386026.

11 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269. ACM, 2014. doi:10.1145/2594291.2594299.

12 Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. NullAway: Practical type-based null safety for Java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE, 2019. doi:10.1145/3338906.3338919.

13 Subarno Banerjee, Siwei Cui, Michael Emmi, Antonio Filieri, Liana Hadarean, Peixuan Li, Linghui Luo, Goran Piskachev, Nicolás Rosner, Aritra Sengupta, Omer Tripp, and Jingbo Wang. Compositional taint analysis for enforcing security policies at scale. In *ESEC/FSE 2023*, 2023. doi:10.1145/3611643.3613889.

14 Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. Vul4j: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*, pages 464–468. ACM, 2022. doi:10.1145/3524842.3528482.

15 Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, pages 3–11, 2015. doi:10.1007/978-3-319-17524-9_1.

16 Checker Framework Inference. https://github.com/opprop/checker-framework-inference, 2022. Accessed: 2022-04-02.

17 Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin Vechev. Scalable taint specification inference with big code. In *PLDI*, pages 760–774, June 2019. doi:10.1145/3314221.3314648.

18 Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, July 1977. doi:10.1145/359636.359712.

19 Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muşlu, and Todd Schiller. Building and using pluggable type-checkers. In *ICSE 2011, Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, Waikiki, Hawaii, USA, May 2011. doi:10.1145/1985793.1985889.

20 Saikat Dutta, Diego Garbervetsky, Shuvendu K. Lahiri, and Max Schäfer. InspectJS: leveraging code similarity and user-feedback for effective taint specification inference for javascript. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '22, pages 165–174, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3510457.3513048.

21 Michael Emmi, Liana Hadarean, Ranjit Jhala, Lee Pike, Nicolás Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. RAPID: checking API usage for the cloud in the cloud. In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 1416–1426. ACM, 2021. doi:10.1145/3468264.3473934.

22 Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward XueJun Wu. Collaborative verification of information flow for a high-assurance app store. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1092–1104. ACM, 2014. doi:10.1145/2660267.2660343.

23 Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*, pages 192–203, 1999. doi:10.1145/301618.301665.

24 Neville Grech and Yannis Smaragdakis. P/Taint: unified points-to and taint analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA):102:1–102:28, 2017. doi:10.1145/3133926.

25 Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. An efficient tunable selective points-to analysis for large codebases. In Karim Ali and Cristina Cifuentes, editors, *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2017, Barcelona, Spain, June 18, 2017*, pages 13–18. ACM, 2017. doi:10.1145/3088515.3088519.

26 Wei Huang, Yao Dong, and Ana L. Milanova. Type-based taint analysis for java web applications. In Stefania Gnesi and Arend Rensink, editors, *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8411 of *Lecture Notes in Computer Science*, pages 140–154. Springer, 2014. doi:10.1007/978-3-642-54804-8_10.

27 Wei Huang, Yao Dong, Ana L. Milanova, and Julian Dolby. Scalable and precise taint analysis for android. In Michal Young and Tao Xie, editors, *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 106–117. ACM, 2015. doi:10.1145/2771783.2771803.

28 Nima Karimipour, Erfan Arvan, Martin Kellogg, and Manu Sridharan. A new approach to evaluating nullability inference tools. *Proc. ACM Softw. Eng*, 2(FSE), July 2025. doi:10.1145/3715732.

29 Nima Karimipour, Kanak Das, Manu Sridharan, and Behnaz Hassanshahi. Practical Type-Based Taint Checking and Inference (Extended Version). *arXiv*, 2025. doi:10.48550/arXiv.2504.18529.

30 Nima Karimipour, Justin Pham, Lazaro Clapp, and Manu Sridharan. Practical inference of nullability types. In Satish Chandra, Kelly Blincoe, and Paolo Tonella, editors, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1395–1406. ACM, 2023. doi:10.1145/3611643.3616326.

31 Martin Kellogg, Daniel Daskiewicz, Loi Ngo Duc Nguyen, Muyeed Ahmed, and Michael D. Ernst. Pluggable type inference for free. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 1542–1554. IEEE, 2023. doi:10.1109/ASE56229.2023.00186.

32 Benjamin Livshits. *Improving software security with precise static and runtime analysis*. PhD thesis, Stanford University, Stanford, CA, USA, 2006. AAI3242585.

33 Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: specification inference for explicit information flow problems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 75–86, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1542476.1542485.

34 V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in Java applications with static analysis. In Patrick D. McDaniel, editor, *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*. USENIX Association, 2005. URL: https://www.usenix.org/conference/14th-usenix-security-symposium/finding-security-vulnerabilities-java-applications-static.

35 Andrew C. Myers. Jflow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. Association for Computing Machinery. doi:10.1145/292540.292561.

36 Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 2008. doi:10.1145/1390630.1390656.

37 Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

38 Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS Symposium 2014*, January 2014. doi:10.14722/ndss.2014.23039.

39 Fausto Spoto, Elisa Burato, Michael D. Ernst, Pietro Ferrara, Alberto Lovato, Damiano Macedonio, and Ciprian Spiridon. Static identification of injection attacks in java. *ACM Trans. Program. Lang. Syst.*, 41(3):18:1–18:58, 2019. doi:10.1145/3332371.

40 Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. Alias analysis for object-oriented programs. In David Clarke, Tobias Wrigstad, and James Noble, editors, *Aliasing in Object-Oriented Programming*. Springer, 2013. doi:10.1007/978-3-642-36946-9_8.

41 Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. Interactive Abstract Interpretation with Demanded Summarization. *TOPLAS*, 46(1), 2024. doi:10.1145/3648441.

42 Tamás Szabó. Incrementalizing production CodeQL analyses. In Satish Chandra, Kelly Blincoe, and Paolo Tonella, editors, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1716–1726. ACM, 2023. doi:10.1145/3611643.3613860.

43 Frank Tip, Robert M. Fuhrer, Adam Kiezun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring using type constraints. *ACM Trans. Program. Lang. Syst.*, 33(3):9:1–9:47, 2011. doi:10.1145/1961204.1961205.

44 Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 87–97. ACM, 2009. doi:10.1145/1542476.1542486.

45 Jie Wang, Yunguang Wu, Gang Zhou, Yiming Yu, Zhenyu Guo, and Yingfei Xiong. Scaling static taint analysis to industrial SOA applications: a case study at Alibaba. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1477–1486. ACM, 2020. doi:10.1145/3368089.3417059.

46 Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 171–180. ACM, 2008. doi:10.1145/1368088.1368112.

47 Tongtong Xiang, Jeff Y. Luo, and Werner Dietl. Precise inference of expressive units of measurement types. *Proc. ACM Program. Lang.*, 4(OOPSLA):142:1–142:28, 2020. doi:10.1145/3428210.

Profile-Guided Field Externalization in an Ahead-Of-Time Compiler

Sebastian Kloibhofer[1]
Institute for System Software, Johannes Kepler University, Linz, Austria

Lukas Makor[1]
Institute for System Software, Johannes Kepler University, Linz, Austria

Peter Hofer
Oracle Labs, Linz, Austria

David Leopoldseder
Oracle Labs, Vienna, Austria

Hanspeter Mössenböck
Institute for System Software, Johannes Kepler University, Linz, Austria

Abstract

Field externalization is a technique to reduce the footprint of objects by removing fields that most frequently contain zero or null. While researchers have developed ways to bring this optimization into the Java world, these have been limited to research compilers or virtual machines for embedded systems. In this work, we present a novel field externalization technique that uses information from static analysis and profiling to determine externalizable fields. During compilation, we remove those fields and define *companion classes*. These are used in case of non-default-value writes to the externalized fields. Our approach also correctly handles synchronization to prevent issues in multithreaded environments. We integrated our approach into the modern Java ahead-of-time compiler GraalVM Native Image. We conducted an evaluation on a diverse set of benchmarks that includes standard and microservice-based benchmarks. For standard benchmarks, our approach reduces the total allocated bytes by 2.76% and the maximum resident set size (max-RSS) by 2.55%. For microservice benchmarks, we achieved a reduction of 6.88% for normalized allocated bytes and 2.45% for max-RSS. We computed these improvements via the geometric mean. The median reductions are 1.46% (alloc. bytes) and 0.22% (max-RSS) in standard benchmarks, as well as 3.63% (alloc. bytes) and 0.20% (max-RSS) in microservice benchmarks.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Object oriented languages; Software and its engineering → Classes and objects

Keywords and phrases compilation, instrumentation, profiling, fields, externalization, memory footprint reduction, memory footprint optimization

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.19

Funding This research project was partially funded by Oracle Labs.

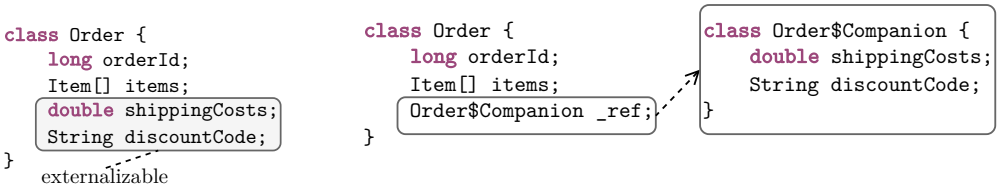
Acknowledgements We thank all members of the Virtual Machine Research Group at Oracle Labs. We also thank all researchers at the Johannes Kepler University Linz's Institute for System Software for their support of and valuable feedback on our work.

1 Introduction

Optimizing a program for speed, efficiency, and safety is a crucial goal of most compilers. In modern languages, such optimizations frequently concern objects and similar structured types. Java is a language that enables object-oriented programming at a high abstraction

[1] Both authors contributed equally to this research.

© Sebastian Kloibhofer, Lukas Makor, Peter Hofer, David Leopoldseder, and Hanspeter Mössenböck; licensed under Creative Commons License CC-BY 4.0



(a) `Order` class with an ID, an array of (unspecified) items, shipping costs, and a discount code. The example assumes that `shippingCosts` and `discountCode` are rarely needed and thus externalizable.

(b) Externalized version of `Order` including its generated companion class that holds the externalized fields. The original class is augmented with a new `_ref` field that references the companion object if allocated. In later parts of the paper we detail improvements that allow us to eliminate this field.

Figure 1 Original and externalized representation of an `Order` class that is used as a continuous example to explain various aspects of our field externalization implementation.

level without sacrificing performance. As manifold as the usage scenarios of objects are in Java, as varied are the optimizations that compilers can apply. Optimizations on objects tend to fall into two categories: Reducing the memory footprint of objects and improving the efficiency of accesses to the objects' properties and methods. Object allocations are made more efficient by directly allocating into thread-local allocation buffers [32] or are eliminated altogether via escape analysis [14, 67]. Escape analysis typically also removes field accesses via scalar replacement, thus improving performance. Dynamic dispatch on methods is also a frequent target for optimizations. Compilers try to devirtualize calls [31], inline them [1, 36], or use inline caches [13, 15, 27, 28, 36, 82] to optimize for the most common cases. Other optimizations aim to shrink the object header [33, 74], a prefix in every object that references, among other things, type information and the virtual method table. While compilers also try to minimize the memory consumption of objects, the range of techniques [12, 72] applicable in this area has been more limited so far.

When analyzing heap dumps of Java applications, we noticed that certain fields of objects often just hold their default values. Other researchers have pointed out similar findings [4, 11, 23, 62]. This shows potential for optimization, as such objects occupy more memory than necessary. Especially in cloud and serverless computing, reducing the memory footprint of applications is important. Thus, we propose a novel form of *field externalization* – a technique for removing such fields, thereby reducing the memory footprint of objects and thus the overall memory consumption of an application.

The `Order` class shown in Figure 1a is used as a running example throughout this paper. In this example, an order consists of an ID, an array of order items (the implementation of which is irrelevant for this example), the shipping costs, and a discount code. We assume that the first two fields are non-zero and non-null in most objects. However, both `shippingCosts` and `discountCode` are not; they are mostly zero or null (the default values for these fields) and only hold relevant values in a small number of objects.

Via field externalization, we can optimize the layout of the `Order` class by removing fields that hold default values in most cases. This results in the layout presented on the left-hand side of Figure 1b. However, as optimizations have to ensure that also corner cases are handled correctly, we must be able to handle `shippingCosts` and `discountCode` if they are ever set to a non-default value. Therefore, we create a new class, the so-called *companion type* `Order$Companion`, that stores the *externalized fields*, i.e., the fields that were removed from the original `Order` class. Additionally, we introduce a pointer in the original class – we call this the *companion reference field*. Initially, this pointer is null. If a non-default value is assigned to an externalized field, an instance of the companion type – the *companion object* –

is allocated to store the field value. This companion object is assigned to the companion reference field. Hence, writing a non-default value to an externalized field introduces some overhead. However, as allocating the companion object should only be necessary in a few cases, these costs are outweighed by the reduced overall footprint of the other objects.

Various researchers have developed approaches to integrate field externalization into compilers [4, 11, 23, 62]. Notably, most are based on just-in-time (JIT) compilers [4, 11, 23], compilers for embedded systems [11, 23, 62], or research compilers [4, 62]. These approaches have shown promising results in combating memory consumption in Java programs. However, we could not find any contemporary approach that enables field externalization in a production-grade compiler without compromising language or run-time features such as multithreading.

Therefore, we propose a novel approach for field externalization in the state-of-the-art ahead-of-time (AOT) compiler framework GraalVM Native Image [78, 79]. It works by first gathering profiling information for a target program in an offline profiling run. Then, we combine the profiling information with information from Native Image's points-to analysis [7, 24, 25, 26, 39, 40, 60, 61, 66, 68, 71, 80] to identify and subsequently externalize fields that most often hold their default value. Furthermore, we developed a synchronization mechanism for accesses to externalized fields that adheres to the Java Memory Model [43].

Our approach targets web and microservice applications hosted in the cloud. Providers of Function-as-a-Service infrastructures (AWS Lambda [3], Google Cloud Functions [21], etc.) typically not only base their pricing on CPU usage but also on the amount of memory that is consumed by the application [84, 19, 18]. Hence, reducing memory usage can save costs.

With this work, we make the following contributions:

1. A profiling approach that utilizes both static analysis as well as field-level profiling to identify fields for optimization.
2. A novel AOT profile-guided field externalization approach that removes fields from objects ahead of time. Our approach is integrated into a state-of-the-art AOT compiler and supports multithreaded access to externalized fields.
3. Novel techniques to reduce the overhead and increase the applicability of field externalization by reusing existing fields as *masked companion references*, supporting *class hierarchies*, and generating *companion factory methods*.
4. An evaluation of our approach on standard and microservice benchmarks measuring memory footprint, run-time performance, and image size. The evaluation demonstrates that our approach can reduce the maximum RSS and the allocated bytes in individual benchmarks, showcasing its viability to reduce the footprint of cloud applications.

The paper is structured as follows: Section 2 describes GraalVM and GraalVM Native Image. In Section 3, we summarize our profiling methodology and its implementation. We explain our field externalization approach, notable improvements, and the integration into Native Image in Section 4. Section 5 presents the results of applying our approach to a large set of benchmarks. We explain our evaluation methodology and discuss the results in detail. In Section 6, we list the limitations of our approach. In Section 7, we compare our techniques to related work.

2 Background

We integrated our approach into GraalVM Native Image [49, 78, 79] based on GraalVM 23.1 with Java version 21: We use its profile-guided optimization feature [5, 9, 53, 64, 78] to first collect information about field values in a profiling run on an instrumented executable and subsequently externalize rarely written fields based on this profiling information during the compilation of the final executable.

2.1 GraalVM

GraalVM is a high-performance, polyglot virtual machine [49, 82] designed to run programs written in a wide range of languages. It can handle traditional Java-bytecode languages, such as Java, Kotlin, and Scala, as well as languages like JavaScript [50], Ruby [52], Python [48], and even C/C++[57]. The GraalVM compiler is a state-of-the-art JIT compiler within GraalVM [16, 38, 67]. It uses a graph-based intermediate representation (the Graal IR) [16, 17] and performs optimizations such as inlining, constant folding, loop unrolling, and escape analysis [37, 82]. When used as a JIT compiler in a regular Java Virtual Machine (JVM), GraalVM collects run-time information to guide its optimizations. For instance, it identifies hot methods, frequently taken branches, and targets of virtual calls. By exploiting this data, the compiler can further optimize the application according to the observed workload. However, relying on run-time profiling data means that optimizations only take full effect once the application has been running long enough to gather representative insights. This can lead to slower startup times.

2.2 GraalVM Native Image

To provide fast startup times and reduced resource usage, GraalVM offers Native Image [78, 79], an AOT compiler that compiles a Java application ahead of time into a self-contained native binary (the so-called *image*). This binary bundles the application code, all necessary libraries, and a lightweight runtime called *SubstrateVM*, which provides essential services such as threading and garbage collection, into a single executable [46]. Once compiled, no standard JVM is needed at run time, and the startup performance can be improved by up to two orders of magnitude compared to the Java HotSpot VM [79], which can be critical for short-lived processes, serverless functions, or microservices. A key concept in GraalVM Native Image is the closed-world assumption [79]. Under this assumption, all classes, methods, and fields that may possibly be accessed at run time must be known at compile time. This contrasts with the traditional JVM, where classes can be dynamically loaded at run time. The closed-world assumption allows GraalVM Native Image to perform powerful whole-program static analyses.

2.2.1 Points-to Analysis

One important step performed by GraalVM Native Image during static analysis is the points-to analysis [7, 24, 25, 26, 39, 40, 60, 61, 66, 68, 71, 80]. This analysis determines which objects, fields, and methods can be reached from the program's entry point. The points-to analysis in GraalVM Native Image is context-insensitive, flow-insensitive for fields, but flow-sensitive for local variables (since the input language is in SSA form) [79]. It is type-based and uses saturation to prune the inter-procedural pointer assignment graph while the analysis is running. This makes the analysis as scalable as a rapid type analysis, while preserving most of the precision of the points-to analysis [80]. By understanding which objects point to which other objects (and thus which fields and methods are relevant), Native Image can identify what parts of the code are actually needed. Points-to analysis builds a global, static view of the application's data flow. The analysis can reveal, for example, that a certain type's fields are never written to or that certain methods are never invoked. Since Native Image has all code available at compile time, it utilizes this analysis to optimize the final binary.

2.2.2 Image Heap

In the context of GraalVM Native Image, the *image heap* refers to a specialized, pre-initialized memory region that is embedded into the native executable [79]. During the Native Image compilation process, GraalVM analyzes the application to determine which objects that are needed at run time can already be created at compile time and stored in the image heap. For example, `java.lang.Class` objects that represent the type descriptors of objects in Java are stored there. Furthermore, class initializers can often be already executed at compile time. Since many objects are already present in the image heap, the native executable doesn't need to perform extensive object allocation and initialization at startup. This leads to significantly reduced startup times compared to traditional JVM applications that rely on JIT compilation and need to allocate all objects at run time [79].

2.2.3 Profile-Guided Optimization

Unlike a JIT compiler, which relies on program behavior observed at run time, an AOT compiler has no direct knowledge of how code executes under real workloads. Without further input, it must rely solely on static heuristics to guide optimizations, which can limit the potential improvements. Profile-guided optimization (PGO) addresses this limitation [5, 9, 53, 64, 78]. PGO is a technique in which the AOT compiler is supplied with run-time execution data collected from a representative workload. This approach mirrors how a JIT compiler uses run-time profiling, but shifts the process to compile time via a two-stage compilation workflow:

1. **Instrumentation and Profiling Phase:**
   First, an instrumented version of the application is generated by Native Image. This instrumented binary contains additional logic to record information about the program as it runs. When executed with a suitable workload that reflects real usage scenarios, the instrumented binary collects detailed run-time data, including call frequencies, branch probabilities, and type occurrences. At the end of this run, the collected profiling information is stored in a file.
2. **Optimized Compilation Phase:**
   In the second stage, the profiling data is fed back into Native Image to produce a new, optimized binary. Equipped with the recorded execution patterns, the AOT compiler can apply more informed optimizations, such as refining method inlining decisions and optimizing frequently executed hot paths.

This two-phase compilation adds complexity and overhead to the build process. However, once the optimized binary is produced, it can be deployed and run without incurring any additional overhead.

PGO enables AOT compilers to tailor optimizations to specific workloads. When the workload employed during the instrumentation phase closely resembles the production environment, the resulting binary is finely tuned for that particular scenario. In essence, PGO provides workload-specific optimizations comparable to those offered by JIT compilers, but in a context where all code is precompiled to native form. This technique is also utilized in related work [10, 22, 35, 41, 73, 81, 83].

It is important to distinguish PGO from machine-learning-based optimization techniques [59], which typically utilize a larger and more diverse set of inputs to optimize a broader range of applications. The objective of machine learning is to train models that can reliably predict optimization opportunities in *new programs* based on extensive training datasets. In contrast to that, PGO aims to optimize a program for a *specific workload*.

Developing a generalized version of the program that performs efficiently across different workloads is an explicit non-goal of PGO. Thus, our evaluation is conducted accordingly. We employ the same workload for both profiling and measurement phases, although the profiling workload is smaller in size compared to that used for measurements.

3 Field Profiling

To enable field externalization, we collect two kinds of information: information that is gathered statically during the compilation of the instrumented program (e.g., information on objects in the image heap) as well as information that is gathered dynamically during the execution of the instrumented program.

The information that is computed at compile time is only available very late in the compilation process, e.g., after the compilation of all methods or when computing the layout of all types. Field externalization, however, has to be initiated early to adapt the corresponding types and field accesses. Therefore, we want to communicate such information from the first compilation (for the instrumented binary) to the subsequent, optimizing compilation. We do this by storing compile-time profiling information in the compiled binary itself. When executing the instrumented binary, we collect the run-time profiling information and merge it with the static profiling information stored in the binary. This combined profiling information is then stored in the profiling file and subsequently used for optimizations in the final compilation of the program.

3.1 Information Needed for Field Externalization

As it is expensive to write a non-default value to an externalized field, we need to make sure to only externalize fields where such writes happen rarely compared to the number of allocated objects of that type. Therefore, we need to know how many objects of a certain type are allocated. For each field, we also need to know in how many objects the respective field ever holds a non-default value. Furthermore, we need information about the size of a type, so that we can calculate how many bytes we need to remove from a type to achieve an actual size reduction (taking alignment into account). The points-to analysis (cf. Section 2.2.1) of GraalVM Native Image is able to detect fields that are unused; they will be removed automatically. We don't want to hinder that optimization and thus have to detect and prevent externalization of such fields. Finally, as we cannot safely externalize fields that are accessed in an unsafe context (via `sun.misc.Unsafe`) or in a VM-internal context, we also have to store information about incompatible accesses per field.

3.2 Metrics

We present our collected metrics in Table 1. The *level* indicates whether the metric is collected per type or per field. Details of the individual metrics are discussed below.

Allocations. To accurately track the number of allocations per type, we consider both compile-time information and run-time information. As explained in Section 2.2.2, Native Image already allocates objects at compile time and stores them in the *image heap*. Thus, we count those instances before they are written into the final binary. We track the objects allocated at run time during garbage collection, i.e., for each object allocation, we increment the counter for the respective type.

Table 1 Overview of all metrics we collect. The *level* describes the granularity at which the tracking is performed. The *time* indicates whether a metric is tracked at run time (RT) and/or at compile time (CT).

Metric	Level	Time	Description
allocations	Type	RT+CT	The number of allocations of a type.
size	Type	CT	The exact and unaligned size of the instances of a type. Used to calculate how many bytes need to be externalized to achieve a reduction of the instance size for that type.
non-default value count	Field	RT+CT	The number of objects of a type that hold a non-default value for a specific field at any point during the execution of the program.
access kind	Field	CT	Indicates whether the field is eligible, unused, or accessed via reflection, in an unsafe context, or in a VM-internal context.

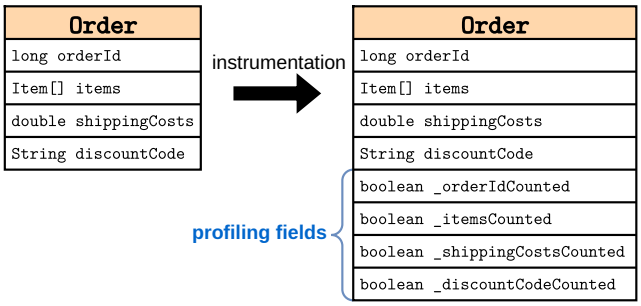


Figure 2 Instrumentation of the `Order` class from Figure 1 to track the *non-default value counts*. A profiling field is added for each original field.

Size. The size of a type is computed late in the compilation process, whereas field externalization requires this information early in the compilation process. Hence, when compiling the instrumented program, we store the types sizes in the profiling data.

Non-default value count. We track this metric at field level. For each field, the metric represents the number of objects that – at any point during the execution of the program – held a non-default value in this specific field. For example, if a non-default value is written ten times to the same field for a single object, the *non-default value count* is only increased by 1. This metric is important for the selection of externalized fields, as only fields that have a low non-default value count – compared to the number of objects allocated of that type – are good candidates for field externalization. We do not count the total number of writes to a field, as a single non-default-value write is enough to cancel out the achieved memory reduction for a single object due to the required companion object. Similar to the *allocations* metric, we also consider the objects in the image heap for this metric.

To identify first-time non-default-value writes for each field/object combination, we instrument all types in the profiling run. For each field, we generate an additional boolean field that indicates whether a non-default value has already been written for that field/object combination, as can be seen in Figure 2. To correctly track the *non-default value count*, we have to instrument all field writes. The instrumentation of field accesses is depicted in Figure 3. First, we check whether the value written to the field is a non-default value, i.e., in the example, we check whether `x` is not `null`. In that case, we verify that the field has not yet been written for this object. If so, we set the profiling field to `true` and increment the counter associated with the *non-default value count* metric for that field.

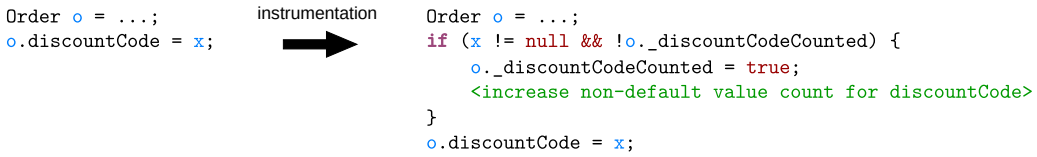


Figure 3 Instrumentation of a write to the `discountCode` field from the example in Figure 1.

Access kind. Native Image performs extensive static analysis, including points-to-analysis (cf. Section 2.2.1), and marks all unsafe, reflection, or JNI accesses that it can detect at build time. Users can provide configuration files to make additional components available for these kinds of accesses. Therefore, we are able to identify where and how fields are accessed. However, again, this information is only fully available late in the compilation process, and thus, similar to the size metric, we store that information in the profiling data when compiling the instrumented program. The access kind denotes one of three cases: *Unused*, when the field is unused and thus deleted by Native Image, *incompatible*, when the field is accessed using reflection, in an unsafe context or in a VM-internal context, or *eligible*, for fields that do not fall into any of the other categories. Only *eligible* fields are considered relevant for field externalization.

4 Field Externalization

The goal of field externalization is to reduce the memory footprint of an application by reducing the size of objects. To do so, we use heuristics based on the profiling information described before to identify fields where the default value dominates within types and to determine whether removing them from a type is beneficial. If we remove fields, we have to adapt their accesses within the compiled code: In case we ever write a non-default value, a companion object has to be allocated that consequently stores this field value. Subsequent accesses to externalized fields of the object then have to access this companion object. To actually benefit from field externalization, we need to minimize the number of required companion objects by tuning the heuristics accordingly.

4.1 Externalization Heuristic

Early in the compilation of a program, we load the profiling information generated during the instrumented run. For every type, we query the profiling data for its fields and use the metrics described in Section 3.2 to determine whether externalization is beneficial. Notably, we disallow externalization of types for which we do not have profiling data or for which the profiling reports no allocations. First, we determine a set F of all externalizable fields of a type as follows:

1. We only externalize fields where the *access kind* shows *eligible* accesses, i.e., we don't support externalization of *unused* or *incompatible* fields.
2. We determine the fraction of *non-default values* in all allocations, i.e., the percentage of instances where the field at some point held a non-default value. We compare this with a configurable threshold and mark the field for externalization if its non-default value fraction does not exceed this threshold:

$$\frac{\text{non-default value count}}{\text{allocations}} \leq \text{externalization threshold}$$

Based on experiments, we use 5% as a default value for this threshold because it yielded the overall best results in our benchmark set.

While this approach yields all externalizable fields of a type, we still need to determine whether externalization of those fields actually shrinks the objects of this type. Consider the `Order` class from the initial example in Figure 1. Figure 4a depicts the initial memory layout of an object of this class: The object header typically takes up 8 bytes. *Compressed object pointers* [45, 79] reduce the size of reference fields to 4 bytes. As the figure shows, this results in an overall size of 32 bytes. Native Image aligns objects at 8-byte boundaries [79]; with a size of 32 bytes this requirement is already met. The requirement for externalization is to reduce the size to at least the next smaller alignment boundary (24 bytes). Otherwise, the reduction in the type size is canceled out by the alignment. In the example, this means that we have to reduce the overall size by 8 bytes. We have to take into account that our externalization approach injects a new reference field (4 bytes) into the type that points to a companion object if this is needed. Therefore, the required reduction in the example increases to 12 bytes.

Figure 4b shows the result of externalizing the `shippingCosts`, an 8-byte `double` field. Unfortunately, if we consider the 4 bytes of the injected `_ref` field, the object size is only reduced to 28 bytes and alignment brings this up again to 32 bytes. The situation is different when externalizing both `shippingCosts` and `discountCode`, as shown in Figure 4c: The overall reduction (including the injected field) is now 8 bytes, which reduces the object size to 24 bytes – again at an 8-byte-alignment boundary.

Profiling information gives us the size t_size of each type before alignment (cf. Section 3.2). Assuming the injected field size to be ref_size, and an alignment of 8 bytes, the minimal required amount of bytes that must be externalized is calculated as follows:

$$\text{min. externalization bytes} = ref_size + \begin{cases} 8, & \text{if } t_size \bmod 8 = 0 \\ t_size \bmod 8, & \text{otherwise} \end{cases}$$

Then, we use the following formula to determine whether externalization is beneficial for the set of theoretically externalizable fields F that we derived in the prior step:

$$\sum_{f \in F} sizeof(f) \geq \text{min. externalization bytes}$$

If the result is to externalize the fields F, we subsequently delete them from the type. For each type with externalized fields, we create a corresponding *companion type*, a synthetic class that only contains the externalized fields of the class. Subsequently, we inject the `_ref` field – also called *companion reference field* – that may reference a companion object if needed into the type with externalized fields.

4.2 Rewiring Accesses to Externalized Fields

Externalizing fields into companion classes is only the first step in the externalization process. Next, we have to adapt all accesses to externalized fields, as shown in Figure 5. Values of externalized fields are now stored in the corresponding fields of the companion object. This object, however, should only be allocated if we write a non-default (non-zero/non-null) value to an externalized field. Therefore, regardless of the access kind, we have to make a case distinction on whether a companion object exists. For each object, we store this information in an additional header bit, which we call the *companion allocated (CA)* bit. If the bit is unset, no companion object exists and the fields never held anything other than the default value. This is the ideal case and our heuristics mentioned in Section 4.1 are tuned to ensure that this is the most frequent case. Hence, we have to adapt the field accesses in such a

object structure	size (bytes)
object header	8
long orderId	8
Item[] items	4
double shippingCosts	8
String discountCode	4
before alignment	**32**
after alignment	**32**
req. externalization	**12**

(a) Layout of an `Order` object from Figure 1 assuming an 8 byte object header, *compressed object pointers* [45, 79], and 8 byte alignment [79].

object structure	size (bytes)
object header	8
long orderId	8
Item[] items	4
double shippingCosts	0
String discountCode	4
Order$Companion _ref	4
before alignment	**28**
after alignment	**32**
size reduction	**0**

(b) Layout of an `Order` object where `discountCode` has been externalized but overall object size remains the same due to the required addition of a `_ref` field.

object structure	size (bytes)
object header	8
long orderId	8
Item[] items	4
double shippingCosts	0
String discountCode	0
Order$Companion _ref	4
before alignment	**24**
after alignment	**24**
size reduction	**8**

(c) Layout of an `Order` object where `discountCode` and `shippingCosts` have been externalized, resulting in a reduction of the object size by 8 bytes.

Figure 4 Memory layout of the `Order` class from Figure 1 before and after externalization.

way that this case is still efficient. If the bit is set, it indicates that an object already has a companion object. This should be the outlier case which we want to avoid as much as possible, as the companion object consumes additional memory.

Read Accesses. Figure 5a shows a read access to the `discountCode` field of the `Order` class. Figure 5c depicts the adaption of the read access when the field is externalized. At every access to an externalized field, we have to introduce a header bit check (CA bit) that tells us whether the object already has a companion object. If the bit is unset, the read is trivial as this tells us that the field still holds its default value. This is the fast path and should be the default case. Thus, we can simply yield the default value as a result of the read access. If the bit is set, the companion object must exist and we have to load it from the `_ref` field to get the actual field value.

Write Accesses. For write accesses, as shown in Figure 5b, we need to do more work, as each write may require the allocation of a companion object. Figure 5d depicts the adaption of write accesses to an externalized field of the `Order` class. Again, the CA bit tells us whether the object already references a companion object. If a companion object exists, we write the new value to the corresponding companion object field. If the bit has not been set, we have to check the value that is written. If we again want to write the default value (null or zero) to this field and the companion object does not exist yet, we can simply skip the write as the field is guaranteed to still hold the default value. This is the optimal case and should be the most frequent path. However, if we write a non-default value, we have to allocate a new companion object and write the value to the field in the companion object. Additionally, we have to store the companion object in the original object and set the CA bit to communicate this change to future accesses. This is the slow path. After the companion object has been allocated, all the benefits that we achieved for this particular object by reducing its size are voided. Thus, our heuristics should minimize the frequency of this case.

```
Order o = ...;
var x = o.discountCode;
```

(a) Read access to `discountCode` before field externalization.

```
Order o = ...;
o.discountCode = y;
```

(b) Write access to `discountCode` before field externalization.

```
Order o = ...;
var x;
// check whether companion exists
if (<ca_bit_set>(o))
  x = o._ref.discountCode;
else
  x = null; // use default value
```

(c) Read access to `discountCode` after field externalization. Since the field no longer exists within the original type, we either yield the default value or have to get the field value from the companion object depending on the result of the CA bit check.

```
Order o = ...;
if (<ca_bit_set>(o)) { // check whether companion exists
  o._ref.discountCode = y;
} else if (y != null) { // default value check
  var _cmp = new Order$Companion(); // allocate companion
  _cmp.discountCode = y; // write companion field
  o._ref = _cmp; // store companion in original object
  <set_ca_bit>(o); // mark existence of a companion
} // no write if we write the default value
```

(d) Write access to `discountCode` after field externalization. Depending on whether the CA bit is set, we may have to allocate a companion object if the value to write is not the default value.

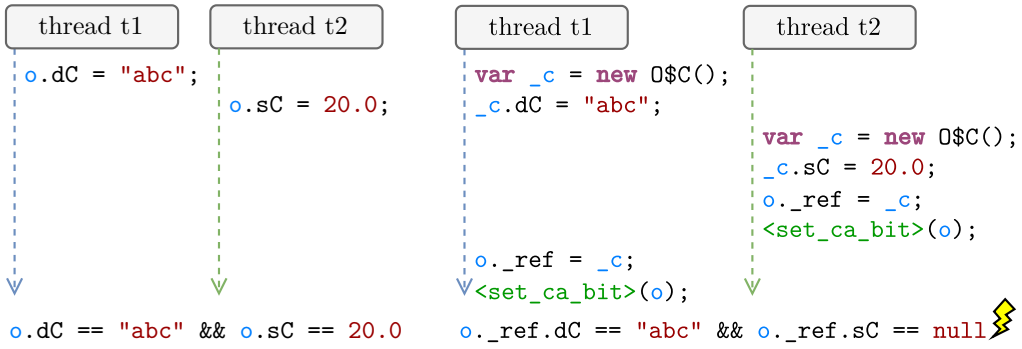
Figure 5 Accesses to fields of the `Order` class before and after field externalization.

4.2.1 Adhering to the Java Memory Model

The adaptations necessary to handle accesses to externalized fields are trivial in principle. However, it gets more complicated when we consider their effects in the context of the Java Memory Model, which defines the language semantics in multithreaded environments [43]. Particularly, the writing process is affected, as a write to an externalized field may now consist of multiple instructions, which may interfere with each other if executed on the same object in different threads. Figure 6a shows the behavior of two write accesses to fields of the same object in multiple threads without externalization. After the writes have been completed, both values are eventually [2] visible in the object. Figure 6b shows what could happen when writing to externalized fields: Here, we assume that the target object does not have a companion object yet – the CA bit is unset – and both threads want to write non-default values to two independent fields, forcing allocation of a companion object. *Thread t1* first allocates a companion object and stores a value to the first field. Then, *thread t2* resumes, also allocates a companion object and stores a value to the second field. In the example, *t2* immediately stores the companion object in the original object and sets the CA bit. When *t1* resumes, it also stores its companion object in the original object and thus overwrites the `_ref` field. This results in a *lost update* as the value written in thread `t2` is lost. In fact, the order of the writes does not even matter: the allocation of two different companion objects for the same original object is a problem in terms of the memory model. Therefore, we need to introduce a synchronization mechanism for externalized field writes.

As read accesses can simply rely on the CA bit to determine whether a companion object exists, they require no modification. If another thread writes a non-default value to the same field at the same time and thus allocates a companion object, this would result in a race condition in the case of a non-synchronized access even without field externalization. With field externalization, a race condition may occur at the header bit access. However, results of this race condition are always within the set of possible results of the original race condition.

[2] Omitting the fact that non-volatile writes within threads may not be visible to other threads immediately [43] – the overall problem still remains.



(a) Write access to `Order` fields from multiple threads before field externalization.

(b) Write accesses to `Order` fields from multiple threads after field externalization if the companion object was not created before. As two threads may simultaneously allocate companion objects and store them in the original object, the result is a lost update as the companion object from thread `t2` is overwritten by the one from thread `t1`.

Figure 6 Write access in multiple threads to fields of the `Order` class before and after field externalization. For brevity, we abbreviate the fields `discountCode` and `shippingCosts` with `dC` and `sC`, and the companion class `Order$Companion` with `O$C`.

Figure 7 describes the synchronization process we introduce to correctly mimic field write semantics in our `Order` example. The dashed green path denotes the fast path and thus the most frequent one. The initial steps remain the same: We check whether a companion object exists (via the CA bit). If so, we write the value into the companion object. Otherwise, we check whether the write concerns the default value. If that is the case, the write can be omitted. The process differs when we write a non-default value and thus require a companion object. First, we allocate the companion object (**A**) and assign the field value (**B**). Note that this can be performed by multiple threads at the same time per object. Next, however, we use an additional header bit – the *companion-in-creation (CC)* bit – that indicates whether some other thread currently allocates the companion. We use a *compare-and-set (CAS)* instruction [29] (**C**) that atomically tries to set the CC bit if and only if it has not been set. A CAS is a three-operand instruction that takes an *expected value* (the object header where the CC bit is zero), a *destination operand* (the address of the object header) and a *source operand* (the object header with the CC bit set). Then, it performs the following steps atomically: First, it compares the *expected value* with the value in the *destination operand*. If the values match (the object header does not have the CC bit set), it writes the value in the *source operand* to the *destination operand* (sets the CC bit) and returns the *expected value*. If the values are different, it returns the value in the *destination operand* (the actual object header). Subsequently, we can use the return value of the instruction to determine whether the current thread could set the CC bit. Therefore, the CAS can only ever succeed for a single thread per object. We call this thread the *CC thread*. The CC thread subsequently enters a *protected region*, where it can safely assign the companion object it previously created (**D**) and also set the CA bit (**E**) to signal that a companion object is available now. Any other thread that tried to CAS the CC bit at the same time has to wait until the CC thread has completed the assignment of the companion. We introduce a *spin lock* [30, 65] (**F**) that loops until the CA bit is set. Note that this branch is entered very infrequently, namely only if two threads try to allocate a companion object for the same object at the same time. Once the thread leaves the spin lock, it can safely assume that a companion object exists and can thus access it and perform its field write (**G**).

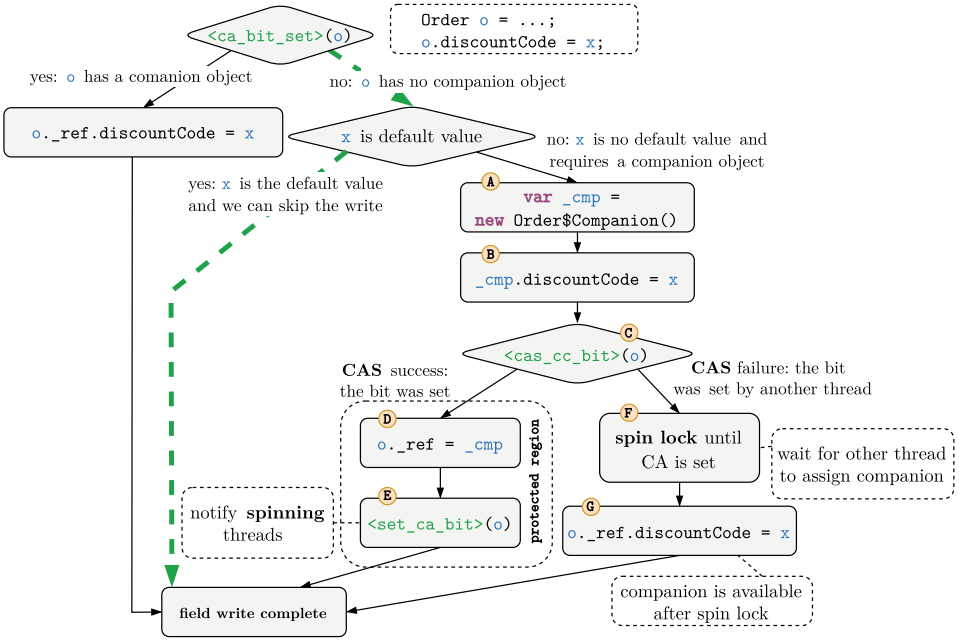


Figure 7 Write accesses to externalized fields including a synchronization mechanism that uses 2 header bits (*CA*, *CC*) to ensure that for a specific object only one thread can assign the companion object. The dashed (green) path is the (most frequent) fast path.

4.3 Compile-Time Externalization

As explained in Section 2.2.2, objects that can be created at compile time, but are needed at run time, are already allocated during compilation and are stored in the image heap. Our externalization process, however, only affects the objects that are created at run time – objects allocated at compile time still adhere to their original layout. To ensure that the objects in the image heap are compatible with the types after externalization, we introduce special handling for all compile-time-allocated objects that have externalized fields. When writing those objects into the image heap, we check whether they contain a non-default value for an externalized field. If so, we allocate a companion object and copy the values of the externalized fields to the companion object. Finally, we link the companion object with the original object (now reduced to its non-externalized fields and the companion reference) and also store the companion object in the image heap. Thus, the object representation in the image heap is compatible with the types after externalization.

4.4 Field Externalization and Class Inheritance

Consider the case when a class `A` has some fields that can be externalized. Suppose that a subclass `B` of `A` also has externalizable fields, making it eligible for externalization according to our heuristic. With our current approach, B would require two companion reference fields (for `A$Companion` and for `B$Companion`), and thus 4 header bits (two bits per reference field). Hence, field externalization does not scale efficiently with class hierarchies. Furthermore, the number of object header bits that can be used for field externalization is limited, hence, preventing us from using more than 2 object header bits. However, as externalizing fields of classes that already have externalized superclasses is beneficial (as it can further shrink the object size), we want to support it as well.

4.4.1 Companion Inheritance

We present the solution to externalizing fields of class hierarchies based on the example in Figure 8. The example shows three classes `A`, `B`, and `C`, where `A` is the superclass of `B` and `B` is the superclass of `C`. In the example, two fields of class `C` can be externalized. However, the superclass `A` also has externalized fields. To solve this problem in an efficient manner, we propose that the companion type that is created for `C` (`C$Companion`) is defined as a subclass of the companion type of the (closest) externalized superclass (here `A$Companion`). Thereby, the problems mentioned above are solved, as at most one companion object is needed for an object. The `C$Companion` instance now contains both: the fields externalized in `C`, as well as the fields externalized in `A`. Therefore, only a single companion reference field and 2 object header bits are needed.

However, this approach introduces an additional challenge: We can no longer determine at compile time which companion object should be allocated when writing to an externalized field. For example, if a method has a parameter of type `A` and subsequently writes a non-default value to the `a1` field, we would either have to allocate an `A$Companion` object or a `C$Companion` object depending on the actual object. Thus, the allocated companion object now depends on the *dynamic type* of an object. We solve this by introducing a run-time type check on the object, but this introduces overhead in terms of performance and code size.

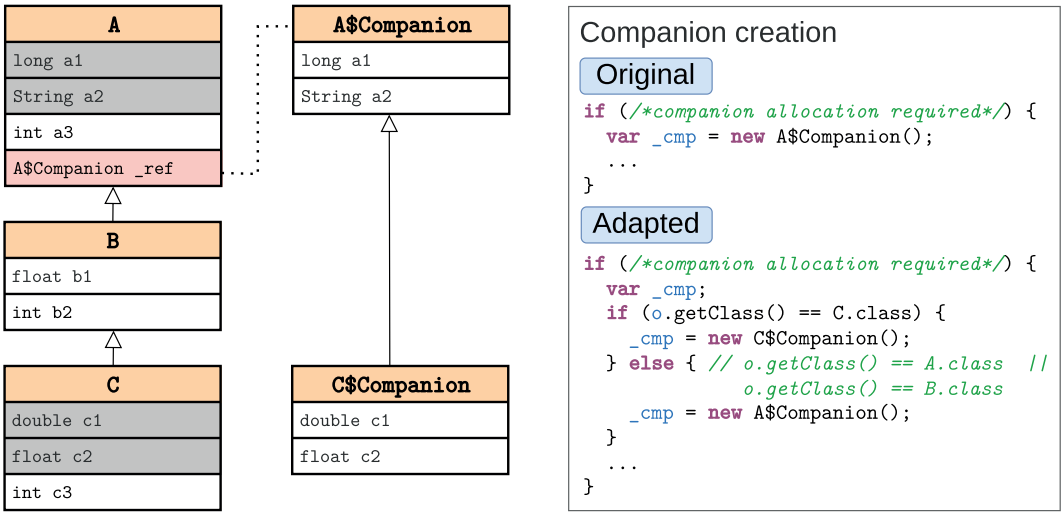


Figure 8 Companion hierarchy for a class hierarchy with multiple types with externalized fields. The necessary adaptations to the companion creation process are illustrated on the right-hand side.

4.4.2 Companion Factory Methods

The companion creation code gets more complicated the more classes of a class hierarchy are externalized. As we need to adapt every write to an externalized field, directly inserting all type checks and allocations of the right companion object within the class hierarchy leads to code bloat. Therefore, we utilize what we call *companion factory methods*. A *companion factory method* is a generated method that receives a type with externalized fields as a parameter and returns a new instance of the matching companion type. Hence, for each root of an externalization hierarchy one companion factory method is generated. That method contains the if-else-if ladder for all externalized subtypes of the externalized root type and returns a companion object of the respective type.

```
...
if (/*companion allocation required*/) {
  var _cmp = createCompanionA(o.getClass());
  ...
}
...
```

```
static A$Companion createCompanionA(Class c) {
  if (c == C.class){
    return new C$Companion();
  } else { // c == A.class || c == B.class
    return new A$Companion();
  }
}
```

Figure 9 Creation of a companion object using a *companion factory method*.

Figure 9 shows an example of a companion factory method and its usage. The *companion factory method* `createCompanionA` is used to create the correct companion object based on the class passed as a parameter. During compilation, we insert a call to this method when writing a non-default value to an externalized field, as shown on the left-hand side of Figure 9. Through these factory methods we can reduce the code bloat. We prevent inlining of these methods to not mitigate their effect but also inform the compiler of their effects to not prevent other optimizations such as partial escape analysis [67].

4.5 Masked Companion References

Our field externalization approach requires adding a reference to a companion object to enable us to handle the cases when non-default values are written to externalized fields. Therefore, externalization also needs to amortize the cost of that additional field, as explained in Section 4.1. Consequently, this limits the applicability of our approach and thus our optimization potential.

To tackle that issue we propose an optimization of field externalization, which we call *masked companion references*. A *masked companion reference* is a non-externalized *reference* field of a type that we reuse as the companion reference field. As long as no companion object is needed, this field is used just like a normal field and stores its normal value. In Figure 10, we again show the `Order` example from the previous sections with the two externalized fields `shippingCosts` and `discountCode`. The remaining `items` field is used as the *masked companion reference*, indicated by the dashed line and light red background. As shown in the first part of the image, the `items` field holds a reference to an ordinary item array, because no companion object is required yet. When a non-default value is written to an externalized field, a companion object is allocated (as explained in Section 4.2) and the reference to the companion object is stored in the `items` field. However, before writing the companion reference, the old value stored in the `items` field needs to be evacuated in order to preserve its value. Therefore, when using *masked companion references*, the companion object contains not only the externalized fields but also the *masked companion reference* field. In the example of Figure 10, when a non-default value is written to the externalized field `shippingCosts` (shown at the right-hand side of the figure), a companion object is allocated and the value that should be stored in `shippingCosts` is stored there. Then, the value stored in `o.items` is evacuated and stored in the companion, and finally, the reference to the companion object is stored in the `items` field (the masked companion reference field) of the `Order` object. By applying this optimization, we do not need to introduce an additional companion reference field that increases the size of all `Order` objects just to hold the reference to an – ideally – rarely needed companion object. If a type does not contain a suitable reference field, we can still fall back to the original approach.

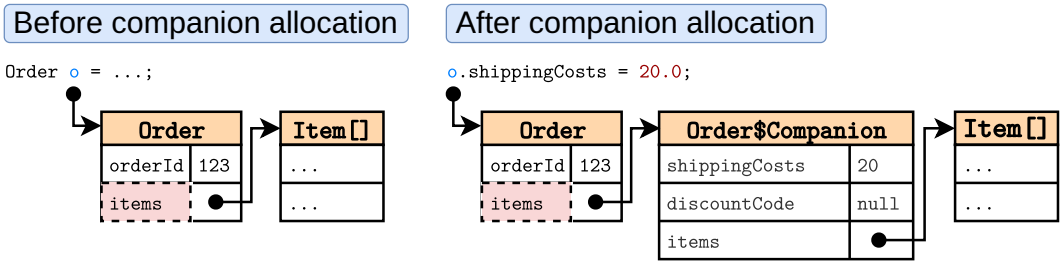


Figure 10 As long as no companion object is need for an object with externalized fields, the *masked companion reference* field (`items`) holds its normal value. When a companion object is needed, the value of the `items` field is evacuated into the companion object and the reference to the companion object is stored in the `items` field.

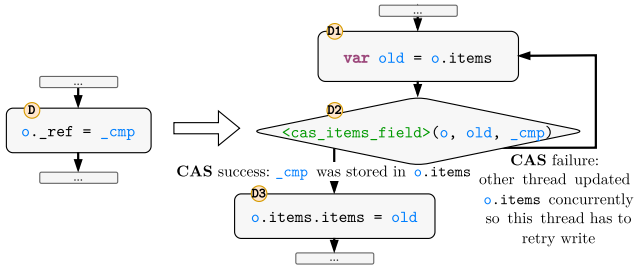


Figure 11 Changes necessary to the process shown in Figure 7 at step **D** for externalized field writes with masked companion references.

4.5.1 Field Access Adaptations for Masked Companion Reference Fields

To accommodate masked companion reference fields we also have to slightly change the field access pattern introduced in Section 4.2.1. First, both field writes and reads now have to access the corresponding masked companion reference to load a companion object if required – the check for its existence is still done via the CA bit. Figure 11 shows the change necessary for writing to an externalized field, as explained in Figure 7. We now no longer have a dedicated companion reference field that points to a companion object. Thus we have to consider both cases: another thread writing a new `Item[]` instance to `Order.items`, and another thread writing a non-default value to an externalized field and thus assigning a newly allocated companion object to the masked companion reference field. Therefore, we have to ensure that both cases are properly synchronized. As step **D** of Figure 7 is in the protected region, we can at least be sure that no other thread concurrently tries to write a companion object. However, regular writes to the masked companion reference field are not guarded by the CC bit. Therefore, we have to make sure to check for concurrent writes when storing the newly allocated companion object. We first load the original value from the companion reference field `items` (**D1**). Then, we use a CAS instruction (**D2**) to atomically check, whether the `old` value in `o.items` has changed. If not, the CAS instruction stores the companion object (`_cmp`) in the `items` field. The result of the CAS tells us, whether the companion object was stored. If the value did change between reading the field and the CAS (and the companion object was not stored), we repeat steps **D1** and **D2** until they succeed. While this is an expensive process, we expect it to be very infrequent. After the CAS, we still have to store the old value of the masked companion reference (`old`) in the corresponding companion object field (`o.items.items` in **D3**).

Writing to a Masked Companion Reference Field

Since the masked companion reference field is a regular field within the original object, we have to adapt accesses to it. When writing to the field, we have to check whether it already references a companion object. The necessary steps are outlined in Figure 12. Once again, the – ideally most frequent – fast path is highlighted with dashed (green) lines. In step **A**, we load the value of the masked companion reference field `o.items`. The CA and the CC bits tell us whether the loaded value might already be a companion object (**B**). If one of the bits is set, we specifically check whether the CA bit is set (**C**). If not, we know that the CC bit is set and that another thread is currently assigning the companion object to this object. Hence, we *spin lock* until this process is complete (**D**). If the CA bit was set or after the spin lock, we know that a companion object must exist and thus assign the value to the corresponding `items` field *within* the companion object (**E**). Note that we cannot use the value read before from the masked companion reference field (`old`), as this value may not have been a companion object at that point. Therefore, we re-read the companion object and assign the actual value (`x`) to its field (**E**).

If step **B** showed that no bit was set, we try to directly store the value in the masked companion reference field (**F**). As this may again interfere with similar accesses in other threads, we use an atomic CAS instruction. This CAS compares the previous value in the masked companion reference field (`old`) with the value currently stored in the field. If this comparison succeeds, it stores the desired value `x` in the masked companion reference field. In this case, we are done with the field write. If the CAS did not succeed, we have to perform a similar procedure as in **B**, as some other thread has changed the value of the field. We cannot know whether the field still holds a *regular* value or a freshly allocated companion object and thus have to check the header bits again (**G**). If any bit was set, we make sure to wait until the CA bit is set (which ensures that the other thread successfully wrote the companion) and only then write the value to the companion object. If no bit was set, we know that the other thread writing the same field wrote another regular value (i.e., some other `Item[]` array) to the field. We consider this a race condition (*racy write*) and can thus skip updating the field.

Reading from a Masked Companion Reference Field

In contrast to read accesses to externalized fields, reading from a masked companion reference field also requires some synchronization, as another thread may concurrently assign a companion object to the same field. We discuss the necessary steps in this process based on Figure 13 – the dashed green path is the fast path. First, we load the current value of the masked companion reference field (`o.items`). Then, similar to the writing process, we check for either the CA or the CC bit, as either tells us that the previously loaded value might be a companion object. If any bit is set, we specifically check the CA bit (**C**) and then either first wait for it to be set or immediately access the corresponding field within the – now safely available – companion object (`o.items.items` in **D**). If no bit was set, we know that the loaded value from step **A** cannot be a companion object and thus, we simply use this value as a result of the field read (**E**).

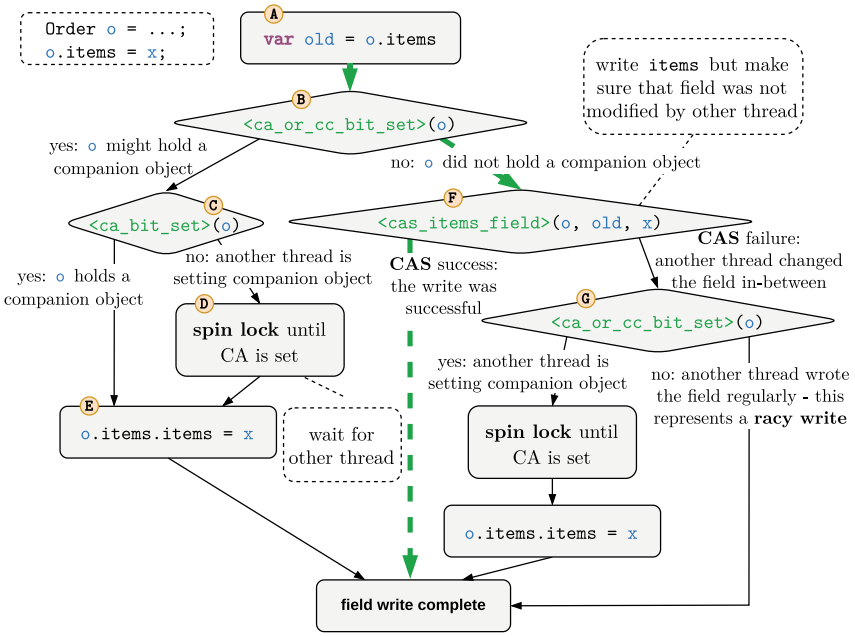


Figure 12 Write access adaptations to masked companion reference fields that take into account multithreaded accesses. As a masked companion reference field can be used both for regular writes and for storing the companion object, synchronization is more expensive than for regular externalized field writes.

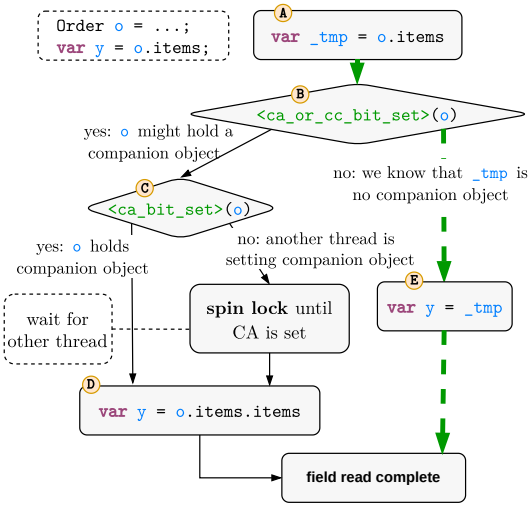


Figure 13 Read access adaptations to masked companion reference fields that take into account multithreaded accesses.

5 Evaluation

We conducted our evaluation on a comprehensive, mixed benchmark corpus. This corpus blends classical benchmark suites (*DaCapo 9.12-MR1-bach* [6], *Scala-DaCapo 0.1.0* [63], *Renaissance 0.9.0* [54]) with additional benchmark suites designed for Native Image performance assessment, all built atop leading Java microservice frameworks – namely Spring [75], Quarkus [56], and Micronaut [58]. These benchmarks encompass both single- and multi-threaded applications. Note that we excluded standard benchmarks that are not supported by the version of Native Image on which our approach is based. Furthermore, our experiments revealed an infrequent bug within GraalVM Native Image related to inlining that distorted our measurements for the *scala-dacapo factorie* benchmark. This bug was reported but has not been fixed yet. Thus, we exclude the benchmark from both analysis and evaluation. Each framework-based benchmark suite includes a concise "helloworld" scenario based on its respective launcher or "getting started" material, thus showcasing the fundamental functionalities of the framework. In addition, every suite contains a second, more complex benchmark with varying workload parameters (tiny, small, medium, etc.). Specifically, the Spring suite employs a tailored version of the Spring Boot PetClinic Sample Application [76]. Quarkus incorporates a microservice benchmark originating from the Apache Tika Quickstart [55]. Micronaut, in turn, provides a second benchmark known as *ShopCart*, a web shopping application. Contemporary works on Native Image also use these microservice benchmarks in their evaluation [7, 8, 34, 42, 80].

Benchmark Methodology

We compared the standard version of GraalVM Native Image to our modified version of Native Image, which automatically externalizes rarely used fields. Both configurations involved an initial profiling run for PGO, followed by the actual measurement run for each benchmark. In standard benchmarks, profiling runs a single iteration, while measurement runs multiple iterations with warm-up. Microservice benchmarks use varying workloads in measurement to assess latency, peak performance, and response time, but only peak-performance is used for profiling. We executed each configuration 8 times – each execution involved a new profiling run to get new profiling data. The experimental platform was an Intel I7-4790K @ 4.4 GHz with 20 G of main memory. Hyper-threading, frequency scaling, and network access were disabled to mitigate fluctuations. All benchmarks were performed using Java 21.

5.1 Benchmark Results

We present the results of the evaluation of our approach on standard benchmarks in Figure 14. Figure 15 contains the results on microservice benchmarks. All numbers – unless otherwise noted – represent the geometric mean (GM) of the corresponding metric across all conducted measurement runs. First, we compare the number of overall allocated bytes compared to the baseline (*relative allocated bytes*). It is important to note that the microservice benchmarks are *throughput* benchmarks, i.e., they are executed for a fixed duration and we measure how many *requests* they manage to process (as they all concern web frameworks). The number of requests furthermore represents the performance metric for these benchmarks (compared to the benchmark *time* metric in standard benchmarks). Hence, the allocated bytes in a benchmark execution also depend on the number of processed requests. Thus, we normalize the *relative allocated bytes* by the number of processed requests. This is represented by the *allocated bytes/request* metric (row 2 in Figure 15). In addition to the allocated bytes, we also compare the execution times/throughput as well as the maximum resident set size (RSS) of each benchmark.

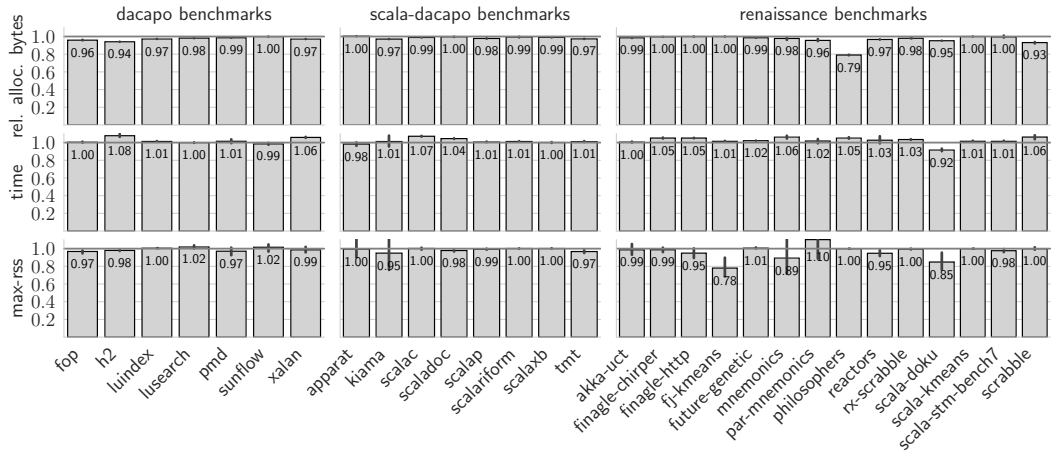


Figure 14 Evaluation of total memory allocation (*relative allocated bytes*), benchmark run time (*time*), and the max-RSS (*max-rss*) on standard benchmarks relative to results on Native Image without our approach.

Allocated Bytes

Because our focus lies on reducing allocated memory, we monitor the total allocated bytes for each benchmark by summing the sizes of newly allocated objects. This metric is also used in related work to evaluate the effectiveness of field externalization [4]. The total allocated bytes are our most stable metric (Figure 14). For this metric, almost all standard benchmarks show reductions, namely 2.72% for *dacapo* (2.74% median reduction), 1.30% for *scala-dacapo* (1.09% median reduction), and 3.60% for *renaissance* (1.41% median reduction). The *renaissance* benchmarks *philosophers* and *scrabble* benefit the most from our approach. In both benchmarks, objects with externalized fields appear frequently and contribute significantly to the overall allocated bytes. At the same time, they do not show many companion objects.

In the microservice benchmarks, the allocated bytes per request show a minor regression in the *tiny* workload of *quarkus tika* (Figure 15). This regression stems from a performance degradation (6.52%) that outweighs the improvement in allocated bytes.

The noticeable differences in the results in the *spring petclinic* benchmarks (Figure 15), where we see large improvements for *huge* and *large* (35.15% and 20.53%, respectively), but only minor improvements for the three smaller workloads, can be explained by the single type `java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject` that shows large fluctuations in its allocations. In the baseline, millions of objects of that type are allocated in each run for all workload sizes. With our approach, we still see millions of allocations for this type in the smaller workloads but only around 22000 allocations in *huge*, thus explaining the significant improvement here. In the *large* workload, the allocation count fluctuates between the baseline values and the 22000 objects, hence the large error.

RSS

Although we use the resident set size to estimate memory consumption, we observed high RSS fluctuations in many benchmarks, as shown in Figure 14 and Figure 15. This is because this metric is highly dependent on the garbage collection behavior in a benchmark. Thus, minor variations in the run-time behavior can lead to different RSS values even for the same

application. Our evaluation shows a max-RSS reduction of 2.55% on standard benchmarks (0.22% median reduction) and a reduction of 2.45% on microservice benchmarks (0.20% median reduction).

Most notable are the results in the *renaissance fj-kmeans* and *scala-doku* benchmarks (21.97% and 15.13% reductions, respectively) – despite a considerable error, their results are consistently below the baseline. Other *renaissance* benchmarks also show high errors: *mnemonics* seems marginally improved, while *par-mnemonics* shows a considerable regression.

For benchmarks from the *dacapo* and *scala-dacapo* suites, we see mixed results – most benchmarks (e.g., *fop*, *pmd*, *scaladoc*, *tmt*) show minor reductions of the max-RSS, while others (e.g., *lusearch*, *sunflow*, *apparat*) exhibit minor regressions.

In the microservice benchmarks, the results for the *spring petclinic* benchmarks are most promising, with a reduction of 6.08% (5.65% median reduction) and only a minor regression in the *helloworld* benchmark. While our approach also reduced the max-RSS in the *small* workload of the *quarkus tika* significantly, the other benchmarks of the suite show minor regressions. Similarly, the *quarkus* benchmarks mostly show regressions (0.90% GM, 1.19% median reduction).

Auxiliary Metrics

As mentioned in prior sections, we had to introduce additional checks for accesses to externalized fields and masked companion reference fields to ensure validity. Therefore, we expect a performance impact. Indeed, the standard benchmarks show a slight regression in terms of performance (2.14% GM and 1.69% median, Figure 14). For most benchmarks, this performance impact correlates with their improvement in terms of allocated bytes. The most notable exceptions are *scala-dacapo scaladoc*, *renaissance finagle-chirper*, and *renaissance finagle-http*. In all three benchmarks, we actually see many externalized objects. Despite that we also see that our approach increases the allocation counts for some of the most used types or introduces allocations of types that we do not see in the baseline runs. These factors may points towards compilation issues: As we insert additional code at field accesses, we may exceed budgets for certain compiler optimizations such as inlining or loop optimizations. Similarly, we might also prevent escape analysis and scalar replacement of certain allocations. As many of these optimizations also are based on heuristics, these individual regressions might by solved by retuning these heuristics for our approach.

Interestingly, the *renaissance scala-doku* benchmark shows an improvement in terms of run-time performance combined with improvements in other metrics as well. The performance of the microservice benchmarks is also impacted; the number of processed requests is decreased by 6.09% (GM, 5.60% median, Figure 15).

In addition to the charts presented above, we also compare the size of the generated image (the binary size). As we introduce additional code (cf. field access modifications in Section 4.2), new types (companion types), and new methods (cf. companion factory methods in Section 4.4.2) into the compiled binary, we expect a slight increase in the image size. Our evaluation shows that the image size increases by 2.16% (GM, 2.13% median) for standard benchmarks and by 3.36% (GM, 3.21% median) for microservice benchmarks.

Externalization Analysis

To evaluate our externalization approach, including our heuristic, we also evaluated how many objects are affected by field externalization and how many companion objects are created. Table 2 presents our findings for each benchmark. The *externalization ratio* specifies

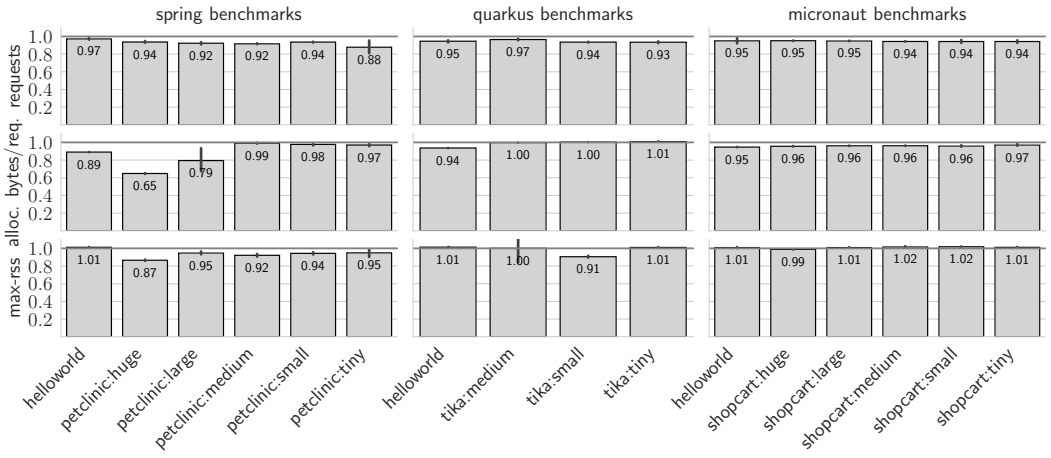


Figure 15 Evaluation of benchmark throughput (*requests*), memory allocation normalized by the performed requests (*allocated bytes/request*), and the max-RSS (*max-rss*) on microservice benchmarks relative to results on Native Image without our approach.

the ratio of objects in the benchmark that had externalized fields, whereas the *companion object ratio* represents the ratio of objects with externalized fields that required a companion object. The ratio of externalized objects varies greatly across benchmarks. For example, there are nearly no externalized objects in *scala-dacapo apparat*, whereas in *renaissance scrabble* 33.4% of the allocated objects have externalized fields. Overall, there is a tendency that benchmarks with a higher externalization ratio profit more in terms of reduction of total allocated bytes from field externalization, as shown by our memory measurements presented above. However, there are outliers as well: For example, the *dacapo xalan* benchmark is the benchmark with the second highest *externalization ratio*, but the allocated bytes are only reduced by 3% and the max-RSS is only reduced by 1%. The reason for the low impact in this benchmark is that the objects with externalized fields only make up a minor part of the overall memory consumption. Thus, big parts of the allocated memory are unaffected by field externalization. Furthermore, the per-object memory savings are quite low in this benchmark, i.e., only few fields are externalized for the types with the most allocations.

The *companion object ratio* should be quite low – in general below 5%, as that is the threshold of our heuristic(cf. Section 4.1). However, as will be discussed in Section 6 there could still be cases with higher companion object ratio even if our profiling information is accurate. In our results, *renaissance rx-scrabble* has the highest *companion object ratio* with 6.5%. We found that `java.util.stream.SliceOps$1`, which is a subclass of the externalized class `java.util.stream.AbstractPipeline` is responsible for creating more than 99% of the companion objects allocated in this benchmark. More specifically, all `java.util.stream.SliceOps$1` instances wrote non-default values to externalized fields and thus required the creation of companion objects. In theory, our heuristic should prevent such cases by not performing externalization when the *non-default value count* is too high. However, our heuristic can only consider the data gathered during profiling, and in this case the ratio of `java.util.stream.SliceOps$1` objects was lower compared to the other subclasses of `java.util.stream.AbstractPipeline` (which did not trigger as frequent companion object allocations) in the profiling run than in the actual benchmark run. One potential reason for this inaccurate profiling information is the reduced workload in the profiling run. We made a similar observation in the *petclinic* benchmarks. Here, the culprit

Table 2 Evaluation of the ratio of objects with externalized fields (externalization ratio, **ER**) compared to the number of total allocated objects. For the companion object ratio (**COR**), the amount of externalized objects that used a companion object was calculated.

Suite	Benchmark	ER	COR
dacapo	fop	9.4%	0.5%
	h2	26.0%	1.4%
	luindex	25.3%	0.0%
	lusearch	9.0%	3.9%
	pmd	4.3%	4.2%
	sunflow	0.3%	0.1%
	xalan	33.3%	1.3%
scala-dacapo	apparat	0.0%	2.1%
	kiama	9.6%	2.9%
	scalac	3.7%	3.4%
	scaladoc	4.3%	2.2%
	scalap	1.0%	1.1%
	scalariform	1.5%	0.8%
	scalaxb	2.7%	0.6%
	tmt	13.8%	0.0%
renaissance	akka-uct	1.9%	0.0%
	finagle-chirper	5.9%	0.1%
	finagle-http	7.4%	0.0%
	fj-kmeans	2.7%	0.0%
	future-genetic	2.9%	0.0%
	mnemonics	16.3%	0.0%
	par-mnemonics	18.7%	0.0%
	philosophers	15.3%	0.0%
	reactors	6.5%	0.1%
	rx-scrabble	8.7%	6.5%
	scala-doku	7.4%	0.2%
	scala-kmeans	0.6%	1.3%
	scala-stm-bench7	1.2%	0.0%
	scrabble	33.4%	0.7%
	Standard Overall	**9.3%**	**1.1%**

Suite	Benchmark	ER	COR
spring	helloworld	9.4%	0.0%
	petclinic:huge	19.0%	5.8%
	petclinic:large	11.3%	5.3%
	petclinic:medium	11.3%	5.7%
	petclinic:small	11.2%	5.1%
	petclinic:tiny	11.4%	4.8%
quarkus	helloworld	18.1%	0.0%
	tika:medium	5.2%	6.0%
	tika:small	5.2%	6.0%
	tika:tiny	5.2%	6.1%
micronaut	helloworld	13.8%	0.0%
	shopcart:huge	11.9%	0.0%
	shopcart:large	11.9%	0.0%
	shopcart:medium	11.9%	0.0%
	shopcart:small	11.9%	0.0%
	shopcart:tiny	11.9%	0.0%
	Microservice Overall	**11.3%**	**2.8%**

is the `org.h2.mvstore.Page` class. We externalize almost all fields of this abstract class because the profiling data reports low usages. However, in the benchmark, we actually allocate a companion object for every `org.h2.mvstore.Page` object, suggesting that the objects of this class behave very differently in the real benchmark run compared to the profiling run. Despite these issues, the benchmark still shows good results across all workloads.

Discussion of Results

While a direct comparison is difficult due to the different benchmarks used, the results of our approach are generally more modest than those of related work [4, 11, 23]. However, this is somewhat expected as our field externalization approach is based on a modern compiler infrastructure without concessions regarding multithreading capability. The various optimizations that are already part of Native Image lower the potential of field externalization; completely unused fields, for example, are already removed by Native Image, so the potential of our approach is reduced.

The evaluation shows that our approach can achieve memory reductions in individual benchmarks while producing no significant regressions on others. Additionally, many results shows a trade-off between performance and memory utilization; some benchmarks achieved reductions in allocated bytes or max-RSS coupled with minor performance regressions. Programs targeted by our approach – cloud-native applications, microbenchmarks, etc. – typically run in environments where both factors, performance and memory usage, contribute to the overall service costs. As our approach can be enabled or disabled on a per-application basis, users can decide whether to use field externalization to, as an example, reduce the footprint of memory-bound applications.

6 Limitations

We prevent profiling and subsequent externalization of fields where we cannot safely adapt their accesses during compilation. We prevent externalization of fields of some internal types of Native Image, such as implementation classes of the garbage collector and the threading implementation. We also exclude types whose fields receive special treatment by the compiler, e.g., they are accessed unsafely via a known offset, or via `VarHandle` [44]. We further exclude volatile fields, as we currently cannot safely mimic their semantics after externalization.

Object serialization in Java makes heavy use of reflection [20]. Similar to how it handles reflection, JNI, and other dynamic features of Java, GraalVM Native Image requires user-supplied metadata to support serialization [47]. Consequently, this information could be used to also exclude fields affected by serialization from externalization.

Deallocation of Companion Objects

We designed our approach such that the allocation of a companion represents a one-way degradation. Hence, once a companion object for an object is allocated, there is no way to deallocate it, even if it is no longer needed at some later point (i.e., if all externalized fields in the companion object hold default values again). Only when the garbage collector collects the original object can the companion object also be collected. Chen et al. [11] solve this by identifying and collecting such "empty" companion objects during garbage collection. However, this requires tighter integration with the garbage collector and also increases the duration of collection cycles. As companion objects should only be created for few objects, empty companions should appear even less frequently. We think optimizing for such a niche case is not necessary.

Deficiencies of the Heuristics

Our heuristics are based on observations on individual fields. While this enables a very precise calculation in terms of the saved memory per externalized field, it also limits the decision on whether or not to externalize at all to thresholds on individual fields. As an example, if a class has 5 fields for which profiling – based on our threshold of 5% – suggests externalization, then our approach would externalize them all. However, the profiling could be misleading here: In the best scenario for our approach, the sets of objects that have non-default values for the respective externalized field are perfectly overlapping. Then only a maximum of 5% of the objects would require a companion object. However, if these sets are completely disjoint, up to 25% of the objects could require a companion object. Identifying such cases via profiling would require profiling the composition of the field values *per object*. However, tracking all field combinations is infeasible due to the associated exponential complexity.

7 Related Work

There is considerable related work on field and general memory footprint optimization techniques. However, most such works are implemented either on research compilers and VMs or in the context of a JIT compiler, which entails a different set of challenges but also opportunities compared to an AOT compiler. Nevertheless, they propose interesting and unique approaches that are in parts comparable to our work or propose other footprint optimization techniques using profiling information.

Chen et al. [11] implemented field externalization in the Kaffe VM [77], a Java Virtual Machine for embedded programs. They profile field usages and classify the fields into three levels based on their profiled value compositions: fields without a dominant value (level 0), fields with a dominant value other than the default value (level 1), and fields where the default value dominates (level 2). At run time, this information is picked up to, on the one hand, strip level 2 fields from objects in a similar manner to our approach and, on the other hand, *share* level 1 fields between objects. Similar to our approach, they use header bits to identify *compressed* (companion object not yet allocated) and *uncompressed/shared* objects (with allocated companion object/shared fields).

While they do not discuss their approach in the context of multithreading in detail, they seem to use some form of synchronization on accesses to externalized and shared fields. They note, however, that they can avoid the synchronization overhead in their target (embedded) JVMs, since they do not allow threads to preempt one another. Our approach is based on a standalone JVM, where multithreading and concurrency are much more widespread. Thus, as detailed in Section 4.2.1, much of the effort of our approach went into designing semantically valid access patterns to externalized fields that conform to the Java memory model. This also complicates integrating their field sharing approach into our work.

Their approach is based on a JIT compiler and ours is based on an AOT compiler. Thus we also have different challenges: They use the interpreter to mark instructions that access externalized fields, which subsequently allows the JIT compiler to optimize them. Our approach is based on a closed-world assumption, thus we have information about all types that occur in the application. However, since there is no interpreter that can perform some levels of profiling when processing objects with externalized fields, all access variants have to be compiled immediately. The results of our approach are hard to compare: They target embedded JVMs and, therefore, use a simulator and the SpecJVM98 benchmark suite [69] that has been retired since to gather benchmark results. We use a set of standard and microservice benchmarks to evaluate our approach.

Guo et al. [23] show an improvement to the aforementioned approach, again targeting an embedded JVM. They use the same field usage classification approach as Chen et al. [11] and generate meta-classes that describe the offsets of externalized and shared fields. As with Chen et al., the differences between our approach and theirs are that they target an embedded JVM, they scan the heap to gather profiling information, and the targeted benchmark set (SpecJVM98). Similarly, they do not mention their synchronization techniques to adhere to the Java memory model.

Ananian and Rinard [4] present a number of optimizations that also include field externalization to reduce the memory footprint of objects. This approach is implemented into a research Java AOT compiler (MIT FLEX compiler system). Their profiling approach is similar to ours; they introduce per-field counters in a separate profiling build and subsequently use a heuristic to select externalizable fields. They do not use companion objects that hold the externalized fields but use a hashtable that maps objects to their field values. The benefit

of using a hashtable is that one single non-default-value write does not produce that much overhead compared to the allocation of the companion object. However, the hashtable itself, as well as the keys, require extra storage and introduce further indirections. Native Image supports lazily appending the identity hash code field and the monitor field to objects [51]. Introducing a hashtable for types with externalized fields would also require the identity hash code field for all types, thus reducing the savings potential.

Sartor et al. [62] present and summarize a number of techniques related to object compression. These also include the original approaches by Chen et al. [11] and Ananian and Rinard [4]. They implemented their approach on the Jikes RVM [2] but performed the evaluation without the optimizing compiler. Hence, comparing the results of our approaches is difficult, as we measured the real effects of field externalization after AOT compilation.

There is also work about other object compression techniques: Venstermans et al. [74] reduce or completely remove object headers in the Jikes RVM [2] by allocating all objects of a specific type in a contiguous memory region and using a *side array* for the header bits and status information used by the garbage collector. Chen et al. [12] implemented a custom garbage collector for the Sun KVM [70] that compresses objects and arrays when space is needed. However, every access to a compressed object subsequently has to decompress it first. Therefore, they also implemented a partitioning mechanism for objects and arrays that also allows lazy allocation of the individual partitions when needed. We see these approaches as orthogonal to our work on field externalization. Their work could allow us to compress objects even further, even after field externalization.

8 Conclusion

In this work, we presented a novel field externalization approach for modern Java VMs that reduces the footprint of objects by removing fields that mostly hold the default value. We use profiling information to identify such fields and subsequently move them into separate *companion classes*, which are generated at compile time. If, at run time, an externalized field is accessed, the corresponding *companion object* is allocated that holds the externalized fields. The companion objects are referenced either via an additional injected field or a regular reference field that is reused as a *masked companion reference field*. Since our approach is integrated into a modern AOT compilation environment, GraalVM Native Image, we also introduce synchronization on accesses to externalized fields by using two header bits. An evaluation on a wide variety of benchmarks shows a modest but consistent reduction of the total allocated bytes across most benchmarks as well as a reduction in the max-RSS.

Overall, our work demonstrates the feasibility of field externalization in a state-of-the-art AOT compiler without sacrificing feature or language support but also highlights the challenges associated with it – particularly when adherence to the language semantics is of critical importance and when there is no run-time compilation or interpreter that enable fallbacks through recompilation.

References

1 Frances E. Allen and John Cocke. A Catalogue of Optimizing Transformations. In *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, Hoboken, NJ, USA, 1972.

2 Bowen Alpern, Clement R. Attanasio, John J. Barton, Michael G. Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan F. Hummel, Derek Lieber, Vassily Litvinov, Mark F. Mergen, Ton Ngo, James R. Russell, Vivek Sarkar, Mauricio J. Serrano, Janice C. Shepherd, Stephen E. Smith, Vugranam C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000. doi:10.1147/sj.391.0211.

3 Amazon Web Services. Serverless Computing - AWS Lambda - Amazon Web Services, 2024. URL: https://aws.amazon.com/lambda/.

4 C. Scott Ananian and Martin Rinard. Data Size Optimizations for Java Programs. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, LCTES '03, pages 59–68, San Diego, CA, USA, June 2003. ACM. doi:10.1145/780732.780741.

5 Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. *Proc. IEEE*, 93(2):449–466, February 2005. doi:10.1109/JPROC.2004.840305.

6 Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, Portland, OR, USA, October 2006. ACM. doi:10.1145/1167473.1167488.

7 Christoph Blumschein, Fabio Niephaus, Codruţ Stancu, Christian Wimmer, Jens Lincke, and Robert Hirschfeld. Finding Cuts in Static Analysis Graphs to Debloat Software. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, pages 603–614, Vienna, Austria, September 2024. ACM. doi:10.1145/3650212.3680306.

8 Rodrigo Bruno, Vojin Jovanovic, Christian Wimmer, and Gustavo Alonso. Compiler-Assisted Object Inlining with Value Fields. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 128–141, Virtual, Canada, June 2021. ACM. doi:10.1145/3453483.3454034.

9 Pohua P. Chang, Scott A. Mahlke, and Wen-Mei W. Hwu. Using Profile Information to Assist Classic Code Optimizations. *Software: Practice and Experience*, 21(12):1301–1321, 1991. doi:10.1002/spe.4380211204.

10 Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. Taming Hardware Event Samples for FDO Compilation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 42–52, Toronto, ON, Canada, April 2010. ACM. doi:10.1145/1772954.1772963.

11 Guangyu Chen, Mahmut Kandemir, and Mary J. Irwin. Exploiting Frequent Field Values in Java Objects for Reducing Heap Memory Requirements. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 68–78, Chicago, IL, USA, June 2005. ACM. doi:10.1145/1064979.1064990.

12 Guangyu Chen, Mahmut T. Kandemir, Narayanan Vijaykrishnan, Mary J. Irwin, Bernd Mathiske, and Mario Wolczko. Heap Compression for Memory-Constrained Java Environments. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 282–301, Anaheim, CA, USA, October 2003. ACM. doi:10.1145/949305.949330.

13 Jiho Choi, Thomas Shull, and Josep Torrellas. Reusable Inline Caching for JavaScript Performance. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 889–901, Phoenix, AZ, USA, June 2019. ACM. doi:10.1145/3314221.3314587.

14 Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape Analysis for Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, volume 34 of *OOPSLA '99*, pages 1–19, Denver, CO, USA, October 1999. ACM. doi:10.1145/320385.320386.

15 L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302, Salt Lake City, UT, USA, January 1984. ACM. doi:10.1145/800017.800542.

16 Gilles Duboscq, Lukas Stadler, Thomas Wuerthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the 2nd Asia-Pacific Programming Languages and Compilers Workshop*, APPLC '13, page 9, Shenzhen, China, February 2013. URL: https://ssw.jku.at/General/Staff/GD/APPLC-2013-paper_12.pdf.

17 Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, pages 1–10, Indianapolis, IN, USA, October 2013. ACM. doi:10.1145/2542142.2542143.

18 Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. Sizeless: Predicting the Optimal Size of Serverless Functions. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, pages 248–259, Virtual, Canada, December 2021. ACM. doi:10.1145/3464298.3493398.

19 Tarek Elgamal, Atul Sandur, Klara Nahrstedt, and Gul Agha. Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement. In *2018 IEEE/ACM Symposium on Edge Computing*, SEC 2008, pages 300–312, Seattle, WA, USA, October 2018. IEEE. doi:10.1109/SEC.2018.00029.

20 Brian Goetz. Towards Better Serialization, June 2019. URL: https://openjdk.org/projects/amber/design-notes/towards-better-serialization.

21 Google. Google Cloud Run Functions, 2025. URL: https://cloud.google.com/functions.

22 Google. Profile-guided optimization – The Go Programming Language, 2025. URL: https://go.dev/doc/pgo.

23 Zhuang Guo, José Nelson Amaral, Duane Szafron, and Yang Wang. Utilizing Field Usage Patterns for Java Heap Space Optimization. In *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '06, pages 67–79, Toronto, ON, Canada, October 2006. IBM Corp. doi:10.1145/1188966.1188974.

24 Shashin Halalingaiah, Vijay Sundaresan, Daryl Maier, and V. Krishna Nandivada. The ART of Sharing Points-to Analysis: Reusing Points-to Analysis Results Safely and Efficiently. *Proc. ACM Program. Lang.*, 8(OOPSLA2):363:2606–363:2632, October 2024. doi:10.1145/3689803.

25 Dongjie He, Jingbo Lu, and Jingling Xue. Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:29, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2022.30.

26 Michael Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, Snowbird, UT, USA, June 2001. ACM. doi:10.1145/379605.379665.

27 Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In Pierre America, editor, *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP'91, pages 21–38, Geneva, Switzerland, 1991. Springer. doi:10.1007/BFb0057013.

28 Urs Hölzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 326–336, Orlando, FL, USA, June 1994. ACM. doi:10.1145/178243.178478.

29 Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture, Chapter 7 - Programming With General-Purpose Instructions*, volume 1 of *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel, Santa Clara, CA, USA, 85.1 edition, October 2024.

30 Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference, A-Z, Chapter 4 Instruction Set Reference, M-U*, volume 2 (2A, 2B, 2C, 2D) of *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel, Santa Clara, CA, USA, 85.1 edition, October 2024.

31 Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A Study of Devirtualization Techniques for a Java™ Just-In-Time Compiler. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 294–310, Minneapolis, MN, USA, October 2000. ACM. doi:10.1145/353171.353191.

32 JavaNexus. Optimizing Java Performance with Thread-Local Allocation Buffers, May 2024. URL: https://javanexus.com/blog/java-performance-thread-local-allocation-buffers.

33 Roman Kennke, Vladimir Kozlov, Aleksey Shipilev, Erik Österlund, John Rose, Stefan Karlsson, and Thomas Stuefe. JEP 450: Compact Object Headers (Experimental), December 2024. URL: https://openjdk.org/jeps/450.

34 David Kozak, Vojin Jovanovic, Codrut Stancu, Tomáš Vojnar, and Christian Wimmer. Comparing Rapid Type Analysis with Points-To Analysis in GraalVM Native Image. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2023, pages 129–142, Cascais, Portugal, October 2023. ACM. doi:10.1145/3617651.3622980.

35 Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization, 2004.*, CGO 2004, pages 75–86, San Jose, CA, USA, March 2004. IEEE. doi:10.1109/CGO.2004.1281665.

36 Junpyo Lee, Byung-Sun Yang, Suhyun Kim, Kemal Ebcioğlu, Erik Altman, Seungil Lee, Yoo C. Chung, Heungbok Lee, Je Hyung Lee, and Soo-Mook Moon. Reducing Virtual Call Overheads in a Java VM Just-in-Time Compiler. *SIGARCH Comput. Archit. News*, 28(1):21–33, March 2000. doi:10.1145/346023.346037.

37 David Leopoldseder, Roland Schatz, Lukas Stadler, Manuel Rigger, Thomas Würthinger, and Hanspeter Mössenböck. Fast-Path Loop Unrolling of Non-Counted Loops to Enable Subsequent Compiler Optimizations. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ManLang '18, pages 1–13, Linz, Austria, September 2018. ACM. doi:10.1145/3237009.3237013.

38 David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. Dominance-Based Duplication Simulation (DBDS): Code Duplication to Enable Compiler Optimizations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO '18, pages 126–137, Vienna, Austria, February 2018. ACM. doi:10.1145/3168811.

39 Ondřej Lhoták and Laurie Hendren. Scaling Java Points-to Analysis Using Spark. In Görel Hedin, editor, *Compiler Construction*, pages 153–169, Berlin, Heidelberg, 2003. Springer. doi:10.1007/3-540-36579-6_12.

40 Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. Precision-Guided Context Sensitivity for Pointer Analysis. *Proc. ACM Program. Lang.*, 2(OOPSLA):141:1–141:29, October 2018. doi:10.1145/3276511.

41 LLVM Project. How To Build Clang and LLVM with Profile-Guided Optimizations — LLVM 18.0.0git documentation, 2025. URL: https://llvm.org/docs/HowToBuildWithPGO.html.

42 Lukas Makor, Sebastian Kloibhofer, Peter Hofer, David Leopoldseder, and Hanspeter Mössenböck. Automated Profile-Guided Replacement of Data Structures to Reduce Memory Allocation. *Programming*, 10(1):43, February 2025. doi:10.22152/programming-journal.org/2025/10/3.

43 Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, Long Beach, CA, USA, January 2005. ACM. doi:10.1145/1040305.1040336.

44 Oracle. Java Development Kit Version 21 API Specification: VarHandle. URL: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/invoke/VarHandle.html.

45 Oracle. Java HotSpot Virtual Machine Performance Enhancements: Compressed Ordinary Object Pointer, September 2023. URL: https://docs.oracle.com/en/java/javase/21/vm/java-hotspot-virtual-machine-performance-enhancements.html#GUID-932AD393-1C8C-4E50-8074-F81AD6FB2444.

46 Oracle. Native Image, 2024. URL: https://www.graalvm.org/latest/reference-manual/native-image/.

47 Oracle. Reachability Metadata, 2024. URL: https://www.graalvm.org/latest/reference-manual/native-image/metadata/.

48 Oracle. GraalPython, 2025. URL: https://github.com/graalvm/graalpython.

49 Oracle. GraalVM, 2025. URL: https://www.graalvm.org/.

50 Oracle. GraalVM JavaScript, 2025. URL: https://github.com/oracle/graaljs.

51 Oracle. Object layout description in GraalVM Native Image when using Epsilon/Serial GC, January 2025. URL: https://github.com/oracle/graal/blob/261c38a7a47dd87f13ec3cbe0fbb85cfdff17963/substratevm/src/com.oracle.svm.hosted/src/com/oracle/svm/hosted/HostedConfiguration.java#L125-L145.

52 Oracle. TruffleRuby, 2025. URL: https://github.com/oracle/truffleruby.

53 Karl Pettis and Robert C. Hansen. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 16–27, White Plains, NY, USA, June 1990. ACM. doi:10.1145/93542.93550.

54 Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 31–47, Phoenix, AZ, USA, June 2019. ACM. doi:10.1145/3314221.3314637.

55 Red Hat. Quarkus: Apache Tika Quickstart, March 2021. URL: https://github.com/quarkusio/quarkus-quickstarts/tree/758dd049407c6aa41c79c0d0dac1cff830ff9217/tika-quickstart.

56 Red Hat. Quarkus, 2025. URL: https://quarkus.io/.

57 Manuel Rigger, Matthias Grimmer, and Hanspeter Mössenböck. Sulong - Execution of LLVM-Based Languages on the JVM: Position Paper. In *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '16, pages 1–4, Rome, Italy, 2016. ACM Press. doi:10.1145/3012408.3012416.

58 Graeme Rocher. Micronaut, 2024. URL: https://micronaut.io/.

59 Nadav Rotem and Chris Cummins. Profile Guided Optimization without Profiles: A Machine Learning Approach, January 2022. doi:10.48550/arXiv.2112.14679.

60 Tobias Roth, Julius Näumann, Dominik Helm, Sven Keidel, and Mira Mezini. AXA: Cross-Language Analysis through Integration of Single-Language Analyses. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pages 1195–1205, Sacramento, CA, USA, October 2024. ACM. doi:10.1145/3691620.3696193.

61 Barbara G. Ryder. Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Görel Hedin, editors, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 126–137. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. doi:10.1007/3-540-36579-6_10.

62 Jennifer B. Sartor, Martin Hirzel, and Kathryn S. McKinley. No Bit Left Behind: The Limits of Heap Data Compression. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 111–120, Tucson, AZ, USA, June 2008. ACM. doi:10.1145/1375634.1375651.

63 Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da Capo con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *Proceedings of the 2011 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '11, pages 657–676, Portland, OR, USA, October 2011. ACM. doi:10.1145/2048066.2048118.

64 Denys Shabalin. *Just-in-Time Performance without Warm-Up*. PhD thesis, EPFL, Lausanne, February 2020. doi:10.5075/epfl-thesis-9768.

65 Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. Operating System Concepts, 10th Edition - Chapter 6: Synchronization Tools. In *Operating System Concepts, 10th Edition*, pages 257–288. Wiley, 10 edition, April 2018. URL: https://os-book.com/OS10/index.html.

66 Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. Alias Analysis for Object-Oriented Programs. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Lecture Notes in Computer Science, pages 196–232. Springer, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-36946-9_8.

67 Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2014, pages 165–174, Orlando, FL, USA, February 2014. ACM. doi:10.1145/2581122.2544157.

68 Codruţ Stancu, Christian Wimmer, Stefan Brunthaler, Per Larsen, and Michael Franz. Comparing Points-to Static Analysis with Runtime Recorded Profiling Data. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 157–168, Cracow, Poland, September 2014. ACM. doi:10.1145/2647508.2647524.

69 Standard Performance Evaluation Corporation. SPEC JVM98. URL: https://www.spec.org/osg/jvm98/.

70 Sun Microsystems, Inc. J2ME Building Blocks for Mobile Devices: White Paper on KVM and the Connected, Limited Device Configuration (CLDC). Technical report, Sun Microsystems, Inc., Palo Alto, CA, USA, May 2000. URL: https://www.oracle.com/technetwork/java/embedded/javame/java-mobile/kvmwp-150240.pdf.

71 Rei Thiessen and Ondřej Lhoták. Context Transformations for Pointer Analysis. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 263–277, Barcelona, Spain, June 2017. ACM. doi:10.1145/3062341.3062359.

72 Po-An Tsai and Daniel Sanchez. Compress Objects, Not Cache Lines: An Object-Based Compressed Memory Hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 229–242, Providence, RI, USA, April 2019. ACM. doi:10.1145/3297858.3304006.

73 Tomoharu Ugawa, Stefan Marr, and Richard Jones. Profile Guided Offline Optimization of Hidden Class Graphs for JavaScript VMs in Embedded Systems. In *Proceedings of the 14th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL 2022, pages 25–35, Auckland, New Zealand, December 2022. ACM. doi:10.1145/3563838.3567678.

74 Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. Java Object Header Elimination for Reduced Memory Consumption in 64-Bit Virtual Machines. *ACM Trans. Archit. Code Optim.*, 4(3):17–es, September 2007. doi:10.1145/1275937.1275941.

75 VMware. Spring, 2025. URL: https://spring.io/.

76 VMware. Spring PetClinic Sample Application, 2025. URL: https://github.com/spring-projects/spring-petclinic.

77 Tim Wilkinson. The Kaffe Virtual Machine, December 2024. URL: https://github.com/kaffe/kaffe.

78 Christian Wimmer. GraalVM Native Image: Large-Scale Static Analysis for Java (Keynote). In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL 2021, page 3, Chicago, IL, USA, October 2021. ACM. doi:10.1145/3486606.3488075.

79 Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. Initialize Once, Start Fast: Application Initialization at Build Time. *Proc. ACM Program. Lang.*, 3(OOPSLA):184:1–184:29, October 2019. doi:10.1145/3360610.

80 Christian Wimmer, Codrut Stancu, David Kozak, and Thomas Würthinger. Scaling Type-Based Points-to Analysis with Saturation. *Proc. ACM Program. Lang.*, 8(PLDI):187:990–187:1013, June 2024. doi:10.1145/3656417.

81 Patrick Wintermeyer, Maria Apostolaki, Alexander Dietmüller, and Laurent Vanbever. P2GO: P4 Profile-Guided Optimizations. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, pages 146–152, Virtual, USA, November 2020. ACM. doi:10.1145/3422604.3425941.

82 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In *Proceedings of the 2013 SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2013, pages 187–204, Indianapolis, IN, USA, October 2013. ACM. doi:10.1145/2509578.2509581.

83 Pengfei Yuan, Yao Guo, and Xiangqun Chen. Experiences in Profile-Guided Operating System Kernel Optimization. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, APSys '14, pages 4:1–4:6, Beijing, China, June 2014. ACM. doi:10.1145/2637166.2637227.

84 Miao Zhang, Yifei Zhu, Cong Zhang, and Jiangchuan Liu. Video Processing with Serverless Computing: A Measurement Study. In *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '19, pages 61–66, Amherst, MA, USA, June 2019. ACM. doi:10.1145/3304112.3325608.

Incremental Computing by Differential Execution

Prashant Kumar
JGU Mainz, Germany

André Pacak
JGU Mainz, Germany

Sebastian Erdweg
JGU Mainz, Germany

Abstract

Incremental computing offers the potential for significant performance gains by efficiently updating computations in response to changing data. However, traditional approaches are either problem-specific or use an inefficient all-or-nothing strategy of rerunning affected computations entirely. This paper presents differential semantics, a novel approach that directly embeds the propagation of changes into the semantics of a general-purpose programming language. Given a precise description of input changes, differential semantics rules define how these changes are tracked and propagated through core language constructs like assignments, conditionals, and loops to produce corresponding output changes. We formalize differential semantics and verify key properties, including correctness, using the Rocq proof assistant. We also develop and formally prove a set of optimizations, particularly for loop handling, that enable asymptotic performance improvements. An implementation of the semantics as a differential interpreter achieves order-of-magnitude speedups over recomputation on the Bellman-Ford shortest path algorithm.

2012 ACM Subject Classification Theory of computation → Operational semantics; Software and its engineering → Formal language definitions; Software and its engineering → Formal methods; Software and its engineering → Incremental compilers

Keywords and phrases Incremental computing, differential semantics, programming language design, formal verification, big-step semantics

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.20

Supplementary Material *Software (ECOOP 2025 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.11.2.13`

Funding This work is supported by ERC grant AutoInc (`https://doi.org/10.3030/101125325`).

1 Introduction

The field of incremental computing aims to develop programming abstractions for describing incremental computations: Computations that react to changes of their inputs efficiently. Most existing approaches follow the idea of selective recomputing [22]: They track which subcomputations are affected by an input change transitively and rerun these subcomputations when their up-to-date result is needed [5, 13]. However, this is an all-or-nothing approach: If a subcomputation is affected in any way, it has to be rerun. For example, consider a grammar checker that processes a large text document represented as a character string. Any change to the text will trigger significant recomputations. Better incremental computing must exploit *how* the input was changed, not just *if* it was changed.

In this paper, we propose incremental computing through differential execution. Here and in the remainder of this paper, we use *differential* to mean "dealing with differences". Hence, differential execution means to execute a program given a description of *how* its inputs are changed. We can exploit these change descriptions in two ways. First, we can use dedicated execution rules that react to input changes directly. For example, when the iteration count

© Prashant Kumar, André Pacak, and Sebastian Erdweg;
licensed under Creative Commons License CC-BY 4.0

of a loop is increased, it is sometimes sufficient to execute the extra loop iterations without repeating the previous ones. Second, we can integrate change-processing language primitives, which are often asymptotically faster than their non-incremental counterpart. For example, we can integrate differential operators from relational algebra, as known from live view maintenance in database systems. This paper focuses on the first aspect: How to provide differential execution rules that exploit how inputs have changed.

We study differential execution based on the big-step operational semantics of a small imperative language that features variable assignments, loops, and conditionals. We explain how to incrementalize these features by developing a differential big-step semantics that reacts to changes in the initial variable assignment. Specifically, while $\sigma \vdash s \Rightarrow \sigma'$ executes statement s in store σ, we define a differential semantics $\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma'$ that executes s based on the store change $\Delta\sigma$. We have implemented our formal model of differential execution in Rocq and use it to reason about the differential behavior of programs. First, we prove that the differential semantics does not get stuck and yields the same result as a recomputation from scratch:

$$\sigma \vdash s \Rightarrow \sigma' \ \wedge \ \sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma' \quad \rightarrow \quad \sigma \oplus \Delta\sigma \vdash s \Rightarrow \sigma' \oplus \Delta\sigma'.$$

And second, we can use the formal semantics to justify a number of important optimizations. For example, we prove that the differential semantics may skip statements that do not read changed variables from $\Delta\sigma$, and we develop provably correct optimizations for the differential execution of loops and conditionals. Compared to prior work on the incremental lambda calculus [9], which also processes change descriptions, we are the first to consider imperative language features and use a completely different methodology: differential semantics instead of type-driven program derivation targeting a standard semantics.

While this paper focuses on the formal development, we also want to demonstrate that the differential semantics enables efficient incremental computations. Unfortunately, a direct implementation of the differential semantics in an interpreter is inefficient because the semantics lacks caching of previously computed information. In particular, the differential semantics $\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma'$ expects the original store σ as input, which has to be computed for each statement unless we cache it. Therefore, we develop a differential interpreter based on the differential semantics, but incorporating caching and some verified optimizations. We demonstrate the efficiency of our differential interpreter empirically based on an imperative implementation of the Bellman-Ford shortest-path algorithm. Given changes to the initial edge weights, our differential interpreter computes updates in the shortest paths orders of magnitude faster than a recomputation could. The differential interpreter is implemented in Scala 3 and available open-source.[1]

In summary, this paper makes the following contributions:

- We define a static typing discipline for changes of a language's values, including typed patching and diffing (Section 3).
- We introduce the approach of differential execution based on the differential big-step semantics of an imperative language. We prove that the differential semantics preserves types, makes progress, and yields the correct result (Section 4).
- We propose, develop, and verify optimizations for differential execution that skip unaffected statements and improve the running time of loops and conditionals asymptotically (Section 5).

[1] `https://gitlab.rlp.net/plmz/artifacts/autoinc-interp-implementation-ecoop25`

- We implement an efficient differential interpreter based on our semantics (Section 6) and use it to incrementalize the Bellman-Ford algorithm, where it yields order-of-magnitude speedups (Section 7).

2 Differential Execution: Goals and Challenges

An incremental computation $\Delta f(\Delta x) = \Delta y$ reacts to input changes Δx and derives output changes Δy. How can we automatically provide Δf given the original function f?

In this work, we assume f is given through its source code s, such that $f(x) = y$ when $x \vdash s \Rightarrow y$. Our goal is to define a differential semantics $\sigma, x, \Delta x \vdash s \Rightarrow_\Delta \Delta y$ that can be used to process changes Δx of the program input. Note that only the input x can change; the source code s is not subject to change in this work. We can thus define $\Delta f(x, \Delta x) = \Delta y$ when $\sigma, x, \Delta x \vdash s \Rightarrow_\Delta \Delta y$, since we will see that the differential semantics is deterministic. The challenge of course is to define the differential semantics such that it is not only correct but also has efficient incremental performance when reacting to input changes. In this section, we motivate our approach through examples.

2.1 Differential execution for epidemiological models

In this paper, we explore differential execution for a statement-based imperative programming language called Imp. Imp features numeric and Boolean values that can be used in assignments, conditionals, and repeat loops, which iterate a loop body a fixed number of times

$$\begin{array}{lrcl} \text{(Statements)} & s & ::= & x := e \mid s; s \mid \textbf{if } e\ s\ s \mid \textbf{repeat } e\ s \\ \text{(Expressions)} & e & ::= & v \mid x \mid e + e \mid e - e \mid e * e \mid e^e \mid e > e \\ \text{(Stores)} & \sigma & & \end{array}$$

We use a big-step reduction semantics $\sigma \vdash s \Rightarrow \sigma'$ that takes a program s and an initial store σ as input, and yields an output store σ'. We can use this language to compute epidemiological models.

Epidemiological models are essential tools for understanding the spread of infectious diseases and informing public health interventions [15]. For example, consider a simplified model that simulates disease spread over a fixed number of n days:

```
# free variables n, I, and β must be bound in the initial store
α = 0.04
S = 0
repeat n:            # n is the number of days for which we model the disease spread
    S = S + I        # S is the cumulative infection count, which grows by I each day
    β = β + α        # β is the disease transmissibility, which grows by α each day
    I = I * 2.718^β  # I is the infected population, which grows exponentially based on β
```

We can now predict how a disease spreads in $n = 6$ days given an initial infectious population $I = 10$ and assuming a transmissibility of $\beta = 0.3$ Figure 1a shows the content of the store σ_i after iteration i of the loop (we omit `n` and α because they are constant). This constitutes the initial, non-incremental run of the program. Note that σ_0 shows the initial state of the model and $\sigma_{out} = \sigma_6$ is the final prediction.

When epidemiologists revise the initial parameters of the model, we want to update our prediction efficiently. In the following, we discuss three change scenarios. First, consider we want to change the initial infected population from 10 to 15. The differential store $\Delta\sigma_0$ in Figure 1b models this change. Our goal is to determine how the final prediction changes, that is, what is $\Delta\sigma_{out}$. To this end, we propose the use of differential execution:

$\sigma_0 = \{S \mapsto 0,\ I \mapsto 10,\ \beta \mapsto 0.3\}$
$\sigma_1 = \{S \mapsto 10,\ I \mapsto 14,\ \beta \mapsto 0.34\}$
$\sigma_2 = \{S \mapsto 24,\ I \mapsto 20,\ \beta \mapsto 0.38\}$
$\sigma_3 = \{S \mapsto 44,\ I \mapsto 30,\ \beta \mapsto 0.42\}$
$\sigma_4 = \{S \mapsto 74,\ I \mapsto 47,\ \beta \mapsto 0.46\}$
$\sigma_5 = \{S \mapsto 121,\ I \mapsto 77,\ \beta \mapsto 0.50\}$
$\sigma_{out} = \{S \mapsto 198, I \mapsto 132, \beta \mapsto 0.54\}$

(a) Stores σ_i after i iterations.

$\Delta\sigma_0 = \{I \mapsto \textbf{inc}\ 5\}$
$\Delta\sigma_1 = \{S \mapsto \textbf{inc}\ 5,\ I \mapsto \textbf{inc}\ 7\}$
$\Delta\sigma_2 = \{S \mapsto \textbf{inc}\ 12,\ I \mapsto \textbf{inc}\ 10\}$
$\Delta\sigma_3 = \{S \mapsto \textbf{inc}\ 22,\ I \mapsto \textbf{inc}\ 15\}$
$\Delta\sigma_4 = \{S \mapsto \textbf{inc}\ 37,\ I \mapsto \textbf{inc}\ 24\}$
$\Delta\sigma_5 = \{S \mapsto \textbf{inc}\ 61,\ I \mapsto \textbf{inc}\ 40\}$
$\Delta\sigma_{out} = \{S \mapsto \textbf{inc}\ 101,\ I \mapsto \textbf{inc}\ 68\}$

(b) Differential stores for $I \mapsto \textbf{inc}\ 5$.

$\Delta\sigma_{0\ldots6} = \{n \mapsto \textbf{inc}\ 3\}$
$\sigma_7 = \{S \mapsto 499,\ I \mapsto 357,\ \beta \mapsto 0.58\}$
$\sigma_8 = \{S \mapsto 856,\ I \mapsto 663,\ \beta \mapsto 0.62\}$
$\sigma_{out} = \{S \mapsto 1519,\ I \mapsto 1282,\ \beta \mapsto 0.66\}$
$\Delta\sigma_{out} = \{S \mapsto \textbf{inc}\ 1321,\ I \mapsto \textbf{inc}\ 1150,$
$\beta \mapsto \textbf{inc}\ 0.12,\ n \mapsto \textbf{inc}\ 3\}$

(c) Differential stores for $n \mapsto \textbf{inc}\ 3$.

$\Delta\sigma_{0\ldots4} = \{n \mapsto \textbf{dec}\ 5\}$
$\sigma_{out} = \{S \mapsto 111,\ I \mapsto 71,\ \beta \mapsto 0.46\}$
$\Delta\sigma_{out} = \{S \mapsto \textbf{dec}\ 1408,\ I \mapsto \textbf{dec}\ 1211,$
$\beta \mapsto \textbf{dec}\ 0.2,\ n \mapsto \textbf{dec}\ 5\}$

(d) Differential stores for $n \mapsto \textbf{dec}\ 5$.

Figure 1 Original and differential stores during execution of the epidemiological model.

Essentially, we rerun the program but only consider and compute changes. Starting from $\Delta\sigma_0$, differential execution of the loop body computes $\Delta\sigma_1$, showing how S and I are affected by the initial change of I. Differential execution continues to iterate the loop body until we reach $\Delta\sigma_6 = \Delta\sigma_{out}$. We observe that 5 additional infections on day 0 lead to 68 additional infections on day 6.

Providing the correct differential result is easy: Reexecute the program and compute the outputs' difference. The challenge for differential execution is to compute output changes efficiently by avoiding recomputations. For example, an increase of I in $\Delta\sigma_0$ does not affect β, which is why β does not occur in Figure 1b. Indeed, as we prove in Section 5, statements that do not read changed variables can be skipped during differential execution. It is shortcuts like this that we need to discover to make differential execution efficient.

For example, consider epidemiologists now want to know what happens after 9 instead of 6 days. We encode this change in $\Delta\sigma_0$ of Figure 1c. Note that we assume the prior change of I remains in place; changes occur in sequence and have to be undone explicitly if so desired. How can differential execution handle the change of n efficiently? In general, it is necessary to repeat each loop iteration to ensure input changes are propagated correctly. However, in our case, n only determines the number of iterations, but it is not used inside the loop body. We prove in Section 5 that in this case we can skip all previous iterations and directly continue with the new ones. Since iterations 7, 8, and 9 have not been executed before, we have to compute the resulting stores from scratch. The final result $\Delta\sigma_{out}$ is then defined by $\sigma_9 \ominus \sigma_6$. In general, when the original loop count n is increased by k, we only need to perform k instead of $n + k$ iterations, which yields significant speedups when $k \ll n$.

Lastly, consider epidemiologists want to rollback n from 9 to 4. This is a case where incremental computing has to trade-off memory and running time. If we still have σ_4 cached, we can compute $\Delta\sigma_{out}$ by $\sigma_4 \ominus \sigma_9$. Otherwise, we may have to do more recomputation. In our study of a formal semantics for differential execution, we ignore all concerns regarding caching. Later, in Section 6, we discuss how we implemented the formal semantics in an efficient interpreter and what caching strategies we considered.

2.2 Branch switching and worst-case complexity

A particular challenge for all kinds of incremental computing is what we call *branch switching*. A branch switch occurs when a change to the input data alters the control flow, causing a different branch of a conditional to be executed compared to the original run. For a simple example, consider this extreme case:

```
# main function with free variable x, which must be bound in the initial store
if x < 0:
    foo()
else:
    bar()
```

When `x` changes from negative to positive, the entire behavior of the program shifts: Instead of running `foo`, we end up running `bar`. This has two important consequences. First, the worst-case complexity of differential execution is at best equal to regular execution. Second, we must retain (or be able to reconstruct) the original state σ that is valid at the beginning of a branch. This state is necessary when a branch switch occurs, as input for executing the other branch. This indicates a lower bound on the memory necessary for differential execution if we want to avoid reconstructing previously seen states.

Does this mean that all hope is lost? No, not at all. Many conditionals do not amount to a reconfiguration of the program, but have a rather limited scope. For example, an improved epidemiological model may consider emergency measures, which are active when infections exceed a threshold T of 40 patients:

```
α = 0.04; γ = 0.02; S = 0; T = 40
repeat n:
    if I > T:           # do infectious cases exceed threshold T?
        β = β - γ       # emergency measures activated, transmissibility shrinks
    else:
        β = β + α       # no emergency measures, transmissibility grows
    S = S + I; I = I * 2.718^β
```

When we change the input variable I, this can now affect the control flow and lead to branch switching. However, the cost of re-executing one of the branches is limited in this case, and only results in different subsequent changes to β. One interesting issue that persists is that branch switching requires diffing between the original and the new state to determine the subsequent changes. In Section 5, we present an optimization that avoids this cost when branches execute similar code.

3 Values and Changes

While regular program execution operates on values, differential execution operates on changes. That is, changes are first-class objects in differential execution. Since the definition of changes is language-specific, we first introduce general rules for how to design well-behaved changes for a statically typed base language. We then define specific changes for our imperative language Imp.

3.1 Change validity, patching, and diffing

Changes must support two fundamental operations: *diffing* to produce changes from values ($new \ominus old$), and *patching* to apply changes to values ($base \oplus change$). Patching and diffing should act as inverses: $v_1 \oplus (v_2 \ominus v_1) = v_2$ and $(v \oplus \Delta v) \ominus v = \Delta v$. These laws are elemental for differential execution as they justify optimizations that avoid redundant work.

However, handling changes correctly is subtle and it is easy to break the second property. For example, consider $v = 2$ and $\Delta v = \mathbf{dec}\ 5$ for the naturals with $2 \oplus \mathbf{dec}\ 5 = 0$. Then,

$$(v \oplus \Delta v) \ominus v = 0 \ominus 2 = \mathbf{dec}\ 2 \neq \mathbf{dec}\ 5 = \Delta v.$$

The problem is that we tried to patch $v = 2$ with $\Delta v = \mathbf{dec}\ 5$, which is invalid because $2 - 5$ does not yield a natural number, and defaulting to 0 contradicts our inversion properties. The same problem occurs for other data types, such as strings, sets, and lists. For example, we get ("`ab`" $\oplus$ **drop** 5) $\ominus$ "`ab`" = "" $\ominus$ "`ab`" = **drop** 2 $\neq$ **drop** 5. Again, we tried to apply a patch that has an undefined behavior, and defaulting to the empty string breaks our inversion laws. Therefore, we follow the incremental lambda calculus [9] and restrict patching to valid changes. But rather than using Coq's dependent typing, we rely on a dedicated typing judgment $v \vdash \Delta v$. Patching and diffing then must satisfy the following properties:

$$\frac{v : \tau \qquad v \vdash \Delta v}{v \oplus \Delta v : \tau}\ \text{T-Patch} \qquad\qquad \frac{v_1 : \tau \qquad v_2 : \tau}{v_1 \vdash v_2 \ominus v_1}\ \text{V-Diff}$$

Patching is only defined when $v \vdash \Delta v$ and yields a value of the same type as v. Diffing is defined for any v_1 and v_2 of the same type, and it yields a valid change that can be applied to v_1. When defining differential execution, it is our responsibility to define changes Δv, validity $v \vdash \Delta v$, patching $\oplus$, and diffing $\ominus$ such that these typing rules hold.

On top of these typing rules for patching and diffing, we can now stipulate the inversion properties we require.

▶ **Property 3.1** (Patch-Diff Inversion)**.** For any well-typed values $v_1 : \tau$ and $v_2 : \tau$, patching inverts diffing: $v_1 \oplus (v_2 \ominus v_1) = v_2$.

▶ **Property 3.2** (Diff-Patch Inversion)**.** For any value v and valid change $v \vdash \Delta v$, diffing inverts patching: $(v \oplus \Delta v) \ominus v \equiv \Delta v$.

Note how the latter property uses equivalence $\equiv$ on changes rather than equality. This is due to the fact that changes often have non-unique representations and diffing might choose any equivalent representation. For example, **inc** 0 and **dec** 0 are equivalent but not equal. In general, $\Delta v_1 \equiv \Delta v_2$ iff for all v, $v \vdash \Delta v_1 \leftrightarrow v \vdash \Delta v_2$ and $v \oplus \Delta v_1 = v \oplus \Delta v_2$.

The above properties justify our use of **noc**, representing no change. Specifically, for any v, we have $v \oplus (v \ominus v) = v$ and therefore $\mathbf{noc} \equiv v \ominus v$. In practice, it is usually easy to decide if a given Δv is equivalent to **noc**.

3.2 Numeric and Boolean changes

Our imperative language Imp uses unsigned integers and Boolean values. We define changes for each type:

$$\begin{array}{ll} \text{(value changes)} & \Delta v ::= \Delta n \mid \Delta b \\ \text{(numeric changes)} & \Delta n ::= \mathbf{noc}_{\mathbb{N}} \mid \mathbf{inc}\ k \mid \mathbf{dec}\ k \\ \text{(Boolean changes)} & \Delta b ::= \mathbf{noc}_{\mathbb{B}} \mid \mathbf{neg} \end{array}$$

Numeric changes consist of no change ($\mathbf{noc}_{\mathbb{N}}$), increasing (**inc** k), and decreasing (**dec** k), where k is any natural number. Boolean changes consist of no change ($\mathbf{noc}_{\mathbb{B}}$) and negation (**neg**). Changes are only valid for an appropriately typed value, and **dec** k is only valid for numbers larger or equal to k:

$$\frac{v : \mathsf{Bool}}{v \vdash \mathbf{noc}_{\mathbb{B}}} \quad \frac{v : \mathsf{Bool}}{v \vdash \mathbf{neg}} \quad \frac{v : \mathsf{Num}}{v \vdash \mathbf{noc}_{\mathbb{N}}} \quad \frac{v : \mathsf{Num}}{v \vdash \mathbf{inc}\ k} \quad \frac{v : \mathsf{Num} \qquad v \geq k}{v \vdash \mathbf{dec}\ k}$$

Whenever $v \vdash \Delta v$, we can patch v while retaining its type as required by T-Patch. Moreover, differencing ensures $v_1 \vdash v_2 \ominus v_1$ as required by V-Diff.

$$
\begin{array}{ll}
n \oplus \mathbf{noc}_{\mathbb{N}} = n & n \ominus n = \mathbf{noc}_{\mathbb{N}} \\
n \oplus \mathbf{inc}\ k = n + k & n \ominus m = \mathbf{inc}\ (n - m), \text{ if } n > m \\
n \oplus \mathbf{dec}\ k = n - k & n \ominus m = \mathbf{dec}\ (m - n), \text{ if } m > n \\
b \oplus \mathbf{noc}_{\mathbb{B}} = b & b \ominus b = \mathbf{noc}_{\mathbb{B}} \\
b \oplus \mathbf{neg} = \neg b & b \ominus c = \mathbf{neg}, \text{ if } b \neq c
\end{array}
$$

We have modeled these changes and their operations in Rocq, where we proved they satisfy the respective typing rules as well as patch-diff and diff-patch inversion.

3.3 Lifting to stores

We have established how patching and diffing operate on individual values, but our imperative programs range over variables that map to values. Therefore, we need to lift changes and their operations from single values to entire stores $\sigma : x \to v$ that map variables to values. To this end, we define differential stores $\Delta\sigma : x \to \Delta v$, which map variables to value changes. Differential stores must satisfy the same properties as differential values: well-typed patching and diffing according to T-Patch and V-Diff, such that patch-diff and diff-patch inversion hold.

We start off by requiring stores to be well-typed given a store type $\Gamma : x \to \tau$. We define store typing $\Gamma \vdash \sigma$ and differential store validity $\sigma \vdash \Delta\sigma$ pointwise:

$$\frac{\forall x.\ \sigma(x) : \Gamma(x)}{\Gamma \vdash \sigma}\ \text{T-Store} \qquad \frac{\forall x.\ \sigma(x) \vdash \Delta\sigma(x)}{\sigma \vdash \Delta\sigma}\ \text{V-Store}$$

Then the following definitions of patching and diffing satisfy T-Patch and V-Diff:

$$
\begin{aligned}
\sigma \oplus \Delta\sigma &= \lambda x.\ \sigma(x) \oplus \Delta\sigma(x) \\
\sigma_1 \ominus \sigma_2 &= \lambda x.\ \sigma_1(x) \ominus \sigma_2(x)
\end{aligned}
$$

Patch-diff inversion holds because

$$\sigma_1 \oplus (\sigma_2 \ominus \sigma_1) \;=\; \lambda x.\ \sigma_1(x) \oplus (\sigma_2(x) \ominus \sigma_1(x)) \;=\; \lambda x.\ \sigma_2(x) \;=\; \sigma_2$$

and conversely diff-patch inversion is due to

$$(\sigma \oplus \Delta\sigma) \ominus \sigma \;=\; \lambda x.\ (\sigma(x) \oplus \Delta\sigma(x)) \ominus \sigma(x) \;\equiv\; \lambda x.\ \Delta\sigma(x) \;=\; \Delta\sigma(x).$$

Again, we have modeled differential stores in Rocq and proved the above properties formally there. This shows how to lift changes from values to stores, which provides the foundation for our differential semantics. The Rocq formalization is provided at

4 Differential Semantics: A Foundation for Incremental Computing

Incremental computing is all about eliminating redundant work in the face of input changes. In the pursuit of efficiency, incremental computing tries to cut as many corners as possible while retaining correctness. Our differential style of incremental computing is particularly susceptible to correctness errors, because we need to capture the precise difference of each computation output. In this section, we introduce the idea of differential semantics to capture the incremental behavior of a program. Specifically, we develop a differential semantics for the imperative language Imp from Section 2 with expressions and statements.

$$\frac{}{\sigma \vdash \textbf{true} \Rightarrow \textbf{true}} \qquad \frac{}{\sigma \vdash \textbf{false} \Rightarrow \textbf{false}} \qquad \frac{}{\sigma \vdash n \Rightarrow n}$$

$$\frac{}{\sigma \vdash x \Rightarrow \sigma(x)} \qquad \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_1 \odot e_2 \Rightarrow v_1 \odot v_2}$$

$$\frac{\sigma \vdash e \Rightarrow v}{\sigma \vdash x := e \Rightarrow [x \mapsto v]\sigma} \qquad \frac{\sigma \vdash s_1 \Rightarrow \sigma' \qquad \sigma' \vdash s_2 \Rightarrow \sigma''}{\sigma \vdash s_1; s_2 \Rightarrow \sigma''}$$

$$\frac{\sigma \vdash e \Rightarrow \textbf{true} \qquad \sigma \vdash s_1 \Rightarrow \sigma'}{\sigma \vdash \textbf{if}\ e\ s_1\ s_2 \Rightarrow \sigma'} \qquad \frac{\sigma \vdash e \Rightarrow \textbf{false} \qquad \sigma \vdash s_2 \Rightarrow \sigma'}{\sigma \vdash \textbf{if}\ e\ s_1\ s_2 \Rightarrow \sigma'}$$

$$\frac{\sigma \vdash e \Rightarrow 0}{\sigma \vdash \textbf{repeat}\ e\ s \Rightarrow \sigma} \qquad \frac{\sigma \vdash e \Rightarrow n+1 \qquad \sigma \vdash \textbf{repeat}\ n\ s \Rightarrow \sigma' \qquad \sigma' \vdash s \Rightarrow \sigma''}{\sigma \vdash \textbf{repeat}\ e\ s \Rightarrow \sigma''}$$

Figure 2 Standard big-step reduction semantics for Imp expressions and statements.

A differential semantics establishes a sound baseline for incremental computing, but it is not usually optimal. Instead, we separately introduce optimizations for the efficient incremental execution of programs and prove them correct with respect to the baseline differential semantics. To support the modular definition and verification of optimizations, we formulate the differential semantics compositionally for each language construct. But first, let us review the standard big-step semantics of our imperative language Imp.

4.1 Standard big-step semantics of Imp

We already introduced the syntax of Imp expressions, statements, and stores in Section 2. Figure 2 shows the standard semantics for Imp using two judgments. For expressions, we write $\sigma \vdash e \Rightarrow v$ to denote that expression e evaluates to value v under store σ. Expressions can read but cannot change the content of the store. The last reduction rule for expressions handles binary expressions $e_1 \odot e_2$. Imp supports a few binary operators $\odot \in \{+, -, *, >, \mathsf{exp}\}$, and we assume that $v_1 \odot v_2$ follows the standard arithmetic interpretation of these operators.

For statements, we write $\sigma \vdash s \Rightarrow \sigma'$ to denote that statement s executes under store σ, yielding a possibly updated store σ'. In these rules, the notation $[x \mapsto v]\sigma$ represents the store obtained from σ by updating the mapping of variable x to value v, while leaving all other mappings unchanged. For repeat loops, we skip the body when the iteration count is zero. Otherwise, we recursively evaluate the loop with a decremented count, followed by running the body. We opted for this left-recursive loop unrolling to simplify some of the proofs.

4.2 Differential semantics for Imp expressions

A differential semantics describes how a program's output changes in reaction to input changes. The inputs of our Imp programs are stores σ. We have described store changes $\Delta\sigma$ in Subsection 3.3. The output of an Imp program is also a store, but we first only consider Imp expressions. An Imp expression computes a value v, hence the differential semantics for Imp expressions computes a value change Δv. To this end, we define a differential reduction relation $\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \Delta v$ to compute how e's output changes:

$$\frac{e \in \{\textbf{true}, \textbf{false}\}}{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \textbf{noc}_\mathbb{B}}\ \text{D-Bool} \qquad \frac{}{\sigma, \Delta\sigma \vdash n \Rightarrow_\Delta \textbf{noc}_\mathbb{N}}\ \text{D-Num} \qquad \frac{}{\sigma, \Delta\sigma \vdash x \Rightarrow_\Delta \Delta\sigma(x)}\ \text{D-Var}$$

$$\frac{\sigma, \Delta\sigma \vdash e_1 \Rightarrow_\Delta \Delta v_1 \qquad \sigma, \Delta\sigma \vdash e_2 \Rightarrow_\Delta \Delta v_2 \qquad (\ \sigma \vdash e_1 \Rightarrow v_1\ ) \qquad (\ \sigma \vdash e_2 \Rightarrow v_2\ )}{\sigma, \Delta\sigma \vdash e_1 \odot e_2 \Rightarrow_\Delta \odot_\Delta(v_1, v_2, \Delta v_1, \Delta v_2)}\ \text{D-BinOp}$$

The first rules handle an expression e that is a Boolean or numeric constant. The value of constant expressions can not change, hence the output is a no change (**noc**). A variable x changes in accordance with $\Delta\sigma$. In particular, when an input is changed, reading from that input yields its change, which can then be propagated.

The last rule handles binary operators $\odot$ and delegates their differential behavior to dedicated functions $\odot_\Delta$. We allow these functions to use the original input values (v_1, v_2) and their changes (Δv_1, Δv_2). For example, for a differential multiplication $m * (n \oplus \textbf{inc}\ k)$, we need to know the original arguments to compute the output change **inc** $(m * k)$. While our implementation supports various operators, in this paper and the Rocq formalization we focus on establishing the core differential semantics. A complete treatment of differential operators is outside the scope of this paper.

Note that we required the original store σ during the differential execution of expressions only to recover the original arguments of binary operators. In places like these, we can expect an actual implementation to cache the previous evaluation result of each operand where needed. We highlight opportunities for caching in the reduction rules by putting preconditions in parentheses (*cached*) and slightly shading their background.

4.3 A differential semantics for Imp statements

We define the differential semantics for statements using the relation

$$\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma'$$

which executes statement s under differential store $\Delta\sigma$ and original store σ to yield an updated differential store $\Delta\sigma'$. The inference rules are shown in Figure 3.

Rule D-Assign evaluates the assigned expressions differentially and then updates the differential store. The store update is a destructive update, overwriting the previous change assigned to x. Together with rule D-Seq, this realizes change propagation behavior. For example, assume a program $x := a; y := x$ with $\Delta\sigma(a) = \Delta v$. Rule D-Seq runs the assignments in order. The first assignment sets $\Delta\sigma(x) = \Delta v$ because $\sigma, \Delta\sigma \vdash a \Rightarrow_\Delta \Delta v$. Then the second assignment sets $\Delta\sigma(y) = \Delta v$ because $\sigma', \Delta\sigma' \vdash x \Rightarrow_\Delta \Delta v$ Note that D-Seq also runs the first statement non-incrementally, but only to obtain σ', which is needed as input for s_2. The highlighting of this and other preconditions in Figure 3 indicates that the precondition's result can be retrieved from a cache in an actual implementation.

For conditional statements, we generally distinguish two situations: branch constant and branch switching. A conditional behaves branch-constant if the value of the condition does not change, handled by rules D-IF$_{\text{tt}}$ and D-IF$_{\text{ff}}$. These rules simply propagate changes to the relevant branch that was selected originally. Rules D-IF$_{\text{tf}}$ and D-IF$_{\text{ft}}$ handle branch switching, which happens when the condition's value changes. If the original value of the condition was **true** and now becomes **false**, then rule D-IF$_{\text{tf}}$ fires and performs branch switching as introduced in Subsection 2.2. Originally, we have executed s_1, but now we need to run s_2. Therefore, we execute s_2 from scratch and compute how its output differs from s_1. Rule D-IF$_{\text{ft}}$ is analogous and handles the case when the condition switches from **false** to **true**.

$$\frac{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \Delta v}{\sigma, \Delta\sigma \vdash x := e \Rightarrow_\Delta [x \mapsto \Delta v]\Delta\sigma}\ \text{D-Assign}$$

$$\frac{\sigma, \Delta\sigma \vdash s_1 \Rightarrow_\Delta \Delta\sigma' \quad (\,\sigma \vdash s_1 \Rightarrow \sigma'\,) \quad \sigma', \Delta\sigma' \vdash s_2 \Rightarrow_\Delta \Delta\sigma''}{\sigma, \Delta\sigma \vdash s_1; s_2 \Rightarrow_\Delta \Delta\sigma''}\ \text{D-Seq}$$

$$\frac{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \mathbf{noc}_\mathbb{B} \quad (\,\sigma \vdash e \Rightarrow \mathbf{true}\,) \quad \sigma, \Delta\sigma \vdash s_1 \Rightarrow_\Delta \Delta\sigma'}{\sigma, \Delta\sigma \vdash \mathbf{if}\ e\ s_1\ s_2 \Rightarrow_\Delta \Delta\sigma'}\ \text{D-IF}_{\text{tt}} \qquad \frac{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \mathbf{noc}_\mathbb{B} \quad (\,\sigma \vdash e \Rightarrow \mathbf{false}\,) \quad \sigma, \Delta\sigma \vdash s_2 \Rightarrow_\Delta \Delta\sigma'}{\sigma, \Delta\sigma \vdash \mathbf{if}\ e\ s_1\ s_2 \Rightarrow_\Delta \Delta\sigma'}\ \text{D-IF}_{\text{ff}}$$

$$\frac{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \mathbf{neg} \quad (\,\sigma \vdash e \Rightarrow \mathbf{true}\,) \quad \sigma \oplus \Delta\sigma \vdash s_2 \Rightarrow \sigma_2 \quad (\,\sigma \vdash s_1 \Rightarrow \sigma_1\,)}{\sigma, \Delta\sigma \vdash \mathbf{if}\ e\ s_1\ s_2 \Rightarrow_\Delta \sigma_2 \ominus \sigma_1}\ \text{D-IF}_{\text{tf}} \qquad \frac{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \mathbf{neg} \quad (\,\sigma \vdash e \Rightarrow \mathbf{false}\,) \quad \sigma \oplus \Delta\sigma \vdash s_1 \Rightarrow \sigma_1 \quad (\,\sigma \vdash s_2 \Rightarrow \sigma_2\,)}{\sigma, \Delta\sigma \vdash \mathbf{if}\ e\ s_1\ s_2 \Rightarrow_\Delta \sigma_1 \ominus \sigma_2}\ \text{D-IF}_{\text{ft}}$$

$$\frac{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \mathbf{noc}_\mathbb{N} \quad (\,\sigma \vdash e \Rightarrow 0\,)}{\sigma, \Delta\sigma \vdash \mathbf{repeat}\ e\ s \Rightarrow_\Delta \Delta\sigma}\ \text{D-Rep-0}$$

$$\frac{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \mathbf{noc}_\mathbb{N} \quad (\,\sigma \vdash e \Rightarrow n+1\,) \quad \sigma, \Delta\sigma \vdash \mathbf{repeat}\ n\ s \Rightarrow_\Delta \Delta\sigma' \quad (\,\sigma \vdash \mathbf{repeat}\ n\ s \Rightarrow \sigma'\,) \quad \sigma', \Delta\sigma' \vdash s \Rightarrow_\Delta \Delta\sigma''}{\sigma, \Delta\sigma \vdash \mathbf{repeat}\ e\ s \Rightarrow_\Delta \Delta\sigma''}\ \text{D-Rep+1}$$

$$\frac{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \mathbf{dec}\ k \quad (\,\sigma \vdash e \Rightarrow n\,) \quad \sigma, \Delta\sigma \vdash \mathbf{repeat}\ (n-k)\ s \Rightarrow_\Delta \Delta\sigma' \quad (\,\sigma \vdash \mathbf{repeat}\ (n-k)\ s \Rightarrow \sigma'\,) \quad (\,\sigma \vdash \mathbf{repeat}\ n\ s \Rightarrow \sigma''\,)}{\sigma, \Delta\sigma \vdash \mathbf{repeat}\ e\ s \Rightarrow_\Delta (\sigma' \oplus \Delta\sigma') \ominus \sigma''}\ \text{D-Rep-Dec}$$

$$\frac{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \mathbf{inc}\ k \quad (\,\sigma \vdash e \Rightarrow n\,) \quad \sigma, \Delta\sigma \vdash \mathbf{repeat}\ n\ s \Rightarrow_\Delta \Delta\sigma' \quad (\,\sigma \vdash \mathbf{repeat}\ n\ s \Rightarrow \sigma'\,) \quad \sigma' \oplus \Delta\sigma' \vdash \mathbf{repeat}\ k\ s \Rightarrow \sigma''}{\sigma, \Delta\sigma \vdash \mathbf{repeat}\ e\ s \Rightarrow_\Delta \sigma'' \ominus \sigma'}\ \text{D-Rep-Inc}$$

Figure 3 Differential semantics for Imp statements. Marked (preconditions) can be cached.

Finally, we provide four rules to handle repeat loops. The differential semantics for **repeat** e s depends crucially on how the iteration count changes. When the differential evaluation of e yields $\mathbf{noc}_\mathbb{N}$, the iteration count remains the same as in the original execution. In this case, we replay the loop but only do change propagation. If the iteration count is zero ($\sigma \vdash e \Rightarrow 0$), we simply return the input differential store unchanged, as no iterations need to be replayed. For a positive count ($\sigma \vdash e \Rightarrow n+1$), we unroll the loop recursively followed by a differential run of the loop body s. The resulting $\Delta\sigma''$ captures the cumulative effect of differentially executing all iterations.

When the iteration count is decreased ($\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \mathbf{dec}\ k$), the process is more complicated. First, we propagate the changes for $(n-k)$ iterations to obtain $\Delta\sigma'$. Then we retrieve the original output after $(n-k)$ iterations σ' and patch it with $\Delta\sigma'$. This represents the updated output after $(n-k)$ iterations, but we need to know how it changed compared to n iterations. To this end, we compute the difference with σ'', which is the original output after n iterations.

Lastly, when the iteration count grows ($\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \textbf{inc}\ k$), we use a two-phase strategy. First, we execute the original n iterations differentially to obtain $\Delta\sigma'$. However, from here on out, we are dealing with iterations that have not occurred before. Therefore, in the second phase, we execute the additional k iterations non-incrementally. To this end, we compute the updated output after n iterations using $\Delta\sigma'$ and feed it as input for the remaining k iterations. Finally, we compute the difference with σ', which was the original output.

▶ **Example.** Consider the following program and stores: repeat x $\{y = y + z;\ z = z + 1\}$ with initial store $\sigma = \{x \mapsto 5,\ y \mapsto 7,\ z \mapsto 3\}$. The original output is $\sigma' = \{x \mapsto 5,\ y \mapsto 32,\ z \mapsto 8\}$. Now consider we apply the following input changes $\Delta\sigma = \{x \mapsto \textbf{inc}\ 2,\ y \mapsto \textbf{inc}\ 3,\ z \mapsto \textbf{inc}\ 5\}$. The example demonstrates how differential execution handles both loop counter changes (x) and variable changes (y, z). Specifically, here rule D-Rep-Inc is leading the execution and proceeds in the following steps:

1. Execute original 5 iterations differentially: $\Delta\sigma' = \{x \mapsto \textbf{inc}\ 2,\ y \mapsto \textbf{inc}\ 28,\ z \mapsto \textbf{inc}\ 5\}$.
2. Compute patched state after 5 iterations: $\sigma' \oplus \Delta\sigma' = \{x \mapsto 7,\ y \mapsto 60,\ z \mapsto 13\}$.
3. Execute new iterations (k = 2) from $\sigma' \oplus \Delta\sigma'$ to obtain $\sigma'' = \{x \mapsto 7,\ y \mapsto 87,\ z \mapsto 15\}$.
4. Compute differential output: $\sigma'' \ominus \sigma' = \{x \mapsto \textbf{inc}\ 2,\ y \mapsto \textbf{inc}\ 55,\ z \mapsto \textbf{inc}\ 7\}$.

4.4 Properties of differential semantics

Our differential semantics aims to replace recomputation with change-based execution. That is, when the original input σ is changed to $\sigma' = \sigma \oplus \Delta\sigma$, the differential semantics must be able to process $\Delta\sigma$ directly. For this to work, we need three guarantees:

Completeness: Whenever a program can execute under the original σ and the updated σ', then the differential semantics must derive an output change for $\Delta\sigma$.

Output validity: The output change must be valid for the original output.

From-scratch consistency Patching the original output with the output changes is consistent with a recomputation starting with σ'.

That is: We can obtain changes using the differential semantics, the changes can be used to patch the original output, and the patched output is correct. We have formalized these properties in Rocq and have a mechanized proof that shows our differential semantics satisfies them. Here, we only present the formal formulation of the properties.

In general, we only make claims about well-typed programs $\Gamma \vdash s$. The typing relation for statements and expressions is standard: variables are global and cannot change type. With this, we can formulate the properties of differential semantics.

▶ **Theorem 4.1** (Completeness). *Given a well-typed program $\Gamma \vdash s$, well-typed store $\Gamma \vdash \sigma$, and valid differential store $\sigma \vdash \Delta\sigma$. Whenever $\sigma \vdash s \Rightarrow \theta$ and $\sigma \oplus \Delta\sigma \vdash s \Rightarrow \theta'$ for arbitrary θ and θ', then $\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma'$ for some $\Delta\sigma'$.*

Proof sketch. By induction on the typing derivation of statement s. For base cases like Assign, we directly construct differential derivations. For compound statements like Seq and If, we apply induction hypotheses to substatements and compose their results. The critical case is Repeat, where we handle changes to the iteration count by carefully tracking execution states for each iteration. ◀

Completeness ensures we find output changes $\Delta\sigma'$. Note that for our Imp language, the original and update execution always succeeds for well-typed programs and stores, but richer languages may include run-time errors and divergence For Imp, completeness ensures that we have defined enough reduction rules to handle all well-typed execution states.

Validity guarantees that differential execution produces changes that are compatible with the values they will be patched with. This ensures we can safely apply all changes produced during execution.

▶ **Theorem 4.2** (Output validity). *Given a well-typed program* $\Gamma \vdash s$*, well-typed store* $\Gamma \vdash \sigma$*, and valid differential store* $\sigma \vdash \Delta\sigma$*. Whenever* $\sigma \vdash s \Rightarrow \sigma'$ *and* $\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma'$*, then* $\sigma' \vdash \Delta\sigma'$ *is valid for the original output.*

Proof sketch. By induction on the differential semantics derivation, showing each rule preserves validity when input changes are valid. Critical cases include branch switching and loop optimizations, where the type system ensures changes remain compatible with their target values. ◀

From-scratch consistency is our key correctness property – it proves that differential execution produces the same final results as re-running the program with changed inputs. This means users can trust that differential execution is just an optimization that does not affect program behavior.

▶ **Theorem 4.3** (From-Scratch Consistency). *Given a well-typed program* $\Gamma \vdash s$*, well-typed store* $\Gamma \vdash \sigma$*, and valid differential store* $\sigma \vdash \Delta\sigma$*. Whenever* $\sigma \vdash s \Rightarrow \sigma'$ *and* $\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma'$*, then* $\sigma \oplus \Delta\sigma \vdash s \Rightarrow \sigma' \oplus \Delta\sigma'$*.*

Proof sketch. By induction on differential semantics, proving patched original outputs match recomputation results. The key insight relies on our change operations satisfying algebraic properties like patch-diff and diff-patch inversion, ensuring consistency across all language constructs. ◀

We have formally verified all three theorems using the Rocq proof assistant. The complete mechanized proofs are available in the repository referenced in Section 3.

4.5 Mechanized Rocq formalization

We formalized the differential semantics of Imp in Rocq and proved completeness, output validity, and from-scratch consistency. The Rocq code is available open-source.[2] The Rocq formalization consists of approximately 1,450 lines of code and was important in shaping our theory of changes. A key challenge emerged when proving the diff-patch inversion property (Property 3.2) introduced in Section 3. While standard equality was sufficient for proving the patch-diff property (Property 3.1), we discovered it was inadequate for the diff-patch case. We needed a custom equivalence relation because multiple differential values (noc, incr 0, decr 0) can represent the same change semantically

The consistency proof (Theorem 4.3) was particularly challenging, especially for the increment case of repeat statements. The complexity of this case revealed that our initial induction hypothesis wasn't strong enough – an observation we only gained after successfully proving all other cases. This led to a reformulation of the theorem with a stronger induction hypothesis.

In our opinion, for the number of results we proved, our Rocq code is reasonably compact. This was achieved through careful proof automation: we developed custom tactics for common proof patterns and registered them as hints for Rocq's eauto system. This automation strategy significantly reduced the proof size.

[2] `https://gitlab.rlp.net/plmz/artifacts/autoinc-interp-formalization-ecoop25`

5 Optimizations as Alternative Reduction Rules

To enhance the efficiency of our differential semantics, we introduce several optimizations which allow us to avoid unnecessary computations during execution. We present them as alternative reduction rules within our differential semantics. Each optimization we present is a shortcut: it lets us compute the same result more efficiently. To ensure these shortcuts are correct, we must prove they are *admissible*. This means that whenever we use an optimization rule to derive a result, we can also derive the same result (perhaps less efficiently) using only the standard rules from Figure 3. Formally, we define admissibility of a rule as follows.

▶ **Definition 5.1** (Rule Admissibility). *An optimization rule R with premises P and conclusion $\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma'$ is admissible if for any well-typed statement s, whenever P holds, there exists a derivation D using only standard rules that concludes $\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma'$.*

Admissibility ensures optimizations are behavior-preserving shortcuts. In the following sections, we present several optimizations. We have proved the admissibility for each of these rules in Rocq, working only with well-typed programs since well-typedness is essential for ensuring the soundness of our optimizations. We categorize the various optimizations into three main types: *short-circuiting optimizations*, *loop optimizations*, and *branch switching optimizations*, and discuss each in detail.

5.1 Short-Circuiting Rules

Short-circuiting rules enable bypassing differential computations entirely under certain conditions. We begin with a special case where all variables of a store map to no change. Then we generalize this rule to handle programs where only referenced variables are unchanged.

$$\frac{\Gamma \vdash s \qquad \sigma \vdash s \Rightarrow \sigma' \qquad \forall x.\ \Delta\sigma(x) = \mathbf{noc}}{\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma}\ \varepsilon\text{-Store}$$

The ε-Store rule shows that when every variable in the differential store maps to **noc**, differential execution preserves the input differential store unchanged. This optimization is useful because when no variables have changed at all, we can completely bypass differential execution, avoiding differential computations entirely. The rule is fundamentally sound since a program operating on unchanged inputs must produce unchanged outputs. The premise $\sigma \vdash s \Rightarrow \sigma'$ in the rule establishes that s executes successfully in the standard semantics, which is necessary for proving the rule's admissibility.

Consider the following program:

```
x = y + z
w = v * 2
```

With the differential store $\Delta\sigma = \{x \mapsto \mathbf{noc}, y \mapsto \mathbf{noc}, z \mapsto \mathbf{noc}, w \mapsto \mathbf{noc}, v \mapsto \mathbf{noc}\}$, the rule lets us skip differential execution entirely.

To handle more realistic scenarios where a differential store might contain changes to many variables but a specific program fragment only references unchanged variables, we need to precisely characterize when a fragment is unaffected by changes. We do this through the No-Change relation.

▶ **Definition 5.2** (No-Change Relation). *The no-change relation for expressions $\Gamma, \Delta\sigma \vdash^0_e e$ and statements $\Gamma, \Delta\sigma \vdash^0_s s$ indicates that under typing context Γ and differential store $\Delta\sigma$, all variables referenced in e or s map to $\mathbf{noc}_\tau$ of their respective type τ in Γ. This relation is defined inductively by the rules in Figure 4.*

$$\frac{}{\Gamma, \Delta\sigma \vdash^0_e \ \mathbf{true}} \qquad \frac{}{\Gamma, \Delta\sigma \vdash^0_e \ \mathbf{false}} \qquad \frac{}{\Gamma, \Delta\sigma \vdash^0_e \ n} \qquad \frac{\tau = \Gamma\ x \quad \Delta\sigma(x) = \mathbf{noc}_\tau}{\Gamma, \Delta\sigma \vdash^0_e \ x}$$

$$\frac{\tau = \Gamma\ x \quad \Delta\sigma(x) = \mathbf{noc}_\tau \quad \Gamma, \Delta\sigma \vdash^0_e \ e}{\Gamma, \Delta\sigma \vdash^0_s \ x := e} \qquad \frac{\Gamma, \Delta\sigma \vdash^0_s \ s_1 \quad \Gamma, \Delta\sigma \vdash^0_s \ s_2}{\Gamma, \Delta\sigma \vdash^0_s \ s_1; s_2}$$

$$\frac{\Gamma, \Delta\sigma \vdash^0_e \ e \quad \Gamma, \Delta\sigma \vdash^0_s \ s_1 \quad \Gamma, \Delta\sigma \vdash^0_s \ s_2}{\Gamma, \Delta\sigma \vdash^0_s \ \mathbf{if}\ e\ s_1\ s_2} \qquad \frac{\Gamma, \Delta\sigma \vdash^0_e \ e \quad \Gamma, \Delta\sigma \vdash^0_s \ s}{\Gamma, \Delta\sigma \vdash^0_s \ \mathbf{repeat}\ e\ s}$$

Figure 4 Inference rules for expressions and statements experiencing no change under $\Delta\sigma$.

Using this relation, we formulate our second optimization rule as follows:

$$\frac{\Gamma, \Delta\sigma \vdash^0_s \ s \qquad \Gamma \vdash s}{\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma}\ \textsc{No-Change}$$

The No-Change rule generalizes the ε-Store rule: when no variables referenced in s are affected by changes ($\Gamma, \Delta\sigma \vdash^0_s \ s$), any differential execution of s must preserve the input differential store. This optimization is particularly valuable as it allows us to skip computations even when some variables in the program have changed, as long as they aren't referenced in the code being optimized.

Consider our earlier program fragment in a larger context:

```
x = y + z
w = v * 2
# ... later code using 'a' ...
```

With differential store $\Delta\sigma = \{x \mapsto \mathbf{noc}, y \mapsto \mathbf{noc}, z \mapsto \mathbf{noc}, w \mapsto \mathbf{noc}, v \mapsto \mathbf{noc}, a \mapsto \mathbf{inc}\ 5\}$, the No-Change rule lets us skip the first two assignments even though a changes elsewhere in the program. This local reasoning about variable usage avoids wasteful expression evaluation – we don't compute changes that would inevitably be **noc**.

5.2 Loop Optimization Rules

Loop constructs often dominate execution time in incremental computations. We present three optimization rules for loops, each handling a different scenario where full recomputation can be avoided. We start with loop idempotence, then present rules for handling changes to iteration counts.

$$\frac{\begin{array}{ll} & n \geq 0 \\ \Gamma \vdash \sigma & \sigma \vdash e \Rightarrow n \\ \sigma \vdash \Delta\sigma & \sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \mathbf{noc} \qquad \sigma \vdash s \Rightarrow \sigma \qquad \sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma \end{array}}{\sigma, \Delta\sigma \vdash \mathbf{repeat}\ e\ s \Rightarrow_\Delta \Delta\sigma}\ \textsc{Loop-Idemp}$$

The Loop-Idemp rule provides a significant optimization opportunity: when we can prove that a loop body preserves both the store and differential store in a single iteration and there is no change in the number of loop iterations, we can completely skip executing the remaining iterations. This is valuable because without this rule, we would need to execute each iteration even though we know the final result would be unchanged.

Consider a loop with operations that cancel out:

```
repeat n:
    x = (x * 1) + (y - y)    # x remains unchanged through identities
    y = (y + x) - x          # y remains unchanged through inverse operations
```

When $\Delta\sigma = \{x \mapsto \textbf{noc}, y \mapsto \textbf{noc}\}$, we can skip all iterations since each iteration preserves the store through these reversing computations. While this example is deliberately simple, the Loop-Idemp rule becomes particularly powerful with richer language features. For instance, in a language with arrays and objects, this rule would recognize loops that appear to modify data structures but actually preserve their state, allowing us to skip such loops entirely when we can prove that their operations are self-cancelling.

Our two most powerful loop optimizations address scenarios where the loop count changes, but the loop body remains unaffected by changes in the differential store. That is, all variables in the loop are assigned **noc** in the input differential store. This optimization reduces computational effort by focusing only on the additional iterations introduced by the change in the loop count, thereby avoiding redundant execution of the unchanged loop body.

$$\frac{\begin{array}{lll}\Gamma, \Delta\sigma \vdash^0_s\ s & & \sigma \vdash \textbf{repeat}\ e\ s \Rightarrow \sigma' \\ \Gamma \vdash s & \sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \textbf{inc}\ m & \sigma' \vdash \textbf{repeat}\ m\ s \Rightarrow \sigma''\end{array}}{\sigma, \Delta\sigma \vdash \textbf{repeat}\ e\ s \Rightarrow_\Delta (\sigma'' \oplus \Delta\sigma) \ominus \sigma'}\ \textsc{Loop-Incr}$$

The Loop-Incr rule handles increases in iteration count. When the loop count increases by m iterations and the loop body references no changed variables ($\Gamma, \Delta\sigma \vdash^0_s\ s$), we can optimize by doing the following.

- Reusing the result σ' of the original execution
- Computing only the additional m iterations from σ' to get σ''
- Computing the differential results as $(\sigma'' \oplus \Delta\sigma) \ominus \sigma'$

The differential output captures how the loop's output differs after running the additional iterations. With cached results from the original execution, we only compute the m new iterations: $\sigma' \vdash \textbf{repeat}\ m\ s \Rightarrow \sigma''$ – precisely the work that must be done. This optimization avoids redoing unchanged iterations while still performing the genuinely new computation. However, when any variable read in the loop body changes (for example, if an array being summed changes), we cannot apply this optimization. This restriction is necessary – changes to variables in the loop body would produce different results in every iteration, invalidating our assumption that earlier iterations remain unchanged and in process invalidating the cached result σ'.

Consider the following example where we accumulate a sum with a variable count:

```
sum = 0
i = 0
repeat n:  # n increases from 5 to 7
  sum = sum + i
  i = i + 1
```

With $\Delta\sigma = \{n \mapsto \textbf{inc}\ 2, \texttt{sum} \mapsto \textbf{noc}, i \mapsto \textbf{noc}\}$, both `sum` and `i` are unchanged in the input store, allowing us to safely reuse cached results. After the original 5 iterations, we compute just 2 more iterations with values we haven't seen before. If instead we had $\Delta\sigma = \{n \mapsto \textbf{inc}\ 2, i \mapsto \textbf{inc}\ 1\}$, the change to `i` would affect every iteration's computation, forcing us to recompute all iterations with the new initial value.

When all premises remain as in the increment case, but with $\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \textbf{dec}\ m$ replacing the increment, we obtain an elegant formulation for the decrement case as shown below.

$$\frac{\begin{array}{lll} \Gamma, \Delta\sigma \vdash^0_s s & \sigma \vdash e \Rightarrow n & \sigma \vdash \textbf{repeat}\ e\ s \Rightarrow \sigma' \\ \Gamma \vdash s & \sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \textbf{dec}\ m & \sigma \vdash \textbf{repeat}\ (n-m)\ s \Rightarrow \sigma'' \end{array}}{\sigma, \Delta\sigma \vdash \textbf{repeat}\ e\ s \Rightarrow_\Delta (\sigma'' \oplus \Delta\sigma) \ominus \sigma'} \textsc{Loop-Decr}$$

For decreased iteration counts, the Loop-Decr rule is even more efficient than the increment case. If we have cached the original computation, we have access to both the original output σ' and the intermediate store σ'' after the $(n-m)^{th}$ iteration. Then, we can compute the final result directly with just a patch and a diff operation, without any loop execution. This makes the decrement case even more efficient than the increment case, where we still needed to perform m additional iterations. Again, this optimization requires $\Gamma, \Delta\sigma \vdash^0_s s$, as changes to loop body variables would invalidate our cached states.

Using the same summation example but with n decreasing from 5 to 3 and $\Delta\sigma = \{n \mapsto \textbf{dec}\ 2, \texttt{sum} \mapsto \textbf{noc}, i \mapsto \textbf{noc}\}$, we don't need to compute anything new from the normal execution. We can directly use our cached state after 3 iterations to compute the change.

In both the increment and the decrement case, we see that caching the original computation is important for efficiency. We discuss various strategies for caching computations in Section 6.1.

5.3 Branch Switching Rules

One of the main challenges in differential execution is handling branch switches efficiently, as we typically need to execute both branches and compare their stores. Consider a common pattern in imperative programs, where an if-statement contains only assignments to the same variable in both branches:

```
if (y) {
    x = z + 1
} else {
    x = z - 1
}
```

With branch switching, the standard differential semantics would compute the result by diffing the resulting stores corresponding to the two different branches that are taken. However, since only variable x changes between branches, we can update just one entry instead of the entire store. This targeted update of x based on the expressions' difference provides orders of magnitude in speedup over store-wide diffing, especially pronounced for large stores. We develop two optimizations here based on this insight.

$$\frac{\begin{array}{lll} & & \Gamma \vdash \textbf{if}\ e\ (x := e_1)\ (x := e_2) \\ & \sigma \vdash e \Rightarrow \textbf{true} & \sigma \vdash e_1 \Rightarrow v_1 \\ \Gamma \vdash \sigma & \sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \Delta v & \sigma \oplus \Delta\sigma \vdash e_2 \Rightarrow v_2 \\ \sigma \vdash \Delta\sigma & \textbf{true} \oplus \Delta v = \textbf{false} & \sigma, \Delta\sigma \vdash \textbf{if}\ e\ (x := e_1)\ (x := e_2) \Rightarrow_\Delta \Delta\sigma' \end{array}}{\Delta\sigma' \equiv [x \mapsto v_2 \ominus v_1]\Delta\sigma} \textsc{If-T-F}$$

The If-T-F rule handles the case where a condition switches from true to false. The rule requires both stores to be well-typed and the conditional statement itself to be well-typed. When the conditional expression e switches from true to false and both branches are assignments to the same variable x, we can directly compute the change to x as $v_2 \ominus v_1$, where v_1 is e_1's value in the original store and v_2 is e_2's value in the patched store.

For our example, suppose y was initially **true** and we have $\Delta\sigma = \{y \mapsto \textbf{neg}, x \mapsto \textbf{noc}, z \mapsto \textbf{noc}\}$ causing a branch switch. Instead of recomputing the entire store, we only need to update the entry corresponding to x as $(z-1) \ominus (z+1) = \textbf{dec}\ 2$. However, if branches modified different variables (e.g., `x = z + 1` vs `y = z - 1`), this optimization cannot apply – the changes aren't localized to a single variable.

$$\frac{\begin{array}{lll} & & \Gamma \vdash \textbf{if}\ e\ (x := e_1)\ (x := e_2) \\ & \sigma \vdash e \Rightarrow \textbf{false} & \sigma \vdash e_2 \Rightarrow v_2 \\ \Gamma \vdash \sigma & \sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \Delta v & \sigma \oplus \Delta\sigma \vdash e_1 \Rightarrow v_1 \\ \sigma \vdash \Delta\sigma & \textbf{false} \oplus \Delta v = \textbf{true} & \sigma, \Delta\sigma \vdash \textbf{if}\ e\ (x := e_1)\ (x := e_2) \Rightarrow_\Delta \Delta\sigma' \end{array}}{\Delta\sigma' \equiv [x \mapsto v_1 \ominus v_2]\Delta\sigma}\ \textsc{If-F-T}$$

The If-F-T rule handles the symmetric case where the condition switches from false to true. Its structure mirrors the true-to-false case, but computes $v_1 \ominus v_2$ since we are switching to the first branch.

This optimization significantly improves performance when stores are large, but changes are localized. In our Bellman-Ford implementation (Figure 5), this optimization avoids diffing the entire shortest-path table on each branch switch within the inner loop's conditional, leading to substantial speedups as shown in Section 7.

6 Implementing a Differential Interpreter

In this section, we describe the implementation of a differential interpreter based on the differential semantics. The implementation is written in Scala 3, is open source and available at `https://gitlab.rlp.net/plmz/artifacts/autoinc-interp-implementation-ecoop25`.

6.1 Caching to avoid Re-Computation

The differential semantics requires access to previous execution states when handling control flow constructs. Consider an if statement **if** $e\ s_1\ s_2$ where branch switching occurs, that is, when the condition's value changes from true to false or vice versa. We reiterate the relevant rule D-IF$_{\text{tf}}$ from Figure 3, highlighting in gray the computations that can be cached from the previous execution:

$$\frac{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \textbf{neg} \qquad (\,\sigma \vdash e \Rightarrow \textbf{true}\,) \qquad \sigma \oplus \Delta\sigma \vdash s_2 \Rightarrow \sigma_2 \qquad (\,\sigma \vdash s_1 \Rightarrow \sigma_1\,)}{\sigma, \Delta\sigma \vdash \textbf{if}\ e\ s_1\ s_2 \Rightarrow_\Delta \sigma_2 \ominus \sigma_1}\ \text{D-IF}_{\text{tf}}$$

Applying this rule requires performing new computations while accessing cached results from the previous execution. Specifically, we must differentially evaluate the condition to get **neg** and execute the new branch s_2, while we can reuse the original condition evaluation result and the result of executing s_1. A similar requirement exists for repeat statements, which need stores from previous iterations to enable differential execution. This dependency on previous states explains why our differential reduction relation $\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma'$ takes the previous store σ as input.

Re-computing previous results and propagating old states throughout differential execution incurs significant performance overhead. To address this, we develop an *initializing interpreter* that caches execution states. However, uniquely identifying cached states requires more than just statement identifiers, particularly for loops. Consider **repeat** 5 s where s executes five times – each iteration operates on a different store and thus requires its own cached state.

We solve this through path-sensitive identifiers that precisely track loop nesting structure. A path P starts at root R (nesting level 0) and records each enclosing loop's iteration count. For example, path $R, 2, 1$ indicates execution within the first iteration of an inner loop, nested within the second iteration of an outer loop. Our cache maps each execution point to its corresponding program state: $\textbf{cache} : \textbf{UID} \times \textbf{P} \mapsto \textbf{S}$, where UID uniquely identifies statements in the program, P captures the loop nesting context through iteration counts, and S represents the program state we need to cache.

Our caching strategy for conditional statements preserves three key pieces: the store before evaluating the condition, the condition's evaluated result, and the store after executing the chosen branch. This enables direct access to previously computed states, eliminating redundant recomputation during differential execution.

For repeat statements, we implement two strategies: a basic strategy that caches the initial store, iteration count, and final store, and an enhanced strategy that additionally caches intermediate states after each iteration. The enhanced strategy enables efficient implementation of the decrement optimization (Section 5.2) by providing direct access to any intermediate store from the original execution, eliminating the need for additional computation.

6.2 Additional Language-Features

We have implemented a differential interpreter that supports more constructs than present in the formalization. In addition to integer and boolean values, we also support string values. In the same vein, the language supports more than just the addition operator. It can call arbitrary operators based on an operator interface which requires an `init` function and an `inc` function:

```
1 trait Operator[T, U]:
2   def init(x: T): U
3   def inc(dx: Change[T]): Change[U]
```

The type `Change[T]` marks a change of value type `T`. Note that operators themselves can keep track of local state that is required for implementing efficient incremental operators. Additionally, the language enables first-order function calls of user-defined top-level functions. Each function ranges over its own local store.

One major difference is that the language supports a mutable 2-dimensional table ranging over integers which is global to all functions. To this end, the language supports a table-write statement $\textbf{table}[i, j] = e$ and a table-read expression $\textbf{table}[i, j]$. Note that we currently assume that the indices cannot change between the initialization and a differential run. This is a restriction we want to address in the future work.

6.3 Applying Optimizations

To enable efficient differential execution, we implemented various optimizations. We describe optimizations in the formalization as overlapping rules, where we decide manually which rule to apply. Our implementation always applies the optimization rule whenever possible. One could imagine that there are multiple applicable optimization rules. This leads to *optimization strategies* which select the optimization rule to apply at a given point at execution time. We want to investigate optimization strategies in future work.

We implemented the short-circuiting optimizations from Section 5.1 by preemptively computing read/written variables within statements and expressions. Because our language supports a 2-dimensional global table, this analysis extends to tracking table cell accesses. Building on the branch switching optimization from Section 5.3, we handle table operations efficiently:

```
def bellmanFord(numNodes: Int, src: Int): Unit {
  decl i: Int, u: Int, v: Int, w: Int
  i = 1
  repeat numNodes {
    table[0, i] = Int.Max
    i = i + 1
  }
  table[0, src] = 0
  repeat numNodes - 1 {
    u = 1
    repeat numNodes {
      v = 1
      repeat numNodes {
        w = table[u, v]
        if (w != 0 && table[0, v] > table[0, u] + w) {
          table[0, v] = table[0, u] + w
        } else {
          skip
        }
        v = v + 1
      }
      u = u + 1
    }
  }
}
```

Figure 5 Bellman-Ford Algorithm implemented in our language.

```
if (c) {
  table[i, j] = e1
} else {
  table[i, j] = e2
}
```

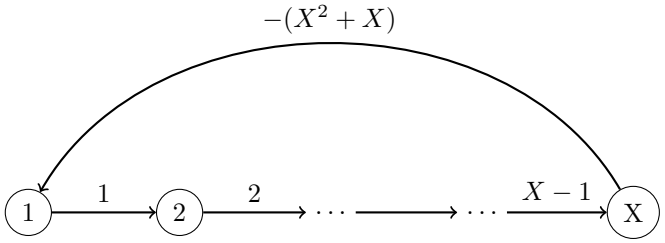
When branches differ only in their expressions (e_1 vs e_2), we can avoid expensive table diffing operations by directly computing $\Delta t[(i,j) \mapsto v_2 \ominus v_1]$ where $\sigma \vdash e1 \Rightarrow v_1$ and $\sigma \oplus \Delta\sigma \vdash e2 \Rightarrow v_2$.

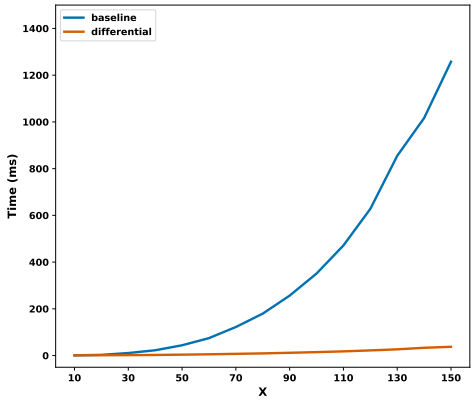
7 Case Study: Shortest-Path with Bellman-Ford Algorithm

To evaluate how precise change descriptions enable efficient incremental computation of dynamic algorithms, we implemented the Bellman-Ford single-source shortest-path algorithm [6] in our language (Figure 5). Our implementation represents the graph using an adjacency matrix stored in a global 2-dimensional table, with shortest path weights from source node `src` to node i stored at indices $(0, i)$.

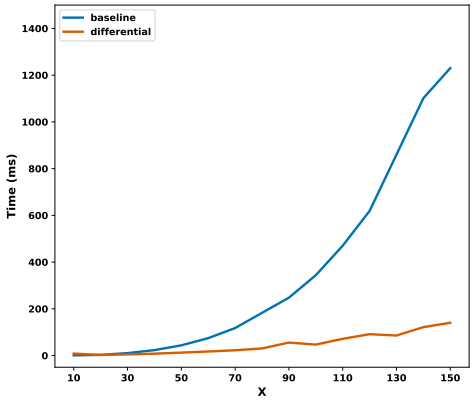
7.1 Setup

To evaluate our differential interpreter, we designed a graph structure that exhibits interesting incremental behavior. The graph is a cycle where we can configure its size by setting the number of nodes $X \in \{10, 30, \ldots, 130, 150\}$:





(a) Differential vs Baseline - Change (i).



(b) Differential vs Baseline - Change (ii).

Figure 6 Running times of Differential Interpretation and Normal Interpretation. Differential execution achieves significant speedups by avoiding redundant work.

The weight of the back edge $X \to 1$ is chosen to introduce new shortest paths, ensuring that small input changes can trigger significant output changes. We compute shortest paths starting from node 1 for different values of X and evaluate two types of changes:

(i) Adding an edge $1 \to (X-1)$ with weight 10^9. This introduces an alternative path that does not affect the shortest path solution.

(ii) Adding six edges forming cross-connections: $1 \to (X-1)$, $2 \to (X-2)$, $6 \to (X-2)$, $(X-5) \to 6$, $(X-3) \to 3$, and $(X-1) \to 2$, each with weight 10^9. Despite creating multiple new paths, these additions do not affect the shortest path solution.

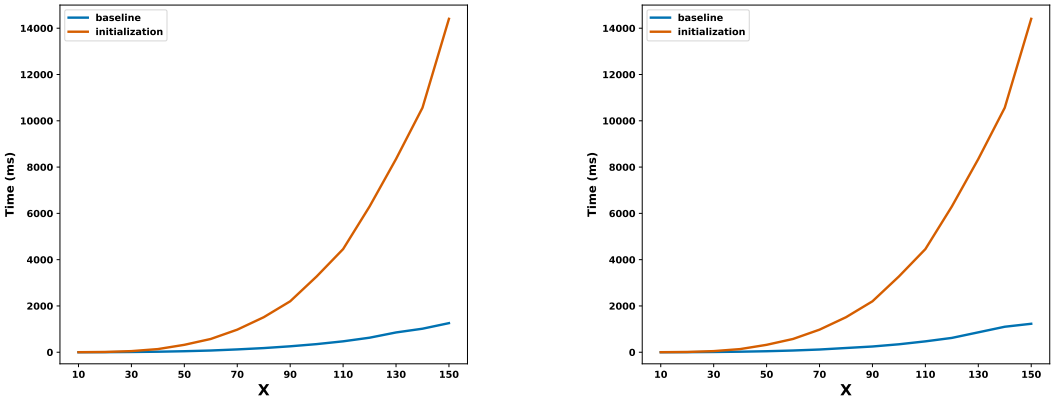
We conducted experiments on an Apple M4 Pro chip with 24GB memory running 64-bit OSX 15.1.1 and OpenJDK 23.0.1. Using the JMH benchmarking framework[3], we performed 5 warmup runs followed by 10 measurement runs for each configuration.

7.2 Results

In Figure 6, we compare the running times of differential interpretation against normal interpretation on the patched input. On the left we show the running times for change (i) and on the right we show them for change (ii). The differential interpreter outperforms normal interpretation for change (i). While Bellman-Ford has a time complexity of $O(X^3)$, the differential interpreter can skip most iterations because we employ an optimization that skips loop iterations when the changed table entries are not affected by the iteration. Since change (i) modifies just a single table entry without introducing new shortest paths, we can skip almost all iterations, yielding significant running time savings. The same benefits apply to change (ii), though to a lesser degree as fewer iterations are skippable.

A second critical optimization avoids diffing when branch switches occur, significantly reducing the differential interpreter's running time. Without this optimization, all branch switches for the inner if statement within the nested repeat loops would require diffing the old and new result tables, incurring an $O(X^2)$ cost. Instead, we apply constant-time updates to the current delta table.

[3] `https://github.com/openjdk/jmh`



(a) Initialization vs Baseline – Change (i).

(b) Initialization vs Baseline – Change (ii).

Figure 7 Running times of Initializing Interpretation and Normal Interpretation. The initial run incurs substantial overhead due to caching.

To evaluate stability over extended use, we executed 40 consecutive changes (Change (ii)) on graphs of varying sizes (10-150 nodes). We measured performance at five points in this sequence (1st, 10th, 20th, 30th, and 40th change), with 5 warmup and 10 measurement runs per point. After filtering outliers using the interquartile range (IQR) method, our analysis confirms that execution times remain consistent across the change sequence for each graph size. The Coefficient of Variation (CV), which measures relative variability as standard deviation divided by mean, is remarkably low (average 8%, maximum 13%), indicating that our differential approach maintains stable performance throughout extended use without degradation. While execution times scale with graph size as expected, the consistent performance within each size category demonstrates that repeated incremental updates do not compromise efficiency, further justifying the one-time initialization cost.

In Figure 7 we compare the running times of the initializing interpreter to normal interpretation on the initial input. On the left we show the running times for change (i) and on the right for change (ii). The graphs show that initialization requires significantly more time than normal interpretation. Normal interpretation takes roughly 1.5s for processing the initial graph where $X = 150$. In contrast, the initialization interpreter requires roughly 15s, a 10x increase. This overhead comes from caching states by inserting entries into hash tables. This overhead is an implementation artifact that we believe can be substantially reduced through engineering improvements and smarter caching strategies.

These results demonstrate that differential interpretation of dynamic algorithms like Bellman-Ford is feasible and can achieve order-of-magnitude speedups, provided we have: 1) optimizations that reduce work by skipping unaffected program fragments, enabled by precise change descriptors, and 2) efficient caching strategies to minimize initialization overhead. Additionally, reducing initialization time remains an important area for improvement.

8 Related Work

Incremental computation has been extensively studied, with various approaches aiming to efficiently update outputs in response to changes in inputs. In this section, we discuss work related to our approach, namely dynamic algorithms, incremental computing by selective recomputing, incremental computing by differential updating, and the incremental lambda calculus.

Dynamic algorithms are designed to handle arbitrary changes to inputs by maintaining data structures that can be efficiently updated [19]. They achieve efficiency through carefully designed update operations that maintain complex invariants specific to each problem domain, as demonstrated in problems like dynamic planar convex hulls [8, 20] and dynamic minimum spanning trees [14, 16]. While these algorithms can offer optimal incremental performance for specific problems, their development requires significant effort and domain expertise [22].

Our approach differs by embedding incrementality into the language semantics rather than relying on specialized data structures. Where dynamic algorithms require manual development of update operations for each problem, our differential semantics provides a systematic approach to deriving incremental behavior for programs written in our imperative language. This language-based approach demonstrates how incremental computation can be treated as a fundamental programming language feature rather than a specialized algorithmic concern.

Selective recomputing techniques aim to reuse previous results by identifying and rerunning only the affected parts of a computation when inputs change [10, 22]. Early methods like function memoization [21, 1] cache function calls and reuse results, but they struggle with programs that have internal state or complex control flow.

Self-adjusting computation [5, 4], representing the state-of-the-art in selective recomputing, introduces sophisticated dependency tracking using dynamic dependency graphs to identify precisely which computations need to be redone. Their change propagation algorithm then selectively recomputes only the affected portions of these dependency graphs. This approach has been enhanced to improve efficiency through data structure-level dependency tracking [3] and extended to handle imperative features [2], but maintains the core mechanism of dependency tracking. Despite these advances, the fundamental dependency tracking approach and its inherent disadvantages remain unchanged. *Adapton* [13] extends this line of work with demand-driven evaluation and better support for cyclic dependencies, but still faces similar overhead challenges from maintaining complex runtime structures.

Where selective recomputation focuses on identifying which computations to redo through dependency information, our differential semantics takes a fundamentally different approach by directly modeling how changes propagate through program constructs. This direct modeling enables more efficient handling of control flow changes: instead of rebuilding and traversing complex dependency graphs when loop bounds or conditions change, our semantics directly computes the precise differential effect of the modified control flow. The key distinction is that while self-adjusting computation and its extensions determine *what* to recompute, our differential semantics specifies *how* changes transform program states through the language semantics itself.

The incremental lambda calculus (ILC) [9, 11] provides a theoretical foundation for incrementalizing purely functional programs through type-directed program transformation. For each function f, it derives a derivative function f' that computes output changes from input changes, executing these derivative programs using standard semantics.

Our approach differs fundamentally from ILC by providing a differential semantics that directly executes programs on changes without requiring program transformation. This distinction has important practical implications – optimizing our differential semantics requires only modifying semantic rules, while ILC must optimize its program transformation process to generate more efficient derivative code. Another obvious difference between our approaches is that while ILC provides powerful techniques for incrementalizing higher-order functional programs, our differential semantics offers direct support for imperative features.

Differential updating originated in databases to maintain materialized views efficiently [7]. For each relational operator, they derive rules that translate changes to input relations into changes to output relations, avoiding full recomputation. View maintenance systems formalize these as delta rules [12] with algebraic properties that ensure correctness: a delta rule applied to input changes must produce the same result as recomputing with updated inputs. This approach has been particularly successful for Datalog, where differential evaluation enables efficient incremental maintenance of recursive views [18].

Our work shares the fundamental insight of deriving change-propagation rules but applies it to programming language constructs rather than relational operators. Like database approaches, we ensure correctness through algebraic properties, but our properties focus on language-level operations rather than relational algebra. This enables incrementalization of general-purpose programs while maintaining the mathematical rigor of differential database techniques.

Differential dataflow [17] extends dataflow systems to support incremental updates through data provenance tracking. While effective for data-parallel computations, it requires programs to be expressed in a restricted dataflow model. Our approach provides similar benefits for general imperative programs through language-level change tracking, making incremental computation accessible to mainstream programming.

9 Conclusion

This paper presents differential semantics, which exploits precise change information rather than using all-or-nothing selective recomputing. Rather than treating incrementality as a separate concern, we embed it directly in the programming language through semantic rules that precisely track and propagate changes. Our key contributions include a type-safe theory of changes that ensures semantic consistency, a differential semantics for an imperative language that handles both data and control flow changes, and a set of verified optimizations that make this approach practical. Through our implementation as a differential interpreter, validated on the Bellman-Ford algorithm, we demonstrate that our approach enables order-of-magnitude speedups compared to recomputation. These results establish that principled language design can make incremental computation both automatic and efficient. Future work could extend these foundations to richer language features like functions and recursion, moving toward general-purpose languages where incrementality is the default rather than a specialized capability.

References

1 Martín Abadi, Butler Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–91, 1996. doi:10.1145/232627.232638.

2 U. A. Acar, A. Ahmed, and M. Blume. Imperative self-adjusting computation. *SIGPLAN Not.*, 43(1):309–322, 2008.

3 U. A. Acar, G. Blelloch, R. Ley-Wild, K. Tangwongsan, and D. Turkoglu. Traceable data types for self-adjusting computation. *SIGPLAN Not.*, 45(6):483–496, 2010. doi:10.1145/1806596.1806650.

4 Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Carnegie Mellon University, 2005.

5 Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 247–259, 2002. doi:10.1145/503272.503296.

6 Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

7 John A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 61–71, 1986.

8 Gerth Stølting Brodal and Riko Jacob. Dynamic planar convex hull. In *Proceedings of the 43rd Annual Symposium on Foundations of Computer Science*, pages 617–626, 2002. doi:10.1109/SFCS.2002.1181985.

9 Yan Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: Incrementalizing lambda-calculi by static differentiation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 145–155, 2014.

10 Allan Demers, Thomas Reps, and Tim Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 105–116, 1981. doi:10.1145/567532.567544.

11 Paolo G. Giarrusso, Yann Régis-Gianas, and Philipp Schuster. Incremental lambda-calculus in cache-transfer style: Static memoization by program transformation. In *Proceedings of the 28th European Symposium on Programming*, pages 553–580, 2019.

12 Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 157–166, 1993. doi:10.1145/170035.170066.

13 Michael A. Hammer, Kyle Y. J. Phang, Michael Hicks, and Jeffrey S. Foster. Adapton: Composable, demand-driven incremental computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–166, 2014.

14 Monika Rauch Henzinger and Valerie King. Algorithmic aspects of dynamic graph connectivity. *Algorithmica*, 22(1–2):119–146, 1998.

15 Herbert W. Hethcote. The mathematics of infectious diseases. *SIAM Review*, 42(4):599–653, 2000. doi:10.1137/S0036144500371907.

16 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48:723–760, 2001. doi:10.1145/502090.502095.

17 Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research*, pages 1–11, 2013.

18 Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. Incremental update of datalog materialisation: The backward/forward algorithm. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 1560–1568, 2015.

19 Mark H. Overmars. *The Design of Dynamic Data Structures*. Springer-Verlag, 1983.

20 Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166–204, 1981. doi:10.1016/0022-0000(81)90012-X.

21 William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 315–328, 1989. doi:10.1145/75277.75305.

22 Ganesan Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 502–510, 1993.

GSOHC: Global Synchronization Optimization in Heterogeneous Computing

Soumik Kumar Basu
Department of Computer Science and Engineering, IIT Hyderabad, India

Jyothi Vedurada
Department of Computer Science and Engineering, IIT Hyderabad, India

Abstract

The use of heterogeneous systems has become widespread and popular in the past decade with more than one type of processor, such as CPUs, GPUs (Graphics Processing Units), and FPGAs (Field Programmable Gate Arrays) etc. A wide range of applications use both CPU and GPU to leverage the benefits of their unique features and strengths. Therefore, collaborative computation between CPU and GPU is essential to achieve high program performance. However, poorly placed global synchronization barriers and synchronous memory transfers are the main bottlenecks to enhanced program performance, preventing CPU and GPU computations from overlapping.

Based on this observation, we propose a new optimization technique called *hetero-sync motion* that can relocate such barrier instructions to new locations, resulting in improved performance in CPU-GPU heterogeneous programs. Further, we propose GSOHC, a compiler analysis and optimization framework that automatically finds opportunities for hetero-sync motion in the input program and then performs code transformation to apply the optimization. Our static analysis is a context-sensitive, flow-sensitive inter-procedural data-flow analysis with three phases to identify the optimization opportunities precisely. We have implemented GSOHC using LLVM/Clang infrastructure. On A4000, P100 and A100 GPUs, our optimization achieves speedups of up to 1.8x, 1.9x and 1.9x over the baseline, respectively.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Frameworks; Software and its engineering → Automated static analysis

Keywords and phrases Static Analysis, Synchronization, CPU-GPU, Heterogeneous Computing, Parallelization

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.21

Supplementary Material *Software (ECOOP 2025 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.11.2.17`

Funding *Soumik Kumar Basu*: Funded by SERB SRG.

Acknowledgements We thank Anilava Kundu for his insightful comments during implementation. Further, we thank the anonymous reviewers of ECOOP for their constructive feedback.

1 Introduction

Heterogeneous computing is a programming paradigm that uses more than one type of processor, such as a CPU, in conjunction with accelerators, such as GPUs (Graphics Processing Units) and FPGAs (Field Programmable Gate Arrays). In a wide range of applications, CPUs and GPUs are both used, and both have strengths and limitations, which are essential for high-performance computing. Over the decades, several compiler optimizations [45, 6, 38] have been proposed to improve the performance of CPU programs. In the same vein, numerous recent research works [12, 33, 51, 1, 2] have developed new analyses and optimizations to enhance only GPU (device) side performance in the CPU-GPU heterogeneous applications. Nevertheless, CPU- and GPU-only optimization strategies are incapable of realizing the full

© Soumik Kumar Basu and Jyothi Vedurada;
licensed under Creative Commons License CC-BY 4.0

```
1  void foo(int *x, int size2)
2  {
3    ...
4    if(k < y_points)
5      x[k] = x[0] + y_points;
6    ...
7  }
8  void main(){
9    ...
10   kernel1<<<grid1, block1>>>(d_u,...);/*Device-side computations*/
11   cudaMemcpy(u, d_u, size1, cudaMemcpyDeviceToHost); /*Sync*/
12   ...
13   new_data = processData(data); /*Host-side computations*/
14   ...
15   cudaMemcpy(d_u, new_data, data_size, cudaMemcpyHostToDevice);
16   kernel2<<<grid2, block2>>>(d_u,...);/*Device-side computations*/
17   cudaMemcpy(v, d_u, size2, cudaMemcpyDeviceToHost);
18
19   cudaDeviceSynchronize();/*Sync*/
20   ...
21   /*Host-side computations*/
22   foo(v, size2);
23   ...
24   for(int i=0; i<size2; i++)
25     v[i] = v[i] + x_points;  ...
26 }
```

Figure 1 Sample CUDA code snippet showing blocking calls, host- and device-side computations.

potential of heterogeneous computing. For example, a GPU can remain idle during memory copy to CPU, and specialized global optimizations [23, 4, 27] are needed to improve overall performance by efficiently managing host-to-device data transfers.

In a similar vein, heterogeneous systems suffer from another important global performance problem: the CPU being idle after launching the kernel and allocating the task to the GPU, resulting in the underutilization of processors. The importance of this problem has been revealed through several manual optimization efforts [7, 46, 55, 47] of several domain-specific heterogeneous applications (efficiently distributing the workload between CPU and GPU). Further, in a performance study by White III et al. [53], manual optimization of CPU-GPU computation overlap yielded significantly better performance than most other optimization methods, such as communication-computation overlap etc. A poorly placed synchronization statement or blocking call is most often the reason for the CPU idleness in a heterogeneous system since it prevents CPU and GPU computations from overlapping.

Optimization Opportunities and Associated Challenges. Consider a sample example in Figure 1. There are two device kernel launches from the CPU function `main`, one at Line 10 and the other at Line 16. A device kernel is a function that runs asynchronously with CPU code on multiple cores of a GPU (in SIMT [20] fashion), and we refer to its computations as *device-side computations*. Programmers must explicitly add synchronization calls (e.g. `cudaDeviceSynchronize` call at Line 19) in their code to ensure that CPU-side computations halt until the device-side computations are complete. Further, memory transfers (e.g. `cudaMemcpy` calls at Lines 11, 15, and 17) also act as synchronization/blocking calls in the CUDA programming model. We can see that the host-side computation at Line 13 is not dependent on the device-side kernel computation at Line 10, as it does not use variables `u`, `d_u` and `size1`. Nevertheless, the host-side computation still awaits for completion of the device kernel due to the poorly placed synchronous data-transfer call at Line 11, resulting in suboptimal performance. As a result, as shown in Figure 2 (above the horizontal dotted lines), the program takes $t_2 - t_0$ time to execute the program statements from Line 10 to

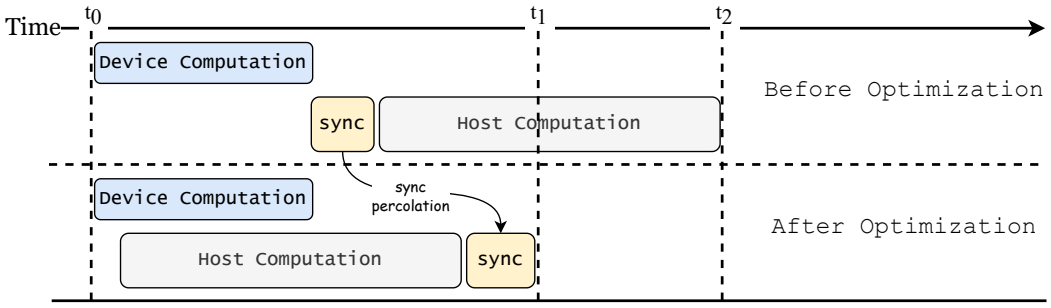


Figure 2 Execution of host- and device-side computations before and after hetero-sync motion.

Line 13. Alternatively, if the barrier is relocated after the host-side computation (say, from Line 11 to Line 14), the CPU will not wait for the kernel to complete, resulting in parallel execution of host- and device-side computations. Figure 2 (below the horizontal dotted lines) indicates that the time will then reduce from $t_2 - t_0$ to $t_1 - t_0$. This single synchronization statement relocation has opened a new avenue for optimizing program execution time. We call this optimization *hetero-sync motion*.

Manually identifying the target location for hetero-sync motion on a given synchronization statement is challenging. Firstly, for the synchronization statement at Line 11, it is required that the relocation target cannot be a successor of Line 15, which uses the variable `d_u`. Further, alias relationships between variables should be taken into account when performing this verification. Secondly, for the synchronization statement at Line 17, simply checking for data dependency may result in incorrectly moving it to Line 5, which is not guaranteed to get executed in all program paths, unlike the source location (this may also result in redundant or spurious data transfers). Thirdly, for the synchronization statement at Line 19, although it has no argument variables, to find its target location, we need to consider the farthest successor in the execution path for performance without violating previous synchronization dependencies. Fourthly, a kernel call (Line 16) can be associated with several synchronization calls (Line 17 and Line 19), and the challenge is to relocate all these synchronizations to the most suitable target location. The code snippet in Figure 3a from the `xlqc` benchmark program, which computes Coulomb and exchange matrices for quantum chemistry simulations, exemplifies such challenges in real-world applications. Here, `memcpy` operations at Lines 8 and 13, though dependent on Line 26, cannot be directly moved because doing so would make them conditional on the for loop at Line 19 resulting in correctness issues or multiple executions that degrade performance. Therefore, we need a dedicated analysis that addresses these challenges precisely and efficiently.

Furthermore, real-world programs exhibit the complexity of finding optimal target locations for synchronization statements across functions. Consider the code snippet in Figure 3b, from `CLINK` [11] (Compact LSTM INference Kernel), which performs efficient sequence prediction by minimizing memory and computation overhead in Long Short-Term Memory (LSTM) models. The synchronous `memcpy` call at Line 8 (along with `cudaDeviceSynchronize` call at Line 6) can be moved outside the `lstm_n5` function and placed after Line 19, thereby overlapping the kernel computation (at Line 5) with `init` (at line 18). Identifying such optimization opportunities requires a thorough analysis of call sites, precise mapping of function arguments, and context- and flow-sensitive analysis, posing significant challenges even for experienced developers. Moreover, the subsequent transformation requires adjusting function definitions, callsite arguments, and relocating instructions across functions, as seen

```
1 void main(){
2  ...
3  while(...){
4   if (use_dp)
5    cuda_mat_J_PI_dp<<<...>>>(...);
6   else
7    cuda_mat_J_PI<<<...>>>(...);
8   cudaMemcpy(h_mat_J_PI, dev_mat_J_PI,
        size_N_BYTES, cudaMemcpyDeviceToHost);
9   if (use_dp)
10   cuda_mat_K_PI_dp<<<...>>>(...);
11  else
12   cuda_mat_K_PI<<<...>>>(...);
13  cudaMemcpy(h_mat_K_PI, dev_mat_K_PI,
        size_N_BYTES, cudaMemcpyDeviceToHost);
14   for (int a = 0; a < p_basis->num; ++a) {
15     for (int b = 0; b <= a; ++b) {
16        ...
17     }
18   }
19   for (int i = 0; i < n_pbf; ++i)
20   { ...
21     for (int j = 0; j < n_pbf; ++j)
22     {
23       int b = h_pbf_to_cbf[j];
24       if (a < b) { continue; }
25       ...
26       ▶h_mat_J[ab] += h_mat_J_PI[ij];
27       h_mat_K[ab] += h_mat_K_PI[ij];
28     }
29   } ...
30  } ...
31 }
```

(a) Code snippet from `xlqc` [58].

```
1 long lstm_n5(..., float* y)
2 { ...
3  float *d_y;
4  ...
5  lstm_inference<<<...>>>(...,
       d_outB, d_y);
6  cudaDeviceSynchronize();
7  ...
8  cudaMemcpy(y, d_y, ...,
       cudaMemcpyDeviceToHost);
9  ...
10 }
11
12 int main(int argc, char* argv[])
13 { ...
14  for (int n = 0; n < repeat; n++)
15  { ...
16    lstm_n5(sample_input, inW,
          intW, intB, outW, &outB,
          infer1_out);
17    ...
18    init(work_path, input_filename
          , weight2_filename,
          sample_input, in, intW,
          intB, outW, &outB);
19    lstm_n5(sample_input, inW,
          intW, intB, outW, &outB,
          infer2_out);
20    ▶...
21    free(infer1_out);
22    free(infer2_out);
23   ...}
24  ...}
```

(b) Code snippet from `clink` [11].

Figure 3 Code snippet demonstrating opportunities in hetero-sync motion. The yellow box highlights synchronization statements, while the blue triangle (▶) marks the destination location.

in this example, where the `memcpy`'s arguments at Line 3 must be moved to the `main` (caller) function and passed as arguments to the `lstm_n5` (source) function. Further, the `memcpy` statement in `lstm_n5` function is invoked from two contexts (Lines 16 and 19), leading to the percolation of synchronization statements from both contexts to the target location. To address these complexities and reduce programming overhead, we propose an automated framework, GSOHC, to identify and move synchronization statements to optimal target locations, streamlining the code transformation process.

Existing Approaches and the Need for a New Solution. Hetero-Sync Motion optimization is analogous to compiler optimizations [42, 19, 16, 41, 17, 43] that relocate inefficient synchronization in the context of alternative programming paradigms such as MPI (Message Passing Interface) and CPU TLP (Thread-Level Parallelism). However, these approaches are inadequate to address the problem of hetero-sync motion due to the semantic differences in synchronization operations: (1) APIs such as `cudaMemcpy` in CPU-GPU heterogeneous programs perform both synchronization and data-transfer operations, unlike `wait` calls in TLP or MPI paradigms that only handle synchronization, (2) by default, synchronization statements in CUDA execute in the default stream, and data transfer begins only after all previous CUDA calls are completed, whereas MPI `wait` calls wait for the completion of a specific communication call based on the corresponding `request_ID` (similar to TLP). Using TLP approaches [42, 19, 43] for hetero-sync motion may percolate *sync* calls into divergent branch paths (e.g., moving Line 17 to Line 5 and Line 25 in Figure 1) resulting in redundant

execution of data transfers that may adversely affect the correctness and performance of a CPU-GPU program. On the other hand, MPI approaches [16, 41, 17] require a one-to-one pairing of `wait` calls with non-blocking communication calls, and handling each pair separately through data dependence checks; whereas, in CPU-GPU heterogeneous systems, the challenge lies in consolidating multiple synchronization calls surrounding a single kernel call into one target location. Further, previous works [23, 27, 4, 26, 30] have focused on CPU-GPU global optimization problems through techniques such as multi-kernel computation overlap and data transfer–kernel computation overlap using hybrid runtime and compile-time approaches. However, they do not address the synchronization relocation challenges posed by hetero-sync motion. Therefore, new static data-flow analysis approaches are needed to address synchronization bottlenecks in CPU-GPU heterogeneous programs by analyzing the flow of multiple synchronization calls and their associated data.

GSOHC. In Figure 2, below the horizontal dotted line, we can see how GSOHC pushes the synchronization statement after the host-side computation, facilitating overlap between host-side and device-side computations, thereby enhancing overall program performance. GSOHC does not optimize the host-side nor device-side computations, but it improves performance by only pushing the barrier to a successor program point in the execution path. Our static analysis is a context-sensitive, flow-sensitive inter-procedural data-flow analysis with three phases to identify the optimization opportunities precisely: (1) identifies the poorly placed synchronization statements in the code and maps them to target location sets (not unique since dependencies can be found in multiple program points in different execution paths), (2) updates the mapped location sets based on the control flow and contextual information, and (3) unifies the location sets so that each synchronization statement in a given context is mapped to a unique location.

We implemented our framework using LLVM/Clang infrastructure [36, 31]. GSOHC takes the input program in LLVM/IR format and produces the optimized IR with overlapping host-side computation and device-side computation, resulting in performance improvement. The key contributions of this work are as follows:

- A novel idea of hetero-sync motion optimization that relocates synchronization statements from a source location to a target location to improve performance in CPU-GPU heterogeneous programs.
- A novel compiler framework with three stages, including an interprocedural, context- and flow-sensitive data-flow analysis pass to identify optimization opportunities and determine optimal target locations for synchronization statements.
- A transformation pass that automatically relocates barrier statements to an optimal location so that host-side and device-side computations can overlap. Implementation of both passes in the LLVM infrastructure.
- An extensive evaluation of our approach and comparison with the base code. On A4000, P100, and A100 GPUs, our optimization achieves up to 1.8x, 1.9x, and 1.9x speedups over baseline, respectively.

2 Formalization and Overview

In this section, we discuss some preliminaries that are required to explain our static analysis in Section 3, and then we formally define our hetero-sync motion optimization.

We use the programming language in Figure 4 for our formalization. While we use this language to simplify our presentation, our implementation (see Section 5 for details) works on LLVM IR. It is easy to correlate every analysis step discussed in this paper with LLVM

Program $\mathcal{P}$	=	$s;\ m^+$
Method m	=	$\mathcal{H}(\vec{v})\{s_h\} \mid \mathcal{D}(\vec{v})\{s_d\}$
Host statement s_h	=	**call** $\mathcal{H}(\vec{e}) \mid sync \mid s \mid s_d \mid$ **return** v
Device statement s_d	=	**call** $\mathcal{D}(\vec{e}) \mid s$
Synchronization call $sync$	=	**call** $global_sync_api(\vec{e})$
Expression e	=	$v \mid c \mid e_1 \oplus e_2 \mid \neg e \mid e_1 \vee e_2 \mid e_1 \wedge e_2 \mid e_1 \otimes e_2$
Statement s	=	$v := e \mid$ **if**(e) s **else** $s \mid$ **while**(e) **do** $s \mid s; s \mid$ **skip**

$v \in$ **Variable** $\quad c \in$ **Constant** $\quad \vec{v} = (v_1, v_2, \ldots v_n)$

$\oplus \in \{+, -, \times, \div\} \quad \otimes \in \{<, >, \leq, \geq, =, \neq\} \quad \vec{e} = (e_1, e_2, \ldots e_n)$

Figure 4 Syntax of the input program.

IR instructions. Our implementation supports full LLVM IR, including pointer operations, though we do not show them in Figure 4. Furthermore, we consider heterogeneous computing systems with a CPU and a single GPU setup. Although our ideas are generic, we use the CUDA [8] programming model whenever we discuss concrete semantics.

The program $\mathcal{P}$ is a heterogeneous program consisting of statements and methods that can run on a CPU or a GPU. $\mathcal{D}(\vec{v})$ represents a method whose body of statements gets executed on a GPU (device) [1]. $\mathcal{H}(\vec{v})$ represents a method executed on a CPU (we do not specify the return type for simplicity). The statement *sync* denotes function calls that serve as global barriers to synchronize CPU and GPU. In CUDA programming model, $global_sync_api(\vec{e})$ can represent `cudaMemcpy` or `cudaDeviceSynchronize` calls. In addition, variable accesses v, expressions e, assignment statements ($v := e$), branch and loop instructions, etc., are all standard programming constructs. We now formally define some terminology that is required to introduce hetero-sync motion optimization.

▶ **Definition 1** (Device-side Computation). *We say an operation d is* device-side computation *if it resides within the body of a method $\mathcal{D}$ executed on a device (using multiple threads in SIMT fashion).*

▶ **Definition 2** (Host-side Computation). *We say an operation h is* host-side computation *if it resides within the body of a method $\mathcal{H}$ executed on a host (using one or more threads in SIMD or MIMD or SISD fashion).*

▶ **Definition 3** (Synchronization Call, *sync*). *In a heterogeneous program $\mathcal{P}$, a program point or statement s_i within a host-side method $\mathcal{H}$ is called a* synchronization call *(or sync) if it invokes dedicated APIs that cause $\mathcal{H}$ to wait at s_i until all device-side methods $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n$ launched prior to s_i complete execution.*

For example, `cudaMemcpy` call (Line 17, Figure 1) and `cudaDeviceSynchronize` call (Line 19, Figure 1) are synchronization calls.

The goal of GSOHC is to identify large regions of code with host-side computations that are independent of a *sync* and result in overlap with device-side computations, maximizing performance through concurrent execution.

▶ **Definition 4** (Data Dependence). *Given two program statements s_i and s_j in a heterogeneous program $\mathcal{P}$, we say s_j is* data dependent *on s_i or vice versa iff*

$$(rd(s_i) \cap wr(s_j)) \cup (wr(s_i) \cap wr(s_j)) \cup (wr(s_i) \cap rd(s_j)) \neq \phi$$

where $rd(s_i)$ gives the set of variables read by statement s_i and $wr(s_i)$ gives the set of variables written by statement s_i.

[1] We use "GPU" and "device" interchangeably and, similarly, "host" and "CPU" in this paper.

For example, in Figure 1, Line 17 is data dependent on Line 5, since Line 5 reads and writes to the same data array that is written by Line 17.

▶ **Definition 5** (Statement Region, Ω). *In a program $\mathcal{P}$ with Inter-procedural Control Flow Graph $ICFG(\mathcal{P})$, a* statement region $\Omega(p_i, p_j)$ *is the sequence of program statements between two program points p_i and p_j (excluding p_i and p_j), where the basic block of p_j is a successor of the basic block of p_i in $ICFG(\mathcal{P})$.*

▶ **Definition 6** (Safe Region, $\mathcal{R}$). *A statement region $\Omega(p_i, p_j)$ is a* safe region, *$\mathcal{R}(p_i, p_j)$, if no statement in $\Omega(p_i, p_j)$ is data-dependent on the statement at p_i.*

▶ **Definition 7** (Refined Safe Region, $\mathcal{K}$). *A* refined safe region, *$\mathcal{K}(p_i, p_j)$ is derived from a safe region $\mathcal{R}(p_i, p_k)$ (represented as $\mathcal{R}(p_i, p_k) \vdash \mathcal{K}(p_i, p_j)$) iff:*

$$(p_i \textit{ dominates } p_j) \wedge (p_j \textit{ post-dominates } p_i) \wedge (p_j \textit{ dominates } p_k)$$

While $\mathcal{R}$ considers only data dependency, $\mathcal{K}$ considers both data dependency (from $\mathcal{R}$) and control dependency (via dominance relations). A statement p_i *dominates* p_j if all paths from the entry node to p_j pass through p_i, and p_j *post-dominates* p_i if all paths from p_i to the exit node pass through p_j.

▶ **Definition 8** (Unified Safe Region, $\mathcal{U}$). *A* unified safe region, *$\mathcal{U}(p_i, p_j)$ corresponds to a unique refined safe region, $\mathcal{K}(p_i, p_m)$, from the set of refined safe regions for p_i $\{\mathcal{K}_1(p_i, p_1)$, $\mathcal{K}_2(p_i, p_2)$, $\mathcal{K}_3(p_i, p_3)$, …, $\mathcal{K}_n(p_i, p_n)\}$ iff:*

$$\exists m \forall s,\ m, s \in \{1, \ldots, n\} \mid (p_m \textit{ dominates } p_s)$$

While the refined safe region $\mathcal{K}$ addresses control and data dependencies, multiple regions for p_i may result in redundant executions of a *sync* call along a path. The unified region $\mathcal{U}$ uniquely updates and unifies these regions for p_i.

▶ **Definition 9** (Hetero-Sync Motion). *Given a program point p_{src} with a sync call in a host-side method $\mathcal{H}$, if $\mathcal{U}(p_{src}, p_{dst})$ is a unified safe region, relocating the sync from p_{src} to p_{dst} to enable concurrent execution of a host-side statement $s_h \in \mathcal{U}(p_{src}, p_{dst})$ and a device-side computation statement s_d is called as* Hetero-Sync Motion.

Since *sync* calls block host-side and device-side computations from overlapping, our optimization *Hetero-Sync Motion* aims to move the *sync* call from program point p_{src} to p_{dst} to enable concurrent execution and improve performance.

▶ **Example.** In Figure 1, Ω(Line 17, Line 5) is an example of a statement region, which is also a safe region i.e., $\mathcal{R}$(Line 17, Line 5). Since Line 5 does not post-dominate Line 17, it cannot form a refined safe region. Hence, Line 17 and Line 3 form a refined safe region, i.e., $\mathcal{K}$(Line 17, Line 3). Similarly, another refined safe region is $\mathcal{K}$(Line 17, Line 23). The unified safe region becomes $\mathcal{U}$(Line 17, Line 3), indicating that hetero-sync motion optimization percolates the *sync* from source Line 17 to target Line 3.

Overview. Figure 5 depicts the pipeline of our GSOHC. Given a program $\mathcal{P}$ as input, GSOHC performs hetero-sync motion optimization (see Definition 9) on $\mathcal{P}$ and produces an optimized program $\mathcal{P}_{opt}$ automatically. GSOHC consists of two major components: a top-down inter-procedural context- and flow-sensitive data-flow analysis pass to identify the opportunities for hetero-sync motion optimization and a transformation pass to optimize the input code using the analysis results. Further, our static analysis framework follows

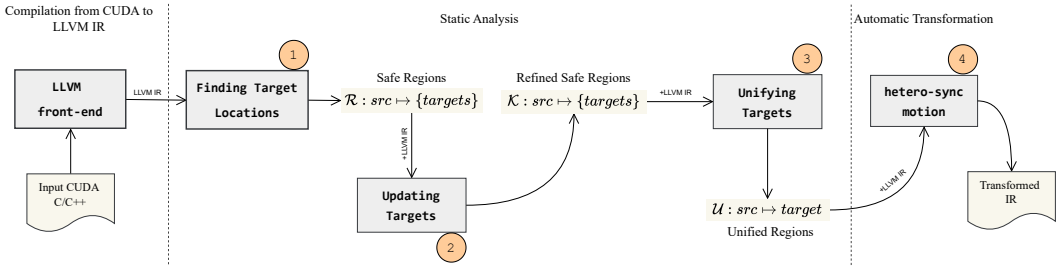


Figure 5 Complete pipeline of GSOHC.

a three-stage approach. The first stage identifies target location sets for poorly placed synchronization statements, represented as safe regions (see Definition 6). The second stage updates these safe regions to maintain the control-flow dependency between the source and target locations to form refined safe regions (see Definition 7). The third stage unifies each target location set into a single, unified safe region (see Definition 8), ensuring each *sync* statement is mapped to a unique location. Finally, the transformation pass relocates *sync* statements from source to target locations, to enhance the performance of the input programs. We will discuss the different stages of the analysis pass in Section 3 and then discuss the transformation pass in Section 4.

3 Static Detection of Hetero-Sync Motion Opportunities

Our static analysis for identifying opportunities of hetero-sync motion optimization takes a program $\mathcal{P}$ as input and outputs a map $\mathcal{M}$, which maps a *sync* statement to a set of target statement(s) in $\mathcal{P}$. The *sync* statement representing a global barrier call (see Definition 3) in $\mathcal{P}$ is considered as a source location, and the corresponding target location (singleton set at the end) is identified by our static analysis based on the concept of unified safe region (Definition 8). We will discuss the three stages of our static analysis in the following sections: finding target location sets (Section 3.1), updating target location sets (Section 3.2), and unifying target location sets (Section 3.3).

3.1 Finding Target Location Sets

The key idea is to perform a top-down context- and flow-sensitive inter-procedural data-flow analysis to identify the hetero-sync motion optimization opportunities in the form of safe regions (see Definition 6). Specifically, our analysis reports for each *sync* statement at p_{src} [2] a set of target locations, which is a collection of p_{dst} program points corresponding to safe regions identified in multiple execution paths.

Figure 6 shows the transfer function rules of our static analysis for statements in the language described in Figure 4. The first rule shows that if a program statement s is a synchronization call (see Definition 3), the analysis collects the variables that are read and written by s into sets $\mathbf{R}$ and $\mathbf{W}$, respectively, and then inserts s into the set Σ. In this paper, we refer to the sets $\mathbf{R}$ and $\mathbf{W}$ as the *read* set and the *write* set, respectively, representing the variables that are read from and written to by the *sync* statements. Likewise, we refer to Σ

[2] We use the words "source statement" and *sync* interchangeably, and, similarly, the words target and destination in this paper.

[SYNCHRONIZATION CALL]

$$\frac{s : \textbf{call } \mathit{global_sync_api}(\vec{e}) \quad e_i \in \vec{e}}{\mathbf{R}^s_{out} := \mathbf{R}^s_{in} \cup \texttt{rd}(e_i) \quad \mathbf{W}^s_{out} := \mathbf{W}^s_{in} \cup \texttt{wr}(e_i) \quad \Sigma^s_{out} := \Sigma^s_{in} \cup \{s\} \quad \mu^s_{out}[s] := \{s\}}$$

[HOST FUNCTION CALL]

$$\frac{s : \textbf{call } \mathcal{H}(\vec{e}) \quad s_1 : \texttt{entryPoint}(\mathcal{H})}{\mathbf{R}^{s_1}_{in} := \mathbf{R}^s_{in} \quad \mathbf{W}^{s_1}_{in} := \mathbf{W}^s_{in} \quad \Sigma^{s_1}_{in} := \Sigma^s_{in} \quad \mu^{s_1}_{in} := \mu^s_{in} \quad \Gamma := \Gamma \cdot s}$$

[ASSIGN]

$$\frac{s : v := e \quad var \in \texttt{rd}(e) \quad w \in \mathbf{W}^s_{in} \quad r \in \mathbf{R}^s_{in} \quad (var \odot w \ \vee \ v \odot w \ \vee \ v \odot r)}{\mathbf{R}^s_{out} := \emptyset \quad \mathbf{W}^s_{out} := \emptyset \quad \Sigma^s_{out} := \emptyset \quad \forall src \in \Sigma^s_{in} \ \mu^s_{out}[src] := \{s\}}$$

[DEVICE FUNCTION CALL]

$$\frac{\begin{array}{c} s : \textbf{call } \mathcal{D}(\vec{e}) \quad e_i \in \vec{e} \quad v_r \in \texttt{rd}(e_i) \ v_w \in \texttt{wr}(e_i) \quad w \in \mathbf{W}^s_{in} \quad r \in \mathbf{R}^s_{in} \\ (v_r \odot w \ \vee \ v_w \odot w \ \vee \ v_w \odot r) \end{array}}{\mathbf{R}^s_{out} := \emptyset \quad \mathbf{W}^s_{out} := \emptyset \quad \Sigma^s_{out} := \emptyset \quad \forall src \in \Sigma^s_{in} \ \mu^s_{out}[src] := \{s\}}$$

Figure 6 Transfer function rules for our flow- and context-sensitive static analysis. Here, $\texttt{rd}(e)$ and $\texttt{wr}(e)$ denote the sets of variables read and written by the expression e, respectively. Further, the target map μ stores each *sync* statement with its corresponding set of target locations.

as the *synchronization* set, which stores the *sync* statements. In the rule, $\mathbf{R}^s$ represent the flow-sensitive instance of the data-flow fact $\mathbf{R}$ at program point s, while sets $\mathbf{R}^s_{in}$ and $\mathbf{R}^s_{out}$ represent the data-flow facts flowing to and from statement s (similarly for $\mathbf{W}$, Σ and μ). Note that for the simplicity of the design of our static analysis, we have assumed that each basic block in an ICFG contains a single program statement.

As the analysis aims to identify an optimization opportunity in the form of safe region $\mathcal{R}(p_{src}, p_{dst})$, the first rule involves identifying the start of a safe region at the program point s [3] (i.e., s represents p_{src} here). This rule also initializes the target locations (end of safe regions) of s by adding itself to the set $\mu^s[s]$, which other rules can update later. We refer to μ as *target* map, which stores the mapping of a *sync* statement to the set of target locations where it can be percolated.

▶ **Example.** In Figure 7, `cudaMemcpy` call at Line 19 reads from `arr_gpu` and writes to `arr_output_gpu`. Hence, the read set and the write set are {`arr_gpu`} and {`arr_output_gpu`}, respectively. Further, the synchronization set includes Line 19, and the target map μ maps Line 19 to itself, i.e. μ[Line 19] = {Line 19}.

The second rule handles host function calls. As the analysis is inter-procedural, it analyzes the callee host function at each call site. It first propagates the data-flow facts from the call site to the entry point of the callee and appends the call site to the *call-string* Γ, which stores the function calling context (i.e., caller-callee sequence as a string) and then analyzes the callee. In our analysis, we restrict the length of the call-string to 4.

[3] Whenever we refer to a program statement, we refer to it in a given calling context Γ, i.e. $s \equiv (\Gamma, s)$.

```
1  void foo(...,arr_output_gpu,...)
2  {
3      /*entry point of function foo*/
4      int *x = malloc(n * sizeof(int));
5      ...
6      if(pred){
7          for(i=0;i<n;i++){
8              x[i] = arr_output_gpu[i];
9          }
10     }
11     ...
12     return;
13 }
14 void main(){
15     ...
16     /* Device-side computation */
17     cuda_kernel<<< grid,block >>>(..., arr_gpu);
18     /* barrier */
19     cudaMemcpy(arr_output_gpu, arr_gpu, size, cudaMemcpyDeviceToHost);
20     ...
21     foo(...,arr_output_gpu,...);
22     ...
23 }
```

Figure 7 Sample code with hetero-sync motion optimization opportunity.

▶ **Example.** In Figure 7, the function `main` invokes function `foo` at Line 21. Hence, the data-flow facts at the call site are propagated to the entry point of function `foo` (Line 4). The corresponding read set, write set, synchronization set, and target map for Line 4 are: $\mathbf{R}^s_{in} = \{\texttt{arr_gpu}\}$, $\mathbf{W}^s_{in} = \{\texttt{arr_output_gpu}\}$, $\Sigma^s_{in} = \{\text{Line 19}\}$, and $\mu^s_{in} = (\text{Line 19}, \{\text{Line 19}\})$. The analysis also appends the call site (Line 21) to the call-string Γ, i.e. $\Gamma := \Gamma$. Line 21.

The next rule addresses the assignment instruction $v := e$ involving RAW (Read-After-Write), WAR (Write-After-Read), and WAW (Writer-After-Write) dependencies. The analysis checks for any variable *var* read in expression e (i.e., $var \in \texttt{rd}(e)$) that aliases with variable w in the write set ($\mathbf{W}^s_{in}$) of the propagated *sync* statements at the program point s, thereby detecting a RAW dependency, represented as $var \odot w$. Here, $x \odot y$ indicates that x and y may refer to the same memory location (i.e. potential aliases). Further, for the variable v written in the assignment statement s, the analysis checks for WAR and WAW dependencies by examining the alias relationship between v and the variables in the set $\mathbf{R}^s_{in} \cup \mathbf{W}^s_{in}$. If a dependency exists, it creates a mapping from the propagated *sync* statements in μ^s to s, marking s as the target. This indicates the end of a safe region $\mathcal{R}$ for a *sync* statement and, thus, an optimization opportunity for hetero-sync motion optimization has been found. Note that the analysis terminates the safe regions for all the *sync* calls that reach s because after the code transformation, even if a single *sync* relocates to s, it becomes a blocking call for the host, and hence it would not be beneficial to extend the safe region of other propagated synchronization calls. Therefore, the analysis reinitializes the data structures $\mathbf{R}^s$, $\mathbf{W}^s$, and Σ^s to identify new optimization opportunities in the unanalyzed code.

▶ **Example.** In Figure 7, the assignment statement at Line 8 reads from array `arr_output_gpu`, which was previously written at Line 19. Due to this RAW data dependence, the analysis updates the target set for Line 19 in map μ to {Line 8}, i.e., $\mu[\text{Line 19}] := \{\text{Line 8}\}$, followed by the re-initialization of other data-flow facts ($\mathbf{R}$, $\mathbf{W}$, and Σ).

The fourth rule addresses device function calls. Since *sync* statements cannot be percolated into device functions, there is no need to analyze the code in these functions. This rule checks the data dependency between the function arguments passed at the calling context and the data-flow facts propagated to that program point. The effect of this rule is similar to that of an assignment statement (discussed above).

For all other program statements, the analysis directly copies the data-flow facts for propagation, i.e., $\mathbf{R}^{s}_{out} := \mathbf{R}^{s}_{in}$ (Similarly for $\mathbf{W}$, Σ, and μ). Upon completing the analysis of a function f, the framework consolidates the data-flow facts from each **return** statement s_r of f and propagates them to each successor *succ* of the corresponding call site in the caller function (found in the call-string Γ). That is, $\mathbf{R}^{succ}_{in} := \mathbf{R}^{succ}_{in} \cup \mathbf{R}^{s_r}_{out}$ (Similarly for $\mathbf{W}$, Σ, and μ). The call site is then removed from the call-string Γ.

▶ **Example.** The data flow facts corresponding to the `return` statement (Line 12, Fig. 7) in function `foo` are copied to the successor (Line 22) of the call site (of function `foo`).

Our analysis handles branches and loops by performing join operations on data-flow facts at control-flow merge points. The confluence operator of the analysis is a union, meaning the data-flow facts at the entry of a basic block (*in* sets) n in a CFG are the union of the *out* sets of all predecessors of n.

3.2 Updating Target Location Sets

After the analysis in the previous step terminates, GSOHC retrieves the target location set for each *sync* statement (from map μ^s at the main function's exit statement s) and provides this information in map $\mathcal{M}$ as input to the current step. The target location sets computed in the previous step respect data dependencies but do not account for control-flow dependencies. For example, consider the *sync* call at Line 19 in Figure 7. The analysis in the previous step identifies the data dependency between the write to `arr_output_gpu` in `cudaMemcpy` call at Line 19 and the read of the same variable at Line 8 (in function `foo`, called at Line 21). Hence, it terminates the safe region at Line 8 and maps this as a target location for the *sync* call at Line 19. However, relocating `cudaMemcpy` synchronization call from Line 19 to Line 8 would make its execution conditional on the value of `pred`, unlike in the original program where it executes on all paths. Thus, this step validates the target mappings $\mathcal{M}$ reported by the previous step and updates invalid target locations with valid ones by checking control-flow dependency through Algorithm 1.

For each *sync* statement *src* (source) and its corresponding target location *dst* (destination), Algorithm 1 validates if *src* dominates *dst* and *dst* post-dominates *src*, establishing a *dom-pdom* relationship. If validation fails, the algorithm iteratively identifies the farthest successor s_t from *src* that satisfies the dom-pdom relationship, updating s_t as the target location and thus establishing a refined safe region (see Definition 7) from *src* to s_t. We propose a context-sensitive, demand-driven algorithm for determining inter-procedural dom-pdom relationships between source and destination instructions. Our approach utilizes the transitivity property of dominance (if s_a dominates s_b and s_b dominates s_c, then s_a dominates s_c) and extends LLVM's intra-procedural dom-pdom tree analysis [34] to handle inter-procedural relationships. Further, unlike previous work [18] that exhaustively computes dominance relationships for all node pairs in the Inter-procedural Control Flow Graph (ICFG), our targeted analysis efficiently focuses only on the specific dom-pdom relationships required for our framework.

To present our algorithm, we extend the source and destination call-strings, Γ_{src} and Γ_{dst}, by appending the source instruction and destination instruction, respectively. We refer to this augmented data structure as *contextual path* $\mathbb{K}$. For example, the source and destination contextual paths are $\mathbb{K}_{src} = \langle sc_1,\ sc_2,\ \ldots,\ sc_n \rangle$, and $\mathbb{K}_{dst} = \langle dt_1,\ dt_2,\ \ldots,\ dt_m \rangle$, where each sc_i or dt_i is a function call, and sc_n and dt_m are the *src* and *dst* instructions, respectively.

Algorithm 1 Updation of Target Locations.

Input: source src, target dst, call-strings Γ_{src}, Γ_{dst}
Output: updated target statement s_t

```
1  Procedure updateTarget
2      K_src, K_dst ← Γ_src.src, Γ_dst.dst
3      n, m ← |K_src|, |K_dst|
4      com_i = max_{1≤i≤min(n,m)}{enclosingFn(sc_i) = enclosingFn(dt_i)}
5      s_extd ← exitDominance(K_src, com_i, n)
6      if s_extd ≠ K_src[com_i] then
7          s_r ← exitNode(enclosingFn(s_extd))
8          s_t ← validateRelocation(s_extd, s_r)
9          return s_t
10     if K_src[com_i] dom K_dst[com_i] ∧ K_dst[com_i] pdom K_src[com_i] = False then
11         s^s_com ← K_src[com_i]
12         s^d_com ← K_dst[com_i]
13         s_t ← validateRelocation(s^s_com, s^d_com)
14         return s_t
15     s_entd ← entryDominance(K_dst, com_i, m)
16     s_e ← entryNode(enclosingFn(s_entd))
17     if s_entd pdom s_e = False then
18         s_t ← validateRelocation(s_e, s_entd) return s_t
19     return dst
```

To determine inter-procedural dom-pdom relationships, we require a reference point within the contextual paths of both the source and destination instructions. We call this reference point the Nearest Common Caller (NCC) function, which is the function where the backward paths from src and dst in their respective contextual paths first meet. We refer to the index in a contextual path corresponding to the NCC function as the NCC index. The NCC index com_i is the maximum index where the functions in the contextual paths $\mathbb{K}_{src}$ and $\mathbb{K}_{dst}$ are same (formally defined at Line 4, Algorithm 1). Using the NCC function as a reference point, the algorithm effectively models the source and destination as if they were inlined within this common context. For instance, in Figure 8a, function $\texttt{f}_1$ is the nearest common function to both src and dst, thereby establishing it as the NCC function. Similarly, in Figures 8b, 8c, and 8d, functions `foo`, `foo`, and `main` are the respective NCC functions.

Algorithm 1 transitively establishes dom-pdom relationships across the entry and exit points of the functions involved in the paths from src to dst that pass via the NCC function, ensuring that src instruction can be safely propagated across function boundaries. This task can be divided into three phases, each addressing dominance at critical points in the paths: (1) Exit Dominance (Lines 5 - 9), (2) Common Dominance (Lines 10 - 14), and (3) Entry Dominance (Lines 15 - 19). In the ICFGs shown in Figure 8, the paths for Exit Dominance, Common Dominance, and Entry Dominance are highlighted in green, red, and blue, respectively. Notice that, together, these paths form a path from src to dst through the NCC function.

Exit Dominance. First, we verify if src ($\mathbb{K}_{src}[n]$) dominates the exit node of its enclosing function f_{src}, ensuring that the refined safe region for src can be extended to the call site invoking f_{src} in the caller function. The `exitDominance` function (Line 1, Algorithm 2)

Algorithm 2 `exitDominance` and `entryDominance`.

```
1  Function exitDominance(𝕂_s, index, n)
2      for i ← n to index + 1 by 1 do
3          s_r ← exitNode(enclosingFn(𝕂_s[i]))
4          if 𝕂_s[i] dom s_r = False then
5              return 𝕂_s[i]
6      return 𝕂_s[index]
7  Function entryDominance(𝕂_d, index, m)
8      for i ← index + 1 to m by 1 do
9          s_e ← entryNode(enclosingFn(𝕂_d[i]))
10         if 𝕂_d[i] pdom s_e = False then
11             return 𝕂_d[i]
12     return 𝕂_d[n]
```

iterates backwards from the last statement in $\mathbb{K}_{src}$, checking if each statement in the contextual path dominates the exit node of its enclosing function. If all these dominances hold, `exitDominance` returns the instruction at the NCC index in $\mathbb{K}_{src}$, establishing a refined safe region for *src* up to $\mathbb{K}_{src}[com_i]$. For example, in Figure 8c, the source function `bar` is invoked by `fun`, which is further called by the destination function `foo`, making `foo` the NCC function. The `exitDominance` function validates the dominance relationship between the source node `S` and the exit node `T` of `bar`, and then between node `Q` and node `U`. If all dominances hold, `exitDominance` returns the instruction at the NCC index (node `O`), indicating that the refined safe region can be established from node `S` to node `O`. (Similarly, in Figures 8a and 8b if nodes `D` and `L` dominate nodes `E` and node `M`, respectively, safe regions are established up to NCC indices `B` and `J`.) If a statement at index i fails to dominate the exit node of its enclosing function f, `validateRelocation` (Algorithm 3) finds a valid target location within f by iterating from $\mathbb{K}_{src}[i]$ to the farthest successor that satisfies the dom-pdom relationship. In Figure 8c, if node `Q` does not dominate node `U`, `validateRelocation` finds a valid target location along the path from `Q` to `U`. Note that if *src* is within the NCC function, i.e., $\mathbb{K}_{src}[com_i] = src$ as shown in Figure 8d, the `exitDominance` function has no effect, as it directly returns $\mathbb{K}_{src}[com_i]$, the instruction at the NCC index, without iterating through the loop. The algorithm proceeds to the Common Dominance phase only when a refined safe region for *src* is established up to the instruction at the NCC index ($\mathbb{K}_{src}[com_i]$).

Common Dominance. This phase verifies the dom-pdom relationship between the instructions $\mathbb{K}_{src}[com_i]$ and $\mathbb{K}_{dst}[com_i]$ at the NCC index to confirm whether the refined safe region for *src* can extend from $\mathbb{K}_{src}[com_i]$ to $\mathbb{K}_{dst}[com_i]$. If validation fails, the algorithm invokes the `validateRelocation` function to find a valid target location on the path from $\mathbb{K}_{src}[com_i]$ to $\mathbb{K}_{dst}[com_i]$ within the NCC function. If successful, the algorithm proceeds to the Entry Dominance phase. For example, in Figure 8a, if nodes `B` and `F` satisfy the dom-pdom relationship, the refined safe region for *src* can be extended into function $\mathtt{f}_3$. Similarly, in Figure 8d, if nodes `W` and `X` satisfy the dom-pdom relationship, the refined safe region extends into `foo` through the Entry Dominance phase.

Algorithm 3 Validation of Target Location Set.

```
1  Function validateRelocation(stmt s_n, stmt s_i)
2      worklist ← {s_n}; s_c ← s_i
3      repeat
4          b ← first(worklist)
5          worklist ← worklist − {b}
6          if b = s_i then
7              break
8          if s_n dom b ∧ b pdom s_n then
9              s_c ← b
10         foreach succ ∈ successor(b) do
11             if succ is not visited then
12                 worklist ← worklist ∪ {succ}
13     until worklist = ∅;
14     return s_c
```

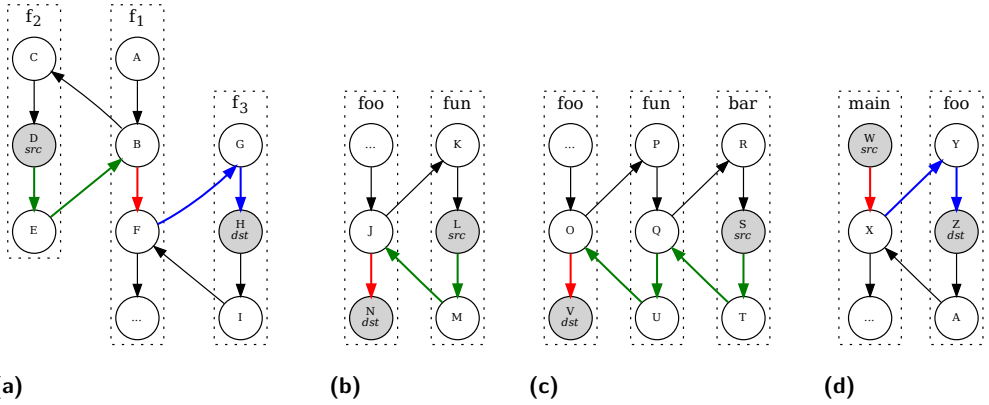


Figure 8 Examples for Exit Dominance, Common Dominance and Entry Dominance phase.

Entry Dominance. This phase verifies if an instruction s_d inside a callee function f_d (of the NCC function) post-dominates the entry node of f_d such that the refined safe region for *src* can be extended up to s_d. The `entryDominance` function (see Line 7, Algorithm 2) iterates through the destination contextual path $\mathbb{K}_{dst}$ to validate if each statement in the contextual path post-dominates the entry node of its enclosing function. If all these post-dominances hold, the algorithm establishes a refined safe region from *src* up to *dst*. For example, in Figure 8a, the refined safe region is extended till node `H` if `H` post-dominates the entry node `G`. Similarly, in Figure 8d, the refined safe region for *src* is established till node `Z` if `Z` post-dominates `Y`. If any statement at index i fails to post-dominate the entry node N of its enclosing function f, `validateRelocation` finds a valid target location within f by iterating from N to the farthest successor in f that satisfies the dom-pdom relationship. Note that, if *dst* resides within the NCC function, i.e., $\mathbb{K}_{dst}[com_i] = dst$, as shown in Figure 8b, `entryDominance` has no effect, as it simply returns $\mathbb{K}_{dst}[com_i]$, without iterating through the destination contextual path. Finally, `entryDominance` and Algorithm 1 terminate by returning the instruction up to which the refined safe region is established for the given *src*.

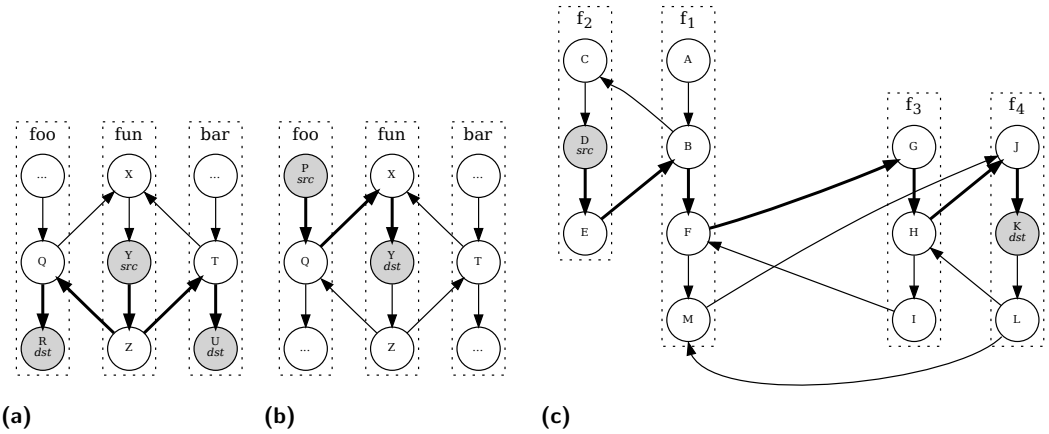


Figure 9 Locations of *src* and *dst* in inter-procedural hetero-sync motion, with paths from *src* to *dst* highlighted in bold.

▶ **Example.** In Figure 7, source *src* is Line 19 and target *dst* is Line 8, and the contextual paths are $\mathbb{K}_{src} = \langle$Line 19$\rangle$ and $\mathbb{K}_{dst} = \langle$Line 21, Line 8$\rangle$. Since, `enclosingFn`$(\mathbb{K}_{src}[1]) =$ `enclosingFn`$(\mathbb{K}_{dst}[1]) =$ `main`, the function `main` becomes the NCC function. Here, the Exit Dominance phase has no effect as *src* belongs to the NCC function (Line 19). In the Common Dominance phase, Line 19 and Line 21 satisfy the dom-pdom relationship, confirming that refined safe region can be extended into `foo`. In the Entry Dominance phase, *dst* (Line 8), however, does not post-dominate the entry node of `foo`. Hence, `validateRelocation` identifies Line 5 as the farthest successor from the entry node, which satisfies the dom-pdom relationship, establishing the refined safe region as $\mathcal{K}$(Line 19, Line 5) and setting Line 5 as the new destination.

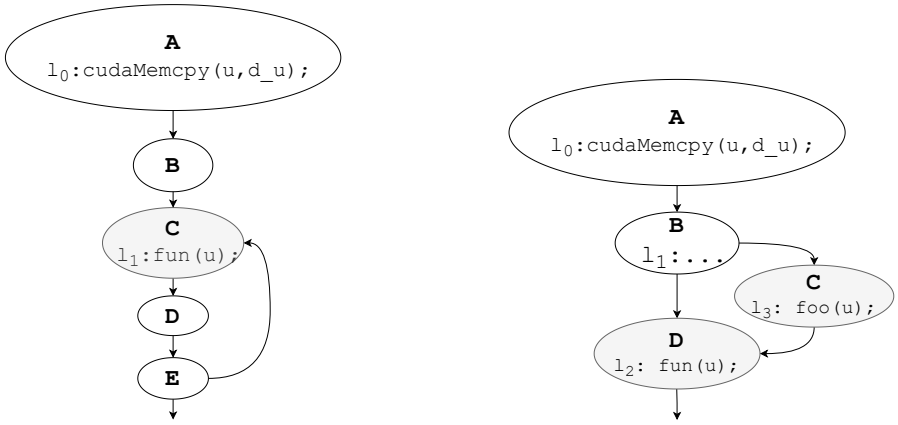
Updating Target Locations Across Contexts. After identifying a *dst* for an *src* in a specific context by means of a refined safe region, we verify its safety across different contexts. For a *src* in a method m, the *dst* location can fall into one of the three cases (see Figure 9): (1) *dst* inside a caller of m (2) *dst* inside a callee of m, and (3) *dst* inside a function f where m and f are called by NCC function.

The first case is safe because each invocation of the source function in different contexts maps *src* to a specific destination(s), such that the effect of percolation remains valid across all contexts when percolated to the respective caller functions. The example in Figure 9a demonstrates this with source function (`fun`) being invoked from two contexts (node `Q` in `foo` and node `T` in `bar`). The source instruction at context `Q` has a destination at node `R`, while at context `T`, the destination is `U`. Thus, percolating *src* to the caller functions ensures its execution along both paths: `Q` → `X` → `Y` → `Z` → `Q` → `R`, and `T` → `X` → `Y` → `Z` → `T` → `U` (as in the original program before optimization).

In the second case, relocating *src* to a callee can be unsafe in some cases. For example, in Figure 9b, `fun` is called from two different contexts in functions `foo` and `bar`. Percolating *src* (the *sync* statement) from `P` (in `foo`) to `Y` (in `fun`) enables computation in node `X` to run in parallel with kernels launched above `P` along the path `P` → `Q` → `X` → `Y` → `Z` → `Q` but introduces unintended memory transfers along path `T` → `X` → `Y` → `Z` → `T`, causing correctness issues. Hence, we percolate *sync* from *src* to the farthest node, whose enclosing function is invoked from a single callsite. In Figure 9b node `Q` is the farthest location from *src* (node `P`), with the enclosing function (`foo`) invoked from a single call site.

Algorithm 4 Updating Targets based on Context.

Input: $\mathbb{K}_{dst}$, com_i
Output: s_t
1 **Procedure** *updateTargets*
2 $n \leftarrow |\mathbb{K}_{dst}|$
3 **foreach** $j \leftarrow com_i$ **to** $n-1$ **by** 1 **do**
4 $s_{call} \leftarrow \mathbb{K}_{dst}[j]$
5 $f \leftarrow$ `getCallee`(s_{call})
6 $s_t \leftarrow s_{call}$
7 **if** *numCallSites*$(f) > 1$ **then**
8 **break**



(a) (b)

Figure 10 Example of two CFGs showing the necessity of loop-based updation and unification.

In the third scenario, where *src* and *dst* reside in functions that lack a direct caller-callee relationship, direct percolation from *src* to *dst* can be unsafe in some cases. For instance, in Figure 9c where function $\texttt{f}_1$ invokes functions $\texttt{f}_2$, $\texttt{f}_3$, and $\texttt{f}_4$ at nodes `B`, `F`, and `M`, respectively, and $\texttt{f}_3$ further invokes $\texttt{f}_4$ at node `H`. The *sync* at `D` in $\texttt{f}_2$ cannot be directly percolated to `K` in $\texttt{f}_4$ due to invocations of $\texttt{f}_4$ from different callsites. Like in the previous case, we percolate *sync* to the farthest node from *src* (node `D`), which is `H` in Figure 9c, where the enclosing function ($\texttt{f}_3$) is called from a single callsite.

Algorithm 4 updates target statements using contextual information by iterating over the destination contextual path from the NCC index com_i and checking whether the callee function at each index is invoked from a single callsite. If a callee at a callsite s_{call} is invoked from multiple callsites, the algorithm updates the target to s_{call} (Line 6), preventing the percolation of *sync* statements to such callee functions. For example, in Figure 9c, with NCC function $\texttt{f}_1$ and destination contextual path $\langle$`F`, `H`, `K`$\rangle$, the algorithm updates the destination to node `H` as the function $\texttt{f}_3$, called at callsite `F`, is invoked from a single callsite, whereas function $\texttt{f}_4$, called at `H`, is invoked from two distinct callsites (nodes `H` and `M`).

Updating Target Locations based on Loops. We now discuss another kind of updating process with an example. Suppose, in Figure 10a, the analysis identifies l_1 in basic block `C` as the optimal target location for the synchronization statement `cudaMemcpy` at statement l_0

```
1  void foo(float *arr_output_gpu)
2  {
3      float *arr_gpu, int size;
4      ...
5      cuda_kernel<<< grid,block >>>(..., arr_gpu); /* Device-side computation */
6      cudaMemcpy(arr_output_gpu, arr_gpu, size, cudaMemcpyDeviceToHost); /* barrier */
7  }
8  void bar(float *arr_output_gpu)
9  {
10     ...
11     for(int i=0; i<n; i++)
12         sum+=arr_output_gpu[i];
13 }
14 void main(){
15
16     float *arr_out_gpu, *cpu_arr;
17     foo(arr_out_gpu);
18     cpu_func(cpu_arr);
19     bar(arr_out_gpu);
20     ...
21 }
```

Figure 11 Sample code showing the challenges in automated transformation.

in basic block `A` (due to data dependency on `u`). While `A` *dominates* `C`, and `C` *post-dominates* `A`, relocating *sync* from l_0 to l_1 may incur performance overhead as `C` is a loop-header. A loop-header dominates its loop and is the target of a back edge forming the loop. The statement l_0, which gets executed once when it resides in `A`, would get executed in multiple iterations after optimization if placed in `C`. Thus, we mark l_0 to move to `B` since it is the farthest successor from the source (`A`) with the same loop scope as `A`.

At the end of this phase, the *sync* instructions and their updated target locations are stored in the map $\mathcal{M}$. This map is then passed to the next phase for further processing.

3.3 Unifying Target Location Sets

After the previous two steps, $\mathcal{M}$ may still map a single *src* (*sync* statement) to multiple destination locations. For example, in Figure 10b, the *sync* statement l_0 in basic block `A` has dependent statements l_3 in `C` and l_2 in `D`, both using variable `u`. While the first step identifies a set of target locations $\{l_2, l_3\}$, the second step refines it to $\{l_2, l_1\}$, replacing l_3 with l_1 in basic block `B` because l_1 post-dominates l_0, unlike l_3. However, since both l_1 and l_2 are on the same execution path, relocating *sync* to both l_1 and l_2 can cause redundant memory transfers, degrading performance. We address this issue in the current phase of our framework by unifying all the target locations that are found and updated by the previous two steps into a single target location. Here, we again use the inter-procedural dominance relationship to solve the problem. Among all the target location statements given by the previous step, GSOHC automatically selects the statement that dominates all the other statements in them. In Figure 10b, since basic block `B` dominates basic block `D`, the final set of target locations will contain only statement l_1. That is, GSOHC outputs a unique target location for a given *sync* statement in the form of a unified safe region (see Definition 8) for performing hetero-sync motion.

4 Code Transformation for Hetero-Sync Motion

The transformation pass takes the map $\mathcal{M}$ and a source program in IR format as inputs and outputs an optimized IR (see Figure 5). The map $\mathcal{M}$ associates a unique target location for a *sync* statement in a given context. When the target location is within the same

function as *sync*, the *sync* can be moved directly before the target statement. However, if the source and target locations are in different functions, direct percolation of the *sync* is infeasible due to the unavailability of argument variables in the target scope. For example, in Figure 11, the *sync* statement at Line 6 in function `foo` blocks potential overlap between kernel computation (Line 5) and host-side function (Line 18). According to our analysis step, the correct location for *sync* to achieve maximum computation overlap is Line 10. Since arguments for the *sync* call are defined locally in `foo` (Line 3), relocation requires transforming the IR to preserve these definitions across functions.

To address percolations across functions, GSOHC transforms the LLVM IR based on three different cases. Firstly, where the source and target functions lack a direct callee-caller relationship, *sync* argument variable definitions are first relocated to the NCC function (see Section 3), added to formal and actual argument lists of the source and target functions and then, the *sync* statement is percolated to the target instruction. In Figure 11, variables `arr_gpu` and `size`, defined at Line 3, are shifted from `foo` to the NCC function `main`, appended to parameter lists, appended to the actual arguments at call sites (Lines 17 and 19) and the *sync* is subsequently moved to the target location. Secondly, if the target is in the caller of the source function, the caller (target function) becomes the NCC function. Therefore, unlike the first case, the *sync* arguments do not need to be added to the formal and actual argument lists of the target function before percolating the *sync*. Thirdly, if the *sync* is in the caller of the target function, the caller (source function) becomes the NCC function. Similar to the second case, here, the argument variables of *sync* do not need to be added to the source function's formal and actual argument lists. Figure 7 demonstrates an example where hetero-sync motion requires transformation detailed in the third case, where the argument variables of *sync* (`arr_gpu` and `size`) at Line 19 are added as formal and actual arguments of the target function `foo`, prior to percolating the *sync* to Line 5.

5 Evaluation

This section investigates the following research questions to evaluate our hetero-sync motion optimization and GSOHC's capability in identifying and performing this optimization.

RQ1. Can GSOHC precisely identify opportunities for hetero-sync motion?

RQ2. How efficient are GSOHC's analysis and transformation passes?

RQ3. What is the impact of hetero-sync motion on the execution time of input benchmarks?

Implementation. GSOHC consists of two primary components: (1) inter-procedural flow- and context-sensitive static analysis to identify hetero-sync motion opportunities, (2) automated code transformation to perform the optimization. We have implemented our analysis and transformation passes using LLVM/Clang-14.x [36, 31] infrastructure. An open-source GPGPU compiler, `gpucc` [54] embedded with LLVM/Clang-14.x, converts CUDA programs into LLVM IR. Our custom analysis pass works on the SSA (Static Single Assignment) representation obtained using `-mem2reg` flag (which promotes memory references into register references, resulting in a "pruned" SSA). In addition, we use LLVM's intra-procedural Dominator and Post Dominator Tree [34], and Loop Info wrapper [35] passes.

Experimental Setup. We run our experiments on NVIDIA RTX A4000 (Ampere), NVIDIA Tesla P100 (Pascal) and NVIDIA A100 (Ampere) GPUs. Table 1 details the system specifications. This multi-generational GPU selection validates the robustness of our framework, enables analysis of performance variations due to communication and computation differences, and provides insights into its scalability across architectures.

Table 1 System configuration.

GPU	NVIDIA RTX A4000 (16 GB)	NVIDIA P100 (16 GB)	NVIDIA A100 (84 GB)
CPU	64 × AMD EPYC CPU @ 2.20GHz	20 × Intel(R) Xeon(R) Silver CPU @ 2.20GHz	64 × Intel(R) Xeon(R) Gold CPU @ 3.50 GHz
OS	Ubuntu 22.04 LTS	Debian GNU/Linux 11	Ubuntu 22.04 LTS
CPU RAM	128 GB	189 GB	84 GB

Benchmarks. To rigorously evaluate GSOHC with respect to diverse real-world scenarios, we use benchmark programs from benchmark suites: PolyBench [21], Rodinia [10], CUDA Sample SDK [44], and HeCBench [25]. Although (many) host-side computations in these benchmarks verify the results produced by GPU processors, they also exhibit complex control flows, encompassing nested loops, branches and aliasing behaviours, and offer rich inter-procedural opportunities replicating real-world challenges. For example, in the `entropy` program [57], a host-side function computes entropy for each matrix element using a 5×5 window with three nested loops, performing local histogram computation and probability evaluation using a logarithmic operation. Section 6 provides an in-depth analysis of CPU computations and various optimization opportunities in real-world programs. PolyBench, Rodinia and CUDA Sample SDK are widely-used GPU benchmark suites used in recent research works [2, 1, 26], while HeCBench is an extensive collection of heterogeneous computing benchmarks written in CUDA [9], HIP [5], SYCL [29], and OpenMP [14] across domains like physics, linear algebra, and machine learning (sourced from other standard suites such as Rodinia [10], PolyBench [21], N-body simulation [48], Hetero Mark [52], CUDA Sample SDK [44], etc.).

From these benchmark suites, we identified 56 benchmark programs (listed in Table 2) that contain opportunities for our hetero-sync motion optimization, excluding those without kernel or `global_synchronization` calls or with fewer device-side or host-side computations. Note that our framework can also analyze programs that have no hetero-sync motion opportunities and correctly report their absence, ensuring no false positives.

5.1 RQ1: Hetero-Sync Motion in Practice

We found 361 optimization opportunities from 56 benchmark programs for performing hetero-sync motion.

The results of our static analysis are shown in Table 2. The second column lists the number of lines in the LLVM IR file corresponding to each benchmark program. In the third column, O_f represents the number of manually found optimization opportunities. Our static analysis successfully identified all the optimization opportunities in the benchmarks, resulting in zero false negatives. For example, in the `xlqc` [58] program (from HecBench), the code computes Coulomb and exchange matrices for quantum chemistry simulations by first calculating their primitive forms on the GPU and transferring the results to host memory. On the CPU, two 2D contracted matrices are initialized, and GPU-computed primitive matrices are aggregated into these contracted matrices, which are then stored in GNU Scientific Library (GSL) data structures. GSOHC identifies an opportunity for hetero-sync motion here, where the performance can be enhanced by overlapping GPU computations with CPU initialization of the contracted matrices.

We have observed that many of our benchmark programs (around 48%) have synchronization barriers spanned across different functions. Using an intra-procedural analysis would have caused to miss many opportunities in this scenario. Moreover, optimization opportunities that span across different functions are much harder to identify manually (see Section 1). To address this, we designed an inter-procedural, flow- and context-sensitive static analysis framework that enables practical and effective hetero-sync motion.

Table 2 Benchmark statistics and the results of GSOHC's static analysis.

suite	Benchmark	LOC	O_f	T_{an}
PolyBench	correlation	1215	6	24
	covariance	885	5	20
	2mm	983	11	26
	3mm	1208	9	34
	atax	702	6	20
	bicg	775	8	47
	doitgen	863	5	19
	gemm	653	5	18
	gemver	1127	11	39
	gesummv	621	2	10
	mvt	732	8	27
	syr2k	668	5	18
	syrk	597	4	14
	gramschmidt	986	5	18
	lu	654	3	11
	adi	1630	7	43
	2DConvolution	636	4	14
	3DConvolution	916	4	16
	fdtd-2d	1015	6	22
	jacobi-1D	612	5	26
	jacobi-2D	763	5	29
	adam	1214	7	8
	affine	1419	5	6
	atan2	1993	11	15
	bilateral	1434	9	14
	bitonic-sort	604	3	5
	burger	1523	4	27
	car	1084	8	8

suite	Benchmark	LOC	O_f	T_{an}
HeCBench	clink	1147	6	12
	channelsum	9202	6	57
	chemv	698	6	7
	chi2	1054	5	5
	conv Sep	241	3	5
	dp	317	6	3
	entropy	871	8	9
	extrema	4635	40	63
	floyd-warshall	571	5	8
	keogh	637	9	9
	langevin	666	7	10
	lif	695	5	8
	lombscargle	938	7	6
	maxpool3D	464	3	6
	mrc	923	11	11
	mt	971	2	6
	nw	1033	3	8
	pool	933	3	9
	rainflow	1028	3	6
	rtm8	1300	3	9
	s8n	1717	7	7
	scel	677	5	8
	shuffle	1144	22	291
	snake	1679	2	11
	softmax	469	3	5
	stddev	509	3	4
	tissue	864	2	11
	xlqc	17848	9	253

Further, we thoroughly tested our analysis and transformation phases. For each benchmark program, we recorded the source and corresponding target location sets at each phase of our static analysis and manually verified them. We ran each program five times before and after transformation with identical inputs, confirming identical outputs. We also manually inspected the transformed IRs to ensure correctness of the transformation pass.

5.2 RQ2: Efficiency of GSOHC

This section presents the time taken by our analysis and transformation passes for the benchmark programs. In Table 2, the last column (T_{an}) shows the analysis time in milliseconds. Our analysis time ranges from a few milliseconds to 291 milliseconds (measured on the CPU node associated with the RTX A4000 GPU, see Table 1). Further, our transformation pass takes only a few microseconds, requiring three steps: (1) adding variables to function argument lists and callsite modifications, (2) declaring variables in the common caller, and (3) percolating *sync* to the target location.

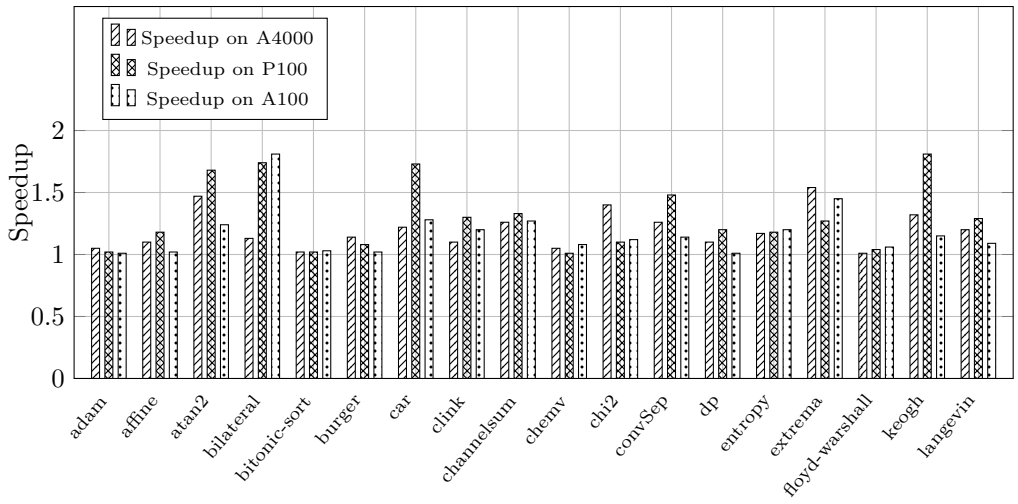
Although the benchmark programs have similar lines of code, the analysis time for the `shuffle` benchmark is significantly higher due to its greater number of hetero-sync motion opportunities and more complex control flow (with more loops, branches, and function calls). On the other hand, though benchmark `extrema` has 40 optimization opportunities, the control flow in it is much simpler (1D loops, fewer function calls and fewer branches with no nesting), resulting in a shorter analysis time. The data flow analysis may require multiple iterations over the loop body, with new flow information in each iteration, taking longer to converge to a fixed point. Therefore, the analysis time is more influenced by the number and location of opportunities rather than program size.

5.3 RQ3: Effectiveness of Hetero-Sync Motion

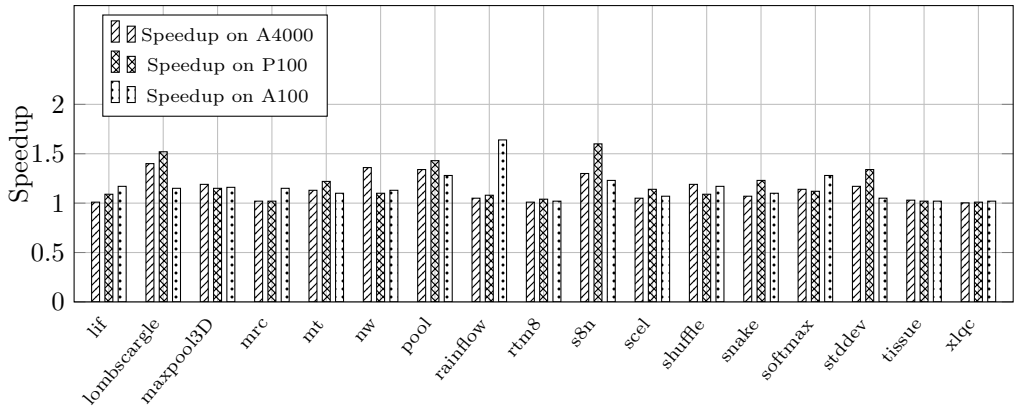
Figure 12a, 12b and 12c show the performance improvements achieved by hetero-sync motion on different benchmarks run on RTX A4000, P100 and A100 machines. We executed each benchmark program five times before and after optimization and then reported their arithmetic mean execution times. Our hetero-sync motion optimization achieves substantial performance gains on all machines, with geomean speedups of approximately 1.15x, 1.17x, and 1.13x with standard deviations of 0.19, 0.20 and 0.15 (and up to 1.8x, 1.9x and 1.9x speedups on A4000, P100 and A100, respectively) and a geometric mean performance improvement of around **11**%, reaching up to **34**%.

In many cases, the reduction in total execution time is also significant. Specifically, 36 benchmarks demonstrated performance improvements exceeding 500 milliseconds on the P100 machine, with 29 of these benchmarks showing improvements of over 1 second (11 benchmarks having improvements over 10 seconds). Given that the total running times of these programs are only a few seconds, hetero-sync motion optimization is notably impactful. The benchmarks with higher execution times also have higher improvements in execution time. As detailed in Figure 12b, the most pronounced improvement was observed in the `snake` benchmark from the HeCbench suite, where the execution time was reduced from 368.5 seconds to 230.6 seconds, yielding a 1.6x speedup and an overall reduction of 137.9 seconds. Further, the transformed code results in the best performance if T_{host}, the execution time of identified host-side computations, and T_{dev}, the execution time of identified device-side computations, are significant and are comparable to one another such that they can hide each other's latency. Thus, benchmarks such as `extrema`, `bilateral`, and `atan2` with significant overlaps show approximately 1.5x to 1.7x speedup with the optimization.

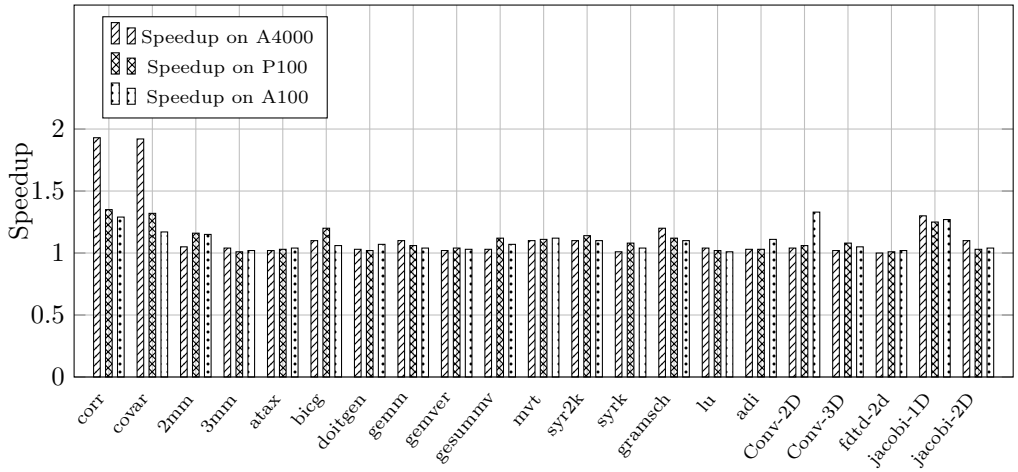
On the other hand, if T_{host} highly dominates T_{dev} or vice versa, the optimized code is constrained by the dominated time and does not result in significant performance gains. Thus the performance gains of less than 1.06x in specific benchmarks can be attributed to an imbalanced load distribution between CPU and GPU computations. For example, in `3mm`, `gemm` (from Polybench), and `bitonic-sort`, `lif` (from HeCBench), the CPU-side computations take significantly longer time than device-side computations. Thus, the optimized code takes almost as long as the CPU-side computation, resulting in smaller speedups. Further, we describe an interesting case in `floyd-warshall` benchmark program where it does not exhibit performance improvement post-optimization. In this case, the synchronization statement is a `cudaMemcpy` call present outside of a loop. However, additional `cudaMemcpy` calls within the loop, occurring before each kernel invocation, cause the kernel to block at each iteration. Since these synchronization calls are not loop-invariants, they cannot be moved outside the loop. Furthermore, relocating the *sync* statement from outside the loop body to a program point beyond the CPU computation may result in minimal overlap between host and device computation (particularly in the final iteration of the loop), yielding negligible speedup.



(a) Speedup achieved by hetero-sync motion on HeCBench.



(b) Speedup achieved by hetero-sync motion on HeCBench (contd.).



(c) Speedup achieved by hetero-sync motion on PolyBench.

Figure 12 Speedup achieved by hetero-sync motion on various benchmark programs.

```
1 void backward_pass(...){
2   ...
3   Matrix_Multiplication_GPU<<<...>>>
        (..., device_first, ...);
4   cudaMemcpy(first->gradient,
        device_first, ..., DtoH);
5
6   /* Compute the gradient for the
        second matrix */
7   for (int row = 0; row<first->nRows
        ; ++row) {
8    for (int column = 0; column<first
        ->nColumns; ++column) {
9     temp[column * first->nRows + row
        ]=first->value[row * first
        ->nColumns + column];}
10  }
11
12  cudaMemcpy(device_first, temp,
        ..., HtoD);
13  cudaMemcpy(device_second, second->
        gradient, ..., HtoD);
14
15  Matrix_Multiplication_GPU<<<...>>>
        (..., device_second, ...);
16
17  cudaMemcpy(..., device_second,
        ..., DtoH);
18 ...
19 }
```

(a) Code snippet from `GraphFlow` [22].

```
1 void divideWork(...){
2   if(value < size) {
3     for(int i = 0; i < value; ++i) {...}}
4   else {
5     int portion = value / size;
6     for(int i = 0; i < size; ++i) {
7       h_head[i] = i * portion;
8       h_tail[i] = (i + 1) * portion;}
9       h_tail[size - 1] = value;}
10  cudaMemcpy(d_head, h_head, ..., HtoD);
11  cudaMemcpy(d_tail, h_tail, ..., HtoD);
12 }
13
14 void main(){...
15  while(!(*h_complete)) {
16   divideWork(h_head, h_tail, d_head, d_tail,
         NBLOCKS, V);
17   assignColorsKernel<<<...>>> (d_head, ...);
18   cudaDeviceSynchronize();
19   *h_complete = true;
20   cudaMemcpy(d_complete, ..., HtoD);
21   divideWork(...);
22   detectConflictsKernel<<<...>>> (...,
         d_complete, ...);
23   cudaDeviceSynchronize();
24   cudaMemcpy(..., d_complete, ..., DtoH);
25   divideWork(...);
26   forbidColorsKernel<<<...>>> (d_head, ...);
27   cudaDeviceSynchronize();}
28  ...}
```

(b) Code snippet from `GCOL` [13].

Figure 13 Hetero-Sync Motion optimization opportunitites in real-world codes. For brevity, `cudaMemcpyDeviceToHost` is abbreviated as `DtoH` and `cudaMemcpyHostToDevice` as `HtoD`.

6 Case Study

Building upon the evaluation in Section 5 that demonstrates GSOHC's effectiveness, this section explores the broader applicability of hetero-sync motion in real-world open-source applications. We have surveyed several top-rated CUDA/C++ projects on GitHub and highlighted relevant opportunities in Figure 13. We present these case studies separately from the broader benchmark evaluation to provide a focused analysis of opportunities identified in specific real-world applications and to highlight practical insights that may otherwise be obscured in larger-scale evaluations. Note that certain real-world opportunities are embedded within large frameworks (rather than stand-alone programs) and benchmark suites that lack the necessary testing environment, such as readily available datasets. As a result, even though our static analysis pass is capable of effectively identifying these opportunities, we could not include them in our empirical evaluation. We first present intra-procedural opportunities, then inter-procedural opportunities.

Intra-procedural Opportunities. `GraphFlow` [22] (see Figure 13a), a CUDA/C++ deep learning framework, facilitates symbolic and automatic differentiation, supports dynamic computation graphs, and provides GPU-accelerated tensor and matrix operations, along with implementations of various state-of-the-art graph neural networks. An opportunity for hetero-sync motion optimization exists here when a code segment (in `GraphFlow`) uses GPU-based matrix multiplication for gradient computation. This code initiates two kernel launches. The first (Line 3) computes the gradient for the first matrix and synchronously transfers (Line 4) the result back to the host, which is followed by a CPU computation (Line 9) transposing the first matrix into a temporary data structure, which is then transferred to the device

memory along with the second matrix. The second kernel launch (Line 15) then performs matrix multiplication, updating the gradient for the second matrix, which is subsequently copied back to the host. In this case, performance can be further improved by overlapping the first kernel's computation with the CPU transposition of the first matrix. However, this optimization is currently restricted by the synchronous host-device data transfer immediately after the first kernel launch in the original implementation.

Inter-procedural Opportunities. We now discuss the `GCOL` and `GCON` programs from ScoR [13] benchmark suite. Both are CUDA/C++ applications focused on GPU-accelerated graph colouring and graph connectivity algorithms, respectively. Because these two applications share similar programming patterns, we will only discuss `GCOL` (see Figure 13b) in detail. The program operates in three phases within a loop that continues until all colour assignments are conflict-free. In the first phase, a GPU kernel (Line 17) is launched to assign initial colours to the vertices, while in the second phase, the code launches another GPU kernel (Line 22) to verify whether no adjacent vertices share the same colour. Lastly, a third GPU kernel (Line 26) is invoked to forbid specific colours from being assigned to prevent conflicts. Between consecutive phases, a host-side function is called (at Line 21) to partition the graph data for the next GPU kernel computation. However, `cudaDeviceSynchronize` calls (Lines 18 and 23) and synchronous data transfer operations (Lines 20 and 24) after each GPU kernel prevent overlap between the CPU's data partitioning and GPU computations, limiting potential performance improvements. This points to a hetero-sync motion optimization opportunity, especially since the CPU computation for workload division is encapsulated in a separate host function (distinct from that of the source function having *sync* call), indicating the need for inter-procedural optimization.

We further discuss real-world opportunities found in a GPU-based recursive filtering application called `gpufilter` [3] (not shown). An opportunity for hetero-sync motion optimization exists in a code segment that invokes three different host-side functions to perform different Gaussian filtering on an input image. Each function first computes the coefficients and weight vectors, followed by the creation of transformation matrices, which include generating zero and identity matrices, transposed convolution matrices, and performing matrix-vector multiplications. These transformation matrices are transferred to the device memory, followed by multiple GPU kernel launches to apply the filtering operations. Finally, the filtered image is transferred back to the host memory. The synchronous data transfer from device to host limits the overlap between the GPU filtering process and the CPU computation of transformation matrices for subsequent filtering, indicating an area for hetero-sync motion optimization.

7 Related Work

Global Optimizations in Heterogeneous Computing. GSOHC relocates synchronization calls on the host side to maximize the overlap between host and device computations. In this context, we refer to global optimizations as those approaches that target host-side code but improve interactions between CPU and GPU in heterogeneous systems, leading to overall better performance. CGCM [23] improves program performance by managing CPU-GPU communication through compiler optimizations and runtime libraries, breaking cyclic communication patterns by transferring data to the GPU early and retrieving only when necessary. Kessler et al. [27] proposed a static-analysis technique to reduce the number of data transfer messages between CPU and GPU by analyzing dependence graphs over kernel

calls, reordering operand arrays in memory, and merging memory allocations of adjacent operands. Ashcraft et al. [4] developed a compiler technique for automatically scheduling data transfers from CPU to GPU and vice-versa based on data used within the kernel to reduce the number of bytes transferred, leading to performance improvement.

Apart from the above-mentioned static analysis-based approaches, architectural-based or run-time-based approaches also have been proposed to address global performance issues in heterogeneous systems. HUM [26] has provided a run time technique that leverages CUDA Unified Memory to overlap `H2Dmemcpy` time with host or kernel computation, resulting in an overall program speedup. Kim et al. [30] have proposed a software-hardware-based approach to improve the performance of workloads that have multiple dependent GPU kernels by automatically overlapping the execution of such kernels and exploiting implicit pipeline parallelism. As there have been many similar attempts to improve the performance of heterogeneous programs, we cite a survey paper [37] that summarizes the efforts of numerous research papers. Unlike GSOHC, which enhances program performance through hetero-sync motion increasing CPU-GPU compute overlap, these approaches achieve performance through other dimensions such as multi-kernel computation overlap, data transfer overlap, and communication optimizations.

Synchronization Optimization. In the field of compilers, there is a large body of work [41, 19, 39, 42, 16, 15, 17, 50, 49, 43, 56, 24, 40] on reducing synchronization overheads in the context of various programming paradigms, including Thread Level Parallelism (TLP) programs, Task Parallel programs, and MPI (Message Passing Interface) programs.

Hetero-Sync Motion is analogous to the synchronization optimization proposed by Nicolau et al. [42, 43] for efficient placement of `post` and `wait` methods in a `non-DOALL` loop increasing the degree of Thread Level Parallelism (TLP) in CPU programs. Because they focus on the percolation of synchronization calls with only blocking properties, they fail to consider the control-flow dependency of inserted `wait` calls resulting in duplicate `wait` calls (and code explosion). This can lead to correctness and performance issues in heterogeneous CPU-GPU systems because synchronization calls such as `cudaMemcpy` have both blocking and data transfer properties. In contrast to this approach, our proposed framework efficiently manages such scenarios with a dedicated static analysis.

Our work on hetero-sync motion optimization can be compared to works by Danalis et al., Nguyen et al. and Das et al. [16, 41, 17], that proposed code motion techniques to improve computation-communication overlap in the MPI programming model. However, these approaches require matching between communication and blocking calls, can percolate only a single blocking or non-blocking call at a time, and are limited in their ability to extend percolation beyond another `MPI` call. Since a GPU kernel call can have multiple synchronization calls surrounding it, and synchronization calls (by semantics) wait for all previous kernel calls to finish in CPU-GPU heterogeneous systems, GSOHC requires a data-flow analysis to facilitate the percolation of multiple synchronization calls together to an optimal program point. Furthermore, as in the case of TLP, `MPI_wait` calls are mere blocking calls, unlike the ones with data transfer in our case.

A number of past works have also proposed transformation techniques for addressing synchronization overheads and improving program performance. Sarkar et al. [50] provided an approximation algorithm to select an ordering of the fork operations and determine the placement of join operations in a given program, allowing independent computations to run simultaneously and thereby improving the overall parallelism. Zhu et al. [56] provided a code motion compiler optimization technique to reduce the communication overhead

caused by remote reads/writes (accessing dynamically allocated data structures) in parallel C programs. Their analysis determines the optimal communication mode (blocking or pipelined) and propagates remote writes in the program that may be subject to coalescing. Diniz et al. [19] aimed at redundant lock elimination to reduce lock overheads through the percolation of synchronization calls in a parallel program. Unfortunately, however, their approach may reduce the overall parallelism of a given program. Nandivada et al. [39] proposed optimizations such as `finish` elimination, loop chunking and `forall`-coarsening to reduce the number of synchronization operations in X10 programs. Nayak et al. [40] introduces ScopeAdvice, a tool designed to detect over-synchronization in CUDA programs, enabling performance optimization by analyzing memory access and synchronization traces using NVIDIA's NVBit library. The cuSync framework [24] provides fine-grained tile-level synchronization to address GPU underutilization caused by uneven tile distribution when the number of computation tiles is not a multiple of the GPU's execution units, enabling concurrent execution of independent tiles with user-defined synchronization policies. While we also address synchronization statements to improve performance, unlike these approaches that directly reduce synchronization overheads, our focus is on percolating synchronization calls to enhance CPU-GPU computation overlap in CUDA programs.

In summary, the existing works on synchronization optimization are inadequate to address the problem of hetero-sync motion due to the semantic differences in synchronization operations. To the best of our knowledge, our work is the first to focus on automatic synchronization motion optimization in CPU-GPU heterogeneous systems.

Static Analyses for Optimizating Device Code in Heterogeneous Computing. Recent efforts [51, 2, 1] focus on accelerating CUDA programs by improving GPU kernels (device-side computations) through static analysis and compiler optimization approaches. DARM [51] presents a static analysis-based framework that identifies and automatically merges divergent blocks in a CFG to reduce the effects of thread divergence in device computation. GPUDrano [1] automatically identifies uncoalesced memory accesses in a GPU kernel code using data flow analysis. GPU Block Independence Analysis [2] represents a static analysis framework that can automatically identify whether a GPU kernel code is independent of the block size, allowing the compiler to tune block size to improve performance. As GSOHC, they are all built on LLVM [31], a general framework for building compilers. These optimizations differ from ours, but they can be effectively combined with GSOHC to improve the performance of the input program further.

Load Balancing of Heterogeneous Workloads. The optimal performance due to hetero-sync motion is attained when the input workload to the source program is load-balanced across various architectures in a heterogeneous system. This can be attributed to our framework's ability to optimize the source code in a manner that allows/improves concurrent execution of host-side and device-side computations. The performance study conducted by White et al. [53] shows that their manually optimized CPU-GPU implementation with workload distributed between CPU and GPU results in better performance. Lee et al. [32] proposed a run time technique that enables the effective utilization of available CPU and GPU computing resources in a heterogenous system through a workload distribution module and a global scheduling queue. Khalid et al. proposed Troodon [28], a machine learning-based framework that automatically schedules jobs to CPU and GPU in a load-balanced way, increasing throughput with high device utilization.

8 Conclusion

We have developed a novel compiler analysis and optimization framework, GSOHC that automatically analyzes CPU-GPU heterogeneous programs and relocates poorly placed synchronization barriers within them. This optimization, which we named hetero-sync motion, facilitates overlapping CPU and GPU computations, thereby improving program performance. GSOHC consisting of three distinct phases employs an interprocedural context- and flow-sensitive data-flow analysis to accurately identify optimization opportunities in the input programs. We have also performed an evaluation of our framework on 56 benchmark programs drawn from several widely-recognized benchmark suites. Our evaluation shows that our static analysis can successfully identify all the actual optimization opportunities within these benchmark programs. Furthermore, our evaluation confirms significant performance improvements in the optimized code, underscoring the practical impact of our approach.

References

1 Rajeev Alur, Joseph Devietti, Omar S. Navarro Leija, and Nimit Singhania. Gpudrano: Detecting uncoalesced accesses in gpu programs. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 507–525, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-63387-9_25.

2 Rajeev Alur, Joseph Devietti, and Nimit Singhania. Block-size independence for gpu programs. In *International Static Analysis Symposium*, pages 107–126. Springer, 2018. doi:10.1007/978-3-319-99725-4_9.

3 Andmax. Github - gpufilter: Gpu recursive filtering. URL: https://github.com/andmax/gpufilter.git.

4 Matthew B Ashcraft, Alexander Lemon, David A Penry, and Quinn Snell. Compiler optimization of accelerator data transfers. *International Journal of Parallel Programming*, 47(1):39–58, 2019. doi:10.1007/S10766-017-0549-3.

5 Michal Babej and Pekka Jääskeläinen. Hipcl: Tool for porting cuda applications to advanced opencl platforms through hip. In *Proceedings of the International Workshop on OpenCL*, IWOCL '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3388333.3388641.

6 David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994. doi:10.1145/197405.197406.

7 Dip Sankar Banerjee and Kishore Kothapalli. Hybrid algorithms for list ranking and graph connected components. In *2011 18th International Conference on High Performance Computing*, pages 1–10, 2011. doi:10.1109/HiPC.2011.6152655.

8 Gautam Chakrabarti, Vinod Grover, Bastiaan Aarts, Xiangyun Kong, Manjunath Kudlur, Yuan Lin, Jaydeep Marathe, Mike Murphy, and Jian-Zhong Wang. Cuda: Compiling and optimizing for a gpu platform. *Procedia Computer Science*, 9:1910–1919, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012. doi:10.1016/j.procs.2012.04.209.

9 Gautam Chakrabarti, Vinod Grover, Bastiaan Aarts, Xiangyun Kong, Manjunath Kudlur, Yuan Lin, Jaydeep Marathe, Mike Murphy, and Jian-Zhong Wang. Cuda: Compiling and optimizing for a gpu platform. *Procedia Computer Science*, 9:1910–1919, 2012. doi:10.1016/J.PROCS.2012.04.209.

10 Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009. doi:10.1109/IISWC.2009.5306797.

11 Zhe Chen, Andrew Howe, Hugh T. Blair, and Jason Cong. Clink: Compact lstm inference kernel for energy efficient neurofeedback devices. In *Proceedings of the International Symposium on Low Power Electronics and Design*, ISLPED '18, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3218603.3218637.

12 Bruno Coutinho, Diogo Sampaio, Fernando Magno Quintao Pereira, and Wagner Meira Jr. Divergence analysis and optimizations. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 320–329, 2011. doi:10.1109/PACT.2011.63.

13 Csl-Iisc. Github - csl-iisc/scor: A (sco)ped (r)acey benchmark suite, containing cuda code for many different gpu benchmarks. URL: https://github.com/csl-iisc/ScoR.git.

14 L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998. doi:10.1109/99.660313.

15 A. Danalis, Ki-Yong Kim, L. Pollock, and M. Swany. Transformations to parallel codes for communication-computation overlap. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 58–58, 2005. doi:10.1109/SC.2005.75.

16 Anthony Danalis, Lori Pollock, Martin Swany, and John Cavazos. Mpi-aware compiler optimizations for improving communication-computation overlap. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 316–325, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1542275.1542321.

17 Dibyendu Das, Manish Gupta, Rajan Ravindran, W. Shivani, P. Sivakeshava, and Rishabh Uppal. Compiler-controlled extraction of computation-communication overlap in mpi applications. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2008. doi:10.1109/IPDPS.2008.4536193.

18 Bjorn De Sutter, Ludo Van Put, and Koen De Bosschere. A practical interprocedural dominance algorithm. *ACM Trans. Program. Lang. Syst.*, 29(4):19–es, August 2007. doi:10.1145/1255450.1255452.

19 Pedro Diniz and Martin Rinard. Synchronization transformations for parallel computing. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 187–200, New York, NY, USA, 1997. Association for Computing Machinery. doi:10.1145/263699.263718.

20 M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966. doi:10.1109/PROC.1966.5273.

21 Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10, 2012. doi:10.1109/InPar.2012.6339595.

22 HyTruongSon. Github - graphflow: Deep learning framework in c++/cuda that supports symbolic/automatic differentiation, dynamic computation graphs, tensor or matrix operations accelerated by gpu and implementations of various state-of-the-art graph neural networks and other machine learning models including covariant compositional networks for learning graphs [risi et al]. URL: https://github.com/HyTruongSon/GraphFlow.git.

23 Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic cpu-gpu communication management and optimization. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 142–151, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1993498.1993516.

24 Abhinav Jangda, Saeed Maleki, Maryam Mehri Dehnavi, Madan Musuvathi, and Olli Saarikivi. A framework for fine-grained synchronization of dependent gpu kernels. In *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '24, pages 93–105. IEEE Press, 2024. doi:10.1109/CGO57630.2024.10444873.

25 Zheming Jin. HeCBench. https://github.com/zjin-lcf/HeCBench.git, 2022. Retrieved: 2022-11-10.

26 Jaehoon Jung, Daeyoung Park, Youngdong Do, Jungho Park, and Jaejin Lee. Overlapping host-to-device copy and computation using hidden unified memory. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, pages 321–335, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3332466.3374531.

27 Christoph Kessler. Global optimization of operand transfer fusion in heterogeneous computing. In *Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems*, pages 49–58, 2019. doi:10.1145/3323439.3323981.

28 Yasir Noman Khalid, Muhammad Aleem, Usman Ahmed, Muhammad Arshad Islam, and Muhammad Azhar Iqbal. Troodon: A machine-learning based load-balancing application scheduler for cpu–gpu system. *Journal of Parallel and Distributed Computing*, 132:79–94, 2019. doi:10.1016/J.JPDC.2019.05.015.

29 Khronos, 2020. URL: https://registry.khronos.org/SYCL/specs/.

30 Gwangsun Kim, Jiyun Jeong, John Kim, and Mark Stephenson. Automatically exploiting implicit pipeline parallelism from multiple dependent kernels for gpus. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 341–352, 2016. doi:10.1145/2967938.2967952.

31 Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.

32 Changmin Lee, Won Woo Ro, and Jean-Luc Gaudiot. Boosting cuda applications with cpu–gpu hybrid computing. *International Journal of Parallel Programming*, 42(2):384–404, 2014. doi:10.1007/S10766-013-0252-Y.

33 Yunsup Lee, Ronny Krashinsky, Vinod Grover, Stephen W. Keckler, and Krste Asanović. Convergence and scalarization for data-parallel architectures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11, 2013. doi:10.1109/CGO.2013.6494995.

34 LLVM. LLVM Dominator Tree Wrapper Pass. URL: https://llvm.org/doxygen/classllvm_1_1DominatorTreeWrapperPass.html.

35 LLVM. LLVM LoopInfoWrapperPass Class Reference. URL: https://llvm.org/doxygen/classllvm_1_1LoopInfoWrapperPass.html.

36 LLVM. LLVM Clang. https://clang.llvm.org/, 2024. Retrieved: 2024-03-19.

37 Sparsh Mittal and Jeffrey S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4), July 2015. doi:10.1145/2788396.

38 Steven Muchnick et al. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

39 V. Krishna Nandivada, Jun Shirako, Jisheng Zhao, and Vivek Sarkar. A transformation framework for optimizing task-parallel programs. *ACM Trans. Program. Lang. Syst.*, 35(1), April 2013. doi:10.1145/2450136.2450138.

40 Ajay Nayak and Arkaprava Basu. Over-synchronization in gpu programs. *MICRO*, 2024.

41 Van Man Nguyen, Emmanuelle Saillard, Julien Jaeger, Denis Barthou, and Patrick Carribault. Automatic code motion to extend mpi nonblocking overlap window. In *High Performance Computing: ISC High Performance 2020 International Workshops, Frankfurt, Germany, June 21–25, 2020, Revised Selected Papers 35*, pages 43–54. Springer, 2020. doi:10.1007/978-3-030-59851-8_4.

42 Alexandru Nicolau, Guangqiang Li, and Arun Kejariwal. Techniques for efficient placement of synchronization primitives. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 199–208, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1504176.1504207.

43 Alexandru Nicolau, Guangqiang Li, Alexander V. Veidenbaum, and Arun Kejariwal. Synchronization optimizations for efficient execution on multi-cores. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 169–180, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1542275.1542303.

44 NVIDIA. CUDA Sample SDK. https://github.com/NVIDIA/cuda-samples.git, 2022. Retrieved: 2024-03-19.

45 David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, December 1986. doi:10.1145/7902.7904.

46 Anthony Pajot, Loic Barthe, Mathias Paulin, and Pierre Poulin. Combinatorial Bidirectional Path-Tracing for Efficient Hybrid CPU/GPU Rendering. *Computer Graphics Forum*, 30(2):315–324, April 2011. doi:10.1111/j.1467-8659.2011.01863.x.

47 Song Jun Park, James Ross, Dale Shires, David Richie, Brian Henz, and Lam Nguyen. Hybrid core acceleration of uwb sire radar signal processing. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):46–57, 2011. doi:10.1109/TPDS.2010.117.

48 Daniel P Playne, Mitchell Johnson, and Kenneth A Hawick. Benchmarking gpu devices with n-body simulations. *CDES*, 9:132, 2009.

49 Martin Rinard and Pedro Diniz. Eliminating synchronization bottlenecks in object-based programs using adaptive replication. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 83–92, New York, NY, USA, 1999. Association for Computing Machinery. doi:10.1145/305138.305167.

50 V. Sarkar. Instruction reordering for fork-join parallelism. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 322–336, New York, NY, USA, 1990. Association for Computing Machinery. doi:10.1145/93542.93590.

51 Charitha Saumya, Kirshanthan Sundararajah, and Milind Kulkarni. Darm: Control-flow melding for simt thread divergence reduction. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '22, pages 28–40. IEEE Press, 2022. doi:10.1109/CGO53902.2022.9741285.

52 Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David Kaeli. Hetero-mark, a benchmark suite for cpu-gpu collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

53 J.B. White III and J.J. Dongarra. Overlapping computation and communication for advection on hybrid parallel computers. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 59–67, 2011. doi:10.1109/IPDPS.2011.16.

54 Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. Gpucc: An open-source gpgpu compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 105–116, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2854038.2854041.

55 Ming Xu, Feiguo Chen, Xinhua Liu, Wei Ge, and Jinghai Li. Discrete particle simulation of gas–solid two-phase flows with multi-scale cpu–gpu hybrid computation. *Chemical Engineering Journal*, 207-208:746–757, 2012. 22nd International Symposium on Chemical Reaction Engineering (ISCRE 22). doi:10.1016/j.cej.2012.07.049.

56 Yingchun Zhu and Laurie J. Hendren. Communication optimizations for parallel c programs. *SIGPLAN Not.*, 33(5):199–211, May 1998. doi:10.1145/277652.277723.

57 Zjin-Lcf. Github – hecbench/src/entropy-cuda. URL: https://github.com/zjin-lcf/HeCBench/.

58 zjin lcf. Github – hecbench/src/xlqc-cuda, 2020. URL: https://github.com/zjin-lcf/HeCBench/.

Quantifying Cache Side-Channel Leakage by Refining Set-Based Abstractions

Jacqueline L. Mitchell
University of Southern California, Los Angeles, CA, USA

Chao Wang
University of Southern California, Los Angeles, CA, USA

Abstract

We propose an improved abstract interpretation based method for quantifying cache side-channel leakage by addressing two key components of precision loss in existing set-based cache abstractions. Our method targets two key sources of imprecision: (1) imprecision in the abstract transfer function used to update the abstract cache state when interpreting a memory access and (2) imprecision due to the incompleteness of the set-based domain. At the center of our method are two key improvements: (1) the introduction of a new transfer function for updating the abstract cache state which carefully leverages information in the abstract state to prevent the spurious aging of memory blocks and (2) a refinement of the set-based domain based on the finite powerset construction. We show that both the new abstract transformer and the domain refinement enjoy certain enhanced precision properties. We have implemented the method and compared it against the state-of-the-art technique on a suite of benchmark programs implementing both sorting algorithms and cryptographic algorithms. The experimental results show that our method is effective in improving the precision of cache side-channel leakage quantification.

2012 ACM Subject Classification Software and its engineering → Software verification and validation; Theory of computation → Program analysis

Keywords and phrases Abstract interpretation, side-channel, leakage quantification, cache

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.22

Related Version *Extended version with appendices and extra details*: `https://github.com/jlmitche23/ecoop25CacheQuantification` [15]

Supplementary Material *Software (ECOOP 2025 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.11.2.18`

Funding This work was partially supported by the NSF grant CCF-2220345.

1 Introduction

Cache side-channel attacks, whereby adversaries gain information about secret data by examining the footprint of program execution in the CPU cache, pose a significant threat to computer security. Cache side-channel attacks have been demonstrated in many critical infrastructure systems, ranging from cryptographic software in embedded devices [13, 30, 29, 1, 21, 19] to cloud computing applications where an adversary only needs remote access to the victim's hardware to successfully launch the attacks [5, 22, 6, 28]. Various techniques have been proposed to mitigate such attacks, including *constant-time programming* [16] along with verification techniques for proving the constant-time property [2, 4].

However, completely eliminating side-channel leakage is a challenging task since it may result in too much computational overhead [8]; it may also be infeasible for certain applications where some information leakage is required [24, 18, 27]. This motivates the development of mathematically rigorous techniques for *quantifying* side-channel leakage, to

© Jacqueline L. Mitchell and Chao Wang;
licensed under Creative Commons License CC-BY 4.0

allow programmers to audit the degree of leakage in software code. While the pioneering work of Doychev et al. [12, 11] show that abstract interpretation [9] using a set-based cache abstract domain is well-suited for quantifying cache side-channel leakage, the main limitation is the loss of precision in the quantification results.

To overcome this limitation, we propose a new method for improving the precision of abstract interpretation based static program analysis for quantifying cache side-channel leakage. Static program analysis based on abstract interpretation has the advantages of being sound, generally performant, and not requiring artificially-bounded loop iterations as in unsound alternative techniques based on bounded model checking [20] or symbolic execution [7]. However, these advantages of abstract interpretation may come at the cost of precision loss. There are two key sources of precision loss in the context of cache side-channel quantification. The first source is spuriously aging memory blocks in the cache while applying the so-called *abstract transfer function* which interprets memory-accessing instructions during the analysis. It does this by taking as input an abstract cache state and returning another abstract state which overapproximates the effect of accessing a memory block on any (concrete) cache state represented by the input abstract cache state. The second source of imprecision is the spurious aging of memory blocks due to the inability to express and leverage disjunctive invariants about the abstract cache states with respect to the control flow of a program (in other words, the incompleteness of the set-based abstract domain). Our new method is designed to mitigate these two key sources of precision loss.

At the center of our method is a novel abstract transfer function for the set-based abstract domain and an automatic lifting of the domain to more accurately capture invariants about the cache state. In this work, we have applied our method in the context of the abstract domain used by CacheAudit [11], a state-of-the-art tool for quantifying cache side-channel leakage. We denote this domain as $\mathcal{C}^\sharp$. An abstract state in the $\mathcal{C}^\sharp$ domain associates each memory block with a set of possible ages, which describes the possible positions for a memory block within the cache. The ages also determine how cache states are updated under a given replacement policy, as the result of interpreting a memory-accessing instruction. During an analysis step where abstract interpretation is used to compute the resulting abstract cache state after an access to memory, we say that a memory block b is *spuriously aged* to an age a if a is in the set of possible ages for b, and yet there is no valid concrete cache state in which b is of age a in the ground truth. In some cases, applying the best abstract transformer [9], which concretizes the abstract cache state to yield a set of concrete cache states, updates each concrete cache state according to a replacement policy, and then re-abstracts the set of updated concrete states into a new abstract cache state, can mitigate spurious aging. However, even if the best abstract transformer is used, a memory block may still be spuriously aged with respect to the collecting semantics, due to the incompleteness of $\mathcal{C}^\sharp$. Consider two abstract cache states C and C' that arise due to a difference in the control flow of a program (perhaps corresponding to two different branches of an if-statement). Even if the best abstract transformer does not age b to a in both C and C', this may not be true of their abstract union; we later provide an example in Section 4. This imprecision arises due to $\mathcal{C}^\sharp$'s inability to express disjunctive invariants at the level of variations in control flow.

To address the first source of precision loss, we propose to carefully leverage information in the abstract state regarding the ages of other memory blocks when deciding to age block b, to more accurately update the abstract cache state for each memory-accessing instruction. Instead of deciding to age b only based on b's age and the age of the accessed memory block, we use the ages of all the memory blocks in the cache to prevent the spurious aging of many

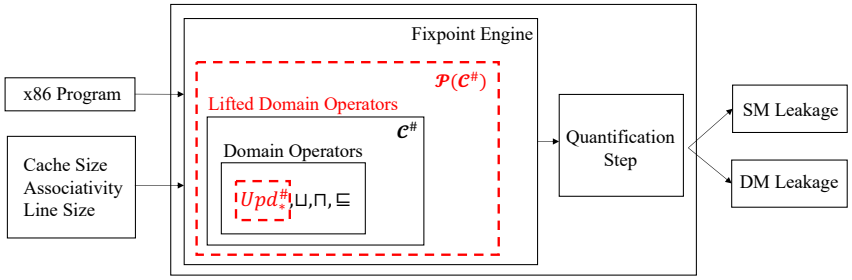


Figure 1 Our method for quantifying cache side-channel leakage based on abstract interpretation. SM corresponds to a shared memory adversary, where an attacker can observe which memory blocks are in the cache. DM corresponds to a disjoint memory adversary, where an attacker can observe which cache lines are occupied in the cache, but not the specific memory blocks occupying them.

memory blocks. As we describe later, we prove our improved transfer function improves upon the baseline transfer function by refining it in such a way that removes cases of spurious aging.

To address the second source of precision loss, we propose to parsimoniously leverage disjunctions of abstract cache states that arise due to variations in control flow. This technique can be implemented as a refinement of $\mathcal{C}^\sharp$, based on the powerset domain introduced by [3]. The powerset domain can refine any abstract domain by lifting its operators (partial order, join, meet, widen) to operate on a lifted version of the abstract domain, whose elements are a member of the *powerset* of elements of the abstract domain.

Figure 1 shows the overall flow of our method. The input to our method consists of a program P and the cache parameters. The program P is represented in x86 binary code. The cache parameters specify the total cache size, the associativity, and the cache line size. The output of our method is the cache side-channel leakage measured in bits, for two kinds of adversaries, explained in the following. The adversary type is either Shared Memory (SM) or Disjoint Memory (DM). At the end of a program's execution, the SM adversary is able to observe the placement of memory blocks in the final cache state, along with which memory blocks are in the various locations. In contrast, the DM adversary is only able to observe which locations of the cache are occupied in the final cache state, but not the specific memory blocks that occupy them. We note that other types of adversaries are possible; we have simply chosen these adversaries to empirically evaluate our techniques. The techniques are not specific to the two adversaries in the sense that other cache analyses may also depend on such set-based abstractions. Internally, our method consists of two innovative components, shown as the new transfer function $Upd^\sharp_*$ and the lifted domain which uses disjunctions, highlighted in red, dashed boxes in Figure 1.

We have evaluated our method on a suite of 29 benchmark programs, which are implementations of various sorting algorithms and cryptographic algorithms. The baseline that we use for comparison is CacheAudit [12]. We compared the two methods on all benchmark programs, with various cache settings and adversary types. In addition to a side-by-side comparison of our method against CacheAudit, we also conducted an ablation study by enabling each of the two new techniques and then comparing the performance. The goal is to check how effective each of the two techniques is in isolation across various cache configurations, and see if they have a synergistic effect when being used together. The experimental results show that, overall, our method significantly outperforms the state-of-the-art method. Furthermore, both of the two new techniques proposed in this paper are effective, and together, they have a synergistic effect.

In summary, this paper makes the following contributions:

- We propose a new method for more accurately quantifying cache side-channel leakage based on abstract interpretation.
- We introduce two novel techniques in our method. The first leverages a new abstract transfer function to prevent spurious aging of memory blocks in the cache during the analysis. The second leverages disjunctions parsimoniously to prevent spurious aging of memory blocks due to the incompleteness of $\mathcal{C}^\sharp$.
- We prove soundness and enhanced accuracy properties of the two novel techniques.
- We implement the method and demonstrate its advantages over the state-of-the-art technique on a suite of 29 benchmark programs.

The remainder of this paper is organized as follows. After providing the technical background in Section 2, we illustrate the limitations of prior work in Section 3 using an example. Then, we present our method in Section 4 and prove the soundness and accuracy properties. We present the experimental results in Section 5. After reviewing the related work in Section 6, we give our conclusion in Section 7.

2 Background

Unlike classic program analysis techniques that focus on functional properties, e.g., control and data flows of a program, quantifying side-channel leakage also requires the modeling and analysis of non-functional properties such as the cache state. Here, we introduce the components required for abstractly modeling cache behavior.

2.1 Modeling the Cache

A cache is used to bridge the latency gap between the fast CPU and the slow main memory, to reduce the overall execution time of a program. A cache is often divided into cache sets, each of which is further divided into cache lines, where each cache line has a fixed size. Formally, a cache with the size S, the associativity n, and the line size L is organized into $m = S/(L \cdot n)$ cache sets. Each cache set consists of n cache lines. Each cache line holds a contiguous block of L bytes. Throughout the paper, let $\mathcal{B}$ refer to the set of memory blocks under consideration.

Each memory block in $\mathcal{B}$ belongs to one cache set. We define the function $set : \mathcal{B} \to \{0, \ldots, m-1\}$ that maps each memory block $b \in \mathcal{B}$ its cache set $set(b) \in \{0, \ldots, m-1\}$. Given $b_1, b_2 \in \mathcal{B}$, the condition $set(b_1) = set(b_2)$ means that the two memory blocks map to the same cache set, whereas $set(b_1) \neq set(b_2)$ means that they map to different cache sets. When $set(b_1) \neq set(b_2)$, the two memory blocks map to different cache sets, and thus do not interfere with each other.

A concrete cache state c maps each memory block in $\mathcal{B}$ to a specific age in the set $\mathcal{A} = \{0, ..., n\}$ (recall, n is the associativity of the cache). Formally, $c : \mathcal{B} \to \mathcal{A}$, where $c(b) = n$ means that the block is outside of the cache, and $0 \leq c(b) \leq n-1$ means the block is inside the cache. The ages of memory blocks are determined by the so-called *cache replacement policy*. For example, with the popular LRU (least-recently used) policy, the age of a memory block b is determined by the number of other memory blocks accessed from the last time that b was accessed during program execution.

Let $\mathcal{C}$ be the set of concrete cache states. From a concrete cache $c \in \mathcal{C}$, executing an instruction that accesses a memory block $w \in \mathcal{B}$ leads to a new cache state $Upd(c, w) \in \mathcal{C}$. Here, $Upd : \mathcal{C} \times \mathcal{B} \to \mathcal{C}$ is called the transfer function.

▶ **Definition 1.** *The transfer function $Upd(c, w)$ for an LRU cache state $c \in \mathcal{C}$ and accessed memory block $w \in \mathcal{B}$ is defined as follows:*

$$Upd(c, w) := \lambda b \in \mathcal{B}. \begin{cases} c(b) & \text{when } set(b) \neq set(w) \\ c(b) & \text{when } set(b) = set(w) \wedge b \neq w \wedge c(b) = n \\ c(b) & \text{when } set(b) = set(w) \wedge b \neq w \wedge c(b) > c(w) \\ c(b) + 1 & \text{when } set(b) = set(w) \wedge b \neq w \wedge c(b) < c(w) \\ 0 & \text{when } set(b) = set(w) \wedge b = w \end{cases}$$

That is, the age of any memory block in a different cache set remains unchanged, as indicated by $set(b) \neq set(w)$. Within the same cache set, the age of the accessed memory block ($b = w$) is set to 0, the age of any memory block previously younger than the accessed block ($c(b) < c(w)$) increases by 1, and the age of any other memory block remains unchanged. In particular, $c(b) = n$ means the memory block b is already outside of the cache, and remains there upon an access to w. We note that following the LRU policy, any two memory blocks which belong to the same cache set cannot have the same age.

2.2 Abstract Interpretation of the Cache

Recall that $\mathcal{A}$ is a set of possible ages. Let $\mathcal{P}(\mathcal{A})$ be the powerset (set of all subsets) of $\mathcal{A}$, such that any element in $\mathcal{P}(\mathcal{A})$ represents a set of ages. Following Doychev et al. [11], we define the abstract cache state as a function $C : \mathcal{B} \to \mathcal{P}(\mathcal{A})$ that maps a block $b \in \mathcal{B}$ to a set of ages $C(b) \in \mathcal{P}(\mathcal{A})$. This is in contrast with the concrete state $c : \mathcal{B} \to \mathcal{A}$, which maps b to a single age $c(b)$.

Let $\mathcal{C}^\sharp$ be the set of abstract cache states. From an abstract cache state $C \in \mathcal{C}^\sharp$, executing an instruction that accesses a memory block $w \in \mathcal{B}$ leads to a new abstract cache state $Upd^\sharp(C, w) \in \mathcal{C}^\sharp$. Here, $Upd^\sharp : \mathcal{C}^\sharp \times \mathcal{B} \to \mathcal{C}^\sharp$ is called the abstract transfer function. Before defining $Upd^\sharp$, we need to define $C \downarrow_{w \mapsto c_w}$, which is a restriction of the abstract cache state C such that the age of block w is set to $c_w \in C(w)$. That is, $C \downarrow_{w \mapsto c_w}$ is an underapproximation of C where, since w occupies the age c_w, no other block can have the same age c_w, unless $c_w = n$ (meaning that w is outside of the cache), as is true in LRU caches. In the following, we define the abstract transfer function for a cache which follows the LRU replacement policy.

▶ **Definition 2.** *The abstract transfer function $Upd^\sharp(C, w)$ for a cache state $C \in \mathcal{C}^\sharp$ and accessed memory block $w \in \mathcal{B}$ is defined as follows [11]:*

$$Upd^\sharp(C, w) := \lambda b \in \mathcal{B}. \begin{cases} C(b) & \text{when } set(b) \neq set(w) \\ O_n\langle w \rangle \cup O_>\langle w \rangle \cup O_<\langle w \rangle & \text{when } set(b) = set(w) \wedge b \neq w \\ \{0\} & \text{when } set(b) = set(w) \wedge b = w \end{cases}$$

where $O_n\langle w \rangle \cup O_>\langle w \rangle \cup O_<\langle w \rangle$ computes a set of ages of block $b \in \mathcal{B}$ for each possible age $c_w \in C(w)$:

- $O_n\langle w \rangle := \bigcup_{c_w \in C(w)} \{c_b \mid c_b = n \wedge c_b \in C \downarrow_{w \mapsto c_w} (b)\}$ *has the ages equal to n,*
- $O_>\langle w \rangle := \bigcup_{c_w \in C(w)} \{c_b \mid c_b > c_w \wedge c_b \in C \downarrow_{w \mapsto c_w} (b)\}$ *has the ages older than c_w,*
- $O_<\langle w \rangle := \bigcup_{c_w \in C(w)} \{c_b + 1 \mid c_b < c_w \wedge c_b \in C \downarrow_{w \mapsto c_w} (b)\}$ *increments ages younger than c_w.*

The sets $O_n\langle w \rangle$, $O_>\langle w \rangle$ and $O_<\langle w \rangle$ in Definition 2 directly correspond to the three cases $c(b) = n$, $c(b) > c(w)$ and $c(b) < c(w)$ in Definition 1.

Abstract Domain ($\mathcal{C}^\sharp$): The universe is the set of abstract cache states. Element $\top$ (top) is a state $C \in \mathcal{C}^\sharp$ such that $\forall b \in \mathcal{B} \,.\, C(b) = \mathcal{A}$. Element $\bot$ (bottom) is a state $C \in \mathcal{C}^\sharp$ such that $\forall b \in \mathcal{B} \,.\, C(b) = \{\}$.

Partial Order ($\sqsubseteq_{\mathcal{C}^\sharp}$): Given two abstract cache states $C, C' \in \mathcal{C}^\sharp$, the ordering relation $C \sqsubseteq_{\mathcal{C}^\sharp} C'$ holds if and only if $\forall b \in \mathcal{B} \,.\, C(b) \subseteq C'(b)$.

Join ($\sqcup_{\mathcal{C}^\sharp}$): Given two abstract cache states $C, C' \in \mathcal{C}^\sharp$, the join is defined as $C \sqcup_{\mathcal{C}^\sharp} C' := \lambda b \in \mathcal{B} \,.\, C(b) \cup C'(b)$.

Meet ($\sqcap_{\mathcal{C}^\sharp}$): Given two abstract cache states $C, C' \in \mathcal{C}^\sharp$, the meet is defined as $C \sqcap_{\mathcal{C}^\sharp} C' := \lambda b \in \mathcal{B} \,.\, C(b) \cap C'(b)$.

Figure 2 The abstract domain $\mathcal{C}^\sharp$ and its partial order, join, and meet operators.

For example, consider $C = \{a \mapsto \{0,1\}, b \mapsto \{1,4\}, c \mapsto \{0,2,4\}\}$, accessed memory block b, and $n = 4$. We have $C \downarrow_{b \mapsto 1} := \{a \mapsto \{0\}, b \mapsto \{1\}, c \mapsto \{0,2,4\}\}$ because, when the age of b is 1, the age of a can no longer be 1. However, $C \downarrow_{b \mapsto 4} := \{a \mapsto \{0,1\}, b \mapsto \{4\}, c \mapsto \{0,2,4\}\}$ because multiple blocks can have the age 4 (meaning they are outside of the cache). Finally, $\mathit{Upd}^\sharp(C, b)$ returns the abstract cache state $\{a \mapsto \{1,2\}, b \mapsto \{0\}, c \mapsto \{1,2,4\}\}$.

2.3 The Baseline Algorithm

The baseline algorithm for quantifying cache side-channel leakage using abstract interpretation consists of two steps. The *analysis step* uses the abstract transfer function to compute an abstract cache state at each program location, to overapproximate the set of concrete cache states at that location. The *quantification step* leverages the abstract cache state C at the program exit point to compute the total number of concrete cache states, which is an upper bound of the information leakage (measured in bits).

The Analysis Step. An iterative procedure using abstract interpretation and the domain operations of $\mathcal{C}^\sharp$ is used to compute an abstract cache state at each program location. The procedure assumes that all memory blocks are outside of the cache initially, i.e., $\forall b \in \mathcal{B} \,.\, C(b) = \{n\}$. Then, it applies the abstract transfer function to the abstract cache state C at each program location to compute a new abstract cache state C'. Then, it conducts standard fixpoint iteration with the abstract transfer function and the domain operations. Fixpoint iteration is required to ensure that the abstract cache computed for each program location is an invariant, i.e., that it soundly overapproximates the set of possible concrete cache states at a given program location.

Figure 2 shows the abstract domain $\mathcal{C}^\sharp$ and its partial order ($\sqsubseteq_{\mathcal{C}^\sharp}$), join ($\sqcup_{\mathcal{C}^\sharp}$) and meet ($\sqcap_{\mathcal{C}^\sharp}$) operators. Consider abstract cache states $C_1, C_2, C_3, C_4 \in \mathcal{C}^\sharp$ as examples. If $C_1 = \{a \mapsto \{0,1\}, b \mapsto \{1,2\}\}$ and $C_2 = \{a \mapsto \{0,1,4\}, b \mapsto \{1,4\}\}$, then $C_1 \sqcup_{\mathcal{C}^\sharp} C_2 = \{a \mapsto \{0,1,4\}, b \mapsto \{1,2,4\}\}$ and $C_1 \sqcap_{\mathcal{C}^\sharp} C_2 = \{a \mapsto \{0,1\}, b \mapsto \{1\}\}$. However, if $C_3 = \{a \mapsto \{0\}, b \mapsto \{1\}\}$ and $C_4 = \{a \mapsto \{1\}, b \mapsto \{0\}\}$, then $C_3 \sqcap_{\mathcal{C}^\sharp} C_4 = \{a \mapsto \{\}, b \mapsto \{\}\}$, which equals the bottom element of $\mathcal{C}^\sharp$, $\bot$. The domain operations are used in the process of fixpoint iteration. For instance, when control flow paths in the program merge, the analysis must combine abstract states using the join operator ($\sqcup_{\mathcal{C}^\sharp}$), to remain a conservative overapproximation of the true set of cache states. Furthermore, the partial order $\sqsubseteq_{\mathcal{C}^\sharp}$ is used to detect if a fixpoint has been reached.

The Quantification Step. The abstract cache state C at the program exit point is used to compute the number of concrete cache states. This is accomplished by first mapping C from the abstract domain $\mathcal{C}^\sharp$ to the concrete domain $\mathcal{P}(\mathcal{C})$. Let $\gamma_{\mathcal{C}^\sharp}$ be the concretization

function, and $\gamma_{\mathcal{C}^\sharp}(C)$ be the set of concrete cache states. The cardinality $|\gamma_{\mathcal{C}^\sharp}(C)|$ represents the number of concrete cache states. In this case, $log_2|\gamma_{\mathcal{C}^\sharp}(C)|$ represents the maximum amount of information leakage measured in bits, according to Shannon's information theory.[1] We note that in our work, we assume that the leakage of each bit is equally valuable to the attacker, which motivates our use of Shannon entropy, as in CacheAudit [11].

▶ **Definition 3.** *The concretization function* $\gamma_{\mathcal{C}^\sharp} : \mathcal{C}^\sharp \to \mathcal{P}(\mathcal{C})$ *computes the set* $\gamma_{\mathcal{C}^\sharp}(C)$ *of concrete cache states for the abstract cache state* C *as follows:* $\gamma_{\mathcal{C}^\sharp}(C) :=$

$$\begin{aligned}\{c \in \mathcal{C} \mid\ & \forall b \in \mathcal{B} : c(b) \in C(b) \wedge \\ & \forall b_1, b_2 \in \mathcal{B} : set(b_1) = set(b_2) \wedge b_1 \neq b_2 \implies c(b_1) \neq c(b_2) \vee c(b_1) = c(b_2) = n \wedge \\ & \forall b_1 \in \mathcal{B} : 0 < c(b_1) < n \implies \exists b_2 \in \mathcal{B}.\ set(b_1) = set(b_2) \wedge (b_1 \neq b_2) \wedge c(b_2) = c(b_1) - 1\}\end{aligned}$$

The first condition $\forall b \in \mathcal{B} : c(b) \in C(b)$ takes the Cartesian product of the set of possible ages for each memory block $b \in \mathcal{B}$, while the last two conditions eliminate the obviously-invalid concrete cache states, according to the following two properties of LRU caches:

- ***No-collision within each cache set:*** If a cache line (age) is assigned to a memory block, it cannot be assigned to another memory block that belongs to the same cache set. Thus, if b_1 and b_2 belong to the same cache set ($set(b_1) = set(b_2)$) and $b_1 \neq b_2$, then $c(b_1) \neq c(b_2) \vee c(b_1) = c(b_2) = n$, meaning that the two blocks are either in different cache lines (ages) or are both outside of the cache.
- ***No-gap within each cache set:*** If a younger cache line (age) is available, an older cache line cannot be assigned to a memory block in a given cache set. Thus, when $c(b_1) \in \{1, ..., n-1\}$, there exists $b_2 \in \mathcal{B}$ such that $set(b_1) = set(b_2) \wedge (b_1 \neq b_2) \wedge c(b_2) = c(b_1) - 1$.

3 Limitations of Prior Work

While the baseline algorithm presented in Section 2 represents the state of the art, it has two main limitations in terms of the precision of its abstract transfer function and abstract domain. In this section, we use an example program to illustrate these limitations and then motivate our work on developing the new method.

3.1 The Example Program

Figure 3 shows the example program, which has a while loop containing an if-else statement. While the program has many variables, only four of them (a, b, c, and d) are being read. The two branches of the if-else statement differ in that the then-branch reads b and d whereas the else-branch reads c and d. This difference is sufficient to demonstrate the limitations of prior work and the advantages of our new method.

The Assumptions. For the sake of demonstration, we assume that all program variables in Figure 3 map to the same cache set. Furthermore, the cache set has only 4 cache lines. Finally, each variable occupies an entire cache line. With all of these assumptions, we have $\mathcal{B} = \{a, b, c, d\}$, $set(a) = set(b) = set(c) = set(d)$ and $n = 4$.

The reason why we focus only on these four variables is because, here, we assume that the cache is a *read-through, write-direct* cache as in Intel CPU's Data Direct I/O technology. That is, data is first read from main memory into the cache on a read operation, but when

[1] The Shannon entropy $H = \Sigma_c\ p(c)\ log_2 \frac{1}{p(c)}$ is maximized when each concrete cache state $c \in \gamma_{\mathcal{C}^\sharp}(C)$ has an equal probability $p(c) = \frac{1}{|\gamma_{\mathcal{C}^\sharp}(C)|}$, thus reducing H to $log_2 \frac{1}{p(c)} = log_2|\gamma_{\mathcal{C}^\sharp}(C)|$.

```
1  void foo (unsigned int d) {
2    unsigned int a, b, c, g, h = 0;
3
4    d = a + d; //read {a,d}
5
6    while (d >= 1 && d < 5) { //read {d}
7      if (d < 3) { //read {d}
8        h = b + 1; //read {b}
9        d--; //read {d}
10     }
11     else {
12       g = c + 1; //read {c}
13       d--; //read {d}
14     }
15   }
16
17 }
```

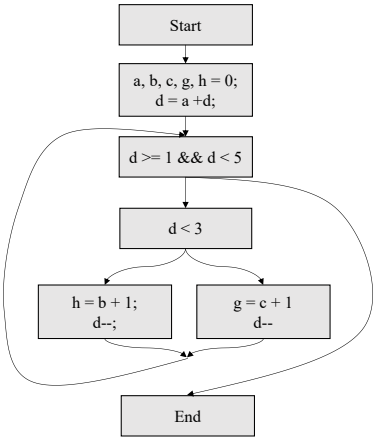


Figure 3 A program on the left-hand side and its control flow graph on the right-hand side.

writing data, it is directly written to the main memory without first updating the cache, effectively bypassing the cache for writes and prioritizing direct access to system memory. This aims to minimize unnecessary memory accesses. Under these assumptions, only read operations change the cache state.

Ground Truth. At the end of the program execution, there are only three valid concrete cache states: $c_1 = \{a \mapsto 1, b \mapsto 4, c \mapsto 4, d \mapsto 0\}$, $c_2 = \{a \mapsto 3, b \mapsto 1, c \mapsto 2, d \mapsto 0\}$, and $c_3 = \{a \mapsto 2, b \mapsto 1, c \mapsto 4, d \mapsto 0\}$. These three concrete cache states correspond to the following set of executions. State c_1 corresponds to the case where the body of the while loop is never entered, leaving c and b uncached (having age 4). State c_2 corresponds with executions in which the while loop is entered, and both branches of the if-statement are executed. (We note that due to the guards of both the while-loop and the if-statement, the then-branch is always executed after the else-branch when both are executed (when $d \geq 3$ at the beginning of the program), causing the cache line age of b to be younger than the cache line age of c. State c_3 corresponds to the case where only the then-branch of the if-statement is executed (when $d < 3$ at the beginning of the program), causing c to be uncached. With three possible concrete cache states, there is a maximum leakage of $\log_2(3)$ bits.

Baseline Algorithm. The abstract cache state at the last location of the program computed by the baseline algorithm in Section 2 is $C_{Last} := \{a \mapsto \{1,2,3,4\}, b \mapsto \{1,2,4\}, c \mapsto \{1,2,3,4\}, d \mapsto \{0\}\}$, which corresponds to 14 possible concrete cache states, where $|\gamma_{\mathcal{C}^\sharp}(C_{Last})| = 14$ (for reference, all 14 concrete cache states are featured in the appendix of the extended version [15]). This leads to a maximum leakage of $\log_2(14)$ bits, which is significantly higher than the ground truth $\log_2(3)$. As mentioned earlier, the baseline algorithm has two sources of imprecision, one of which is in the abstract transfer function $Upd^\sharp$ and the other is in the abstract domain $\mathcal{C}^\sharp$.

3.2 Imprecision of Abstract Transfer Function $\mathrm{Upd}^\sharp$

To see the imprecision in $Upd^\sharp$, consider the following abstract state C, which occurs prior to the third fixpoint iteration (using loop unrolling) of the while loop in Figure 3 using the baseline algorithm. That is, $C := \{a \mapsto \{2,3\}, b \mapsto \{1,2,4\}, c \mapsto \{1,2,4\}, d \mapsto \{0\}\}$.

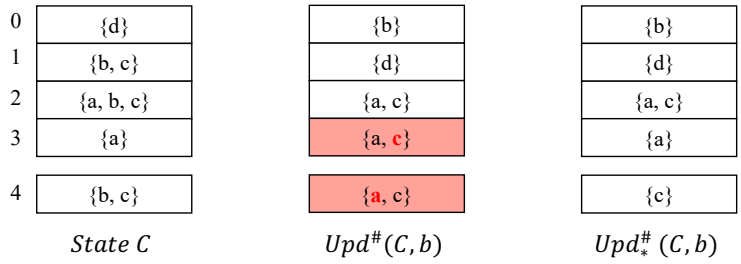


Figure 4 Differences in (baseline and new) abstract transfer functions, applied to abstract cache state $C \in \mathcal{C}^\sharp$ and the accessed memory block $b \in \mathcal{B}$.

The inverse of C, which maps from ages to memory blocks (variables), is shown in the left-most state of Figure 4.

Given abstract cache state C, consider the case of accessing (reading) variable b. As a result, the transfer function will return the abstract cache state $Upd^\sharp(C, b) := \{a \mapsto \{2, 3, 4\}, b \mapsto \{0\}, c \mapsto \{2, 3, 4\}, d \mapsto \{1\}\}$. Note that, due to spurious aging, the age 4 has become possible for a, and that the age 3 has become possible for c. However, according to the ground truth, there is no valid concrete cache state where a is outside of the cache, and there is no valid concrete cache state where c occupies the third cache line either.

In this work, we want to design a new transfer function $Upd^\sharp_*$ to eliminate such contradictory cache states. Intuitively, $Upd^\sharp_*(C, b)$ works as follows: when considering incrementing $3 \in C(a)$ to 4, $Upd^\sharp_*$ capitalizes on the fact that when a is of age 3, the set of variables with possible ages younger than 3 are $\{b, c, d\}$, as seen in the leftmost abstract cache state in Figure 4. Because there are only three such variables, and b is one of them, b must be younger than a when a is of age 3. Thus, it is unnecessary to increment $3 \in C(a)$ to 4 when accessing b, as b will already be younger than a. Similarly, $2 \in C(c)$ is not incremented to 3. Therefore, $Upd^\sharp_*(C, b) := \{a \mapsto \{2, 3\}, b \mapsto \{0\}, c \mapsto \{2, 4\}, d \mapsto \{1\}\}$, which corresponds to the rightmost abstract state in Figure 4. We shall explain more formally in Section 4.1 that, by replacing $Upd^\sharp$ with $Upd^\sharp_*$ in the iterative procedure used to analyze the program in Figure 3, our method will compute a better final abstract cache state, $C_{Last} := \{a \mapsto \{1, 2, 3\}, b \mapsto \{1, 2, 4\}, c \mapsto \{1, 2, 4\}, d \mapsto \{0\}\}$, which corresponds to seven (instead of 14) concrete cache states at the end of program execution.

3.3 Imprecision of Abstract Domain $\mathcal{C}^\sharp$

To understand the limitation of the $\mathcal{C}^\sharp$ domain, consider the final abstract cache state $C_{Last} := \{a \mapsto \{1, 2, 3\}, b \mapsto \{1, 2, 4\}, c \mapsto \{1, 2, 4\}, d \mapsto \{0\}\}$ computed at the exit point of the example program in Figure 3 by using $Upd^\sharp_*$. As mentioned earlier, this abstract cache state corresponds to seven concrete cache states. Compared to the ground truth, which has three concrete cache states c_1, c_2 and c_3 (defined in the previous subsection), the abstract cache state C_{Last} has 4 more (spurious) concrete cache states shown below: $c_4 = \{a \mapsto 3, b \mapsto 2, c \mapsto 1, d \mapsto 0\}$, $c_5 = \{a \mapsto 2, b \mapsto 4, c \mapsto 1, d \mapsto 0\}$, $c_6 = \{a \mapsto 1, b \mapsto 4, c \mapsto 2, d \mapsto 0\}$, and $c_7 = \{a \mapsto 1, b \mapsto 2, c \mapsto 4, d \mapsto 0\}$. These spurious cache states are due to the fact that $\mathcal{C}^\sharp$ is not capable of precisely capturing disjunctive invariants that arise due to variations in control flow.

Specifically, these spurious states result from an inability of $\mathcal{C}^\sharp$ to distinguish between when the while loop is entered or not, and whether the else branch is entered at least once in the program in Figure 3. To see why, consider the final abstract cache state C_{Last}, where 1 is

a possible age for a, 0 is a possible age for d, 2 is a possible age for c, and 4 is a possible age for b, thus allowing the concrete cache state $c_6 = \{a \mapsto 1, b \mapsto 4, c \mapsto 2, d \mapsto 0\}$. However, a is of age 1 only when the loop is not entered, but c being in the cache indicates that c was accessed in Line 12 of the program, and that the loop body was entered.

In this work, we want to remove these spurious states by leveraging the finite powerset framework of Bagnara et al. [3], which computes a bounded set of states (instead of a single state) at each program location. We shall explain in Section 4.2 that, in the context of cache side-channel analysis, this is accomplished by lifting the abstract domain $\mathcal{C}^\sharp$ to the powerset domain $\mathcal{P}(\mathcal{C}^\sharp)$ where each element has a cardinality of less than or equal to K. In practice, the bound K may be a small number, e.g., $K = 10$.

For the example program in Figure 3, $K = 3$ would be sufficient. That is, by using an abstract domain whose elements consist of a set of at most three elements of $\mathcal{C}^\sharp$ (as opposed to a single abstract state) to conduct fixpoint iteration with a lifted version of $Upd^\sharp_*$, we end up with the following abstract state: $\{\{a \mapsto \{1\}, b \mapsto \{4\}, c \mapsto \{4\}, d \mapsto \{0\}\}, \{a \mapsto \{3\}, b \mapsto \{1\}, c \mapsto \{2\}, d \mapsto \{0\}\}, \{a \mapsto \{2\}, b \mapsto \{1\}, c \mapsto \{4\}, d \mapsto \{0\}\}\}$, which corresponds to the three valid concrete cache states in the ground-truth.

We also emphasize that maintaining disjunctive invariants is able to prevent spurious aging caused by merging two abstract cache states. To see this, consider the following minor modification of code: suppose that the statement `h = g` is added between lines 8 and 9, indicating that g is read at that program location, in the then-branch of the if-statement. In the *first* iteration of analyzing the code with loop unrolling, the abstract states to be merged at the end of the if-statement are $C_{Then} := \{a \mapsto \{3\}, b \mapsto \{2\}, c \mapsto \{4\}, d \mapsto \{0\}, g \mapsto \{1\}\}$ and $C_{Else} := \{a \mapsto \{2\}, b \mapsto \{4\}, c \mapsto \{1\}, d \mapsto \{0\}, g \mapsto \{4\}\}$. Then, consider in the next iteration accessing variable b; $Upd^\sharp(C_{Then}, b) := \{a \mapsto \{3\}, b \mapsto \{0\}, c \mapsto \{4\}, d \mapsto \{1\}, g \mapsto \{2\}\}$. $Upd^\sharp(C_{Else}, b) := \{a \mapsto \{3\}, b \mapsto \{0\}, c \mapsto \{2\}, d \mapsto \{1\}, g \mapsto \{4\}\}$. Notice that in either case, a is not aged to 4. Now consider $C_{Both} := C_{Then} \sqcup_{\mathcal{C}^\sharp} C_{Else} = \{a \mapsto \{2, 3\}, b \mapsto \{2, 4\}, c \mapsto \{1, 4\}, d \mapsto \{0\}, g \mapsto \{1, 4\}\}$. We can see that $Upd^\sharp(C_{Both}, b)$ ages a from 3 to 4. Thus, maintaining disjunctive invariants (avoiding merging C_{Then} and C_{Else}) at this point can also prevent spurious aging. As we describe in more detail in Section 4, $Upd^\sharp_*$ is also unable to prevent spurious aging in this case, necessitating disjunctive invariants.

4 Our Method

We now present the two new techniques of our method for overcoming limitations of prior work. The first is a new abstract transfer function that prevents spurious aging of memory blocks in the cache. The second is a technique that lifts the abstract domain $\mathcal{C}^\sharp$ of states to *sets* of abstract cache states, to prevent spurious combinations of cache states.

4.1 The Abstract Transfer Function $Upd^\#_*$

Given an abstract cache state $C \in \mathcal{C}^\sharp$ and the accessed memory block $w \in \mathcal{B}$, we want to define $Upd^\#_*(C, w)$ such that it is significantly more accurate than the baseline $Upd^\sharp(C, w)$ defined in Section 2.3. Here, the focus is on eliminating contradictory cache states due to spurious aging of memory blocks, to tighten the gap between $Upd^\sharp$ and the *best abstract transformer* for $\mathcal{C}^\sharp$, which concretizes the abstract state C using $\gamma_{\mathcal{C}^\sharp}$, applies the concrete update function Upd to each concrete state, and abstracts the resulting set of concrete states.

4.1.1 The Intuition

To this end, recall that for the example program in Figure 3, when b is the accessed memory block and $C := \{a \mapsto \{2,3\}, b \mapsto \{1,2,4\}, c \mapsto \{1,2,4\}, d \mapsto \{0\}\}$, the spurious aging of a to 4 and the spurious aging of c to 3 will occur in $Upd^\sharp(C,b)$ when considering the case where b is of age 4 in C, meaning that, previously, b was outside of the cache.

Increasing the age of a from 3 to 4 is *spurious aging* because, when a is of age 3, to avoid a gap in the cache, the younger cache lines (with ages 0, 1, and 2) must hold b, c and d. Since the age of b is either 1 or 2, accessing b should not increase the age of a from 3 to 4. Increasing the age of c from 2 to 3 is also *spurious aging* because, when c is of age 2, to avoid a gap in the cache, the younger cache lines (with ages 0 and 1) must hold b and d. Since the age of b must be 1, accessing b should not increase the age of c from 2 to 3.

Leveraging the above reasoning, we want the new transfer function to return $Upd^\sharp_*(C,b) := \{a \mapsto \{2,3\}, b \mapsto \{0\}, c \mapsto \{2,4\}, d \mapsto \{1\}\}$. It is more accurate than $Upd^\sharp(C,b)$ as shown by the middle and right-most states in Figure 4 where the spurious ages in $Upd^\sharp(C,b)$ are highlighted in red. In fact, this is the best result that any transfer function can possibly achieve; that is, even if we concretize the abstract state C, apply $Upd(c,b)$ for every concrete state c, then re-abstract these concrete states, we will get the same abstract state. However, applying the aforementioned "best" abstract transformer will not be computationally efficient. In the subsections that follow, we introduce two core components of our new transfer function, $Upd^\sharp_*$, defined in Definition 4, to capitalize on the intuition.

4.1.2 The Function $Var(C, c_b, b)$

We first define $Var : \mathcal{C}^\sharp \times \mathcal{A} \times \mathcal{B} \to \mathcal{P}(\mathcal{B})$ as a function that takes an abstract cache state $C \in \mathcal{C}^\sharp$, an age $c_b \in \mathcal{A}$, and a memory block $b \in \mathcal{B}$ as input and returns the set of memory blocks belonging to the same cache set as b which are possibly younger than c_b in C. Formally, $Var(C, c_b, b) := \{b' \in \mathcal{B} \mid \exists c'_b \in C(b') \;.\; c'_b < c_b \wedge set(b) = set(b')\}$.

For example, consider $C := \{a \mapsto \{2,3\}, b \mapsto \{1,2,4\}, c \mapsto \{1,2,4\}, d \mapsto \{0\}\}$ (for the ease of demonstration, we assume that all memory blocks map to the same cache set). If $c_b = 3$, the set of memory blocks that are possibly younger are $\{a,b,c,d\}$; thus, we have $Var(C,3,b) = \{a,b,c,d\}$. However, if $c_b = 2$, we have $Var(C,2,b) = \{b,c,d\}$.

Given memory block c of age 2, we use $Var(C,2,c)$ to represent the set of memory blocks (in the same cache set as c) which are possibly younger than 2 in C, and then use $Var(C,2,c) \setminus \{c\}$ to remove the memory block c itself. To decide if another block b (such that $set(b) = set(c)$) may be younger than c, we check $b \in Var(C,2,c) \setminus \{c\}$. For our running example, where $Var(C,2,c) = \{b,c,d\}$ and $Var(C,2,c) \setminus \{c\} = \{b,d\}$, the check passes, meaning that b may be younger than c (when c is of age 2).

To summarize, the above discussion shows that, in general, the condition $w \in Var(C, c_b, b) \setminus \{b\}$ checks if block $w \in \mathcal{B}$ may be younger than block $b \in \mathcal{B}$, when b is of age $c_b \in C(b)$ and $set(b) = set(w)$. In the next subsection, we show how to convert this "may" information into "must" information, to understand when a memory block b *must* be younger than a certain cache line age.

4.1.3 The Cardinality $|Var(C, c_b, b) \setminus \{b\}|$

Since $Var(C, c_b, b) \setminus \{b\}$ is the set of blocks in the same cache set which are younger than $b \in \mathcal{B}$, when b is of age $c_b \in C(b)$, the cardinality of the set is the number of such younger blocks. When $|Var(C, c_b, b) \setminus \{b\}| \leq c_b$, to avoid gaps in the cache, the younger cache lines (of

ages $0, \ldots, c_b - 1$) must be filled with these younger blocks. Thus, if $w \in Var(C, c_b, b) \setminus \{b\}$ also holds, the age of block w is younger than the age of block b, when b is of age c_b. Thus, accessing block w should not increase the age of block b when b is of age c_b.

The above condition holds in the running example when b is the accessed memory block and a is of age 3. Since $Var(C, 3, a) \setminus \{a\} = \{b, c, d\}$ and $|Var(C, 3, a) \setminus \{a\}| = 3$, both conditions $|Var(C, 3, a) \setminus \{a\}| \leq 3$ and $b \in (Var(C, 3, a) \setminus \{a\})$ hold, meaning that the age of b is younger than the age of a, when a is of age 3. Thus, accessing b should not increase the age of a, when a is of age 3. We emphasize that if the condition $|Var(C, 3, a) \setminus \{a\}| \leq 3$ does not hold, i.e., $|Var(C, 3, a) \setminus \{a\}| > 3$, we would not be able to ascertain that b must be younger than 3. This comes down to the "pigeon-hole" principle, where we know that if there are 4 variables for 3 possible cache lines, then b is not guaranteed to be younger than 3.

4.1.4 The Algorithm for Computing $Upd^\sharp_*(C, w)$

We define $Upd^\sharp_*(C, w)$ by revising the sets $O_>\langle w\rangle$ and $O_<\langle w\rangle$ shown in Definition 2 for $Upd^\sharp$.

▶ **Definition 4** ($Upd^\sharp_*$). *The abstract transfer function $Upd^\sharp_*(C, w)$ for cache state $C \in \mathcal{C}^\sharp$ and accessed memory block $w \in \mathcal{B}$ is defined as follows:*

$$Upd^\sharp_*(C, w) := \lambda b \in \mathcal{B}. \begin{cases} C(b) & \text{when } set(b) \neq set(w) \\ O_n\langle w\rangle \cup O_>\langle w\rangle \cup O_<\langle w\rangle & \text{when } set(b) = set(w) \wedge b \neq w \\ \{0\} & \text{when } set(b) = set(w) \wedge b = w \end{cases}$$

where $O_n\langle w\rangle \cup O_>\langle w\rangle \cup O_<\langle w\rangle$ computes a set of ages of block $b \in \mathcal{B}$ for each possible age $c_w \in C(w)$:

- $O_n\langle w\rangle := \bigcup_{c_w \in C(w)} \{c_b \mid c_b = n \wedge c_b \in C \downarrow_{w \mapsto c_w} (b)\}$ *has the ages equal to n,*
- $O_>\langle w\rangle := \bigcup_{c_w \in C(w)} \{c_b \mid (c_b > c_w \vee (\ |\mathbf{Var(C, c_b, b) \setminus \{b\}}| \leq \mathbf{c_b} \wedge \mathbf{w} \in \mathbf{Var(C, c_b, b) \setminus \{b\}}\ )) \wedge c_b \in C \downarrow_{w \mapsto c_w} (b)\}$ *has the ages older than c_w,*
- $O_<\langle w\rangle := \bigcup_{c_w \in C(w)} \{c_b + 1 \mid (c_b < c_w \wedge (\neg(\ |\mathbf{Var(C, c_b, b) \setminus \{b\}}| \leq \mathbf{c_b} \wedge \mathbf{w} \in \mathbf{Var(C, c_b, b) \setminus \{b\}}\ )) \wedge c_b \in C \downarrow_{w \mapsto c_w} (b)\}$ *represents the effect on ages younger than c_w.*

The sets $O_>\langle w\rangle$ and $O_<\langle w\rangle$ are revised such that, when the **highlighted** condition in $O_>\langle w\rangle$ is satisfied, we avoid incrementing the age of block b. The condition holds when the number of variables (excluding b) younger than c_b is less than or equal to c_b, and the accessed block w is one of the younger blocks. This is to prevent the spurious aging of block b.

For the example in Figure 3, in particular, the newly added conditions to $O_>\langle b\rangle$ and $O_<\langle b\rangle$ avoid the spurious aging of a from 3 to 4 and c from 2 to 3, as shown in Figure 4. Thus, by replacing $Upd^\sharp$ with $Upd^\sharp_*$ in the iterative procedure, the final abstract cache state at the end of the program in Figure 3 becomes $\{a \mapsto \{1, 2, 3\}, b \mapsto \{1, 2, 4\}, c \mapsto \{1, 2, 4\}, d \mapsto \{0\}\}$, which corresponds to seven (instead of 14) concrete cache states.

4.1.5 The Soundness Property

This technique is sound in that it computes an overapproximation of the concrete cache states. Recall that $Upd(c, w)$ is the concrete transfer function for a concrete cache state c and the accessed memory block w, and $\gamma_{\mathcal{C}^\sharp}$ is the concretization function. To streamline notation in the following sections, we denote Upd_w as a function which takes as input a concrete cache state, and returns the cache state after having accessed w. (This can be thought of as currying the w argument in Definition 1).

To prove soundness, we will show that $Upd^\sharp_*$ subsumes the result of the best abstract transformer. To prove this, we first explicitly define the corresponding abstraction function $\alpha_{\mathcal{C}^\sharp} : \mathcal{P}(\mathcal{C}) \to \mathcal{C}^\sharp$, which takes as input a set of *concrete* cache states and returns an *abstract* cache state overapproximating the set.

▶ **Definition 5** ($\alpha_{\mathcal{C}^\sharp}$). *Let S denote some set of concrete cache states. Then, $\alpha_{\mathcal{C}^\sharp}(S) := \lambda b \in \mathcal{B}.\{c(b) \mid c \in S\}$*

We now state the formal claim of soundness in the following theorem:

▶ **Theorem 6.** *$Upd^\sharp_*$ is sound in that, for any $w \in \mathcal{B}$ and $C \in \mathcal{C}^\sharp$, $\alpha_{\mathcal{C}^\sharp} \cdot Upd_w \cdot \gamma_{\mathcal{C}^\sharp}(C) \sqsubseteq_{\mathcal{C}^\sharp} Upd^\sharp_*(C, w)$.*

Proof. In the interest of space, we defer the full proof to the appendix of the extended version [15], and instead provide a proof sketch here. In the following, let b refer to some memory block in $\mathcal{B}$ whose ages are being updated a result of accessing memory block w.

1. We prove the soundness of $Upd^\sharp_*$ by showing that it subsumes the result of the best abstract transformer.
2. We show this by proving that if there is some concrete state c' that is the result of applying Upd_w to some state $c \in \gamma_{\mathcal{C}^\sharp}(C)$, where $c'(b) = a'$, then $a' \in Upd^\sharp_*(C, w)(b)$.
3. If $b = w$, then for all concrete states $c \in \gamma_{\mathcal{C}^\sharp}(C)$, $c(b) = 0$. It is clear to see that $0 \in Upd^\sharp_*(C, w)(b)$.
4. Otherwise, if $b \neq w$, there are three cases. First, if there is some state c, where $c(b) = n$, then $c'(b) = n$. It is clear from the definition of $Upd^\sharp_*$, that $n \in Upd^\sharp_*(C, w)(b)$ (Case $O_n\langle w\rangle$). Second, we show that if there is a concrete state c where block b is older than w, that $c(b) \in Upd^\sharp_*(C, w)(b)$ (Case $O_>\langle w\rangle$). Third, we show that if there is a concrete state c where b is younger than w, then $c(b) + 1 \in Upd^\sharp_*(C, w)(b)$ (Case $O_<\langle w\rangle$).
5. By showing that 3-4 hold, we have proved our claim. ◀

4.1.6 The Accuracy Property

We now argue that $Upd^\sharp_*$, as defined in Definition 4, is a refinement of $Upd^\sharp$, as defined in Definition 2. More formally stated, $\alpha_{\mathcal{C}^\sharp} \cdot Upd_w \cdot \gamma_{\mathcal{C}^\sharp} \sqsubseteq Upd^\sharp_*(\cdot, w) \sqsubseteq Upd^\sharp(\cdot, w)$. We now present the key theorem, describing the refinement relationship between the two transformers.

▶ **Theorem 7.** *The abstract transformer $Upd^\sharp_*$ is always more precise than, or equal to the abstract transformer $Upd^\sharp$.*

Proof. To show this, we will proceed by demonstrating that given abstract cache state C, and a memory block w to be accessed, for all $b \in \mathcal{B}$, $Upd^\sharp_*(C, w)(b) \subseteq Upd^\sharp(C, w)(b)$. We will proceed by cases.

1. **Case $\mathbf{set(b) \neq set(w)}$.** In this case, $Upd^\sharp_*(C, w)(b) = C(b)$ and $Upd^\sharp(C, w)(b) = C(b)$ by their respective definitions. Thus, $Upd^\sharp_*(C, w)(b) \subseteq Upd^\sharp(C, w)(b)$ follows immediately.
2. **Case $\mathbf{set(b) = set(w)}$.** In the case where w and b belong to the same cache set, we split up the proof into the following two cases:
   a. $\mathbf{w = b}$. In this case, memory block b is the block being accessed. Therefore $Upd^\sharp_*(C, w)(b) = \{0\}$ and $Upd^\sharp(C, w)(b) = \{0\}$, by definition. Thus, the subset relationship follows immediately.
   b. $\mathbf{w \neq b}$. In this case, we will use $O^*_n\langle w\rangle$, $O^*_>\langle w\rangle$, $O^*_<\langle w\rangle$, and $O_n\langle w\rangle$, $O_>\langle w\rangle$, $O_<\langle w\rangle$ to refer to the corresponding components of $Upd^\sharp_*$ and $Upd^\sharp$, respectively. For the sake of brevity, let $\mathbf{M}$ refer to the predicate $\mathbf{|Var(C, c_b, b) \setminus \{b\}| \leq c_b \wedge w \in Var(C, c_b, b) \setminus \{b\}}$. To prove the claim, we will show that $O^*_n\langle w\rangle \cup O^*_>\langle w\rangle \cup O^*_<\langle w\rangle \subseteq O_n\langle w\rangle \cup O_>\langle w\rangle \cup O_<\langle w\rangle$.
   It follows by definition that $O^*_n\langle w\rangle \subseteq O_n\langle w\rangle$.
   To see why $O^*_<\langle w\rangle \subseteq O_<\langle w\rangle$, it can be shown that for any c_b, c_w, that $\{c_b + 1 \mid (c_b < c_w \wedge \neg\mathbf{M}) \wedge c_b \in C\downarrow_{w\mapsto c_w}(b)\} \subseteq \{c_b + 1 \mid (c_b < c_w) \wedge c_b \in C\downarrow_{w\mapsto c_w}(b)\}$. This simply follows from the fact that $c_b < c_w \wedge \neg\mathbf{M} \implies c_b < c_w$.

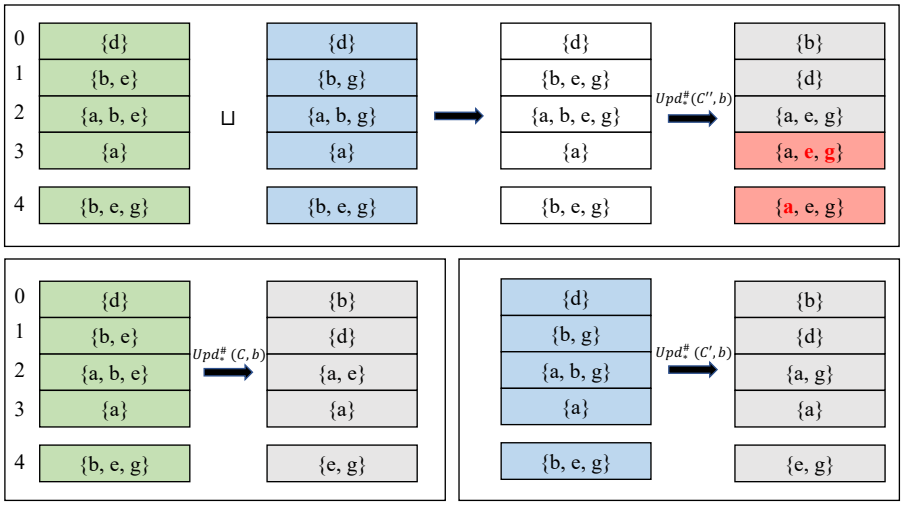


Figure 5 Merging two cache states which leads to spurious aging.

Finally, it can be shown that $O^*_>\langle w\rangle \subseteq O_>\langle w\rangle$. Let $c_b \in O^*_>\langle w\rangle$. Then, either of the two conditions hold:

i. $\exists \mathbf{c_w} \,.\, \mathbf{c_b} > \mathbf{c_w}$. If $c_b > c_w$ holds, then $c_b \in O_>\langle w\rangle$, by definition of $Upd^\sharp$.
ii. **M**. If **M** holds, then this implies that $w \in Var(C, c_b, b) \setminus \{b\}$. This in turn implies that there exists some c'_w such that $c'_w < c_b$. If this is the case, then $c_b \in \{c_b \mid (c_b > c'_w) \wedge c_b \in C \restriction_{w \mapsto c'_w} (b)\}$. Therefore, $c_b \in O_>\langle w\rangle$, by definition.

Given the fact that $O^*_n\langle w\rangle \subseteq O_n\langle w\rangle$, $O^*_>\langle w\rangle \subseteq O_>\langle w\rangle$, and $O^*_<\langle w\rangle \subseteq O_<\langle w\rangle$, it follows that $O^*_n\langle w\rangle \cup O^*_>\langle w\rangle \cup O^*_<\langle w\rangle \subseteq O_n\langle w\rangle \cup O_>\langle w\rangle \cup O_<\langle w\rangle$.

Therefore, in any case, $Upd^\sharp_*(C, w)(b) \subseteq Upd^\sharp(C, w)(b)$. ◀

4.2 Refining the $\mathcal{C}^\sharp$ Abstract Domain

We now present the technique for extending the abstract domain to a finite powerset domain, through the framework of Bagnara et al. [3] to improve the precision of the analysis.

4.2.1 The Intuition

We first use examples to illustrate the benefit of maintaining disjunctive invariants and show how to instantiate the framework in the context of cache analysis, which leverages it to maintain a set of elements of $\mathcal{C}^\sharp$, rather than a single element of $\mathcal{C}^\sharp$.

▶ **Example 8.** As an example of why refining $\mathcal{C}^\sharp$ is useful, consider the example in Figure 5, a case where $Upd^\sharp_*$ is unable to prevent precision loss. For both the blue and green abstract cache states, when applying $Upd^\sharp_*(\cdot, b)$ on both states individually (the bottom row of the figure), we can see that it is not possible for a be age 4, nor is it possible for e or g to be age 3. However, this is not the case in their abstract join. We emphasize that applying the best abstract transformer on the joined state does not prevent this either. This indicates an imprecision of the abstract domain $\mathcal{C}^\sharp$ as opposed to sub-optimality of $\mathcal{C}^\sharp$'s operators. Thus, it is desirable to keep these two abstract states separate. More explicitly, it is desirable to have an abstract domain which maintains a set of elements of $\mathcal{C}^\sharp$, e.g. $\{C, C'\}$.

Cache-Based Merging Operator ($\Omega_R^{\mathcal{C}^\sharp} : \mathcal{VC}^\sharp \to \mathcal{VC}^\sharp$). $\Omega_R^{\mathcal{C}^\sharp}$ takes in an abstract state S and merges any two elements of S if they have the same abstract **cache** state.

Subsumption Operator ($\Omega^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} : \mathcal{VC}^\sharp \to \mathcal{VC}^\sharp$). $\Omega^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}$ takes in an abstract state S and removes elements of S if they are subsumed by some other state in S. $\Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(S) \mapsto S'$, where $S' := S \backslash \{s \in S \mid s = \bot_{\mathcal{VC}^\sharp} \vee \exists s' \in S. s \sqsubset_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s'\}$.

Figure 6 The subsumption and merging operators with respect to $\sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}$ and $\sqsubseteq_{\mathcal{C}^\sharp}$.

Furthermore, the refinement can be conducted when a main-channel (program values) analysis is conducted simultaneously with a side-channel (cache states) analysis. We refer to the abstract domain used in the main-channel analysis as $\mathcal{V}^\sharp$. Let $\mathcal{V}$ be some numerical abstract domain which approximates a numerical domain ($\mathcal{P}(\mathbb{Z})$) (for instance, $\mathcal{V}$ may be the domain of intervals or a domain of integer-valued sets). Let V be the set of program variables. Then, we assume $\mathcal{V}^\sharp := V \to \mathcal{V}$ is the abstract domain for the set of variables in the program which consists of maps from variables to an abstract value representation $\mathcal{V}$.

In this case, the abstract domain to be refined is the abstract domain which has elements of tuples of an abstract state in $\mathcal{V}^\sharp$ and an abstract state in $\mathcal{C}^\sharp$. The respective abstract domain operators are applied, independently, pointwise. The domain is denoted by $\mathcal{V}^\sharp \times \mathcal{C}^\sharp$.

In fact, refinement at the level of both the program value and cache abstractions can be useful, because if the value abstractions are more precise, then certain paths in the control of the program (and subsequently, memory accesses) may be eliminated, possibly leading to abstract cache states that are more precise. We write the rest of the section with this in mind (and it corresponds to the set-up in our evaluation). Therefore, in the remainder of this section we consider the concrete domain to be $\mathcal{P}(V \to \mathbb{Z}) \times \mathcal{P}(\mathcal{C})$.

4.2.2 The Finite Powerset Domain

In this section, we introduce the finite powerset domain. We first begin by introducing relevant notation and operators.

The maximum number of abstract states of type $\mathcal{V}^\sharp \times \mathcal{C}^\sharp$ allowed in the aforementioned set, denoted k, is pre-defined by the user. We refer to elements of such a set as disjuncts. In our case, the finite powerset framework can be thought of taking $\mathcal{V}^\sharp \times \mathcal{C}^\sharp$, which approximates $\mathcal{P}(V \to \mathbb{Z}) \times \mathcal{P}(\mathcal{C})$, and replacing it with an abstract domain which still approximates $\mathcal{P}(V \to \mathbb{Z}) \times \mathcal{P}(\mathcal{C})$, but using a *set* of abstract values in $\mathcal{V}^\sharp \times \mathcal{C}^\sharp$, that is, an element of the powerset of $\mathcal{V}^\sharp \times \mathcal{C}^\sharp$, $\mathcal{P}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$. With a slight abuse of notation, let $\mathcal{V}^\sharp \times \mathcal{C}^\sharp := \langle \mathcal{V}^\sharp \times \mathcal{C}^\sharp, \sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}, \bot_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}, \top_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}, \sqcup_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}, \sqcap_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} \rangle$ denote the abstract domain $\mathcal{V}^\sharp \times \mathcal{C}^\sharp$, along with its operators. We say that an element $S \in \mathcal{P}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$ is *non-redundant* with respect to $\sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}$ if and only if $\bot_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} \notin S$ and $\forall s_1, s_2 \in S. s_1 \sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s_2 \implies s_1 = s_2$. Non-redundancy ensures that a set of abstract states does not contain unnecessary elements that are already represented by other elements in the set.

A *subsumption operator* serves to normalize an element $S \in \mathcal{P}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$ by removing redundant elements. The formal definition of the subsumption operator is in Figure 6. The $\Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}$ operator removes redundant states based on both abstract program states and abstract cache states. The $\Omega_R^{\mathcal{C}^\sharp}$ operator merges abstract states which share the same abstract *cache* states; we emphasize that $\Omega_R^{\mathcal{C}^\sharp}$ does not remove any states based on redundancy, it simply merges abstract states which share the same abstract *cache* states. As we will see later on, the subsumption operator is used in the definition of the join, meet, and widening operators, while $\Omega_R^{\mathcal{C}^\sharp}$ is used in the definition of the join operator.

Let $\mathcal{P}_{fn(k)}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp, \sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp})$ denote the set of all elements of $\mathcal{P}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$ which have a cardinality of at most k. Formally, $\mathcal{P}_{fn(k)}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp, \sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}) := \{S \in \mathcal{P}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp) \mid |S| \leq k\}$, where every element S is non-redundant according to $\sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}$. With this in place, we now formally define the finite powerset domain:

▶ **Definition 9** (Finite Powerset Domain $\mathcal{VC}^\sharp$). *Let* $\mathcal{VC}^\sharp := \langle \mathcal{P}_{fn(k)}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp, \sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}), \sqsubseteq_{\mathcal{VC}^\sharp}, \bot_{\mathcal{VC}^\sharp}, \top_{\mathcal{VC}^\sharp}, \oplus_{\mathcal{VC}^\sharp}, \sqcap_{\mathcal{VC}^\sharp} \rangle$ *denote the finite powerset domain. Here,* $\bot_{\mathcal{VC}^\sharp} = \emptyset$ *and* $\top_{\mathcal{VC}^\sharp} = \{\top_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}\}$. $\sqsubseteq_{\mathcal{VC}^\sharp}$ *is defined as:* $S \sqsubseteq_{\mathcal{VC}^\sharp} S' \iff \forall s \in S : \exists s' \in S'.s \sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s'$*, as in [3].* $S \oplus_{\mathcal{VC}^\sharp} S'$ *is defined to be* $\Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(S \cup S')$.

$\mathcal{VC}^\sharp$ *is related to the concrete domain* $\mathcal{P}(V \to \mathbb{Z}) \times \mathcal{P}(\mathcal{C})$*, with the following concretization function:* $\gamma : \mathcal{VC}^\sharp \to (\mathcal{P}(V \to \mathbb{Z}) \times \mathcal{P}(\mathcal{C}))$*, where* $\gamma(S) \mapsto \bigcup\{(v, c) \in \gamma_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(s) \mid s \in S\}$.

In summary, Definition 9 states that the lifted abstract domain $\mathcal{VC}^\sharp$ consists of sets of elements of $\mathcal{V}^\sharp \times \mathcal{C}^\sharp$, where $S \in \mathcal{VC}^\sharp$ is lower than $S' \in \mathcal{VC}^\sharp$ w.r.t. the partial order $\sqsubseteq_{\mathcal{VC}^\sharp}$ if every element in S is subsumed by some element in S', according to $\sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}$. $\mathcal{VC}^\sharp$ relates to the concrete domain via the concretization function γ that takes an abstract element S and returns the union of the concretization of each element of S w.r.t. $\mathcal{V}^\sharp \times \mathcal{C}^\sharp$.

We now introduce each of the necessary domain operations for $\mathcal{VC}^\sharp$. We begin by introducing the lifted versions of the abstract transfer functions and the meet operator, and relegate join to its own subsection.

Abstract Transfer Functions. For both instructions that impact program values as well as the abstract cache state, we lift the application of the transfer function to be element-wise. Let $s[C]$ and $s[V]$ denote the cache abstraction and value abstraction components, respectively. Let $T_{\mathcal{V}^\sharp}$ be a transfer function that affects the abstract state corresponding to the program values. Then, the lifted version for $\mathcal{VC}^\sharp$ is a function such that $S \mapsto \Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}\{(T_{\mathcal{V}^\sharp}(s[V]), s[C]) \mid s \in S\}$. Similarly, let $T_{\mathcal{C}^\sharp}$ be a transfer function that affects the part of the abstract state corresponding to the abstract cache state. Then, the lifted version for $\mathcal{VC}^\sharp$ is a function such that $S \mapsto \Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}\{(s[V], T_{\mathcal{C}^\sharp}(s[C])) \mid s \in S\}$.

The Meet Operator. Meet is defined by taking the pairwise meet w.r.t. $\sqcap_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}$. Specifically, if $S, S' \in \mathcal{VC}^\sharp$, then $S \sqcap_{\mathcal{VC}^\sharp} S'$ is defined as $\Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(\{s \sqcap_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s' \mid s \in S, s' \in S'\})$. We note that this set may be larger than k. In this case, we can view the meet operator as replacing Line 1 of the algorithm for join (Algorithm 1) with $\Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(\{s \sqcap_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s' \mid s \in S, s' \in S'\})$. The justification for the validity of the meet operator is in the appendix of the extended version [15].

Now, in the next section, we introduce our join operator.

4.2.3 The Join Operator

We now present the abstract **join** operator, which is the key novelty of our technique. In effect, this will replace the role of $\oplus_{\mathcal{VC}^\sharp}$ in Definition 9 to ensure that the number of disjuncts remains at most k when the join operator is applied by the analysis. Typically, deciding how to maintain and manage the disjunctive components in techniques like trace-partitioning [23], disjunctive completion [10], and the finite powerset framework is a key challenge in effectively implementing these techniques. In order to do so, we first consider when it is necessary to maintain certain disjuncts to prevent spurious aging:

▶ **Example 10.** Consider the two following abstract cache states for a fully-associative cache (all blocks map to one cache set) which can store four memory blocks (associativity = 4): $C = \{d \mapsto \{0, 1\}, b \mapsto \{1, 2, 4\}, c \mapsto \{0, 1, 2\}, a \mapsto \{3\}\}$ (green) and $C' = \{d \mapsto \{0\}, b \mapsto$

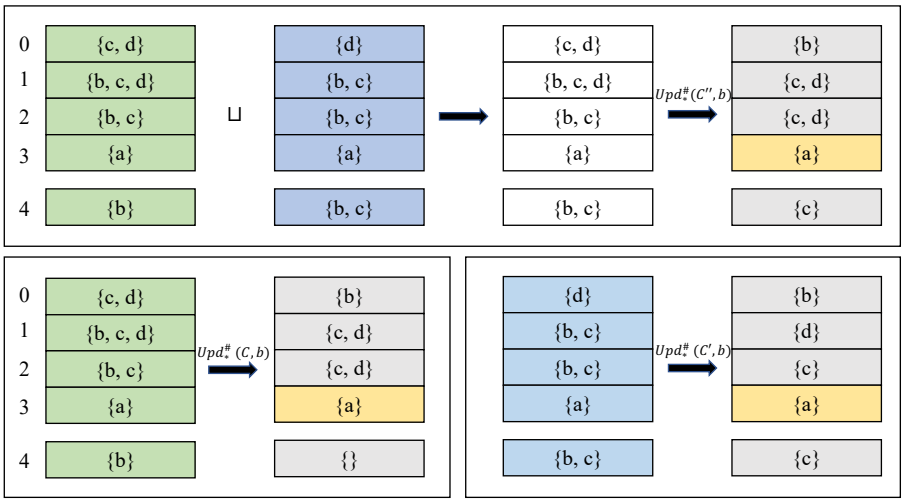


Figure 7 Merging two cache states does not cause spurious aging.

$\{1,2,4\}, c \mapsto \{1,2,4\}, a \mapsto \{3\}\}$ (blue), depicted in Figure 7. We can see that upon an access to variable b, $Upd^{\sharp}_{*}$ will not increment $3 \in C(a)$ to 4, meaning that a is definitely in the cache.

We can also see that this is true for their abstract join $C'' = C \sqcup_{\mathcal{C}^{\sharp}} C' = \{d \mapsto \{0,1\}, b \mapsto \{1,2,4\}, c \mapsto \{0,1,2,4\}, a \mapsto \{3\}\}$, where $4 \notin Upd^{\sharp}_{*}(C'', b)(a)$. In this example, we can see that merging the blue and green cache states did not impact the ability of $Upd^{\sharp}_{*}$ to prevent spuriously aging a from 3 to 4. Despite neither abstract state being subsumed by the other, the reason why $Upd^{\sharp}_{*}$ is able to prevent spurious aging on the union of both states is that the set of concrete cache states represented by the blue abstract state is a subset of the concrete states represented by the green state. This, combined with the fact that $Upd^{\sharp}_{*}$ prevents a from spuriously aging in either state, means that the same holds for the joined state.

The scenario referred to in Example 10, is a sufficient, but not necessary condition. To see why, consider another example, in which the set of valid concrete states of C and C' do not subsume one another, as in the following example:

▶ **Example 11.** Consider the two following abstract cache states: $C = \{d \mapsto \{0,1\}, b \mapsto \{1,2,4\}, c \mapsto \{0,2\}, a \mapsto \{2,3\}\}$ and $C' = \{d \mapsto \{0,1\}, b \mapsto \{0,2,4\}, c \mapsto \{1,3\}, a \mapsto \{2,3\}\}$. We can see that the set of concrete states represented by C and C' are not subsumed by one-another. (For example, $\{c \mapsto 0, d \mapsto 1, b \mapsto 2, a \mapsto 3\} \in \gamma_{\mathcal{C}^{\sharp}}(C)$, but is not in $\gamma_{\mathcal{C}^{\sharp}}(C')$ and $\{d \mapsto 0, c \mapsto 1, b \mapsto 2, a \mapsto 3\} \in \gamma_{\mathcal{C}^{\sharp}}(C')$, but is not in $\gamma_{\mathcal{C}^{\sharp}}(C)$.) However, in their abstract join, $C'' = \{d \mapsto \{0,1\}, b \mapsto \{0,1,2,4\}, c \mapsto \{0,1,2,3\}, a \mapsto \{2,3\}\}$, the following concrete state becomes possible: $\{b \mapsto 0, d \mapsto 1, c \mapsto 2, a \mapsto 3\}$, which is not a valid cache state in either C or C'. But, $Upd^{\sharp}_{*}(C'', b)$ is still able to avoid spuriously aging a from 3 to 4.

The scenarios described by Example 10 and Example 11 are in contrast with the example in Figure 5. In the Figure 5 example, a concrete cache state which is not possible in either cache state becomes possible in their abstraction union, and a memory block was spuriously aged as a result. In the case of Example 10 no new (valid) concrete cache state is introduced by the result of the join of the two abstract states. However, in Example 11, a new concrete cache state is introduced by the result of their join, but spurious aging was prevented. Having such a wide range of possibilities motivates the search for understanding when to merge abstract cache states, and when not to. If computational resources were no limit, only

merging abstract cache states such that $\gamma_{\mathcal{C}^\sharp}$ is *distributive* over the two cache states is ideal, meaning that no infeasible concrete cache states will be introduced. However, this is not applicable in practice due to being too costly, for two reasons. The first is that the number of disjuncts needed to be maintained may be very large (perhaps infinite in certain cases, depending on the control flow of the program, taking us out of the scope of the finite powerset construction), and the second is that checking the abstract states by concretizing them each time may lead to a large computational overhead.

Therefore, instead, we aim to maintain a reasonable number of disjunctions while retaining some precision, by carefully merging abstract states. To do so, we introduce our join operator to replace $\oplus_{\mathcal{VC}^\sharp}$ in Definition 9.

The Algorithm for Join. The abstract join is parameterized by the maximum number of disjuncts allowed, as well as a similarity relation $\sim_R$, to merge states when the number of allowed disjuncts is exceeded. The similarity relation takes in two states $s, s' \in \mathcal{V}^\sharp \times \mathcal{C}^\sharp$ and returns true or false, depending on whether they should be merged.

Algorithm 1 Join Operation $\oplus^*_{\mathcal{VC}^\sharp}\langle k : \mathbb{N}^+, \sim_R : (\mathcal{V}^\sharp \times \mathcal{C}^\sharp) \times (\mathcal{V}^\sharp \times \mathcal{C}^\sharp) \to \{\texttt{tt}, \texttt{ff}\}\rangle$.

Input: $S, S' \in \mathcal{VC}^\sharp$ // $\oplus^*_{\mathcal{VC}^\sharp} : \mathcal{VC}^\sharp \times \mathcal{VC}^\sharp \to \mathcal{VC}^\sharp$
Output: Joined state set S''
1. Set $S'' := \Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(S \cup S')$
2. **if** $|S''| \leq k$ **then**
   return S''
3. **else**
   a. Set $S'' := \Omega_R^{\mathcal{C}^\sharp}(S'')$
   b. **if** $|S''| \leq k$ **then**
      return $\Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(S'')$
   c. **else**
      i. **while** $|S''| > k$ **and** $\exists s_1, s_2 \in S'' : s_1 \sim_R s_2$ **do**
         Merge states s_1 and s_2 where $s_1 \sim_R s_2$
      ii. **while** $|S''| > k$ **do**
         Arbitrarily merge any pair of states
      iii. return $\Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(S'')$

The join operator begins by combining the disjuncts in S, S' and removes redundant elements w.r.t. the value and cache abstractions. If, after doing this step, the cardinality of the resulting set is less than or equal to k, we stop. This choice is motivated by the fact that preserving disjunctive information that differs on the value domain may lead to more precise control-flow information, and thus, possibly result in fewer spurious memory accesses. Otherwise, the join operator aims to merge elements which share the same abstract cache states, using $\Omega_R^{\mathcal{C}^\sharp}$. After this, the subsumption operator is applied to ensure non-redundancy. If the number of disjuncts are within limit, the join operator returns the resulting abstract state. However, if the number of disjuncts still exceeds the number of those which are allowed, the join operator merges pairs of states which satisfy $\sim_R$. Merging via $\sim_R$ is done as much as possible until the number of disjuncts remaining are at most k. If all states satisfying $\sim_R$ have been merged pairwise and the number of disjuncts exceeds k, then states are merged arbitrarily pairwise, until the number of disjuncts is at most k. After this is complete, the subsumption operator is applied to ensure non-redundancy. We show that the join operator is valid in the appendix of the extended version [15]. In the next section, we discuss the possibilities for $\sim_R$.

4.2.4 The Merging Strategies

Recall that in Example 11, the concrete cache state $\{b \mapsto 0, d \mapsto 1, c \mapsto 2, a \mapsto 3\}$ is captured by the abstract cache state C'', but not by C or C', but we can still prevent the spurious aging of a from 3 to 4 by applying $Upd^\sharp_*$ to the abstract state C''. The key factor in $Upd^\sharp_*(C'', b)$ being able to avoid spuriously aging a from 3 to 4 is that the set of variables younger than age 3 (excluding a) is the same across C, C', C'' – the set being $\{d, b, c\}$. This corresponds with the second condition from the definition of $O_>\langle w\rangle$ in $Upd^\sharp_*$. Thus, it is of interest to preserve that property whenever we can.

To this end, we consider a condition under which merging two abstract states preserves the ability of $Upd^\sharp_*$ to prevent spurious aging of a given memory block when accessing some memory block w on the two abstract states separately. Specifically, we consider when two abstract cache states C and C' can be merged such that if $Upd^\sharp_*(C, w)$ and $Upd^\sharp_*(C', w)$ do not age a memory block b from t to $t+1$, then $Upd^\sharp_*(C \sqcup_{\mathcal{C}^\sharp} C', w)$ does not age memory block b from t to $t+1$, where t is some possible cache line age between 1 and $n-1$. If they satisfy the property that the set of memory blocks who have ages younger than t are the same in C and C', then if $Upd^\sharp_*(C, w)$ and $Upd^\sharp_*(C', w)$ do not spuriously age block b from t to $t+1$, then $Upd^\sharp_*(C'', w)$ where $C'' = C \sqcup_{\mathcal{C}^\sharp} C'$ does not age block b from t to $t+1$ spuriously.

We first introduce a helper function used in the proceeding Lemma that formalizes the aforementioned property. Let $\max_{<t}(A)$ be a function that takes a set of integers (A), and returns the maximum value that is less than t. If no such value exists, it returns $-\infty$. For example, $\max_{<4}(\{1,2,3,4\})$ returns 3, while $\max_{<5}(\{5,6,7\})$ returns $-\infty$. We now state the Lemma:

▶ **Lemma 12.** *Let $C, C' \in \mathcal{C}^\sharp$. Let $w \in \mathcal{B}$ be the memory block being accessed. Let $t \in \{1, ..., n-1\}$ be some cache line age, where n is the associativity of the cache. If for each $b \in \mathcal{B}$, $\max_{<t}(C(b)) = \max_{<t}(C'(b))(\star)$, then, if $Upd^\sharp_*(C, w)$ and $Upd^\sharp_*(C', w)$ do not age b from t to $t+1$, then $Upd^\sharp_*(C \sqcup_{\mathcal{C}^\sharp} C', w)$ does not age b from t to $t+1$.*

Proof. For the proof, please refer to the appendix of the extended version [15]. ◀

Lemma 12 yields two key corollaries for our purposes. The first states that if the universe of concrete cache states are partitioned using a set of abstract cache states based on groups of states which satisfy $(\star)$ pairwise, then there will be no spurious aging of memory blocks caused by combining states with the abstract join operator. The second one states a special case of the lemma, which prevents memory blocks from being spuriously uncached as a result of combining states with the abstract join operator. (We note that there is still possible imprecision due to the gap between $Upd^\sharp_*$ and the best abstract transformer.)

▶ **Corollary 13.** *Merging abstract cache states based on the strategy of only merging abstract cache states C, C' that satisfy the following formula $\forall t \in \{1, ..., n-1\}.\forall b \in \mathcal{B}. \max_{<t}(C(b)) = \max_{<t}(C'(b))$ will result in no block being spuriously aged, purely due to combining abstract cache states with the join operator.*

Proof. If only pairs of disjuncts which satisfy this property are merged, then for any block $b \in \mathcal{B}$, if b is not aged in either disjunct, then it will not be aged in their abstract union. This corresponds to the case where there is no precision loss due to using the $\mathcal{C}^\sharp$ abstract domain compared to $\mathcal{P}(\mathcal{C}^\sharp)$. ◀

▶ **Corollary 14.** *If $t = n-1$, then merging based on the aforementioned strategy will prevent a memory block from becoming possibly uncached as a result of merging two abstract states.*

Proof. This is just a special case of Lemma 12, where $t = n-1$. ◀

Egli-Milner Partial Order $\sqsubseteq_{EM}$ ($\sqsubseteq_{EM}$: $\mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp) \times \mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp) \to \{\texttt{tt}, \texttt{ff}\}$). The Egli-Milner partial order is defined as follows: $S \sqsubseteq_{EM} S' \iff S = \emptyset \vee (\forall s \in S \,.\, \exists s' \in S' \,.\, s \sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s' \wedge \forall s' \in S'.\exists s \in S.s \sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s')$.

k-Collapsor ($\Uparrow_k$: $\mathcal{P}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp) \to \mathcal{VC}^\sharp$). Given $S \in \mathcal{VC}^\sharp$ such that S is non-redunandant according to $\sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}$, a k-Collapsor $\Uparrow_k (S)$ yields $S' \in \mathcal{VC}^\sharp$ such that $S \sqsubseteq_{EM} S'$ and, moreover, $|S'| \leq k$.

∇-Reduction Map ($\Omega^\nabla : \mathcal{P}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp) \to \mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$). The ∇-Reduction map is defined recursively: $\Omega^\nabla(S) := ite(\exists s, s' \in S \,.\, s \sqsubset_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s', \Omega^\nabla((S \setminus \{s, s'\}) \cup \{s \nabla_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s'\}), S)$.

Widen for $\mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$ ($_k\nabla_P : \mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp) \times \mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp) \to \mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$). Given $S, S' \in \mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$ such that $S \sqsubset_{\mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)} S'$, $S_k \nabla_P S' := \Omega^\nabla(S \cup S'')$, where $S'' := \Uparrow_k (S')$.

Widen for $\mathcal{VC}^\sharp$ ($_k\nabla_P : \mathcal{VC}^\sharp \times \mathcal{VC}^\sharp \to \mathcal{VC}^\sharp$). Given $S, S' \in \mathcal{VC}^\sharp$ such that $S \sqsubset_{\mathcal{VC}^\sharp} S'$, $S_k \nabla_P S'' := \Omega^\nabla(S \cup S'')$, where $S'' := S' = \Uparrow_k (S')$, by virtue of $|S'| \leq k$, as it is a member of $\mathcal{VC}^\sharp$.

Figure 8 Details of the widening operator.

The two corollaries could lead to two different merging strategies to serve as similarity relations in the definition of the abstract join operator. The first being to prevent the spurious aging of any memory block as a result of merging two abstract cache states, and the second being to prevent any block from becoming spuriously uncached in the same scenario. While the two merging strategies have properties that are desirable, they have their limitations in terms of their utility in practice. First, it may require many disjuncts to be able to merge only according to the strategy suggested by Corollary 13. Second, using the strategy suggested by Corollary 14, the number of disjuncts required may be large, but furthermore, the user is forced to pick a specific t $(n-1)$.

Therefore, in practice, we merge two states according to the following similarity relation: $\forall b \in \mathcal{B}. \max_{\neq n}(C(b)) = \max_{\neq n}(C'(b))$. That is, if there is no memory block whose maximum (non-associativity) age differs between C and C', then we merge the two abstract states.

The goal is to encourage falling into either of the two cases where Lemma 12 indicates that the merging will not lead to spurious aging, while still keeping the number of disjunctions manageable by enforcing less stringent requirements than suggested by either Corollary 13 or Corollary 14. Of course, many other relations could be used, including strategies based on the syntax and semantics of the program, which we intend to explore in future work.

4.2.5 The Widening Operator

Finally, the last domain operation to be defined is the widening operator. A widening operator serves to enforce termination of analyses which use abstract domains with infinitely increasing chains, or to speed up the analysis, regardless of the abstract domain. Given that $\mathcal{C}^\sharp$ has finite height, no widening is required for it. However, $\mathcal{V}^\sharp$ abstracts program values and therefore may not be of finite height; thus we must introduce a widening operator for $\mathcal{VC}^\sharp$.

One way to instantiate a widening operator for the finite powerset domain is the through the use of a *cardinality-based* widening [3], which is what we do in our instantiation of the framework. The formal details of the cardinality-based widening (written as slight adaptations from the details in [3]) are shown in Figure 8. In a nutshell, cardinality-based widening ensures termination by first ensuring that the cardinality of the *widening* argument is bounded by a fixed (user-specified) size k. Then, a reduction map, which takes a set of elements and removes and replaces pairs of elements where one strictly subsumes the other by the widening of these two elements, is applied in a recursive manner until the set no longer changes. Together, the two form a ∇-connected extrapolation heuristic [3], which lifts the base-level widening $\nabla_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}$ to the powerset domain, while preventing unbounded growth, guaranteeing termination.

In the case of the finite powerset domain of non-redundant (w.r.t. $\sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}$) sets *without* a restriction on the cardinality sets (whose elements are denoted, with a slight abuse of notation, by $\mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$ in Figure 8), the two steps are accomplished by a using a k-Collapsor ($\Uparrow_k$) and the reduction map Ω^∇. The k-Collapsor takes an element of $\mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$, S, and returns an element $S' \in \mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$, such that $|S'| \leq k$ and $S \sqsubseteq_{EM} S'$. The Elgi-Milner partial order ($S \sqsubseteq_{EM} S'$) means that for two sets S, S', every element in S is overapproximated by an element in S' AND every element in S' overapproximates some element in S. There are many ways to define a k-Collapsor [3], but it is worth noting that our join operator defined in Algorithm 1, can be used to define a k-Collapsor, e.g., $S \mapsto S \oplus^*_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} S$. By construction, the resulting set is of size less than or equal to k. Furthermore, since elements of S are merged, every element in the resulting set subsumes some element in S, enforcing $S \sqsubseteq_{EM} S \oplus^*_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} S$.

In our abstract domain, $\mathcal{VC}^\sharp$, where each set is restricted to be of size at most k, the widening operator for $\mathcal{VC}^\sharp$ can be defined as shown at the bottom of Figure 8. That is, the k-Collapsor is the identity function, as all elements in the domain are bounded by cardinality k. The termination and soundness guarantees follow from the results established in [3].

5 Experiments

We have implemented our method in a static program analyzer designed for quantifying cache side-channel leakage. Our analyzer is written in OCaml and built upon the CacheAudit analysis framework, the state-of-the-art tool for computing upper bounds of cache side-channel leakage via abstract interpretation. Our techniques are implemented as functors for the existing CacheAudit abstract domains, transforming the existing abstract interpreter to gain precision in the key ways we identified.

5.1 Experimental Setup

We conducted all experiments on a computer with an Intel Xeon W-2245 CPU and 128 GB RAM, running Ubuntu 20.04 operating system. The experiments were designed to answer the following questions:

- **RQ1**. Do the upper bounds computed by our method improve upon the state-of-the-art?
- **RQ2**. Do the two innovative techniques presented in Section 4 have a synergistic effect in practice?

Our benchmark consists of 29 C programs that implement a variety of sorting algorithms and cryptographic functions. For every sorting algorithm, we introduce a "structured" version, meaning that the elements of the arrays to be sorted are data structure types, consisting of several other components: character arrays and integers. This set-up reflects real-world applications of algorithms that carry some "informational payload".

Each sorting algorithm was assumed to run on an array of size 24. While loop unrolling is not strictly necessary for abstract interpretation based methods, it was required for certain programs (independent of the technique used), and thus, we allowed a loop unrolling limit of 1024 for those programs, as recommended by the tool. We limit the maximum number of disjunctions to 10.

5.2 Results for Answering RQ1

To answer RQ1, i.e., do the upper bounds computed by our method improve upon the state-of-the-art, we compare the results of our method and the existing method on all 29 benchmark programs. The results are shown in Table 1. Column 1 shows the name of the

Table 1 Comparing existing methods (**B** [11]) and our new method (**NM**) on a 32KB cache with associativity 8 and line size 32.

Program	Leakage Quantification			Time (s)	
	B (SM / DM)	**NM** (SM / DM)	Comp.	**B**	**NM**
bingosort	1.0 / 1.0	0.0 / 0.0	✔	4	18
bingosortstruct	25.0 / 25.0	23.0 / 23.0	✔	101	227
bubblesort_opt	3.0 / 3.0	1.0 / 1.0	✔	0	1
bubblesort_opt_struct	25.0 / 25.0	1.0 / 1.0	✔	2	8
bubblesort_struct	1.0 / 1.0	0.0 / 0.0	✔	2	5
cocktailsort	0.0 / 0.0	0.0 / 0.0	✔	17	45
cocktailsortstruct	1.0 / 1.0	0.0 / 0.0	✔	108	284
gnomesort	2.0 / 2.0	2.0 / 2.0	same	3	9
gnomesortstruct	23.0 / 23.0	19.0 / 19.0	✔	38	75
iterativeheapify	2.0 / 2.0	1.0 / 1.0	✔	11	21
iterativeheapifystruct	22.0 / 22.0	21.0 / 21.0	✔	92	146
odd_even_sort	0.0 / 0.0	0.0 / 0.0	✔	8	20
odd_even_sort_struct	1.0 / 1.0	0.0 / 0.0	✔	54	148
shellsort	0.0 / 0.0	0.0 / 0.0	✔	0	1
shellsortstruct	1.0 / 1.0	1.0 / 1.0	same	1	4
defensive_gather	96.0 / 96.0	1.0 / 1.0	✔	14	38
scatter_gather_openssl_1_0_2	97.0 / 97.0	1.0 / 1.0	✔	2	3
window_mod_exp_libgcrypt_161	2.0 / 2.0	1.5 / 1.5	✔	1	1
window_mod_exp_libgcrypt_163	0.0 / 0.0	0.0 / 0.0	✔	1	1
rabbit	0.0 / 0.0	0.0 / 0.0	✔	4	14
salsa	0.0 / 0.0	0.0 / 0.0	✔	4	10
aes-128-preloading	15.6 / 0.0	14.5 / 0.0	✔	14	51
aes-192-preloading	15.6 / 0.0	15.0 / 0.0	✔	16	82
aes-256-preloading	16.5 / 0.0	16.0 / 0.0	✔	23	123
aes-128-rom	142.5 / 132.7	141.5 / 132.6	✔	23	63
aes-192-rom	142.6 / 132.6	142.1 / 132.6	✔	26	92
aes-256-rom	143.2 / 132.6	142.6 / 132.6	✔	34	114
sosemanuk	64.0 / 64.0	64.0 / 64.0	same	90	191
hc-128	29.0 / 0.0	29.0 / 0.0	same	2242	3853

benchmark program. Columns 2-4 compare leakage quantification results for two types of adversaries: SM stands for the shared-memory adversary and DM stands for the disjoint-memory adversary. In general, the leakage for DM is smaller than or equal to the leakage for SM. In both cases, the quantification results of the existing and new methods are measured in bits – a smaller number means a better result (less leakage). In Column 4, the ✔symbol means that our method obtains a better result, and the ✔symbol means that our method obtains the best-possible result (e.g., when the leakage is already 0). Columns 5-6 compare the total analysis time in seconds. In general, we find that in most cases the running time is about twice as long compared to the baseline methodology. This is expected due to the extra domain operations required due to refining the $\mathcal{C}^\sharp$ domain to use *sets* of abstract states.

Table 1 shows that the new method obtains either better or the best-possible quantification results on 13/15 benchmark programs that implement various sorting algorithms. For example, the quantification results for *cocktailsortstruct* and *shellsort* are the best-possible because

the leakages obtained by our method are equal to 0. On the other 2/15 benchmark programs (*gnomesort* and *shellsortstruct*), the new method obtains quantification results that are as good as those of the existing method. As for the benchmark programs that implement cryptographic algorithms, the new method obtains either better quantification results on 9/14 of them, and obtains the same results as the existing method on 5/14 of them. We note that the results are also dependent on the cache configuration used. For example, using a $32K$ cache with associativity 16, and line size 32, on *sosemanuk* and *hc-128*, in particular, the precision of the leakage improves by close to 10 bits and 13 bits by using our method, corresponding to elimination of 1024 and 8192 spurious cache states, respectively. Overall, the results show that the upper bounds computed by our method improve upon the state-of-the-art significantly.

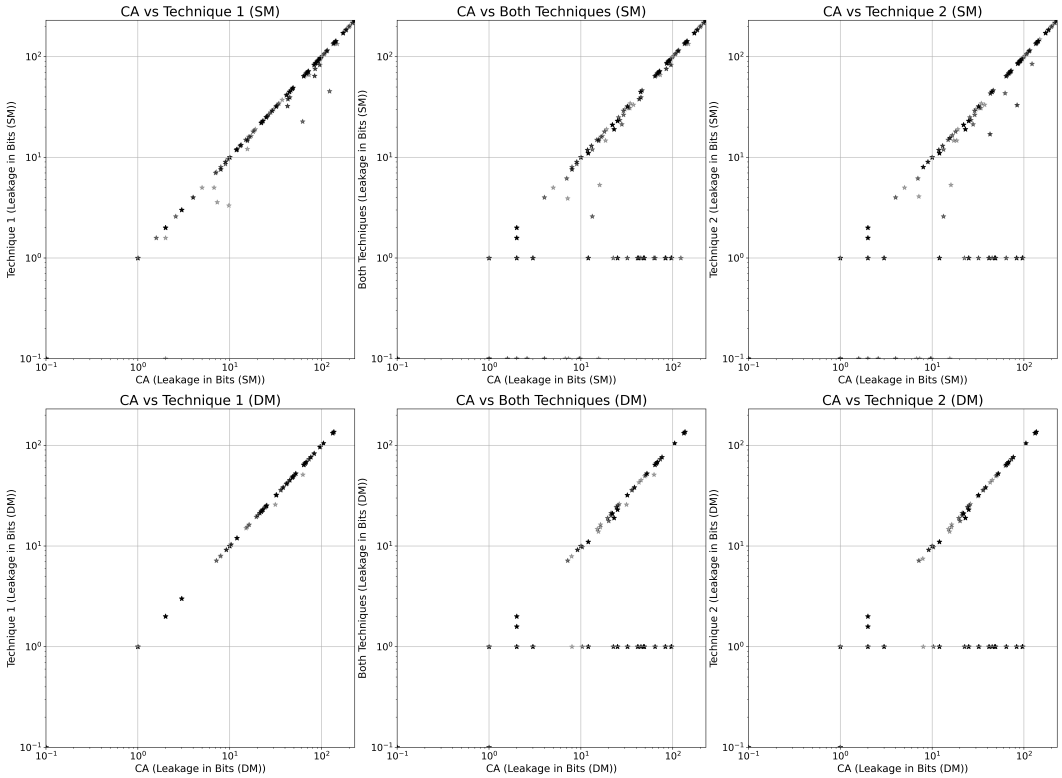


Figure 9 Evaluating the impact of the two new techniques in our method, by comparing them against the existing method. CA is the existing method (CacheAudit), Technique 1 is the first new technique in our method (the new abstract transfer function), Technique 2 is the second new technique in our method (refining the $\mathcal{C}^\sharp$ domain), and Both Techniques is our method with both of the two new techniques. The scatter plots on top are for the SM adversaries, while the scatter plots at the bottom are for the DM adversaries. In all of these scatter plots, points below the diagonal line ($y = x$) are winning cases for our method against the existing method.

5.3 Results for Answering RQ2

To answer RQ2, i.e., do the two techniques presented in Section 4 have a synergistic effect in practice, we conducted an ablation study, by enabling each individual technique and comparing it against the state-of-the-art. These comparisons were conducted on all 29 benchmarks programs, with various cache settings. That is, we set the cache size S to 4KB,

8KB, 16KB, 32KB and 64KB, the associativity n to 4, 8, and 16, and the cache line size L to 32 and 64 bytes. While we recognize that 16 is not a common associativity for real-world caches, our goal was to stress-test our abstract transformer under higher associativities. The results are shown as scatter plots in Figure 9. In each scatter plot, the x-axis is the leakage (in bits) obtained by the state-of-the-art method, and the y-axis is the leakage (in bits) obtained by our method (with one or both techniques enabled). Thus, the diagonal line represents the cases where our method is tied with the existing method, whereas points below the diagonal line are winning cases for our method.

In Figure 9, the two scatter plots on the left-hand side show the effectiveness of the new abstract transfer function. While most of the points are on the diagonal line, meaning that the two methods are tied, there are some points that are significantly below the diagonal line, indicating the effectiveness of the proposed technique for these cases. The two scatter plots on the right-hand side show the effectiveness of using disjunctions. Many of the points are below the diagonal line, which are the winning cases for our method. The two scatter plots in the middle show that using both the new abstract transfer function and leveraging disjunctions together perform very well, with even more points below the diagonal line.

Table 2 Evaluating the impact of the two new techniques in our method on the benchmark program *scatter_gather_openssl_1_0_2* using various cache settings. S is the cache size in bytes, n is the associativity level, and L is the cache line size in bytes.

Cache Setting (S, n, L)	Leakage Quantification			Time (s)		
	Technique-1 (SM / DM)	Ours (both) (SM / DM)	Technique-2 (SM / DM)	Tech-1	Ours (both)	Tech-2
(4096, 4, 32)	64.3 / 64.3	1.0 / 1.0	33.0 / 1.0	5	6	3
(4096, 4, 64)	32.3 / 32.3	1.0 / 1.0	17.0 / 1.0	3	3	3
(4096, 8, 32)	45.2 / 45.1	1.0 / 1.0	84.7 / 1.0	8	9	3
(4096, 8, 64)	22.6 / 22.6	1.0 / 1.0	43.3 / 1.0	5	5	4
(8192, 4, 32)	83.3 / 83.3	1.0 / 1.0	1.0 / 1.0	3	4	2
(8192, 4, 64)	41.9 / 41.9	1.0 / 1.0	1.0 / 1.0	2	2	2
(8192, 8, 32)	64.3 / 64.3	1.0 / 1.0	33.0 / 1.0	5	5	2
(8192, 8, 64)	32.3 / 32.3	1.0 / 1.0	17.0 / 1.0	3	3	3
(16384, 4, 32)	97.0 / 97.0	1.0 / 1.0	1.0 / 1.0	3	4	3
(16384, 4, 64)	49.0 / 49.0	1.0 / 1.0	1.0 / 1.0	2	2	3
(16384, 8, 32)	83.3 / 83.3	1.0 / 1.0	1.0 / 1.0	3	4	2
(16384, 8, 64)	41.9 / 41.9	1.0 / 1.0	1.0 / 1.0	2	2	3
(32768, 4, 32)	97.0 / 97.0	1.0 / 1.0	1.0 / 1.0	4	6	2
(32768, 4, 64)	49.0 / 49.0	1.0 / 1.0	1.0 / 1.0	2	4	1
(32768, 8, 32)	97.0 / 97.0	1.0 / 1.0	1.0 / 1.0	3	3	3
(32768, 8, 64)	49.0 / 49.0	1.0 / 1.0	1.0 / 1.0	2	3	2
(64512, 4, 32)	97.0 / 97.0	1.0 / 1.0	1.0 / 1.0	5	5	2
(64512, 4, 64)	49.0 / 49.0	1.0 / 1.0	1.0 / 1.0	3	3	3
(64512, 8, 32)	97.0 / 97.0	1.0 / 1.0	1.0 / 1.0	4	5	2
(64512, 8, 64)	49.0 / 49.0	1.0 / 1.0	1.0 / 1.0	2	4	1

We also collected detailed results of the experimental comparison, which are shown in Table 2. Various cache settings were used in the experiments, as shown in Column 2. For brevity, we only show these detailed results for a representative benchmark program named *scatter_gather*.

Overall, the results show that each of the two techniques is effective in isolation; furthermore, when the two techniques are used together, they often have a synergistic effect in terms of improving the precision of leakage quantification.

6 Related Work

As mentioned earlier, the most closely related work is that of Doychev et al. [11], which we regard as the baseline algorithm for quantifying cache side-channel leakage. The key difference in our work is a new abstract transfer function and a disjunctive refinement for increasing the precision of abstract interpretation.

Doychev et al. [11] support other adversaries, including trace-based and timing adversaries. Given that our abstractions fundamentally improve the precision of the abstract cache states in an abstract domain that is specialized for quantification, we expect that our techniques will help improve quantification results on downstream static analyses that rely on abstract cache states.

Kopf et al. [17] target cache-based adversaries, and conduct quantification via counting formulae, combined with an abstract interpreted-based static analysis. They recognize that trace partitioning [23] during the static analysis led to increased precision in the quantification results. However, trace-partitioning was conducted by manual program transformation, whereas our method is automated via abstract interpretation to parsimoniously leverage disjunctive information. Beyond abstract interpretation, which is a *sound* analysis technique, there are methods based on alternative analysis techniques such as bounded modeling checking [20] or symbolic execution [7]. However, their results may not be sound.

There are also methods targeting other kinds of adversaries. In the case of trace-based adversaries, it is assumed that a malicious attacker may observe the sequences of memory accesses throughout program execution; thus, quantification techniques aim to compute an upper bound on the number of distinct memory access traces possible. Various tools have been developed to compute an upper bound for the number of possible distinct memory access traces. Ma et al. [20] introduce an abstraction known as differential set that tracks, for each memory access, all possible addresses that might be accessed by that operation or its "sibling" operations in other control flows. Their abstraction is combined with bounded model-counting to compute a sound upper bound of information leakage. Other works leverage techniques such as symbolic execution to compute these upper bounds.

Beyond quantification, there are methods for cache hit/miss classification [26, 25, 14]. In particular, Touzeau et al. [25] combine abstract interpretation with model checking to classify memory accesses in LRU caches as "always hit", "always miss", or "definitely unknown". Touzeau et al. [26] also introduce a method that represents cache states using anti-chains of minimal/maximal elements rather than full state sets, thus enabling efficient computation while preserving precision. Gysi et al. [14] introduce symbolic techniques to count cache misses without having to enumerate all memory accesses, making the analysis practical through a hybrid approach that combines symbolic computation with selective enumeration. However, these works are not designed for quantifying cache side-channel leakage.

7 Conclusion

We have presented a method for significantly improving the precision of abstract interpretation based static analysis for quantifying cache side-channel leakage. The method uses a new abstract transfer function to prevent spurious aging and abstract domain refinement during the analysis step, which uses disjunctions parsimoniously to prevent spurious combinations of cache states. Our experimental evaluation on benchmark programs consisting of sorting

and cryptographic algorithms shows that the method is more accurate in quantifying cache side-channel leakage than the state-of-the-art technique. Furthermore, both of the two new techniques in our method contribute to the performance improvement.

References

1 Onur Aciiçmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In Luca Breveglieri, Shay Gueron, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *Fourth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2007, FDTC 2007: Vienna, Austria, 10 September 2007*, pages 80–91. IEEE Computer Society, 2007. doi:10.1109/FDTC.2007.4318988.

2 José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 53–70. USENIX Association, 2016. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida.

3 Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Widening operators for powerset domains. *Int. J. Softw. Tools Technol. Transf.*, 9(3-4):413–414, 2007. doi:10.1007/S10009-007-0029-Y.

4 Sandrine Blazy, David Pichardie, and Alix Trieu. Verifying constant-time implementations by abstract interpretation. *J. Comput. Secur.*, 27(1):137–163, 2019. doi:10.3233/JCS-181136.

5 Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In William Enck and Collin Mulliner, editors, *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017*. USENIX Association, 2017. URL: https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser.

6 Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*, pages 4:1–4:6. ACM, 2017. doi:10.1145/3152701.3152706.

7 Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. Quantifying the information leakage in cache attacks via symbolic execution. *ACM Trans. Embed. Comput. Syst.*, 18(1):7:1–7:27, 2019. doi:10.1145/3288758.

8 Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *30th IEEE Symposium on Security and Privacy (SP 2009), 17-20 May 2009, Oakland, California, USA*, pages 45–60. IEEE Computer Society, 2009. doi:10.1109/SP.2009.19.

9 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.

10 Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 269–282. ACM Press, 1979. doi:10.1145/567752.567778.

11 Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In Samuel T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 431–446. USENIX Association, 2013. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/doychev.

12 Goran Doychev and Boris Köpf. Rigorous analysis of software countermeasures against cache attacks. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 406–421. ACM, 2017. doi:10.1145/3062341.3062388.

13 David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *32nd IEEE Symposium on Security and Privacy, SP 2011, 22-25 May 2011, Berkeley, California, USA*, pages 490–505. IEEE Computer Society, 2011. doi:10.1109/SP.2011.22.

14 Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefler. A fast analytical model of fully associative caches. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 816–829. ACM, 2019. doi:10.1145/3314221.3314606.

15 Chao Wang Jacqueline Mitchell. Quantifying cache side-channel leakage by refining set-based abstractions (extended version). https://github.com/jlmitche23/ecoop25CacheQuantification, 2025.

16 Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996. doi:10.1007/3-540-68697-5_9.

17 Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 564–580. Springer, 2012. doi:10.1007/978-3-642-31424-7_40.

18 Robert Kotcher, Yutong Pei, Pranjal Jumde, and Collin Jackson. Cross-origin pixel stealing: timing attacks using CSS filters. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1055–1062. ACM, 2013. doi:10.1145/2508859.2516712.

19 Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 605–622. IEEE Computer Society, 2015. doi:10.1109/SP.2015.43.

20 Cong Ma, Dinghao Wu, Gang Tan, Mahmut Taylan Kandemir, and Danfeng Zhang. Quantifying and mitigating cache side channel leakage with differential set. *Proc. ACM Program. Lang.*, 7(OOPSLA2):1470–1498, 2023. doi:10.1145/3622850.

21 Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006. doi:10.1007/11605805_1.

22 Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 199–212. ACM, 2009. doi:10.1145/1653662.1653687.

23 Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5):26, 2007. doi:10.1145/1275497.1275501.

24 Isabell Schmitt and Sebastian Schinzel. Waffle: Fingerprinting filter rules of web application firewalls. In Elie Bursztein and Thomas Dullien, editors, *6th USENIX Workshop on Offensive Technologies, WOOT'12, August 6-7, 2012, Bellevue, WA, USA, Proceedings*, pages 34–40. USENIX Association, 2012. URL: `http://www.usenix.org/conference/woot12/waffle-fingerprinting-filter-rules-web-application-firewalls`.

25 Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. Ascertaining uncertainty for efficient exact cache analysis. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 22–40. Springer, 2017. `doi:10.1007/978-3-319-63390-9_2`.

26 Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. Fast and exact analysis for LRU caches. *Proc. ACM Program. Lang.*, 3(POPL):54:1–54:29, 2019. `doi:10.1145/3290367`.

27 Tom van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 1382–1393. ACM, 2015. `doi:10.1145/2810103.2813632`.

28 Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. STACCO: differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 859–874. ACM, 2017. `doi:10.1145/3133956.3134016`.

29 Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time RSA. *J. Cryptogr. Eng.*, 7(2):99–112, 2017. `doi:10.1007/S13389-017-0152-Y`.

30 Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 305–316. ACM, 2012. `doi:10.1145/2382196.2382230`.

Wastrumentation: Portable WebAssembly Dynamic Analysis with Support for Intercession

Aäron Munsters
Vrije Universiteit Brussel, Brussels, Belgium

Angel Luis Scull Pupo
Vrije Universiteit Brussel, Brussels, Belgium

Elisa Gonzalez Boix
Vrije Universiteit Brussel, Brussels, Belgium

Abstract

Dynamic program analyses help in understanding a program's runtime behavior and detect issues related to security, program comprehension, or profiling. Instrumentation platforms aid analysis developers by offering a high-level API to write the analysis, and inserting the analysis into the target program. However, current instrumentation platforms for WebAssembly (Wasm) restrict analysis portability because they require concrete runtime environments. Moreover, their analysis API only allows the development of analyses that observe the target program but cannot modify it. As a result, many popular dynamic analyses present for other languages, such as runtime hardening, virtual patching or runtime optimization, cannot currently be implemented for Wasm atop a dynamic analysis platform. Instead, they need to be built manually, which requires knowledge of low-level details of the Wasm's semantics and instruction set, and how to safely manipulate it.

This paper introduces Wastrumentation, the first dynamic analysis platform for WebAssembly that supports intercession. Our solution, based on source code instrumentation, weaves the analysis code directly into the target program code. Inlining the analysis into the target's source code avoids dependencies on the runtime environment, making analyses portable across Wasm VMs. Moreover, it enables the implementation of analyses in *any* Wasm-compatible language. We evaluate our solution in two ways. First, we compare it against a state-of-the-art source code instrumentation platform using the WasmR3 benchmarks. The results show improved memory consumption and competitive performance overhead. Second, we develop an extensive portfolio of dynamic analyses, including novel analyses previously unattainable with source code instrumentation platforms, such as memoization, safe heap access, and the removal of NaN non-determinism.

2012 ACM Subject Classification Software and its engineering → Object oriented frameworks; Software and its engineering → Dynamic analysis; Information systems → Web applications; Security and privacy → Information flow control

Keywords and phrases WebAssembly, dynamic analysis, instrumentation platform, intercession

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.23

Funding *Aäron Munsters*: Funded by the Research Foundation Flanders, project number 1S53725N. *Angel Luis Scull Pupo*: Funded by the Cybersecurity Research Program Flanders (CRPF) from the Flemish Government.

1 Introduction

WebAssembly (Wasm) [13] has become a compilation target for many high-level languages such as C, C++, and Rust, thanks to its portability and predictable performance [31]. It was originally designed as a binary instruction set for the web and is currently supported by all major browsers.

Wasm offers predictable performance through its compact program representation, enabling efficient loading, validation and execution as parts of the program are being loaded. To safely execute untrusted code, it features control-flow integrity (ensuring programs cannot

© Aäron Munsters, Angel Luis Scull Pupo, and Elisa Gonzalez Boix;
licensed under Creative Commons License CC-BY 4.0

jump to unintended code locations), a modular system with strict encapsulation and type safety, and a sandboxed linear memory layout that isolates data from code and prevents unauthorized memory access [13].

Although Wasm was designed with built-in security and efficient mechanisms, it does not inherently address questions of software quality. For example, recent research reports performance issues in server-side Wasm applications [14] and memory safety issues [18, 23]. Dynamic analysis tools are increasingly being leveraged for Wasm to enhance security [9, 2], improve program comprehension [20, 30, 26, 39] and enable profiling [21].

To implement dynamic analyses, developers often rely on code transformation tools or general-purpose analysis platforms. In the context of Wasm, transformation tools like Brewasm [5] and WasmManipulator [29] allow developers to instrument the target program code by injecting additional Wasm instructions implementing the analysis. However, this approach requires a deep understanding of the instruction set and how it can be manipulated, such that the injected instructions uphold type safety and do not unintentionally interfere with the target program.

Alternatively, developers can use a general-purpose dynamic analysis framework for Wasm, either Wizard [38], or Wasabi [19]. These frameworks provide developers with a high-level API to implement analyses while avoiding unintentional analysis interference and ensuring type-safe instrumentation. Wizard's instrumentation framework is implemented within the Wizard execution engine. The integration at the level of the execution engine allows the instrumentation code to be turned on and off at runtime. Wasabi on the other hand instruments the target program with hooks that call into a JavaScript analysis which co-exists with the instrumented input program. Unfortunately, analyses implemented with these frameworks are restricted to concrete runtime environments. Wizard's instrumentation executes only on top of the Wizard engine, while Wasabi requires a JavaScript Virtual Machine (VM) along the Wasm VM. Moreover, analyses implemented with Wizard and Wasabi can only observe a program execution but not modify it, i.e. they do not support intercession. As a result, a wide range of dynamic analyses cannot be implemented with these frameworks, including runtime hardening [27, 18], virtual patching [34, 42], or runtime optimization [34, 43].

This paper presents Wastrumentation, the first general-purpose framework for dynamic analysis of WebAssembly that supports intercession and is portable among the many Wasm environments through its design as a source code instrumentation platform. To avoid the requirement of a JavaScript VM to be present for the instrumentation, Wastrumentation exposes its instrumentation API as a Wasm application binary interface (ABI). Furthermore, targeting a Wasm ABI has the benefit that any language that compiles to Wasm can be used to implement the analysis. Wastrumentation then merges both the target program and the analysis implementation as one Wasm program, making the instrumentation portable to any Wasm VM. To enable the construction of dynamic analyses requiring intercession, the instrumentation API of Wastrumentation enables the analysis to alter or skip operations happening at the target program side.

In summary, the main contributions of the paper are:

- We develop a novel instrumentation approach for Wasm programs to deploy dynamic analyses through source code rewriting featuring an instrumentation ABI akin to state-of-the-art high-level APIs but language-agnostic and extended with support for intercession.
- We show the usefulness of Wastrumentation by building an extensive portfolio of dynamic analyses, including a variety of existing analyses from Wasabi, as well as the implementation of three novel dynamic analyses that require intercession support, previously

unattainable with existing source code instrumentation platforms for Wasm. These analyses cover the domains of runtime optimization (memoization analysis), security (safe-heap analysis) and program repair ("DeNaN" analysis).
- We conducted a performance evaluation of Wastrumentation for the developed dynamic analyses on real-world programs from the Wasm-R3 benchmark suite [1], which shows improved memory consumption and competitive performance overhead compared to the state-of-the-art source code instrumentation platform, Wasabi [19].

2 Approach

This section describes our source code instrumentation platform for dynamic analysing Wasm applications. We first introduce Wastrumentation from the point of view of the analysis developer. Section 2.1 shows how to build a dynamic analysis atop of Wastrumentation, and Section 2.2 details the analysis API. Then, we turn our attention to the platform itself. Section 2.3 shows its architecture and gives an overview of the design choices. Section 2.4 details the transformations at the Wasm level where the instrumentation code is injected.

2.1 Developing a Dynamic Analysis with Wastrumentation

We now explain Wastrumentation from the perspective of an analysis developer implementing a concrete dynamic analysis. To this end, we present the "DeNaN" dynamic analysis, a novel dynamic analysis inspired by the "`--denan`" pass from the Wasm compiler framework Binaryen [10]. The "DeNaN" analysis ensures determinism among different Wasm VMs regarding floating point operations, as the Wasm language specification for "NaN Propagation"[1] does not enforce a canonical floating-point NaN representation. To address this, the "DeNaN" analysis replaces all runtime floating-point NaN values with a deterministic value 0.0.

To implement the "DeNaN" analysis, all Wasm instructions that may operate on runtime floating-point values must be intercepted. For each intercepted instruction, the analysis distinguishes between integer types (`i32`, `i64`), which are unaffected by NaN propagation, and floating-point type (`f32`, `f64`). Integer values remain unchanged, while floating-point NaN values must be replaced with 0.0. The relevant Wasm instructions include:
- Stack operations: `const`, pushes a constant on the value stack.
- Local and global variable operations: `local.get`, `local.set`, `global.get`, `global.set`, a "get" reads from a variable or global and pushed onto the value stack, a "set" does the opposite.
- Memory operations: `load`, `store`, a "load" reads from linear memory to the value stack, a "store" does the opposite.
- Arithmetic operations: `unary`, `binary`, operates on value(s) atop of the value stack.
- Function calls: passing arguments and receiving results for local, imported or exported functions.

Listing 1 shows a Rust implementation of the "DeNaN" analysis using the Wastrumentation API. Each Wastrumentation analysis implements *traps* that allow developers to register for specific Wasm events. The "DeNaN" analysis registers traps corresponding to the aforementioned instructions (Sections 2.1–2.1): `const_`, `local`, `global`, `load`, `store`, `unary`, `binary` and `apply`. The traps are defined using the `advice!` macro and are invoked by the instrumentation platform when corresponding events occur.

[1] WebAssembly 2.0 Specification – NaN Propagation (accessed on 28 Nov 2024).

The analysis instructs each trap function to capture each involved runtime value `v` and replace `v` with the result of `v.denan()`. It implements the method `denan` for Wasm values as an extension trait for the type `Value` (Sections 2.1–2.1). In Rust, extension traits allow to extend types with additional methods. This analysis declares the extension trait `Denan` (Section 2.1) and implements it for the type `Value` (Section 2.1) to allow invoking `denan` on Wasm values.

The implementation of `denan` (Section 2.1) matches on the type of the value. Integer values (matching `WasmType::I32` or `WasmType::I64`) are left untouched (Section 2.1). Floating point values (matching `WasmType::F32` or `WasmType::F64`) are cast to Rust's respective floating point type. On this floating type the method `is_nan` is called to intercept NaN instances. If `is_nan()` identifies a NaN instance, the value is replaced with 0.0, otherwise, it is returned unchanged (Section 2.1 and Section 2.1).

Listing 1 The Rust implementation of the "DeNaN" analysis. This analysis replaces all NaN floating-point values with 0.0 to ensure deterministic NaN-propagation across Wasm VMs.

```
#![no_std]
use wastrumentation_rs_stdlib::*;
advice! {
    const_ (v: Value, _l: Loc) {
        v.denan() }
    local (v: Value, _i: LocalIndex, _l: LocalOp, _l: Loc) {
        v.denan() }
    global (v: Value, _i: GlobalIndex, _g: GlobalOp, _l: Loc) {
        v.denan() }
    load (i: LoadIndex, o: LoadOffset, op: LoadOperation, _l: Loc) {
        op.perform(&i, &o).denan() }
    store (i: StoreIndex, v: Value, o: StoreOffset, op: StoreOperation, _l: Loc) {
        op.perform(&i, &v.denan(), &o); }
    unary (opt: UnaryOperator, opnd: Value, _l: Loc) {
        opt.apply(opnd.denan()).denan() }
    binary (opt: BinaryOperator, l_opnd: Value, r_opnd: Value, _l: Loc) {
        opt.apply(l_opnd.denan(), r_opnd.denan()).denan() }
    apply (func: WasmFunction, args: MutDynArgs, ress: MutDynResults) {
         args.update_each_arg(|_index, v| v.denan());
         func.apply();
         ress.update_each_res(|_index, v| v.denan()); } }
trait Denan { fn denan(self) -> Self; }
impl Denan for Value {
    fn denan(self) -> Self {
        use WasmType::{I32, I64, F32, F64};
        match self.type_() {
            I32 | I64 => self,
            F32 => self.as_f32().is_nan().then(|| 0_f32.into()).unwrap_or(self),
            F64 => self.as_f64().is_nan().then(|| 0_f64.into()).unwrap_or(self),
        } }}
```

2.2 Dynamic Analysis API

Table 1 shows the trap functions supported by Wastrumentation to build dynamic analyses. The table shows the high-level conceptual API, referring to Wasm constructs such as "location", "operator" and "operand". The trap functions are named after their corresponding program event. Each trap function gets as arguments details on the corresponding Wasm instruction and its code location. Such a conceptual API can then be implemented for a concrete high-level language that compiles to Wasm. Currently, our platform offers a Rust and AssemblyScript implementation of that API to implement dynamic analysis. The concrete language API is mapped onto the low-level Wasm instrumentation ABI by Wastrumentation at instrumentation time. The actual ABI for the conceptual API in Table 1 is specificied in Appendix A in [25].

Table 1 The dynamic analysis API supported by Wastrumentation. The columns show the trap name, the arguments provided to the trap, the type of value the trap must return and the implementation to ensure a transparent execution.

Trap name	Typed arguments	Return Type	Transparent Base
`if-then`$_{pre}$, `if-then-else`$_{pre}$	c:$type_{bool}$, input:$i32$, arity:$type_{i32}$, l:Loc	$type_{bool}$	c
`br`	l:$label$, l:Loc	$\emptyset$	
`br-if`	c:$type_{bool}$, l:$label$, l:Loc	$type_{bool}$	c
`br-table`	t:$i32$, effective-target:$label$, d:$label$, l:Loc	$type_{i32}$	t
`select`	c:$type_{bool}$, l:Loc	$type_{bool}$	c
`call`$_{pre}$, `call`$_{post}$	fn:idx_{func}, l:Loc	$\emptyset$	
`call-indirect`$_{pre}$	fn-idx:$i32$, tbl:idx_{tbl}, l:Loc	$type_{i32}$	fn-idx
`call-indirect`$_{post}$	tbl:idx_{tbl}, l:Loc	$\emptyset$	
`unary`	op:$unary$, o:$type_{val}$, l:Loc	$type_{val}$	op.apply(o)
`binary`	op:$binary$, l:$type_{val}$, r:$type_{val}$, l:Loc	$type_{val}$	op.apply(l, r)
`const`	v:$type_{val}$, l:Loc	$type_{val}$	v
`local`	v:$type_{val}$, i:idx_{local}, l:op_{local}, l:Loc	$type_{val}$	v
`global`	v:$type_{val}$, i:idx_{global}, g:op_{global}, l:Loc	$type_{val}$	v
`load`	i:idx_{load}, o:$offset_{store}$, op:op_{load}, l:Loc	$type_{val}$	op.perform(i, o)
`store`	i:idx_{store}, v:$type_{val}$, o:$offset_{store}$, op:op_{store}, l:Loc	$\emptyset$	op.perform(i, v, o)
`memory-size`	s:$type_{val}$, i:idx_{memory}, l:Loc	$type_{val}$	s
`memory-grow`	a:$type_{val}$, i:idx_{memory}, l:Loc	$type_{val}$	i.grow(a)
`block`$_{pre}$, `loop`$_{pre}$	input:$i32$, arity:$type_{i32}$, l:Loc	$\emptyset$	
`if-then`$_{post}$, `if-then-else`$_{post}$, `block`$_{post}$, `loop`$_{post}$, `drop`, `return`	l:Loc	$\emptyset$	
`apply`	f:$Function$, args:$Arguments$, ress:$Results$,	$\emptyset$	f.apply()

Our high-level API is inspired by Wasabi's instrumentation API, but it supports *intercession*, i.e. the ability of the analysis to modify the target program. This has two major implications for the design of the instrumentation API. We need to conceive (1) how the analysis controls the target program given the built-in safety and security characteristics of Wasm (detailed in Section 2.2.1), and (2) how the analysis can forward the intercepted operation to the target Wasm program (detailed in Section 2.2.2). The column "Transparent Base" in Table 1 denotes the operation that the analysis can call to forward the intercepted operation (captured at the trap function) to the target program.

2.2.1 Intercession: Modifying the Target Program

Dynamic analyses written on Wastrumentation can not only observe all the runtime behaviour of a target application but also control it. More concretely, analyses can change *part of* the target program behavior during the execution of the trap function. To illustrate this, consider the trap function `const_` that intercepts pushing a constant on the Wasm value stack. This trap function receives two arguments: the constant value `v` that would be pushed and the source code location `l` of the instruction. Wastrumentation allows the analysis to decide what value is pushed on the stack when the control flow returns to the target program by using the return value of the trap function. For example, the "DeNaN" analysis (cf. Section 2.1) uses this to alter the constant value `v` to a deterministic NaN value.

Generally, the API of Wastrumentation uses the return value of trap functions to alter the behavior of the target instruction. Whereas this is not always applicable (due to reasons further explained in Section 2.4), Table 1 shows when a return value dictates (part of) the target program behavior in the column "Return Type". Below, we further outline how the analysis is in control once a trap function is invoked.

- Intercession of control flow: For the trap functions `if`$_{\texttt{pre}}$, `if-then-else`$_{\texttt{pre}}$ `select`, and `br-if`, the return value dictates control flow of the branch expression as a boolean. For the trap function `br-table` the return value dictates the target label that control-flow will jump to. The return value of the trap function `call-indirect`$_{\text{pre}}$ determines the index in the function table of the instruction that will be invoked.
- Intercession of value stack manipulation: The following trap functions intercept instructions that push on the value stack: `unary`, `binary`, `const`, `local`, `global`, `load`, `store`, `memory-size` and `memory-grow`. The return value of the corresponding trap function is used as the outcome value that will be pushed on the stack.
- Introspection only: Aside for general intercession, the following trap functions cannot influence their respective instruction: `br`, `call`$_{\text{pre}}$, `call`$_{\text{post}}$, `call-indirect`$_{\text{post}}$, `block`$_{\text{pre}}$, `loop`$_{\text{pre}}$, `if`$_{\text{post}}$, `if-then-else`$_{\text{post}}$, `block`$_{\text{post}}$, `loop`$_{\text{post}}$, `drop` and `return`. This is due to how the respective program events do not depend on runtime information (eg. `br`) or report the termination of a program event (eg. `if-then-else`$_{\text{post}}$).
- Intercession of function application: The trap function `apply` can control a Wasm function application. When a function application takes place, this trap receives two pointers. The former points to an array containing the function arguments, and the latter points to the array containing the function results. This trap can alter the passed arguments before the function application takes place, and it can alter the return values before the control flow returns to the caller. Furthermore, this trap function is in control of whether the function application takes place at all.

2.2.2 Forwarding Intercepted Operations Back to the Target Program

Even if an analysis does not need to modify the target program, it remains responsible for the execution of intercepted program events. For example, if the trap function `const_` must remain transparent then the unaltered argument `v` must be the return value of the trap function. For the traps involved, the analysis is thus responsible for ensuring that the target operation takes place within the execution of the trap function. As mentioned before, Table 1 shows within the column "Transparent Base" the functions to be called within the trap function to forward the intercepted operation back to the target program.

Actually, the functions in the column "Transparent Base" represent the *continuation* of the intercepted operation that yields a return value. This design enables the analysis developer to add analysis code around the continuation of the intercepted operation and decide either to call the continuation or skip it altogether. This is most notable for the program events "apply" and "memory-grow". For "apply" the analysis controls function applications. This may further imply calling recursive functions. For "memory-grow", the return value either points to the next allocated page in linear memory or -1 if no allocation could be performed. This enables the analysis to artificially limit the available memory by not calling the memory to growth continuation and early returning with -1. For all other operations, the transparent base continuation is side-effect free and has no significant impact on analysis code that may go before or after the continuation.

2.3 Wastrumentation Design

Figure 1 depicts at a high-level the instrumentation process of Wastrumentation. We first describe the input provided to Wastrumentation, followed by the instrumentation process and the additional code that is generated. Then we elaborate on how the different modules are merged into a single output program. The numerical labels in the figure allow us to reference to visual elements in the following text.

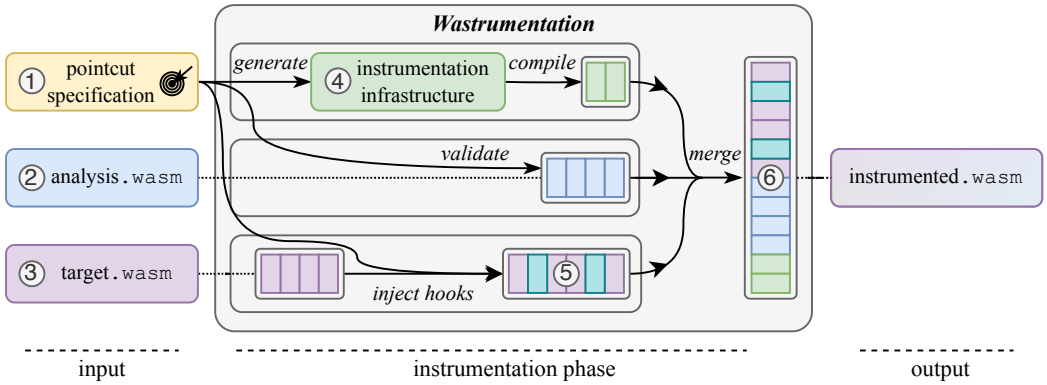


Figure 1 High-level overview of the architecture of Wastrumentation.

Input and Pointcut specification. In what follows, we adopt the pointcut specification terminology from aspect-oriented programming (AOP) to describe Wastrumentation inputs and their relationship. Note, however, that our implementation is limited compared to AOP, as our pointcut specification only supports static instruction targeting. As Figure 1 depicts, the input for Wastrumentation consists of the pointcut specification ①, the analysis program ②, and the target program to instrument ③. The pointcut specification enables selective instrumentation, by declaring the set of program instructions I and the set of functions of interest F within the target program. For example, the Rust implementation of the "denan" analysis in Listing 1 can be compiled to a Wasm module ②, along with a pointcut specification to target all implemented traps $I = \{const, local, global, load, store, unary, binary, apply\}$, and restrict instrumentation to the Wasm functions 4, 5, and 6 of the input program, $F = \{4, 5, 6\}$.

Instrumentation Process. For a given pointcut specification, Wastrumentation validates that the analysis module implements the trap functions corresponding to the specified target program instructions I. This validation takes place by iterating over the set I and validating that each corresponding trap function is exported by the analysis with the proper signature. If validation fails, Wastrumentation aborts the current instrumentation. If validation succeeds, Wastrumentation rewrites the identified instructions I within target functions F to inject hooks to placeholder analysis functions ⑤, according to the strategy explained in Section 2.4.

On-demand Instrumentation Code Generation. Some hooks pass compound data structures to traps functions that go beyond what the primitive number types of Wasm (i32, i64, i64, f64) can represent. For example, the *apply* trap function receives an array of Wasm arguments passed to the intercepted function call (cf. Section 2.4). Wastrumentation will generate additional instrumentation infrastructure as a new module ④. This infrastructure is generated on-demand, based on the input modules at hand. The module is responsible for managing such complex data structures, leaving the other modules untouched.

Bundling and Output. The final phase of Wastrumentation *bundles* the intermediate Wasm modules as a single instrumented module ⑥. To *bundle* these modules, Wastrumentation leverages the tool `wasm-merge` that is provided as part of the Binaryen tools [10]. This tool acts like a linker where multiple modules are merged into one. In doing so, `wasm-merge` replaces module imports that cross-reference other modules' exports with inline references to local functions in the resulting Wasm binary.

Preserving the Program State Space. Injecting analysis code into a target program comes with the challenge of preserving its original behavior. Any changes made during rewriting must not be observable for the input program. This requirement is critical to prevent deviations that could, for example, hide the program from a security analysis. The applied changes during instrumentation cannot interfere with the program state, including its memory, value stack, or call stack. This is unique to Wastrumentation. Other tools, e.g. Wasabi and Wizard, avoid this complexity because their analysis operates in a separate address space within the JavaScript interpreter or at the level of the Wasm interpreter, respectively. Our solution to preserve the input program state departs from the following properties of Wasm.

No Reflection. Wasm disallows runtime reflection regarding program code, the value stack, or the call stack [13]. This ensures that program code cannot inadvertently observe nor modify these components. Consequently, at execution time, the original input program code within the instrumented module remains unaware of the additional instrumentation code structure, values on the value stack and activation records on the call stack.

Separation of Linear Memory. Each memory load and write instruction in the instrumented module is statically tied to a unique memory view. When bundling the different code units (the target program, the analysis module, and additional instrumentation code), Wastrumentation leverages `wasm-merge` to assign distinct memory regions to each code unit. This separation leverages the Wasm standardized Multi-Memory proposal [12]. Multi-Memory enables to statically associate a unique memory index for every memory-related Wasm-instruction. This allows each module to operate exclusively on its unique linear memory index, ensuring complete isolation at runtime.

Figure 2 shows the statically enforced memory separation of merged program modules before and after program bundling. In the figure, the memory-relevant instructions such as `i32.load` and `i32.store` have a static memory index and a dynamic value that dictates the offset in the memory. Whereas each module has instructions addressing on its own set of linear memories, the instrumented module will contain the input program addressing the first indices of linear memory, followed by instrumentation and analysis modules operating at an on-demand decided memory index.

We ensure that the input program remains at its own offset, as external interoperation with the linear memory might make assumptions in the memory index, such as is the case for the widely adopted Wasm System Interface[2].

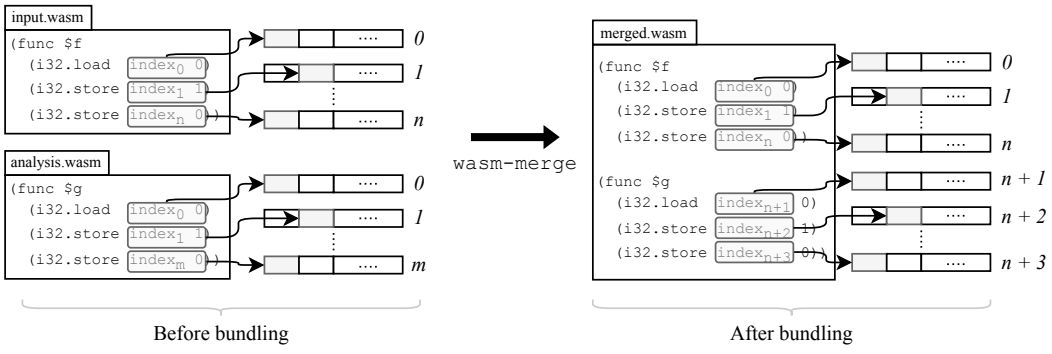


Figure 2 Visualization of the multi-memory access model before and after transformation.

[2] For our work we target the WASI P1, `https://wasi.dev` (accessed on 17 Dec 2024).

Listing 2 A analysis program that intercepts function applications and forwards the application.

```
(module ;; input analysis
  (func $apply (type $f_base:i32 $buf_sig:i32 $buf_sigtype:i32 $c_arg:i32 $c_res ⇒ void:i32)
    (; pre apply analysis code ;)
    (call $apply_base $f_base $buf_sig)
    (; post apply analysis code ;))
  (import "instrumented_input" "apply_base") (func $apply_base (param i32 i32)))
```

2.4 Instrumentation through Transformation

In what follows we detail the transformation strategy taken by Wastrumentation of the input program. The transformation follows two distinct patterns, one for function applications and another more general pattern for all other operations.

2.4.1 Function Application Instrumentation

The API of the "apply" trap receives a pointer to the *continuation function* and a pointer to a buffer containing the runtime arguments and results. The continuation function represents the next step in the execution flow. For the apply hook, this is the original function being intercepted. As Section 2.2.2 explains, the "apply" trap is responsible for calling the *continuation function* to allow a transparent execution of the target program. Inspired by the CLOS auxiliary "`around:`" method definition [3], the analysis code can (a) manipulate the arguments *before* the *continuation*, (b) determine *when* or *if at all* the function application takes place and (c) manipulate the results *after* the function application.

Listing 2 shows the minimal transparent analysis as a Wasm program that instruments function applications. The only implementation is the trap definition $*apply* (Section 2.4.1) that receives five arguments. The first argument (f_{base}) is a pointer to the *continuation function*. The second and third arguments (buf_{sig}, $buf_{sigtype}$) are pointers to buffers that contain the arguments and results for the current intercepted function application. The last two arguments (c_{arg}, c_{res}) dictate the number of arguments and results for that call respectively (to bound the value access of buf_{sig}, $buf_{sigtype}$). Section 2.4.1 calls the continuation, which may be preceded or succeeded with additional analysis code that is left out of the listing for brevity. Since the implementation of $\$apply_{base}$ resides in the transformed target program, the analysis must declare it as a function import (Section 2.4.1).

Listing 3 shows the input program for the transformation, containing a single function definition $\$f$. Whereas the transformation generalizes to any input set of function definitions, we restrict the example transformation to that of $\$f$ for brevity. Some intentional design considerations that influenced the transformation approach are the following:

- The analysis must be notified of all function applications of a target function, including (a) internal function applications through `call` or `call_indirect` statements local to the target module, (b) external function applications that may stem from the host and (c) for target functions that have no implementation details specification but are defined through an import statement.
- The API of the $*apply* trap is polymorphic for the target function application signature. This design choice enables low coupling between the analysis and the target input program, as the analysis program needs not to monomorphize the $*apply* trap implementation per signature as shown in [19]. Consequently, the analysis implementation must dispatch on function-specific behavior at runtime which may incur a performance penalty.

Listing 4 shows the input program result after transformation. The transformation of target function $\$f$ from Listing 3 results in two definitions $\$f_{original}$ (Section 2.4.1) and $\$f$ (Section 2.4.1). It is transparent to client code of $\$f$ (both internal and external to the Wasm

Listing 3 An input program candidate for dynamic analysis on function applications.

```
(module ;; input program
  (func $f (export "f") (type f_i^m ⇒ f_o^n) body_f))
```

module) since after transformation $\$f$ preserves its identity[3], its signature and its export declaration. However, $\$f$ will now call the trap $\$apply$ defined in Listing 2 with the required information.

In preparation for the call to $\$apply$, $\$f$ first stores arguments and runtime types to buffers (Sections 2.4.1–2.4.1) that $\$apply$ may manipulate at will. Next, $\$f$ yields control flow to $\$apply$ (Section 2.4.1), after which the results are loaded from the buffers (Sections 2.4.1–2.4.1) and eventually $\$f$ returns. The arguments to $\$apply$ (Section 2.4.1) are 0, the function pointer to the continuation function index in the table $\$table_{base-functions}$, the pointers $\$buf_{sig}$ and $\$buf_{sigvalues}$ pointing to the value- and types-buffers respectively and the the number of arguments m and results n.

If the trap $\$apply$ in Listing 2 calls the continuation $\$apply_{base}$, in Listing 4 the applicable continuation is resolved at runtime with the function pointer argument $\$f_{base}$ that is looked up in the function table $\$table_{base-functions}$ (Section 2.4.1). The call to the resolved applicable continuation $\$f_{base}$ loads the arguments from the buffer $\$buf_{sig}$ (Section 2.4.1) and calls $\$f_{original}$ (Section 2.4.1), after which the results are stored to $\$buf_{sig}$ (Section 2.4.1) and control is yielded to the post-analysis code in the trap $\$apply$.

Listing 4 The input program from Listing 3 after the instrumentation phase.

```
(module ;; input program, instrumented
  (func $f_original (type f_i^m ⇒ f_o^n) body_f)
  (func $f (export "f") (type f_i^m ⇒ f_o^n)
    (local $buf_sig i32) (local $buf_sigtype i32)
    ;; push values on stack
    (local.get 0) (local.get 1) ... (local.get m)
    ;; store args & types to instr-module-heap from value stack
    (local.set $buf_sig (call $alloc-vals-f_i^m-f_o^n))
    (local.set $buf_sigtype (call $alloc-typs-f_i^m-f_o^n))
    ;; call analysis apply
    (call $apply 0 $buf_sig $buf_sigtype m n)
    ;; retrieve results from instr-module-heap onto value stack
    (call $free-vals-f_i^m-f_o^n)
    (call $free-typs-f_i^m-f_o^n))
  (func $f_base (type $buf_sig:i32 ⇒ void)
    ;; retrieve args from instr-module-heap onto value stack
    (call $load-args-f_i^m-f_o^n (local.get $buf_sig))
    ;; call original body
    (call $f_original)
    ;; store results to instr-module-heap from value stack
    (call $store-results-f_i^m (local.get $buf_sig)))
  (import "analysis" "apply" (func $apply (type i32 i32 i32 i32 i32 ⇒ void)))
  (import "instr" "alloc-vals-f" (func $alloc-vals-f_i^m-f_o^n (type f_i^m ⇒ i32)))
  (import "instr" "alloc-typs-f" (func $alloc-typs-f_i^m-f_o^n (type void ⇒ i32)))
  (import "instr" "load-args-f" (func $load-args-f_i^m-f_o^n (type i32 ⇒ f_i^m)))
  (import "instr" "store-results-f" (func $store-results-f_i^m (type i32 ⇒ f_o^n)))
  (import "instr" "free-vals-f" (func $free-vals-f_i^m-f_o^n (type i32 ⇒ void)))
  (import "instr" "free-typs-f" (func $free-typs-f_i^m-f_o^n (type i32 ⇒ void)))
  (table $table_base-functions [$f_base] funcref)
  (func (export "apply_base") (param $f_base i32) (param $buf_sig i32) ;; => void
    (call_indirect $table_base-functions (type i32 ⇒ void) $f_base)))
```

[3] Identifiers are defined as alphanumeric tokens prefixed with a $-sign. In practice, Wasm functions are defined by an index as an i32-value, so a function name is actually a number that dictates its index.

The buffers used to communicate the runtime arguments and results between the function application, the analysis trap and the continuation function are handled by an external "instrumentation" module. This "instrumentation" module can orchestrate its linear memory while leaving the linear memory of the input program and the analysis code untouched. Functions to allocate and free memory, to load and store values in the buffers are imported in the instrumented module (Sections 2.4.1–2.4.1) and can be imported by the input analysis, though the pre- and post-analysis code is left out.

2.4.2 General Program Operation Instrumentation

All other instructions follow more concise rewrite strategies by Wastrumentation. These rewrite strategies rewrite each target operation into instructions that prepare context information on the value stack, followed by a call to an associated analysis trap function.

Table 2 summarizes the rewrite strategy, grouping instructions with similar rewrite patterns. The "Original instruction" column lists the unaltered instructions, while the "Rewritten instruction" column shows the instrumented result. The trap function is denoted as "$trap" for brevity but is specific to each instruction, such as "$return" for "return" or "$br" for "br".

As shown in Table 1, each trap function also receives the instruction's location. To provide this information, two constants (function index and instruction offset) are pushed on the stack before each trap call. These details are omitted from Table 2 for simplicity.

Maintaining the transformed program's well-typed property is essential. Thus the instrumented hooks must preserve the type correctness, including their interaction with the VM stack. Therefore, the analysis may modify the values handled by the target program, but it cannot alter their types. For instance, the rewrite of $t.\mathsf{const}\ c$ involves pushing the constant onto the value stack, followed by a call to the trap function "$const". This trap function receives the constant, returning a value of the same type while controlling its final value. Similarly, instructions like $t.unop$ and $t.binop$ are replaced with trap function calls that take the instruction operands and compute the resulting value on behalf of the program. The operands are available on the stack as they must have been for the original instruction.

For control-flow-related instructions such as "br-if", "br-table", and "select", the rewrite introduces a preceding trap function that controls the dynamic control flow. For "br-if" and "select," this is an `i32` value representing the condition. For "br-table," it is an `i32` value selecting a branch label from the statically predefined table associated with the instruction.

Instructions like "if-then-else", "if-then", "call-indirect", "call", "block", and "loop", include both pre- and post-trap functions. This design allows the analysis to trace the flow of execution per instruction, as additional instrumented code can be executed in between. For "if-then-else" and "if-then," the pre-trap function may modify the boolean condition controlling the flow. In contrast, trap functions for "return" and "drop" are observational, i.e. unable to alter control flow. For instructions like "local-get," "global-get," "local-set," "local-tee," "global-set," and "global-tee," the trap functions can alter the values being read or written to the respective variables.

3 Implementation

We implemented Wastrumentation in Rust and modularized the platform across different crates for improved maintainability and reusability. The core crate, as described in Figure 1, is responsible for transforming an input program based on a given analysis specification and pointcut specification. This crate consists of 5,118 lines of Rust code.

Table 2 The transformations applied by Wastrumentation.

Original Instruction	Rewritten instruction	Transformation description
	const	
t.const *c*	*t*.const *c*	Push constant on top of stack (TOS).
	call $trap	Call trap with constant, return value is result.
	unary, binary	
		Operands are on TOS, will be args to trap.
t.unop	*call* $trap	Call trap, return value is instruction result.
	br, br-if, br-table, select	
	(i32.const *l*)	Serialize target label ("br").
	call $trap	Call trap, passing serialized target label.
br *l*	br *l*	Return value of trap can change control flow for "br-if" and "br-table" and "select".
	if-then-else, if-then	
	(*call* $trap$_{pre}$ (i32.const *n*) (i32.const *m*))	Call pre-trap with blocktype arity: *n* input values on stack, *m* result values on stack.
if *blocktype* *instr**	if *blocktype* *instr**	Return value of pre-trap can alter control flow.
	(*call* $trap$_{post}$)	Call post-trap.
else	else	
instr*	*instr**	
	(*call* $trap$_{post}$)	Call post-trap.
end	end	
	call-indirect, call	
	i32.const *x*	Push f-idx ("call") or tbl-idx ("call-indirect").
	(*call* $trap$_{pre}$)	Trap does not take call args, only context.
call_indirect *x y*	call_indirect *x y*	To view or alter args / results, use trap *apply*.
	(*call* $trap$_{post}$ (i32.const *x*))	Call post-trap.
	block, loop	
	(*call* $trap$_{pre}$ (i32.const *n*) (i32.const *m*))	Call pre-trap with blocktype arity.
block *blocktype* *instr**	block *blocktype* *instr**	
end	end	
	(*call* $trap$_{post}$ (i32.const *n*) (i32.const *m*))	Call post-trap with blocktype arity.
	return, drop	
	call $trap	Call trap before instruction.
return	return	Trap can only observe, not alter semantics.
	local-get, global-get	
local.get *x*	local.get *x*	Push value on the value stack.
	(*call* $trap (i32.const *x*))	Call trap, return value is instruction result.
	local-set, local-tee, global-set, global-tee	
	(*call* $trap)	Instruction argument is TOS, call trap.
local.set *x*	local.set *x*	Trap may alter argument for instruction.

The core crate accepts only Wasm modules as analysis inputs, as explained in Section 2.3. However, these analyses must adhere to the analysis ABI (see Appendix A in [25]), which is typically not written by hand. To facilitate interaction with this ABI, we developed a domain-specific language (DSL) that exposes the high-level API described in Section 2.2. The primary and most mature implementation of this DSL is written in Rust as a set of Rust macros, and it is used in our evaluation (in Section 4). We also have a less mature implementation written in AssemblyScript as an AssemblyScript library, demonstrating the language agnosticism of the analysis ABI. The Rust DSL comprises 1,943 lines of code, and the AssemblyScript DSL, which currently only partially abstracts the low-level ABI, consists of 450 lines of code. Figure 3 illustrates how the analysis, along with the DSL implementation, is compiled into a Wasm module that serves as input for Wastrumentation. Note that once an analysis is compiled to a Wasm module, the trap functions must adhere to the statically typed signature in the ABI (Appendix A in [25]), or otherwise, Wastrumentation will reject the analysis.

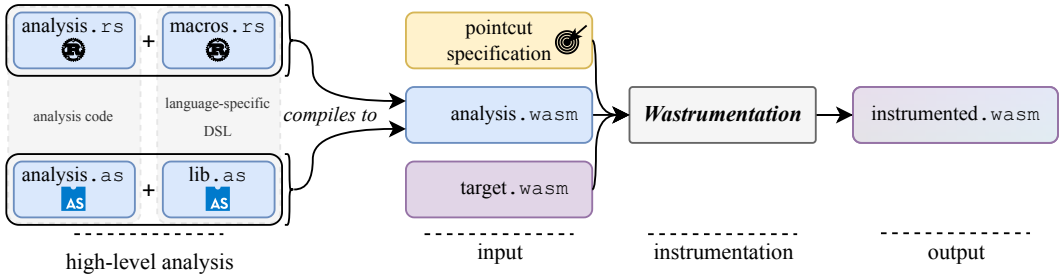


Figure 3 Overview of implementing analyses for Wastrumentation using a high-level language.

Rust and AssemblyScript are also supported for generating the instrumentation infrastructure. The code to generate the instrumentation infrastructure in either language amounts to a total of 4,451 lines of Rust code. Note that other languages can be supported, as long as a Rust-wrapper library can built around them, that implements the compiler interface used by Wastrumentation to compile the high-level program to a Wasm module.

4 Evaluation

We evaluate our work to check whether Wastrumentation is suitable for building dynamic analyses for Wasm applications. To this end, we assess the practicality and performance of our approach. We conduct several experiments to answer the following research questions (inspired by the ones found to evaluate other instrumentation platforms [19, 33, 35, 8, 38]):

RQ1. Does Wastrumentation aid the development of dynamic analyses compared to state-of-the-art approaches?
RQ2. Do the instrumented Wasm programs remain faithful to their original execution?
RQ3. How much does the code size increase after transformation?
RQ4. What is the runtime overhead due to instrumentation?
RQ5. What is the memory overhead due to instrumentation?

4.1 Methodology

We performed our experiments on two machines. The first is a server running Ubuntu 24.04.1 LTS (GNU/Linux 6.8.0-48-generic x86_64). The server has an AMD Ryzen 9 7950X processor (16 cores, 32 threads, 4.5–5.7GHz) and 128GB of main memory. The second machine is an Apple Mac Mini, with an M4 processor (4 performance and 6 efficiency cores) and 24GB of main memory.

We employed WasmR3 [1] commit 299be52, as target for a set of dynamic analyses. This benchmark suite includes 27 programs and serves as a reference for "realistic and standalone Wasm benchmarks" based on a 2024 record-reduce-replay pass of real-world programs.

To compare our approach to three state-of-the-art tools operating at three different levels:

- We use Wasabi to compare our approach to another source code instrumentation platform. We patched it to be compatible with WebAssembly 2.0 programs, commit ee2fb70
- We use Binaryen v122 to compare to a bytecode rewriting framework.
- We use Wizard [38] commit 1810b61 (compiled with Virgil [37] commit e64483e) to compare to a VM-level Wasm instrumentation framework.

In terms of runtimes, we execute Wasabi on Node V22.5.1, and we execute standalone Wasm programs on Wasmtime V30.0.2 and the aforementioned version of Wizard.

To avoid skewed results due to the VM warmup time or noise from other processes on the system, we compute the median of 30 executions of the target Wasm program for each individual runtime instance. When deploying an analysis with an instrumentation platform, we target every function of the input program and enable instrumentation for every program operation that the analysis depends on.

4.2 RQ1: Evaluating the Development of Dynamic Analysis Using Wastrumentation

To evaluate how Wastrumentation aids the development of dynamic analyses compared to state-of-the-art approaches, we implemented twelve dynamic analyses summarized in Table 3. The analysis code is written in Rust.

Table 3 The dynamic analyses implemented in Wastrumentation. We include the type of hooks and lines of code for the analyses in Wastrumentation and Wasabi. Those cells marked with ✗ correspond to analyses not possible to implement in Wasabi. The hooks used in [a] ("Basic Block Profiling") are `if_then_else, if_then, call pre, call_indirect pre, block pre, loop_ pre`, in [b] ("Branch Coverage") `if_then_else, if_then, br_if, br_table, select` and in [c] ("DeNaN") `apply, unary, binary, const_, local, global, load, store`.

Analysis Name	Hooks		Lines of code	
	Wasabi	Wastrumentation	Wasabi (JavaScript)	Wastrumentation (Rust)
Instruction Mix	all	all (introspection)	62	81
Basic Block Profiling	`begin`	6[a]	11	33
Instruction Coverage	all	all (introspection)	16	52
Branch Coverage	`if, br_if, br_table, select`	5[b]	21	30
Call Graph	`call_pre`	`call pre`	29	52
Dynamic Taint	all	all (introspection)	272	500
Memory Access Tracing	`load, store`	`load, store`	99	107
Cryptominer Detection	`binary`	`binary`	10	32
Forward	all	all	26	33
Denan	✗	8[c]	✗	35
Safe Heap	✗	`load, store`	✗	69
Memoization	✗	`apply`	✗	82

The first eight analyses in Table 3 ("Instruction Mix", "Basic Block Profiling", "Instruction Coverage", "Branch Coverage", "Call Graph", "Dynamic Taint", "Memory Access Tracing", "Cryptominer Detection") are analyses we ported from Wasabi [19][4]. We use the Wasabi analysis implementations available on the latest commit 21a322b in their public repository. The lines of code for the ported implementation in Rust remain in the same order of magnitute than those implemented in JavaScript for Wasabi. The main reason for the code size increase is to conform with the Rust type system and borrow checker (e.g., strict typing, synchronize on mutable global state). The ported analyses show that Wastrumentation does not sacrifice in terms of its ability to implement existing analyses.

The latter four analyses in Table 3 ("Forward", "DeNaN", "Safe Heap", "Memoization") are new analyses. More concretely, we implemented "Forward" in both platforms, and the last three are novel analyses that require intercession (which is not supported in Wasabi). We describe those analyses in what follows.

[4] Due to space constraints, the code for those analyses is included in the artifact. The semantics of the analyses are the same as they were implemented for Wasabi for the evaluation [19].

- **Forward**. This analysis implements all trap functions without altering the original program (cf. Table 1, column "Transparent Base"). It provides insight into the instrumentation platform's minimum performance overhead as no additional analysis code is included. Section 4.3 also uses it as a reference analysis to validate that input programs remain faithful to their execution semantics after instrumentation.
- **Denan**. This analysis replaces all `NaN`-floating point values in the program with the floating point value zero. The analysis is inspired by the "`--denan`" instrumentation pass from Binaryen. Section 2.1 showed our implementation and Section 4.2.2 compares it to the ad-hoc implementation in Binaryen.
- **Safe Heap**. This analysis checks for incorrect heap access. It is inspired by the "`--safe-heap`" instrumentation pass from Binaryen. Listing 5 shows its implementation and Section 4.2.2 compares it to the ad-hoc implementation in Binaryen. For every load or store operation, the analysis checks null dereferencing, reading past available memory and usage of addresses with incorrect memory alignment (Sections 4.2–4.2).
- **Memoization**. This analysis implements a memoization strategy of pure functions [24]. Its goal is to cache the outcome of pure function applications to save on repeated computation efforts. Section 4.2.1 details our implementation.

For didactical purposes, we omit (1) synchronization primitives on global state, (2) type casting and (3) method invocations for type conversion in Listing 5 and Listing 6.

Listing 5 Implementation of Binaryen's "`--safe-heap`" pass in Wastrumentation.

```
use wastrumentation_rs_stdlib::*;

const WASM_PAGE_SIZE:i64 = 65_536;
fn bounds(idx:i64,byts:i64,ofst:i64) {
 // strict_add: assert no overflow
 let last_idx_byte =
  idx.strict_add(byts).strict_add(ofst);
 assert!((last_idx_byte != 0) &&
  (last_idx_byte <=
   (base_memory_size(0)) *
    WASM_PAGE_SIZE)); }

// turn size into bitmask (‘-1‘), then
// assert mask-region is zero-only (‘&‘)
fn alignment(idx:i64, size:i64) {
 // size == 4 (_32) | 8 (_64)
 assert!(idx & (size - 1) == 0); }

advice! {
 load (i:LoadIndex,o:LoadOffset,
       op:LoadOperation,l:Loc) {
  bounds(i,op.target_size(),o.offset());
  alignment(i,op.target_size());
  op.perform(&i,&o)
 }
 store (i:StoreIndex,v:WasmValue,
        o:StoreOffset,op:StoreOperation,
        l:Loc) {
  bounds(i,op.target_size(),o.offset());
  alignment(i,op.target_size());
  op.perform(&i,&v,&o); } }
```

Listing 6 A dynamic analysis implementation that caches the result of stateless function.

```
use std::collections::HashMap;
use wastrumentation_rs_stdlib::*;

struct Key{fn_idx:i32,ags:Vec<Byte>}
static mut CACHE:HashMap<Key,Vec<Value>>
 = HashMap::new();

fn cache_hit(key:&Key) -> bool {
 CACHE.contains_key(&key) }

fn cache_retrieve(
 key:&Key, results:&mut Results) {
 CACHE.get(&key).unwrap().into_iter()
  .enumerate().for_each(|(i, v)|
   results.set_res(i, v.to_owned()));
}

fn cache_insert(key:Key, rss:&Results) {
 let rss = rss.rss_iter().collect();
 CACHE.insert(key, rss);
}

advice! { apply (
 fnc:WasmFn, ags:Args, rss:Results) {
  let ags = ags.map(to_bytes).collect();
  let key = Key {fn_idx: fnc.idx, ags};
  if cache_hit(&key) {
   cache_retrieve(&key, &mut rss);
  } else {
   fnc.apply();
   cache_insert(key, &rss); } } }
```

4.2.1 The Memoization Analysis

We now discuss the details of the memoization analysis. Listing 6 shows its implementation in Wastrumentation. The analysis maintains a global hashmap as *cache* where it associates a function index and its arguments with the results for that function call (Sections 4.2–4.2). The analysis targets the *apply* trap since only function applications will be memoized (Section 4.2). The trap implementation computes the cache key (Sections 4.2–4.2) and returns the associated result values if the key is stored in the cache. If the cache had no associated entry for the computed key, the function application continues and the computed result is stored in the cache for successive function applications (Sections 4.2–4.2).

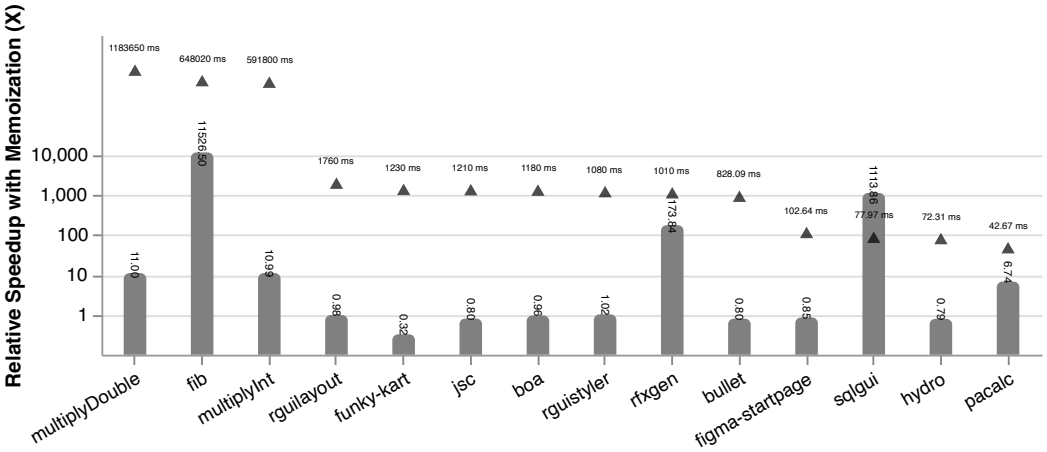


Figure 4 Execution times after instrumentation with the memoization analysis. The programs are sorted by their absolute runtime in miliseconds (▲) from left to right. The bar charts indicate the relative speedup on a logarithmic scale.

To assess the effectiveness of the memoization analysis, we conduct an experiment to check whether it would benefit the programs in the WasmR3 benchmark suite executed on Wasmtime. We first perform a static analysis phase to identify the programs with pure functions which could be good candidates for memoization. This phase identified 14 candidate programs (out of the 27) featuring pure functions. Figure 4 plots the measured runtimes of those programs and their instrumented variants. Our results show that the analysis can incur a performance penalty for some programs, and it can speed up others. The performance penalty ranges from 1.02x slowdown (rguilayout) to 3.10x slowdown (funky-kart), while the speed-up ranges from 1.02x speedup (rguistyler) to 1,1526.50x speedup (fib). This experiment shows that such analyses can be developed at low cost for exploration, and yield the potential to then be ported within concrete VM implementations if they offer promising results.

4.2.2 Comparison to Bytecode Rewriting Frameworks

As mentioned before, state-of-the-art dynamic analysis platforms for Wasm do not support intercession, but bytecode rewriting frameworks such as Binaryen [10] and Walrus [32] do. We observe that Binaryen v122 offers six instrumentation passes: "`--safe-heap`", "`--denan`", "`--instrument-locals`", "`--instrument-memory`", "`--log-execution`", "`--trace-calls`". Out of the six, all offer support for intercession except for "`--log-execution`" . Unfortunately, the "`--denan`" analysis is the only instrumentation pass that rewrites the input program into a variant that can execute standalone. All other instrumentation passes rewrite the target

program such the host must implement a part of the analysis, eg. what to perform when observing reads or writes into locals for the "`--instrument-locals`" pass. This couples the analysis with host-provided functionality. Titzer et al. developed in [38] two standalone ad-hoc analyses, "branches" and "opcodes", in the Walrus bytecode rewriting framework. To further evaluate RQ1, we compare the implementation of these three ad-hoc analysis implementations ("DeNaN", "branches", and "opcodes") that can execute standalone with an equivalent analysis on Wastrumentation.

When comparing the implementation of a given analysis on Wastrumentation to an ad-hoc solution, we observe three main limitations of bytecode rewriting frameworks.

Higher implementation effort. We observe repeated efforts of constructing the low-level Wasm constructs and injecting these within the target program when using bytecode rewriting frameworks. This most notably affects the code size of the analyses: "DeNaN" (195 LoC on Binaryen, 35 on Wastrumentation), "branches" (291 LoC on Walrus, 24 on Wastrumentation), and "opcodes" (242 LoC on Walrus, 81 on Wastrumentation). The lines of code for analyses in Wastrumentation are smaller since the transformation efforts are reused among analyses as they happen within Wastrumentation and Wasm-Merge.

Semantic abstraction loss. We observe a notable loss of high-level semantic expressiveness in bytecode rewriting systems because analysis implementations are restricted to low-level, elementary operations due to the lack of access to rich abstractions, such as complex data structures and a strong type system. For example, "DeNaN" analysis is limited to changing a value to 0 while the "branches" and "opcodes" increment a fixed set of global counters by 1. The other analyses implemented by Binaryen ("`--safe-heap`", "`--instrument-locals`", "`--instrument-memory`", "`--log-execution`", "`--trace-calls`") avoid additional complexity by forcing the analysis developer to implement more complex data structures within the host, such as logging infrastructure, hashmap dictionaries, or debugging infrastructure. In contrast, implementing the analyses on a general-purpose analysis platform is easier since developers do not need to manipulate Wasm instructions nor ensure the type safety of the analysis code. In Wastrumentation, developers can implement the analysis using a high-level API and in a high-level language like Rust. This reduces the complexity of the analysis by raising the abstraction level, allowing for a compiler to type-check, validate, and optimize the analysis code.

Lack of Separation of Concerns. We observe the analysis logic is tightly interwoven with the transformation pass, resulting in reduced modularity and maintainability. This means that instructions such as 'increment a counter' (analysis code) are scattered among instructions such as 'extend target body with increment' (transformation code). Wastrumentation decouples these concerns by treating the analysis implementation as a unit separate from the instrumentation phases. As a result, it becomes easier to maintain, comprehend, and evolve the semantics of the analysis independently from instrumentation.

Finally, we would like to note that the Binaryen instrumentation pass "DeNaN" is incomplete. The implementation of the "`--denan`" pass in Binaryen documents does not support the instrumentation of `local.get`, as doing so "would cause problems if we ran this pass more than once (the added functions use gets, and we don't want to instrument them)" [11]. Since Wastrumentation decouples instrumentation from the analysis implementation, the instrumentation of `local.get` operations do not run the risk that analysis code and its instructions of `local.get` are instrumented too.

Table 4 Violated assertions when deploying the Forward analysis using Wastrumentation on the Wasm official test suite.

	Tool	Execution Stage	Warning	# Modules	# Assertions
(1)	Wastrumentation	Transformation	Expected type f64, got f32	1	0
(2)	Wasm-Merge	Transformation	Unknown misc operation	8	36
(3)	Wasmtime	Runtime	Uninitialized element	4	4
(4)	Wasmtime	Runtime	Equality assertion failure	2	2
(5)	Wasmtime	Validation	Undeclared func. reference	2	8

4.3 RQ2: Evaluating the Faithful Execution

We take an experimental approach to validate that input programs remain faithful to their execution semantics after instrumentation. We validate that instrumented programs maintain structural integrity and correct execution semantics across (a) the 4,082 programs in the official Wasm test suite and (b) the 27 programs in the WasmR3 benchmark suite.

To validate the faithful execution of the 4,082 programs from the official Wasm test suite, we instrumented them with the "Forward" analysis (presented in Section 4.2). This analysis targets every supported trap without altering the semantics of the original program. The official Wasm test suite includes 26,632 assertions. We disabled those that assert failures (i.e., assertions that expect an invocation to trap)[5], as the goal of the experiment is to show that the "Forward" analysis does not alter successful tests. Disabling assertions on failures ensures proper execution of subsequent assertions. Otherwise, an exception in the input program halts the module's execution, preventing the instrumentation code from being notified and unable to clean up its internal state[6]. While the forward analysis is stateless, the instrumentation infrastructure remains stateful with respect to the apply trap (cf. Listing 4). For this reason, we disable failure assertions to retain all success assertions.

For the remaining 21,392 assertions, Wastrumentation has a success rate of 99.77%, with only 50 assertions failing across 17 modules. Table 4 categorizes the failures by the tool reporting the failure (Wastrumentation, Wasm-Merge, or the Wasmtime runtime), the stage at which the failure occurred (Transformation, Validation, or Runtime), the reported warning, the number of affected modules, and the number of affected assertions.

Failure (1) stems from `wasabi-wasm` [17], the underlying transformation library used by Wastrumentation which fails to infer the types of the input program. Failure (2) originates from Wasm-Merge, which cannot identify an input operation. We suspect this can be solved by enabling more feature flags provided by Wasm-Merge as they enable more Wasm instructions. Failure (3) results from uninitialized elements (static declared Wasm memory segments), which Wastrumentation does not directly manipulate. This issue could be addressed with further investigation into the serialization of our transformed module or the merge step. Failure (4) is a runtime failure caused by disabling previous assertions after they failed, violating the expected state. Finally, failure (5) requires more investigation.

In a second experiment, we verify Wastrumentation using the 27 programs of the WasmR3 benchmark suite across all 12 analyses listed in Table 3. For each input program in the WasmR3 and each analysis, we used `wasm-validate`, a Wasm binary validation tool, to verify the structural integrity of the code after instrumentation. The results showed no errors reported by `wasm-validate` for any of the analyses, confirming that Wastrumentation produces valid Wasm code that fully adheres to the requirements of the Wasm binary format.

[5] The following assertions (including count) were disabled: "assert_invalid" (1,476), "assert_trap" (2,388), "assert_exhaustion" (15), "assert_malformed" (1,277), "assert_unlinkable" (84).

[6] A proposal for Wasm exception handling is not yet standardized. If it were standardized, a set of exceptions could be accommodated by the instrumentation platform to clean up the analysis state.

4.4 RQ3: Evaluating Code Size Increase

We now compare the code size increase of Wastrumentation with respect to the state-of-the-art source code instrumentation platform Wasabi. To compare binary output size after instrumentation for both platforms, we computed $s = \frac{\mathsf{code{-}size}_{\mathsf{wasabi-instrumented}}}{\mathsf{code{-}size}_{\mathsf{wastrumentation-instrumented}}}$. This ratio s indicates where Wastrumentation-instrumented programs require fewer bytes ($s > 1$) or when Wasabi-instrumented programs require fewer bytes ($s < 1$).

We calculated the s ratio for each instrumentation of the WasmR3 benchmark suite using the dynamic analyses supported by both platforms. More concretely, the dynamic analyses used are the first eight analyses in Table 3 along with an adapted version of the "Forward" analysis. To ensure a fair comparison, we modified our "Forward" analysis to only include hooks that are compatible with Wasabi.

Note that, for Wasabi, the analysis logic is implemented on the JavaScript side, whereas, in Wastrumentation, it is embedded as Wasm code within the instrumented binary. Since JavaScript and Wasm serve different execution models and cannot be directly compared, we exclude the JavaScript analysis implementation from the code size comparison for Wasabi's instrumented programs to ensure a fair evaluation.

4.4.1 Results

Figure 5 shows the ratio s for each instrumentation in the cross-product. In cases where Wasabi failed to instrument a specific input program, we leave the cell blank as no ratio can be computed. Additionally, Figure 6 shows the size of the input programs prior to any instrumentation. The code size increases for each platform individually are presented in Appendix B in [25].

Out of a total of 180 successful ratios, 140 indicate the output Wasm module code generated by Wastrumentation is smaller, while the remaining 40 favor Wasabi. The analyses "Basic Block Profiling", "Branch Coverage" and "Call Graph" tend to favor Wasabi, which aligns with the low number of target operations in these analyses (6, 5 and 1, respectively). For the other analyses, which involve a larger number of traps, the size ratio generally favor Wastrumentation, resulting in smaller binary sizes compared to Wasabi.

The noticeable "red stripe" for the input program "game-of-life" indicates a significant advantage for Wasabi's instrumentation. This is largely due to the significant difference in binary sizes among the input programs. Whereas "game-of-life" has an uninstrumented size of just 1.199 bytes, the other input programs range from 38.965–22.264.896 bytes. This suggests that Wastrumentation introduces a relatively larger size overhead for smaller programs compared to Wasabi, which may be expected as this metric counts for Wastrumentation the additional analysis code, whereas it does not count the JavaScript analysis code for Wasabi.

4.5 RQ4: Evaluating the Runtime Overhead of Wastrumentation

To assess the runtime overhead of Wastrumentation, we performed 4 experiments. The first experiment evaluates the runtime overhead for Wastrumentation for its load, validate and execution of the "Forward" analysis. The second experiment computes the execution overhead of Wastrumentation for the 12 analyses discussed in Table 3. The third experiment compares the runtime overhead to the Wasabi source code instrumentation platform. Lastly, we compare the performance overhead to state-of-the-art bytecode rewriting frameworks.

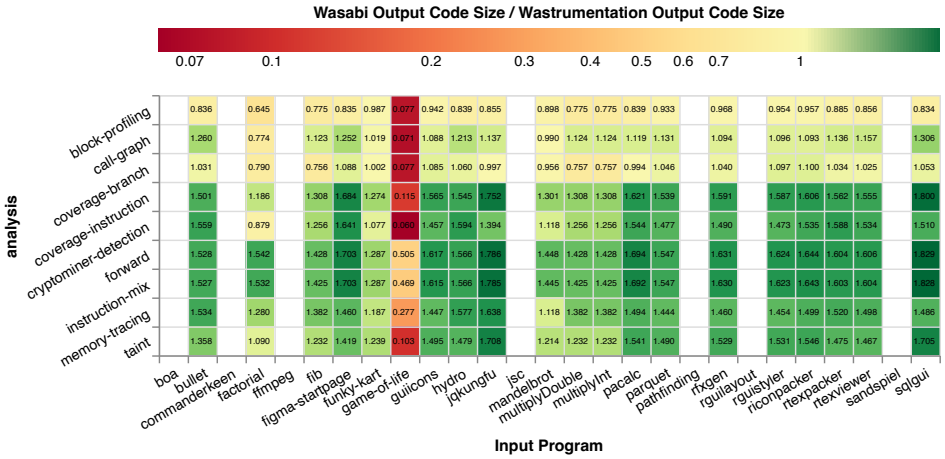


Figure 5 Code size ratio of Wasabi-over Wastrumentation-analyzed programs.

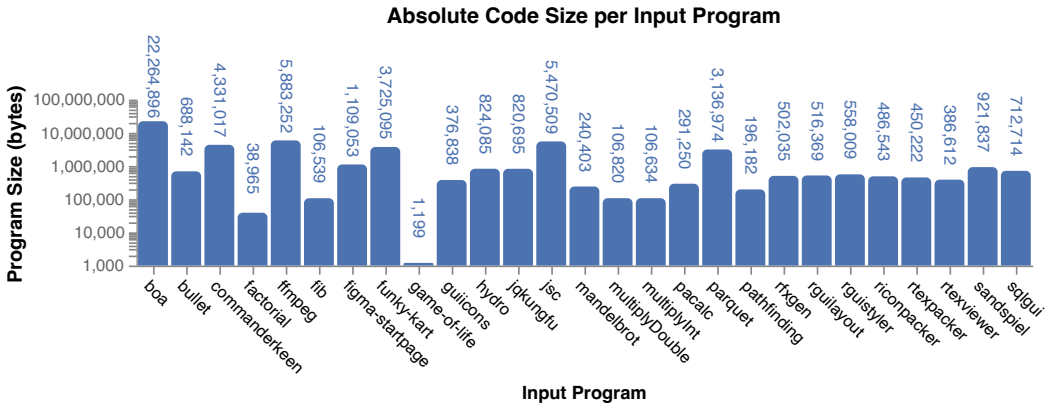


Figure 6 Code size of input programs from the WasmR3 benchmark suite.

4.5.1 Load, Validate and Execute Time

In this section, we evaluate the cost of loading, validating and executing programs for each stage for the input programs of the WasmR3 input suite when instrumenting with the "Forward" analysis. Figure 7 plots the overhead for each stage for 20 instrumented input programs with Wastrumentation. The overhead is computed using the Wizard engine. The median is computed for both the uninstrumented and instrumented variant over 30 runs. For the remaining input programs, 5 timed out, and 2 could not execute.

We observe that the relative increase for loading and validating an instrumented variant correlates with the absolute code size increase from Section 4.4. This is because both phases require reading the input program into the execution engine and performing a static validation on it. For loading, this is in the range 1.13–5.37, for validating in the range 1.84–5.65.

Second, we observe that the increase in execution time spent for the input programs differs significantly depending on the input program. We further ported 10 analyses other than "Forward" to Wastrumentation for further comparison, however other analyses execution failed with the message "`!trap[MEM_OUT_OF_BOUNDS]`". We believe this error does not originate for the "Forward" analysis since it does not reserve additional linear memory in that analysis code. To the best of our knowledge, this may point out a regression for the Wizard baseline engine, since no such exceptions occurred when executing the same experiments on other execution engines such as NodeJS, Wasmtime, and Wasmer.

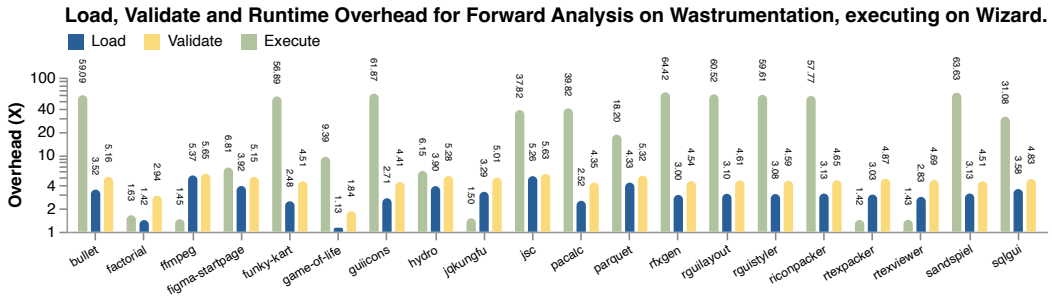


Figure 7 Overhead for loading, validating and executing time after instrumenting WasmR3 programs with the "Forward" analysis on Wastrumentation, executing on Wizard.

4.5.2 Setup for NodeJS experiments

To determine the runtime overhead of Wastrumentation, we compute the ratio o as the performance of instrumented execution relative to uninstrumented execution. An o value of 2 means instrumentation slows the execution time down by a factor of 2.

To compare the performance overhead for Wastrumentation and Wasabi, we computed a ratio p as the execution time of a Wasabi-instrumented module relative to the execution time of a Wastrumentation-instrumented module. A value $p > 1$ indicates Wastrumentation-instrumented execution requires less time to complete, and a value of $p < 1$ indicates Wasabi-instrumented execution requires less time to complete. For example, a p value of 0.5 means that "Wasabi is twice as fast at runtime to analyse the program", while a p value of 2 means that "Wastrumentation is twice as fast at runtime to analyse the program".

Similar to Section 4.4, we compute the ratio's o and p for the programs in WasmR3 using the analyses supported by both platforms. We profile the execution of uninstrumented and instrumented programs using the performance observation API from NodeJS "perf_hooks". The recorded times do not include the program loading from main memory or the verification or compilation of the module. As mentioned in Section 4.1, we calculate the median of 30 executions of the program within a single NodeJS instance. During our experiments, the execution of an instrumented program was limited to a timeout of 300 seconds.

4.5.3 Runtime overhead of Wastrumentation

Figure 8 shows the results for the overhead ratio o. The white cells indicate missing data, either due to the instrumented execution exceeding the timeout (T) or due to raising an uncaught exception (MD). In total, 51 combinations timed out, while 1 raised an uncaught exception ("memory-tracing" for the input program "pathfinding").

The results show that the instrumentation overhead for Wastrumentation ranges from 1x–514.0x. The analyses "coverage-instruction" and "taint" generally incur a higher overhead of 5.5x–514.0x and 12.85x–376.9x, respectively. These analyses are, however, heavy-weight as they implement all traps and maintain heap-allocated data structures that grow during the target program execution (a coverage hashmap and growable vectors, respectively). Nevertheless, Wastrumentation's overhead aligns with expectations for similar source code instrumentation platforms such as Wasabi [19], Jalangi [33] and Aran [6].

4.5.4 Comparison to Wasabi

Figure 9 shows the results for p ratio used to compare the runtime overhead for Wastrumentationand Wasabi. Note that figure only plots 20 input programs, as 7 input programs could not be instrumented by Wasabi so no ratio could be computed. The white cells indicate missing data, labeling T for "timeout", followed by the platform "M" (Wastrumentation), "S" (Wasabi) or "M&S" (both). For a total of 180 combinations where both Wastrumentation and Wasabi yield a valid instrumented module, 140 combinations yield a measurement. For 31 combinations both Wastrumentation and Wasabi timeout. For Wastrumentation an additional 3 combinations timeout and for Wasabi an additional 6. The performance overhead individually per platform for additional reference are further detailed in Appendix C in [25].

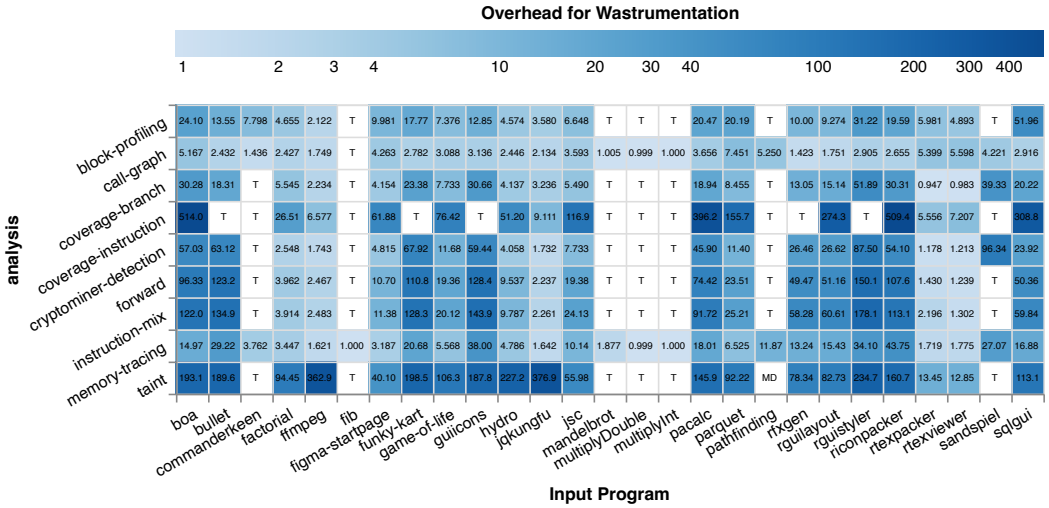


Figure 8 Execution overhead for instrumenting programs using Wastrumentation. Instrumentation is done for the cross-product of shared dynamic analyses and the WasmR3 benchmark suite.

The results of this experiment show that the distribution does not favor any particular instrumentation platform. Overall, the most suitable platform is determined by the specific input program, rather than the type of analysis applied. For example, for some input programs ("figma-startpage"–"hydro") Wastrumentation incurs a notable lower overhead across all analyses, while for other input programs ("jqkungfu", "rtexpacker", "rtexviewer") the opposite holds. Some notable outliers are ["jqkungfu","taint"] where Wasabi-instrumented execution is 17.5x faster and the "memory-tracing" analysis for "bullet", "funky"-kart, "guiicons", "ffxgen", and "rguistyler" where Wastrumentation is >9.0x faster than Wasabi.

To coroborate that our findings are consistent across virtual machines, we also deployed the "Forward" analysis for Wastrumentation on Wasmtime. Appendix D in [25] plots our findings, and confirms that across NodeJS and Wasmtime the overhead for the "Forward" analysis remains within the same order of magnitude.

4.5.5 Comparison to bytecode rewriting systems

Lastly, we compare the performance overhead of Wastrumentation with that of analyses implemented in bytecode rewriting systems. To this end, we compute the overhead of the bytecode rewriting analyses that we ported to Wastrumentation as described in Section 4.2.2.

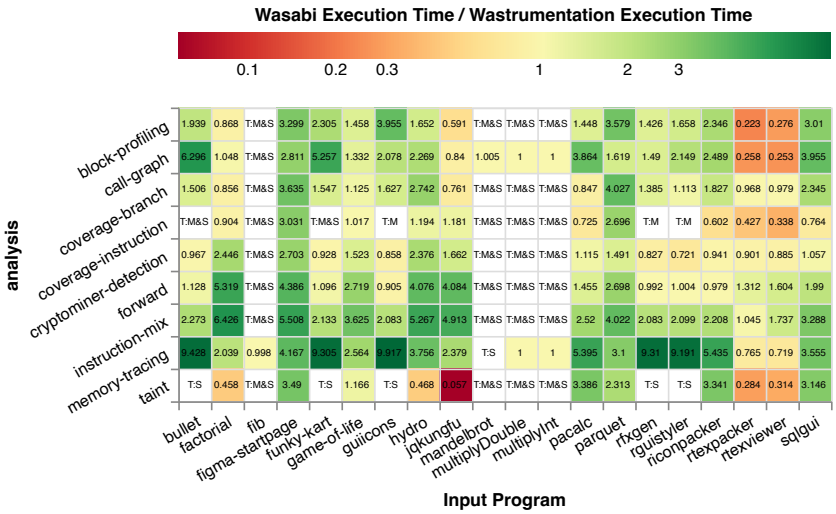


Figure 9 Execution time ratio of Wasabi-analyzed programs over Wastrumentation-analyzed programs.

Figure 10 plots the performance overhead for various input programs of the WasmR3 benchmark suite. For all analyses and input programs, the runtime overhead of bytecode rewriting is one or two orders of magnitude lower. These results are confirmed by earlier experiments, as pointed out by Titzer et al. [38].

The performance advantage of bytecode rewriting systems stems from two aspects. First, bytecode rewriting systems allow developers to have more fine-grained control over which specific locations to inject instrumentation code statically. Second, they can directly inline the analysis within the target operations. In contrast, Wastrumentation, requires runtime dispatching on generic operations. For example, our "binary" trap encapsulates 76 different instructions requiring additional dispatch at runtime.

4.6 RQ5: Evaluating the Memory Overhead of Wastrumentation

We now assess the runtime memory overhead incurred by Wastrumentation and compare it to Wasabi. To evaluate the memory overhead, we used the "Forward" analysis on the WasmR3 benchmark suite for both platforms. We measured the runtime memory using Node.js's built-in `process.memoryUsage()` method after executing each program. The memory overhead for each instrumented program was calculated relative to an uninstrumented baseline, representing the increase in memory usage (expressed as x times more memory usage).

4.6.1 Results

Figure 11 shows the results for each input program. For Wasabi, memory overhead could not be computed for the seven programs that failed to be instrumented, and these programs show no value in the figure. We observe that Wastrumentation's memory overhead ranges from 1.01x to 37.65x, while the memory overhead for Wasabi ranges from 1.03x to 12.34x. We suspect that the higher memory overhead in Wasabi is due to Node.js JIT employing more memory, as the analysis is written in JavaScript, but further experimentation is needed to quantify this hypothesis.

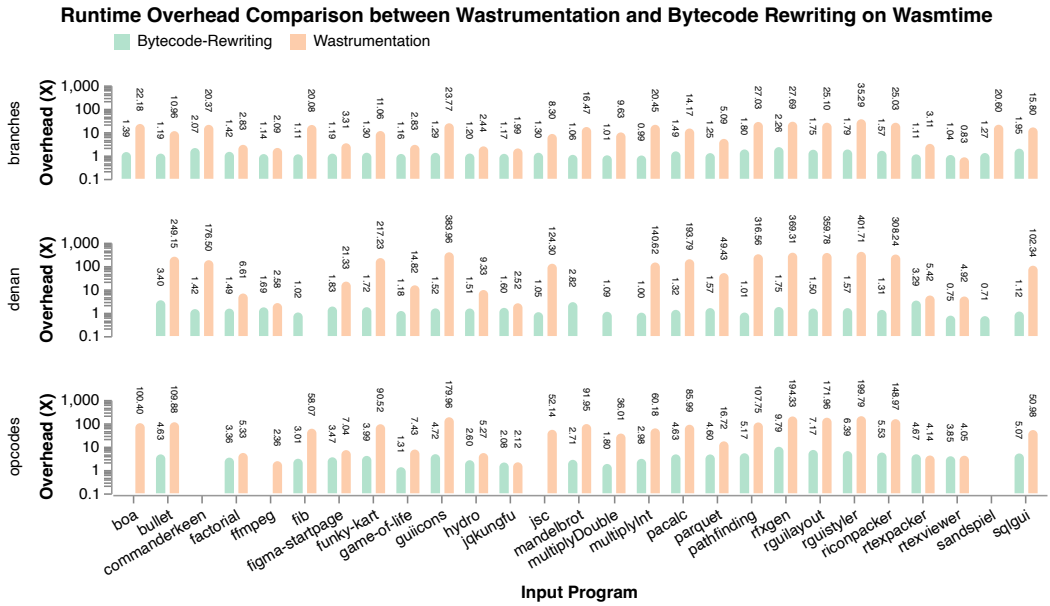


Figure 10 Performance overhead comparison between Wastrumentation and bytecode rewriting systems across various input programs from the WasmR3 benchmark suite.

In the context of Wasm programs, runtime memory is often a limited resource, particularly in environments such as cloud-based platforms or microcontrollers. As a result, minimizing memory overhead is critical for ensuring efficient execution. These results indicate that Wastrumentation may be better suited for deployment in memory-constrained environments.

5 Related Work

In recent years, WebAssembly has become a popular target for dynamic analysis. These analyses span various domains, including security [2], taint tracking [9, 36], program comprehension [20, 30, 26, 39], and profiling [21]. This section compares Wastrumentation to other instrumentation platforms for Wasm and discusses closely related work beyond Wasm.

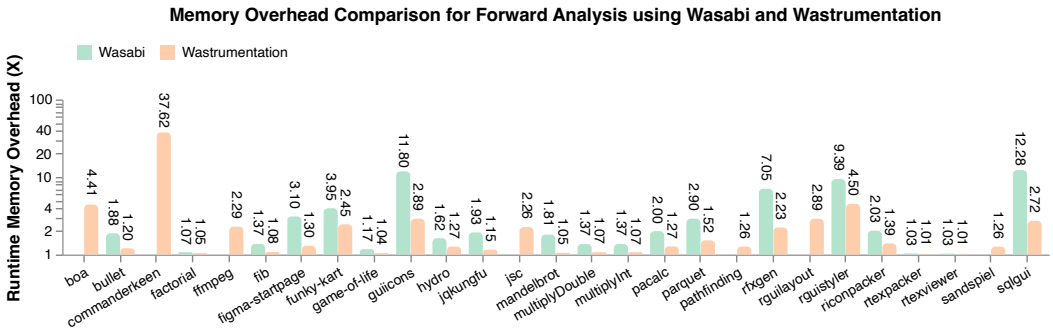


Figure 11 Runtime memory overhead of programs analyzed with Wasabi and Wastrumentation using the "Forward" analysis on the WasmR3 benchmarks. Baseline runs uninstrumented on NodeJS.

5.1 Instrumentation Platforms for WebAssembly

As mentioned before, there exist several tools that enable code transformation and instrumentation for Wasm binaries, Brewasm [5], WasmManipulator [29], and Binaryen [10]. These tools facilitate code injection by providing low-level APIs to modify the input program binary code. However, the analysis developer is responsible for the correctness of the transformation.

Wasabi [19] was the first dynamic analysis platform for Wasm, enabling developers to implement analysis against a high-level instrumentation API and ensuring a correct transformation. The analyses are written in JavaScript by implementing functions that execute code for specific program events of interest. Wasabi achieves instrumentation through source-level code rewriting, generating an instrumented version of the target program that calls the analysis code at runtime. Like Wasabi, Wastrumentation also performs source code instrumentation. In contrast, Wastrumentation does not require a specific language for the analysis, as long as the language can compile to Wasm. Moreover, analyses can operate in environments outside the web, where the Wasm VM may run without a JavaScript VM.

Wizard [38] follows a different instrumentation approach than that of Wasabi and Wastrumentation. Wizard is a Wasm VM with built-in support for instrumentation. In Wizard, analyses are implemented as instrumentation "probes". These probes are developed as extensions to classes within the Wizard VM, which upon execution of a target program will invoke the relevant probe. Wizard focuses on minimizing instrumentation overhead through features that VM-level instrumentation can benefit from, e.g., dynamically adapting analysis code during execution and JIT intrinsification of instrumentation code. However, with this approach, applications executed in the Wizard VM are the only ones that could be analysed. In contrast, dynamic analyses implemented with Wastrumentation do not have additional requirements for the runtime environment, making them portable to any Wasm VM.

5.2 Instrumentation Platforms Supporting Intercession

Dynamic analysis frameworks supporting intercession have been explored for languages other than Wasm. The GraalVM [41] is a polyglot VM that supports instrumentation and dynamic language execution. GraalVM's Truffle framework [40] enables high-performance instrumentation and runtime adaptation [7]. This capability has been used for building a taint analysis platform [16, 15]. In contrast to GraalVM, Wastrumentation performs the instrumentation ahead of time, allowing the instrumented program to run on any Wasm VM, at the cost of not being able to disable instrumentation once the program is executed.

Pin [22], Valgrind [28], and DynamoRIO [4] are dynamic binary instrumentation frameworks for native code analysis. Valgrind provides tools like Memcheck, Callgrind, and Helgrind, which allow intercession and monitoring of x86, ARM, and other instruction set architectures. Its approach of translating code into an intermediate representation enables detailed runtime analysis, including memory checking and profiling. Wastrumentation differs from these approaches for its target domain of applications. The distinct semantics of Wasm, such as its static strong type system and lack of reflection, challenge instrumentation.

6 Conclusion

This paper presented Wastrumentation, a general-purpose source code instrumentation platform for Wasm. Developers can build dynamic analyses using a high-level API that compiles to the Wasm application binary interface supported by Wastrumentation. As a result, analyses can be implemented in *any* language that compiles to Wasm. The platform

then merges the target program and the analysis code into a single Wasm program. The resulting instrumented code can be run in *any* Wasm VM. The high-level API allows for implementing analysis requiring intercession, e.g., altering or skipping operations at the target program. Our evaluation with real-world applications from the WasmR3 benchmark suite shows a competitive performance overhead compared to the state-of-the-art source code instrumentation platforms. We also implemented three novel analyses requiring intercession, which were unattainable with existing source code instrumentation platforms for Wasm.

In future work, we aim to improve the platform's modularity so that analyses that do not need intercession do not incurr any performance costs. Currently, it is not possible to eliminate intercession when it is not needed. We also aim to explore a more fine-grained specification language, enabling analysis developers to specify precisely what needs to be instrumented. This could further reduce performance overhead and the binary size of the instrumented application. Finally, we aim to gain more experience and insights in intercession analyses and explore ways to prevent them from compromising overall integrity or efficiency, e.g., detect analyses that pollute the target linear memory or impose a significant overhead.

References

1 Doehyun Baek, Jakob Getz, Yusung Sim, Daniel Lehmann, Ben L. Titzer, Sukyoung Ryu, and Michael Pradel. Wasm-R3: Record-Reduce-Replay for Realistic and Standalone WebAssembly Benchmarks. *Proc. ACM Program. Lang.*, 8(OOPSLA2):2156–2182, October 2024. doi:10.1145/3689787.

2 Iulia Bastys, Maximilian Algehed, Alexander Sjösten, and Andrei Sabelfeld. SecWasm: Information Flow Control for WebAssembly. In Gagandeep Singh and Caterina Urban, editors, *Static Analysis - 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5-7, 2022, Proceedings*, volume 13790 of *Lecture Notes in Computer Science*, pages 74–103, Cham, 2022. Springer. doi:10.1007/978-3-031-22308-2_5.

3 Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification X2JI3 document 88-002r. *ACM SIGPLAN Notices*, 23(SI):1.1–2.94, 1988. doi:10.1145/885631.885632.

4 Derek Bruening. *Efficient, transparent, and comprehensive runtime code manipulation.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, USA, 2004. AAI0807735. URL: https://hdl.handle.net/1721.1/30160.

5 Shangtong Cao, Ningyu He, Yao Guo, and Haoyu Wang. BREWasm: A General Static Binary Rewriting Framework for WebAssembly. In Manuel V. Hermenegildo and José F. Morales, editors, *Static Analysis - 30th International Symposium, SAS 2023, Cascais, Portugal, October 22-24, 2023, Proceedings*, volume 14284 of *Lecture Notes in Computer Science*, pages 139–163, Cham, 2023. Springer. doi:10.1007/978-3-031-44245-2_8.

6 Laurent Christophe, Elisa Gonzalez Boix, Wolfgang De Meuter, and Coen De Roover. Linvail: A General-Purpose Platform for Shadow Execution of JavaScript. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 260–270. IEEE Computer Society, 2016. doi:10.1109/SANER.2016.91.

7 Michael L. Van de Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. Fast, Flexible, Polyglot Instrumentation Support for Debuggers and other Tools. *Art Sci. Eng. Program.*, 2(3):14, March 2018. doi:10.22152/programming-journal.org/2018/2/14.

8 Aryaz Eghbali and Michael Pradel. DynaPyt: a dynamic analysis framework for Python. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pages 760–771. ACM, 2022. doi:10.1145/3540250.3549126.

9 William Fu, Raymond Lin, and Daniel Inge. TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly. *CoRR*, abs/1802.01050, 2018. doi:10.48550/arXiv.1802.01050.

10 WebAssembly Group. Binaryen: Optimizer and compiler/toolchain library for WebAssembly, 2025. URL: https://github.com/WebAssembly/binaryen.

11 WebAssembly Group. DeNaN.cpp. https://github.com/WebAssembly/binaryen/blob/8c0429ac09d06d6056687e36fd4fb37f61681233/src/passes/DeNaN.cpp#L44-L46, 2025.

12 WebAssembly Group. Multi-Memory Proposal: Multiple per-module memories for Wasm. https://github.com/WebAssembly/multi-memory, 2025.

13 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with WebAssembly. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, 2017. ACM. doi:10.1145/3062341.3062363.

14 Shuyao Jiang, Ruiying Zeng, Zihao Rao, Jiazhen Gu, Yangfan Zhou, and Michael R. Lyu. Revealing Performance Issues in Server-Side WebAssembly Runtimes Via Differential Testing. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 661–672. IEEE, 2023. doi:10.1109/ASE56229.2023.00088.

15 Jacob Kreindl, Daniele Bonetta, David Leopoldseder, Lukas Stadler, and Hanspeter Mössenböck. Polyglot, Label-Defined Dynamic Taint Analysis in TruffleTaint. In Elisa Gonzalez Boix and Tobias Wrigstad, editors, *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*, MPLR '22, pages 152–153, New York, NY, USA, 2022. ACM. doi:10.1145/3546918.3560807.

16 Jacob Kreindl, Daniele Bonetta, Lukas Stadler, David Leopoldseder, and Hanspeter Mössenböck. Low-overhead multi-language dynamic taint analysis on managed runtimes through speculative optimization. In Herbert Kuchen and Jeremy Singer, editors, *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2021, pages 70–87, New York, NY, USA, 2021. ACM. doi:10.1145/3475738.3480939.

17 Daniel Lehmann. Wasabi_Wasm. https://github.com/danleh/wasabi/tree/master/crates/wasabi_wasm, 2025.

18 Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything Old is New Again: Binary Security of WebAssembly. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium (USENIX Security 20)*, pages 217–234. USENIX Association, August 2020. doi:10.5555/3489212.3489225.

19 Daniel Lehmann and Michael Pradel. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 1045–1058, New York, NY, USA, 2019. ACM. doi:10.1145/3297858.3304068.

20 Daniel Lehmann and Michael Pradel. Finding the Dwarf: Recovering Precise Types from WebAssembly Binaries. In Ranjit Jhala and Isil Dillig, editors, *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 410–425. Association for Computing Machinery, 2022. doi:10.1145/3519939.3523449.

21 Zhibo Liu, Dongwei Xiao, Zongjie Li, Shuai Wang, and Wei Meng. Exploring Missed Optimizations in WebAssembly Optimizers. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, pages 436–448, New York, NY, USA, 2023. ACM. doi:10.1145/3597926.3598068.

22 Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM. doi:10.1145/1065010.1065034.

23 Quentin Michaud, Yohan Pipereau, Olivier Levillain, and Dhouha Ayed. Securing Stack Smashing Protection in WebAssembly Applications. *CoRR*, abs/2410.17925, 2024. doi:10.48550/arXiv.2410.17925.

24 Donald Michie. "Memo" functions and machine learning. *Nature*, 218(5136):19–22, April 1968. doi:10.1038/218019a0.

25 Aäron Munsters, Angel Luis Scull Pupo, and Elisa Gonzalez Boix. Wastrumentation: Portable WebAssembly Dynamic Analysis with Support for Intercession (Appendix), 2025. URL: https://soft.vub.ac.be/Publications/2025/vub-tr-soft-25-03.pdf.

26 Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. Thieves in the Browser: Web-based Cryptojacking in the Wild. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ARES '19, pages 4:1–4:10, New York, USA, 2019. ACM. doi:10.1145/3339252.3339261.

27 Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean M. Tullsen, and Deian Stefan. Swivel: Hardening WebAssembly against Spectre. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium (USENIX Security 21)*, pages 1433–1450. USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/narayan.

28 Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM. doi:10.1145/1250734.1250746.

29 João Rodrigues and Jorge Barreiros. Aspect-Oriented Webassembly Transformation. In *2022 17th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–6, 2022. doi:10.23919/CISTI54924.2022.9820136.

30 Alan Romano and Weihang Wang. WasmView: Visual Testing for WebAssembly Applications. In Gregg Rothermel and Doo-Hwan Bae, editors, *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 13–16, Seoul, Korea (South), 2020. ACM. doi:10.1145/3377812.3382155.

31 Mike Rourke. *Learn WebAssembly: Build web applications with native performance using Wasm and C/C++*. Packt Publishing, 2018.

32 Rust and WebAssembly community. Walrus. https://github.com/rustwasm/walrus, 2025.

33 Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498. ACM, 2013. doi:10.1145/2491411.2491447.

34 Suhyeon Song, Seonghwan Park, and Donghyun Kwon. metaSafer: A Technique to Detect Heap Metadata Corruption in WebAssembly. *IEEE Access*, 11:124887–124898, 2023. doi:10.1109/ACCESS.2023.3327817.

35 Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. Efficient dynamic analysis for Node.js. In Christophe Dubach and Jingling Xue, editors, *Proceedings of the 27th International Conference on Compiler Construction*, CC '18, pages 196–206. ACM, 2018. doi:10.1145/3178372.3179527.

36 Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. Taint Tracking for WebAssembly. *CoRR*, abs/1807.08349, 2018. doi:10.48550/arXiv.1807.08349.

37 Ben L. Titzer. Virgil: objects on the head of a pin. In Peri L. Tarr and William R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, volume 41, pages 191–208. ACM, October 2006. doi:10.1145/1167473.1167489.

38 Ben L. Titzer, Elizabeth Gilbert, Bradley Wei Jie Teo, Yash Anand, Kazuyuki Takayama, and Heather Miller. Flexible Non-intrusive Dynamic Instrumentation for WebAssembly. In Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir, editors, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, pages 398–415, New York, NY, USA, 2024. ACM. doi:10.1145/3620666.3651338.

39 Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W. Hamlen, and Shuang Hao. SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks. In Javier López, Jianying Zhou, and Miguel Soriano, editors, *Computer Security*, volume 11099 of *Lecture Notes in Computer Science*, pages 122–142, Cham, 2018. Springer. doi:10.1007/978-3-319-98989-1_7.

40 Christian Wimmer and Thomas Würthinger. Truffle: a self-optimizing runtime system. In Gary T. Leavens, editor, *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 13–14, New York, NY, USA, 2012. ACM. doi:10.1145/2384716.2384723.

41 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. ACM. doi:10.1145/2509578.2509581.

42 Ziyao Zhang, Wenlong Zheng, Baojian Hua, Qiliang Fan, and Zhizhong Pan. VMCanary: Effective Memory Protection for WebAssembly via Virtual Machine-assisted Approach. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*, pages 662–671. IEEE, 2023. doi:10.1109/QRS60937.2023.00070.

43 Wenxuan Zhao, Ruiying Zeng, and Yangfan Zhou. Wapplique: Testing WebAssembly Runtime via Execution Context-Aware Bytecode Mutation. In Maria Christakis and Michael Pradel, editors, *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, pages 1035–1047. ACM, 2024. doi:10.1145/3650212.3680340.

Fair Termination of Asynchronous Binary Sessions

Luca Padovani
University of Bologna, Italy

Gianluigi Zavattaro
University of Bologna, Italy
INRIA OLAS team, Sophia Antipolis, France

Abstract

We study a theory of asynchronous session types ensuring that well-typed processes terminate under a suitable fairness assumption. Fair termination entails *starvation freedom* and *orphan message freedom* namely that all messages, including those that are produced *early* taking advantage of asynchrony, are eventually consumed. The theory is based on a novel *fair asynchronous subtyping* relation for session types that is coarser than the existing ones. The type system is also the first of its kind that is firmly rooted in linear logic: fair asynchronous subtyping is incorporated as a natural generalization of the cut and axiom rules of linear logic and asynchronous communication is modeled through a suitable set of *commuting conversions* and of *deep cut reductions* in linear logic proofs.

2012 ACM Subject Classification Theory of computation → Type structures; Theory of computation → Linear logic; Theory of computation → Program analysis

Keywords and phrases Binary sessions, fair asynchronous subtyping, fair termination, linear logic

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.24

Related Version *Full Version*: `https://doi.org/10.48550/arXiv.2503.07273` [43]

Funding The authors have been partially supported by the ANR project SmartCloud ANR-23-CE25-0012 and the PNRR project CN–HPC Spoke 9 Innovation Grant RTMER.

Acknowledgements We are grateful to the ECOOP reviewers for their thoughtful feedback and questions.

1 Introduction

Session type systems [32, 33, 35] have become a widespread formalism for ensuring a variety of safety and liveness properties of communicating processes through static analysis. Session types specify the *sequences* of messages that can be sent over a channel, and the type system makes sure that (1) well-typed processes comply with this protocol specification and that (2) the peer endpoints of the channel are used according to compatible protocols.

Many session type systems are defined for a synchronous calculus or language even if the actual underlying communication model is meant to be asynchronous. The point is that synchronous theories of session types are simpler and easier to work with and most if not all properties of a synchronous session type system hold even if the actual communication model is asynchronous. However, awareness of the communication model can help relaxing the type system and thus enlarging the family of well-typed processes. This observation has led to the study of *asynchronous subtyping relations* for session types [40, 39, 11] allowing processes to *anticipate* output messages with respect to the protocol specification they are expected to comply with, provided that anticipated outputs *do not depend* on incoming messages. This apparent violation of the protocol specification enabled by asynchronous subtyping is harmless precisely because output actions are non-blocking in an asynchronous setting.

The available session type systems based on asynchronous subtyping focus on the enforcement of safety properties but struggle at ensuring liveness properties of many simple communication patterns. As an illustration of such patterns consider a server that, in order

© Luca Padovani and Gianluigi Zavattaro;
licensed under Creative Commons License CC-BY 4.0

to fulfill a request received from a session x, splits the request into an arbitrary number of tasks handled by a separate worker and then gathers the partial results to answer the client. We might be interested in establishing whether the client eventually receives a response.

To be more concrete, let us model the server as the term

$$x \triangleright \{\mathsf{req} : (y)(\mathit{Split}\langle x, y\rangle \mid \mathit{Worker}\langle y\rangle)\}$$

which indicates the input of the request followed by the spawning of a *Split* process and of a *Worker* process connected by a new session y.

The *Split* process is modeled by the following definitions:

$$\mathit{Split}(x, y) = y \triangleleft \mathsf{task}.\mathit{Split}\langle x, y\rangle \oplus y \triangleleft \mathsf{stop}.\mathit{Gather}\langle x, y\rangle \tag{1}$$

$$\mathit{Gather}(x, y) = y \triangleright \{\mathsf{res} : \mathit{Gather}\langle x, y\rangle, \mathsf{stop} : \mathsf{wait}\ y.x \triangleleft \mathsf{resp}.\mathsf{close}\ x\} \tag{2}$$

according to which the server sends to the worker a non-deterministically chosen number of tasks followed by a stop on session y, it then gathers from the worker an arbitrary number of results followed by a stop, and finally sends back to the client a response on session x.

One possible modeling of the worker process is according to the definition

$$\mathit{Worker}(y) = y \triangleright \{\mathsf{task} : y \triangleleft \mathsf{res}.\mathit{Worker}\langle y\rangle, \mathsf{stop} : y \triangleleft \mathsf{stop}.\mathsf{close}\ y\} \tag{3}$$

so that the worker sends a result for each task it receives. The eye-catching aspect of *Worker* is that it does not interact according to the "complementary" protocol implemented on the server side, at least not in the sense that is usually intended in (synchronous) session type theories. Indeed, while the server first sends all the tasks *and then* gathers all the results, the worker eagerly sends one result after receiving each task.

In an asynchronous setting, the fact that *Worker* sends some messages earlier than expected is not an issue since output actions are non-blocking. However, we would like to be sure that these early results do not keep accumulating and are eventually consumed by *Gather*. There is nothing in the modeling of *Split*, *Gather* and *Worker* that prevents this from happening, but *we can prove the eventual consumption of every* result *only under the assumption that sooner or later Split will send* stop *to Worker*. If we broaden our viewpoint, we see that the same assumption is necessary to prove that the client interacting with the server does not starve. While the server is running, the client is awaiting for a response from session x. In order to *prove* that the client will eventually receive a response, we have to assume that *Worker* will terminate the session y, which in turn requires the assumption that *Split* will eventually send stop to *Worker*. In general, the proof of any non-trivial liveness property that concerns the eventual production or consumption of a message on a certain session may require the assumption that every other session eventually terminates. For this reason, ensuring the eventual termination of sessions should be a primary goal of any session type system aimed at enforcing liveness properties. As we have seen in the discussion above, proving the eventual termination of a session may require some *fairness assumptions* like the fact that *Split* will eventually stop sending tasks. For this reason, in the literature such eventual termination property is referred to as *fair termination* [30, 26, 2].

In this work we study a theory of asynchronous session types and an associated session type system ensuring that well-typed processes are fairly terminating under a suitable fairness assumption. The fair termination of processes entails the fair termination of the sessions they operate on, hence that all messages produced – including those sent earlier than expected – are eventually consumed and that all processes waiting for a message eventually receive one. In other words, the very same type system prevents *process starvation* and ensures the absence of *orphan messages*. Related works achieve these goals only partially:

- The theories of asynchronous session types developed by Mostrous *et al.* [40, 39], Chen *et al.* [11] and Ghilezan *et al.* [29] require that every output can be only anticipated with respect to finitely many inputs. Since there is no finite upper bound to the number of tasks that *Split* can produce, each early res produced by *Worker* anticipates an unbounded number of inputs leaving *Worker* out of reach for the aforementioned theories.
- Bravetti, Lange and Zavattaro [8] study an asynchronous subtyping relation for session types that allows early outputs to anticipate an unbounded number of inputs. However, their subtyping relation disallows any *covariance* of outputs, which is needed to account for the fact that the behavior of *Worker* is *more deterministic* than the behavior expected by *Gather*. Indeed, while *Gather* expects to receive an arbitrary number of results, *Worker* produces exactly as many results as the number of tasks it receives from *Split*.
- None of the aforementioned works provides guarantees on the eventual fulfillment of the client's request, either because they do not make sure that every session eventually terminates [11, 29] or because they do not take multiple sessions into account [29, 8].
- Ciccone, Dagnino and Padovani [14, 16, 13] study session type systems ensuring the fair termination of sessions, but their works are based on synchronous communication models that do not support any form of output anticipation like the one exemplified by *Worker*.

The theory of asynchronous session types that we propose in this paper addresses all these limitations. In addition, we also make the following technical contributions:

- We define a *fair asynchronous subtyping relation* for session types that is coarser than those in the literature for the family of eventually terminating session types. Unlike many existing fair/asynchronous subtyping relations [39, 14, 13, 8] our subtyping relation is *closed under duality*. This property is key for proving the type system sound.
- We give the first *fair asynchronous semantics* of session types using a labelled transition system (LTS) defined by *bounded coinduction* [1, 18]. The adoption of this semantics allows us to characterize fair asynchronous subtyping in a way that is structurally the same as the one for the well-known synchronous subtyping defined by Gay and Hole [28].
- Our theory of asynchronous session types with fair asynchronous subtyping is the first one where the process model and the type system are rooted in linear logic [45, 10, 37]. We incorporate fair asynchronous subtyping in the type system as generalized forms of the cut and linear logic axiom thanks to the aforementioned closure under duality. Also, instead of introducing explicit message buffers, we model asynchronous communications by means of suitable *commuting conversions* and *deep cut reductions* in linear logic proofs.

Structure of the paper. Section 2 describes our calculus of asynchronous processes and the properties we enforce. Section 3 introduces the semantics of asynchronous session types while Section 4 studies fair asynchronous subtyping and its properties. Section 5 describes the type system and Section 6 illustrates some common usage patters of fair asynchronous subtyping. Section 7 shows how our LTS can be easily tailored to characterize various asynchronous subtyping relations that appear in the literature and compares them with our subtyping relation. Section 8 discusses related work in more detail and Section 9 recaps and outlines future work. Proofs and additional technical material are provided in the full paper [43].

2 A Calculus of Asynchronous Processes

In this section we present syntax and semantics of a calculus of asynchronous processes called CaP and we formulate the safety and liveness properties that our type system guarantees. At the surface level, CaP closely resembles other calculi of binary sessions based on linear logic

Table 1 Syntax of CaP.

P, Q	::=	**Process**	|	done	termination
	$x \leftrightarrow y$	link	|	$A\langle \overline{x} \rangle$	invocation
|	close x	signal output	|	wait $x.P$	signal input
|	$x \lhd \mathsf{a}.P$	tag output	|	$x \rhd \{\mathsf{a}_i : P_i\}_{i \in I}$	tag input
|	$x(y)[P].Q$	channel output	|	$x(y).P$	channel input
|	$P \oplus Q$	choice	|	$(x)(P \mid Q)$	composition

such as CP [45] or μCP [37]. The main differences between CaP and these calculi are that CaP supports general recursion, it includes a non-deterministic choice operator and above all it models asynchronous communication by giving a non-blocking semantics to output actions, which essentially act like message buffers.

The syntax of *processes* is shown in Table 1 and makes use of an infinite set of *channels* ranged over by x, y and z, a set Tags of *tags* ranged over by a, b, ... and a set of *process names* ranged over by A, B, etc. The terminated process, which performs no actions, is denoted by done. The term $A\langle \overline{x} \rangle$, where $\overline{x}$ is a possibly empty sequence of channels, represents the *invocation* of the process named A. We assume that for each such invocation there exists a unique global definition of the form $A(\overline{x}) = P$ that gives meaning to the name A. We also assume that all invocations occur guarded by a prefix or by a non-deterministic choice. The term $x \leftrightarrow y$ models a *link*, that is a process that forwards every message received from x to y and vice versa, effectively unifying the two channels. Links are typical of calculi based on linear logic since their typing rule correspond to the axiom of the logic. The terms close x and wait $x.P$ model processes that respectively send and receive a termination signal on x. Note that close x has no continuation, as is the case in most calculi based on linear logic, whereas wait $x.P$ continues as P. The term $x(y)[P].Q$ models a bifurcating session: it creates a new channel y, sends y on x, forks a new process P that uses y, and then continues as Q. The process $x(y).P$ waits for a channel y from x and then continues as P. The terms $x \lhd \mathsf{a}.P$ and $x \rhd \{\mathsf{a}_i : P_i\}_{i \in I}$ model processes that respectively send and receive a tag. The sender selects one particular tag a to send. The receiver continues as P_i depending on the tag a_i that it receives. In the examples we sometimes use tags as an abstract representation of more complex messages, such as requests or tasks. The term $(x)(P \mid Q)$ models the parallel composition of two processes P and Q connected by the channel x. We often refer to this term as a *cut*, since its typing rule coincides with the cut rule of linear logic. Finally, the term $P \oplus Q$ models the non-deterministic choice between P and Q.

It is known that links in conjunction with bifurcating sessions can be used to encode *session delegation*, whereby processes exchange an *existing* (rather than new) channel z on x. This behavior can be modeled by a term of the form $x(y)[y \leftrightarrow z].P$. As we will see in Section 6, CaP links also act as *explicit casts* enabling useful forms of subsumption.

The notions of *free* and *bound* channels for processes are defined as expected with the proviso that a process of the form $x(y)[P].Q$ binds y in P but not in Q. We identify processes up to renaming of bound channels and we write $\mathsf{fn}(P)$ for the set of channels occurring free in P. Also, for every global definition $A(\overline{x}) = P$ we assume $\mathsf{fn}(P) = \{\overline{x}\}$.

We have anticipated that CaP adopts an asynchronous communication model. In practice this would be implemented by FIFO buffers storing messages that have been produced but not consumed. In CaP, we model asynchrony giving a non-blocking semantics to the output actions $x(y)[Q].P$ and $x \lhd \mathsf{a}.P$. That is, we allow the continuation P to reduce and/or

Table 2 Structural pre-congruence and reduction semantics of CaP.

$$
\begin{array}{lrcll}
\text{[S-CALL]} & A\langle \overline{x} \rangle & \sqsupseteq & P & A(\overline{x}) = P \\
\text{[S-LINK]} & x \leftrightarrow y & \sqsupseteq & y \leftrightarrow x & \\
\text{[S-COMM]} & (x)(P \mid Q) & \sqsupseteq & (x)(Q \mid P) & \\
\text{[S-WAIT]} & (x)(\mathcal{B}_x[\mathsf{wait}\ y.P] \mid Q) & \sqsupseteq & \mathsf{wait}\ y.(x)(\mathcal{B}_x[P] \mid Q) & x \neq y \\
\text{[S-CASE]} & (x)(\mathcal{B}_x[y \rhd \{\mathsf{a}_i : P_i\}_{i \in I}] \mid Q) & \sqsupseteq & y \rhd \{\mathsf{a}_i : (x)(\mathcal{B}_x[P_i] \mid Q)\}_{i \in I} & x \neq y \\
\text{[S-JOIN]} & (x)(\mathcal{B}_x[y(z).P] \mid Q) & \sqsupseteq & y(z).(x)(\mathcal{B}_x[P] \mid Q) & x \neq y, z \notin \mathsf{fn}(\mathcal{B}_x) \\
\text{[S-PULL-0]} & (x)(\mathcal{B}_x[P] \mid \mathcal{B}'_x[x \leftrightarrow y]) & \sqsupseteq & \mathcal{B}_y[(x)(P \mid \mathcal{B}'_x[x \leftrightarrow y])] & \\
\text{[S-PULL-1]} & (x)(y(z)[P].Q \mid R) & \sqsupseteq & y(z)[(x)(P \mid R)].Q & x \neq y, x \in \mathsf{fn}(P) \\
\text{[S-PULL-2]} & (x)(\mathcal{B}_y[P] \mid Q) & \sqsupseteq & \mathcal{B}_y[(x)(P \mid Q)] & x \neq y, x \notin \mathsf{fn}(\mathcal{B}_y) \\
\text{[S-PULL-3]} & \mathcal{B}_x[y(z)[P].Q] & \sqsupseteq & y(z)[\mathcal{B}_x[P]].Q & x \neq y, x \in \mathsf{fn}(P) \\
\text{[S-PULL-4]} & \mathcal{B}_x[\mathcal{B}'_y[P]] & \sqsupseteq & \mathcal{B}'_y[\mathcal{B}_x[P]] & x \neq y, x \in \mathsf{fn}(P)
\end{array}
$$

$$
\begin{array}{lrcll}
\text{[R-CHOICE]} & P_1 \oplus P_2 & \to & P_k & k \in \{1, 2\} \\
\text{[R-LINK]} & (x)(x \leftrightarrow y \mid P) & \to & P\{y/x\} & \\
\text{[R-CLOSE]} & (x)(\mathsf{close}\ x \mid \mathsf{wait}\ x.P) & \to & P & \\
\text{[R-SELECT]} & (x)(x \lhd \mathsf{a}_k.P \mid \mathcal{B}_x[x \rhd \{\mathsf{a}_i : Q_i\}_{i \in I}]) & \to & (x)(P \mid \mathcal{B}_x[Q_k]) & k \in I \\
\text{[R-FORK]} & (x)(x(y)[P].Q \mid \mathcal{B}_x[x(y).R]) & \to & (y)(P \mid (x)(Q \mid \mathcal{B}_x[R])) & \\
\text{[R-CUT]} & (x)(P \mid R) & \to & (x)(Q \mid R) & P \to Q \\
\text{[R-BUFFER]} & \mathcal{B}_x[P] & \to & \mathcal{B}_x[Q] & P \to Q \\
\text{[R-STR]} & P & \to & Q & P \sqsupseteq R \to Q
\end{array}
$$

interact with other sub-processes even if the prefix has not been consumed. In a sense, we consider the prefixes $x(y)[Q]$ and $x \lhd \mathsf{a}$ as *floating messages* or parts of a *buffer* associated with channel x. In general, we call *buffer* any term generated by the following grammar:

Buffer $\quad \mathcal{B}_x ::= [\,] \mid x \lhd \mathsf{a}.\mathcal{B}_x \mid x(y)[P].\mathcal{B}_x$

A buffer is either empty, represented by a *hole* $[\,]$, or a tag a sent on x followed by a buffer for x, or a fresh channel y (with associated process P) sent on x and followed by a buffer for x. Note that the annotation x in the metavariable $\mathcal{B}_x$ is meant to *bind* the channel on which the messages in the buffer have been sent. Therefore, having at our disposal a buffer $\mathcal{B}_x$, we can write $\mathcal{B}_y$ for the buffer that has the same structure as $\mathcal{B}_x$ but where x has been replaced by y. Buffers vaguely resemble *reduction contexts*, except that they allow us to place a hole *behind* output prefixes since these are meant to be non-blocking. Hereafter we write $\mathcal{B}_x[P]$ for the process obtained by replacing the hole in $\mathcal{B}_x$ with P.

The operational semantics of CaP is given by a structural pre-congruence relation $\sqsupseteq$ and a reduction relation $\to$, both defined in Table 2. In simple words, structural pre-congruence relates processes that are essentially equivalent except for the order of independent actions, while reduction describes communications and the resolution of non-deterministic choices.

We can roughly classify the rules for structural pre-congruence in four groups. The first group contains [S-CALL], [S-LINK] and [S-COMM], which capture expected properties of process invocations, links and parallel compositions: a process invocation $A\langle \overline{x} \rangle$ is indistinguishable from P if $A(\overline{x}) = P$; a link $x \leftrightarrow y$ is indistinguishable from $y \leftrightarrow x$; a cut $(x)(P \mid Q)$ is indistinguishable from $(x)(Q \mid P)$, that is parallel composition is commutative.

The second group of rules contains [S-WAIT], [S-CASE] and [S-JOIN]. These rules allow an input action on some channel y to be extruded from a cut on x when $x \neq y$. The purpose of these transformations is to move inputs on y close to outputs on y, so as to enable interactions

in the session y. These transformations correspond to well-known rearrangements of linear logic proofs (for example, they are sometimes referred to as *external reductions* [3, 25]), except that in CaP we allow the input prefix to be found *within an arbitrary buffer* $\mathcal{B}_x$, coherently with the intuition that (output) actions in a buffer are non-blocking. The side condition $z \notin \mathsf{fn}(\mathcal{B}_x)$ in [S-JOIN] makes sure that free occurrences of z in $\mathcal{B}_x$ are not accidentally captured by the binding prefix. Since we consider processes (and buffers) modulo renaming of bound names and there is an infinite supply of names, the bound z can always be renamed so as to enable this process transformation.

The third group of rules contains [S-PULL-0], [S-PULL-1] and [S-PULL-2]. These rules are similar to those of the second group, except that they allow whole *buffers* of messages on y to float through cuts on x when $x \neq y$. [S-PULL-2] is simple and speaks for itself while [S-PULL-1] deals with the case in which the name x bound by the cut occurs in the process P associated with a fresh channel z sent on $y \neq x$. In this case the cut as a whole becomes associated with z. In principle we should also specify the side condition $x \notin \mathsf{fn}(Q)$, but this condition holds for well-typed processes when $x \in \mathsf{fn}(P)$. The rule [S-PULL-0] covers a somewhat peculiar case of [S-PULL-2], whereby a buffer $\mathcal{B}_x$ can be extruded from the cut on x because there is a link $x \leftrightarrow y$ that acts as a forwarder from x to y. Note that the extruded buffer $\mathcal{B}_x$ turns into $\mathcal{B}_y$, so as to reflect the forwarding effect of the link.

The fourth and final group of rules contains [S-PULL-3] and [S-PULL-4]. These rules allow buffers for different channels to be permuted. Again [S-PULL-3] deals with the special case in which a channel x occurs in the process P associated with a fresh channel z.

In the definition of structural pre-congruence there are some glaring omissions (*e.g.* associativity of parallel composition) and very few rules are invertible. This is not because the missing rules would be unsound, but because they turn out to be unnecessary for proving that well-typed processes are deadlock free and fairly terminating.

Base reductions consist of [R-CHOICE], [R-CLOSE], [R-SELECT] and [R-FORK] which respectively model the reduction of a non-deterministic process, the termination of the session x and the consumption of tags and channels sent on channel x. The rules are almost standard, except that [R-SELECT] and [R-FORK] allow input actions on x to operate from within arbitrary buffers for x. The buffers represent asynchronously sent messages that do not block subsequent actions. At the logical level, these interactions correspond to *deep cut reductions* in a sense that resembles *deep inference* [31], whereby logical rules can be applied deep within a context. Unlike [R-SELECT] and [R-FORK], there is no buffer around $\mathsf{wait}\ x.P$ in [R-CLOSE]. This rule implies that a session cannot be closed unless all the messages (asynchronously) produced therein have also been consumed. Note that this property is enforced by the type system and is not meant to be checked at runtime. Rules [R-CUT], [R-BUFFER] and [R-STR] propagate reductions across cuts and buffers and close them by structural pre-congruence.

Hereafter we write $\to^*$ for the reflexive, transitive closure of $\to$. We write $P \to$ if $P \to Q$ for some Q and $P \not\to$ if not $P \to$.

We can now formally define the safety and liveness properties we are interested in.

▶ **Definition 1** (deadlock freedom). *We say that P is* deadlock free *if for every Q such that $P \to^* Q \not\to$ we have $Q \sqsupseteq \mathsf{done}$.*

Deadlock freedom is an instance of safety property. A deadlock-free process either reduces or it is (structurally pre-congruent to) done. For example, the process $\Omega() = \Omega\langle\rangle \oplus \Omega\langle\rangle$ is deadlock free (we have $\Omega\langle\rangle \to \Omega\langle\rangle$) whereas $(x)(\mathsf{close}\ y \mid \mathsf{wait}\ x.\mathsf{done})$ is deadlocked. When a deadlock-free process stops reducing, it contains no pending actions and all of its sessions have been closed. In particular, all the messages in buffers have been consumed.

The liveness properties we are interested in are related to *termination*, of which there exist several variants. A *reduction sequence* of P is a sequence $(P_0, P_1, \dots)$ such that $P_0 = P$ and $P_i \to P_{i+1}$ whenever $i+1$ is not greater than the length of the sequence. A *run* is a maximal reduction sequence, in the sense that either it is infinite or the last process in the sequence (say P_n) does not reduce (that is, $P_n \not\to$). We say that P is *weakly terminating* if it has a finite run, that P is *terminating* if every run of P is finite, and that P is *diverging* if every run of P is infinite. For example, $\Omega\langle\rangle \oplus \mathsf{done}$ is weakly terminating but not terminating, $\mathsf{done} \oplus \mathsf{done}$ is terminating, and $\Omega\langle\rangle$ is diverging. Note that here we call "termination" the mere inability to reduce further and not the fact that a process has become done. For example, done, close x and $(x)(\mathsf{close}\ y \mid \mathsf{wait}\ x.\mathsf{done})$ are all terminated (they do not reduce), but only done is also deadlock free. So it really is the combination of deadlock freedom (Definition 1) and some termination property that we wish to enforce with our type system.

The termination property we target in this work is called *fair termination* [30, 26, 2]. Fair termination consists of those processes such that all of their infinite runs are considered to be unrealistic or *unfair* and therefore can be ignored insofar termination is concerned. We could say that these processes may diverge in principle, but they terminate in practice. Clearly, fair termination depends on a *fairness notion* that discriminates *fair* runs from *unfair* ones. Among all fairness notions, here we consider a particular instance of *full fairness* [44].

▶ **Definition 2.** *A run is* fair *if it contains finitely many weakly terminating processes.*

Remember that a run is a *maximal* reduction sequence of a process. So, along a fair run the process only has finitely many chances to terminate. This can happen either because the run is finite (the process eventually terminates) or because the run contains a diverging process (at some point termination is no longer possible). For example, the infinite run $(\Omega\langle\rangle \oplus \mathsf{done}, \Omega\langle\rangle, \dots)$ is fair because only the first process in it is weakly terminating. We find it useful to also look at the *negation* of the notion of fair run: an *unfair run* is necessarily infinite and contains infinitely many weakly terminating processes. In other words, an unfair run describes a computation along which termination is always reachable, but it is never reached as if the process is avoiding it on purpose. For example, if $A() = A\langle\rangle \oplus \mathsf{done}$ then the infinite run $(A\langle\rangle, A\langle\rangle, \dots)$ is unfair because done is always reachable but never reached.

▶ **Definition 3.** *We say that P is* fairly terminating *if every* fair *run of P is finite.*

For example, $A() = A\langle\rangle \oplus \mathsf{done}$ is fairly terminating whereas $\Omega\langle\rangle \oplus \mathsf{done}$ is not because it has an infinite fair run. There are two reasons why full fairness is a suitable fairness assumption in our setting. First, full fairness has been shown to be the *strongest* conceivable fairness assumption [44], which means that it allows us to target the *largest* family of fairly terminating processes. Second, it has been observed [12] that this family admits the following alternative characterization which does not mention fair runs at all.

▶ **Theorem 4.** *P is fairly terminating iff each Q such that $P \to^* Q$ is weakly terminating.*

The relevance of this characterization rests in the fact that it provides the key proof method for the soundness of our type system. Indeed, suppose that the type system ensures that well-typed processes *weakly* terminate. We expect the type system to also enjoy subject redution, namely the property that well-typed processes always reduce to well-typed processes. But then, using the right-to-left implication in Theorem 4, the very same type system also ensures that well-typed processes fairly terminate.

▶ **Example 5.** Consider the *Split* process defined in (1). We derive

$$\mathit{Split}\langle x, y\rangle \to y \lhd \mathsf{task}.\mathit{Split}\langle x, y\rangle \to \cdots \to (y \lhd \mathsf{task})^n.\mathit{Split}\langle x, y\rangle$$

where $(y \lhd \mathsf{task})^n$ denotes n subsequent $y \lhd \mathsf{task}$ prefixes, using repeated applications of [R-CHOICE] and [R-BUFFER]. Notice how the messages pile up as the process reduces and also that, at any time, the process may reduce to $(y \lhd \mathsf{task})^n.y \lhd \mathsf{stop}.\mathit{Gather}\langle x, y\rangle$ which is weakly terminating.That is, $\mathit{Split}\langle x, y\rangle$ is fairly terminating by Theorem 4. We also have

$$(y)(\mathit{Split}\langle x, y\rangle \mid \mathit{Worker}\langle y\rangle) \to^* (y)((y \lhd \mathsf{task})^n.\mathit{Split}\langle x, y\rangle \mid (y \lhd \mathsf{res})^m.\mathit{Worker}\langle y\rangle)$$

for every n and m, indicating that *Split* has produced $m+n$ tasks and *Worker* has consumed only m of them. Since *Split* can always send stop and consume all the res messages produced by *Worker*, we also have

$$(y)((y \lhd \mathsf{task})^n.\mathit{Split}\langle x, y\rangle \mid (y \lhd \mathsf{res})^m.\mathit{Worker}\langle y\rangle) \to^* x \lhd \mathsf{resp.done} \not\to$$

indicating that $(y)(\mathit{Split}\langle x, y\rangle \mid \mathit{Worker}\langle y\rangle)$ too is fairly terminating by Theorem 4. ⌟

3 Asynchronous Session Types

3.1 Syntax

Session types are generated by the productions below

Session type $\quad S, T ::= \mathbf{1} \mid \bot \mid \oplus\{\mathsf{a}_i : S_i\}_{i\in I} \mid \&\{\mathsf{a}_i : S_i\}_{i\in I} \mid S \otimes T \mid S \parr T$

and adhere to the usual interpretation given to propositions of multiplicative additive linear logic (MALL) [10, 45]: the constants $\mathbf{1}$ and $\bot$ describe processes respectively sending and receiving a termination signal; the additive connectives $\oplus\{\mathsf{a}_i : S_i\}_{i\in I}$ and $\&\{\mathsf{a}_i : S_i\}_{i\in I}$ describe processes sending and receiving a tag a_i and then behaving according to S_i; the multiplicative connectives $S \otimes T$ and $S \parr T$ describe processes exchanging a channel of type S and then behaving according to T.

Compared to the usual linear logic propositions, we observe the following differences:

- The additive connectives are n-ary instead of binary and make use of explicit tags for improved generality and readability. In each additive connective $\oplus\{\mathsf{a}_i : S_i\}_{i\in I}$ and $\&\{\mathsf{a}_i : S_i\}_{i\in I}$ we assume that the set I is finite and that the tags a_i are pairwise disjoint.
- The additive constants $\mathbf{0} \stackrel{\text{def}}{=} \oplus\{\}$ and $\top \stackrel{\text{def}}{=} \&\{\}$ are defined as degenerate (empty) versions of the additive connectives instead of being built-in.
- The productions shown above are meant to be interpreted *coinductively*. That is, we consider session types the possibly infinite *regular* trees built using the above productions. We define possibly infinite session types as solutions of equations of the form $S = \cdots$ where the metavariable S may occur (guarded) on the right hand side of "=". It is a known fact that every such finite system of equations admits a unique regular solution [17].

The *dual* of a session type S, denoted by $S^\bot$, describes the mirrored protocol of S and is corecursively defined by following equations:

$$\mathbf{1}^\bot = \bot \qquad \oplus\{\mathsf{a}_i : S_i\}_{i\in I}^\bot = \&\{\mathsf{a}_i : S_i^\bot\}_{i\in I} \qquad (S \otimes T)^\bot = S^\bot \parr T^\bot$$
$$\bot^\bot = \mathbf{1} \qquad \&\{\mathsf{a}_i : S_i\}_{i\in I}^\bot = \oplus\{\mathsf{a}_i : S_i^\bot\}_{i\in I} \qquad (S \parr T)^\bot = S^\bot \otimes T^\bot$$

We say that the session types of the form $\mathbf{1}$, $\oplus\{\mathsf{a}_i : S_i\}_{i\in I}$ and $S \otimes T$ are *positive*, whereas the session types of the form $\bot$, $\&\{\mathsf{a}_i : S_i\}_{i\in I}$ and $S \parr T$ are *negative*. Positive session types describe protocols that begin with an *output action*, whereas negative session types describe protocols that begin with an *input action*. We write $\mathsf{pos}(S)$ and $\mathsf{neg}(S)$ to state that S is positive and negative, respectively.

Table 3 Labelled transition system for asynchronous session types.

$$[\text{MUST-}\mathbf{1}]\ \frac{}{\mathbf{1} \xrightarrow{!*} \mathbf{1}} \qquad [\text{MUST-}\otimes]\ \frac{}{S \otimes T \xrightarrow{!S} T} \qquad [\text{MAY-}\otimes]\ \frac{T \xrightarrow{?\sigma} T'}{S \otimes T \xrightarrow{?\sigma} S \otimes T'}$$

$$[\text{MUST-}\bot]\ \frac{}{\bot \xrightarrow{?*} \bot} \qquad [\text{MUST-}\parr]\ \frac{}{S \parr T \xrightarrow{?S} T} \qquad [\text{MAY-}\parr]\ \frac{T \xrightarrow{!\sigma} T'}{S \parr T \xrightarrow{!\sigma} S \parr T'}$$

$$[\text{MUST-}\oplus]\ \frac{}{\oplus\{\mathsf{a}_i : S_i\}_{i\in I} \xrightarrow{!\mathsf{a}_k} S_k} \qquad [\text{MAY-}\oplus]\ \frac{\forall i \in I : S_i \xrightarrow{?\sigma} T_i}{\oplus\{\mathsf{a}_i : S_i\}_{i\in I} \xrightarrow{?\sigma} \oplus\{\mathsf{a}_i : T_i\}_{i\in I}} \qquad [\text{FAIR-}\oplus]\ \frac{\exists k \in I : S_k \xrightarrow{?\sigma} T_k}{\oplus\{\mathsf{a}_i : S_i\}_{i\in I} \xrightarrow{?\sigma} \oplus\{\mathsf{a}_i : T_i\}_{i\in I}}$$

$$[\text{MUST-}\&]\ \frac{}{\&\{\mathsf{a}_i : S_i\}_{i\in I} \xrightarrow{?\mathsf{a}_k} S_k} \qquad [\text{MAY-}\&]\ \frac{\forall i \in I : S_i \xrightarrow{!\sigma} T_i}{\&\{\mathsf{a}_i : S_i\}_{i\in I} \xrightarrow{!\sigma} \&\{\mathsf{a}_i : T_i\}_{i\in I}} \qquad [\text{FAIR-}\&]\ \frac{\exists k \in I : S_k \xrightarrow{!\sigma} T_k}{\&\{\mathsf{a}_i : S_i\}_{i\in I} \xrightarrow{!\sigma} \&\{\mathsf{a}_i : T_i\}_{i\in I}}$$

3.2 Fair Asynchronous Semantics

We define the labelled transition system (LTS) for session types using the rules in Table 3. Labels of the transition system can be of the form $?\sigma$ (input of a message of type σ) or $!\sigma$ (output of a message of type σ) where σ is a *message type* of the form $*$ (the type of a termination signal), a (the singleton type of the tag a) or S (the session type of a channel). Hereafter we use σ and τ to range over message types and α and β to range over labels.

Before we describe the rules in detail, we must point out two unusual but important aspects of the LTS. First of all, the LTS is specified as a *Generalized Inference System* (GIS for short [1, 18]). A GIS consists of two sets of rules, those that are meant to be interpreted coinductively (the singly-lined rules in Table 3) and those that are meant to be interpreted inductively (the singly-lined rules plus the doubly-lined rules in Table 3). If we call $\xrightarrow{\alpha}_{\mathsf{ind}}$ the relation that is defined by the inductive part of the GIS, then the actual relation $\xrightarrow{\alpha}$ being defined is the largest one included in $\xrightarrow{\alpha}_{\mathsf{ind}}$ that satisfies the singly-lined rules in Table 3. The interested reader may refer to the literature for a thorough presentation of GIS [1, 18], but the examples we are about to discuss should suffice to clarify the nature of transitions.

The other unusual aspect of the LTS is that a transition $S \xrightarrow{\alpha} T$ is not an indication of what a process complying with S necessarily does, but rather of what the process is allowed or able to do. In particular, a transition $S \xrightarrow{!\sigma} T$ means that a process complying with S is allowed to output a message of type σ, even though S may be negative. We say that this is an *early output transition* because it describes an output that may occur ahead of time. Dually, a transition $S \xrightarrow{?\sigma} T$ means that a process complying with S is eventually able to input a message of type σ, even though S may be positive. We say that this is a *late input transition* because it describes the consumption of a message that may occur later on.

The axioms [MUST-*] are used to derive what we call *immediate transitions*. These are the expected transitions of session types, whereby no input is late and no output is early. For technical reasons it is convenient to have transitions also for $\mathbf{1}$ and $\bot$. In this way we do not have to distinguish $\mathbf{1}$ and $\bot$ from other non-terminated protocols when defining our notions of generalized duality (Definition 8) and subtyping (Definition 12).

The rules [MAY-*] deal with late inputs and early outputs. As an example, consider the session type $S = \&\{\mathsf{a} : \oplus\{\mathsf{c} : T\}\}$ for which we may derive the transition sequences

$$S \xrightarrow{?\mathsf{a}} \oplus\{\mathsf{c} : T\} \xrightarrow{!\mathsf{c}} T \qquad \text{and} \qquad S \xrightarrow{!\mathsf{c}} \&\{\mathsf{a} : T\} \xrightarrow{?\mathsf{a}} T$$

The sequence on the left, obtained by [MUST-&] followed by [MUST-⊕], is an ordinary one: actions are performed according to the syntactic structure of the type. The sequence on the right, obtained using [MUST-⊕], [MAY-⊕] and [MUST-&], is peculiar to the asynchronous setting: it describes a situation in which the output !c may be performed earlier than the input ?a. Since communication is asynchronous and c is going to be sent anyway, a process complying with S might choose to send it *early*, before waiting for a. Note that this is just a possibility: a process strictly adhering to the transition sequence on the left would still comply with S.

A dual reasoning applies to late inputs. If we consider the type $S' = \oplus\{\mathsf{a} : \&\{\mathsf{c} : T'\}\}$ we may derive the transition sequences

$$S' \xrightarrow{!\mathsf{a}} \&\{\mathsf{c} : T'\} \xrightarrow{?\mathsf{c}} T' \qquad \text{and} \qquad S' \xrightarrow{?\mathsf{c}} \oplus\{\mathsf{a} : T'\} \xrightarrow{!\mathsf{a}} T'$$

Again, the sequence on the left is standard. The sequence on the right, obtained using [MUST-&], [MAY-&] and [MUST-⊕], says that a process complying with S' is able to consume a c message, even though this will happen only after the process has sent a. The fact that the input action is late cannot cause issues, since both the outgoing a and the incoming c are sent asynchronously with a non-blocking operation.

As it is clear looking at the rules [MAY-⊕] and [MAY-&], late inputs and early outputs concerning a branching session type must be derivable for every branch: a message may be sent early (before an input) only if it is independent of the input; a message may be received late (after an output) only if it is independent of the output. For example, if we take $S = \&\{\mathsf{a} : \oplus\{\mathsf{c} : S_1\}, \mathsf{b} : \oplus\{\mathsf{c} : S_2, \mathsf{d} : S_3\}\}$ then we can derive

$$S \xrightarrow{!\mathsf{c}} \&\{\mathsf{a} : S_1, \mathsf{b} : S_2\} \quad \text{and also} \quad S \xrightarrow{?\mathsf{b}} \oplus\{\mathsf{c} : S_2, \mathsf{d} : S_3\} \xrightarrow{!\mathsf{d}} S_3 \quad \text{but not} \quad S \xrightarrow{!\mathsf{d}} S'$$

no matter what S' could be. A process complying with S may send c early, before receiving either a or b, since the output of c is allowed regardless of the input. On the contrary, the output d is allowed only if the input is b and so it cannot be anticipated before the input. Symmetrically, if we consider the session type $T = \oplus\{\mathsf{a} : \&\{\mathsf{c} : T_1\}, \mathsf{b} : \&\{\mathsf{c} : T_2, \mathsf{d} : T_3\}\}$, we may derive

$$T \xrightarrow{?\mathsf{c}} \oplus\{\mathsf{a} : T_1, \mathsf{b} : T_2\} \quad \text{and also} \quad T \xrightarrow{!\mathsf{b}} \&\{\mathsf{c} : T_2, \mathsf{d} : T_3\} \xrightarrow{?\mathsf{d}} T_3 \quad \text{but not} \quad T \xrightarrow{?\mathsf{d}} T'$$

The (late) input transition c is enabled because the process is able to receive c regardless of the tag a or b that it sends. On the contrary, the input transition on d is enabled only if the process sends b.

A subtler case of late input is illustrated by the session type $S_1 = \oplus\{\mathsf{a} : S_1, \mathsf{b} : \&\{\mathsf{c} : S_2\}\}$, which describes the behavior of a process that sends an arbitrary number of a's or a b and then waits for a c. Since the singly-lined rules in Table 3 are interpreted coinductively, we can derive $S_1 \xrightarrow{?\mathsf{c}} S_1'$ where $S_1' = \oplus\{\mathsf{a} : S_1', \mathsf{b} : S_2\}$ by means of the following *infinite* derivation:

$$\dfrac{\dfrac{\vdots}{S_1 \xrightarrow{?\mathsf{c}} S_1'}\,[\text{MAY-}\oplus] \qquad \dfrac{}{\&\{\mathsf{c} : S_2\} \xrightarrow{?\mathsf{c}} S_2}\,[\text{MUST-}\&]}{S_1 \xrightarrow{?\mathsf{c}} S_1'}\,[\text{MAY-}\oplus] \tag{4}$$

The transition $S_1 \xrightarrow{?\mathsf{c}} S_1'$ says that a process complying with S_1 performs a late input of c. Compared to the other examples of late inputs, this one looks more questionable and for a good reason: it may be the case that a process sending an *infinite* sequence of a's complies with S_1. So, claiming that such process is eventually able to input a c may lead to a message remaining orphan. In this work, however, we *assume* that a well-typed process complying with S_1 will also be *fair*, in the sense that it will eventually stop sending a's and will send the b ensuring that the c message is consumed. The same assumption is made, in dual form, for early outputs. For example, the session type $T_1 = \&\{\mathsf{a} : T_1, \mathsf{b} : \oplus\{\mathsf{c} : T_2\}\}$ performs the early output transition $T_1 \xrightarrow{!\mathsf{c}} T_1'$ where $T_1' = \&\{\mathsf{a} : T_1', \mathsf{b} : T_2\}$. That is, a process complying with T_1 is allowed to send c early because it is guaranteed to eventually receive a b message.

The infinite derivation that we have just shown in (4) reminds us that we must be careful in the use of coinduction for defining the LTS. Indeed, an unconstrained use of coinduction would allow us to derive early/late transitions that do not correspond to any "real" action of a session type, solely using [MAY-*] rules. Consider for example the infinite session type $S = \oplus\{\mathsf{a} : S\}$, which allows sending a neverending sequence of a's. It is easy to derive $S \xrightarrow{?\sigma} S$ for every σ by an infinite derivation consisting of [MAY-$\oplus$] rules only, despite the protocol described by S does *not* enable any input transition! Of course we want to make sure that, whenever we derive a late/early transition, this is justified by the use of at least one [MUST-*] rule somewhere in the derivation. This is the reason why the LTS is defined through a GIS and not simply by the (coinductively interpreted) singly-lined rules in Table 3. On the one hand, we want to make sure that that a late/early transition is enabled along *every* branch of a session type. This is an *invariant property* enforced by the [MAY-*] rules. On the other hand, we want to make sure that there exists at least one branch along which the transition eventually originates for real. This is a *well-founded property* enforced by the [MUST-*] rules. As it has already been observed elsewhere [15], GIS are a convenient way of defining relations like $\xrightarrow{\alpha}$ that mix invariant and well-founded properties at the same time. The effect of defining the LTS as a GIS is that, whenever we build a (possibly infinite) derivation for $S \xrightarrow{\alpha} T$ using the singly-lined rules, we must also be able to find, *for each* judgment $S_i \xrightarrow{\alpha_i} T_i$ in this derivation, a *finite* derivation for $S_i \xrightarrow{\alpha_i}_{\mathsf{ind}} T_i$. In (4) this is achieved easily, as shown below:

$$\dfrac{\dfrac{}{\&\{\mathsf{c} : S_2\} \xrightarrow{?\mathsf{c}}_{\mathsf{ind}} S_2}\,[\text{MUST-}\&]}{S_1 \xrightarrow{?\mathsf{c}}_{\mathsf{ind}} S_1'}\,[\text{FAIR-}\oplus]$$

Notice the key role of [FAIR-$\oplus$] in building this *finite* derivation for $S_1 \xrightarrow{?\mathsf{c}}_{\mathsf{ind}} S_1'$. Since S_1 is an *infinite* session type, we would not be able to find a finite derivation for $S_1 \xrightarrow{?\mathsf{c}}_{\mathsf{ind}} S_1'$ if we insisted on using [MAY-$\oplus$] only, since this rule requires us to derive the late input transition in *every* branch of the session type. Instead, according to [FAIR-$\oplus$] it suffices to find *one* branch along which we can eventually derive the late input transition directly. In the literature on GIS the doubly-lined rules are called *corules*. Here, we have called them [FAIR-$\oplus$] and [FAIR-&] because they somehow capture the fairness assumption of the LTS: whenever we derive late/early transitions in a looping session type like S_1 and T_1, the fairness assumption makes sure that the conversation eventually follows a branch leading out of the loop.

There is one exception to what we have just said about transitions being eventually derived by [MUST-*] rules. Recalling that $\mathbf{0} = \oplus\{\}$ and $\top = \&\{\}$, by [MAY-$\oplus$] and [MAY-&] we can easily derive $\mathbf{0} \xrightarrow{?\sigma} \mathbf{0}$ and $\top \xrightarrow{!\sigma} \top$ for every σ. These derivations are trivially valid for the GIS since they are finite. We might be tempted to flag these cases as pathological. After

all, the protocol $\mathbf{0}$ describes an unrealistic process that is able to *input anything* and the protocol $\top$ describes a uncontrollable process that may *output anything*. While it is true that such behaviors are practically useless, we will see in Section 4 that the derivability of these transitions for $\mathbf{0}$ and $\top$ has an important impact in the resulting subtyping relation, for which $\mathbf{0}$ and $\top$ will play the role of least and the greatest element.

Hereafter, we let φ and ψ range over finite sequences of labels, we write $\xrightarrow{\alpha_1 \cdots \alpha_n}$ for the composition $\xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n}$, we write $S \xrightarrow{\alpha}$ if $S \xrightarrow{\alpha} T$ for some T and $S \not\xrightarrow{\alpha}$ if not $S \xrightarrow{\alpha}$.

3.3 Properties of Asynchronous Session Types

To substantiate the claim that each transition derivable by the GIS is "real" (*i.e.* it originates from the syntax of the session type), we prove that every input/output transition can be derived by the application of an axiom in Table 3 after every maximal, strongly fair sequence of immediate outputs/inputs. Strong fairness is a weaker assumption implied by the full fairness of Definition 2 [44]. Formally, a (possibly infinite) sequence of transitions $S_0 \xrightarrow{\alpha_1} S_1 \xrightarrow{\alpha_2} \cdots$ is *strongly fair* [26, 44] if, whenever some S' occurs infinitely often in the sequence $S_0 S_1 \cdots$ and $S' \xrightarrow{\alpha} T'$, then also T' occurs infinitely often in the same sequence. Intuitively, a strongly fair sequence of transitions does not discriminate those transitions that are enabled infinitely often. For example, if $S = \oplus\{\mathsf{a} : S, \mathsf{b} : T\}$, the infinite sequence $S \xrightarrow{!\mathsf{a}} S \xrightarrow{!\mathsf{a}} \cdots$ of transitions is strongly unfair, because the transition $S \xrightarrow{!\mathsf{b}} T$ is infinitely often enabled but never performed. On the contrary, if $S = \oplus\{\mathsf{a} : S\}$, then the infinite sequence $S \xrightarrow{!\mathsf{a}} S \xrightarrow{!\mathsf{a}} \cdots$ of transitions is strongly fair.

▶ **Theorem 6.** *Let α be the label of an input/output transition. Then $S \xrightarrow{\alpha}$ if and only if every maximal, strongly fair sequence of immediate output/input transitions $S \xrightarrow{\alpha_1} S_1 \xrightarrow{\alpha_2} \cdots$ is finite and ends in some T such that $T \xrightarrow{\alpha}$ is derivable by an axiom in Table 3.*

Note that, in the statement of Theorem 6, we cannot require the transition $T \xrightarrow{\alpha}$ to be immediate because of the phony early/late transitions enabled by $\mathbf{0}$ and $\top$.

We are accustomed to think that session types are used to describe race-free interactions, but we have seen examples of session types that simultaneously enable both input and output transitions. When this happens, such transitions are independent and do not interfere with each other. We formalize this fact by establishing a diamond property for session types.

▶ **Proposition 7.** *If $S \xrightarrow{?\sigma} S'$ and $S \xrightarrow{!\tau} S''$, then $S' \xrightarrow{!\tau} T$ and $S'' \xrightarrow{?\sigma} T$ for some T.*

4 Fair Asynchronous Subtyping

We define the subtyping relation for asynchronous session types in three steps. First of all, we formalize what we mean by *correct asynchronous composition* between the session types describing the protocols implemented by two processes using the two endpoints of a session. In many session type systems, this notion coincides with session type duality. Since we have to take asynchrony into account, duality alone is too strict so we need a notion of correct composition using the LTS defined in Section 3. Once this notion is in place, asynchronous subtyping can be defined as the relation that preserves correct asynchronous composition. This relation is "sound" by definition, but its properties can be intrinsically difficult to grasp. The final step will be to provide a *precise* (*i.e.* sound and complete) alternative characterization of asynchronous subtyping that sheds light on its properties.

The definition of correct asynchronous composition is given below.

▶ **Definition 8.** *We say that* $\mathcal{R}$ *is a* correct asynchronous composition *if* $(S,T) \in \mathcal{R}$ *implies:*
1. *either* $\mathsf{pos}(S)$ *or* $\mathsf{pos}(T)$*;*
2. *if* $S \xrightarrow{!\sigma} S'$ *and* $\sigma \in \{*\} \cup \mathsf{Tags}$*, then* $T \xrightarrow{?\sigma} T'$ *and* $(S',T') \in \mathcal{R}$*;*
3. *if* $T \xrightarrow{!\sigma} T'$ *and* $\sigma \in \{*\} \cup \mathsf{Tags}$*, then* $S \xrightarrow{?\sigma} S'$ *and* $(S',T') \in \mathcal{R}$*;*
4. *if* $S \xrightarrow{!S_1} S_2$*, then* $T \xrightarrow{?T_1} T_2$ *and* $(S_1,T_1) \in \mathcal{R}$ *and* $(S_2,T_2) \in \mathcal{R}$*;*
5. *if* $T \xrightarrow{!T_1} T_2$*, then* $S \xrightarrow{?S_1} S_2$ *and* $(S_1,T_1) \in \mathcal{R}$ *and* $(S_2,T_2) \in \mathcal{R}$*.*

We write $\bowtie$ *for the largest correct asynchronous composition.*

In words, Items 2–5 state that (S,T) forms a correct composition if, whenever one of the two types performs a (possibly early) output transition $!\sigma$, the other type is able to respond with a (possibly late) compatible input transition $?\tau$ and the continuations remain correct. In general this is not enough to guarantee progress because early outputs are only allowed but not mandatory. For example, the types $\&\{\mathsf{a} : \oplus\{\mathsf{b} : \mathbf{1}\}\}$ and $\&\{\mathsf{b} : \oplus\{\mathsf{a} : \bot\}\}$ satisfy Items 2 and 3, but two processes strictly adhering to these protocols (*i.e.* without performing early outputs) would starve. Item 1 requires that at least one of the two types is positive, namely that at least on one side of the session the outputs are guaranteed to be immediate. This is enough to ensure progress. For example, we have $\&\{\mathsf{a} : \oplus\{\mathsf{b} : \mathbf{1}\}\} \bowtie \oplus\{\mathsf{a} : \&\{\mathsf{b} : \bot\}\}$ as well as $\oplus\{\mathsf{b} : \&\{\mathsf{a} : \mathbf{1}\}\} \bowtie \oplus\{\mathsf{a} : \&\{\mathsf{b} : \bot\}\}$.

It is easy to see that $\bowtie$ is symmetric and that duality implies correctness:

▶ **Proposition 9.** *$S^\bot \bowtie S$ holds for every session type S.*

Other properties of $\bowtie$ are more surprising. For example, if $S = \&\{\mathsf{a} : S, \mathsf{b} : \oplus\{\mathsf{c} : \mathbf{1}\}\}$ and $R = \oplus\{\mathsf{a} : R\}$, we have that $S \bowtie R$ does *not* hold because of the early output transition $S \xrightarrow{!\mathsf{c}}$ to which R is unable to respond. The lack of compatibility between S and R is due to the fact that (the process behaving as) S makes a *fairness assumption* on the behavior of the process it is interacting with. More precisely, a process complying with S assumes that sooner or later a b message will be received, finally enabling the output of c. In anticipation of this, the process may decide to perform an early output of c, but in doing so it would generate an orphan message when interacting with another process adhering to R, which is not honoring this fairness assumption. It is also easy to see that $\mathbf{0} \bowtie S$ holds for every S and that $\top \bowtie S$ implies $S = \mathbf{0}$. These properties of $\mathbf{0}$ and $\top$ follow directly from the [MAY-$\oplus$] and [MAY-&] rules that we have commented in Section 3.

▶ **Example 10.** Let $!\mathsf{a}.S$ stand for $\oplus\{\mathsf{a} : S\}$ and $(!\mathsf{a})^n.S$ stand for $!\mathsf{a} \ldots !\mathsf{a}.S$ with n $!\mathsf{a}$ prefixes. Consider the session types $S = \oplus\{\mathsf{task} : S, \mathsf{stop} : T\}$ and $T = \&\{\mathsf{res} : T, \mathsf{stop} : \bot\}$ and $U = \&\{\mathsf{task} : \oplus\{\mathsf{res} : U\}, \mathsf{stop} : \oplus\{\mathsf{stop} : \mathbf{1}\}\}$ which respectively describe the behaviors of *Split*, *Gather* and *Worker* of Section 1 on the channel y. It is easy to establish that

$$\mathcal{S} \stackrel{\text{def}}{=} \{(S, (!\mathsf{res})^n.U) \mid n \in \mathbb{N}\} \cup \{(T, (!\mathsf{res})^n.!\mathsf{stop}.\mathbf{1}) \mid n \in \mathbb{N}\} \cup \{(\bot, \mathbf{1})\}$$

is a correct asynchronous composition hence $S \bowtie U$. ⌟

We can now define fair asynchronous subtyping semantically using *Liskov's substitution principle* [38] where the property being preserved is session correctness (Definition 8).

▶ **Definition 11** (fair asynchronous subtyping)**.** *We say that* S *is a* fair asynchronous subtype *(or just* subtype*) of* T*, notation* $S \leqslant T$*, if* $R \bowtie T$ *implies* $R \bowtie S$ *for every* R*.*

Paraphrasing, this definition says that a process using a channel x according to T can be safely replaced by a process using x according to S when S is a subtype of T. Indeed, the peer process, which is assumed to use the same channel x according to some session type R such that $R \bowtie T$, will still interact correctly after the substitution has taken place.

A few subtyping relations are easy to figure out. For example, we have $\mathbf{0} \leqslant S$ and $S \leqslant \top$ for every S because of the properties of $\mathbf{0}$ and $\top$ that we have pointed out above. It is also easy to see that $\oplus\{\mathsf{a} : \&\{\mathsf{b} : S\}\} \leqslant \&\{\mathsf{b} : \oplus\{\mathsf{a} : S\}\}$ holds in an asynchronous setting. After all, the process that behaves according to $\oplus\{\mathsf{a} : \&\{\mathsf{b} : S\}\}$ is sending a tag a that also the process that behaves according to $\&\{\mathsf{b} : \oplus\{\mathsf{a} : S\}\}$ *may* anticipate. Note that the inverse relation $\&\{\mathsf{b} : \oplus\{\mathsf{a} : S\}\} \leqslant \oplus\{\mathsf{a} : \&\{\mathsf{b} : S\}\}$ does not hold, despite the fact that $\oplus\{\mathsf{a} : \&\{\mathsf{b} : S\}\}$ and $\&\{\mathsf{b} : \oplus\{\mathsf{a} : S\}\}$ perform exactly the same transitions, because $\&\{\mathsf{a} : \oplus\{\mathsf{b} : S^{\perp}\}\}$ forms a correct asynchronous composition with $\oplus\{\mathsf{a} : \&\{\mathsf{b} : S\}\}$ but not with $\&\{\mathsf{b} : \oplus\{\mathsf{a} : S\}\}$ (Item 1 of Definition 8 is violated).

The above notion of subtyping is sound "by definition", but provides little information concerning the shape of related session types. To compensate for this problem, we also give a sound and complete coinductive characterization of $\leqslant$, which relates directly to Definition 8.

▶ **Definition 12** (coinductive asynchronous subtyping)**.** *We say that $\mathcal{S}$ is a* coinductive asynchronous subtyping *if $(S,T) \in \mathcal{S}$ implies:*

1. *either* $\mathsf{pos}(S)$ *or* $\mathsf{neg}(T)$*;*
2. *if $T \xrightarrow{?\sigma} T'$ and $\sigma \in \{*\} \cup \mathsf{Tags}$, then $S \xrightarrow{?\sigma} S'$ and $(S',T') \in \mathcal{S}$;*
3. *if $S \xrightarrow{!\sigma} S'$ and $\sigma \in \{*\} \cup \mathsf{Tags}$, then $T \xrightarrow{!\sigma} T'$ and $(S',T') \in \mathcal{S}$;*
4. *if $T \xrightarrow{?T_1} T_2$, then $S \xrightarrow{?S_1} S_2$ and $(S_1,T_1) \in \mathcal{S}$ and $(S_2,T_2) \in \mathcal{S}$;*
5. *if $S \xrightarrow{!S_1} S_2$, then $T \xrightarrow{!T_1} T_2$ and $(S_1,T_1) \in \mathcal{S}$ and $(S_2,T_2) \in \mathcal{S}$.*

Items 2–5 of Definition 12 specify the expected requirements for a session subtyping relation: every input transition of the supertype T must be matched by an input transition of the subtype S and the corresponding continuations should still be related by subtyping; dually, every output transition of the subtype S must be matched by an output transition of the supertype T and the corresponding continuations should still be related by subtyping. Interestingly, these items are essentially the same found in analogous characterizations of *synchronous subtyping for session types* [28], modulo the different orientation of $\leqslant$ due to our viewpoint based on the substitution of processes rather than on the substitution of channels.[1] However, the clauses of Definition 12 are not mutually exclusive, because the same session type may perform both input and output transitions. Also, session types related by subtyping need not start with the same type constructor. In this respect, Item 1 makes sure that the smaller session type can only anticipate (and not postpone) outputs, as argued above.

Definition 12 is a sound and complete characterization of $\leqslant$.

▶ **Theorem 13.** *$\leqslant$ is the largest coinductive asynchronous subtyping.*

Using Definition 12 and Theorem 13 we can prove some significant properties of $\leqslant$:

- (input contravariance) $\&\{\mathsf{a}_i : S_i\}_{i\in I} \leqslant \&\{\mathsf{a}_i : S_i\}_{i\in J}$ if $J \subseteq I$ by Item 2;
- (output covariance) $\oplus\{\mathsf{a}_i : S_i\}_{i\in I} \leqslant \oplus\{\mathsf{a}_i : S_i\}_{i\in J}$ if $I \subseteq J$ by Item 3;
- (output anticipation) $\oplus\{\mathsf{a}_j : \&\{\mathsf{b}_i : S_{ij}\}_{i\in I}\}_{j\in J} \leqslant \&\{\mathsf{b}_i : \oplus\{\mathsf{a}_j : S_{ij}\}_{j\in J}\}_{i\in I}$. Note that the inverse relation does not hold, despite these two session types have exactly the same transitions, because of Item 1.

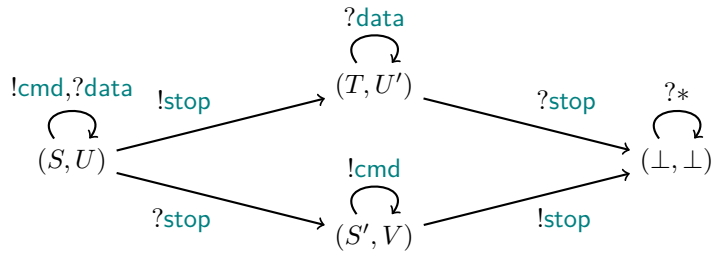
There are exceptions to output covariance and input contravariance when one of the two types specifies a non-terminating protocol. The next example illustrates one of such cases.

[1] The interested reader may refer to Gay [27] for a comparison of the two viewpoints.

▶ **Example 14.** Consider $S = \&\{\mathsf{a} : S, \mathsf{b} : \oplus\{\mathsf{c} : \mathbf{1}\}\}$ and $T = \&\{\mathsf{a} : T\}$. Despite S is a subtype of T for other session subtypings [28, 39, 11, 8], we have $S \not\leqslant T$ because $S \xrightarrow{!\mathsf{c}}$ and $T \not\xrightarrow{!\mathsf{c}}$. To see the reason why admitting this relation could cause a problem, consider a process that complies with the protocol $R = \oplus\{\mathsf{a} : R\}$. Such a process would output infinitely many a's and would not expect to input anything. Still, if we consider $S' = \&\{\mathsf{a} : S', \mathsf{b} : \mathbf{1}\}$ we have that $\oplus\{\mathsf{c} : S'\} \leqslant S$ holds. That is, a process complying with $\oplus\{\mathsf{c} : S'\}$ performs immediately the early output in S. By transitivity of $\leqslant$ (whose validity is implied by Theorem 13), we would also have $\oplus\{\mathsf{c} : S'\} \leqslant T$. Now the problem is clear: $R \bowtie T$ holds, but a process complying with R would not be able to handle the incoming c message if we composed it with a process complying with $\oplus\{\mathsf{c} : S'\}$. ⌟

The cases in which co/contra variance does not hold have no impact on the typeability of CaP processes: since our type system ensures the fair termination of well-typed processes, protocols like T in Example 14 are not inhabited. We will see in Section 7 that $\leqslant$ includes other subtyping relations supporting full co/contra variance for the family of fairly terminating session types, those that always allow the protocol to end.

▶ **Example 15.** We borrow a scenario from Bravetti, Lange and Zavattaro [8] to showcase an interesting example of fair asynchronous subtyping. Imagine a system made of a ground station and a satellite such that, at each flyby, the satellite sends data from the previous orbit and receives commands to execute in the next one. In principle, the ground station should follow the protocol $U = \&\{\mathsf{data} : U, \mathsf{stop} : V\}$ where $V = \oplus\{\mathsf{cmd} : V, \mathsf{stop} : \bot\}$. However, since the flyby window may be short, it makes sense to implement the ground station so that it communicates with the satellite in full duplex by anticipating the output of the commands. In this case, the ground station follows the protocol $S = \oplus\{\mathsf{cmd} : S, \mathsf{stop} : T\}$ where $T = \&\{\mathsf{data} : T, \mathsf{stop} : \bot\}$. Is this implementation correct? We can answer in the affirmative by proving $S \leqslant U$ with the diagram below, which represents a coinductive asynchronous subtyping containing the pair (S, U). In the diagram we also use the types $S' = \oplus\{\mathsf{task} : S', \mathsf{stop} : \bot\}$ and $U' = \&\{\mathsf{res} : U', \mathsf{stop} : \bot\}$.



Note that S allows for the anticipation of an unbounded number of outputs before an unbounded number of inputs. ⌟

We now list a few properties of $\leqslant$. First of all, we establish that $\leqslant$ is closed by duality.

▶ **Proposition 16.** *If $S \leqslant T$, then $T^\perp \leqslant S^\perp$.*

Proof. By Theorem 13 it suffices to show that if $\mathcal{S}$ is a coinductive asynchronous subtyping, then so is $\mathcal{S}^\perp \stackrel{\text{def}}{=} \{(T^\perp, S^\perp) \mid (S, T) \in \mathcal{S}\}$. This follows immediately from Definition 12. ◀

Then, we show how to characterize $\bowtie$ solely in terms of $\leqslant$.

▶ **Theorem 17.** *$S \bowtie T$ if and only if $S \leqslant T^\perp$*

Finally, we observe that similarly to the other asynchronous session type theories correct composition is undecidable. The source of undecidability follows from the possibility to use communicating buffers to model unbounded memories like tapes of Turing Machines [36] or queues of Queue Machines [6].

▶ **Theorem 18.** *Given two types S and T, the problem of checking $S \bowtie T$ is undecidable.*

As a direct consequence of Theorems 17 and 18, we have that checking $S \leqslant T$ is also an undecidable problem. Despite these negative results, the coinductive characterizations of correctness (Definition 8) and subtyping (Definition 12) are useful to prove that these relations hold in specific cases, as done in Examples 10 and 15.

5 Type System

In this section we describe the type system for CaP ensuring that well-typed processes weakly terminate, besides being free from communication errors and deadlocks. Theorem 4 then allows us to conclude that the very same type system also ensures fair termination when we make the fairness assumption stated in Definition 2. The type system is based on the proof rules of MALL but also includes typing rules for the non-logical process forms, namely termination, process invocation and non-deterministic choice.

5.1 Measuring Processes and Type Annotations

It is a known fact that infinitary proof systems for linear logic (*e.g.* μMALL [3, 25]) require validity conditions to enjoy the cut elimination property. As a consequence, the infinitary type system for CaP requires validity conditions to ensure the (weak) termination property. We implement these validity conditions using the technique of Dagnino and Padovani [19] and decorate types and typing judgments with quantitative information that estimates the amount of effort required to terminate a process. It is natural to measure such effort in terms of *number of reductions* of that process (Table 2). Since most reductions arise from session interactions and each session interaction involves a process outputting a message on a channel (with a *positive* type) and a process inputting a message from the same channel (with a *negative* type) we count the number of process forms representing outputs to establish the number of reductions that are necessary to normalize a process. In the presence of branching processes (those whose behavior depends on a tag received from a channel), we can tentatively compute an upper bound for terminating *each* branch.

Let us use the definitions in Section 1 to walk through this approach on increasingly complex examples. To start, consider the process *Gather*, which we repeat here for convenience:

$$Gather(x, y) = y \triangleright \{\mathsf{res} : Gather\langle x, y\rangle, \mathsf{stop} : \mathsf{wait}\ y.x \triangleleft \mathsf{resp.close}\ x\}$$

Now suppose that n is the measure associated with *Gather* and note that *Gather* is a branching and recursive process. The res branch does not contain any output actions, whereas the stop branch performs two outputs on x. Therefore, n should satify the relations $n \geq n$ (for the res branch) and $n \geq 2$ (for the stop branch) whose least solution is $n = 2$. That is, *Gather* weakly terminates by performing at most two outputs. Note that the actual number of interactions performed by *Gather* depends on the number of received res messages and may be larger than 2. However, these received messages are accounted for in the measure of the sender, while the measure 2 we give to *Gather* only accounts for the messages sent by *Gather*.

If we now consider the process

$$Split(x,y) = y \lhd \mathsf{task}.Split\langle x,y\rangle \oplus y \lhd \mathsf{stop}.Gather\langle x,y\rangle$$

we have to measure a non-deterministic choice. Since non-deterministic choices are performed autonomously by a process and we are interested in proving that processes *weakly* terminate, we can measure a non-deterministic choice by considering the branch with the least measure and the additional reduction due to [R-CHOICE]. So, in this case, the measure of *Split* – say m – must satisfy the equation $m \geq 1 + \min\{1+m, 3\}$ where $1+m$ is the measure of the left branch and 3 is the (least) measure of the right branch, having already established that *Gather* can be given measure 2. The least least solution of this constraint is $m = 4$.

A more challenging process to measure is

$$Worker(y) = y \rhd \{\mathsf{task} : y \lhd \mathsf{res}.Worker\langle y\rangle, \mathsf{stop} : y \lhd \mathsf{stop}.\mathsf{close}\ y\}$$

in which each task input is immediately followed by a res output. If we try to measure *Worker* following the same reasoning applied to *Gather*, we end up looking for some n satisfying the system of inequations $n \geq 1+n$ (for the task branch) and $n \geq 2$ (for the stop branch). Since no n satisfies these inequations, we have to refine our measuring strategy or else *Worker* would end up being ill typed. This is precisely what happens in some type systems ensuring the fair termination of sessions [14, 12] which is unfortunate because *Worker* is not ill behaved after all: the fairness assumption we are making ensures that the number of tasks will be finite, albeit unbounded. Likewise, the number of results will be finite but unbounded as well. The problem is that we are unable to find a *single* measure for *Worker* that accounts for all possibilities.

We refine our measuring technique allowing the measure of *Worker* to *depend* on the tags it receives. We realize this dependency by annotating tags with measures that are charged to senders and discharged from receivers. To see this mechanism at work in the case of *Split* and *Worker*, consider the following annotated session types:

$$S = \oplus\{\mathsf{task}^1 : S, \mathsf{stop}^0 : T\} \tag{5}$$

$$T = \&\{\mathsf{res}^0 : T, \mathsf{stop}^0 : \bot\} \tag{6}$$

$$U = \&\{\mathsf{task}^1 : \oplus\{\mathsf{res}^0 : U\}, \mathsf{stop}^0 : \oplus\{\mathsf{stop}^0 : \mathbf{1}\}\} \tag{7}$$

The annotations in task^1 and stop^0 mean that a process like *Split*, which complies with S, is *charged* by 1 unit of measure whenever it sends task and by 0 when it sends stop. This measure is charged *in addition* to the cost of the output it is performing and accounts for the cost of sending the result in *Worker*. Since this cost is already charged on *Split*, it can be *discharged* from the task branch of *Worker* when we compute its measure, as shown by U. This way we end up solving for *Worker* the equations $n \geq n$ (for the task branch) and $n \geq 2$ (for the stop branch) whose least solution is $n = 2$. In principle we have to reconsider the measure we gave to *Split* to account for the 1 unit that is now charged for each output of task, but since the measure of *Split* was determined by the stop branch (which has a null annotation) it remains unchanged.

From now on we consider a refinement of the session types presented in Section 3 where tags are annotated with natural numbers. These annotations do not interfere in any way with the properties and characterizations of session types we have presented there and in Section 4, with the proviso that wherever we have written matching message types in Sections 3 and 4 we mean that measures also match. In particular, Definition 8 entails that the amount of measure charged on one side of the session matches the amount of measure discharged from

Table 4 Typing rules for CaP.

$$[\textsc{done}]\ \frac{}{\mathsf{done} \vdash^n \emptyset} \qquad [\textsc{call}]\ \frac{P \vdash^n \overline{x : S}}{A\langle \overline{x} \rangle \vdash^{m+n} \overline{x : S}}\ A(\overline{x}) = P \qquad [\mathbf{1}]\ \frac{}{\mathsf{close}\ x \vdash^{1+n} x : \mathbf{1}} \qquad [\bot]\ \frac{P \vdash^n \Gamma}{\mathsf{wait}\ x.P \vdash^n \Gamma, x : \bot}$$

$$[\textsc{choice}]\ \frac{P \vdash^{n_1} \Gamma \qquad Q \vdash^{n_2} \Gamma}{P \oplus Q \vdash^{1+n_k} \Gamma} \qquad [\otimes]\ \frac{P \vdash^m \Gamma, y : S \qquad Q \vdash^n \Delta, x : T}{x(y)[P].Q \vdash^{1+m+n} \Gamma, \Delta, x : S \otimes T} \qquad [\parr]\ \frac{P \vdash^n \Gamma, y : S, x : T}{x(y).P \vdash^n \Gamma, x : S \parr T}$$

$$[\oplus]\ \frac{P \vdash^n \Gamma, x : S_k}{x \lhd \mathsf{a}_k.P \vdash^{1+n+m_k} \Gamma, x : \oplus\{\mathsf{a}_i^{m_i} : S_i\}_{i \in I}} \qquad [\&]\ \frac{\forall i \in I : P_i \vdash^{n+m_i} \Gamma, x : S_i}{x \rhd \{\mathsf{a}_i : P_i\}_{i \in I \cup J} \vdash^n \Gamma, x : \&\{\mathsf{a}_i^{m_i} : S_i\}_{i \in I}}$$

$$[\textsc{link}]\ \frac{}{x \leftrightarrow y \vdash^{1+n} x : S, y : T}\ S^\bot \leqslant T \qquad [\textsc{cut}]\ \frac{P \vdash^m \Gamma, x : S \qquad Q \vdash^n \Delta, x : T}{(x)(P \mid Q) \vdash^{m+n} \Gamma, \Delta}\ S \bowtie T$$

the other side of the session (like for S and U above) and the clauses of Definition 12 entail that matching actions in session types related by subtyping carry the same measure. In principle, it would be possible to relax these constraints and allow some variance of measure annotations. Since we do not have concrete examples that take advantage of this further refinement, we spare the additional complexity.

5.2 Typing Rules

The typing rules for CaP are shown in Table 4. Judgments have the form $P \vdash^n \Gamma$ where P is the process being typed, n is its measure and Γ is a *typing context*, namely a partial function that maps channels to types. We let Γ and Δ range over typing contexts, we write $\emptyset$ for the empty context, $x : S$ for the singleton context that maps x to S, and Γ, Δ for the union of Γ and Δ when they have disjoint domains. The typing rules shown in Table 4 are meant to be interpreted coinductively, hence a process is well typed provided there is a (possibly infinite) derivation built using those rules. The structure of the rules is essentially the same of other session type systems based on (classical) linear logic [45, 37, 19], so we will mainly focus on those aspects of the rules that are new or different in our setting.

The rule [done] states that the terminated process can have any measure and is well typed only in the empty context.

The rule [call] states that a process invocation is well typed in a context $\overline{x : S}$ if so is its definition in the same context. Recall that a typing derivation can be infinite in the case of recursive processes and note that the measure of the invocation may be larger than that of its definition, allowing some measure to be discarded from one invocation to the next. This is not strictly necessary for the soundness of type system, but it is sometimes convenient to obtain simpler typing derivations.

The rules [$\mathbf{1}$] and [$\bot$] deal with session termination in the expected way. The measure of close x is strictly positive whereas the measure of wait $x.P$ coincides with that of P since the measure n in a typing judgment $\Gamma \vdash^n P$ only accounts for the outputs performed by P.

The rules [⊗] and [⅋] deal with the communication of channels. The measure of a channel output $x(y)[P].Q$ accounts for both the measure of P (the process being spawned that uses one end of the fresh session y) and the measure of Q, while the additional unit of measure accounts for the output being performed. The measure of a channel input $x(y).P$ simply coincides with that of the continuation process.

The rules [⊕] and [&] deal with the communication of tags. The measure of a tag output $x \lhd \mathsf{a}.P$ accounts for the measure of the continuation P and of the measure annotation m associated with a in the type of x, while the additional unit of measure accounts for the output being performed. The measure of a tag input $x \rhd \{\mathsf{a}_i : P_i\}_{i \in I}$ is the residual measure of each branch after the measure annotation m_i associated with a_i in the type of x has been discharged. Note that a tag input may in general provide more branches than those actually occurring in the type of x. As a special case, [&] also captures the introduction rule for $\top$ in MALL. In particular, we have $x \rhd \{\} \vdash^n \Gamma, x : \top$ for every n and every Γ.

The rule [CHOICE] deals with non-deterministic choices in the expected way. The rule does not specify which branch of a non-deterministic choice determines the measure of the whole choice. Since the typeability of a process rests on its measurability, the obvious strategy is to pick the continuation with the smallest measure (*cf.* Section 5.1).

The rule [LINK] is a generalization of the axiom in MALL. It allows the unification of x and y provided that $S^\perp \leqslant T$ (or equivalently $T^\perp \leqslant S$, by Proposition 16) where S is the type of x and T is the type of y. As we will see in Section 6, this formulation of the link typing rule allows us to capture some recurring usage patterns of subtyping. A link has a strictly positive measure since it accounts for one [R-LINK] reduction.

Finally, the rule [CUT] rule ensures that the processes P and Q connected by the new session x use the channel correctly requiring $S \bowtie T$ to hold where S is the type of x as used by P and T is the type of x as used by Q. Note that this correctness condition can be equivalently expressed in terms of subtyping with the relation $S \leqslant T^\perp$ by Theorem 17. The measure of the composition is the combination of the measures of P and Q.

▶ **Example 19.** Let us build a typing derivation for the server $x \rhd \{\mathsf{req} : (y)(\mathit{Split}\langle x, y\rangle \mid \mathit{Worker}\langle y\rangle)\}$ using the types S, T and U defined in Equations (5)–(7). It is convenient to also use the type $V = \oplus\{\mathsf{resp}^0 : \mathbf{1}\}$. Table 5 shows the typing derivations for the *Split*, *Gather* and *Worker* processes. Note that the measures obtained for these processes coincide with those previously inferred in Section 5.1. From $S \bowtie U$ (Example 10) we derive

$$\dfrac{\dfrac{\dfrac{\vdots}{\mathit{Split}\langle x, y\rangle \vdash^4 x : V, y : S} \quad \dfrac{\vdots}{\mathit{Worker}\langle y\rangle \vdash^2 y : U}}{(y)(\mathit{Split}\langle x, y\rangle \mid \mathit{Worker}\langle y\rangle) \vdash^6 x : V}\,[\text{CUT}]}{x \rhd \{\mathsf{req} : (y)(\mathit{Split}\langle x, y\rangle \mid \mathit{Worker}\langle y\rangle)\} \vdash^6 x : \&\{\mathsf{req}^0 : V\}}\,[\&]$$

confirming that the server is well typed. ⌟

5.3 Properties of Well-Typed Processes

We conclude this section presenting a series of results showing the main properties of well-typed processes, starting from the preservation of typing under reductions (*i.e.* subject reduction) which underlies most of the subsequent results. Subject reduction is formulated in a slightly non-standard way because the typing context may become "smaller" – *i.e.* "more precise" – according to the subtyping relation $\leqslant$ after each reduction. This is a consequence of the formulation of the rule [LINK], where the types of x and y need not be dual to each other, but can be related by subtyping. In the statement of Theorem 20, we write $\leqslant$ for the pointwise extension of subtyping to typing contexts.

Table 5 Typing derivations for *Split*, *Gather* and *Worker*.

$$\dfrac{\dfrac{\vdots}{Gather\langle x,y\rangle \vdash^2 x : V, y : T} \quad \dfrac{\dfrac{\dfrac{}{\mathsf{close}\ x \vdash^1 x : \mathbf{1}}\,[\mathbf{1}]}{x \lhd \mathsf{resp}.\mathsf{close}\ x \vdash^2 x : V}\,[\oplus]}{\mathsf{wait}\ y.x \lhd \mathsf{resp}.\mathsf{close}\ x \vdash^2 x : V, y : \bot}\,[\bot]}{y \rhd \{\mathsf{res} : Gather\langle x,y\rangle, \mathsf{stop} : \mathsf{wait}\ y.x \lhd \mathsf{resp}.\mathsf{close}\ x\} \vdash^2 x : V, y : T}\,[\&]}{Gather\langle x,y\rangle \vdash^2 x : V, y : T}\,[\textsc{call}]$$

$$\dfrac{\dfrac{\dfrac{\vdots}{Split\langle x,y\rangle \vdash^4 x : V, y : S}}{y \lhd \mathsf{task}.Split\langle x,y\rangle \vdash^6 x : V, y : S}\,[\oplus] \quad \dfrac{\dfrac{\vdots}{Gather\langle x,y\rangle \vdash^2 x : V, y : T}\,[\textsc{link}]}{y \lhd \mathsf{stop}.Gather\langle x,y\rangle \vdash^3 x : V, y : S}\,[\oplus]}{y \lhd \mathsf{task}.Split\langle x,y\rangle \oplus y \lhd \mathsf{stop}.Gather\langle x,y\rangle \vdash^4 x : V, y : S}\,[\textsc{choice}]}{Split\langle x,y\rangle \vdash^4 x : V, y : S}\,[\textsc{call}]$$

$$\dfrac{\dfrac{\dfrac{\vdots}{Worker\langle y\rangle \vdash^2 y : U}}{y \lhd \mathsf{resp}.Worker\langle y\rangle \vdash^3 y : \oplus\{\mathsf{resp}^0 : U\}}\,[\oplus] \quad \dfrac{\dfrac{}{\mathsf{close}\ y \vdash^1 y : \mathbf{1}}\,[\mathbf{1}]}{y \lhd \mathsf{stop}.\mathsf{close}\ y \vdash^2 y : \oplus\{\mathsf{stop}^0 : \mathbf{1}\}}\,[\oplus]}{y \rhd \{\mathsf{task} : y \lhd \mathsf{resp}.Worker\langle y\rangle, \mathsf{stop} : y \lhd \mathsf{stop}.\mathsf{close}\ y\} \vdash^2 y : U}\,[\&]}{Worker\langle y\rangle \vdash^2 y : U}\,[\textsc{call}]$$

▶ **Theorem 20** (subject reduction). *If $P \to Q$, then $P \vdash^n \Gamma$ implies $Q \vdash^m \Delta$ for some $\Delta \leqslant \Gamma$.*

In general it is not possible to establish an order relationship between m (the measure of the process after the reduction) and n (the measure of the process before the reduction) because of the rule [CHOICE]. However, when $P \to$ it is always possible to reduce P so that its measure strictly decreases. This is the key reasoning for the forthcoming termination results.

While deadlock freedom (Definition 1) is usually taken for granted for session type systems based on linear logic, where it is a straightforward consequence of cut elimination, its proof for CaP is more elaborated because of the deep cut reductions that model asynchrony. In particular, all of the reduction rules of Table 2 except [R-CLOSE] take into account the presence of message buffers. The commuting conversions dealing with message buffers play a key role in rearranging the structure of processes so as to guarantee deadlock freedom in the presence of asynchronous communications.

▶ **Theorem 21** (deadlock freedom). *If $P \vdash^n \emptyset$, then P is deadlock free.*

Theorem 21 holds for closed processes (those well typed in the empty context), but its proof is based on a more general *productivity* result [43] that applies to open processes as well. Intuitively, a productive process is always able to reduce or it is structurally precongruent to another process that exposes some actions on a free name.

Next we have two termination results. The first one ensures that well-typed processes are weakly terminating, namely that they have (at least) one finite run.

▶ **Theorem 22** (weak termination). *If $P \vdash^n \Gamma$, then P is weakly terminating.*

Weak termination is a liveness property and, as such, its proof is based on a well-founded argument. The quantity n is one of the components of a lexicographically ordered pair that serves as basis for the induction. The other component of the pair measures the depth of the

"unguarded" top-level structure of the process which is guaranteed to be finite because of the guardedness restriction we have assumed on processes in Section 2. This component is needed to cope with the typing rules [CALL] and [CUT] for which the measure of the premise(s) is not necessarily smaller than the measure in the conclusion of the rules.

Theorem 22 can be strengthened to a termination result for *deterministic* processes, those that do not contain non-deterministic choices.

▶ **Theorem 23** (termination). *If P is deterministic and $P \vdash^n \Gamma$, then P is terminating.*

The combination of Theorem 22 with Theorem 21 also guarantees the absence of *orphan messages*. Indeed, suppose that P is a well-typed closed process and $P \to^* Q$ where Q contains non-empty buffers. By Theorems 21 and 22 we know that $Q \to^* \sqsupseteq \mathsf{done}$. Since in a closed process messages cannot simply disappear and the only way of consuming them is by means of the reductions [R-CLOSE], [R-FORK] and [R-SELECT], we know that all the messages in Q have been received.

Finally, using the proof principle stated in Theorem 4 we obtain, as a straightforward corollary of Theorem 22, that well-typed processes are *fairly terminating* (Definition 3).

▶ **Corollary 24.** *If $P \vdash^n \Gamma$, then P is fairly terminating.*

6 Subtyping Usage Patterns

A typical usage of subtyping is to make sure that processes occurring in different branches of a non-deterministic choice $P_1 \oplus P_2$ or of a tag input $x \rhd \{\mathsf{a}_i : P_i\}_{i \in I}$ can be typed in the same typing context. Indeed, the typing rules [CHOICE] and [&] require each P_i to be typed in the same typing context (with the exception that in [&], the type of x can be different in each branch), but there are cases in which this requirement is overly restrictive.

As an example, consider a variation of the scenario depicted in Section 1 in which the worker behaves either as $\mathit{Worker}\langle y \rangle$ (as defined as in Equation (3)) or as $\mathit{BatchWorker}\langle y \rangle$ that first gathers all tasks and then sends back all the results. The $\mathit{BatchWorker}\langle y \rangle$ version can be modeled by the equations

$$\mathit{BatchWorker}(y) = y \rhd \{\mathsf{task} : \mathit{BatchWorker}\langle y \rangle, \mathsf{stop} : \mathit{SendResults}\langle y \rangle\}$$
$$\mathit{SendResults}(y) = y \lhd \mathsf{res}.\mathit{SendResults}\langle y \rangle \oplus y \lhd \mathsf{stop}.\mathsf{close}\ y$$

for which it is easy to obtain a derivation $\mathit{BatchWorker}\langle y \rangle \vdash y : S^\perp$ where S is defined as in Equation (5). Ideally, we could express the server using a non-deterministic choice, thus:

$$x \rhd \{\mathsf{req} : (y)(\mathit{Split}\langle x, y \rangle \mid (\mathit{Worker}\langle y \rangle \oplus \mathit{BatchWorker}\langle y \rangle))\}$$

however, this process turns out to be ill typed because $\mathit{Worker}\langle y \rangle$ and $\mathit{BatchWorker}\langle y \rangle$ use the channel y according to different types. Moreover, it should be clear that there is no one-size-fits-all session type that can be associated with y.

We can exploit the relation $U \leqslant S^\perp$ to make $\mathit{Worker}\langle y \rangle$ (which complies with U defined in Equation (7)) look like a process that complies with $S^\perp$. In many type systems with subtyping, this is simply achieved with an application of the *subsumption rule*. The type system we have presented for CaP does not have a subsumption rule, but this rule is derivable thanks to the formulation of [LINK]. In our specific example, we can derive

$$\dfrac{\dfrac{}{z \leftrightarrow y \vdash z : U^\perp, y : S^\perp}\,[\text{LINK}] \qquad \dfrac{\vdots}{\mathit{Worker}\langle z \rangle \vdash z : U}}{(z)(z \leftrightarrow y \mid \mathit{Worker}\langle z \rangle) \vdash y : S^\perp}\,[\text{CUT}]$$

to obtain an implementation of the streaming worker that complies with $S^\perp$. Now we derive

$$(z)(z \leftrightarrow y \mid \mathit{Worker}\langle z\rangle) \oplus \mathit{BatchWorker}\langle y\rangle \vdash y : S^\perp$$

to obtain a well-typed worker. Incidentally, this example also illustrates the reason why Theorem 20 is formulated so that, when $P \to Q$, the typing context of Q may be smaller (according to $\leqslant$) than the typing context of P. Indeed we have $(z)(z \leftrightarrow y \mid \mathit{Worker}\langle z\rangle) \oplus \mathit{BatchWorker}\langle y\rangle \to (z)(z \leftrightarrow y \mid \mathit{Worker}\langle z\rangle) \to \mathit{Worker}\langle y\rangle$ and we would not be able to obtain a typing derivation for $\mathit{Worker}\langle y\rangle \vdash y : S^\perp$.

Another typical usage pattern for links is the implementation of *delegation*, whereby the endpoint of an existing session is exchanged through another session. Because of our formulation of [LINK], in our type system this mechanism also enables a form of substitution principle for channels so that a channel of type T can be safely used where a channel of type S is expected if $S \leqslant T$ (the reason why this formulation of the substitution principle may seem to go in the wrong direction is again due to the fact that we adopt the viewpoint whereby session types describe processes rather than channels).

To illustrate this usage pattern, consider the typing derivation

$$\dfrac{\dfrac{}{y \leftrightarrow z \vdash^1 y : U, z : T}\,[\textsc{link}] \quad \dfrac{\vdots}{P \vdash^n \Gamma, x : V} \quad \dfrac{\vdots}{Q \vdash^m \Delta, x : W, y : S}}{\dfrac{x(y)[y \leftrightarrow z].P \vdash^{n+1} \Gamma, x : U \otimes V, z : T}{(x)(x(y)[y \leftrightarrow z].P \mid x(y).Q) \vdash^{n+m+1} \Gamma, \Delta, z : T}\,[\textsc{cut}]}\,[\otimes] \quad \dfrac{x(y).Q \vdash^m \Delta, x : S \parr W}{}\,[\parr]$$

which concerns the composition of a process $x(y)[y \leftrightarrow z].P$ that delegates z to another process $x(y).Q$. Note that the sender delegates a channel of type T while the receiver expects to receive a channel of type S. In order for the cut to be well typed we must have $U \otimes V \bowtie S \parr W$, that is $U \bowtie S$ and $V \bowtie W$. The application of [LINK] requires $U^\perp \leqslant T$, that is $T^\perp \leqslant U$ by Proposition 16. From this relation and $U \bowtie S$ we deduce $T^\perp \bowtie S$ by definition of $\leqslant$. Using Theorem 17 we obtain $T^\perp \leqslant S^\perp$, that is $S \leqslant T$.

7 Comparison With Other Subtyping Relations

In this section we provide a more detailed comparison between $\leqslant$ and some connected subtyping relations appeared in the literature. We show that $\leqslant$ is coarser than these relations when we focus on the family of *First-order Fairly-terminating Session Types* (FFST). Fairly terminating session types are those describing protocols that can always eventually terminate. The reason why this family is relevant for us is that *fairly terminating session types are the only inhabited types in our type system*: since well-typed processes are fairly terminating (Corollary 24), they only use sessions that can always eventually terminate. For simplicity we also focus on *first-order* session types (without the multiplicative connectives $\otimes$ and $\parr$) because the handling of higher-order communications varies substantially depending on whether the type system is based on linear logic or not.

To carry out the comparison between $\leqslant$ and the other subtyping relations we define two restricted versions of the transition relation $\xrightarrow{\alpha}$. We write $\overset{\alpha}{\longmapsto}$ for the relation inductively defined by the [MUST-*] rules and we write $\overset{\alpha}{\Longmapsto}$ for the relation inductively defined by the [MUST-*] and [MAY-*] rules. So we have $\overset{\alpha}{\longmapsto} \subset \overset{\alpha}{\Longmapsto} \subset \xrightarrow{\alpha}$ but the relation $\overset{\alpha}{\longmapsto}$ only allows *immediate transitions* and $\overset{\alpha}{\Longmapsto}$ differs from $\xrightarrow{\alpha}$ because early outputs and late inputs can only be preceeded by a bounded number of inputs and outputs, respectively. For example, if $S = \&\{\mathsf{a} : S, \mathsf{b} : \oplus\{\mathsf{c} : \mathbf{1}\}\}$ and $T = \&\{\mathsf{a} : T, \mathsf{b} : \mathbf{1}\}$, we have $S \xrightarrow{!\mathsf{c}} T$ and $S \overset{!\mathsf{c}}{\Longmapsto}$.

We define fairly terminating session types thus:

▶ **Definition 25.** *We say that S is* fairly terminating *if, for every maximal strongly fair sequence $S = S_0 \overset{\alpha_1}{\longmapsto} S_1 \overset{\alpha_2}{\longmapsto} S_2 \overset{\alpha_3}{\longmapsto} \cdots$ starting from S, there exists k such that $S_k \in \{\mathbf{1}, \bot\}$.*

Gay and Hole [28] introduced the first *synchronous subtyping relation* for session types. We qualify this relation as "synchronous" because it does not allow any form of output anticipation. However, it supports unconstrained output covariance and input contravariance as shown in its characterization below, which uses immediate transitions only.

▶ **Definition 26.** *We write $\leqslant_{\mathsf{s}}$ for the largest relation such that $S \leqslant_{\mathsf{s}} T$ implies:*

1. *either* $\mathsf{pos}(S)$ *or* $\mathsf{neg}(T)$*;*
2. *if $T \overset{?\sigma}{\longmapsto} T'$ then $S \overset{?\sigma}{\longmapsto} S'$ and $S' \leqslant_{\mathsf{s}} T'$;*
3. *if $S \overset{!\sigma}{\longmapsto} S'$ then $T \overset{!\sigma}{\longmapsto} T'$ and $S' \leqslant_{\mathsf{s}} T'$.*

Mostrous *et al.* [40, 39] studied the first *asynchronous subtypings* for session types. Chen *et al.* [11] subsequently refined the relation of Mostrous and Yoshida [39] so as to avoid orphan messages. The resulting relation has been shown to be the largest one included in the original asynchronous subtyping that is closed under duality [7]. Using our LTS we can characterize the asynchronous subtyping relations in this line of research by the following definition.

▶ **Definition 27.** *We write $\leqslant_{\mathsf{a}}$ for the largest relation such that $S \leqslant_{\mathsf{a}} T$ implies:*

1. *either* $\mathsf{pos}(S)$ *or* $\mathsf{neg}(T)$*;*
2. *if $T \overset{?\sigma}{\longmapsto} T'$ then $S \overset{?\sigma}{\Longmapsto} S'$ and $S' \leqslant_{\mathsf{a}} T'$;*
3. *if $S \overset{!\sigma}{\longmapsto} S'$ then $T \overset{!\sigma}{\Longmapsto} T'$ and $S' \leqslant_{\mathsf{a}} T'$.*

Like $\leqslant_{\mathsf{s}}$, also $\leqslant_{\mathsf{a}}$ supports unconstrained contravariance of inputs and covariance of outputs but the clauses (2) and (3) are more genereous because early/late transitions enable the anticipation of outputs. For example, $\oplus\{\mathsf{a} : \&\{\mathsf{b} : S, \mathsf{c} : \&\{\mathsf{d} : T\}\}\} \leqslant_{\mathsf{a}} \&\{\mathsf{b} : \oplus\{\mathsf{a} : S\}, \mathsf{c} : \&\{\mathsf{d} : \oplus\{\mathsf{a} : T\}\}\}$. However, the use of $\overset{\alpha}{\Longmapsto}$ (instead of $\overset{\alpha}{\longmapsto}$) in Definition 27 means that anticipated outputs can only be preceeded by a bounded number of inputs. If we consider $S = \&\{\mathsf{b} : S, \mathsf{c} : \mathbf{1}\}$ and $T = \&\{\mathsf{b} : T, \mathsf{c} : \oplus\{\mathsf{a} : \mathbf{1}\}\}$ we have $\oplus\{\mathsf{a} : S\} \not\leqslant_{\mathsf{a}} T$ because $S \overset{!\mathsf{a}}{\longmapsto}$ and $T \overset{!\mathsf{a}}{\Longmapsto}$. Note that clause (2) incorporates the condition of Chen *et al.* [11] that guarantees orphan-message freedom: if the subtype S anticipates an output and T starts with an input, *i.e.* $S \overset{!\sigma}{\longmapsto}$ and $T \overset{?\tau}{\longmapsto}$ for some σ and τ, then clause (2) guarantees that S performs a corresponding late input transition.

The use of $\overset{\alpha}{\Longmapsto}$ is both a strength and a limitation of $\leqslant_{\mathsf{a}}$. It is a strength because, just like $\leqslant_{\mathsf{s}}$, the *property that $\leqslant_{\mathsf{a}}$ prevents orphan messages holds regardless of any fairness assumption.* It is a limitation in the sense that, as we have seen since Section 1, there are contexts in which it is desirable to work with a coarser asynchronous subtyping relation. To overcome this limitation, Bravetti *et al.* [8] have characterized a *fair asynchronous subtyping relation* which, in our setting, translates to the use of $\overset{\alpha}{\longmapsto}$ instead of $\overset{\alpha}{\Longmapsto}$, as shown below.

▶ **Definition 28.** *We write $\leqslant_{\mathsf{fa}}$ for the largest relation such that $S \leqslant_{\mathsf{fa}} T$ implies:*

1. *either* $\mathsf{pos}(S)$ *or* $\mathsf{neg}(T)$*;*
2. *if $T \overset{?\sigma}{\longmapsto} T'$ then $S \overset{?\sigma}{\longmapsto} S'$ and $S' \leqslant_{\mathsf{fa}} T'$;*
3. *if $S \overset{!\sigma}{\longmapsto} S'$ then $T \overset{!\sigma}{\longmapsto} T'$ and $S' \leqslant_{\mathsf{fa}} T'$ and $T \overset{\varphi!\tau}{\longmapsto}$ implies $S \overset{!\tau}{\longmapsto}$ for every sequence φ of input labels.*

Note that the clauses (2–3) of Definition 28 are not symmetric, implying that $\leqslant_{\mathsf{fa}}$ is not closed under duality. While clause (2) supports input contravariance, differently from all the other considered subtypings *clause* (3) *disallows output covariance altogether*. If we take $S = \oplus\{\mathsf{task} : S, \mathsf{stop} : T\}$, $T = \&\{\mathsf{res} : T, \mathsf{stop} : \bot\}$ and $U = \&\{\mathsf{task} : \oplus\{\mathsf{res} : U\}, \mathsf{stop} : \oplus\{\mathsf{stop} : \mathbf{1}\}\}$ from Example 10, we have $U \not\leqslant_{\mathsf{fa}} S^\bot$ because $S^\bot \xmapsto{?\mathsf{stop}} \xmapsto{!\mathsf{res}}$ but $U \xmapsto{?\mathsf{stop}} \not\xmapsto{!\mathsf{res}}$. This formulation of clause (3) is due to the fact that $\leqslant_{\mathsf{fa}}$, unlike $\leqslant_{\mathsf{s}}$, $\leqslant_{\mathsf{a}}$ and $\leqslant$ as well, is meant to preserve fair session termination (at the type level) which requires a controlled form of output covariance that Bravetti *et al.* [8] have conservatively approximated in this way. The next example shows that removing the additional conditions in clause (3) may compromise fair termination.

▶ **Example 29.** Consider the session types $S = \&\{\mathsf{play} : \oplus\{\mathsf{win} : S, \mathsf{lose} : S\}, \mathsf{quit} : \mathbf{1}\}$ and $U = \oplus\{\mathsf{play} : \&\{\mathsf{win} : \oplus\{\mathsf{quit} : \bot\}, \mathsf{lose} : U\}\}$ where S specifies the behavior of a slot machine that allows players to play an unbounded number of games and U specifies the behavior of a player who plays relentlessly until he wins a game. Not only do we have $S \bowtie U$, but also the composition of S and U is fairly terminating in the sense that, at each stage of the interaction between player and slot machine, it is always possible to extend the interaction so that the player wins. If we now consider an *unfair* implementation of the slot machine such that its actual behavior is described by the session type $T = \&\{\mathsf{play} : \oplus\{\mathsf{lose} : T\}, \mathsf{quit} : \bot\}$, we have that $T \leqslant S$ and therefore $T \bowtie U$ still holds, but the composition of T and U is no longer fairly terminating. Therefore $T \not\leqslant_{\mathsf{fa}} S$ and rightly so. In our setting, preserving fair session termination at the type level is not so important because the type system is able to enforce fair termination at the process level anyway thanks to its logical foundation. ⌟

▶ **Theorem 30.** *For FFST session types we have* $\leqslant_{\mathsf{s}} \subset \leqslant_{\mathsf{a}}$ *and* $\leqslant_{\mathsf{a}} \subset \leqslant$ *and* $\leqslant_{\mathsf{fa}} \subset \leqslant$.

8 Related Work

Asynchronous subtyping. The asynchronous subtyping relations for binary session types defined in the literature [39, 11, 8] are formulated syntactically using *input contexts* to identify the output actions in the larger session type that have been anticipated in the smaller session type. The work of Ghilezan *et al.* [29] considers multiparty sessions: in their setting a smaller session type can anticipate outputs also w.r.t. outputs sent to a different partner. This is achieved by considering also appropriate *output contexts*. We follow a different approach: once we have defined a LTS for session types that captures their asynchronous semantics (Definitions 11 and 12), we are able to provide a characterization of $\leqslant$ that is essentially the same as the one for synchronous subtyping (Definition 26).

It is worth pointing out that there is no contradiction in the fact that our subtyping relation turns out to be coarser than others that have been proved *complete* [9] or *precise* [11, 29]. The point is that the completeness of a subtyping relation for session types is always relative to a notion of correct session composition. Depending on this notion, the induced subtyping relation may vary. Our subtyping relation is coarser than $\leqslant_{\mathsf{a}}$ [39, 11] because it relies on a fairness assumption that enables a larger degree of output anticipation and it is coarser than $\leqslant_{\mathsf{fa}}$ [8] because it is not meant to preserve fair session termination and therefore it allows (almost) unconstrained output covariance.

Asynchronous subtyping relations for session types are known to be undecidable [36, 6, 7] and $\leqslant$ is no exception (Theorems 17 and 18). Despite the proof technique adopted to prove our undecidability results is similar to those used in other papers [6, 8], we could not directly derive our results from the undecidability of other relations because $\leqslant$ is coarser.

A synchronous version of fair subtyping for session types has been studied by Padovani [41]. This relation is stricter than $\leqslant$ because it does not allow early outputs and it is meant to preserve fair session termination. Unlike the other fair/asynchronous subtyping relations for session types (with the exception of the orphan message free subtyping of Chen *et al.* [11]), $\leqslant$ is closed under duality (Proposition 16).

Fair session termination. Type systems ensuring the fair termination of binary sessions have been studied by Ciccone, Dagnino and Padovani [14, 16, 42, 13]. These works are all based on a synchronous communication model. The technique adopted in this paper for measuring processes is essentially the same presented by Dagnino and Padovani [19] which in turn is an elaboration of resource-aware session type systems [22, 21, 23]. Fair termination is a weaker form of termination [30, 2, 26]. Session type systems enforcing the termination of well-typed processes are typically based on linear logic [45, 37] where termination is a direct consequence of the cut elimination property of the logic.

Logical approaches to asynchrony and subtyping. The literature on asynchronous sessions is mostly based on calculi that are not connected to linear logic and where buffers are modeled explicitly. A notable exception is the work of DeYoung *et al.* [24] which presents a type system based on linear logic for an asynchronous calculus of sessions. In that work, asynchrony is intended as the fact that outputs are non-blocking, but no anticipation is allowed. The basic idea is the same used in the encoding of binary session into the linear π-calculus [20]: session communications make use of explicit continuation channels and outputs can be spawned as soon as they are produced leaving the sender process free to reduce further.

The deep cut reductions of CaP bear some similarities with *deep inference* [31], whereby logical rules (including the cut) may operate on "deep" fragments of a proof derivation. At this stage we are unable to say whether there is a more meaningful connection between our modeling of asynchrony and deep inference.

The use of links to incorporate subtyping in the type system echoes the approach studied by Horne and Padovani [34] who give a *coercive semantics* to subtyping in a logical setting: a subtyping relation $S \leqslant T$ corresponds to a forwarder process (*i.e.* a link) that consumes a channel of type $S^\perp$ and operates on a channel of type T. In CaP, the availability of the native link is fundamental since buffers may grow arbitrarily and the forwarder process is neither finitely representable nor finitely measurable in general.

9 Concluding Remarks

Sessions are meant to enable the compositional enforcement of safety and liveness properties of communicating processes. However, most liveness properties concerning one particular session – like the fact that a given message is eventually produced or consumed – can only be ensured if one makes the assumption that every other session eventually terminates. For this reason, the eventual termination of sessions is of primary importance. From the property that a session eventually terminates, other properties of interest usually follow "for free" as a straightforward consequence of session typing.

In this paper we have introduced a new theory of asynchronous session types that ensures the eventual termination of every session under a full fairness assumption. As a consequence, every produced message is guaranteed to be eventually consumed and every process waiting for a message is guaranteed to eventually receive one. This theory improves previous ones in a number of ways: we define a novel fair asynchronous subtyping relation that supports the

anticipation of outputs before an unbounded number of inputs, output covariance and that is closed under duality; we extend the soundness properties to multiple, possibly interleaved and dynamically created sessions; we base our theory on a calculus of asynchronous processes whose features, types and typing rules are rooted in linear logic.

Clearly, the undecidability results about the correctness of session composition and of fair asynchronous subtyping are an obstacle to the applicability of the presented theory. While decidable approximations of these notions have been presented in the literature [5, 8, 4], they cover relatively small families of session types. In future work we envision the possibility to define sound (but necessarily incomplete) algorithms for checking correct composition and subtyping inspired by the coinductive characterizations of correct session composition and of fair asynchronous subtyping introduced in this paper.

We also conjecture that our semantic approach to subtyping based on an LTS with early/late transitions scales smoothly to the multiparty setting where it may lead to simpler characterizations of (fair) asychronous subtyping relations. It may be interesting to verify the validity of this conjecture in future work.

References

1 Davide Ancona, Francesco Dagnino, and Elena Zucca. Generalizing inference systems by coaxioms. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 29–55. Springer, 2017. doi:10.1007/978-3-662-54434-1_2.

2 Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 189–198. ACM Press, 1987. doi:10.1145/41625.41642.

3 David Baelde, Amina Doumane, and Alexis Saurin. Infinitary proof theory: the multiplicative additive case. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 - September 1, 2016, Marseille, France*, volume 62 of *LIPIcs*, pages 42:1–42:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICS.CSL.2016.42.

4 Laura Bocchi, Andy King, and Maurizio Murgia. Asynchronous subtyping by trace relaxation. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*, volume 14570 of *Lecture Notes in Computer Science*, pages 207–226. Springer, 2024. doi:10.1007/978-3-031-57246-3_12.

5 Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. A sound algorithm for asynchronous session subtyping and its implementation. *Log. Methods Comput. Sci.*, 17(1), 2021. URL: https://lmcs.episciences.org/7238.

6 Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. Undecidability of asynchronous session subtyping. *Inf. Comput.*, 256:300–320, 2017. doi:10.1016/J.IC.2017.07.010.

7 Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. On the boundary between decidability and undecidability of asynchronous session subtyping. *Theor. Comput. Sci.*, 722:19–51, 2018. doi:10.1016/J.TCS.2018.02.010.

8 Mario Bravetti, Julien Lange, and Gianluigi Zavattaro. Fair asynchronous session subtyping. *Log. Methods Comput. Sci.*, 20(4), 2024. doi:10.46298/LMCS-20(4:5)2024.

9 Mario Bravetti and Gianluigi Zavattaro. Asynchronous session subtyping as communicating automata refinement. *Softw. Syst. Model.*, 20(2):311–333, 2021. doi:10.1007/S10270-020-00838-X.

10 Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Math. Struct. Comput. Sci.*, 26(3):367–423, 2016. doi:10.1017/S0960129514000218.

11 Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. On the preciseness of subtyping in session types. *Log. Methods Comput. Sci.*, 13(2), 2017. doi:10.23638/LMCS-13(2:12)2017.

12 Luca Ciccone, Francesco Dagnino, and Luca Padovani. Fair termination of multiparty sessions. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, volume 222 of *LIPIcs*, pages 26:1–26:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ECOOP.2022.26.

13 Luca Ciccone, Francesco Dagnino, and Luca Padovani. Fair termination of multiparty sessions. *J. Log. Algebraic Methods Program.*, 139:100964, 2024. doi:10.1016/J.JLAMP.2024.100964.

14 Luca Ciccone and Luca Padovani. Fair termination of binary sessions. *Proc. ACM Program. Lang.*, 6(POPL):1–30, 2022. doi:10.1145/3498666.

15 Luca Ciccone and Luca Padovani. Inference systems with corules for combined safety and liveness properties of binary session types. *Log. Methods Comput. Sci.*, 18(3), 2022. doi:10.46298/LMCS-18(3:27)2022.

16 Luca Ciccone and Luca Padovani. An infinitary proof theory of linear logic ensuring fair termination in the linear π-calculus. In Bartek Klin, Slawomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland*, volume 243 of *LIPIcs*, pages 36:1–36:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.CONCUR.2022.36.

17 Bruno Courcelle. Fundamental properties of infinite trees. *Theor. Comput. Sci.*, 25:95–169, 1983. doi:10.1016/0304-3975(83)90059-2.

18 Francesco Dagnino. Coaxioms: flexible coinductive definitions by inference systems. *Log. Methods Comput. Sci.*, 15(1), 2019. doi:10.23638/LMCS-15(1:26)2019.

19 Francesco Dagnino and Luca Padovani. small caps: An infinitary linear logic for a calculus of pure sessions. In Alessandro Bruni, Alberto Momigliano, Matteo Pradella, Matteo Rossi, and James Cheney, editors, *Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming, PPDP 2024, Milano, Italy, September 9-11, 2024*, pages 4:1–4:13. ACM, 2024. doi:10.1145/3678232.3678234.

20 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017. doi:10.1016/j.ic.2017.06.002.

21 Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. Resource-aware session types for digital contracts. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*, pages 1–16. IEEE, 2021. doi:10.1109/CSF51468.2021.00004.

22 Ankush Das, Jan Hoffmann, and Frank Pfenning. Work analysis with resource-aware session types. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 305–314. ACM, 2018. doi:10.1145/3209108.3209146.

23 Ankush Das and Frank Pfenning. Rast: A language for resource-aware session types. *Log. Methods Comput. Sci.*, 18(1), 2022. doi:10.46298/LMCS-18(1:9)2022.

24 Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. Cut reduction in linear logic as asynchronous session-typed communication. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, volume 16 of *LIPIcs*, pages 228–242. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPICS.CSL.2012.228.

25 Amina Doumane. *On the infinitary proof theory of logics with fixed points. (Théorie de la démonstration infinitaire pour les logiques à points fixes)*. PhD thesis, Paris Diderot University, France, 2017. URL: https://tel.archives-ouvertes.fr/tel-01676953.

26 Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer, 1986. doi:10.1007/978-1-4612-4886-6.

27 Simon J. Gay. Subtyping supports safe session substitution. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 95–108. Springer, 2016. doi:10.1007/978-3-319-30936-1_5.

28 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005. doi:10.1007/S00236-005-0177-Z.

29 Silvia Ghilezan, Jovanka Pantovic, Ivan Prokic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for asynchronous multiparty sessions. *ACM Trans. Comput. Log.*, 24(2):14:1–14:73, 2023. doi:10.1145/3568422.

30 Orna Grumberg, Nissim Francez, and Shmuel Katz. Fair termination of communicating processes. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, pages 254–265, New York, NY, USA, 1984. Association for Computing Machinery. doi:10.1145/800222.806752.

31 Alessio Guglielmi and Lutz Straßburger. Non-commutativity and MELL in the calculus of structures. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2001. doi:10.1007/3-540-44802-0_5.

32 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

33 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFB0053567.

34 Ross Horne and Luca Padovani. A logical account of subtyping for session types. *J. Log. Algebraic Methods Program.*, 141:100986, 2024. doi:10.1016/J.JLAMP.2024.100986.

35 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.

36 Julien Lange and Nobuko Yoshida. On the undecidability of asynchronous session subtyping. In Javier Esparza and Andrzej S. Murawski, editors, *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10203 of *Lecture Notes in Computer Science*, pages 441–457, 2017. doi:10.1007/978-3-662-54458-7_26.

37 Sam Lindley and J. Garrett Morris. Talking bananas: structural recursion for session types. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 434–447. ACM, 2016. doi:10.1145/2951913.2951921.

38 Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994. doi:10.1145/197320.197383.

39 Dimitris Mostrous and Nobuko Yoshida. Session typing and asynchronous subtyping for the higher-order π-calculus. *Inf. Comput.*, 241:227–263, 2015. doi:10.1016/J.IC.2015.02.002.

40 Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 316–332. Springer, 2009. doi:10.1007/978-3-642-00590-9_23.

41 Luca Padovani. Fair subtyping for multi-party session types. *Math. Struct. Comput. Sci.*, 26(3):424–464, 2016. doi:10.1017/S096012951400022X.

42 Luca Padovani. On the fair termination of client-server sessions. In Delia Kesner and Pierre-Marie Pédrot, editors, *28th International Conference on Types for Proofs and Programs, TYPES 2022, June 20-25, 2022, LS2N, University of Nantes, France*, volume 269 of *LIPIcs*, pages 5:1–5:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.TYPES.2022.5.

43 Luca Padovani and Gianluigi Zavattaro. Fair termination of asynchronous binary sessions. Technical report, University of Bologna, 2025. doi:10.48550/arXiv.2503.07273.

44 Rob van Glabbeek and Peter Höfner. Progress, justness, and fairness. *ACM Comput. Surv.*, 52(4):69:1–69:38, 2019. doi:10.1145/3329125.

45 Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014. doi:10.1017/S095679681400001X.

Reusing Highly Optimized IR in Dynamic Compilation

Andrej Pečimúth
Charles University, Prague, Czech Republic
Oracle Labs, Prague, Czech Republic

David Leopoldseder
Oracle Labs, Vienna, Austria

Petr Tůma
Charles University, Prague, Czech Republic

Abstract

Virtual machines (VMs) with dynamic compilers typically specialize compiled code to the state of the running VM instance and thus cannot reuse the code between multiple runs of the same application. The JIT compiler must recompile the same methods for each run of the application separately, which can prolong the application's warmup time. We propose a technique to reduce compilation time by reusing a highly optimized intermediate representation (IR). We achieve this by tracing compiler-interface calls during compilation. The validity of the specializations in the IR is verified during a replay stage, and the replay also facilitates the relocation of runtime object references. The IR is stored on a compilation server, which can compile it to machine code and provide the code to local or remote VM instances. We implemented a compilation server with IR caching for GraalVM, a high-performance production-grade Java Virtual Machine (JVM). We present an evaluation based on four industry-standard benchmark suites. In each suite, our approach reduces compilation time by 23.6% to 36.8% and warmup time by 13.1% to 21.2% on average while preserving peak application performance.

2012 ACM Subject Classification Software and its engineering → Dynamic compilers; Software and its engineering → Just-in-time compilers; Software and its engineering → Runtime environments

Keywords and phrases code reuse, compilation time, warmup, remote compilation, dynamic compilation, virtual machines

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.25

Funding This research was supported in part by projects GAUK 74824 and SVV 260821.

1 Introduction

Modern virtual machines (VMs) achieve high application performance by leveraging optimizing dynamic compilers. After a VM starts, the application typically experiences a warmup phase [2] until the performance-sensitive methods are compiled at the highest optimization level. The VM usually has to repeat these compilations for each run of the application. The latency of the compilation tasks is one of the factors contributing to the application's cold start. This problem is amplified when many VM instances execute the same application, such as in horizontally scaled deployments with demand-based elasticity and in serverless computing [5].

The goal of our work is to reuse compilation results from a past run of the same application on the same VM version and platform. For brevity, we refer to a single run of the application as a *VM run*, and we want to reuse compilations from a *source VM instance* in a *target VM instance*. Reusing past compilation results reduces the work performed by the JIT compiler in a VM run (compilation time). Depending on the workload's sensitivity to the latency of

© Andrej Pečimúth, David Leopoldseder, and Petr Tůma;
licensed under Creative Commons License CC-BY 4.0

Listing 1 An example where the compiler may specialize the compiled `log` method to the state of the running VM instance.

```
1  class Logger {
2      static final int level = Integer.parseInt(System
3          .getProperty("logLevel"));
4
5      static final PrintStream stream = createStream();
6
7      static void log(int logLevel, String message) {
8          if (logLevel >= level) {
9              stream.println(message);
10         }
11     }
12 }
```

compilation tasks, we can also reduce the time it takes for the application to reach peak performance (warmup time). However, it is challenging to reuse the code compiled by a highly optimizing compiler because the compiled code is specialized to the source VM instance.

To illustrate how a dynamic compiler tailors code to the source VM instance, consider the `log` method in Listing 1. The compiler may read the value of the `level` field at compilation time and embed its value in the machine code. Similarly, the compiler can read the value of the `stream` field, obtaining an object address that it can embed in the generated code.

These transformations improve performance by moving computation from run time to the compilation stage, but they pose a challenge to code reuse. For example, the compiled code implicitly assumes that the `level` field holds a particular value. The field's value could differ in another VM run, rendering the code incorrect for reuse. Moreover, the reference to the Java object constant obtained from the `stream` field requires relocation before we can reuse the code in another VM run. We give an overview of more problems with code reuse in Section 2.

The main idea of our paper is that the specializations of the code to the state of the running VM can be valid even in future VM runs of the same application. We can reuse compilation results from the source VM instance in the target VM instance, provided that it is possible to validate that the reused code would behave correctly in the target VM instance and relocate the references to VM objects. Both issues are solved by recording and replaying a *compilation trace*. To reduce the overall compilation time with this technique, we also need to ensure that validation succeeds for enough compilation units when the source and target VM instances execute the same application. We achieve this by modifying the compiler and caching the optimized intermediate representation (IR) rather than the compiled code because the IR is less specific to the target VM instance and easier to patch.

To reuse the IR compiled for method `log`, we must verify that the transformations performed during the original compilation for the source VM instance are correct in the target VM instance. Therefore, during compilation in the source VM instance, we record the calls from the compiler to the VM that query the VM's state, obtaining a compilation trace such as the one in Listing 2. In the target VM instance, we replay these calls to validate that they produce compatible results, and we use the computed VM object references as relocations. `%2` is a *trace variable* that stands for the `level` field, which we obtained as index `32` into the constant pool of the `Logger` class. The field's value read by the compiler was `4`,

Listing 2 A snippet of a compilation trace captured during a compilation of the `log` method from Listing 1.

```
1                                (%0 is the Logger#log method)
2  declaringClass(%0) = %1       (%1 is the Logger class)
3     getBytecode(%0) = ...      (the result is binary data)
4  lookupField(%1,32) = %2       (%2 is the Logger#level field)
5    isFinalField(%2) = true
6  readFieldValue(%2) = 4
7  lookupField(%1,48) = %3       (%3 is the Logger#stream field)
8    isFinalField(%3) = true
9  readFieldValue(%3) = %4       (%4 is a Java object constant)
10 isConstantNull(%4) = false
```

which we can validate in the target VM instance. The value read from the `stream` field is an object constant stored in trace variable `%4`, which we can use for relocation. We explain compilation tracing and replay in Section 3.

We implemented the approach for GraalVM [28]. GraalVM is the HotSpot [29] Java Virtual Machine (JVM), where the C2 compiler [35] is replaced with the high-performance GraalVM compiler. Based on the GraalVM compiler, we created a compilation server [18, 19, 1, 24] capable of compiling code for local and remote client VM instances over the network. The server can also cache the IR intended for reuse and compile the stored IR into machine code. Thanks to this design, multiple client VMs can share and benefit from the IR cache. To minimize the overall compilation latency, the client VMs employ a *hybrid compilation* strategy: they offload only large compilations where the server can leverage the IR cache. All other compilations are completed using the VM's local JIT compiler. We describe the design in Section 4.

We evaluate the approach on the DaCapo 23.11 MR1 [3], Renaissance 0.15 [39], and ScalaBench 0.1.0 [44] benchmark suites. Compared to the unmodified GraalVM baseline, hybrid compilation using a co-located compilation server with a pre-populated IR cache decreases compilation time by 23.6% to 36.8% and warmup time by 13.1% to 21.2% in every evaluated suite on average. We verified using the DaCapo suite that the benefits of our approach remain substantial when the server and the client VM are separated by a local network. We present the evaluation in Section 5 and describe the related work in Section 6.

Although we focus on the context of a modern JVM, the key concepts of the approach and the challenges faced by our implementation are likely transferable to other similar platforms. Specifically, the technique to cache and reuse the optimized IR assumes that there is a well-defined and interceptable interface between the VM and the compiler and that the compiler starts with an IR that is not overly specific to the target VM instance (other than references to VM objects). We believe these assumptions to be a common design element of environments with dynamic compilation, though other technical details can also influence the practicality of the approach.

In summary, we contribute the following:

- An approach to reuse highly optimized IR by compilation tracing and replay.
- A case study on a production-grade VM outlining the changes to the compilation pipeline to leverage the approach to achieve an overall improvement in compilation metrics.
- An empirical evaluation on industry-standard benchmarks, which shows that the technique reduces compilation time and can improve warmup time while preserving the application's peak performance.

Listing 3 A class with a method whose compiled code may not be reusable.

```
1 class Filter {
2     static final Options options = getOptions();
3
4     void filter(Collection<Integer> numbers) {
5         Predicate<Integer> isOutlier =
6             (n) -> n > options.threshold;
7         numbers.removeIf(isOutlier);
8     }
9 }
```

2 Reusability of Optimized Code

The code compiled by an optimizing dynamic compiler is usually not reusable across VM runs. This section summarizes the obstacles to code reusability in a modern JVM [37], with practical examples from the HotSpot JVM [29]. These challenges are caused by dynamic code loading and by optimizations that specialize the code to the state of the target VM instance, which are concepts relevant to many modern VMs.

2.1 Equivalence of Loaded Classes

A dynamic compiler tailors the code to the definitions of the loaded classes at a given time. To reuse code compiled for a source VM instance, we must ensure that the classes on the target VM instance are equivalent to the classes loaded by the source VM instance. However, a modern JVM may generate classes at runtime and modify the bytecode for optimization purposes. An application can trigger class loading at any time, so we must not expect that the target VM has precisely the same set of loaded classes.

For example, consider the `filter` method in Listing 3. In the HotSpot JVM, the first execution of the method runs the initialization code for the lambda expression [27], which loads a dynamically generated class representing the function object instance. We must ensure that even the generated classes have compatible definitions between VM runs. However, the generated class is assigned a name that is not stable between VM runs [6]. In addition, HotSpot modifies an operand of a bytecode instruction to link it with a function object factory. As a result, the bytecode of the `filter` method may also differ between VM runs.

2.2 Specialization to Application State

The compiler may evaluate loads from static final fields and even final instance fields if the receiver is a constant. This transformation removes a potentially expensive operation from the code and may also enable other optimizations. For example, in the body of the lambda expression in Listing 3, the compiler may fold the load of the `options` field. If `threshold` is a final instance field, the compiler can fold the load of this field as well. Although these specializations improve code quality, they restrict reusability. In future VM runs, the values of the static and instance fields could differ, rendering the compiled code incorrect for reuse.

2.3 Speculations and Assumptions

The compiler's optimization decisions may involve speculation [11] based on the application's past behavior and profiling feedback. These speculations and assumptions narrow the scope where the compiled code is applicable, but they may improve performance.

For example, the call to the `removeIf` method in Listing 3 produces a null pointer exception if the call's receiver is `null`. If such a situation never occurred previously, the compiler may speculatively omit the exception-throwing branch and instead emit a deoptimization [16]. This allows the compiler to focus only on the likely execution paths, but such compiled code is only beneficial if `removeIf` is not invoked with a null argument.

The compiler can make assumptions about the loaded classes. For example, `List` is an interface with possibly multiple implementations, making the `removeIf` call indirect. However, if only a single implementation is loaded, the compiler converts the call to a direct one and records the assumption about the interface having a single implementation. The VM invalidates the compiled code when it loads another implementation of the `List` interface. Such assumptions limit the reusability of the code, but they typically improve performance.

Lastly, the compiled code may contain implicit assumptions about the VM's state. For example, in Java, classes have static initializers [25], which the runtime invokes when a class is first referenced. If a class is already initialized, the compiled code does not need to check its initialization status. However, such code may not be reusable in another VM run, where the same classes might not yet be initialized.

2.4 Coupling with VM State

The compiler may encode absolute addresses, relative addresses, and offsets of various VM objects in the compiled code. These values usually change between VM runs, making the compiled code not directly reusable. For example, the call to the `removeIf` method in Listing 3 is compiled with the callee's address embedded in the machine code.

We could attempt to reuse the code by finding relocations for the referenced VM objects and patching their values in the machine code. Java objects have identities, so we must relocate them with identical objects to preserve the program's correctness. However, patching the values in the machine code is difficult because accessing the replacement value may require a different instruction sequence than the one emitted initially.

3 Compilation Tracing and Replay

Although dynamic compilers tailor code to the running VM, these specializations can often remain valid even in future VM runs, much like past profiles can inform future optimization decisions. To capture all dependencies of the compilation result on the state of the running application and source VM instance, we record the compiler's calls through the VM's compiler interface, yielding a compilation trace. In the target VM instance, where we want to reuse the compilation, we replay the compilation trace to validate that the compilation is compatible with the actual state of the running application and to compute relocations for VM object references.

Figure 1 shows a summary of compilation tracing, which this section explains in detail. In a *tracing stage*, we compile a method in a source VM instance up to a *point of reuse* to obtain the compilation unit's optimized IR (step 1). We use the compilation trace (2) to replace VM object references in the IR with trace variables (3). Lastly, we store the IR (4) along with the compilation trace in the IR cache (5). The IR depicted in Figure 1 is that of the `log method` from Listing 1. Interpreting the IR semantics is not essential to understanding the approach; Figure 1 uses Graal IR [10] as an illustration.

In a *replay stage*, we retrieve the compilation trace from the IR cache (step 6) to replay the trace in a target VM instance and validate the results. If the validation succeeds (7), we retrieve the cached IR (8) and replace the references to trace variables in the IR with references to VM objects (9). Finally, we compile the IR to machine code (10) and prepare it for execution on the target VM.

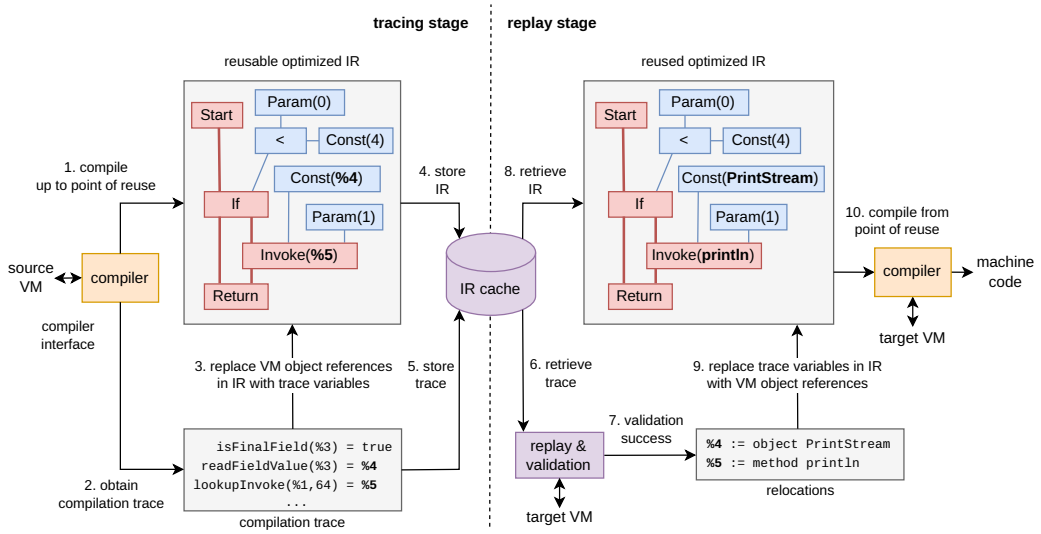


Figure 1 Overview of compilation tracing and replay. The IR format is based on Graal IR [10]; thick lines are control-flow edges, thin lines are data-flow edges.

3.1 Point of Reuse

We cache partial compilation results (the optimized IR) [37] instead of the final compiled code because it is less specialized to the state of the target VM instance. To enable IR reuse, we trace the compiler-interface calls in the compiler passes up to the designated point of reuse. The compiler can resume the compilation pipeline starting from the point of reuse in order to compile the cached IR to machine code.

The point of reuse should be as late as possible in the compilation pipeline to minimize the amount of work necessary to finish the compilation. However, it must be early enough so that references to VM objects can be relocated. If the point of reuse is early enough so that it is still possible to optimize the IR, the transformations that make the IR overly specific to the running VM instance can be deferred [37] until after the point of reuse. With these transformations deferred, the compiled machine code can benefit from specialization to the target VM instance while keeping the cached IR reusable.

In summary, the advantages of caching the IR are that

(i) patching the IR is easier than the compiled code,

(ii) we avoid all coupling introduced by late compiler passes after the point of reuse, and

(iii) transformations that introduce coupling to the source VM instance can be deferred until after the point of reuse.

3.2 Tracing

Compilation tracing records inputs to the compiler that influence a method's compilation. Therefore, the trace contains all assumptions about the application's and VM's state. Moreover, the trace captures the origin of every VM object reference that appears in the IR.

In the tracing stage, we run the compilation pipeline up to the point of reuse, obtaining an IR and an associated compilation trace. The compilation trace is a list of the compiler's queries sent to the VM with their arguments and results.

A value in the compilation trace may be a literal or a trace variable substituting a VM object. For example, the bytecode of the compiled methods and the values of primitive constants are embedded in the trace as literals. In Figure 1, trace variable `%4` substitutes a Java object constant obtained by reading the value stored in field `%3`. Trace variable `%5` substitutes a VM's representation of a method.

After obtaining the IR and the compilation trace, we replace all references to VM objects in the IR with the trace variables. Finally, we store the IR and the compilation trace in the IR cache.

3.3 Replay

In the replay stage, we retrieve the IR and the associated compilation trace from the IR cache. The goal is to verify that we can safely reuse the IR without causing the application to misbehave and to finish compiling the IR to machine code.

We replay the trace by iterating over the individual operations. For each operation, we first replace the trace variables in the argument list with the instances of VM objects they substitute. After that, we execute the operation in the target VM and examine its return value. If the trace's result is a trace variable referenced for the first time, then the return value of the operation defines the value of that trace variable. Otherwise, we validate that the return value is compatible with the result captured in the trace. Compatibility can be verified by checking that the results are equal or with an operation-specific check, as we explain in Section 4.

If the validation fails for an operation, we cannot use the IR for the target VM instance at this time. If the validation succeeds, all trace variables have assigned values. We proceed by substituting the trace variables in the IR with their values. Finally, we run the rest of the compilation pipeline, starting from the IR and obtaining the machine code for the target VM instance.

4 Implementation

Compilation tracing serves as a blueprint for validating the reusability of prior compilation results. To improve the overall compilation time using this approach, we must ensure that the cached IR is generally reusable, and we need a mechanism to share it among multiple VMs that might execute on different machines.

We implemented compilation tracing in GraalVM [28]. GraalVM is the HotSpot JVM [29] where the GraalVM compiler replaces the C2 compiler [35]. On top of this, we created a compilation server [19, 24] capable of IR caching, allowing local or remote VM instances to share a single IR cache.

Figure 2 shows an overview of our design, which this section explains in detail. In a *training run* (left side of Figure 2), a source VM instance executes an application and offloads compilation tasks to the compilation server (step 1). The server queries information from the source VM instance (2), caches the compilation result (3), and returns the compiled code to the source VM (4).

A subsequent run of the application may be executed as a *replay run*, which is sketched on the right side of Figure 2. In a replay run, the target VM obtains a compilation trace from the server (step A1), replays and validates it (A2), and returns the computed relocations to the server (A3). The server compiles the cached IR (A4, A5) and returns the compiled code to the target VM (A6). Compilation tasks that the target VM decides not to offload to the server are completed using the VM's local JIT compiler (step B1).

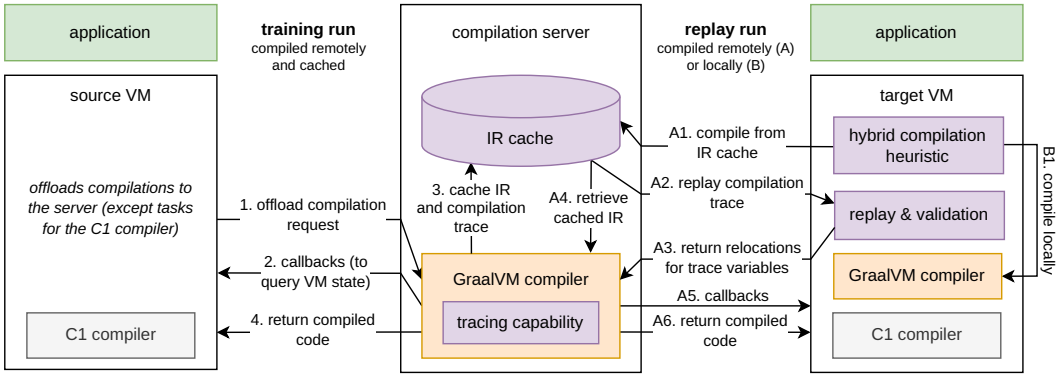


Figure 2 Overview of the implementation with a compilation server.

The compilation server is a standalone Java application that uses the GraalVM compiler as a library. The compilation client is implemented as an additional entry point of the GraalVM compiler, with some modifications and additions to the compiler and the VM's compiler interface.

4.1 Practical Compilation Tracing

We implemented compilation tracing by instrumenting the classes comprising the VM's compiler interface, creating wrapper classes with a compatible interface. The traced operations generally align with the granularity of the compiler-interface [40] calls exposed by the VM. A key challenge of compilation tracing is handling information that varies between VM runs, such as the bytecode, profiles, and dynamically generated classes.

Reusing Graal IR. The GraalVM compiler's compilation pipeline is divided into two stages: the front end and the back end. The front end parses the bytecode into a sea-of-nodes-based IR [10], optimizes it, and lowers the IR to a control-flow-graph-based representation for the back end. The compiler performs the majority of the optimization effort in the front end. Based on our measurements (Section 5), the compiler spends on average about 81% of the time in the front end, 14% in the back end, and 5% in the garbage collector triggered after compilation or waiting for the VM to prepare the compiled code for execution. We cache and reuse the IR at a point relatively late in the front end – before the compiler lowers portable address nodes to machine-specific nodes. This allows us to patch the IR with VM-specific information and defer the folding of field loads after the point of reuse (Section 4.2), while still saving a significant portion of compilation time.

Tracing Profiling Information. The IR stored in the cache is compiled using profiles recorded by the source VM instance. When the target VM instance reuses the IR from the cache, the compiled code should be performant and unlikely to deoptimize [16] in the future. Therefore, the cached IR should ideally be compiled using accurate and mature profiles.

After a target VM instance starts, we want to reuse IR as early as possible to leverage the performance of the compiled code. By reusing cached IR early in an application's lifecycle, we can potentially benefit from the fact that the profiles used to compile the IR may more accurately characterize the application's future behavior than the profiles collected by the target VM instance up to that point. For this reason, the replay stage does not verify the

Listing 4 Encoding of the operation that returns the recorded receiver types.

```
1 getReceiverTypes(%0, 42) = {
2   com.example.Map/com.example.Main: %1,
3   com.example.Set/com.example.Main: %2,
4 }
```

equivalence of profiling information between the source and target VM instances, and the compilation trace does not capture any profiling information other than receiver-type profiles. Receiver-type profiles are used during replay to locate some VM objects.

Receiver-Type Profiles. The VM records the types of receiver objects at virtual call sites to guide the inlining [38] heuristics. The compiler queries the receiver-type profiles from the VM, and the receiver types may be referenced in subsequent operations. Therefore, we must assign trace variables to the recorded receiver types. We do this by capturing the receiver-type profiles in the compilation trace.

Listing 4 illustrates the encoding of these profiles in the form of globally unique type identifiers mapped to trace variables. In the example, the `getReceiverTypes` operation returns the receiver types for a particular call site identified by the bytecode offset 42. The recorded receiver types are `com.example.Map` and `com.example.Set` assigned to trace variables `%1` and `%2`, respectively.

Since multiple class loaders may load a class with the same name, the globally unique identifier of a type comprises the class name and an identification of the type's class loader. We identify the class loader using the heuristic introduced by JITServer [19]: the loader's identifier is the name of the first class it loaded. In the above example, `com.example.Main` is the class loader's identifier.

In the replay stage, the goal is to map the trace variables to the respective types identified by the globally unique name. If we cannot find a type (it may not be loaded yet), the replay cannot continue and ends with a validation failure.

Dynamically Generated Classes. During execution, the VM may dynamically generate classes whose class name is not a stable identity across VM runs. For example, the HotSpot JVM loads generated classes to support lambda expressions and other core features. These generated classes are loaded as *hidden classes* [6], which are intentionally designed not to be discoverable by name, and their VM-assigned names vary between VM runs. The VM may compile the methods of hidden classes, and they may also appear in profiles and get inlined. To successfully validate compilation traces referencing hidden classes, we create stable class names based on a digest computed from the binary data that defines the hidden class and also from the Java objects passed to the static initializer of the hidden class.

Additionally, the code of an application may dynamically generate classes without using the hidden class feature. We can successfully reuse the compilations referencing these classes if their definitions are stable between VM runs and their class names uniquely identify them. However, these conditions are not always satisfied: for example, the Catalyst Optimizer [13] from the Apache Spark framework generates and assigns the exact same name to every generated class. Our prototype does not support such a use case and experiences validation failures. The problem could be fixed by adding a digest of the class definition to the class name.

Bytecode Modification. The VM can modify the operands of some bytecodes for optimization purposes in a way that is not stable across runs. The compiler may use these values as arguments in compiler-interface calls. To ensure the replayability of the compilation trace, we do not embed any VM-specific operand values in the compilation trace. Instead, we use an indirection specifying where to find the actual value in the target VM instance.

For example, our compiler interface offers an operation called `lookupInvoke`, which takes a class and index argument and returns the representation of the method invoked at that index. The compiler may obtain the index from an operand of a bytecode instruction and call the `lookupInvoke` operation with the operand value. In the compilation trace, we encode the operand value indirectly using the expression `operandValue(%0, 16)`.

```
lookupInvoke(%1, operandValue(%0, 16)) = %2
```

During replay, we find the concrete index as the value in the bytecode of the `%0` method at bytecode offset 16. Then, we can replay the `lookupInvoke` operation with the concrete argument value specific to the target VM instance.

Lastly, we must also validate that the bytecode of a particular method on the target VM instance matches the bytecode of this method on the source VM instance. We do this by comparing a canonicalized version of the bytecode, which does not contain any VM-specific values.

4.2 Improving IR Reusability

To improve the overall compilation time using our approach, we need to increase the chances that the IR stored in the cache passes validation so that we can reuse it. To this end, we validate the results of some operations with less strict checks than equivalence. Moreover, we use heuristics to defer optimizations that could break the IR's reusability.

Relaxed Validation Checks. We reuse the cached IR even if it is more general than required by the target VM instance. Specifically in the context of Java, if the cached IR leverages an assumption about the loaded classes (e.g., only a single implementation of an interface is loaded), we must check that the assumption holds in the target VM instance. However, if such a fact holds in the target VM instance but the cached IR does not leverage it, we consider this a validation success. Such a situation is expected when the cached IR is more mature than the state of the target VM instance.

Similarly, classes, methods, and fields in Java have initialization stages [25]. To facilitate IR reuse, we only require that a class, method, or field is in the same or later initialization stage on the target VM instance as in the source VM instance.

Patterns Preventing Reuse. If the compiler specializes the IR to computed constants whose values could change between VM runs, the IR may not be reusable. For example, consider the `ThreadLocal` implementation [4] shown in Listing 5. When a `ThreadLocal` instance is initialized, the `threadLocalHashCode` field is assigned a value based on a global counter. If a thread-local value is stored in a static final field, the thread-local's hash code is likely to differ between VM runs. The compiler may embed this field's value into methods that manipulate the thread-local value, leading to non-reusable IR and replay failures. In OpenJDK, the `ClassValue` implementation [41] also uses the same pattern, and similar patterns may appear in the code of applications. We can mitigate these problems using deferred optimization.

Listing 5 A snippet of the `ThreadLocal` implementation in OpenJDK (simplified).

```
class ThreadLocal {
    final int threadLocalHashCode = next.getAndAdd(INCREMENT);
    static AtomicInteger next = new AtomicInteger();
}
```

Deferred Optimization. We can allow the compiler to fold loads of computed constants while keeping the IR reusable by deferring [37] this optimization to the replay stage. In the context of Java, if we know that a particular field like the one in Listing 5 hinders reusability, we do not allow the compiler to fold a load of this field in the tracing stage. The compiler folds the field load in the replay stage when the VM-specific value of the field is known.

Deferring the folding of all fields is a disruptive change that leads to performance degradation [37]. Therefore, we target only fields that may cause validation failures. Our prototype implementation uses a simple condition: if the field's type is `int` and its name contains `hash`, we defer the folding. A more refined approach could select these fields based on past observations: if validation fails due to mismatching values read from a field, the field's folding would be deferred in future tracing stages.

4.3 Remote Compilation

We implemented a compilation server [19, 1, 24] with the capability to reuse the IR from past compilations. This allows multiple client VMs to share a single IR cache. Figure 2 shows a summary of the design. We implemented remote compilation and IR caching only for the top-tier GraalVM compiler since the low-tier C1 compiler [20] completes compilation tasks quickly and with a lower optimization level. To minimize compilation latency, the client VMs utilize hybrid compilation, in which a heuristic decides whether a method should be compiled by the VM's local JIT compiler or using the cached IR on the server.

Compilation Server. In a remote compilation setup, a client VM sends compilation requests to the server. The server compiles methods for the client VM's platform, utilizing only the information provided by the client. It is usually not possible to pack all the necessary information [19] in the forwarded compilation request. Instead, the server queries the client's VM state via remote callbacks. We employ aggressive caching and prefetching to minimize the number of remote callbacks. After completing the compilation, the server returns the compiled code to the client VM.

Training Run. The compilation server starts with an empty IR cache, which gets populated by a client VM that connects in training mode. In a training run, the compilation server handles requests with compilation tracing enabled. When the compilation pipeline reaches the point of reuse, the server stores the compilation trace and copies the IR into the cache. After that, the server completes the rest of the compilation pipeline and returns the compiled code to the client VM.

The IR cache is indexed by a symbolic representation of the compilation request. This symbolic representation comprises the fully qualified name, arguments, and the return value of the method to be compiled and the entry point if it is an on-stack-replacement (OSR) [12] compilation.

Replay Run. In a replay run, the client VM either chooses to compile locally or issues a remote compilation request. The decision is based on a server-provided list of compilation requests and other conditions, which we summarize later. In case of a remote compilation, the server looks up a cached compilation result in the IR cache and transfers the stored compilation trace to the client VM. The client VM replays and validates the trace. If validation succeeds, the client replies with the computed mapping of trace variables to VM objects. The server substitutes the trace variables with the VM object references in the IR, completes the compilation, and returns the compiled code. In case of a validation failure, the client VM completes the compilation request using its local JIT compiler instead.

The client VM must also handle recompilations. A recompilation of a method previously compiled from the cached IR may occur after deoptimization [16] or a violated assumption about the class hierarchy. In such a case, we cannot reuse the same IR to get a usable compilation result and prevent endless deoptimization. For this reason, the client VM handles recompilations by compiling from scratch using the local JIT compiler.

Permanent and Transient Validation Failures. Retrying to replay a compilation trace that previously failed can improve the latency of some compilation tasks. As explained above, the VM compiles a method using the local JIT compiler if the validation of a compilation trace fails. However, the code compiled by the local JIT may get invalidated later, and the VM may reissue an identical compilation request. We attempt to replay and validate the compilation trace again if the reason for the past failure is an uninitialized or unresolved class, unresolved field, or an uninitialized method. We consider these failures to be *transient* because they may be resolved by the VM performing the required initialization [25]. We consider other kinds of validation failures to be *permanent*, and the client VM never retries to replay the same compilation trace again.

Hybrid Compilation. To improve the overall compilation time, client VMs offload only the compilation requests where the IR cache is expected to bring a benefit. If there is no expected benefit, the client VM should avoid increasing compilation latency by communicating with the server. We estimate the potential savings based on the time the server spent compiling the bytecode to the reusable IR in the training run, which we refer to as the *pre-compilation time*. The pre-compilation time is an upper-bound estimate of the time a client VM can save by compiling the cached IR on the server rather than locally from scratch. Therefore, at the beginning of a replay run, the client VM obtains a list of compilation requests from the server for which the server has cached IR and the pre-compilation time above a client-provided threshold.

In the evaluated prototype, we set the pre-compilation time threshold to 20 milliseconds. The optimal threshold value depends on the workload, the communication latency between the client and the server, and the computing resources available to the client and server.

In summary, a client VM offloads a compilation request to the server if and only if all conditions are satisfied:

(i) The compilation request is in the server-provided list.

(ii) There was no previous attempt to complete this compilation request that ended with a permanent validation failure.

(iii) This compilation request was not already completed using the cached IR.

5 Evaluation

The goal of this section is to understand how hybrid compilation impacts the performance characteristics of standard Java and Scala workloads. We evaluate the workloads from major benchmark suites [3, 39, 44]. To ensure that the improvements are not achieved by offloading work to an additional machine or allocating more resources to the JIT compiler, we run the compilation server co-located on the same machine as the client VM, and we do not increase the number of concurrent compilation tasks. We first show the impact on key compilation metrics reported by the HotSpot JVM [29]. Then, we analyze the impact on peak performance and warmup time. Lastly, we evaluate how successful our prototype is in reusing and validating the cached IR and explain the reasons for the observed validation failures.

In a setup where a client VM connects to a remote server, the network latency adds to the compilation latency. For this reason, we consider low-latency environments (e.g., a fast local network) the most appropriate deployment option. The benefits would likely diminish in environments with higher latency. To demonstrate that the presented improvements are applicable to deployments on the local network, we also conduct experiments using the DaCapo benchmarks [3] where the client VM connects to a dedicated remote compilation server through the local network. We do not increase the number of concurrent compilation tasks to keep the results comparable to the co-located and baseline setups.

Notes on Performance Evaluation. Every benchmark suite we evaluate contains multiple benchmarks. The harness of every suite executes a benchmark by iterating a fixed workload multiple times in a dedicated JVM instance. The *iterations* usually get progressively faster as the VM warms up. The harness repeats the workload until it executes for at least the preset duration (e.g., 12 minutes). Due to the inherent non-determinism of the JVM, we also need to execute each benchmark for multiple VM runs [14]. The key metrics we evaluate are the wall-clock time it takes to complete an iteration (*iteration time* or *duration*) and the timestamps of their completion since the start of the JVM instance (*iteration timestamp*). In the rest of the section, we always aggregate absolute metric values using the arithmetic mean and relative metric values (ratios) using the geometric mean.

Hardware and Software. We executed the experiments on 20 identical blade servers, each equipped with an Intel Xeon CPU E3-1230 v6 (with four cores) and 32 GB of main memory. The blades are located in the same enclosure with separate 1 Gbps Ethernet network cards connected through a dedicated network router. We disabled hardware multithreading and power management. The blades run Fedora Linux 35 with kernel version 5.16.11.

We implemented our prototype on top of a development version of GraalVM Enterprise based on a development version of OpenJDK 22. We use the same binaries and the same JVM options for the modified and baseline runs, using additional JVM options to enable the remote JIT functionality. This avoids the potential variance introduced by the ahead-of-time compilation of the compiler itself. In every benchmark run, we use a JVM option to disable compressed class pointers (`-XX:-UseCompressedClassPointers`) because our prototype does not implement patching the compression parameters in the cached IR. We disable isolation between compiler threads (`-XX:JVMCIThreadsPerNativeLibraryRuntime=0`) to enable data sharing required by the client's compilation threads. In addition, we prevent the VM from shutting down the compiler runtime after a period without compilation activity (`-XX:JVMCICompilerIdleDelay=0`) to avoid potentially costly reinitialization of the runtime

for the client. The options `-Xms12G` and `-Xmx12G` set a fixed Java heap size, and we use the default garbage collector (`-XX:+UseG1GC`). Lastly, we use options to enable selected counters and timers in the GraalVM compiler to collect compilation statistics.

We use modified versions of the DaCapo 23.11 MR1 [3] harness and the ScalaBench 0.1.0 [44] harness to collect additional statistics. We execute the Renaissance 0.15 [39] benchmark suite with additional plugins to collect the same statistics. All DaCapo benchmarks are executed using the default workload size. We execute the ScalaBench benchmarks with default or larger sizes, selected to ensure the iterations are not excessively short. We exclude the workloads incompatible with OpenJDK 22.

5.1 Compilation Metrics

Experimental Setup. We measure the compilation metrics described below by running each benchmark for 12 minutes. On our platform, 12 minutes is typically sufficient time for the VM to warm up and then perform multiple iterations with peak performance. We manually verified that the tested benchmarks warmed up in the first 6 minutes of the run by plotting the iteration durations. For each benchmark, we repeat 30 VM runs of the baseline setup and the same number of VM runs with the hybrid setup that uses the cached IR for selected compilations. The client VM with hybrid compilation connects to a server running either on the same machine or a remote machine. In the *co-located* setup, the server runs on the same machine. In the *remote* setup, the compilation server runs on a dedicated machine, and all client VMs connect through the local network. Since the server is a Java application, we warm it up[1] with three unmeasured training and replay runs. Before each measured run of the client VM, we populate the IR cache of the compilation server with a single 12-minute training run of the same benchmark. To capture the potential variability between training runs, we do not reuse one training run for multiple replay runs.

Metrics. We collect the metrics reported by the JVM using the option `-XX:+CITime`. *Compilation time* is the sum of the wall-clock durations of all compilation tasks. Therefore, for the client VM with hybrid compilation, the compilation time metric includes the time spent on communication and serialization, as well as any failed attempts to reuse the cached IR. *Compiled bytecodes* is the total size of the bytecodes compiled in every compilation task, comprising the bytecode size of the root method and inlinees. *Compiled methods* is the total number of completed compilation tasks. Although we cache the IR only for the top-tier GraalVM compiler and not the C1 compiler [20], we report the sums of values for both compilers to present a complete picture. For multiple benchmark runs in the same setup, we report the arithmetic average of each metric value.

Statistical Evaluation. To ensure that the reported averages are not excessively impacted by random fluctuations, we used bootstrapping to compute 99% confidence intervals for all relative compilation and performance metrics reported in the paper. We did this by resampling with replacement from the VM runs of every benchmark in a particular setup. Because the confidence intervals are narrow relative to the reported effect size for every metric except peak performance, where the change is insignificant, we omit them from the presentation to enhance readability.

[1] We could avoid the server's warmup by compiling it ahead of time, similarly to how the GraalVM compiler is built.

Table 1 Compilation metrics in DaCapo benchmarks with a co-located server (lower is better).

benchmark	compilation time		compiled bytecodes		compiled methods	
	baseline	hybrid	baseline	hybrid	baseline	hybrid
eclipse	155.68 s	**−28.6**%	16.25 MB	**−9.5**%	30342	**−5.1**%
pmd	115.08 s	**−42.8**%	8.52 MB	**−13.2**%	18188	**−5.0**%
fop	95.19 s	**−33.6**%	8.83 MB	**−4.1**%	20191	**−3.0**%
spring	85.38 s	**−17.9**%	9.71 MB	**−3.7**%	25970	**−1.7**%
tomcat	72.84 s	**−15.2**%	9.82 MB	**−7.4**%	25230	**−3.3**%
cassandra	61.71 s	**−18.6**%	6.34 MB	**−9.6**%	19851	**−2.9**%
jython	56.75 s	**−38.4**%	6.24 MB	**−5.7**%	9591	**−6.0**%
zxing	48.87 s	**−59.3**%	3.01 MB	**−21.3**%	5705	**−7.6**%
kafka	46.88 s	**−27.9**%	5.40 MB	**−9.5**%	18418	**−2.9**%
h2	40.83 s	**−33.0**%	3.19 MB	**−21.7**%	5749	**−8.2**%
batik	38.91 s	**−20.4**%	2.45 MB	**−10.8**%	6302	**−3.1**%
xalan	18.76 s	**−28.2**%	1.69 MB	**−3.4**%	4521	**−0.6**%
luindex	18.31 s	**−33.8**%	2.23 MB	**−8.2**%	5950	**−2.9**%
lusearch	16.12 s	**−45.1**%	1.63 MB	**−14.9**%	4160	**−2.8**%
jme	12.43 s	**−40.2**%	1.43 MB	**−11.3**%	4480	**−2.1**%
sunflow	9.24 s	**−53.2**%	0.89 MB	**−18.4**%	2277	**−3.8**%
graphchi	9.04 s	**−43.2**%	0.65 MB	**−17.0**%	2389	**−2.6**%
biojava	8.75 s	**−51.4**%	0.82 MB	**−13.3**%	2553	**−3.7**%
avrora	8.30 s	**−45.0**%	0.88 MB	**−19.9**%	3209	**−2.4**%

Table 2 Changes in compilation metrics for every benchmark suite (lower is better).

benchmark suite	server setup	compilation time	compiled bytecodes	compiled methods
DaCapo	co-located	**−36.8**%	**−11.9**%	**−3.7**%
Renaissance	co-located	**−23.6**%	**−8.6**%	**−3.5**%
ScalaBench	co-located	**−35.9**%	**−9.8**%	**−4.3**%
DaCapo	remote	**−31.8**%	**−11.0**%	**−3.3**%

Results. Table 1 lists the compilation metrics for the individual DaCapo benchmarks with a co-located compilation server. The columns marked *baseline* show the arithmetic average of a metric value for all baseline runs of the given benchmark. The columns marked *hybrid* show the change in the arithmetic average of a metric value for the hybrid runs relative to the baseline. Hybrid compilation significantly reduces compilation time for every benchmark from 15.2% up to 59.3%. The size of the compiled bytecodes decreases by 3.4% to 21.7%, and the number of compiled methods decreases by up to 8.2%.

Table 2 summarizes the changes in compilation metrics in every evaluated suite. We aggregate the metric values for a benchmark suite as the geometric mean of the relative metric values of the individual benchmarks. In the setup with a co-located compilation server, the overall decrease in compilation time ranges between 23.6% and 36.8%, the size of the compiled bytes decreases by between 8.6% and 11.9%, and the number of compiled methods decreases by between 3.5% and 4.3%. The overall improvements in the DaCapo suite with a remote server are substantial but slightly lower than those in the co-located setup.

Discussion. The drop in compilation time is large in all benchmark suites, which is a result of multiple factors. The cached IR directly decreases compilation time by reducing the amount of work performed. As an indirect effect, faster compilations can lead to fewer compilation tasks being scheduled. This is because compilation tasks for the JIT compilers are scheduled when method invocation and loop back-edge counters overflow. By speeding up the compilation tasks, the compiled methods (including inlined methods) are executed via the compiled code that does not increment the counters. Consequently, these counters have fewer opportunities to overflow, and fewer compilation tasks may be scheduled. The data shows that both the total size of the compiled methods and their number decreased significantly, which is evidence of less work scheduled for the JIT compilers.

Another indirect effect that can save compilation time is the higher maturity of the cached IR, i.e., the compilations may have a lower chance of deoptimizing [16] on an assumption or speculation [11] and requiring a recompilation. This is because the cache stores the last completed compilation of a method from the training run, and, potentially, the IR may not contain speculations or assumptions that are bound to be violated later during the VM run.

During development, we found that our prototype can sometimes suffer from an excessive number of recompilations. This happens when the compiler deoptimizes from a compiled method to an earlier block in the control-flow graph, and the VM executes the bytecode instruction that triggered the deoptimization as part of a more mature compiled method rather than in the interpreter. Consequently, the VM fails to update the profiles for future recompilations. Our prototype can trigger this problem due to the presence of compiled code of varying maturity (from the IR cache and locally compiled). Changing how speculations are tracked in the IR [11] could improve the results, and we reported the issue to the GraalVM compiler developers. A few runs of the DaCapo `luindex` benchmark in the remote setup seem to suffer from excessive recompilation, inflating the difference between the co-located and remote setups. Without `luindex`, the compilation time savings in the remote setup are 33.6% instead of 31.8%.

The improvements vary between the benchmarks because multiple variables determine the success of our approach. For example, some benchmarks use code patterns that lead to non-reusable code, which we analyze later. In some workloads, the VM often compiles different methods across two VM runs, leading to cache misses. Lastly, compilation units differ in their size, and IR caching may benefit larger compilations more.

The difference between the co-located and remote setups can likely be attributed to the communication latency. A possible mitigation for this problem is hiding the latency by increasing the number of concurrent compilation tasks [18, 19].

5.2 Performance Metrics

We use the measurement data collected from the experiment introduced above to analyze how our approach impacts peak performance and warmup time.

Peak Performance. The quality of the code compiled by a dynamic compiler is sensitive to factors such as the timing of compilation tasks. Therefore, we consider it a good practice to confirm that performance is unaffected. We measure the peak performance in terms of the average iteration time when the VM is warmed up. We acknowledge that the workloads may not reach a single steady state [2], and variation in iteration time is expected. As a pragmatic choice, we compute the arithmetic average of the durations of iterations completed after the 6-minute mark since VM start (lower is better). Note that the workloads typically warm up in the first 6 minutes of the run on our platform.

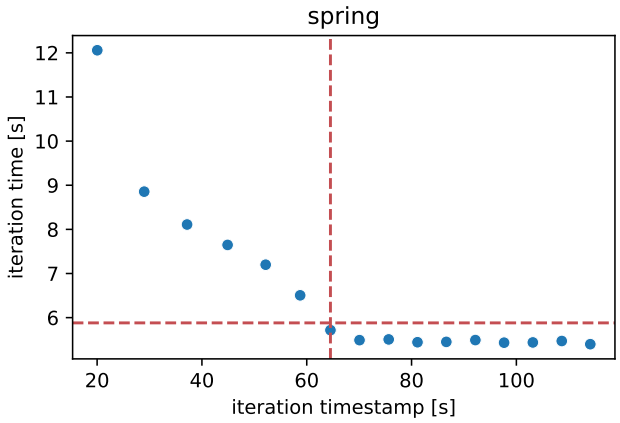


Figure 3 Iterations from the beginning of a baseline run of the DaCapo `spring` benchmark, highlighting the warmup threshold (horizontal line) and the warmup time (vertical line).

Warmup Time. There are multiple options for precisely defining warmup, but there is no standard and straightforward option. Related work [19] uses 90% of peak throughput as the threshold to consider an application warmed up. Since we evaluate benchmarks consisting of multiple workload iterations, we define warmup time as the timestamp of the first iteration where the iteration time is below a warmup threshold. We compute the warmup threshold analogously to the related work [19] as 1/(90%) of the mean iteration time after the 6-minute mark (the peak performance metric). We performed a sensitivity analysis to verify that the overall results vary by no more than a few percent with values of the warmup threshold parameter between 1/(80%) and 1/(95%). Due to performance anomalies that we explain below, we exclude the `scala-kmeans` and `gauss-mix` benchmarks from the warmup time evaluation in the Renaissance suite.

To visualize our warmup detection method, Figure 3 plots the iteration timestamps and durations of the DaCapo `spring` benchmark from the start of a single run. The horizontal line is the computed warmup threshold. The vertical line marks the computed warmup time.

Benchmarks with high iteration times present a potential problem for our and similar warmup detection methods. Because we determine warmup only after an iteration completes, we risk overestimating small changes in warmup time that occur within an iteration. This concern is reduced in benchmarks with low iteration times. We verified that restricting the evaluation to 25% of the benchmarks with the lowest baseline iteration time yields similar overall results.

We encountered these performance anomalies that are relevant to warmup detection:

1. high variance in iteration time within a VM run (in DaCapo `xalan` and other benchmarks),
2. the application experiences a small but continuous speedup (in DaCapo `fop` and other benchmarks),
3. slowing down after the first iteration (in DaCapo `kafka`), and
4. large shifts in performance (in Renaissance `scala-kmeans` and `gauss-mix`).

These anomalies occur in both the baseline and hybrid setups. We manually inspected plots similar to Figure 3 to verify that the problems caused by the first three anomalies are mitigated by the warmup threshold. In DaCapo `kafka`, the first iteration is already below the warmup threshold, so the warmup detection yields reasonable results. The performance shifts in Renaissance `scala-kmeans` and `gauss-mix` cause significant variance in the measured

Table 3 Performance metrics in DaCapo benchmarks with a co-located server (lower is better).

benchmark	warmup time		iteration time	
	baseline	hybrid	baseline	hybrid
cassandra	104.69 s	**−15.2**%	11.73 s	**−0.2**%
pmd	84.15 s	**−45.1**%	1.99 s	**+0.8**%
eclipse	76.03 s	**−24.6**%	12.43 s	**+0.2**%
spring	64.23 s	**−17.7**%	5.27 s	**+0.1**%
jython	63.63 s	**−5.6**%	2.99 s	**−1.0**%
fop	54.69 s	**−37.5**%	0.48 s	**−4.7**%
h2	45.36 s	**−7.0**%	8.46 s	**−1.0**%
tomcat	35.75 s	**−0.3**%	30.24 s	**−0.1**%
luindex	32.12 s	**−16.6**%	4.50 s	**−0.1**%
zxing	30.85 s	**−60.5**%	4.04 s	**+0.5**%
lusearch	25.55 s	**−6.9**%	9.17 s	**−0.6**%
graphchi	23.76 s	**−0.7**%	5.29 s	**−0.2**%
sunflow	22.54 s	**−7.3**%	8.96 s	**+1.5**%
biojava	17.77 s	**−4.2**%	6.33 s	**+0.2**%
xalan	16.50 s	**−23.1**%	2.63 s	**−0.3**%
jme	16.11 s	**−1.0**%	6.88 s	**+0.1**%
kafka	15.19 s	**−0.7**%	14.20 s	**−1.4**%
batik	13.45 s	**−21.0**%	1.54 s	**+0.4**%
avrora	6.15 s	**+4.0**%	4.93 s	**−0.3**%

Table 4 Changes in performance metrics for every benchmark suite (lower is better).

benchmark suite	server setup	warmup time	iteration time
DaCapo	co-located	**−17.4**%	**−0.3**%
Renaissance	co-located	**−13.1**%	**−0.2**%
ScalaBench	co-located	**−21.2**%	**0.0**%
DaCapo	remote	**−17.3**%	**−0.2**%

warmup time. In multiple runs, warmup is detected at the point of a performance drop occurring several minutes into the run, which can substantially impact the average warmup time. The performance shifts may be caused by inlining instability [36], which is a known compiler issue that manifests in these benchmarks. For the above reasons, we consider these benchmarks unsuitable for comparing warmup time between two systems, and we exclude them when computing the overall warmup change in the Renaissance suite. The DaCapo and ScalaBench suites do not experience performance shifts of this nature.

Results. Table 3 lists the performance metrics for the individual DaCapo benchmarks with a co-located compilation server. As before, we show the absolute values for the baseline runs and the relative values for the hybrid runs. Our approach improves the warmup time for almost all DaCapo benchmarks. The peak performance of the hybrid runs is usually close to the baseline, with few minor deviations in both directions.

Table 4 summarizes the changes in performance metrics for every evaluated setup. In the setup with a co-located server, the overall decrease in warmup time is between 13.1% and 21.2%. The iteration time of the hybrid setup does not significantly deviate from the baseline. The overall warmup improvement in the DaCapo suite with a remote server is similar to that of the co-located setup.

Discussion. Our approach significantly improves warmup time in most workloads, which we likely achieve primarily by reducing the latency of compilation tasks. When compilation latency is reduced, the VM can start executing efficient code earlier. However, due to the nature of dynamic compilation, compilation time reductions do not transfer linearly to warmup improvements. Just by changing the latencies of compilation tasks, we also impact the collected profiles, the methods selected for compilation, and, consequently, how the compiler optimizes code. Lastly, compilation speed is only one of the many factors that influence warmup.

The presented approach aims not to degrade peak performance. In the vast majority of benchmarks, there is no significant impact, and the overall change is also small. However, in individual benchmarks, our approach can change when a particular method is compiled (compared to the baseline), which in turn impacts profiles and optimization. In the Renaissance suite, a small number of workloads are susceptible to performance changes in either direction. These changes are likely caused by inlining instability [36].

5.3 Analysis of IR Reusability

We evaluate the success of our prototype in reusing the cached IR based on the relative size of the bytecode compiled from the cache, which we refer to as the cache hit rate. We also analyze the reasons for the observed validation failures and discuss the options for improving IR reusability in future work.

Cache Hit Rate. In every 12-minute run of the hybrid setup, we compute the cache hit rate as the ratio of the bytecode size of the compilation tasks served from the IR cache and the total bytecode size of all compilation tasks. We use the geometric mean to aggregate the metric.

In the co-located setup, the cache hit rate is about 69% in the DaCapo suite, 64% in the Renaissance suite, and 77% in the ScalaBench suite. In the DaCapo suite with a remote server, the cache hit rate is 71%.

The results show that the majority of the compiled bytecodes originate from the IR cache. The client VMs never reach a cache hit rate of 100% in any of the runs since we deliberately try to compile small methods using the local JIT compiler to keep compilation latency low. The other factors that decrease the cache hit rate are cache misses, recompilations, and validation failures.

Transient Validation Failures. The evaluated benchmarks experience a varying number of validation failures. *Transient failures* are the most common type of validation failures, comprising between 89% and 96% of the validation failures in every evaluated benchmark suite. As explained before, they are caused by the cached IR referencing a class that is not yet loaded by the target VM or referencing a class, method, or field that is in an earlier initialization stage than required to reuse the IR. These failures occur in almost every benchmark because we attempt to reuse mature IR as early as possible. The average number of transient failures varies between benchmarks. DaCapo `eclipse` experiences these failures most frequently, averaging more than 250 failures per run.

Permanent Validation Failures. All other types of validation failures are permanent and together comprise between 4% to 11% of the observed failures in every benchmark suite. The vast majority of Renaissance workloads are affected by permanent failures. In contrast, about half of the DaCapo benchmarks and the vast majority of ScalaBench benchmarks do not experience permanent failures at all.

Class hierarchy mismatch is a permanent failure that occurs when a class hierarchy query on the target VM returns an incompatible result. For example, it occurs when an interface has a single loaded implementation during the tracing stage but multiple loaded implementations during the replay stage. This kind of failure occurs in most Renaissance benchmarks and in about a third of DaCapo benchmarks, but only a low number of compilation tasks are affected on average: no DaCapo benchmark experiences more than 4 of these failures per run.

The *field value mismatch* permanent validation failure occurs when the compiler folds a read of a static final field and the value differs from the expected one. The failure occurs in most Renaissance benchmarks and in about a third of the DaCapo benchmarks. The `cassandra` benchmark is the most affected in the DaCapo suite, experiencing about 13 such failures on average.

Lastly, *type definition mismatch* and *method definition mismatch* are permanent failures occurring due to an incompatible definition of a class or a method. For example, an incompatible class may have different fields, and an incompatible method may have different bytecodes. These failures occur frequently in only a few of the benchmarks. In the Renaissance `dec-tree` benchmark, these incompatibilities are caused by the classes generated by the Catalyst Optimizer [13] from the Apache Spark framework. As we explained previously, our prototype cannot distinguish between the multiple generated classes in this case since they all share the same class name.

Mitigating Validation Failures. We invested a substantial engineering effort to minimize the number of validation failures, and there are still options for further improvement. One of the problems with our prototype is recording the compilation trace also for the expanded inlining candidate methods [38], even if these candidates are not ultimately inlined. There is an opportunity to avoid the validation failures caused by the inlining candidates, which we could achieve by filtering out the operations irrelevant to the inlined methods.

We could mitigate many of the transient validation failures by loading pre-initialized classes [47], which is a planned but so far unrealized feature of the OpenJDK project Leyden [15]. The client VM could trigger the loading of a pre-initialized class when the class is first referenced in a compilation trace. Pre-initialization requires caching [48] the values of the static final fields, which would also mitigate the failures caused by field value mismatch. Together, these features would open up the option for eagerly compiling [18] the cached IR and prefetching the code compiled at the highest optimization level.

6 Related Work

Reusing Compilation Results. Shared Class Cache (SCC) [17] is a technology to improve startup and warmup performance in the OpenJ9 JVM. SCC stores preprocessed class data in a memory-mapped file, and it can also store compiled code. Reusability of the code is achieved by compiling it with a lower optimization level [19] and patching the compiled machine code [8]. Due to the lower performance of the cached code, the JIT compiler must recompile the hottest methods to reach peak throughput. SCC uses a process similar to

compilation tracing to establish the provenance of VM objects [9], such as methods and classes. However, the mechanism is not used to facilitate specializing the code for the running VM instance. In our approach, we leverage compilation tracing almost without any restrictions on the optimizations that can be applied. Thus, the code compiled from our IR cache achieves peak performance and does not require recompilation. ShareJIT [46] and ShMVM [7] are systems similar to SCC that facilitate sharing code compiled with reduced optimization level on a single machine. `jaotc` [21, 22] was a tool to precompile Java class files ahead of time to native code for the HotSpot JVM.

Pečimúth et al. [37] explain the reasons why highly optimized code is not reusable in GraalVM. The authors suggest caching the IR and making it reusable by deferring all optimizations that potentially specialize the IR to the running VM. However, this approach reduces peak performance in many workloads. In our work, we instead let the compiler perform the optimizations that specialize the IR to the VM, and we employ compilation tracing to facilitate IR reuse without sacrificing peak performance.

The Truffle language implementation framework [45] can store the compiled code along with other data structures in a file [31] to improve warmup in future VM runs. The guest language running in this mode (e.g., JavaScript) must satisfy specific conditions. For example, the language implementation cannot speculate on object identity, since it would lead to a guaranteed deoptimization. The approach we present does not have this restriction because replaying the compilation trace allows us to relocate object references in the IR.

Ř+ [26] is a system to reuse compiled code for the R language. In Ř+, the compiler tracks its assumptions and stores them along the compiled code. The code repository may contain multiple versions of the same function, each fit to a different context. At run time, calls are dispatched to the appropriate version based on the current context. Although the HotSpot JVM does not perform such contextual dispatch, our IR cache could potentially benefit from storing multiple versions of a method. For example, the most mature version of a method is sometimes not usable early in the application's lifetime due to assumptions about the loaded classes.

Remote Compilation. JITServer [19, 18] is a remote JIT compilation server for the OpenJ9 JVM. The design of our compilation server, with callbacks for VM runtime information and the caching of their results, is inspired by JITServer. JITServer leverages compiled-code caching based on OpenJ9's SCC [17] infrastructure. Therefore, the code loaded from the cache requires recompilation at the highest optimization level. The main use case presented by the authors is remote compilation for resource-constrained containers in cloud data centers. Resource-constrained containers benefit from the offloading of compilation tasks to a remote server and achieve warmup improvements even without requiring code caching. The compilation server running on another, non-constrained machine effectively extends the computing resources available to the VM running in a container. JITServer can hide the latency of the offloaded compilation tasks by issuing more concurrent compilation tasks. It can also trigger remote compilation eagerly and prefetch [18] the compiled code to the client VMs.

In contrast, our work focuses on reusing the IR compiled at the highest optimization level without needing recompilation to reach peak performance. We decrease the compilation latency by reusing the cached IR. We present an evaluation where the compilation server is co-located with the client without increasing the number of concurrent compilation tasks or dedicating more computing resources to JIT compilation.

Cloud Native Compiler [1] is a production-grade compilation server that can reuse profiles from past JVM runs. However, there is little technical information available. The compilation server for Jikes RVM by Lee et al. [24] is an earlier work in which compilation tasks are served in a single request without callbacks. However, this is difficult to achieve in a modern JVM due to the dynamicity of the runtime and method inlining.

Improving Warmup. OpenJDK project Leyden [15] is a comprehensive effort addressing the startup and warmup phase of Java applications via various techniques, such as storing compiled code in class-data archives [42], ahead-of-time loading of classes in the linked [23] or initialized [47] state, and other techniques that may require new language features. OpenJDK project CRaC [30] mitigates warmup time by checkpointing running Java applications (creating snapshots), which can then be restored. However, the checkpointed application must manage resources such as open files and sockets, so the process is not transparent to the programmer. Jump-Start [34] is a technique that improves warmup by sharing profiling information for the HipHop VM, which eliminates the overhead of collecting profiles.

Ahead-of-Time (AOT) Compilation. Native Image [32] builds native executables by compiling Java class files ahead of time with the closed-world assumption (i.e., classes cannot be loaded at run time). AOT compilation can eliminate warmup but may impact peak performance. The performance of dynamic languages like JavaScript often depends on speculation and runtime feedback, so AOT compilers [43, 33] usually cannot produce as efficient code as JIT compilers.

7 Conclusion

This paper introduces a novel technique to reuse highly optimized IR in dynamic compilers. We do this by tracing the arguments and results of compiler-interface calls, allowing us to verify that the assumptions made during a compilation of reusable IR hold in a particular target VM instance. Using this approach, we can relocate the IR's VM object references and compile the IR for the target VM instance. A compilation server caches the IR from previous runs of the application and can compile it to machine code for local or remote client VMs. An empirical evaluation based on the GraalVM compiler and industry-standard benchmark suites shows that the approach reduces compilation time by 23.6% to 36.8% and warmup time by 13.1% to 21.2% in each suite without hindering peak performance.

References

1 Azul. Cloud Native Compiler, 2024. Retrieved August 14, 2024. URL: `https://docs.azul.com/optimizer-hub/about/cloud-native-compiler`.

2 Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual Machine Warmup Blows Hot and Cold. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), October 2017. `doi:10.1145/3133876`.

3 Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. *SIGPLAN Not.*, 41(10):169–190, October 2006. `doi:10.1145/1167515.1167488`.

4 Josh Bloch and Doug Lea. ThreadLocal.java in OpenJDK, 2023. Retrieved September 2, 2024. URL: https://github.com/openjdk/jdk/blob/fe9f05023e5a916b21e2db72fa5b1e8368a2c07d/src/java.base/share/classes/java/lang/ThreadLocal.java.

5 Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. From warm to hot starts: leveraging runtimes for the serverless era. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 58–64, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3458336.3465305.

6 Mandy Chung. JEP 371: Hidden Classes, 2020. Retrieved February 20, 2025. URL: https://openjdk.org/jeps/371.

7 Grzegorz Czajkowski, Laurent Daynès, and Nathaniel Nystrom. Code sharing among virtual machines. In Boris Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming*, pages 155–177, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. doi:10.1007/3-540-47993-7_7.

8 Irwin D'Souza. Ahead Of Time Compilation: Relocation, 2018. Retrieved August 22, 2024. URL: https://blog.openj9.org/2018/10/26/ahead-of-time-compilation-relocation/.

9 Irwin D'Souza. Ahead Of Time Compilation: Validation, 2018. Retrieved August 22, 2024. URL: https://blog.openj9.org/2018/11/08/ahead-of-time-compilation-validation/.

10 Gilles Duboscq, Lukas Stadler, Thomas Wuerthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, February 2013. URL: http://ssw.jku.at/General/Staff/GD/APPLC-2013-paper_12.pdf.

11 Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, pages 1–10, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2542142.2542143.

12 Stephen J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '03, pages 241–252, USA, 2003. IEEE Computer Society. doi:10.1109/CGO.2003.1191549.

13 Apache Software Foundation. CodeGenerator.scala in Apache Spark, 2025. Retrieved February 20, 2025. URL: https://github.com/apache/spark/blob/bbb9c2c1878e200d9012d2322d979ae794b1d41d/sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/expressions/codegen/CodeGenerator.scala#L1516.

14 Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1297027.1297033.

15 Dan Heidinga. Choose your own performance; a Project Leyden update, 2024. Retrieved August 23, 2024. URL: https://openjdk.org/projects/leyden/slides/leyden-heidinga-devnexus-2024-03.pdf.

16 Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. Association for Computing Machinery. doi:10.1145/143095.143114.

17 IBM. Introduction to class data sharing, 2024. Retrieved August 14, 2024. URL: https://eclipse.dev/openj9/docs/shrc/.

18 Alexey Khrabrov. *Disaggregated Just-in-Time Compilation for the Java Virtual Machine*. PhD thesis, University of Toronto, June 2024. URL: http://hdl.handle.net/1807/140205.

19 Alexey Khrabrov, Marius Pirvu, Vijay Sundaresan, and Eyal de Lara. JITServer: Disaggregated Caching JIT Compiler for the JVM in the Cloud. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 869–884, Carlsbad, CA, July 2022. USENIX Association. URL: https://www.usenix.org/conference/atc22/presentation/khrabrov.

20 Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1), May 2008. doi:10.1145/1369396.1370017.

21 Vladimir Kozlov. JEP 295: Ahead-of-Time Compilation, 2018. Retrieved September 2, 2024. URL: https://openjdk.org/jeps/295.

22 Vladimir Kozlov. JEP 410: Remove the Experimental AOT and JIT Compiler, 2021. Retrieved September 2, 2024. URL: https://openjdk.org/jeps/410.

23 Ioi Lam, Dan Heidinga, and John Rose. JEP 483: Ahead-of-Time Class Loading & Linking, 2024. Retrieved August 27, 2024. URL: https://openjdk.org/jeps/483.

24 Han B. Lee, Amer Diwan, and J. Eliot B. Moss. Design, implementation, and evaluation of a compilation server. *ACM Transactions on Programming Languages and Systems*, 29(4):18–es, August 2007. doi:10.1145/1255450.1255451.

25 Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. *The Java® Virtual Machine Specification: Initialization*, chapter 5.5. Oracle, Java SE 22 edition, 2024. URL: https://docs.oracle.com/javase/specs/jvms/se22/html/jvms-5.html#jvms-5.5.

26 Meetesh Kalpesh Mehta, Sebastián Krynski, Hugo Musso Gualandi, Manas Thakur, and Jan Vitek. Reusing just-in-time compiled code. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2), October 2023. doi:10.1145/3622839.

27 Oracle. LambdaMetafactory.java in OpenJDK, 2021. Retrieved March 2, 2025. URL: https://github.com/openjdk/jdk/blob/fe9f05023e5a916b21e2db72fa5b1e8368a2c07d/src/java.base/share/classes/java/lang/invoke/LambdaMetafactory.java.

28 Oracle. GraalVM Overview, 2024. Retrieved August 14, 2024. URL: https://www.graalvm.org/latest/docs/introduction/.

29 Oracle. HotSpot Internals, 2024. Retrieved August 14, 2024. URL: https://wiki.openjdk.org/display/HotSpot.

30 Oracle. Project CRaC, 2024. Retrieved August 14, 2024. URL: https://openjdk.org/projects/crac.

31 Oracle. Auxiliary Engine Caching, 2025. Retrieved February 26, 2025. URL: https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/AuxiliaryEngineCachingEnterprise/.

32 Oracle. Native Image, 2025. Retrieved February 26, 2025. URL: https://www.graalvm.org/latest/reference-manual/native-image/.

33 Oracle. Truffle AOT Overview , 2025. Retrieved February 26, 2025. URL: https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/AOTOverview/.

34 Guilherme Ottoni and Bin Liu. HHVM jump-start: boosting both warmup and steady-state performance at scale. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, pages 340–350. IEEE Press, 2021. doi:10.1109/CGO51591.2021.9370314.

35 Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot™ server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, page 1, USA, 2001. USENIX Association.

36 Andrej Pečimúth, David Leopoldseder, and Petr Tůma. Diagnosing Compiler Performance by Comparing Optimization Decisions. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2023, pages 47–61, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3617651.3622994.

37 Andrej Pečimúth, David Leopoldseder, and Petr Tůma. An Analysis of Compiled Code Reusability in Dynamic Compilation. In *Proceedings of the 16th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL '24, pages 43–53, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3689490.3690406.

38 Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Würthinger. An optimization-driven incremental inline substitution algorithm for just-in-time compilers. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 164–179. IEEE Press, 2019. doi:10.1109/CGO.2019.8661171.

39 Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proc. 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 17, 2019. doi:10.1145/3314221.3314637.

40 John Rose. JEP 243: Java-Level JVM Compiler Interface, 2019. Retrieved February 27, 2025. URL: https://openjdk.org/jeps/243.

41 John Rose. ClassValue.java in OpenJDK, 2024. Retrieved September 2, 2024. URL: https://github.com/openjdk/jdk/blob/b1163bcc88a5b88b9a56d5584310f1d679690ab2/src/java.base/share/classes/java/lang/ClassValue.java.

42 John Rose. JEP draft: Ahead-of-Time Code Compilation, 2025. Retrieved February 25, 2025. URL: https://openjdk.org/jeps/8335368.

43 Manuel Serrano. JavaScript AOT compilation. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2018, pages 50–63, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3276945.3276950.

44 Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da Capo con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *Proceedings of the 26th Conference on Object-Oriented Programming, Systems, Languages and Applications*, OOPSLA '11, pages 657–676, New York, NY, USA, 2011. ACM. doi:10.1145/2076021.2048118.

45 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2509578.2509581.

46 Xiaoran Xu, Keith Cooper, Jacob Brock, Yan Zhang, and Handong Ye. ShareJIT: JIT Code Cache Sharing across Processes and Its Practical Implementation. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), October 2018. doi:10.1145/3276494.

47 Jiangli Zhou. Java Class Pre-resolution and Pre-initialization, 2020. Retrieved February 27, 2025. URL: https://cr.openjdk.org/~jiangli/Leyden/Java%20Class%20Pre-resolution%20and%20Pre-initialization%20(OpenJDK).pdf.

48 Jiangli Zhou and Thomas Schatzl. Caching Java Heap Objects, 2019. Retrieved February 20, 2025. URL: https://wiki.openjdk.org/display/HotSpot/Caching+Java+Heap+Objects.

Lightweight Diagramming for Lightweight Formal Methods: A Grounded Language Design

Siddhartha Prasad
Brown University, Providence, RI, USA

Ben Greenman
University of Utah, Salt Lake City, UT, USA

Tim Nelson
Brown University, Providence, RI, USA

Shriram Krishnamurthi
Brown University, Providence, RI, USA

Abstract

Tools such as Alloy enable users to incrementally define, explore, verify, and diagnose specifications for complex systems. A critical component of these tools is a visualizer that lets users graphically explore generated models. As we show, however, a default visualizer that knows nothing about the domain can be unhelpful and can even actively violate presentational and cognitive principles. At the other extreme, full-blown custom visualization requires significant effort as well as knowledge that a tool user might not possess. Custom visualizations can also exhibit bad (even silent) failures.

This paper charts a middle ground between the extremes of default and fully-customizable visualization. We capture *essential* domain information for *lightweight* diagramming, embodying this in a language. To identify key elements of lightweight diagrams, we ground the language design in both the cognitive science research on diagrams and in a corpus of 58 custom visualizations. We distill from these sources a small set of orthogonal primitives, and use the primitives to guide a diagramming language called Cope-and-Drag (`CnD`). We evaluate it on sample tasks, three user studies, and performance, and find that short `CnD` specifications consistently improve model comprehension over the Alloy default. `CnD` thus defines a new point in the design space of diagramming: a language that is lightweight, effective, and driven by sound principles.

2012 ACM Subject Classification Human-centered computing → Visualization toolkits; Software and its engineering → Constraint and logic languages; Software and its engineering → Specification languages

Keywords and phrases formal methods, diagramming, visualization, language design

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.26

Related Version *Previous Version*: `https://arxiv.org/abs/2412.03310v1`

Supplementary Material *Software (Source Code)*: `https://github.com/sidprasad/copeanddrag/tree/ecoop-25` [38], archived at `swh:1:dir:b92dbb01ca5fcbe40ac64c8026fb3076fbf85d2a`
Software (Interactive Diagrams): `https://doi.org/10.5281/zenodo.15278550`
Software (ECOOP 2025 Artifact Evaluation approved artifact): `https://doi.org/10.4230/DARTS.11.2.12`

Funding This work is partially supported by US NSF grant DGE-2208731.

Acknowledgements We thank Tristan Dyer for sharing his insights on Sterling. We are also deeply grateful to Rob Goldstone for gently educating us about cognitive science. We thank Skyler Austen and Ji Won Chung for their feedback on study design and supplementary material.

© Siddhartha Prasad, Ben Greenman, Tim Nelson, and Shriram Krishnamurthi;
licensed under Creative Commons License CC-BY 4.0

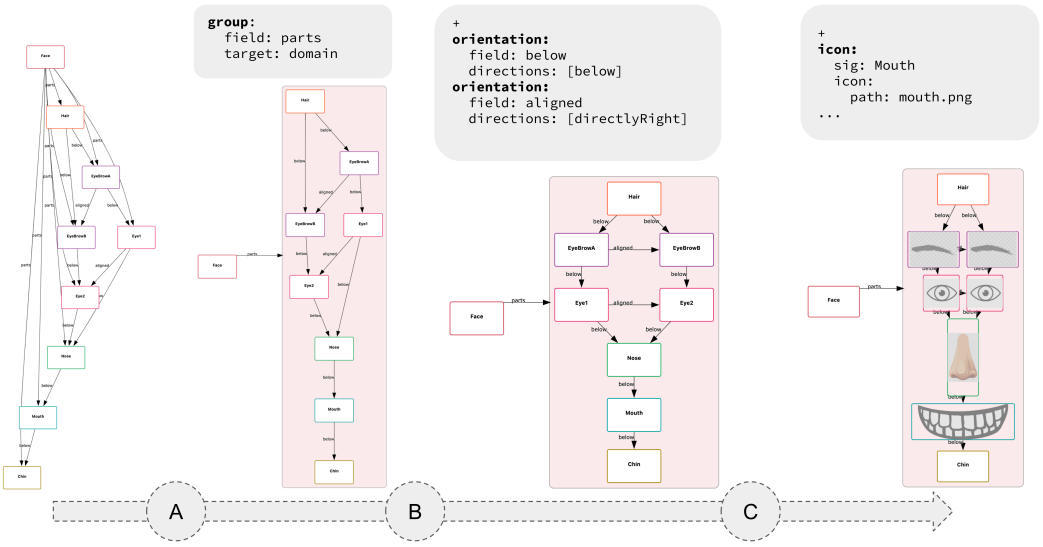


Figure 1 Cope-and-Drag is a language built to support lightweight diagramming for lightweight formal methods. A small set of orthogonal primitives allows users to incrementally refine model-finder output into more effective diagrams. This figure shows how to incrementally refine an instance describing a face using (A) grouping, (B) orientation, and (C) icons.

1 Introduction

In *lightweight formal methods*, proposed by Jackson and Wing [15] and realized in tools such as Alloy [16], users build formal models iteratively, exploring the design space as they go.

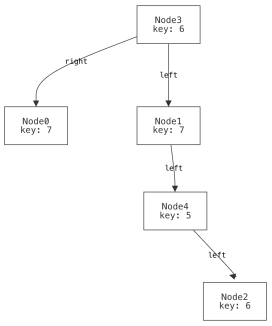
Alloy specs consist of sig[nature]s (sets of atoms), fields (relations between sigs), and descriptions of system behavior. Whereas traditional verification tools are inert without properties, Alloy users can begin reasoning with *just* system descriptions. Alloy uses a (SAT) solver to automatically generate instances that satisfy the spec, presenting them most commonly visually (e.g., Figure 2). This makes using Alloy as close an analog to programming as most formal methods tools get. Unlike a user of a proof assistant (where one must at least somewhat manually construct a proof) or model checker (which requires properties to verify), an Alloy user can enter a spec (akin to a program) and just click "Execute", and see a sequence of examples. This modality, in fact, goes beyond that of a typical programming language, as the user doesn't need to provide specific inputs to see output.

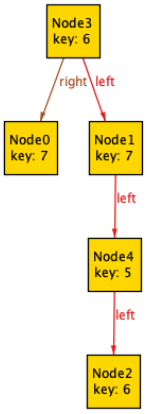
This immediacy and ability to work in the absence of properties has two benefits. First, users can spot ways in which the output does not conform to their intent and improve their specification. Second, seeing these violations suggests desirable and undesirable *properties*, which can then be formalized and verified – addressing the important and often neglected question of where properties come from. Thus, realizing lightweightness hinges heavily on the quality of visualizations.

Alloy and its recent sibling tool Forge [32] visualize relations as graphs. Alloy comes with a default GUI tool [16]. Forge uses Sterling [11] to render interactive graphs in a web browser (Sterling can visualize Alloy instances as well). Both the Alloy default and Sterling are generic, in that they do not embody any knowledge of the domain being modeled, which makes them largely indistinguishable from the perspective of our work: e.g., see Figure 2c and Figure 2b. While obtaining these visualizations requires zero effort from users, research shows that users struggle to make sense of them as models grow in size and complexity [24].

```
sig Node {
  // one key per Node
  key    : one Int,
  //At most one left/right child
  left   : lone Node,
  right  : lone Node
}

pred binary_tree {
  all disj n1, n2: Node | {
    n1 in n2.^(
     left + right + ~left + ~right)
  }
  all n: Node | {
    n not in n.^(left + right)
    some n.left => n.left != n.right
    lone parent: Node | {
      n in parent.(left+right)
   }
  }
}
run {binary_tree} for exactly 5 Node
```





(a) Alloy spec. **(b)** Sterling output. **(c)** Alloy Visualizer output.

Figure 2 Alloy spec of a binary tree, without any properties, alongside a generated instance (visualized by Alloy Visualizer). `Node` is a sig that defines a set of atoms, while fields `left`, `right`, and `key` describe relations between nodes. Predicate `binary_tree` imposes constraints on these fields.

Sterling also provides support for custom JavaScript visualizations. They can be narrowly crafted to a domain, and can be quite visually attractive (e.g., [32, Figure 11]; Figure 11b). By being programmable (in addition to being interactive), they can be applied repeatedly and consistently to all output. However, these custom visualizations suffer from a dual problem. As we discuss in Section 2.4, users must: know the details of drawing libraries, which are orthogonal to formal methods; expend a great deal of effort; and not inadvertently run afoul of visual design principles. Furthermore, custom visualizations suffer from a critical mode of *silent failure* when the specification and visualization disagree on model invariants. Silent failures, we argue, are especially dangerous for lightweight formal methods (Section 4).

In contrast, we want customization to not only be programmable but also: be lightweight; avoid complex library over-dependence; embody good visual design principles; and prevent failure modes. By embodying these in a programming language, we can be confident that *all* programs written in that language enjoy these desirable traits. Thus, this paper views the visualization task as a *language design problem*.

We therefore present a new language, Cope-and-Drag (`CnD`), for this purpose. `CnD` is driven both top-down, by cognitive science principles of visualization, and bottom-up, by manually distilling ideas from dozens of actual visualizations (Section 2). The resulting language is small, requires little annotation, and can be used incrementally (Section 3). `CnD` is designed to avoid the critical failure mode instead of silently rendering a misleading diagram (Section 4). We show through experiments that the features of the language result in visualizations that are more effective than the default visualizer (Section 5). `CnD` is embedded in an open-source tool that serves as a substitute visualizer for Alloy and Forge, available at `https://github.com/sidprasad/copeanddrag/tree/ecoop-2025`.

While this work falls in the realm of recent language designs for visual output, we show why this work is novel (Section 6) and particularly well-suited to the important domain of lightweight formal methods (Sections 3.3 and 4). Furthermore, this work or its extensions can also apply to other formal methods tools that employ diagramming (Section 6.3).

Vocabulary

- A **specification** (or, in the lightweight spirit, *spec*) is a formal description of a system's behavior (e.g., Alloy or Forge code).
- An **instance** is a concrete assignment of values to variables that satisfies the constraints and properties defined by the spec (e.g., a model generated by SAT, lifted to the domain).
- A **bad-instance** is an instance that fails to satisfy part of the *implicit* spec that the author is *trying* to write. A bad instance reflects a discrepancy between what the explicit spec currently represents and what the author intends it to represent. We use the colloquial word "bad" intentionally, since it is human judgment that identifies such a discrepancy. The discovery and extirpation of bad-instances while driving towards the implicit spec is a key part of the lightweight formal methods process.
- A **default-graph** is a directed-graph visualization produced by the Alloy Visualizer [16] or Sterling [11] (without customization). These graphs are agnostic to the domain being modeled. (While there are differences between the two tools, discussed in Section 6.1, for the purposes of this paper, Alloy and Sterling graphs are effectively the same, so we refer to them by the same general label.)
- A **triad** is a three-part figure that consists of default-graph output, `CnD` output, and the corresponding `CnD` source code. We use triads extensively in the supplement.

Supplemental Material

A powerful aspect of `CnD` diagrams is their interactive nature. While we include screenshots in this paper, we also provide interactive versions in supplementary material. Any diagram name that appears in a blue box, as in **name**, has an interactive variant available through a Docker image. Instructions for running the Docker are in the supplement readme (`supplement.pdf`). Instructions on accessing the exact survey instruments used in Section 5.2 are available in the supplement as well.

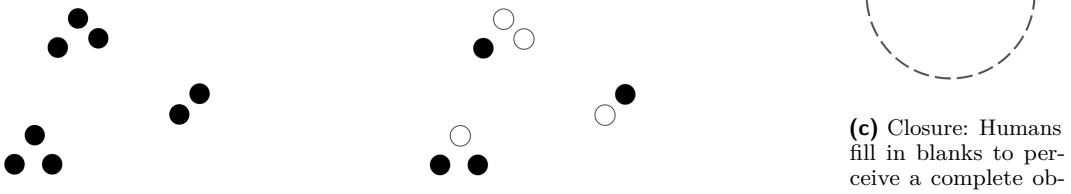
2 Diagramming Language Design

Languages can be designed from many perspectives, such as personal taste, carefully subjecting every feature to a user study, or programmer opinion, which can be broadly expressed or narrowly targeted. While these all have their validity, we focus on *grounded* design: designs that are derived from, and can be substantiated by, external artifacts.

There are roughly two ways to proceed. One is bottom-up: start with a large collection of desired artifacts, cluster their commonalities and differences, identify their principal dimensions, and use those to inform the design. Another is top-down: start with general principles established by experts (usually based on prior grounded efforts) and concretize them into operations. Because these approaches complement each other well, we use both methods. We start top-down, distilling key principles from the diagramming literature (Section 2.1), and then examine a corpus of concrete examples in the context of these principles (Section 2.2). We then align these perspectives to derive a set of diagramming primitives that are both theoretically grounded and practically applicable in the context of lightweight formal methods (Section 2.3).

2.1 Top Down

We begin our language design by drawing on established principles in cognitive science, visualization, and diagramming.



(a) Proximity: Our minds perceive a connection between objects in close spatial proximity.

(b) Similarity: Objects sharing the same color, shape, or size are perceived as related.

(c) Closure: Humans fill in blanks to perceive a complete object. While this diagram consists of a set of disconnected dashes, a viewer perceives a circle.

Figure 3 Examples of the Gestalt principles of proximity, similarity, and closure.

2.1.1 Visual Principles

A diagram is a visual representation that uses shapes and symbols to convey information or ideas. In this role as an information presentation tool, effective diagrams rely on perceptual properties common in information visualization, the most relevant (and best established) of which relate to pre-attentive processing and the Gestalt principles [5].

Pre-Attentive Processing. Pre-attentive processing is a cognitive function that allows humans to quickly detect certain visual features (e.g., color, shape, and size) of objects without conscious effort. This rapid processing can both help and hinder understanding.

The Stroop effect, for instance, demonstrates that the brain often automatically processes text, such as in word recognition, faster than stimuli such as color [43] and spatiality [49]. In the context of diagrams, this means that understanding is hindered when visual elements conflict with textual content (e.g., laying out a binary tree's `left` children to the right and `right` children to the left). Other common features that might cause undesirable pre-attentive processing include irrelevant visual elements [45], unnecessarily overlapping lines [39], or ambiguous labeling [5, 22]. Effective diagrams, therefore, use visual features to highlight important information while being mindful of extraneous cognitive load [44]. For example, Figure 5 leverages different colors to help viewers quickly distinguish between Alloy identifiers and `CnD` keywords.

Gestalt Principles. The Gestalt principles [17, 41, 48] describe how humans perceive and organize visual information, grouping objects that are close together (proximity), share attributes (similarity), form complete shapes (closure), follow a continuous path (continuity), stand out from a background (figure-ground), appear balanced (symmetry), or move together (common fate).[1] These principles play a crucial role in visual design, user interface design, and data visualization, helping to create more intuitive and easily comprehensible visual representations. Figure 3 illustrates some of these principles.

The Gestalt principles so accurately describe human perception, and leveraging pre-attentive processing is so inherently effective, that they both underpin many diagramming conventions, even when their influence is not explicitly acknowledged. For a more detailed description of how these principles apply to visualization, see [5, page 8].

[1] While these principles were originally formulated by Wertheimer in 1912 [48], we authors are more familiar with Spillman's English translation [41] and Koffka's framing in the field of psychology [17].

2.1.2 Diagramming Principles

While visualizations focus on displaying information, diagrams act as cognitive tools that facilitate deeper comprehension and analysis of complex systems [20, 22, 46]. A diagram's effectiveness stems not only from its visual appeal, but also from its ability to organize information in a way that facilitates efficient cognitive processes [20]. These processes depend on the task one is trying to perform.

Subway maps, for example, leverage the observation that semantic similarity is intuitively tied to spatial proximity [14]. By grouping together stations that are functionally related (e.g., transfer points or lines serving the same neighborhoods), they serve as effective diagrams for planning public transportation routes. However, this layout is poorly suited for understanding the actual geographic distances between stations, as the map distorts spatial relationships in favor of simplifying the navigation task.

Thus, while a "good" diagram must be well laid out, it must also relate well to the *problem* it is designed to address [5, 22]. Diagrams are of help when they:

1. Leverage spatial metaphors and natural correspondences in the source domain [20, 46].
2. Convey patterns and relations in the source domain in spatial terms [14, 20].
3. Group together all information that will be used together [14, 20].

2.1.3 Core Diagramming Operations

From the above literature, we identify that users require the following operations in order to create effective diagrams:

Relative Positioning. Users should be able to specify the relative positions of diagram elements. This allows diagram elements to be spatially arranged to reflect the underlying structure of the domain, relating to the Gestalt principles of continuity, closure, and proximity. Placing graph elements consistently in relation to others can imply diagrammatic flow, groupings, and even underlying shapes. We identify two key aspects of relative positioning:

1. *Directional relationships* between diagram elements. For example, specifying that elements should be placed above, below, to the left, or to the right of one another.
2. *Angular flow* to describe element positions. For example, specifying that elements should be arranged in a circle, or that they should form a shape.

Grouping. Users should be able to group related diagram elements together. Leveraging the Gestalt principle of proximity, grouping related elements together can help users to better understand the relationships between them. This also allows diagrammers to group together elements that will be used together, reducing extraneous cognitive load. Grouping relationships in interactive diagrams can also help users reason about higher-level structures and relationships, leveraging the Gestalt principle of common fate.

Styling. Users should be able to control some visual aspects of how diagram elements are rendered. We identify two ways in which users might want to style their diagrams: theming and iconography. Theming involves influencing the aesthetic properties of diagram elements, such as the color, size, and shape of elements. This allows users to create diagrams that tap into the Gestalt principles of similarity and figure-ground.

Iconography, on the other hand, involves the use of pictorial elements in diagrams that themselves carry semantic meaning, e.g., representing an eye with an image instead of the word "eye" (Figure 1). Icons and symbols in diagrams can tap into pre-attentive processing, allowing users to quickly understand what a diagram element represents without needing to examine the diagram in detail.

With these primitives, diagrammers should be able to create artifacts that:

- Focus on important parts of the problem.
- Are tools for knowledge presentation. That is, they offer insight into instances of the spec being explored.
- Act as tools for knowledge discovery. That is, they show key aspects of bad-instances of the spec.

2.2 Bottom Up

While the top-down approach provides a theoretical foundation for our language design, much of the literature draws on abstract principles that may not be directly applicable to the formal methods domain.

To study how these principles manifest in practice, we contacted the Forge [32] team, who kindly gave us access to anonymized data on visualizations that students created in their upper-level undergraduate course on formal methods, as well as the results of a follow-up survey on students' visualization experiences. Students had given permission for this use.[2]

Figure 4 A student-drawn diagram illustrating the layout of their spec. This layout relies on relative positioning, grouped elements, and iconography.

The dataset contained anonymized retrospective comments on 58 final group projects (21 from 2022, 21 from 2023, and 16 from 2024), most of which were written in Forge, and visualized either as default-graphs or using D3 [3] to customize Sterling. As a result, we were able to gain insight into how two methods of visualization work for users who have roughly a semester of experience with formal modeling.

Projects covered a wide range of domains, from games like Solitaire to cryptographic protocols and group theory. Each project involved a code submission (a spec and optional visualization) as well as a recorded presentation or written summary explaining the domain being modeled, the design decisions made, and an optional demonstration of the visualization.

Most projects (47 total, 81 %) involved some kind of custom visualization (typically relying on Sterling's D3 integration). We examined these projects to determine how the principles we derived from the cognitive science literature were reflected in the visualizations students created. We also included projects in our analysis that did not include a custom visualization, but where authors were especially clear about wanting a custom visual component even if they did not have one, e.g., by presenting a hand-drawn diagram (e.g., Figure 4) or by manually rearranging nodes of the Sterling default-graph in their presentation.

[2] Some of these visualizations are shown in figure 11 of [32].

In our analysis, we found that:

- 44 projects (75 %) involved custom visualizations that had clear representation salience and minimal visual clutter. Each atom in the model was represented by at most one element in the visualization, and each element in the visualization corresponded to an important aspect of the model.
- 41 visualizations (70 %) communicated semantic relationships between elements via their relative orientation. Examples include the grid layout of cells when visualizing the games Minesweeper and Battleships, and the relative positions of nodes in red black trees.
- 19 visualizations (33 %) grouped nodes together inside explicit shapes to imply relatedness. Examples include piles of cards in the card game Solitaire, player hands in the game Chopsticks, and sets of elements in group theory.
- 4 visualizations (7 %) conveyed spec semantics by laying out diagram elements to imply shapes or directionality. Examples include the cyclic order of player turns in the card game Coup and travel paths in a map routing algorithm.

The Forge team also shared anonymized results of a survey. Students explained that they created custom visualizations due to the large, unstructured instances presented by the default-graphs. Specific issues included: too many edges and nodes, complicated states, and overlapping components. Their custom visualizations hid unnecessary information, conveyed key aspects of the source domain, and could even act as useful debugging tools.

While the decisions made by students may not fully represent those of experienced formal methods practitioners, they do provide valuable insight into some key aspects of the modeling process. While working on open-ended projects, students had to carry out real steps a practitioner would take: choose a domain they were interested in and knowledgeable about, formally model it, debug the models, and effectively communicate their output.

2.3 Top-Down and Bottom-Up Alignment

Our analysis of student projects suggests that diagramming principles from the cognitive science literature (Section 2.1), such as reducing cognitive load and highlighting important information, are very much applicable in the context of lightweight formal methods. Across a diverse range of domains, student visualizations preserved domain-specific spatial relations, grouped together related elements, and encoded spec semantics in terms of spatial layout and directionality. They also draw attention to a key additional requirement in the context of formal methods: support for bad-instances. We discuss this in further detail in Section 4.

2.4 The Pain Barrier to Custom Visualization (with D3)

We have motivated the value of custom visualizations. However, they come at a heavy price. Not only do users need familiarity with JavaScript, but D3 – one of the most influential online visualization libraries [2] – is notorious for its steep learning curve, API verbosity, and low-level operations [2, 28]. It is also worth noting that both JavaScript and D3 are entirely orthogonal learning objectives to someone learning or performing formal methods.

The nature of D3 also means that Sterling-with-D3 scripts are often enormous relative to the specs they are built to visualize. For example, the code for the river-crossing visualization in the Sterling demo [10] is *four times* longer than the spec (148 lines vs. 35 lines). It can also be difficult during maintenance and debugging to understand which parts of their D3 code correspond to which parts of the spec. Furthermore, a typical custom visualization is an "all-or-nothing" proposition: until enough of the visualization has been written, the user gets

Program ::= Constraint* Directive*

Constraint ::= CyclicConstraint
| OrientationConstraint
| GroupingConstraint

CyclicConstraint ::= Field FlowDirection

OrientationConstraint ::= Field Direction+
| Sig Direction+

GroupingConstraint ::= Field Target

FlowDirection ::= clockwise | counterclockwise

Direction ::= above | below | left | right
| directlyAbove | directlyBelow
| directlyLeft | directlyRight

Target ::= domain | range

Directive ::= PictorialDirective
| ThemingDirective

PictorialDirective ::= Sig Icon

ThemingDirective ::=
Attribute | Color
| Projection | VisibilityFlag

Icon ::= path height width

Attribute ::= Field

Color ::= SigName color

Projection ::= Sig

VisibilityFlag ::= hideDisconnected
| hideDisconnectedBuiltIns

Figure 5 Abstract syntax of Cope-and-Drag. Identifiers from the source Alloy spec are in orange, while `CnD`-specific keywords are in blue.

no benefit at all. This critically violates the "pay as you go" flavor that is characteristic of lightweight formal methods. It also makes it difficult to iterate, debug, and maintain custom visualizations, discouraging the use of D3 for exploratory diagramming tasks.

In Section 2.2, we have already indicated that some students used paper drawings or manually dragged boxes to simulate what they wanted. In the same survey in which students praised the value of custom visualizations, 37 respondents (44%) did not use JavaScript/D3 *at all*. These students primarily identified the complexity of D3 and the need to use JavaScript as a significant barrier to building custom visualizations, reporting difficulty learning JavaScript and D3, having enough time, starting, and finishing, corroborating past findings [2].

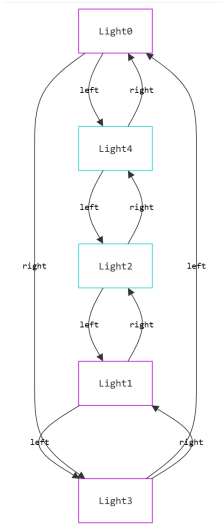
We thus see that the potential for custom visualization is undermined by the current tooling, which is onerous and often too daunting for users. (We discuss other drawing systems and languages in Section 6.) This observation demands a better, lightweight way of capturing the *essential* elements of diagrams that draws on the above principles and examples.

3 The Cope-and-Drag System

Cope-and-Drag (`CnD`) is a lightweight diagramming language designed for use with Alloy-like languages. The tool's name is inspired by the cope-and-drag casting process in metallurgy, reflecting its adaptable approach to diagram creation. `CnD` implements the diagramming operations identified in Section 2, and applies them to directed graphs with the goal of improving upon default-graphs. These generated diagrams support user interactions *while maintaining consistency with the diagram spec*. Thus, `CnD` not only facilitates diagram creation, but also supports *faithful* interactive exploration of instances being visualized.

3.1 `CnD` Primitives

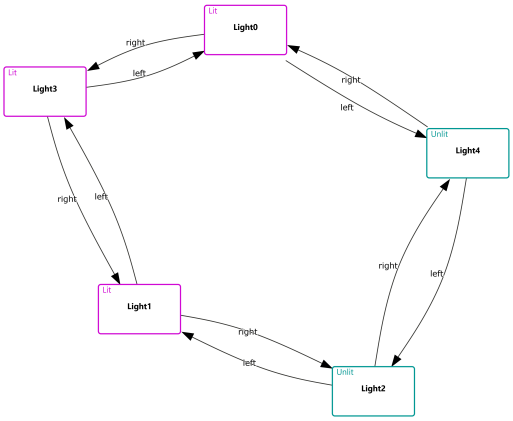
`CnD` primitives operate upon Alloy/Forge sigs (which are essentially types) and their fields. These primitives fall into two categories: directives and constraints. Directives are used to specify the visual representation of a sig or field. Constraints are used to specify the relationships between visual representations. The resultant diagrams are interactive, allowing



(a) Default-graph.

```
constraints:
  - cyclic:
      field:  left
      direction: clockwise
```

(b) `CnD` spec featuring a `cyclic` constraint. Identifiers from the source Alloy spec are in orange.



(c) `CnD` diagram.

Figure 6 ring-lights triad, contrasting a default-graph with `CnD` for an instance of a puzzle involving 5 lightbulbs arranged in a ring. The default misleadingly positions the nodes in a line. Supplement Figure 1 presents a larger version of the diagrams.

users to drag nodes around and explore the spec, while ensuring that the visual representation remains consistent with the specified constraints. For instance, if a constraint specifies that `A` must be to the left of `B`, a user will not be allowed to drag `B` to the left of `A` (ab). In cases where constraints cannot be satisfied, no diagram is produced (Section 3.2).

Each `CnD` primitive is designed to fulfill a specific requirement identified in Section 2.

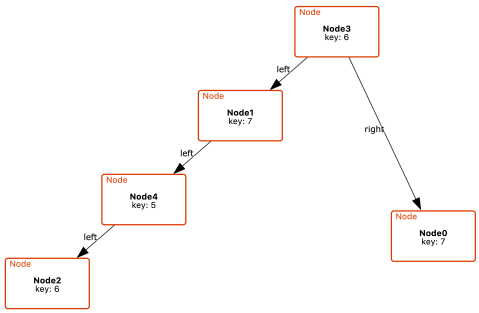
Relative Positioning: Cyclic and Orientation Constraints. Cyclic and Orientation constraints are used to specify the *relative positioning* (Section 2.3) of diagram elements.

Cyclic constraints lay out atoms related by a sig field along the perimeter of a notional circle, suggesting they form a shape. If a field does not define a complete cycle, each connected subset is arranged on the circle's perimeter based on a depth-first exploration of the field. Atoms in the relation can be positioned either clockwise (the default) or counterclockwise. Figure 6 presents a triad. The `CnD` diagram arranges nodes in a circle, in accordance with the spec, because the `CnD` program includes a `cyclic` constraint. The default-graph, lacking this domain-specific information, arranges nodes in a line. A user would need to carefully read the edge labels or the underlying spec to discover that the nodes have a cyclic relationship.

Orientation constraints specify the placement of atoms in a diagram instance relative to one another. These constraints can be applied between sig types (e.g., all atoms of type `A` must be to the left of those of type `B`) or along sig fields (e.g., atoms related by the `child` field must be placed downward). For example, the orientation constraints in Figure 7a are at the heart of a `CnD` program for binary tree instances produced by Figure 2. Supplement Figure 2presents a triad with the full `CnD` spec and a default-graph. The default produces a Stroop effect, putting the left child on the right and vice versa. Users have no way to prevent this effect.

```
- orientation:
    field: left
    directions: [left, below]
- orientation:
    field: right
    directions: [right, below]
```

(a) CnD spec.



(b) CnD diagram.

Figure 7 bt CnD spec and diagram for binary tree instances produced by Figure 2.

Grouping Constraints. Grouping constraints specify that atoms related by a sig field should be grouped together spatially. Groups can be created based on either the range (default) or domain of the sig field. Grouping goes beyond merely associating atoms: CnD replaces the individual edges between the group source and target atoms with a single edge, and places grouped atoms within a bounding box. Groups cannot intersect unless one is entirely subsumed by another.[3]

Figure 8 shows the CnD-rendered state of a river-crossing puzzle, using groups to indicate shores. This picture significantly reduces the number of edges; it also shows which entities are on the same shore, thereby avoiding confusion because near and far shore entities are interleaved in the default-graph version. A careful look at the default Sterling visualization of a similar river crossing spec in the Forge paper [32, Figure 5, left] shows that it too exhibits a Stroop effect.

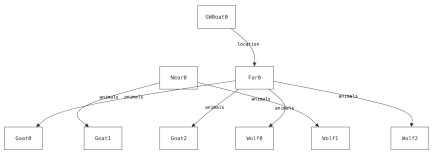
In general, grouping constraints allow users to focus on higher-level structures and relationships without being overwhelmed by individual connections.

Directives. Directives allow users to control the visual aspects of how atoms of a particular sig type are rendered.

Pictorial directives specify that atoms of a particular sig must be represented by a specific icon (with or without a textual label as well). These icons help to visually situate the output in the domain being modeled, making the output less uniformly textual and exploiting pre-attentive processing. Two examples are the facial features in Figure 1 and the fruit in Figure 14.

CnD also supports a subset of the theming directives made available by the Alloy Visualizer and Sterling. Attribute directives display a field relation as a label on atoms of a particular sig type. Sig color directives associate specific colors with atoms of a certain sig type. Projection directives allow diagrams to focus on the atoms of a particular sig and their directly connected relations, hiding all other elements. Users can also hide disconnected atoms, depending on the provenance of their sig type, by using the `hideDisconnected` and `hideDisconnectedBuiltIns` visibility flags.

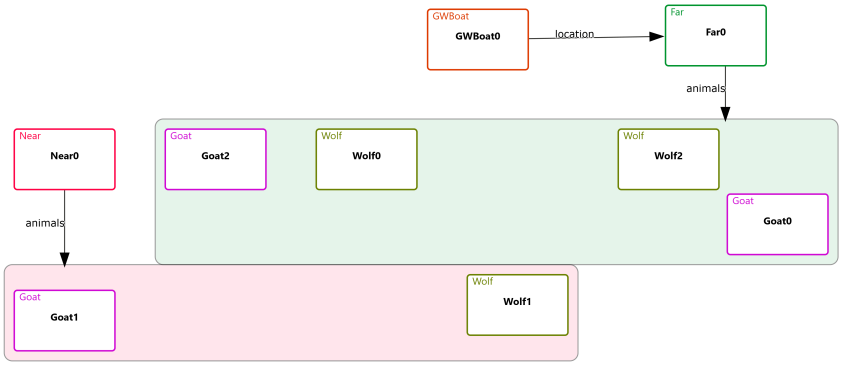
[3] The requirement that groups not intersect is a technical limitation of the WebCola library [8] used by CnD. Overlapping groups affect the computation of bounding boxes and convex hulls. Group bounding boxes, as determined by WebCola, are also not guaranteed to be minimal, and can sometimes be larger than necessary.



(a) Alloy Visualizer interleaves near and far shore entities.

```
constraints:
  - group:
      field:  animals
      target: domain
```

(b) CnD spec.



(c) CnD visualization groups together by the shore they are on.

Figure 8 rc Alloy Visualizer and CnD representations of a river-crossing puzzle instance.

3.2 When Constraints Cannot be Satisfied

All CnD constraints are hard constraints, and thus *must* be satisfied for a diagram to be produced. Constraints might not be satisfied for one of two reasons:

1. A single CnD constraint definition could be inherently unsatisfiable. For example a constraint that requires the same field to be laid out both leftwards and rightwards:

   ```
   - orientation: {field:  next , directions: [right, left]}
   ```

   This represents a bug in the CnD spec, and can be identified statically. In this case, CnD produces an error message in terms of the constraints that could not be satisfied.

   ```
   Inconsistent orientation constraint:
   Field next cannot be laid out with directions: right, left.
   ```

   Checking for inconsistencies is a relatively simple process: CnD just needs to ensure that the same field is not laid out in conflicting directions.
2. Some constraints may be individually satisfiable, but become contradictory when laying out a specific instance. This is akin to a dynamic error, as it depends on the structure of the instance being visualized. For example, consider the constraints:

   ```
   - orientation: {sigs : [ X ,  Y ], directions: [below]}
   - orientation: {sigs : [ Y ,  Z ], directions: [below]}
   - orientation: {sigs : [ Z ,  X ], directions: [below]}
   ```

   If an instance's elements form a cyclic dependency, the constraints become unsatisfiable. However, an instance may lack atoms of, say, Z, in which case the constraints are satisfiable. CnD identifies these inconsistencies by treating layout as a linear optimization problem. Constraint primitives are incrementally added to the Cassowary linear constraint solver [1]

to ensure that a satisfying layout exists. `CnD` stops at the first indication that constraints cannot be satisfied, and provides an error message in terms of the currently added constraints and instance atoms (e.g., Figure 10).

In both these cases, `CnD` does *not* produce a diagram. Instead, it provides an error message explaining that the constraints could not be met. If all constraints can be satisfied, `CnD` relies on the WebCola library [8, 9] to produce a diagram.

3.3 The Lightweightness of `CnD`

Having described the design of `CnD`, we now briefly justify calling it "lightweight".

Critically, `CnD` *refines* a default-graph output. Thus, the user can apply it incrementally: the empty `CnD` program still produces output, indeed, the exact same output as Sterling. (We discuss this design principle in more detail in Section 8.) Users can thus decide which aspects of the output most need refinement (just as they already do with Alloy Visualizer directives) and apply these rules incrementally. Furthermore, it is easy to turn a rule on or off, to see whether it has the desired effect. This incremental input refinement does not, however, imply incremental output refinement. Adding a new rule to a `CnD` program a new rule may change the layout substantially. Enforcing minimal output changes requires implicit constraints, which can obscure bad instances (Section 4) or introduce unintended Stroop effects (Section 2.1).

`CnD` programs also tend to be very brief. There is only a small number of primitives (constraints and directives) in the language (Section 3.1). Because these primitives apply to sigs and fields, there is an upper-bound on how many we can write, and this is independent of the size of the *operational* part of the spec.

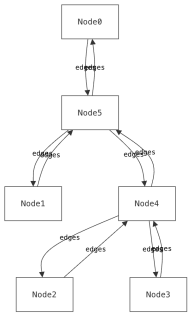
`CnD` output is also "lightweight" in another sense: the language offers few bells-and-whistles, and the output is not necessarily pretty. We view the path from a `CnD` program to a full-fledged custom visualization as akin to that from an Alloy or Forge spec to a correct-by-construction spec verified by a proof assistant. The goal is not to produce diagrams that are attractive, but rather ones that are functionally useful and do not confuse, mislead, or otherwise abuse cognitive principles.

3.4 Where do `CnD` Programs Belong?

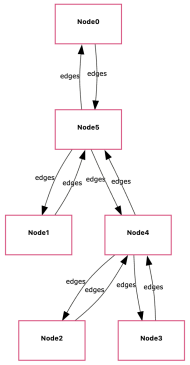
A design question is where to write a `CnD` program. Is it part of the spec, or does it live separately from the spec? This partially depends on whether it even makes sense to have more than one `CnD` program for a spec.

We argue that it is not only possible but also sensible to have multiple views on instances of the same spec. As an example, suppose a user is modeling a self-stabilizing protocol for a distributed system. Depending on the goal, they might want to use a ring to show network topology (e.g., Figure 12b), a DAG to examine hierarchies, or groups to explain permissions. No one view is more privileged than the other. They might even switch between these as they explore and debug different aspects.

In our current implementation, the `CnD` program resides in the visualizer. This is for three more reasons beyond the one given above. First, it is easy to explore commands incrementally even for a given instance (which follows Ma'ayan et al.'s recommendation [22]). Second, the same visualizer works with both Alloy and Forge (and any other tools that use the same protocol), providing portability of these ideas. Finally, it saves us the effort of modifying the implementations of Alloy and Forge.



(a) Default-graph.



(b) undirected-tree CnD, from an empty program.

Figure 9 Default Alloy and default CnD outputs for an undirected tree spec.

That said, CnD programs are not completely separate from the spec; they are really a *spatial refinement* of the spec. Therefore, it would be meaningful to extend the spec languages to include (and name) each of these views. The "static" checking that CnD provides (Section 3.2) is then arguably part of the act of checking the spec itself. The user interface could then provide a menu of these pre-defined views, which were presumably chosen by the domain expert, while enabling the user of the spec to write their own views as well (as they currently can).

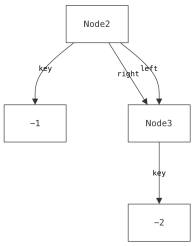
3.5 Visual Details

Whenever possible, CnD diagrams also encode a set of visual principles designed to reduce cognitive load and enhance user comprehension. Graph edges are laid out in a way that minimizes crossings and with as few bends as possible [39]. To reduce edge ambiguity, CnD also ensures that arrowheads are not incident on the same point of a node. We see an example of this in Figure 9, whose CnD program is empty, so the CnD output embodies only the above principles relative to the default-graph: observe the absence of overlapping edge labels. In addition, in CnD output, atoms of each sig are assigned a unique color by default, making salient the type of each node in the diagram. This is not visible in Figure 9 because all the atoms belong to the same sig, but we see it in several outputs such as Figures 1, 12a, and 12c (whose CnD programs do not include any coloring directives).

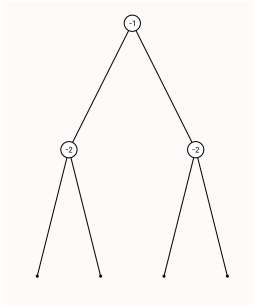
4 Bad-Instances: When Custom Visualizations Can Fail Silently

As we have discussed in Section 1, unlike tools like verifiers – which are inert in the absence of a property – lightweight formal methods can consume *just* a spec and show concrete instances of it. (Researchers have shown how to leverage non-instances as well [12, 26].) This helps bootstrap the formalization process by surfacing latent properties.

In practice, the modeling process is a series of incremental efforts that produces several incorrect specs before reaching an acceptable spec. Incorrect specs are the source of what we dub *bad-instances*: instances that do not align with the informal spec that the user is trying to formally model. It is critical for a visualizer to faithfully render bad-instances, since the user needs to see them and realize that the spec is incorrect.



(a) Default-graph.



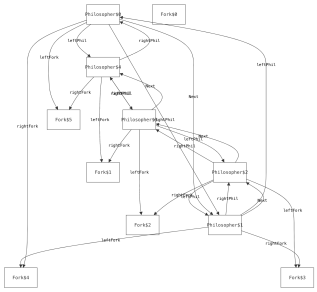
(b) Sterling-with-D3.

Error: The instance being visualized is inconsistent with layout constraints.

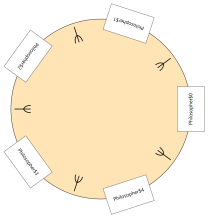
Constraint:
`ENSURE: Node3 is to the left of Node2`
conflicts with the following constraints:
`ENSURE: Node2 is above Node3`
`ENSURE: Node2 is to the left of Node3`
`ENSURE: Node2 is above Node3`

(c) **bt-dag** `CnD` reports an error.

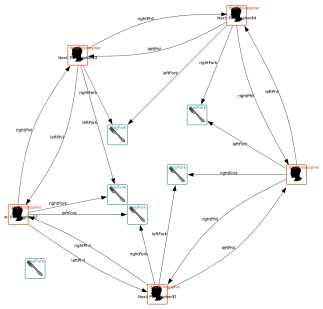
Figure 10 Outputs for a bad-instance of a binary search tree spec. The instance is a DAG with two nodes. The custom Sterling visualization misleadingly draws a tree.



(a) Default-graph (6 forks).



(b) Sterling-with-D3 (5 forks).



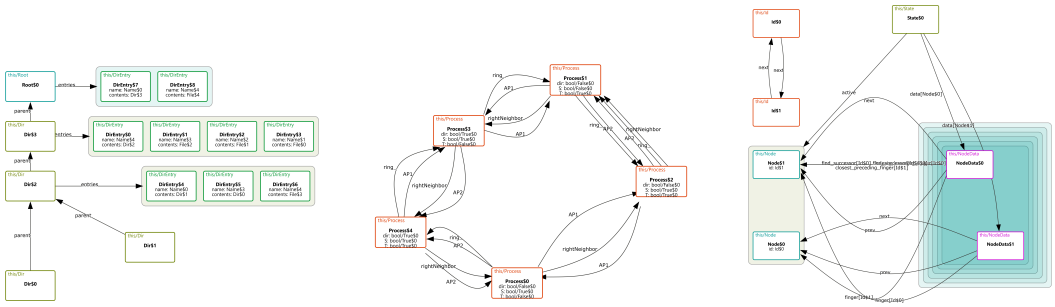
(c) **phil-inv** `CnD` (6 forks).

Figure 11 Outputs for a bad instance of a dining-philosophers spec. The instance has 6 forks but only 5 philosophers. Only `CnD` shows the extra fork and a roughly circular layout.

Faithful rendering is not an issue with default-graphs: for all their weaknesses described earlier in this paper, to their credit, instances and bad-instances alike are displayed as directed graphs, with no information lost or hidden (except as stated in an explicit user directive). In contrast, a custom visualization needs to be sensitive to bad-instances and not misrepresent them or, worse, accidentally suppress the way in which they are bad (so the user may never discover that the spec is flawed). Nelson et al. [32] discuss this in their study of custom Sterling-with-D3 visualizations. They find that as users modify their spec, domain-specific visualizations would behave unexpectedly, leading to confusion and frustration for users.

This problem is not limited to students. Consider these two examples writen by people who are expert in both formal modeling and Sterling:

- A common underconstraint when modeling a binary search tree is to forget to enforce that a node's left and right children are distinct. Figure 10 shows the result of applying an expert-authored visualization to a bad instance: a DAG. The custom visualizer is not robust to this possibility and thus masks an important failure. The default-graph reveals the underlying structure, while `CnD` does even better, displaying an error message about the inconsistency between the bad-instance and diagramming constraints.
- An expert-authored visualization of the dining philosophers problem from the Sterling website [10] also fails silently when presented with bad-instances. A common underconstraint when modeling the dining philosophers problem is not ensuring that the number



(a) filesystem. (b) ring-orientation. (c) chord.

Figure 12 CnD diagrams for three example specs.

of forks equals the number of philosophers. As shown in Figure 11, the expert-written visualization presents the bad-instance as an instance, hiding the presence of an extra fork. The default-graph hides no information, but the lack of semantically meaningful layout means that the sixth fork is not particularly salient. The CnD diagram, in contrast, arranges the domain in a meaningful way, and adds icons to distinguish philosophers from forks, making the extra fork easier to see.

Of course, an attentive visualization author can protect against some of these issues with checks, preconditions, and assertions. Besides being onerous and easy to overlook, these suffer from a more fundamental problem: they are limited by the author's assumptions of what can go wrong. That is, they can only handle *known* unknowns: faulty specs that the author thinks of. However, these would probably be turned into properties in the first place. Problems that the author does not think about (*unknown* unknowns) – the very kind for which lightweight formal methods are ideal – are, by definition, impossible to account for, and may well result in misleading or deceptive visualizations. This problem is especially salient in educational contexts, where it is well-documented that experts (if they provide visualizations) suffer from blind spots [29, 30] about what mistakes students might make.

CnD strikes a happy medium between generic and custom visualization. Because it only refines the generic output, it does not hide information. Because the refinements encode domain information, bad instances either fail noisily (Figure 10) or produce a structured diagram (Figure 11). We report on a user study involving bad-instances in Section 5.2.3.

5 Evaluation

We now evaluate CnD. Our evaluation has three parts. In Section 5.1, we examine its ability to generate visualizations for new domains. In Section 5.2, we provide empirical data on the effectiveness of the generated diagrams. Finally, in Section 5.3, we discuss CnD's performance when generating various diagrams.

5.1 Examples

It is unsurprising that CnD can capture well the demands of the examples in Section 2.2, given that these were effectively the "training set." We instead need a distinct "test set." For that, we use examples from the Alloy Models repository, a publicly available community-maintained collection of specs (`github.com/AlloyTools/models`). It is unclear how to *comprehensively*

study these; instead, we chose a sample of specs for domains that were sufficiently different from one another and that we understood well (for which we could therefore meaningfully write `CnD` programs) and assess their effect.

Filesystem. Figure 12a shows a generic filesystem rendered in `CnD`. It uses an orientation constraint to lay out directories hierarchically, and a combination of grouping and orientation constraints to lay out entries in the same directory. Entry contents and names are shown as attributes. Supplement Figure 4presents a triad contrasting the default-graph and the `CnD` program that refined the graph.

Ring Orientation. Figure 12b shows how we can visualize instances of an Alloy spec for the self-stabilization problem in distributed systems [7] for a ring of processes. A cyclic constraint is used to convey the circular topology of processes, while boolean flags are shown as attributes. Supplement Figure 5presents a triad.

Chord. Figure 12c visualizes an instance of the Chord [42] distributed hash table protocol. This diagram groups active nodes together, and lays out the previous node seen by a node to its direct left. Finally, it uses grouping constraints to show the results of the `find_successor` and `find_predecessor` operations. (The multiple subsuming rectangles in this diagram are a limitation of the WebCola library used by `CnD`, and are used to indicate subsumed groups. One could imagine alternate implementations that collapse these groups into a single rectangle.) Supplement Figure 7 presents a triad.

5.2 User Studies

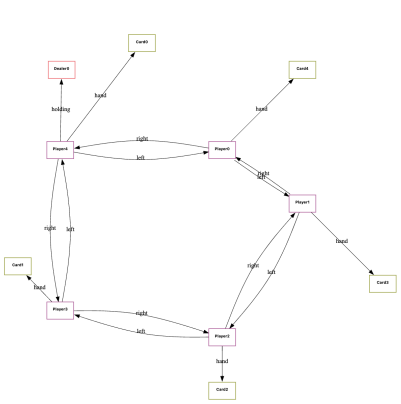
We now present user studies that examine the effectiveness of `CnD`:
Q1. Do `CnD` specs help users understand the instances being explored (Section 5.2.1)?
Q2. Can pictorial directives further enhance spec understanding (Section 5.2.2)?
Q3. Do `CnD` diagrams make salient key aspects of bad-instances (Section 5.2.3)?
We use the default-graph as a baseline for comparison, for three reasons. First, it is what a user would get by default. Second, it is a fixed entity (up to the choice of specific visualizations), whereas the set of possible custom visualizations is limitless and our choices might be biased. Finally, as noted in Section 4, it does not suppress problems with bad-instances.

For all three studies, we recruited participants from the Prolific platform's pre-defined pool of users identified as having "computer programming skills". Each participant was compensated USD 5 for about 15 minutes of time, and no individual took part in more than one study. Our study was deemed to not be human subjects research by our review board; we nevertheless took reasonable safeguards to protect the participants. The full versions of the study diagrams are available in the supplement.

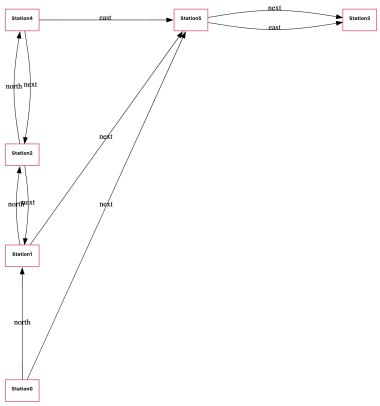
5.2.1 Instance understanding

In order to better understand how `CnD` diagrams offer insight into spec instances, we designed three specs, each of which lends itself well to one kind of `CnD` constraint.

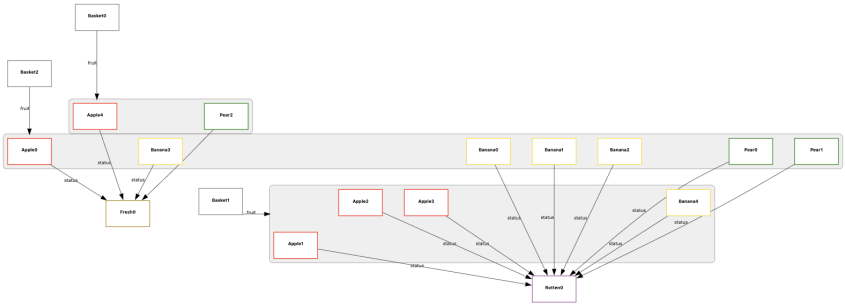
- Cards (Figure 13a): This scenario tested the efficacy of cyclic constraints by asking participants questions about the movement of player roles across rounds of a card game.
- Subway (Figure 13b): This scenario tested the efficacy of orientation constraints by asking participants questions about the relative geographical positions of subway stations.
- Fruit (Figure 13c): This scenario tested the efficacy of grouping constraints by asking participants questions about the relative distribution of fruit across baskets.



(a) cards diagram.



(b) subway diagram.



(c) fruit diagram.

Figure 13 CnD outputs for the three Q1 user study scenarios (Section 5.2.1).

Users were shown one instance of the spec. For each spec, we developed 2–3 questions that would exercise the user's understanding of the instance they were shown. The specs avoided details that would enable a user to answer the questions based on prior knowledge rather than the instance itself.

Each participant (n = 38) was randomly assigned 2 scenarios, one visualized using the default-graph and the other using CnD with *only* the associated constraint. Participants shown CnD diagrams were significantly more likely to get questions correct than those shown default-graph visualizations (62.75 % vs 48.04 % correct, $Z = 2.11$, $p < 0.05$), with a small-to-medium effect size ($d = 0.26$). Participants took approximately the same amount of time to complete the tasks (mean time on task: 105.3s for CnD, 100.46s for default-graph, $t = -0.81$, $p = 0.22$).

5.2.2 Pictorial directives

In order to test the efficacy of pictorial directives, we conducted a second study extending the Fruit scenario described in Section 5.2.1. While the scenario remained unchanged, participants (n = 30) were assigned one of two variations of the fruit diagram: one with CnD's default node representation (Figure 13c) and one with images of fruits on certain nodes (Figure 14). (Supplement Figure 8presents both diagrams together.)

Table 1 Scenarios used to study the effectiveness of `CnD` constraints on spec understanding.

Scenario	Constraint	Correct Answer Percent		Mean Time on Task	
		default-graph	`CnD`	default-graph	`CnD`
cards	Cyclic	28 %	36 %	161.96s	128.81s
subway	Orientation	31 %	56 %	84.02s	121.73s
fruit	Grouping	69 %	74 %	60.15s	70.32s

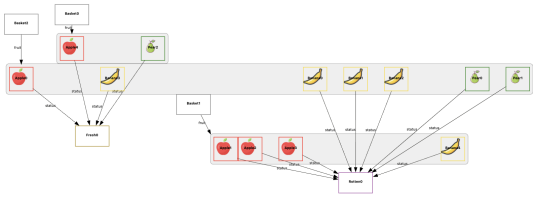


Figure 14 fruit-icons diagram for the Q2 user study (Section 5.2.2).

Participants shown the diagram with pictorial directives answered questions correctly more often, and faster than those shown the diagram without them (86.67 % vs 73.33 % correct, mean time-on-task: 56.76s vs 68.55s). However, these were not statistically significant at a 95 % confidence level (correctness: $Z = 1.60$, $p = 0.11$, time-on task: $t = -0.96$, $p = 0.34$).

Participants were then shown the fruit scenario with three different diagram styles: default-graph, default `CnD`, and `CnD` with pictorial directives, and asked to rank them in order of preference. Participants tended to rank the `CnD` with icons diagram highest (mean rank 1.45), followed by the `CnD` with default nodes (mean rank 2.00), and then the default-graph (mean rank 2.57). This difference in ranking was statistically significant (Friedman $Q = 19.27$, $p = 6.55e - 5$). A pairwise comparison showed that the ranking for `CnD` with icons was significantly higher than that of the default-graph ($p = 3.4e - 5$). Participants identified that the `CnD` with icons diagram conveyed the same information as `CnD` with default nodes, but still showed a slight preference for the former ($p = 0.07$). Their feedback praised the "clarity" of the "structured layout", and that the icons "made relationships more visible and easy to understand" and their absence made it "harder to visualize the elements quickly", while default-graph had "complex" layout and made it "difficult to find what they are looking for".

5.2.3 Bad-Instances

To identify *bad* instances, the participant must have some intuitive sense of what constitutes a *good* instance. To avoid the confound of training (which some participants may do only half-heartedly), we picked two situations that users are likely to be able to implicitly judge: the game Tic-Tac-Toe and a human face (Figure 1 (B), *excluding* the icons (C), which we felt would introduce a confound). For each scenario, we constructed a valid and invalid instance and visualized them using the default-graph and `CnD`. The Tic-Tac-Toe instances are shown in Figure 15, while the face instances (similar to Figure 1) are in Supplement Figure 10.

Out of n = 40 participants, half of them saw `CnD` diagrams for Tic-Tac-Toe and default-graph diagrams for the Face scenario, while the other half saw the reverse. For each scenario, participants were first shown a diagram of a good instance, asked to judge its validity, and then asked a follow-up question about the scenario that required them to parse the diagram's structure. This helped build familiarity with the model, scenario, and diagramming style.

Table 2 Scenarios used to study the understanding of bad-instances.

Scenario	Good Instance	Bad Instance
Tic-Tac-Toe	3x3 grid of Xs and Os representing a Tic-Tac-Toe board (**ttt**).	9 node graph marked with Xs and Os that do not form a grid (**ttt-inv**).
Face	Facial features arranged to match a human face (**face**).	Facial features arranged to align one eyebrow with the eyes (**face-inv**).

Participants were then shown a diagram of a bad-instance, and asked to judge its validity. (To be clear, participants did not *know* they were being shown instances in this order: they were simply passing judgment on two diagrams. But we showed the valid instance first to avoid confusing them.)

We divide responses into two groups:

- Validity judgments: The participant asserted that the bad-instance diagram was either valid or invalid. Participants who said it was invalid were asked to explain why, while those who asserted it was valid were asked the same follow-up question as before.
- Uncertain responses: The participant was unsure about the diagram's validity. These participants were asked why they were uncertain.

Uncertain Responses. It is tricky to interpret uncertainty. If a user is spot-checking a spec, uncertainty may arouse their suspicions, leading them to investigate further. On the other hand, if a user is browsing a spec, they may simply move on to the next instance. However, we did not examine this in depth because relatively few participants expressed uncertainty about validity. Only 5 participants (12.5 %, mean time = 37.84s) were unsure about the validity of `CnD` diagrams, while 3 participants (7.5 %, mean time = 29.52s) were not sure about the validity of default-graphs.

Validity Judgments. Participants were significantly more likely to identify bad-instances and correctly explain *why* they were bad when shown `CnD` diagrams than when shown the default-graphs (71.43 % vs 43.24 %, $p = 0.03$, $\chi^2 = 4.73$). Participants also made these validity judgments faster with `CnD` (mean of 58.06s vs. 69.82s), but this difference was not statistically significant ($p = 0.37$, $U = 568.00$).

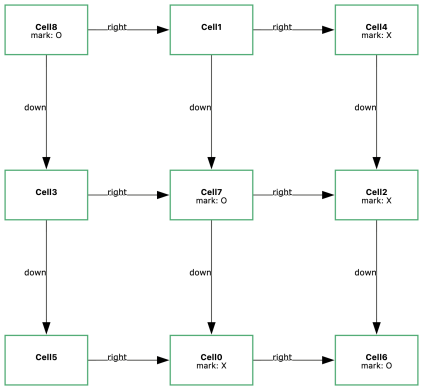
5.2.4 Threats to Validity

Several factors may limit the generalizability and interpretation of our findings. While study participants had prior programming experience, this does not imply familiarity with formal methods, Alloy, or instance visualization. Effects on understanding could differ for users with more experience in these areas [24]. We believe, however, that this wide pool of participants might help indicate `CnD`'s utility across several kinds of programmers.

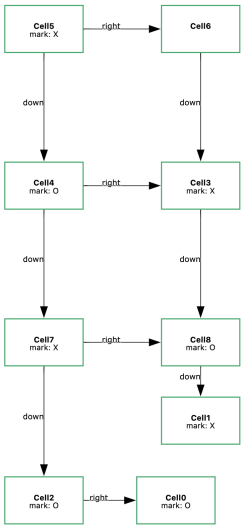
While we attempted to control for domain expertise by selecting scenarios that were either novel (Sections 5.2.1 and 5.2.2) or universally familiar (Section 5.2.3), different kinds of expertise could affect understanding. For instance, the ability to interpret a diagram might be different if the interpreter is also the author of the spec.

The relatively simple and well-understood nature of study scenarios might also limit how well our results generalize to "real world" formal methods scenarios. A bad instance, for example, might be less salient when dealing with more complex or less familiar domain.

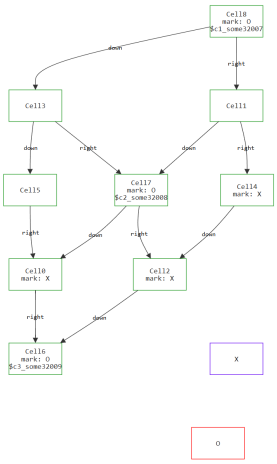
Finally, as with any online study, there are inherent threats related to participant engagement and motivation, which could influence the outcomes. We believe that these limitations do not invalidate our findings, but they do warrant caution in interpreting the results as directly applicable to experienced Alloy users in real-world settings.



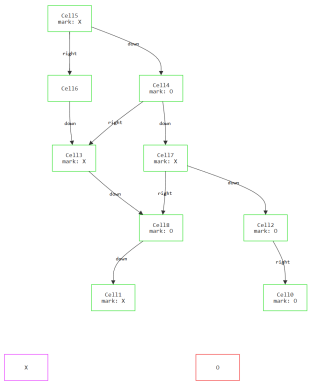
(a) `ttt` `CnD` good-instance.



(b) `ttt-inv` `CnD` bad-instance.



(c) default-graph good-instance.



(d) default-graph bad-instance.

Figure 15 Good and bad Tic-Tac-Toe instances from the Q3 bad-instance study (Section 5.2.3).

5.3 Performance

The Cassowary solver used by `CnD` to determine constraint satisfaction is guaranteed to find a solution if one exists [1]. This means that for any set of constraints that admit a valid visualization, `CnD` will always produce a solution. To assess performance, we measured the average time (over 50 runs) taken to generate diagrams for all 20 scenarios referenced in this paper and supplementary material. All measurements were performed on a 2.30 GHz processor (AMD Ryzen 7) with 16GB memory. With the exception of `chord`, `face`, and `face-inv`, which took 0.98s, 1.03s, and 1.06s, all the others took half a second or less. All times are within range for interactive use. Constraint solving took under 50ms in each case; the rest of the time was spent on graph layout. Supplement Section 4 presents full numbers.

6 Related Work

CnD sits between default visualizations, which lack domain-specific semantics, and programmatic tools, which require users to define every aspect of the diagram from scratch:

1. Generic visualizers support all instances (Section 6.1). While they are flexible and broadly applicable, their lack of domain-specific insight can lead to confusing visualizations (Section 3.1).
2. Drawing and diagramming tools offer users fine-grained control over visualizations, enabling them to craft detailed domain-specific diagrams (Section 6.2). This control demands significant effort. Users have to manage the nuts-and-bolts of diagram creation, dealing not only with domain-level constructs, but also actual "drawing" primitives (e.g., lines, shapes, points).

6.1 Default-graph Visualizers

The Alloy Visualizer [16] is the default visualizer for Alloy instances. It generates directed graph visualizations of the spec instance, with nodes representing atoms and edges representing relations between them. Users are provided a range of theming options, such as being able to change the color or shape of graph nodes or hide nodes via projection. When Alloy's magic-layout feature [40] is enabled, it can even adjust its theming and layout heuristically. Users, nevertheless, struggle to make sense of these as specs grow in size and complexity [24].

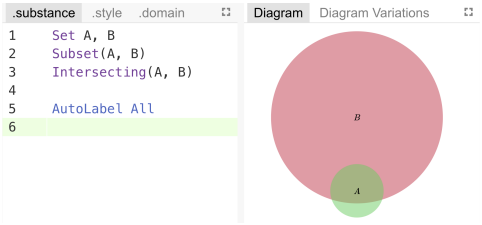
Sterling [11] is a web-based variant of the Alloy Visualizer that is designed to make large instances more comprehensible. Nodes and edges are laid out hierarchically, in an effort to reduce cognitive load, and layouts are consistent across instances, making the differences between them more salient.

We group both these tools together as default-graphs – generic visualizers that do not encode any domain-specific knowledge about the spec being visualized. Although both Alloy and Sterling provide numerous theming options, users cannot exercise control over *layout* decisions. On the other hand, the simple node-edge representation makes clear how domain constructs map to visual elements (*representation salience*), preserves domain terminology (*vocabulary correspondence*), and remains robust to bad-instances.

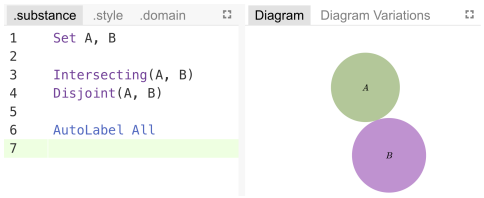
Of the prior work on Alloy visualization, CnD is most related to Couto et al. [6], which proposes *layout managers* for Alloy. For example, one might say that parking spaces should be arranged in a linear order with corresponding cars above. This work covers a larger set of primitives than CnD, such as grid and tree layouts. However, orientation appears to be defined by a single relation for each sig, which prevents expressing, e.g., the bidirectional layout of Figure 9. It is also unclear how layout managers compose (Figure 1) and how conflicts (Section 3.2) would be resolved. These works are thus complementary, with Couto et al. focused on expressive breadth, and CnD on composition and bad-instance discovery.

6.2 Drawing and Diagramming Tools

Sterling-with-D3. In addition to its default layout, Sterling also allows users to define custom visualizations using the D3 visualization library [3]. D3 is a powerful, expressive JavaScript library that gives users a great deal of control over the appearance of their diagrams. This enables diagrams that use a wide range of domain-specific visual encodings to represent relationships between entities. This complexity, however, comes at a cost. We observe barriers to adoption of D3 in Section 2.4.



(a) Penrose diagram of a bad-instance. Set A is a subset of set B in the spec but not in the visualization because the style file assumes that intersecting sets cannot subsume one another.



(b) Penrose diagram of a nonsensical instance in which sets A and B are declared to be intersecting and disjoint. This produces output with no indication of an error.

Figure 16 Two misleading Penrose diagrams.

Penrose. Penrose [51] is a domain-specific language designed to build diagrams of mathematical objects. The language's design is informed by both Gestalt principles (Section 2.1.1) and the diagramming tool requirements described by Ma'ayan et al. [22].

Penrose introduces a structured approach to the diagramming process by requiring users to separate the content of their diagram from how it is drawn. Diagrams, therefore, are constructed as three sub-programs: domain, substance, and style. The domain specifies the rules of the mathematical object being diagrammed, the substance specifies the instance of the object being visualized, and the style specifies how the object should be visualized. This separation of concerns makes Penrose a very powerful diagramming language, with the ability to generate diagrams across a wide range of domains.

Penrose substances and domains can be thought of as analogous to specs in formal modeling languages. A Penrose domain is similar to an Alloy spec, while a Penrose substance corresponds to an instance of that spec. In contrast, "style files" are more akin to programs. Users must describe how diagram elements should be constructed from rendering primitives (e.g., lines, shapes, points) before they can write higher-level constraints on how these elements should be laid out. Given the choice of rendering primitives, a user may even have to specify the order in which higher-level constraints are applied.

This represents a heavy up-front cost. A user must have already built the diagram from the ground up before they can begin to add layout constraints relevant to their domain.

Penrose's heavy up-front cost disincentivizes its use as an exploratory tool. The Penrose directed graphs demo,[4] for instance, involves nearly 150 lines of style code, including 13 constraints, 4 constraint application phases, and complex visualization calculations (e.g., for the angle of incidence of arrowheads on nodes). Much of this would have to be re-written if the shape of nodes were to change from circles to, say, squares.

After spending significant time on the nuts-and-bolts of rendering, a Penrose user might plausibly still find themselves looking at a misleading visualization for one of two reasons:

1. Since users have to build visualizations from scratch, Penrose diagrams may be brittle to bad-instances (Section 4). We show an example of this in Figure 16a, based on the Penrose Euler Diagrams demo.[5]
2. Penrose's numerical optimization problem is designed to always try to find a solution. This means that the system can, misleadingly, generate diagrams even when some user-specified constraints are not satisfied (Figure 16b).

[4] https://github.com/penrose/penrose/tree/main/packages/examples/src/box-arrow-diagram
[5] https://penrose.cs.cmu.edu/try/?examples=set-theory-domain%2Ftree-euler

BlueFish. Bluefish [37] is a declarative diagramming framework inspired by the principles of component-based UI frameworks. The core Bluefish primitive is the *relation*, which is used to capture semantic associations between diagram elements. Bluefish is particularly well suited to diagramming because these relations are composable and extensible. For example, relations are a natural way to define a diagram where elements overlap with one another.

However, Bluefish relations are defined in terms of concrete diagram elements, rather than abstracting over their types. As a result, an author who wishes to *dynamically* populate relations must fall back on Bluefish's host language (JavaScript). This is especially pertinent in the context of model-finding tools, like Alloy and Forge, where the exact elements appearing in an instance (or bad-instance) are not known in advance, and where the author may be unfamiliar with JavaScript.

6.3 Diagramming for Formal Methods

We note that multiple other formal methods tools also provide some form of diagramming. While in some cases very powerful, we are not aware of any of them providing *lightweight* diagramming in the manner of `CnD`.

Pro B. The Pro B [21] suite provides multiple tools (e.g., BMotionWeb [19] and VisB [47]) for visualizing traces. Moreover the Alloy2B [18] project can translate a rich subset of Alloy to B, meaning that Pro B can be used for some Alloy visualizations. For example, BMotionWeb [19] enables interactive, domain-specific visualizations. Creating these visualizations requires some knowledge of web programming. In VisB [47], users modify attributes of a base SVG image by linking elements of the image to aspects of the underlying spec.

Lean. The Lean [27] proof assistant provides domain-specific visualizations for structures like graphs, trees, etc. [25, 31] and for proofs as well [50]. Creating new visualizations with these libraries requires either familiarity with JavaScript (as Sterling does) or with a diagramming tool such as Penrose (Section 6.2).

GUPU. GUPU [33] is a pedagogic tool for visualizing Prolog substitution via domain-specific *viewers*. Students can use these graphical tools alongside tailored feedback and testing capabilities to better understand Prolog solutions. GUPU's viewers are written in Prolog [34], and are capable of generating sophisticated diagrams. Unlike `CnD` specs, however, GUPU viewers are heavyweight. Authors must invest significant time to get started, specifying each diagram construct, the minutiae of layout, and reconstructing the relationships between diagram elements. This is a significant barrier to entry: custom viewers presented by the GUPU authors [34] are Prolog programs that are much more complex than the Prolog programs they visualize.

6.4 Layout Tools

`CnD` builds heavily on two libraries for layout:

- WebCola [8] is a JavaScript library designed to support constraint-based graph layout [9]. WebCola constraints are specified in terms of a graph's nodes, and can be used to control node alignment, separation, and grouping. These constraints, however, are always "soft". This means that WebCola will always generate a layout, even if the constraints are not satisfied. Nevertheless, WebCola is still useful for layout, so `CnD` uses WebCola *after* constraints are deemed consistent and satisfiable (see Section 3.2).

- DAGRE is a JavaScript library for generating directed graph layouts, organizing nodes into hierarchical layers to minimize edge crossings and improve readability [36]. While DAGRE allows users to exercise some control over graph layout (e.g., specifying the direction of graph flow and node separation), users are unable to control the layout of individual nodes or edges. DAGRE's focus on hierarchical layout means that it is not well suited to more general graph layouts (e.g., cyclic graphs). Taking inspiration from Sterling, `CnD` uses DAGRE if no layout constraints are specified.

7 Limitations

Since `CnD` acts as a refinement on default-graphs, it inherits many of the same limitations. For instance, layout constraints may not be applicable to non-functional, non-hierarchical relations. Consider the example of an undirected tree, where each edge between nodes takes the form of a symmetric relation (Figure 9). Any orientation constraint on symmetric relations is necessarily inconsistent, so `CnD` cannot enforce a layout based on tree edges.

Furthermore, `CnD` constraints cannot add *new* information not reflected in the default-graph (though pictorial directives can). The instance in Figure 9 does not contain any information on which node is the parent of another. Thus, `CnD` alone cannot dictate which node is visualized as the root of the tree. Doing so without building a custom visualization requires somehow indicating the root the tree by, for instance, a field or sig in the spec.

A silver lining here, however, is that the worst-case scenario for `CnD` is the default-graph. As we have shown in Section 3.5, even in the absence of any `CnD` primitives, its output likely leads to less cluttered, more informative diagrams than the default-graph. Thus, `CnD` diagrams should not be less useful than default-graphs.

8 Discussion

Once we strip away the finery, the effectiveness of many custom visualizations boils down to a small number of key spatial relationships. We found that these relationships can be captured by a small set of primitives, which we encoded into a lightweight, declarative language. We discuss some of the lessons we have learned from this work below.

"Useful but not Pretty" diagrams. While many visualization tools and frameworks focus on drawing, with an emphasis on high visual quality, much of the value of a diagram comes from its ability to preserve key semantic relationships. Cope-and-Drag diagrams are often less visually appealing than those generated by D3 or Penrose, but help effectively communicate the model's structure while avoiding the pitfalls of focusing too much on aesthetic concerns.

Diagramming by Refinement: Specifications versus Programs. There is a fundamental divide between the practices of writing programs and writing specs. In personal communication, Daniel N. Jackson (the lead architect of Alloy) once explained to the last author the difference between a program and a spec: the empty program exhibits no behaviors, while the empty spec admits all behaviors. Roughly speaking, growing a program typically leads to additional behaviors, whereas growing a spec (by writing constraints) typically leads to fewer behaviors.

We believe a similar divide applies to diagramming. The Alloy Visualizer behaves like a spec: the default visualizer shows everything. As this paper has noted, this genericity and domain-independence causes problems (Section 2.2) but also has virtues (Section 4). At

the other extreme are systems like D3. By providing full control, they can address some of the weaknesses we have noted (Sections 6.1 and 7). However, the user gets nothing for free: every piece of output requires programmer effort. That puts them on the program end of the program–spec split. Systems like Penrose also fall at this end, even though they provide more structure to the program and thus enable some separation of concerns.

In this typology, `CnD` is "spec," not "program." It consciously alters the default-graph output, with the empty `CnD` program leaving the output unchanged. Adding constraints and directives shapes and sometimes limits the amount of output and the number of specs or instances that can be displayed. Thinking of diagramming as spatial refinement also provides design insight, both into where a `CnD`-like spec can live (Section 3.4) and how errors can be handled by analogy to types (Section 3.2).

Floors and Ceilings: The Value of Lightweightness. Another useful typology is that introduced by the Scratch programming language [23], of *floors* and *ceilings*. Scratch presents itself as having a low floor and high ceiling: it's very easy to start programming but there is little limit to how sophisticated a program one can write. Without dwelling on Scratch's claims, we can view the systems we have discussed through the same lens. Default-graphs have the lowest floor of all – one needs to do nothing to obtain output – but also have a very low ceiling. Systems like D3 and Penrose have an arbitrarily high ceiling, but also an elevated floor (which in D3 is known to cause difficulties for some users; we are not aware of much evidence either way about Penrose). We would classify `CnD` as having the lowest of floors (since the program can be empty, still producing default-graph output), but also a moderate ceiling: it does some things very well, but other things (Section 7) poorly relative to a custom visualization.

Showing Erroneous Output. In Section 3.2 we discuss how `CnD` handles errors. In particular, in both cases, it refuses to produce a diagram. This avoids creating output that might mislead the user. However, it is unclear that this option is ideal. The entire premise of this paper is that seeing visual output is useful for knowledge discovery, including for spotting problems. Similarly, there are situations where producing "bad" output can quickly help a user spot why a bug in the *`CnD` program* (or, more subtly, its relationship to the spec). Providing no output at all cuts off that avenue.

For this reason, we believe it would be worthwhile to investigate a mode where we produce visual output even in the case of errors, but make salient which elements should have prevented any output from occurring. We believe there are important algorithmic, cognitive, and presentational considerations. Algorithmically, we have constraints that cannot be satisfied; we have to somehow identify constraints to break to even generate output, and then label which parts of the output are invalid. Cognitively, there are surely factors that influence which constraints to choose (which may not always be the "smallest"). Presentationally, we have to both make salient that the user is viewing an invalid instance and highlight which parts of are invalid.

Accessibility for the Visually-Impaired. Like the vast majority of data-visualization tools, the Alloy Visualizer relies primarily on modalities that are inaccessible to visually-impaired users [13]. `CnD` does not reduce the accessibility of the Alloy Visualizer. However, the encoding of essential spatial relationships in `CnD` specifications presents an opportunity to improve how visually-impaired users are able to interact with default output. In particular, the `CnD` constraints may lead to improved output: e.g., exploiting spatial audio effects for sonification [4, 35, 52].

Conclusion. `CnD` is not meant to be a last word in the diagramming conversation, but rather an interesting point in the design space. It is unclear how to make the ceiling for `CnD` much higher, at least not without "lifting" the floor. This is very difficult in the current design (which consciously modifies default-graph output) and may cause problems such as masking bad-instances. Instead, we believe there is value to having other languages that can provide other floor–ceiling ranges.

References

1 Greg J Badros, Alan Borning, and Peter J Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306, 2001. doi:10.1145/504704.504705.

2 Leilani Battle, Danni Feng, and Kelli Webber. Exploring D3 implementation challenges on Stack Overflow. In *2022 IEEE Visualization and Visual Analytics (VIS)*, pages 1–5. IEEE, 2022. doi:10.1109/VIS54862.2022.00009.

3 Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011. doi:10.1109/TVCG.2011.185.

4 Stanley J Cantrell, Bruce N Walker, and Øystein Moseng. Highcharts sonification studio: an online, open-source, extensible, and accessible data sonification tool. In *Proceedings of the International Conference on Auditory Display*, volume 2, 2021.

5 Hsuanwei Michelle Chen. Information visualization principles, techniques, and software. *Library Technology Reports*, 53(3):8–16, 2017.

6 Rui Couto, José Creissac Campos, Nuno Macedo, and Alcino Cunha. Improving the visualization of Alloy instances. In Paolo Masci, Rosemary Monahan, and Virgile Prevosto, editors, *Proceedings 4th Workshop on Formal Integrated Development Environment*, volume 284, pages 37–52, 2018. doi:10.4204/EPTCS.284.4.

7 Edsger W Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974. doi:10.1145/361179.361202.

8 Tim Dwyer. cola.js: Constraint-based layout in the browser, 2017. Accessed: 2024-12-02. URL: https://ialab.it.monash.edu/webcola/.

9 Tim Dwyer, Kim Marriott, and Michael Wybrow. Topology preserving constrained graph layout. In *Graph Drawing: 16th International Symposium, GD 2008, Heraklion, Crete, Greece, September 21-24, 2008. Revised Papers 16*, pages 230–241. Springer, 2009. doi:10.1007/978-3-642-00219-9_22.

10 Tristan Dyer. Sterling JS demo, 2024. Accessed: 2024-11-12. URL: https://sterling-js.github.io/demo/.

11 Tristan Dyer and John Baugh. Sterling: A web-based visualizer for relational modeling languages. In *International Conference on Rigorous State-Based Methods*, pages 99–104. Springer, 2021.

12 Tristan Dyer, Tim Nelson, Kathi Fisler, and Shriram Krishnamurthi. Applying cognitive principles to model-finding output: the positive value of negative information. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–29, 2022. doi:10.1145/3527323.

13 Frank Elavsky, Lucas Nadolskis, and Dominik Moritz. Data navigator: an accessibility-centered data navigation toolkit. *IEEE transactions on visualization and computer graphics*, 30(1):803–813, 2023. doi:10.1109/TVCG.2023.3327393.

14 Robert Goldstone. An efficient method for obtaining similarity data. *Behavior Research Methods, Instruments, & Computers*, 26:381–386, 1994.

15 D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, pages 21–22, April 1996.

16 Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2nd edition, 2012.

17 Kurt Koffka. Perception: an introduction to the Gestalt-Theorie. *Psychological bulletin*, 19(10), 1922.

18 Sebastian Krings, Joshua Schmidt, Carola Brings, Marc Frappier, and Michael Leuschel. A translation from Alloy to B. In *Conference on Abstract State Machines, Alloy, B, and Z*, pages 71–86, 2018. doi:10.1007/978-3-319-91271-4_6.

19 Lukas Ladenberger and Michael Leuschel. BMotionWeb: A tool for rapid creation of formal prototypes. In *Software Engineering and Formal Methods*, pages 403–417, 2016. doi:10.1007/978-3-319-41591-8_27.

20 Jill H Larkin and Herbert A Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive science*, 11(1):65–100, 1987. doi:10.1111/J.1551-6708.1987.TB00863.X.

21 Michael Leuschel and Michael Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *International Symposium on Formal Methods (FM)*, 2003. doi:10.1007/978-3-540-45236-2_46.

22 Dor Ma'ayan, Wode Ni, Katherine Ye, Chinmay Kulkarni, and Joshua Sunshine. How domain experts create conceptual diagrams and implications for tool design. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2020. doi:10.1145/3313831.3376253.

23 John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The Scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):1–15, 2010. doi:10.1145/1868358.1868363.

24 Niloofar Mansoor, Hamid Bagheri, Eunsuk Kang, and Bonita Sharif. An empirical study assessing software modeling in Alloy. In *2023 IEEE/ACM 11th International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 44–54. IEEE, 2023. doi:10.1109/FORMALISE58978.2023.00013.

25 Lean Manual. The user-widgets system. https://lean-lang.org/lean4/doc/examples/widgets.lean.html, 2024. [Accessed Nov 19, 2024].

26 Vajih Montaghami and Derek Rayside. Bordeaux: A tool for thinking outside the box. In *Fundamental Approaches to Software Engineering: 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 20*, pages 22–39. Springer, 2017. doi:10.1007/978-3-662-54494-5_2.

27 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pages 625–635. Springer International Publishing, 2021. doi:10.1007/978-3-030-79876-5_37.

28 Lekha Nair, Sujala Shetty, and Siddhanth Shetty. Interactive visual analytics on big data: Tableau vs D3.js. *Journal of e-Learning and Knowledge Society*, 12(4), 2016.

29 Mitchell J. Nathan, Kenneth R. Koedinger, and Martha W. Alibali. Expert blind spot: When content knowledge eclipses pedagogical content knowledge. In *Proceedings of the Third International Conference on Cognitive Science*, pages 644–648, 2001. URL: https://api.semanticscholar.org/CorpusID:14779203.

30 Mitchell J Nathan and Anthony Petrosino. Expert blind spot among preservice teachers. *American Educational Research Journal*, 40(4):905–928, 2003.

31 Wojciech Nawrocki, Edward W. Ayers, and Gabriel Ebner. An extensible user interface for Lean 4. In Adam Naumowicz and René Thiemann, editors, *Interactive Theorem Proving*, volume 268 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:20, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITP.2023.24.

32 Tim Nelson, Ben Greenman, Siddhartha Prasad, Tristan Dyer, Ethan Bove, Qianfan Chen, Charles Cutting, Thomas Del Vecchio, Sidney LeVine, Julianne Rudner, Ben Ryjikov, Alexander Varga, Andrew Wagner, Luke West, and Shriram Krishnamurthi. Forge: A tool and language for teaching formal methods. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):613–641, 2024. doi:10.1145/3649833.

33 Ulrich Neumerkel and Stefan Kral. Declarative program development in Prolog with GUPU. *arXiv preprint cs/0207044*, 2002.

34 Ulrich Neumerkel, Christoph Rettig, and Christian Schallart. Visualizing solutions with viewers. In *LPE*, pages 43–50, 1997.

35 Sandra Pauletto and Andy Hunt. A toolkit for interactive sonification. In *ICAD*, 2004.

36 Chris Pettitt and contributors. Dagre: A JavaScript library for directed graph layouts. `https://github.com/dagrejs/dagre`, 2014. Accessed: 2024-11-21.

37 Josh Pollock, Catherine Mei, Grace Huang, Elliot Evans, Daniel Jackson, and Arvind Satyanarayan. Bluefish: Composing diagrams with declarative relations. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, pages 1–21, 2024. `doi:10.1145/3654777.3676465`.

38 Siddhartha Prasad, Ben Greenman, Tim Nelson, and Shriram Krishnamurthi. sidprasad/-copeanddrag. Software, swhId: `swh:1:dir:b92dbb01ca5fcbe40ac64c8026fb3076fbf85d2a` (visited on 2025-06-16). URL: `https://github.com/sidprasad/copeanddrag/tree/ecoop-25`, `doi:10.4230/artifacts.23608`.

39 Helen Purchase. Which aesthetic has the greatest effect on human understanding? In *International Symposium on Graph Drawing*, pages 248–261. Springer, 1997. `doi:10.1007/3-540-63938-1_67`.

40 Derek Rayside, Felix Sheng-Ho Chang, Greg Dennis, Robert Seater, and Daniel Jackson. Automatic visualization of relational logic models. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, 7, 2007. `doi:10.14279/tuj.eceasst.7.94`.

41 Lothar Spillmann, editor. *On Perceived Motion and Figural Organization*. MIT Press, 2012. Includes English translations of Max Wertheimer's 1912 and 1923 papers on Gestalt psychology.

42 Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM computer communication review*, 31(4):149–160, 2001. `doi:10.1145/383059.383071`.

43 J Ridley Stroop. Studies of interference in serial verbal reactions. *Journal of experimental psychology*, 18(6):643, 1935.

44 John Sweller and Paul Chandler. Evidence for cognitive load theory. *Cognition and instruction*, 8(4):351–362, 1991.

45 Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, 1983.

46 Barbara Tversky. Spatial schemas in depictions. In M. Gattis, editor, *Spatial schemas and abstract thought*, pages 79–112. The MIT Press, 2001.

47 Michelle Werth and Michael Leuschel. VisB: A lightweight tool to visualize formal models with SVG graphics. In *Rigorous State Based Methods*, pages 260–265, 2020. `doi:10.1007/978-3-030-48077-6_21`.

48 Max Wertheimer. Experimentelle studien über das sehen von bewegung. *Zeitschrift für Psychologie*, 61:161–265, 1912.

49 Benjamin W White. Interference in identifying attributes and attribute names. *Perception & Psychophysics*, 6:166–168, 1969.

50 David Wren. animate-lean-proofs, 2021. Accessed: 2024-11-22. URL: `https://github.com/dwrensha/animate-lean-proofs`.

51 Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. Penrose: from mathematical notation to beautiful diagrams. *ACM Transactions on Graphics (TOG)*, 39(4):144–1, 2020.

52 Jonathan Zong, Isabella Pedraza Pineros, Mengzhu Chen, Daniel Hajas, and Arvind Satyanarayan. Umwelt: Accessible structured editing of multi-modal data representations. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, pages 1–20, 2024.

PoTo: A Hybrid Andersen's Points-To Analysis for Python

Ingkarat Rak-amnouykit
Rensselaer Polytechnic Institute, Troy, NY, USA

Ana Milanova
Rensselaer Polytechnic Institute, Troy, NY, USA

Guillaume Baudart
Université Paris Cité, Inria, CNRS, IRIF, France

Martin Hirzel
IBM Research, Yorktown Heights, NY, USA

Julian Dolby
IBM Research, Yorktown Heights, NY, USA

Abstract

As Python is increasingly being adopted for large and complex programs, the importance of static analysis for Python (such as type inference) grows. Unfortunately, static analysis for Python remains a challenging task due to its dynamic language features and its abundant external libraries. To help fill this gap, this paper presents PoTo, an Andersen-style context-insensitive and flow-insensitive points-to analysis for Python. PoTo addresses Python-specific challenges and works for large programs via a novel hybrid evaluation, integrating traditional static points-to analysis with concrete evaluation in the Python interpreter for external library calls. We evaluate the analysis with two clients: type inference and call graph construction. This paper presents PoTo+, a static type inference for Python built on PoTo. We evaluate PoTo+ and compare it to two state-of-the-art Python type inference techniques: (1) the static rule-based Pytype and (2) the deep-learning based DLInfer. Our results show that PoTo+ outperforms both Pytype and DLInfer on existing Python packages. This paper also presents PoToCG, a call-graph construction analysis for Python built on PoTo. We compare PoToCG to PyCG, the state of the art for this problem, and show that PoTo produces more complete and more precise call graphs.

2012 ACM Subject Classification Software and its engineering → General programming languages

Keywords and phrases Python, Points-to analysis, Machine learning libraries

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.27

Supplementary Material *Software*: `https://github.com/Ingkarat/PoTo`

Funding US National Science Foundation for supporting our work through awards #1814898 and #2232061.

Acknowledgements We thank the ECOOP reviewers for their valuable comments.

1 Introduction

Points-to analysis is a fundamental static analysis that determines what objects a reference variable may point to. It has a wide variety of applications (aka. *client* analyses), including call graph construction, type inference, debugging, and security vulnerability detection. Essentially all interesting questions one could ask about a program require some form of points-to information and points-to analysis has been studied extensively in the context of mainstream languages such as C, C++, Java, and JavaScript. Surprisingly, points-to analysis for Python has received less attention, despite Python's popularity.

© Ingkarat Rak-amnouykit, Ana Milanova, Guillaume Baudart, Martin Hirzel, and Julian Dolby;
licensed under Creative Commons License CC-BY 4.0

This paper presents PoTo, an Andersen-like point-to analysis for Python. Andersen's analysis [2] is a classical flow-insensitive, context-insensitive, and inclusion-constraint-based analysis. It computes a points-to graph Pt where the nodes are reference variables and heap objects, and the edges represent points-to relations. For example, an edge $x \rightarrow o$ represents that x points to heap object o (i.e., reference x stores the address of o), and $o_1 \xrightarrow{f} o_2$ represents that field f of o_1 points to o_2.

While PoTo leverages well-known techniques, it also adapts to the unique challenges of Python: complex syntax and module system, dynamic semantics, and ubiquitous use of external libraries whose code is unavailable. First, our solution presents principled translation from Python source into the 3-address code intermediate representation demanded by points-to analysis. It specifies explicitly which Python features are handled precisely and faithfully to the feature semantics and which are handled approximately via a default interpretation; due to the complexity of Python, handling is unsound. Second, a novelty of PoTo is *hybridization*. We observe that many expressions, particularly ones stemming from external libraries and built-in functions, are available for *concrete evaluation* but are unavailable for standard *abstract evaluation* by virtue of their code being unavailable. Therefore we *concretely evaluate* the expressions and propagate the concrete objects through the points-to graph. This yields more information during inclusion constraint resolution and improves program coverage.

We evaluate PoTo on concrete type inference and call graph construction. We choose type inference because there has been significant interest in the problem in recent years. In addition to traditional approaches such as Pytype [12], there has been significant advance in deep-learning-based approaches, e.g., [1, 30, 46]. PoTo+ is the type inference client that largely draws types from PoTo's result: given $Pt(x)$ computed by the points-to analysis, the type of x is the union of the types of objects $o \in Pt(x)$. We compare PoTo+ against (1) Pytype [12], arguably the most advanced rule-based type inference tool, and (2) DLInfer [46], a state-of-the-art deep-learning-based tool, on a benchmark suite of Python packages ranging from 3,556 to 285,515 lines of code. Our results show that PoTo+ is comparable to Pytype and both techniques outperform DLInfer in terms of coverage (i.e., percentage of variables for which a type is reported) and correctness of inferred types. Furthermore, PoTo scales better than Pytype. Call graph construction is a classical client of points-to analysis. We build PoToCG based on PoTo and compare it to PyCG [34], the leading call graph construction analysis for Python. The results show that PoToCG constructs more complete and more precise call graphs compared to PyCG and it scales better.

The contributions of our work are as follows:

- PoTo, the first Andersen-style points-to analysis for Python.
- Hybridization weaving concrete and abstract evaluation.
- Evaluation on two clients of points-to analysis: type inference and call graph construction.

Ultimately, we hope that this work contributes to tools that make Python developers more productive and Python code more robust.

2 Problem Statement and Overview

The problem at hand is to design an Andersen-style points-to analysis for Python and *scale the analysis to large real-world Python packages*. Andersen's points-to analysis is a classical static analysis problem. It is a whole-program flow-insensitive and context-insensitive analysis that tracks flow of values via inclusion constraints. E.g., an assignment statement `x = y` triggers an inclusion constraint $Pt(\mathtt{y}) \subseteq Pt(\mathtt{x})$ indicating that the points-to set of `y` flows to the points-to set of `x`.

There are numerous challenges for Andersen's analysis and more generally static analysis for Python. First, Andersen's analysis is defined over a 3-address code intermediate representation (IR) serving as the foundation for the inclusion constraints. Translation from high-level C, C++, Java, and JavaScript code into 3-address code is well-studied and there are mature tools that provide the translation, notably LLVM [20], Soot [41], WALA [17], and Doop [36]. Surprisingly, static analysis for Python is largely ad-hoc AST-based analysis with each new work (e.g., [12, 46]) embedding its own interpretation of Python AST constructs and translation (if any) of those constructs into a 3-address code IR. Translating Python AST constructs into a 3-address code IR is challenging due to (1) Python's complex syntax and dynamic semantics and (2) Python's rich module system and scoping issues arising from it. Python allows for reflection and metaprogramming, subclassing conditioned on runtime values, complex With statements and context managers, and many other features [47] that are difficult or impossible to handle soundly in a static semantics. Even a construct such as Subscript, e.g., `a[index_expr]`, is non-trivial in the sense that its canonical interpretation as element access of a list-like structure is unsound (we choose the canonical interpretation for both Subscript and Attribute; a more faithful but also more complex interpretation would be to treat `__getitem__` and `__getattr__` as virtual calls). A principled analysis therefore must choose a subset of constructs to interpret precisely. We present a minimal syntax that specifies a subset of Python, and for that subset we present a precise interpretation that is faithful to the Python construct's semantics. All other Python AST constructs are abstracted by a fall-through structure *Other* and receive a default interpretation that can be both over-approximate and unsound. Going forward we sometimes refer to precisely-interpreted constructs as *interpreted* and to *Other* as *uninterpreted*. A key goal is to describe the points-to analysis as completely as possible, grounding the syntax and specifying interpretation semantics for both kinds of constructs, while at the same time abstracting away unnecessary detail. Section 3 specifies interpreted and uninterpreted constructs and their semantics, and makes it clear that we support memory-accesses and nested functions. It also explicitly lists the features under *Other*, e.g., generator expressions and With statements.

Another challenge is use of external libraries. Whole-program static analysis generally assumes that library code is available. However, Python has a particularly extensive ecosystem of open-source libraries; it is a dynamically-typed language where library interfaces usually lack type annotations; Python libraries and their interfaces are often rapidly evolving; and libraries are often at least partially implemented in C instead of Python. We address this with a novel hybridization technique that weaves concrete and abstract evaluation during 3-address code generation and constraint resolution.

While to the best of our knowledge we are the first to present a points-to analysis for Python, the problem has been studied extensively in mainstream languages, notably C, C++, Java, and JavaScript. (We detail related work in Section 6.) JavaScript is a dynamically typed language, as is Python. However, JavaScript poses unique challenges and program analysis aims to address these challenges: event-driven nature of applications, interaction with the DOM, runtime construction of object property strings, and runtime creation and deletion of object properties. None of these is a significant issue for Python, which increasingly is a general-purpose language used to build libraries and applications similarly to Java. We therefore model Andersen's analysis after Java's analysis, extracting and analyzing a Java-like subset of Python. Python presents new challenges though, including lack of a 3-address code IR, missing external libraries, and first-class functions and classes.

Consider the example in Figure 1, adapted from DLInfer, a recent paper on neural type inference for Python. Our analysis has two core phases (PoTo), followed by a client phase:

```python
import re

def url_regex():
  regex_cache = None
  if regex_cache is None:
    regex_cache = re.compile(r"p"
        )
  return regex_cache

def str_validator(value):
  if isinstance(value,str):
    return value.value
  else:
    return value

def validate(value):
  url = str_validator(value)
  m = url_regex().match(url)
  if m.end() != len(url):
    raise Exception(m.end())

def main():
  validate("p abcd")
```

(a) Python source.

```python
import re # External import

def url_regex():
  regex_cache = None
  regex_cache=(const,<class 're.Pattern'>,...)
  url_regex_ret = regex_cache

def str_validator(value):
  str_validator_ret = value.value
  str_validator_ret = value

def validate(value):
  url = str_validator(value)
  t1 = url_regex()
  t2 = t1.match
  m = t2(url)
  t3 = m.end
  t4 = t3()
  t5 = len(url)
  t6 = (const,<class 'Exception'>,...)
  t7 = m.end
  t8 = t7()
  t9 = t8(t7)
```

(b) 3-address code for (a), excluding `main`.

Figure 1 Illustrating example, adapted from DLInfer [46].

1. $x = \text{object}$ *new assignment*
2. $x = y$ *copy propagation*
3. $x.f = y$ *field write*
4. $x = y.f$ *field read*
5. $x = y(z)$ *closure call*

Figure 2 3-address code statements.

Phase 1: Python source ⟶ 3-address code
Phase 2: 3-address code ⟶ Points-to graph
Phase 3 (client): Points-to graph ⟶ Concrete type inference or call graph

Phase 1 takes as input Python source code and produces 3-address code; this phase works at the granularity of a function. Phase 2 processes the 3-address statements as inclusion constraints computing the points-to graph. The two phases are intertwined – roughly, the analysis starts at the main function and immediately invokes Phase 1 on main generating 3-address code for main; it then invokes Phase 2 to solve the 3-address code for main. As new functions become reachable, the analysis invokes 3-address code generation, then it proceeds to solve the constraints until the points-to graph reaches a fixpoint. As expected, one can implement many client analyses on top of the points-to results. Phase 3 focuses on concrete (i.e., non-polymorphic) type inference and call graphs as client analyses.

The first phase does principled translation of Python AST constructs into the standard 3-address statements shown in Figure 2. For example, the Python source in Figure 1(a) Line 17 translates into the 3-address sequence of statements in Figure 1(b) Lines 14–16. A

novelty in our treatment is the weaving of concrete evaluation into 3-address code translation and later constraint resolution. During translation, the analysis concretely evaluates every expression in its enclosing import environment. If evaluation succeeds, translation returns the resulting constant; otherwise, it proceeds recursively and returns the corresponding 3-address code statements. For example, in Figure 1(a) Line 6, the right-hand-side evaluates into a concrete object: `(const,<class 're.Pattern'>,...)` in Figure 1(b) Line 5 (we write a concrete object as a triple of `const`, the object type, and its value, with the value typically elided). As is standard for flow-insensitive analysis, the translation ignores control flow and basic blocks. Each function in Figure 1(b) is a straight-line sequence of 3-address statements. Suffix `_ret` indicates the function's return value (e.g., `str_validator_ret`).

The second phase of the analysis solves the 3-address statements as inclusion constraints, incorporating (1) class and function objects as first-class values, and (2) concrete evaluation. The analysis maintains *abstract objects* (constructed from code in the package under analysis) and *concrete objects* (constructed from concrete evaluation). Abstract objects, in turn, fall into three categories: *meta-class* objects, *meta-func* objects (more precisely closure objects), and *data* objects. The analysis maintains an abstract reference environment, which is the package-under-analysis environment, and uses it to resolve names defined in the package.

When processing a call or a field access statement, the analysis retrieves each object in the points-to set of the receiver variable. Roughly speaking, if the function object at the call is an abstract one, the analysis proceeds with abstract evaluation. If it is a concrete one, it attempts concrete evaluation. For example, in `url = str_validator(value)`, the analysis examines the points-to set of `str_validator` and retrieves the abstract function object representing the `str_validator` function. This triggers abstract evaluation and (standard) addition of the points-to set of `str_validator_ret` (the special return variable) to the points-to set of the left-hand-side variable `url`. Eventually, the points-to set of `url` becomes {`(const, <class 'str'>, 'p abcd')`}. Concrete evaluation of `t2 = t1.match` returns the concrete closure object corresponding to the match function, and finally concrete evaluation of `m = t2(url)` returns a concrete `re.Match` object. The hybrid analysis infers concrete types for identifiers `url` and `m`, respectively `<class 'str'>` and `<class 're.Match'>`, while Pytype reports only the fall-through type Any for both.

3 Syntax and Semantics

Recent work defines syntax for a subset of Python and a corresponding interpretation semantics for the purpose of weakest precondition inference [32]. A notable idea in this work is the separation of Python constructs into interpreted and uninterpreted ones. In this paper, we follow the idea of separating constructs into interpreted and *Other* but differ in important ways as our goal is interprocedural flow- and context-insensitive points-to analysis rather than weakest precondition inference. Unlike [32], we track object creation and flow of values and construct a call graph on the fly. Flow-insensitive points-to analysis demands a different set of interpreted constructs and a different interpretation.

Section 3.1 defines the syntax for the purposes of flow-insensitive points-to analysis, Section 3.2 defines the interpretation that generates 3-address code, and Section 3.3 presents Andersen-style constraint resolution, highlighting Python-specific semantics.

3.1 A Minimal Python Syntax

Figure 3 specifies the syntax which grounds the analysis. An expression e can be a constant c (`42` or `"foo"`), a variable (`x`), an attribute access (`x.foo`), a subscript access (`x["bar"]`), a list, a tuple, a dictionary, a list comprehension (`[2*x for x in range(10)]`), a dictionary

e ::= $c \mid x \mid e.x \mid e[e] \mid e(e,...,e)$	const | Name | Attribute | Subscript | Call
$\mid [e,...,e] \mid \{e\text{: }e,...,e\text{: }e\} \mid (e,...,e)$	List | Dictionary | Tuple
$\mid [e \text{ for } x,...,x \text{ in } e \text{ if } e]$	ListComp
$\mid \{e\text{:}e \text{ for } x,...,x \text{ in } e \text{ if } e\}$	DictComp
$\mid e \ op \ e \mid e \ cop \ e$	BinOp | Compare
$\mid Other\ (e,...,e)$	*All other Python AST expressions*
s ::= pass $\mid x = e \mid e.x = e \mid e[e] = e$	Pass | Assign
$\mid s \ ; \ s \mid$ for e in e: s	Suite | For
$\mid$ def $f(x,...,x)$: s ; return e	FunctionDef | Return
$\mid$ class $C(e,...,e)$: s	ClassDef
$\mid Other\ (s,...,s)$	*All other Python AST statements*
i ::= import p (as x)? | from p import x (as x)?	Import | ImportFrom
$\mid i \ ; \ i$	
m::= $i \ ; \ s$	Module

Figure 3 Syntax of a subset of Python. The non-terminal names on the right are based on the official Python abstract syntax tree (AST); e.g., the AST node for a statement sequence is Suite.

comprehension (`{x:f(x)for x in range(x)}`), a binary operation or a compare operation. All other Python expressions are under *Other* $(e_1,...,e_n)$ and receive the default interpretation; their AST nodes are BoolOp, NamedExpr, UnaryOp, Lambda, IfExp, Set, SetComp, GeneratorExp, Await, Yield, YieldFrom, FormattedValue, JoinedStr, Starred, and Slice.

A statement s can be pass, the assignment of either a variable (`x = 42`), an attribute (`x.foo = 42`), or a subscript (`x["bar"] = 42`), a sequence, a loop, a function definition, or a class definition. All other Python statements are captured under *Other* $(s_1,...,s_n)$ and receive the default interpretation; their AST nodes are Delete, TypeAlias, AugAssign, AnnAssign, AsyncFor, While, If, With, AsyncWith, Match, Raise, Try, TryStar, Assert, Global, Nonlocal, Break, and Continue. This default interpretation is often all that is needed. For example, Break and Continue are irrelevant to flow-insensitive analysis. As another example, for If statements, the default interpretation descends into the AST node of the children and processes them. As illustrated in Figure 1, this is adequate for flow-insensitive analysis. To simplify the presentation, we leave out the syntax of complex left-hand-side expressions in assignments (e.g., `a, *(b, c)= (1, (2, 3), 4)`), variable-length arguments, and keywords arguments. We use deconstruct-and-bind preprocessing to reduce these into sequences of assignments as in Figure 3. For example, `a, *(b, c)= (1, (2, 3), 4)` reduces into a sequence including `a = 1`, `b = (2, 3)` and `c = 4`.

A module m starts with a sequence of import i, import p or from p import x, with an optional alias name (as x), followed by a statement. A package under analysis is a sequence of modules.

3.2 3-address Code Generation

3-address-code translation takes as input the Python source (i.e., AST) of a function and produces a sequence of 3-address-code statements of the form in Figure 2.

```
# Initialize Γ0:
Γ0 = [], Φ = {}
for ⟨module M: i; s⟩ in package under analysis
    for ⟨class C(...): ...⟩ in s: Γ0 ← [(M.C, t)] + Γ0, t is fresh
    for ⟨def f(...): ...⟩ in s: Γ0 ← [(M.f, t)] + Γ0, t is fresh
    # Imports from p import x' as x are implicit assignments
    for x = ... in s: Γ0 ← [(M.x, t)] + Γ0, t is fresh
# Next, compute class hierarchy H and MRO:
H ← C3(Γ0) # H maps (⟨class C(...): ...⟩, f) to ⟨def f(...): ...⟩
# Interpret main and add to worklist:
Φ[⟨def main(...): s⟩] ← I(s, [])
W ← {⟨def main(...): s⟩} # Entry point
# Interpret module initializers and add to worklist:
for ⟨module M: ...⟩ in package under analysis
    Φ[⟨def M.module_init(...): i; s⟩] ← I(i; s, [])
    W ← {⟨def M.module_init(...): i; s⟩} # Entry points
# Solve constraints in reachable functions:
while W ≠ ∅
   ⟨def f(...): ...⟩ ← remove function from W
    for c in Φ[⟨def f(...): ...⟩][1]: W ← W + c.solve()
```

Figure 4 Worklist algorithm.

3.2.1 Environment and Interpretation Functions

The analysis interprets expressions and statements in an *abstract* reference environment split into two components: Γ and Γ_0. Γ is a *local* reference environment associated to the enclosing function. It is a map of (id,t) tuples where id is a local and t is the *analysis variable* representing the local. Γ_0 is the *global* reference environment mapping identifiers for module-level constructs to analysis variables representing those constructs. In addition, there is an *external* environment, Γ_{ext}, necessary for concrete evaluation. We analyze imports and group them into two categories: (1) internal imports are relative imports (e.g., `from ..metrics import get_scorer`) and ones referencing the analysis package, and (2) the remaining external imports (e.g., `import re`). External imports comprise Γ_{ext}.

The interpretation of a statement $\mathcal{I}(s, \Gamma)$ returns a pair (Γ', S) where Γ' is the augmented local environment resulting from the interpretation of s into 3-address code, and S is *the sequence of 3-address statements* corresponding to s. E.g., $\mathcal{I}(\mathsf{x} = \mathsf{y.f.g}, [(\mathsf{y}, \mathsf{t1})])$ returns the augmented environment [(x,t2), (y,t1)] and the following sequence of 3-address statements: t4 = t1.f; t3 = t4.g; t2 = t3. Here t1 and t2 are the analysis variables associated with local variables y and x respectively (in code examples we sometimes ignore analysis variables and use the source-level identifier directly). We leave Γ_0 and Γ_{ext} implicit in the writeup as they are uniquely defined for each statement under interpretation.

The interpretation of an expression $\mathcal{I}(e, \Gamma)$ returns a pair (V, S) where V is *a set of analysis variables* and S is the sequence of 3-address statements corresponding to e. E.g., $\mathcal{I}(\mathsf{y.f.g}, [(\mathsf{y}, \mathsf{t1})])$ returns fresh variable t3 and the sequence of 3-address statements t4 = t1.f; t3 = t4.g.

Φ is the environment of interpreted functions: a mapping from a function definition (Python AST FunctionDefs in the implementation) to a pair (Γ, S), where Γ is the local reference environment resulting from the translation of the function definition and S is the sequence of 3-address statements corresponding to the function body.

The analysis uses the worklist algorithm in Figure 4.

3.2.2 Global Environment Initialization

The analysis first initializes a global environment Γ_0 that contains mappings for identifiers of module-level constructs, $M.f$ (functions), $M.C$ (classes), and $M.x$ (identifier definitions) to their corresponding analysis variables. The scope of these constructs spans the entire package, hence the analysis augments the environment and makes these constructs available during 3-address code generation. This is similar to the *let rec* construct in functional programming which extends the scope for let-bound identifiers across all right-hand-side expressions. Initialization does not interpret functions, classes and right-hand-side of assignments; these are interpreted during reachability analysis.

3.2.3 Class Hierarchy Analysis

Next, the analysis computes the class hierarchy H using the C3 linearization algorithm for method resolution with multiple inheritance [4]. H is a mapping from a pair of class definition $\langle$class C(...): ...$\rangle$ and function name f to the corresponding function definition $\langle$def f(...): ...$\rangle$ resulting from a lookup in the *Method Resolution Order* (MRO) of C. Additionally, the step creates a (meta-cls, $\langle$class C(...): ...$\rangle$) object for each module-level class definition and associates the corresponding analysis variable t to that object. In other words, this step creates an initial set of points-to edges which allows for data object creation during the second phase of the analysis.

3.2.4 Iteration

As it is customary for whole-program analysis, the analysis starts from a main function. It translates main and all module initializers into 3-address code (the calls to $\mathcal{I}(s, [])$) and adds them to the worklist. The analysis removes a function from the worklist and processes the 3-address statements c for that function. The call $c.solve()$ solves the semantic constraints associated to c which has side effects on the 3-address code environment Φ and points-to graph Pt. A call statement triggers interpretation and placement on the worklist of the callee function and it is an invariant that when a function is removed from the worklist, its three address code is available in Φ.

The call $c.solve()$ returns a minimal set of functions that are affected by solving constraint c. For example, if c is the 3-address code statement t1 = t2 and solving it changes the points-to graph of t1, *solve* returns the enclosing function of the statement. A change in the points-to graph due to t1.f = t2 returns all functions in Φ, as the effect of new objects added to Pt(o.f), where o is an object in the points-to set of t1, may propagate to arbitrary functions.

3.2.5 Interpretation of statements and expressions

The definition of the interpretation function is presented in Figures 5 and 6. Interpretation of statements is in the abstract while interpretation of expressions weaves concrete and abstract evaluation. Below we highlight the most interesting points.

3.2.5.1 Statements

Consider interpretation of assignment $x = e$. If the variable x appears in a module initializer, we retrieve the analysis variable from Γ_0 (imports also create implicit global assignments). Otherwise, we only augment the local environment with a fresh variable if the variable x is not in the current environment. We target flow-insensitive points-to analysis, and thus, a Python sequence `x = 1; x = "a"` gives rise to 3-address code sequence t1 = 1; t1 = "a" and fails to distinguish that `x` is an integer at the first assignment and a string at the second.

$$
\begin{array}{lcl}
\mathcal{I}(\textsf{pass}, \Gamma) & = & \textsf{return } (\Gamma, \{\}) \\
\\
\mathcal{I}(x = e, \Gamma) & = & (R, S) \leftarrow \mathcal{I}(e, \Gamma) \\
 & & \text{if scope is } \mathit{M.module_init}: \\
 & & \quad \mathsf{t} \leftarrow \mathit{lookup}(M.x, \Gamma_0) \\
 & & \quad \textsf{return } (\Gamma, S \cup \{\mathsf{t} = t_e \mid t_e \in R\}) \\
 & & \text{if } x \in \Gamma: \\
 & & \quad \mathsf{t} \leftarrow \mathit{lookup}(x, \Gamma) \\
 & & \quad \textsf{return } (\Gamma, S \cup \{\mathsf{t} = t_e \mid t_e \in R\}) \\
 & & \text{else:} \\
 & & \quad \mathsf{t} \leftarrow \text{ fresh variable} \\
 & & \quad \Gamma' \leftarrow [(x, \mathsf{t})] + \Gamma \\
 & & \quad \textsf{return } (\Gamma', S \cup \{\mathsf{t} = t_e \mid t_e \in R\}) \\
\\
\mathcal{I}(s_1 \ ; \ s_2, \Gamma) & = & (\Gamma_1, S_1) \leftarrow \mathcal{I}(s_1, \Gamma) \\
 & & (\Gamma_2, S_2) \leftarrow \mathcal{I}(s_2, \Gamma_1) \\
 & & \textsf{return } (\Gamma_2, S_1 \cup S_2) \\
\\
\mathcal{I}(\textsf{for } x \textsf{ in } e\text{: } s, \Gamma) & = & \mathcal{I}(x = e \ ; \ s, \Gamma) \\
\\
\mathcal{I}(\textsf{def } f(\mathit{args})\text{: } \mathit{body}, \Gamma) & = & \text{if scope is } \mathit{M.module_init}: \\
 & & \quad \mathsf{t} \leftarrow \mathit{lookup}(M.f, \Gamma_0) \\
 & & \quad \textsf{return } (\Gamma, \{\mathsf{t} = (\textsf{meta-func}, \ \langle \textsf{def } f(\mathit{args})\text{: } \mathit{body} \rangle)\}) \\
 & & \text{else:} \\
 & & \quad \mathsf{t} \leftarrow \text{fresh variable} \\
 & & \quad \Gamma' \leftarrow [(f, \mathsf{t})] + \Gamma \\
 & & \quad \textsf{return } (\Gamma', \{\mathsf{t} = (\textsf{meta-func}, \ \langle \textsf{def } f(\mathit{args})\text{: } \mathit{body} \rangle)\}) \\
\\
\mathcal{I}(\textsf{class } C(\mathit{args})\text{: } \mathit{body}, \Gamma) & = & \textsf{return } (\Gamma, \{\}) \\
\\
\mathcal{I}(\mathit{Other}\ (s_1, ..., s_n), \Gamma) & = & (\Gamma_1, S_1) \leftarrow \mathcal{I}(s_1, \Gamma) \\
 & & ... \\
 & & (\Gamma_n, S_n) \leftarrow \mathcal{I}(s_n, \Gamma_{n-1}) \\
 & & \textsf{return } (\Gamma_n, S_1 \cup ... \cup S_n)
\end{array}
$$

Figure 5 From Python statements to 3-address-code. Given an environment Γ, the interpretation function for a statement $\mathcal{I}(s, \Gamma) = (\Gamma', S)$ returns an updated environment Γ' and the 3-address code S. M is the enclosing module, and Γ_0 is the global environment.

$$
\begin{aligned}
\mathcal{I}(x, \Gamma) \quad &= \quad \text{if } x \in \Gamma\text{: return } (\{lookup(x, \Gamma)\}, \{\}) \\
&\quad\; \text{if } M.x \in \Gamma_0\text{: return } (\{lookup(M.x, \Gamma_0)\}, \{\}) \\
&\quad\; \text{if } (\mathsf{const}, ty, v) \leftarrow eval(x, \Gamma_{ext})\text{:} \\
&\quad\;\; \text{return } (\{x\}, \{x = (\mathsf{const}, ty, v)\}) \\
&\quad\; \text{else: return } (\{\}, \{\}) \\
\mathcal{I}(e.f, \Gamma) \quad &= \quad \mathsf{t} \leftarrow \text{ fresh variable} \\
&\quad\; \text{if } (\mathsf{const}, ty, v) \leftarrow eval(e.f, \Gamma_{ext})\text{:} \\
&\quad\;\; \text{return } (\{\mathsf{t}\}, \{\mathsf{t} = (\mathsf{const}, ty, v)\}) \\
&\quad\; (V, S) \leftarrow \mathcal{I}(e, \Gamma) \\
&\quad\; \text{return } (\{\mathsf{t}\}, S \cup \{\mathsf{t} = t_e.f \mid t_e \in V\}) \\
\mathcal{I}(e_1[e_2], \Gamma) \quad &= \quad \mathsf{t} \leftarrow \text{ fresh variable} \\
&\quad\; \text{if } (\mathsf{const}, ty, v) \leftarrow eval(e_1[e_2], \Gamma_{ext})\text{:} \\
&\quad\;\; \text{return } (\{\mathsf{t}\}, \{\mathsf{t} = (\mathsf{const}, ty, v)\}) \\
&\quad\; (V_1, S_1) \leftarrow \mathcal{I}(e_1, \Gamma) \\
&\quad\; (V_2, S_2) \leftarrow \mathcal{I}(e_2, \Gamma) \\
&\quad\; \text{return } (\{\mathsf{t}\}, S_1 \cup S_2 \cup \{\mathsf{t} = t_1[] \mid t_1 \in V_1\}) \\
\mathcal{I}(e_1(e_2), \Gamma) \quad &= \quad \mathsf{t} \leftarrow \text{ fresh variable} \\
&\quad\; \text{if } (\mathsf{const}, ty, v) \leftarrow eval(e_1(e_2), \Gamma_{ext})\text{:} \\
&\quad\;\; \text{return } (\{\mathsf{t}\}, \{\mathsf{t} = (\mathsf{const}, ty, v)\}) \\
&\quad\; (V_1, S_1) \leftarrow \mathcal{I}(e_1, \Gamma) \\
&\quad\; (V_2, S_2) \leftarrow \mathcal{I}(e_2, \Gamma) \\
&\quad\; \text{return } (\{\mathsf{t}\}, S_1 \cup S_2 \cup \{\mathsf{t} = t_1(t_2) \mid t_1 \in V_1, t_2 \in V_2\}) \\
\mathcal{I}([e], \Gamma) \quad &= \quad \mathsf{t} \leftarrow \text{ fresh variable} \\
&\quad\; \text{if } (\mathsf{const}, ty, v) \leftarrow eval([e], \Gamma_{ext})\text{:} \\
&\quad\;\; \text{return } (\{\mathsf{t}\}, \{\mathsf{t} = (\mathsf{const}, ty, v)\}) \\
&\quad\; (V, S) \leftarrow \mathcal{I}(e, \Gamma) \\
&\quad\; S' \leftarrow \{\mathsf{t} = (\mathsf{data}, \langle \mathsf{class\ list} \rangle)\} \cup \{\mathsf{t}[] = t_e \mid t_e \in V\} \\
&\quad\; \text{return } (\{\mathsf{t}\}, S \cup S') \\
\mathcal{I}(Other\ (e_1, ..., e_n), \Gamma) \quad &= \quad \text{if } (\mathsf{const}, ty, v) \leftarrow eval(Other\ (e_1, ..., e_n), \Gamma_{ext})\text{:} \\
&\quad\;\; \mathsf{t} \leftarrow \text{ fresh variable} \\
&\quad\;\; \text{return } (\{\mathsf{t}\}, \{\mathsf{t} = (\mathsf{const}, ty, v)\}) \\
&\quad\; \text{else:} \\
&\quad\;\; (V_1, S_1) \leftarrow \mathcal{I}(e_1, \Gamma) \\
&\quad\;\; ... \\
&\quad\;\; (V_n, S_n) \leftarrow \mathcal{I}(e_n, \Gamma) \\
&\quad\;\; \text{return } (V_1 \cup ... \cup V_n, S_1 \cup ... \cup S_n) \\
\mathcal{I}(\text{from } p \text{ import } x' \text{ as } x, []) \quad &= \quad M' \leftarrow \text{find module of } x' \\
&\quad\; \mathsf{t1} \leftarrow lookup(M'.x', \Gamma_0) \\
&\quad\; \mathsf{t2} \leftarrow lookup(M.x, \Gamma_0) \\
&\quad\; \text{return } (\{\}, \{\mathsf{t2} = \mathsf{t1}\})
\end{aligned}
$$

Figure 6 From Python expressions and imports to 3-address-code. Given an environment Γ, the interpretation function for an expression $\mathcal{I}(e, \Gamma) = (V, S)$ returns a set of analysis variables V and the 3-address code S. M is the expression's enclosing module, and Γ_{ext} is its external environment. Γ_0 is the global environment.

The interpretation of a loop for x in e: s reduces to a sequence of assignment $x = e$ followed by s. The assignment binds identifier x before descending into the interpretation of s.

A function definition in a module initializer has an entry $M.f$ in Γ_0. Otherwise, i.e., if this function is nested into another function, we augment the local environment with f and return the augmented environment. Function definition gives rise to a "new" statement, assigning the abstract meta-func object to t. Class definitions have no effect during interpretation. Module-level class definitions are processed during class hierarchy analysis.

Note also that we ignore static reference environments for nested functions, which is unsound in general. A function value may flow to arbitrary points of the program and it is interpreted into 3-address code when it is called (detailed in calls in Section 3.3); however, interpretation happens in the empty environment rather than the actual static reference environment and references coming from enclosing scopes evaluate to empty sets.

For uninterpreted statements, the algorithm descends into each sub-statement and extracts the corresponding 3-address code. This does not "glue" components according to the statement's semantics. However, recursive descent processes all nested assignments and calls; it augments the environment and generates 3-address code that captures value flow.

3.2.5.2 Expressions

Recall that a key feature of our analysis is concrete evaluation and that concrete evaluation happens both during 3-address code generation and constraint resolution. We employ the following heuristic for 3-address code generation, as the rules for expression interpretation in Figure 6 detail. For simple expressions (i.e., variables), we first attempt resolution in the abstract environment: first Γ then Γ_0. If it fails, we attempt concrete evaluation. For all other expressions (i.e., complex expressions), we first attempt concrete evaluation in Γ_{ext}, and if it fails to produce a concrete object we proceed with interpretation in the abstract.

The interpretation of a variable (first rule in Figure 6) first searches the local environment Γ (i.e., the enclosing function), and then the global environment Γ_0 if the first lookup fails. If lookup in Γ_0 fails as well, it attempts concrete evaluation. If concrete evaluation succeeds and returns a concrete object, $\mathcal{I}(x, \Gamma)$ returns x and an object creation statement. The last clause aims at built-in functions, e.g., `len` and enables reporting of built-in callees. The reader may have noticed here that we reuse the original variable name as an analysis variable, but create a new concrete instance of the built-in function object. Thus, there are as many concrete instances representing `len` as there are (static) calls to `len` in the code. Our implementation does not optimize this case yet, but it is trivial to get rid of such redundant concrete objects.

For remaining expressions, including *Other*, the analysis first tries concrete evaluation, and if it succeeds, it returns the new analysis variable t along with the assignment of the concrete object to t. If an expression does not evaluate (which is the common case), interpretation continues recursively in the abstract environment. For example, `np.array(arg)` matches Call in Figure 6. Concrete evaluation fails with a NameError as `arg` is a local program variable; interpretation proceeds in the abstract. It recurses into Attribute `np.array` and this time it does evaluate to the numpy function. Interpretation of the call expression returns ({t1}, {t2 = (const, <... builtin ...>,...); t1 = t2(t3)}) where t3 is the analysis variable for `arg`.

Subscript expressions treat [] as a special field in the abstract, which is standard in points-to analysis. To interpret a list in the abstract, we create a new list object and generate subscript assignments to populate the list. Tuple, set, and dictionary are analogous.

For uninterpreted expressions we return the union resulting from the interpretation of each sub-expression when concrete evaluation fails.

3.2.5.3 Import

Finally, to interpret imports in module initializers (last rule of Figure 6), we first find the enclosing module M' of imported construct x', and then look up for the representative analysis variables t1 and t2 corresponding to the imported construct $M'.x'$ and the alias x in the current module M. Statement t2 = t1 propagates the value (e.g., a function definition, a class definition) from module M' to M. For example, consider statement `from cerberus.platform import _get_args as get_args` in module M. Lookup of `cerberus.platform._get_args` yields t1 which points to the function def of __get_args. Lookup of M.get_args yields t2 and t2 = t1 ensures the def propagates to t2 and references to get_args in M find that function def.

For simplicity, the rule treats x' as a module-level construct, however, it can be a module. E.g., `errors` in `from cerberus import errors` is a module and code can access module-level constructs as in `errors.DocumentErrorTree`. We handle this case in the attribute rule: if lookup of e in Γ during $\mathcal{I}(e, \Gamma)$ yields no result, the analysis maps $e.f$ to a module-level-construct identifier $M.C$, $M.f$ or $M.x$, and searches Γ_0 to retrieve the corresponding analysis variable (the step is not shown in the rule to avoid the complexity of lookup failure).

3.3 Constraint Resolution

As mentioned earlier, the analysis maintains abstract objects and concrete objects. Abstract objects are explicitly grouped into *data objects*, *function objects*, and *class object*:

(data, ⟨class C(...): ...⟩)	*an abstract data object*
(meta-func, ⟨def f(...): ...⟩)	*an abstract function object*
(meta-cls, ⟨class C(...): ...⟩)	*an abstract class object*
(const, ty, v)	*a concrete object*

The 3-address statements are as specified in Figure 2. These statements induce constraints that populate a points-to graph Pt. The nodes in the points-to graph are analysis variables as well as objects o of the above kinds. The edges represent the points-to relation. E.g., there could be an edge from variable t to an object o, and this is denoted as $\{o\} \subseteq Pt(t)$ or equivalently as $t \rightarrow o$. There could be a field edge indicating that field f of o_1 points to o_2 and this is denoted as $\{o_2\} \subseteq Pt(o_1.f)$ or equivalently as $o_1 \xrightarrow{f} o_2$.

The rules for new assignment, copy propagation and field write are largely standard (except for their returns) and we elide them from the presentation. We elaborate on the rules for field read $t_1 = t_2.f$ and function call $t_1 = t_2(t_3)$ as they illustrate Python-specific semantics and concrete evaluation.

3.3.1 Indirect read

The rule in Figure 7 examines each object in the points-to set of receiver variable t_2 and does case-by-case analysis on the object type. E.g., if it is an abstract data object of some user-defined class, the analysis searches the class hierarchy (MRO) to determine the function referenced by $o.f$, forms the closure by binding *self* to the receiver object o and adds the closure object to the points-to set of left-hand-side t_1. If the object is a concrete object, the analysis evaluates the field access returning a new concrete object and adding it to the points-to set of t_1. Here, and in Calls, we restrict the number of times a statement is evaluated successfully. The restriction, currently set to 2, is not only an optimization but also necessary for termination, otherwise the analysis would keep creating distinct concrete objects at the same statement during fixpoint iteration.

solve for $t_1 = t_2.f$ in $\langle$def f'(...): ...$\rangle$ with Γ_{ext} :
  for $o \in Pt(t_2)$
    case o **of**
      (data, $\langle$class C(...): ...$\rangle$) $\rightarrow$
        $\langle$def f(self, p): ...$\rangle \leftarrow H[(\langle$class C(...): ...$\rangle, f)]$
        $Pt(t_1) \leftarrow Pt(t_1) + \{$(meta-func, $\langle$def f(self, p): ...$\rangle)\}$
      (meta-cls, $\langle$class C(...): ...$\rangle$) $\rightarrow$
        $\langle$def f(self, p): ...$\rangle \leftarrow H[(\langle$class C(...): ...$\rangle, f)]$
        $Pt(t_1) \leftarrow Pt(t_1) + \{$(meta-func, $\langle$def f(self, p): ...$\rangle)\}$
      (const, ...) $\rightarrow Pt(t_1) \leftarrow Pt(t_1) + \{eval(o.f, \Gamma_{ext})\}$
    $Pt(t_1) \leftarrow Pt(t_1) + Pt(o.f)$ # *when f is an object field, add its points-to set*
  return $\{\langle$def f'(...): ...$\rangle\}$ if $Pt(t_1)$ changed else $\{\}$

Figure 7 Indirect read. Solving $t_1 = t_2.f$ in enclosing function $\langle$def f'(...): ...$\rangle$ with Γ_{ext}.

The last line returns the enclosing function f' to the worklist if there is a change to the points-to set of t_1, as the change may trigger changes to other points-to sets in f'.

3.3.2 Call

Figure 8 details the rule for calls. There are two cases at the top level, an abstract object or a concrete object as function value. For an abstract object, we do case-by-case analysis. If o is a data object, it queries the hierarchy to retrieve the corresponding ___call___ function – this is the function that is being called. If it is a meta class object, this leads to the retrieval of the constructor. If it is a meta function, there are two cases: a closure where self is already bound to a receiver, or the function is just a value with a null reference environment.

Once the analysis identifies the function to be called at this site, it checks if an interpretation of this function into 3-address code already exists. If it does not we interpret it. Notably, we interpret the AST of the function in the empty environment (i.e., only parameters are bound). This means that a first-class function does not carry its static reference environment and the analysis introduces unsoundness. While it is possible to extend the analysis with such bindings, this will complicate the code, while we believe in practice it will have limited impact (based on looking at Python code and our results). A notable departure from standard Java analysis is that there is no explicit new A() site. Different meta class objects may flow to receivers of calls accounting for data object creation (but the number of meta class objects is bounded and thus the number of data objects instantiated at the call is finite).

Once the callee function is interpreted, the analysis propagates points-to sets from actual arguments t_3 to formal parameters p and return values to the left-hand-side of the call t_1. In case of a concrete receiver, if the analysis finds concrete arguments for all parameters, it executes the function.

4 Results

We evaluate our Andersen-style points-to analysis, PoTo, on two clients: type inference (Section 4.1) and call graph construction (Section 4.2). However, points-to analysis has a wide variety of applications and we envision other clients as well. The analysis can be run on any Python package. It starts at a provided entry function and computes a points-to graph containing information on reachable variables and their inferred types. To thoroughly analyze a library package, a set of entry functions are needed. We follow DLInfer, a neural type inference for Python, and use the same 10 Python packages from their experiment available

solve for $t_1 = t_2(t_3)$ in $\langle$def f'(...): ...$\rangle$ with Γ_{ext} :
  for $o \in Pt(t_2)$
    if o is an abstract object
      case o **of**
        (data, $\langle$class C(...): ...$\rangle$) $\rightarrow$ # *call on data/instance*
          callee $\leftarrow H[(\langle$class C(...): ...$\rangle$, '__call__')]
          rcv $\leftarrow \{o\}$
        (meta-cls, $\langle$class C(...): ...$\rangle$) $\rightarrow$ # *constructor call*
          callee $\leftarrow H[(\langle$class C(...): ...$\rangle$, '__init__')]
          rcv $\leftarrow \{$(data, $\langle$class C(...): ...$\rangle$)$\}$ # *new object*
        (meta-func, $\langle$def $f(o', p)$: $s\rangle$)) $\rightarrow$ # *closure call*
          callee $\leftarrow \langle$def f(self, p): $s\rangle$
          rcv $\leftarrow \{o'\}$
        (meta-func, $\langle$def $f(p)$: $s\rangle$) $\rightarrow$
          callee $\leftarrow \langle$def $f(p)$: $s\rangle$ # *the function def*
          rcv $\leftarrow$ None
        $\langle$def $f(p)$: $s\rangle$ = *callee* # *match and deconstruct callee*
        if $\langle$def $f(p)$: $s\rangle \notin \Phi$ # *callee is not interpreted*
          t $\leftarrow$ fresh variable
          $\Phi[\langle$def $f(p)$: $s\rangle] \leftarrow \mathcal{I}(s, [(p, \text{t})])$ # *env. includes self when f is an instance function*
        if *rcv* $\neq$ None # *there is a receiver*
          $t_4 \leftarrow \Phi[\langle$def f(self, p): $s\rangle][0]$[self] # *retrieve analysis variable corresponding to self*
          $Pt(t_4) \leftarrow Pt(t_4) +$ *rcv* # *receiver to self*
        $t_5 \leftarrow \Phi[\langle$def $f(p)$: $s\rangle][0][p]$
        $Pt(t_5) \leftarrow Pt(t_5) + Pt(t_3)$ # *actual to formal*
        $t_6 \leftarrow \Phi[\langle$def $f(p)$: $s\rangle][0][f$_ret]
        $Pt(t_1) \leftarrow Pt(t_1) + Pt(t_6)$ # *ret var to lhs of call*
      else # *o is a concrete object*
        for o_1 in $Pt(t_3)$
          if o_1 is a concrete object
            $Pt(t_1) \leftarrow Pt(t_1) + \{eval(o(o_1), \Gamma_{ext})\}$
  return $\{\langle$def f(...): ...$\rangle, \langle$def f'(...): ...$\rangle\}$ if change else {}

Figure 8 Call: solving for $t_1 = t_2(t_3)$ in enclosing function $\langle$def f'(...): ...$\rangle$ with Γ_{ext}.

with DLInfer's artifact. The 10 packages range in size from 3,556 LOC to 285,515 LOC (see Table 1). These packages contain rich test cases which suit our need of diverse entry points. We use each function in a package's test directory as an entry point. For five of the packages (cerberus, mtgjson, pygal, sc2, and zfsp), we additionally create custom entry functions targeting remaining unreachable public functions. For the other five packages (anaconda, ansible, bokeh, invoke, and wemake_python_styleguide), we only use default test suites.

4.1 Type Inference

To target unreachable methods for the purposes of type inference, we enhance the analysis. We achieve this by performing a shallow analysis that collects built-in type information and type annotations from assignment and return statements. For example, consider an assignment with a built-in type: `rules = set(schema.get(field, ()))`. The shallow analysis infers that `rules` can be of type `set`. This has impact when the method is unreachable from PoTo's entry points, as it creates a key and infers a type for the `rules` local variable. The final results, called PoTo+, are stored as a dictionary of *keys* to their inferred types. A *key* is a tuple of (module name, function name, variable name), describing the variable and its

scoping information. Keys largely correspond to local variables (including arguments and returns) in a package and are an abstraction for flow-insensitive analysis, the target of our work. The remainder of this section uses the terms keys and variables interchangeably.

We compare the results of PoTo+ against four other type inference techniques. Three of these are based on the recent neural type inference work DLInfer [46]: DL-ST, DL-DY, DL-ML. We make use of the result files available with DLInfer's artifact [45]. DL-ST is their ground truth information, which is a combination of running the Pysonar2 static tool and extracting type information [42]. DL-DY is a set of dynamic type information obtained from executing the test suites. This dynamic set contains only variables whose type can be collected only this way, meaning that this is the set difference of the actual dynamic run and DL-ST set [46]. Lastly, DL-ML is the result of DLInfer's machine-learning approach. We aggregate each DLInfer result in the same way as our analysis, which is a dictionary of keys to their types. We choose DLInfer for several reasons: (1) DLInfer is recent work in a top conference, (2) it compares with several state-of-the-art deep-learning techniques: Type4Py [27], Typilus [1], PYInfer [9], and DeepTyper [15], (3) DLInfer infers types for local variables, not only parameters and returns, and (4) its artifact [45] includes full Python packages along with analysis results allowing for a comparison over the same code base.

In addition to DLInfer, we also compare our result to Pytype, a prominent static type checking and type inference tool [12]. To get type information for all variables, we instrument the package by inserting Pytype's command `reveal_type(var)` for each local variable at the end of a function, and run Pytype on each file in the package directory. The types are then collected and combined to be the inferred result of Pytype. This process takes time to complete but only needs to be done once. This is a departure from standard use of Pytype as baseline for type inference work, which uses Pytype's result on parameters and returns only (e.g. [29]); Pytype can infer types for local variables as well and we make use of this in our comparison.

Our evaluation considers 4 research questions:

RQ1: How high is the coverage of PoTo+ compared to other type inference techniques?
RQ2: To what extent are the types from PoTo+ equivalent to those from other techniques?
RQ3: If the types do not match those from other techniques, which one is correct?
RQ4: Does the time to run PoTo+ scale well and how does it compare to Pytype?

To answer RQ1, we measure the percentage of total keys in a package for which our analysis reports types against the percentage of total keys for which the four other type inference techniques report types (more detail on methodology in Section 4.1.1). Our analysis collects types for a *larger percentage of total keys* compared to the four other techniques. To answer RQ2, we compare our inferred types to those inferred by each of the four other techniques, measuring the equivalence in term of total match, partial match, and mismatch (more detail in Section 4.1.2). Our inferred types *largely match with Pytype's*. While there is some match with DLInfer, there is disagreement in many keys.

To answer RQ3, we inspect a sample of mismatches according to the result from RQ2 for each pair comparisons: PoTo+ vs. Pytype, PoTo+ vs. DL-*. We found that in the few cases where there is a mismatch with Pytype, Pytype is correct in all cases. A mismatch with DL-* is nearly always a correct result by PoTo but an incorrect result by DL-*. The conclusion from RQ1-RQ3 is that *traditional techniques outperform a state-of-the-art neural technique* for the task of type inference. Lastly, for RQ4, we compare the total time to run PoTo (with dozens of entry functions for the smaller packages and thousands for the larger ones) to the time to run Pytype to collect reveal-type information. On all but one package, ansible, PoTo outperforms Pytype significantly.

Table 1 Statistics for the dataset.

Package	Files		LOC		Keys
	Total	Test Dir	Total	Test Dir	
cerberus	45	32	6,694	3,011	966
mtgjson	54	2	6,912	58	1,378
pygal	78	24	13,780	3,208	2,439
sc2	69	6	11,205	603	5,970
zsfp	54	7	3,556	207	1,345
anaconda	370	9	90,207	587	21,183
ansible	1,445	961	285,515	174,296	22,346
bokeh	1,133	280	131,931	43,316	14,978
invoke	133	65	26,159	9,878	3,474
wemake	403	291	55,841	32,798	3,184

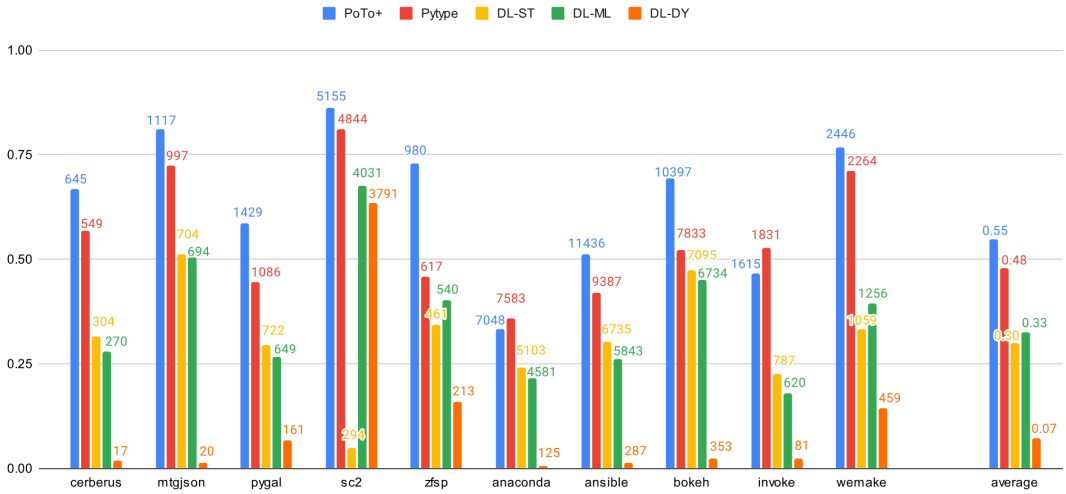


Figure 9 Coverage percentages of non-empty keys to total keys (RQ1).

4.1.1 Coverage (RQ1)

Table 1 shows the statistic of all 10 packages. We are interested in inferring types for the core package and exclude test files from our reports. The total keys include all variables, function arguments, and function return types. They serve as an upper limit of possible variables (or keys) for each package.

Figure 9 shows the percentages of non-empty keys to the total keys for each package. Non-empty keys are ones for which the analysis (ours or other techniques) gives type information; empty keys are variables in total keys for which the analysis gives no information. In the case of Pytype, a key is designated as empty if its inferred type is the trivial `Any` type. The numbers above each bar are the numbers of non-empty keys. For the five packages (cerberus, mtgjson, pygal, sc2, and zfsp) where we add custom entry functions, PoTo+ covers more non-empty keys than other techniques, followed closely by Pytype. For the remaining packages where we only use default test suites, we have slightly worse coverage than Pytype on anaconda and invoke, reflecting worse test coverage by the underlying test suites. PoTo+ still has the highest average coverage percentage.

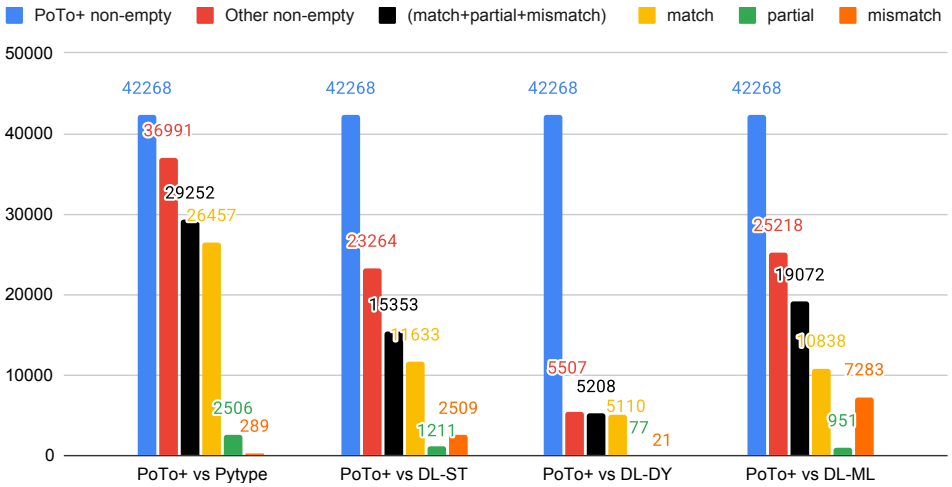


Figure 10 Equivalence comparison between PoTo+ and other type inference techniques (RQ2). Black bars (sums) show the number of keys for which both techniques report types. Numbers are summation of all 10 packages.

4.1.2 Equivalence (RQ2)

Next, we measure the similarity of inferred types. For each variable in a package, our analysis and the other type inference techniques can either have information on this variable, and the information is the set of its inferred types, or have no information, i.e., an empty set.

For built-in containers such as `dict`, `list`, `set`, and `tuple`, we compare only the top-level part and deem that they match if they are the same container types. The main reason for this shallow comparison is to facilitate processing and comparison as each type inference technique infers and reports types in different forms. For example, DLInfer reports only the type of a container, e.g., `dict`. Pytype reports parametric information, e.g., `Dict[str, int]`, but not always, while PoTo+'s inferred types come from abstract and concrete objects, e.g., `{'rename_handler': <class 'int'>}`. PoTo can collect parametric information, but it requires significant processing and raises issues on reporting types for polymorphic containers.

Matching is automatic but some comparisons require manual verification because of rendering of PoTo concrete objects. For example, PoTo+'s `1970-01-01` matches with DL-ST's `datetime.date` type. Another example is a match between PoTo+'s `<function <lambda> at 0x109297520>` and Pytype's `Callable[[Any],Any]`.

We are interested in 3 groups of equivalence: total match, partial match, and mismatch, where partial match is when the two sets of types have non-empty intersection and mismatch is an empty intersection. Figure 10 shows these as summation across the 10 Python packages. In addition, Figure 10 shows bars with the total number of non-empty keys by PoTo+ next to bars with the total number of non-empty keys by the other technique; for convenience it draws a bar that sums match, partial match and mismatch, i.e., the set of keys for which both techniques report a type. On all Python packages, PoTo+ shares the most non-empty keys with Pytype, and has high numbers of total matching. DL-ST and DL-ML have some matching and partial matching with our analysis, but also contain many keys that are mismatches. The discrepancy is discussed in Section 4.1.3. For the sc2 package, almost all common keys between PoTo+ and DL-ML are mismatches. The mismatches are due to simple assignments of class attributes in /sc2/ids/ directory, e.g. `NULL = 0`, `RADAR25 = 1`, `TAUNTB = 2`. Our analysis, Pytype, and DL-DY correctly infer the types as integer, but DL-ML labels all

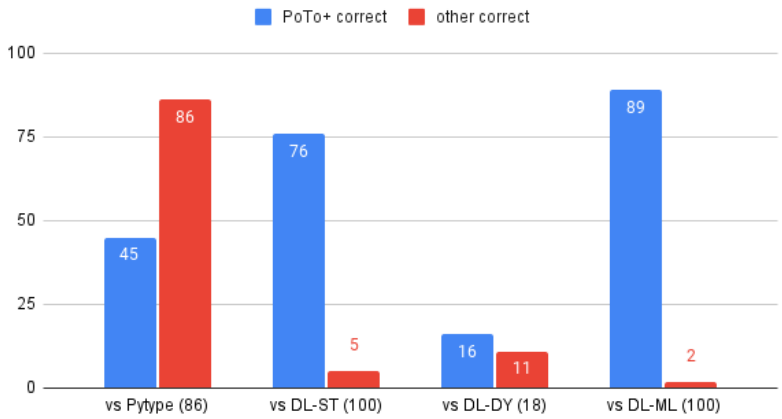


Figure 11 Correctness comparison between PoTo+ and other techniques (RQ3). Numbers in parentheses are # of samples. This figure zooms in on a sample of the mismatched keys from Figure 10; for Pytype there are 289 mismatched keys which is less than 1% of all variables.

of them as `num` which is incorrect as it is not a type. There are 3,791 such assignments out of 3,984 common keys between PoTo+ and DL-ML in the sc2 package. Lastly, DL-DY has low coverage of types information, and most of them are a total match with our analysis.

Figure 10 shows that PoTo+ reports a larger number of types compared to the other techniques; for example, for Pytype, there are 29,252 keys for which both report type information (the black bar) and there are 13,016 keys for which PoTo+ reports a type but Pytype does not. In contrast there are 7,739 keys for which Pytype reports a type but PoTo+ is empty. Appendix A.1 includes per benchmark versions of this figure.

4.1.3 Correctness (RQ3)

To carry out the comparison we manually examine a sample of 10 non-match keys for each package and each pair of comparisons, i.e., PoTo+ vs. Pytype, PoTo+ vs. DL-ST, PoTo+ vs. DL-DY, and PoTo+ vs. DL-ML. In cases where there are fewer than 10 non-match keys, we exhaustively examine all of them, (e.g., there are only 8 non-matches between PoTo+ and Pytype for pygal, so there are 8 instead of 10 keys for pygal). There are only 18 total non-matches keys for PoTo+ vs. DL-DY. For the remaining 3 techniques, the total is slightly under 300 samples. Results are shown in Figure 11.

4.1.3.1 PoTo+ vs. Pytype

Comparison with Pytype shows that a mismatch usually means that PoTo+ is incorrect while Pytype is correct. Of the 86 pairs we examined, PoTo+ is correct 45 times, while Pytype is correct 86 times, because Pytype handles a larger set of Python features precisely, while PoTo defaults to Other, thus polluting points-to sets. For example, PoTo does handle list and dictionary comprehensions precisely, but defaults to Other on set comprehensions. Consider the example: `rampPoints = {p for p in rampDict if rampDict[p]}`. Recall that handling of Other propagates points-to sets of `p`, `rampDict` and `rampDict[p]` into `rampPoints`. This leads to PoTo+ reporting type `[dict, bool]` as `p`'s set is empty and `rampDict`'s set contains a single dictionary object of `Point` object keys, and bool values (it is straightforward to add handling of set comprehensions and other features to PoTo to improve matching).

Table 2 Running times for PoTo and Pytype (RQ4).

Package	PoTo	Pytype	Package	PoTo	Pytype
cerberus	51s	4m22s	anaconda	1m59s	1h21m40s
mtgjson	1m39s	15m39s	ansible	6h23m07s	6h14m59s
pygal	1m48s	1h00m17s	bokeh	1h41m16s	10h15m41s
sc2	52s	40m29s	invoke	1m32s	20m45s
zsfp	11s	7m28s	wemake	3m25s	9m17s

For the majority of cases where both are correct, PoTo+ reports a concrete type while Pytype reports a type parameter. For example, PoTo+ reports correctly `[dict, None]`, i.e., an optional dictionary for the return type of `_normalize_purge_unknown` in the cerberus package; while Pytype reports type variable `[_T0]` and this is correct with respect to Pytype's type system which allows for polymorphic functions.

4.1.3.2 PoTo+ vs. DL-ML, DL-ST and DL-DY

Of the 100 pairs we examine for PoTo+ vs. DL-ML, PoTo+ is correct 89 times, while DL-ML is correct 2 times. We mark PoTo+ incorrect conservatively – essentially, when the result is partially correct but not sound and not complete, we mark it incorrect. One common source of incorrect result for DL-ML was the `num` types mentioned earlier. Another source is usage of the function name instead of the function's return type at call assignments. Consider the following example from package pygal: `background = lighten('#e6e7e9',7)`. DL-ML reports type `[lighten]` for `background`, while PoTo+ correctly reports `[str, tuple]`. Yet another source of incorrect results is over-reliance on `dict` types. For example, in the wemake package: `AnyFunctionDef=Union[ast.FunctionDef,ast.AsyncFunctionDef]`, PoTo+ correctly infers type `[typing.Union[...]]` for `AnyFunctionDef` (via concrete evaluation). DL-ML infers `[dict]`.

Of the 100 samples we examine for PoTo+ vs. DL-ST, PoTo+ is correct 84 time and DL-ST is correct 5 times. There are no `num` types, as they are correctly assigned `int`; however, we observe similar patterns of function name instead of return type, and over-reliance on `dict`.

There are only 18 total mismatches for PoTo+ vs. DL-DY; 8 from ansible and 10 from bokeh. PoTo+ is correct 16 times and DL-DY is correct 11 times. Most cases are from accessing a dictionary or calling a function such as `children = kwargs.get('children')` or `typ = type(obj)`. On 5 occasions, DL-DY infers types that appear wrong such as `type`.

4.1.4 Scalability of PoTo (RQ4)

Table 2 show the running times of PoTo (total time for thousands of entry functions) and of Pytype. We run on a commodity Mac book Pro with 2.4 GHz 8-Core Intel Core i9 and 32 GB of Memory (one of our development machines). The "+" phase (type inference) is instantaneous and type aggregation and processing are negligible. We do not include these in the timing. In both cases, execution is IO-dominated, as PoTo writes points-to results into pkl files and Pytype's reveal-type reveals types by printing special error message.

PoTo runs in 1–3 minutes for all but the two largest packages, bokeh and ansible. Pytype is significantly more expensive ranging between 4 and 81 minutes on those packages. Without the reveal-type instrumentation Pytype runs 30% faster than with; thus, its underlying static analysis is expensive and PoTo still outperforms Pytype. PoTo runs slower than Pytype on ansible because ansible has nearly 500 test file (we filtered out ones that do not reference ansible packages from the 961 original ones to speed up testing) and many hit the same bottleneck of ansible code in the points-to analysis.

4.2 Call Graphs

To strengthen the case for applicability of PoTo, we evaluate it on another classical client of points-to analysis: call graph construction. We call the client PoToCG. We compare PoToCG to PyCG, the leading Python call graph construction analysis [34]. Unfortunately, PoToCG and PyCG are two different analyses, with different goals, and comparing them is non-trivial. PoToCG is a standard whole-program reachability analysis [3, 33, 23], whose goal is as follows: given a *main function* and a package, compute the smallest, i.e., most precise, call graph of reachable functions. PyCG, on the other hand, takes a set of files as input and computes a call graph that aims to include every call site in these files. Both graphs are unsound, by virtue of being analyses of Python, and both techniques aim to minimize the number of missing call graph nodes and edges. Nevertheless, PyCG is still the best Python call-graph analysis to compare to representing the state-of-the-art, and we made the comparison as apples-to-apples as possible modulo the different nature of the two approaches. We do not expand PoToCG to unreachable functions for this comparison, because such an expansion would entail a brand new call graph analysis that is antithetical to the reachability analysis inherent in Andersen-based call graph construction. Instead, we report on the number of reachable functions in PoToCG compared to PyCG, then we zoom in on the PoToCG-reachable functions and compare the quality of the PoToCG call graph for those functions compared to PyCG.

The research questions we address are as follows:

RQ5: How does PoToCG compare to PyCG in terms of reachable functions?

RQ6: For the PoToCG-reachable functions, how *complete* and how *precise* is the PoToCG call graph compared to PyCG?

We construct the PoToCG call graph by traversing call 3-address statements $x = y(z)$ in PoTo reachable functions and examining the points-to set of y. Meta-func objects represent abstract functions and lead to callees in the package. Concrete objects lead to callees that are either built-in functions (e.g., `len`) or external library functions (e.g., `re.sub`).

Consider function `UnconcernedValidator.__init__` in benchmark cerberus containing the following statement: `self.document_error_tree = errors.DocumentErrorTree()`. It gives rise to a corresponding 3-address-code call statement $x = y(z)$. Analysis variable y's points-to set contains meta-func abstract object `cerberus.errors.DocumentErrorTree.__init__`. Thus, we record a call graph edge from caller `UnconcernedValidator.__init__` to callee function `cerberus.errors.DocumentErrorTree.__init__` (this edge is missing from the PyCG results). PoToCG handles calls to built-in functions as well by recording the concrete object corresponding to the built-in function, e.g., `len`, `int`, etc. PyCG captures built-in callees as well. We aggregate results across all test cases, as in the type inference client.

Unfortunately, PyCG ran only on the five smaller packages – cerberus, pygal, mtgjson, sc2, and invoke. It did not terminate on any of the remaining packages. (PoToCG terminates on them.) To run PyCG, all files in a package are passed to PyCG as entry points. The five packages that terminated took 30 seconds to a few minutes, which are about the same as PoToCG as shown in Table 2. The other packages timed out after two hours per package.

Table 3 shows function coverage of PoToCG and PyCG (RQ5). The top three rows provide the raw statistic of each package we produced by scanning its Python AST. For the result of both techniques (row 4 and 5), we are interested in functions that have at least one outgoing edge or one incoming edge in their respective call graphs, meaning that we exclude isolated empty functions. Except for invoke, which relies entirely on its test suite (we did not write additional tests for it), PoToCG achieves relatively good coverage: 49% for cerberus, 72% for

Table 3 Function coverage of PoToCG and PyCG (RQ5). * indicates that the number is unusual due to non-existent callees, i.e., callees that do not correspond to function definitions.

Functions	cerberus	pygal	mtgjson	sc2	invoke
Total functions	200	354	237	567	782
Funcs without a call stmt	38	83	32	256	111
Funcs with call stmt	162	271	205	311	671
Funcs in PoToCG	91	218	211	266	99
Funcs in PyCG	161	816*	259	412	738
Funcs in PoToCG but not in PyCG	5	26	0	15	2
Funcs in PyCG but not in PoToCG	75	624	48	161	641
Funcs in both. Same # of edges	37	88	86	168	50
Exactly the same set of edges	29	71	57	123	37
Different set of edges	8	17	29	45	13
Funcs in both. #Edges: PoToCG > PyCG	43	54	76	67	33
Funcs in both. #Edges: PyCG > PoToCG	6	50	49	16	14

Table 4 Edges in PoToCG reachable functions (RQ6).

Edges	cerberus	pygal	mtgjson	sc2	invoke
True positive PoToCG	**339**	**599**	**679**	**438**	**260**
True positive PyCG	192	398	550	351	220
False positive PoToCG	**1**	**0**	**0**	**0**	**0**
False positive PyCG	12	813	116	4	6

pygal, 91% for mtgjson, and 77% for sc2. We measure coverages by percentages of functions in PoToCG that have call statements, to total functions with call statements. PyCG numbers for pygal and mtgjson exceed the upper bound of total functions, because PyCG reports caller and callee nodes that cannot be mapped to actual function definitions, e.g., `pygal.etree.etree.fromstring.findall.node.serie.get.split.append`. PyCG includes each module as a node as it can potentially have call edges. PyCG also maps functions in `__init__.py` files differently. For example, function `safe_load()` in `invoke/vendor/yaml3/__init__.py` has its path as `invoke.vendor.yaml3.safe_load`. We adjust PoToCG to accommodate these cases.

Table 4 zooms in on the functions that are both in PoToCG and PyCG for which the two techniques report *different sets of callee functions* (RQ6). These are the bottom three lines in Table 3 surrounded with a solid rectangle. We examined all the edges in these functions. We classified a caller-callee edge as *true positive* when reaching a specific call statement in the caller would indisputably lead to the callee for some receiver (when there is a receiver); this is the case for the `DocumentErrorTree` constructor example above. We classified an edge as *false positive* otherwise. In practice, essentially all false positives were in PyCG where the reported callee could not be mapped to a function definition in the package or in an external library. One can see in Table 4 that PoToCG reports considerably better call graphs for the functions it reaches – it reports a much larger number of true positive callees compared to PyCG as well as virtually no false positives, while PyCG reports a large number of false positives for two of the benchmarks. Therefore, PoTo's call graph is more complete and more precise for the functions it reaches. We observed that when reported callees could be

mapped to function definitions, both PoToCG and PyCG's callees were true positives. The reason why PyCG had a large number of false positives was that it reported many callees that did not correspond to function definitions.

We consider cases that highlight PoToCG's better recall (i.e., soundness) and better precision. PoToCG's better recall is due to its flow analysis as in the following example:
`validator = self._get_child_validator(..); mapping[field] = validator.normalized(..)`
In this example, the analysis must infer that a `Validator` object flows to the `validator` variable. PoTo infers that the analysis variable corresponding to `validator` points to a Validator object and correctly infers a call edge to `cerberus.base.UnconcernedValidator.normalized`. In contrast, PyCG leaves this call empty.

Another reason for PoToCG's better coverage is concrete evaluation. Consider
`result = validator.normalized(...); mapping[field] = value_type(result.values())`
In the above code, an empty concrete dictionary flows to the analysis variable corresponding to `result` and concrete evaluation deduces that the `values` member function is called. In contrast, PyCG does not infer a callee at this call.

As mentioned previously, PyCG reports hundreds of non-existent callees as in the following example: `pygal.util.decorate.serie.node`. Here, `decorate` is a function in the `util` module. We surmise PyCG infers `decorate` to return an object that has a field `serie`; however, it is not clear how that access path corresponds to an actual function definition in the pygal package. Another example is `pygal.graph.public.PublicApi.svg.serie.node`. We observed large numbers of such false positive callees in the pygal and mtgjson benchmarks but nearly none in the remaining benchmarks. In contrast, PoToCG reports were precise – if the call is reached then the reported function will be called.

Overall, we observed that recall (i.e., soundness) rather than precision is a more pressing issue for all analyses. PoToCG has better recall overall for the functions it reaches as detailed in Table 4, however, in the absence of ground truth it is difficult to judge how complete PoTo's call graph is. We did not observe precision issues in either analysis results, other than PyCG reporting non-existent callees. We further examined several applications and their hierarchies and we could not easily identify a call site that would have benefitted from a context- or object-sensitive analysis. For example, in pygal, a charting library and one of the larger applications in the suite, classes encapsulated predominantly string and numerical values, as opposed to complex objects; superclasses contained nearly all the functionality; and there were no setter and getter functions. As a specific case, one class hierarchy had root `class Style` which defined several class-level string and numerical fields representing colors and fonts. It also defined a function `get_colors` which formatted color strings to interact with CSS. Subclasses of `Style` added a small number of string or numerical class-level fields and defined no functions. Appendix A.2 gives an example where PoTo shows a clear advantage over an object-sensitive analysis. We conjecture that Python applications use class hierarchies differently from Java although further studies are needed. We consider PoTo a necessary baseline. Improved precision, possibly via context-sensitive, is a promising direction for future work in Python program analysis.

5 Threats to Validity

Section 4.1.3 discusses correctness comparison that depends on limited samples out of thousands of mismatches. To mitigate this, we collect all 18 mismatches against DL-DY, plus 10 samples per packages for all packages across 3 other techniques, resulting in total of 300 hand-labeled examples with broad coverage.

The analysis relies on unit tests for entry points, thus presuming the existence of test suites and good test coverage of the test suites. To increase coverage, we added custom entry-points for 5 of the packages; notably, coverage remains robust even for those large packages where we used only the existing test suites.

Lastly, PoTo is a hybrid analysis that alternates between concrete and abstract evaluation, and it employs semantic choices to deal with Python's complexity. This makes the analysis unsound by design, but it is the typical kind of trade-off made by most real-world static analysis [24]. Our results with two client analyses show that PoTo compares favorably against respective state-of-the-art techniques, thus validating its trade-offs.

6 Related Work

This section discusses prior work related to each of the three contributions stated in Section 1.

Points-to analysis. At its core, PoTo is an implementation of Andersen's points-to analysis [2], but unlike the original, it works for Python and is hybridized with concrete evaluation. The only other points-to analysis for Python we have found is in Scalpel [22]; however, Scalpel's analysis is not based on Andersen's, does not use concrete evaluation, and the paper lacks empirical results. Few static analyses have been shown to work on real-world Python programs, including PyCG [34] (which finds call graphs) and Tree-sitter [8] (which is limited to syntactic queries) [8]. Neither PyCG nor Tree-sitter does points-to analysis, nor do they use concrete evaluation. This paper empirically compares PoToCG against PyCG.

Outside of Python, work on points-to analysis and related alias and value-flow analysis dates decades back. Chatterjee et al. [6] describe a context-sensitive points-to analysis for C++, Rountev et al. [33] present an Andersen-style analysis for Java, and Fähndrich et al. [10] define a context-sensitive points-to analysis for C, among many other works. Siu and Xue [38] and Shi et al. [35] present scalable value-flow analysis over LLVM IR. Jang and Kwang [18] define the first Andersen-like points-to analysis for JavaScript and Sridharan et al. [37] improve Andersen's analysis with better reasoning about property accesses. These analyses work over established 3-address code IR in LLVM, Wala, and Soot. In contrast, we needed to construct 3-address code for points-to analysis from Python AST constructs.

Recent works on points-to analysis for Java (e.g., [25], [19]) focus on context- and object-sensitivity to improve precision. Our analysis is a context-insensitive and flow-insensitive baseline; we envision that the 3-address code and hybridization will serve as a foundation for building context- and flow-sensitive analysis for Python in the future.

Hybridization weaving concrete and abstract evaluation. PoTo uses concrete evaluation to solve the problem of analyzing Python programs that use external libraries. Two other works hybridize abstract (i.e. static) with concrete (i.e. dynamic) analysis for Python: PyCT [7] (which does concolic testing) and Rak-amnouykit et al.'s analysis [32] (which finds weakest preconditions). Neither is based on points-to analysis, nor has been used for type inference.

Looking back outside of Python, the idea of combining static and dynamic analyses has a long history in program analysis of Java and JavaScript. Hirzel et al. [16] hybridize Andersen's analysis with dynamic analysis to handle Java's dynamic class loading. Grech et al. [13] define a hybrid analysis for Java that takes whole heap snapshots and integrates them into the analysis. Wei and Ryder [43] present blended analysis for JavaScript that collects traces of execution then uses static analysis to reason about flow of tainted values. Tripp et al.'s analysis for JavaScript [40] addresses the problem of modeling the DOM; it

runs a dynamic analysis to concretize the external environment, followed by a static analysis in the concrete environment. Similarly, Laursen et al. [21] do a dynamic pre-analysis followed by a static analysis that uses runtime results; they target property access in JavaScript, a notorious issue in JavaScript codes [5]. Our analysis differs from all these works as it targets Python. It presents a novel blend of static and dynamic interpretation – it weaves concrete evaluation in both 3-address code generation and constraint resolution and elevates concrete objects to first-class status in the analysis. Park et al. [28]'s analysis of JavaScript does concrete execution of selected functions using carefully constructed objects; this is similar to our usage, however, their goal is to improve precision, while our goal is to improve soundness.

We view Toman and Grossman's Concerto [39] as most closely related to our work. Concerto is a combined concrete and abstract interpretation framework. It considers concrete state (for framework code) and abstract state (for application code) and allows calls from one domain into the other. Concerto is a theoretical framework, while we focus on Python and develop 3-address code generation and a specific analysis, Andersen's points-to analysis.

Type inference and call graph analysis for Python. PYInfer [9] and DeepTyper [15] use deep learning for Python type inference, and several other works combine deep learning with simple static analysis [1, 27, 29, 30, 44, 46]. This paper empirically compares PoTo+ against the latest of those, DLInfer [46], chosen because it compares with several earlier works and its artifact [45] has points-to sets for 10 real-world Python programs. A handful of other works use static analysis for Python type inference [11, 12, 14, 26]. Unlike PoTo+, none of these use points-to analysis nor concrete evaluation. In addition to DLInfer, this paper also empirically compares PoTo+ against Pytype [12], which is based on static analysis. We chose Pytype as a second baseline because it is widely used in practice and, like PoTo+, handles real-world Python programs [31]. Even though call-graph analysis is a popular client for points-to analysis in other languages, for Python, it has received little attention from the academic community. The main work is PyCG [34], and thus, we chose it as a baseline.

Instead of hybridizing static analysis with concrete evaluation, several works hybridize static analysis with machine-learning (ML). Xu et al. present a static type inference for Python augmented with ML for guessing types based on variable names [44]. Typilus uses static analysis to build a Python program dependency graph, then uses a graph neural network for type inference [1]. TypeWriter performs deep-learning (DL) type inference, then uses a static type checker to repair hallucinations [30]. Type4Py [27], HiTyper [29], and DLInfer [46] are primarly DL type inferences for Python, assisted by simple static analyses.

7 Conclusions

This paper presents PoTo, the first Andersen-style points-to analysis for Python. PoTo works on real-world Python programs, which use dynamic features as well as external packages for which source code is often missing (and which often involve non-Python code). To handle external packages, PoTo introduces a novel hybridization of Andersen's static analysis with dynamic concrete evaluation. In terms of clients, this paper presents PoTo+, a type inference built upon PoTo. While several recent papers explore deep learning for Python type inference, our results indicate that (at least as of now) static analysis solves this problem with superior coverage and correctness. This paper also presents PoToCG, a call-graph analysis built upon PoTo that performs better than the state-of-the-art for Python.

References

1 Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 91–105, 2020. doi:10.1145/3385412.3385997.

2 Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994. DIKU report 94/19. URL: ftp://ftp.diku.dk/pub/diku/semantics/papers/D-203.dvi.Z.

3 David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1996, San Jose, California, USA, October 6-10, 1996*, pages 324–341. ACM, 1996. doi:10.1145/236337.236371.

4 Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for dylan. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 69–82, 1996. doi:10.1145/236337.236343.

5 Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. Automatic Root Cause Quantification for Missing Edges in JavaScript Call Graphs. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:28, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2022.3.

6 Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 133–146, New York, NY, USA, 1999. Association for Computing Machinery. doi:10.1145/292540.292554.

7 Yu-Fang Chen, Wei-Lun Tsai, Wei-Cheng Wu, Di-De Yen, and Fang Yu. Pyct: A python concolic tester. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 38–46, 2021. doi:10.1007/978-3-030-89051-3_3.

8 Timothy Clem and Patrick Thomson. Static analysis at GitHub. *Communications of the ACM (CACM)*, 65(2):44–51, January 2022. doi:10.1145/3486594.

9 Siwei Cui, Gang Zhao, Zeyu Dai, Luochao Wang, Ruihong Huang, and Jeff Huang. PYInfer: Deep learning semantic type inference for python variables, June 2021. doi:10.48550/arXiv.2106.14316.

10 Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 253–263, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/349299.349332.

11 Levin Fritz and Jurriaan Hage. Cost versus precision for approximate typing for python. In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 89–98, January 2017. doi:10.1145/3018882.3018888.

12 Google. pytype, January 2016. Retrieved June, 2024. URL: https://github.com/google/pytype.

13 Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Heaps don't lie: countering unsoundness with heap snapshots. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi:10.1145/3133892.

14 Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. MaxSMT-based type inference for Python 3. In *Conference on Computer Aided Verification (CAV)*, pages 12–19, 2018. doi:10.1007/978-3-319-96142-2_2.

15 Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 152–162, 2018. doi:10.1145/3236024.3236051.

16 Martin Hirzel, Amer Diwan, and Michael Hind. Pointer analysis in the presence of dynamic class loading. In *European Conference for Object-Oriented Programming (ECOOP)*, pages 96–122, 2004. doi:10.1007/978-3-540-24851-4_5.

17 IBM. T. j. watson libraries for analysis (wala), 2006. Retrieved June, 2024. URL: https://github.com/wala/WALA.

18 Dongseok Jang and Kwang-Moo Choe. Points-to analysis for javascript. In Sung Y. Shin and Sascha Ossowski, editors, *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009*, pages 1930–1937. ACM, 2009. doi:10.1145/1529282.1529711.

19 Minseok Jeon and Hakjoo Oh. Return of cfa: call-site sensitivity can be superior to object sensitivity even for object-oriented programs. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–29, 2022. doi:10.1145/3498720.

20 C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization (CGO)*, pages 75–86, March 2004. doi:10.1109/CGO.2004.1281665.

21 Mathias Rud Laursen, Wenyuan Xu, and Anders Møller. Reducing static analysis unsoundness with approximate interpretation. *Proceedings of the ACM on Programming Languages*, 8(PLDI):1165–1188, 2024. doi:10.1145/3656424.

22 Li Li, Jiawei Wang, and Haowei Quan. Scalpel: The Python static analysis framework, February 2022. doi:10.48550/arXiv.2202.11840.

23 Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. Precision-guided context sensitivity for pointer analysis. *Proc. ACM Program. Lang.*, 2(OOPSLA):141:1–141:29, 2018. doi:10.1145/3276511.

24 Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM (CACM)*, 58(2):44–46, January 2015. doi:10.1145/2644805.

25 Jingbo Lu and Jingling Xue. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi:10.1145/3360574.

26 Eva Maia, Nelma Moreira, and Rogério Reis. A static type inference for Python. In *Workshop on Dynamic Languages and Applications (DYLA)*, 2011. URL: http://scg.unibe.ch/download/dyla/2011/dyla11_submission_3.pdf.

27 Amir M. Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. Type4Py: Practical deep similarity learning-based type inference for Python. In *International Conference on Software Engineering (ICSE)*, pages 2241–2252, May 2022. doi:10.1145/3510003.3510124.

28 Joonyoung Park, Jihyeok Park, Dongjun Youn, and Sukyoung Ryu. Accelerating javascript static analysis via dynamic shortcuts. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1129–1140, 2021. doi:10.1145/3468264.3468556.

29 Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. Static inference meets deep learning: a hybrid type inference approach for Python. In *International Conference on Software Engineering (ICSE)*, pages 2019–2030, May 2022. doi:10.1145/3510003.3510038.

30 Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. TypeWriter: Neural type prediction with search-based validation. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 209–220, November 2020. doi:10.1145/3368089.3409715.

31 Ingkarat Rak-amnouykit, Daniel McCrevan, Ana Milanova, Martin Hirzel, and Julian Dolby. Python 3 types in the wild: A tale of two type systems. In *Dynamic Languages Symposium (DLS)*, pages 57–70, 2020. doi:10.1145/3426422.3426981.

32 Ingkarat Rak-amnouykit, Ana Milanova, Guillaume Baudart, Martin Hirzel, and Julian Dolby. Principled and practical static analysis for Python: Weakest precondition inference of hyperparameter constraints. *Software – Practice and Experience (SP&E)*, 54(3):363–393, March 2024. doi:10.1002/spe.3279.

33 Atanas Rountev, Ana L. Milanova, and Barbara G. Ryder. Points-to analysis for java using annotated constraints. In Linda M. Northrop and John M. Vlissides, editors, *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14-18, 2001*, pages 43–55. ACM, 2001. doi:10.1145/504282.504286.

34 Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. Pycg: Practical call graph generation in python. In *International Conference on Software Engineering (ICSE)*, pages 1646–1657, May 2021. doi:10.1109/ICSE43902.2021.00146.

35 Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: fast and precise sparse value flow analysis for million lines of code. *SIGPLAN Not.*, 53(4):693–706, June 2018. doi:10.1145/3192366.3192418.

36 Yannis Smaragdakis and Martin Bravenboer. Using Datalog for fast and easy program analysis. In *Datalog Reloaded*, 2011. doi:10.1007/978-3-642-24206-9_14.

37 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of javascript. In James Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 435–458. Springer, 2012. doi:10.1007/978-3-642-31057-7_20.

38 Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 265–266. ACM, 2016. doi:10.1145/2892208.2892235.

39 John Toman and Dan Grossman. Concerto: a framework for combined concrete and abstract interpretation. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290356.

40 Omer Tripp, Pietro Ferrara, and Marco Pistoia. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 49–59, 2014. doi:10.1145/2610384.2610385.

41 Raja Vallée-Rai. Soot: A Java bytecode optimization framework. Master's thesis, McGill University, Montreal, October 2000. URL: https://dl.acm.org/citation.cfm?id=782008.

42 Yin Wang. Pysonar2 - a semantic indexer for python with interprocedural type inference, December 2017. URL: https://github.com/yinwang0/pysonar2.

43 Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for javascript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 336–346, 2013. doi:10.1145/2483760.2483788.

44 Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. Python probabilistic type inference with natural language support. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 607–618, 2016. doi:10.1145/2950290.2950343.

45 Yanyan Yan. DLInfer dataset, January 2023. doi:10.5281/zenodo.7575545.

46 Yanyan Yan, Yang Feng, Hongcheng Fan, and Baowen Xu. DLInfer: Deep learning with static slicing for python type inference. In *International Conference on Software Engineering (ICSE)*, pages 2009–2021, May 2023. doi:10.1109/ICSE48619.2023.00170.

47 Yi Yang, Ana L. Milanova, and Martin Hirzel. Complex Python features in the wild. In *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*, pages 282–293. ACM, May 2022. doi:10.1145/3524842.3528467.

A Appendix

A.1 Per-application Equivalence Comparisons including Bars for Non-Empty Variables

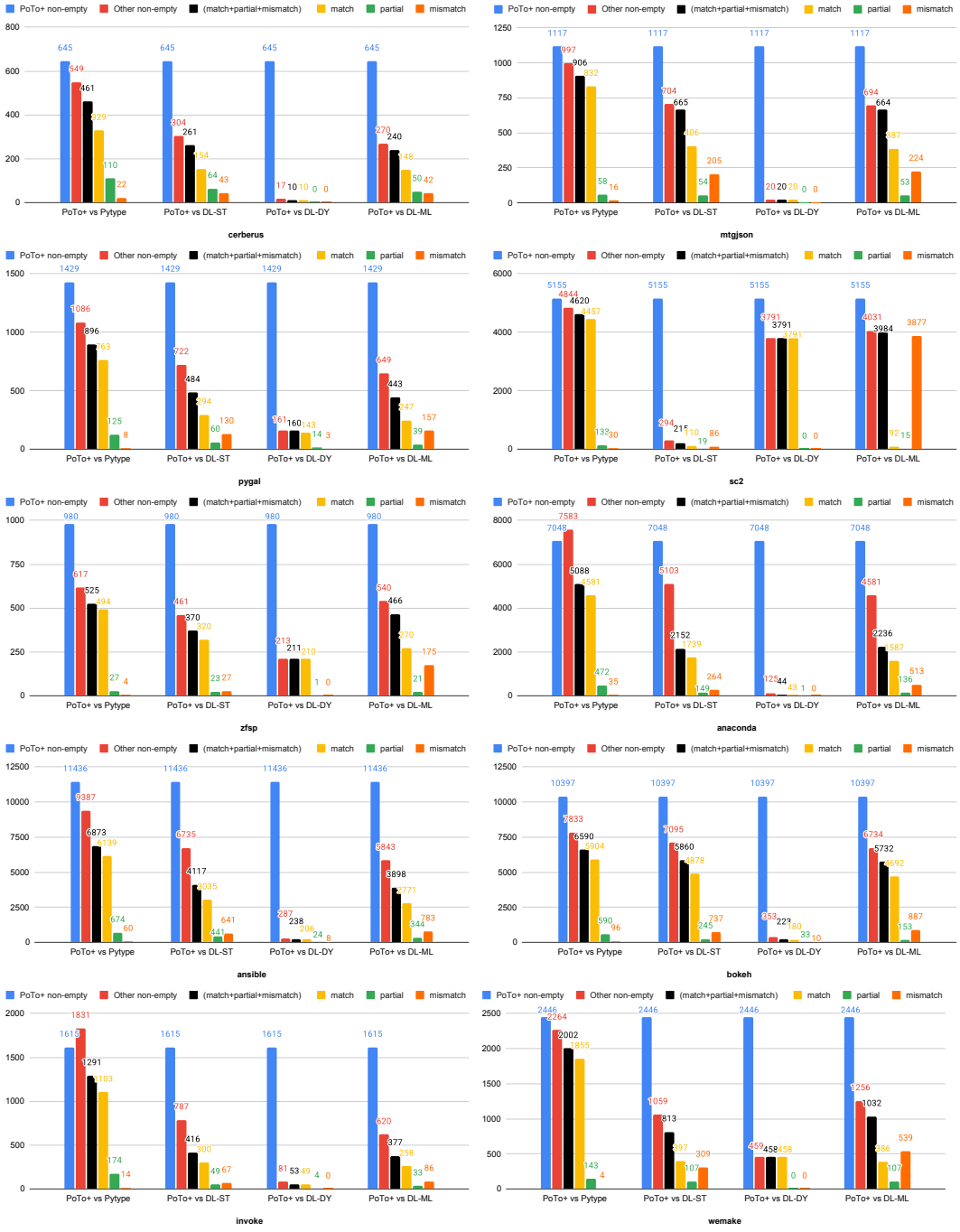


Figure 12 Equivalence comparison between PoTo+ and other type inference techniques (RQ2) for each individual package.

A.2 Difficult-to-Analyze Call Sites

```
1  class PublicApi:
2     ...
3     def render(self):
4       self.setup()
5       self.svg.render()
6
7  class BaseGraph(PublicApi):
8     ...
9     def setup(self):
10      self.svg = Svg(self)
11
12 class Line(BaseGraph):
13    ...
14    # Defines several new functions
15    # Does not override render or setup
16
17 # A test:
18 line1 = Line()
19 line1.show_legend = False
20 line1.fill = True
21 line1.pretty_print = True
22 line1.no_prefix = True
23 line1.x_labels = ['a', 'b', 'c']
24 line1.add('_', [1, 2, 3])
25 line1.render()
26
27 line2 = Line(show_legend=False, fill=True, pretty_print=True, no_prefix=True,
       x_labels=['a', 'b', 'c'])
28 line2.add('_', [1, 2, 3])
29 line2.render()
```

The listing shows an excerpt of a hierarchy from pygal. The majority of functionality is in base classes `PublicApi` and `BaseGraph` and the initialization context for graphing objects (e.g., line) consists largely of boolean, string, and numerical values. In this example the two line objects are initialized with the same values, as the test case tests their equivalence, but the important point is the types of encapsulated objects (booleans and lists of strings or integers), not the values themselves.

The call at line 5 illustrates a performance advantage of PoTo over a 1-object-sensitive analysis. Clearly, PoTo resolves the call to `Svg.render()`. A 1-object-sensitive points-to analysis will replicate functions `render` and `setup` per each one of the line objects hoping to distinguish between the two initialization contexts. However, replication is redundant as field `svg` is initialized in each replica of `setup` to the same `Svg` object, which flows into each replica of `render`; the analysis therefore discovers the same call target twice. While this is one example and we do need further studies, we observed similar flows in pygal and other applications – initialization context is of values of simple types that have no impact on the flow of objects and no impact on the call graph.

Reusing Caches and Invariants for Efficient and Sound Incremental Static Analysis

Mamy Razafintsialonina
Université Paris-Saclay, CEA, List, Palaiseau, France
Sorbonne Université, CNRS, LIP6, Paris, France

David Bühler
Université Paris-Saclay, CEA, List, Palaiseau, France

Antoine Miné
Sorbonne Université, CNRS, LIP6, Paris, France

Valentin Perrelle
Université Paris-Saclay, CEA, List, Palaiseau, France

Julien Signoles
Université Paris-Saclay, CEA, List, Palaiseau, France

Abstract

Static analysis by means of abstract interpretation is a tool of choice for proving absence of some classes of errors, typically undefined behaviors in C code, in a sound way. However, static analysis tools are hardly integrated in CI/CD processes. One of the main reasons is that they are still time- and memory-expensive to apply after every single patch when developing a program. For solving this issue, incremental static analysis helps developers quickly obtain analysis results after making changes to a program. However, existing approaches are often not guaranteed to be sound, limited to specific analyses, or tied to specific tools. This limits their generalizability and applicability in practice, especially for large and critical software. In this paper, we propose a generic, sound approach to incremental static analysis that is applicable to any abstract interpreter. Our approach leverages the similarity between two versions of a program to soundly reuse previously computed analysis results. We introduce novel methods for summarizing functions and reusing loop invariants. They significantly reduce the cost of reanalysis, while maintaining soundness and a high level of precision. We have formalized our approach, proved it sound, implemented it in Eva, the abstract interpreter of Frama-C, and evaluated it on a set of real-world commits of open-source programs.

2012 ACM Subject Classification Theory of computation → Program reasoning; Software and its engineering → Software verification and validation; Software and its engineering → Formal methods; Software and its engineering → Software notations and tools

Keywords and phrases Abstract Interpretation, Static Analysis, Incremental Analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.28

Related Version *Extended Version*: `https://hal.science/hal-05110966`

Supplementary Material *Software (Source Code)*: `https://git.frama-c.com/pub/frama-c/-/tree/publi/2025-ecoop-rbmps`
archived at `swh:1:dir:7b4b0787dd0a308bbd0abd211d2f4174224aebba`
Software (ECOOP 2025 Artifact Evaluation approved artifact):
`https://doi.org/10.4230/DARTS.11.2.15`

Funding This work was funded by project ANR-22-PECY-0005 "Secureval" managed by the French National Research Agency for France 2030.

Acknowledgements We thank the anonymous reviewers for their valuable feedback, which improved the final version of the paper.

© Mamy Razafintsialonina, David Bühler, Antoine Miné, Valentin Perrelle, and Julien Signoles;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

Abstract interpretation [10, 9] is a sound flow-sensitive static analysis technique that allows the user to prove absence of programming errors, such as undefined behaviors in C code. To be precise enough, it is also usually implemented as a context-sensitive analysis, i.e., the analyzer relies on the function context (such as values of arguments and global variables) when analyzing a function call. However, soundness and preciseness come at a cost: Abstract interpretation can be time- and memory-expensive when analyzing real-world programs. It is not necessarily an issue when analyzing a program once, but it can prevent developers from introducing such a static analysis technique in a CI/CD process to check that their changes do not introduce bugs: Even a seemingly minor syntactic change can have a huge impact on the behavior of the software.

Incremental analyses are one way to reduce the cost of analyzing software after a change by reusing as many results as possible from the analysis of a previous version when analyzing the new version of the code. However, providing a sound flow- and context-sensitive incremental analysis is hard: existing approaches are either unsound [22, 27] or limited to specific classes of analyses [35, 16] or tied to specific analysis tools [1, 33, 28]. This paper presents a generic abstract-interpretation based approach for incremental flow- and context-sensitive analysis which is sound, and supports unbounded memory locations. Our approach is based on soundly reusing analysis results from previous analysis runs and is designed to be is applicable to most existing abstract interpreters. For each type of analysis result to be reused, we provide the conditions under which soundness is guaranteed. Such conditions typically involve program matching, which is the process of determining whether two parts of a program are equivalent or not. We consider two different types of analysis results: Function summaries and loop invariants. For function summaries, we propose a new way to summarize the effect of the analysis of a function in order to avoid re-analyzing the body of a function when the context of the analysis is the same. Our function summaries are soundly reusable and always precise for functions that have not been impacted by the modification of the program. For loop invariants, we propose a way to accelerate the analysis of loops by reusing the loop invariants from a previous analysis of the loop. Loop invariants are always soundly reusable even if the loop has been modified, but they may introduce imprecision in the analysis. However, we provide a way to limit the loss of precision and show that in practice, only little imprecision are introduced. Both approaches are complementary and can be used together to accelerate a static analysis, either an ongoing one or an incremental one.

Both approaches are formalized and proved sound. They have been implemented in Eva [6], the abstract interpreter of Frama-C [3, 19], an analysis framework for C code. This implementation has been evaluated on a set of real-world commits of open-source programs: The evaluation shows that our incremental analysis is much more time-efficient than the original analysis when analyzing two successive commits, while remaining precise in practice. Therefore, we think that such an incremental analysis helps integrate sound static analyzers in CI/CD process.

Contributions

- An approach for summarizing functions and reusing the summaries in subsequent analyses;
- An approach for reusing loop invariants in subsequent analyses;
- A formalization and a proof of soundness for both approaches for an imperative language with dynamic allocations;
- An experimental evaluation on real-world commits of open-source C programs for both approaches.

The rest of the paper is organized as follows. Section 2 presents the background on abstract interpretation. Section 3 presents an example to illustrate our approach for reusing function summaries and loop invariants. Section 4 introduces a representative programming language and its formal semantics. Using this language, Sections 5 and 6 formalize our approaches for function summaries and loop invariants, respectively. Section 7 presents an implementation and evaluation of both approaches. Section 8 discusses related work. Section 9 concludes the paper and discusses future work.

2 Elements of Abstract Interpretation

This section introduces the basic elements of abstract interpretation that are used in the rest of the paper. Abstract interpretation [10, 9] is a general theory of sound approximation of program semantics. It is based on the idea of computing an over-approximation of the set of possible states of a program using abstract domains, and provides a formal guarantee that the computed over-approximation is sound, i.e., it contains all possible states of the program. More formally, starting from a so-called concrete semantic domain $\mathcal{D}$, which can express the properties of interest (such as the set of memory states, to prove non-reachability of erroneous states), but is too difficult to compute with, abstract interpretation introduces the concept of abstract domain $\mathcal{D}^\sharp$ where the elements of $\mathcal{D}^\sharp$ are sound approximations of the elements of $\mathcal{D}$. The domains $\mathcal{D}$ and $\mathcal{D}^\sharp$ are partially ordered by a relation $\sqsubseteq$ and $\sqsubseteq^\sharp$. We also define the concretization function $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$ as a monotonic function that maps an abstract element to a concrete element. We require that the abstract domain provides a sound approximation of the concrete join $\sqcup$ and meet $\sqcap$ operators, such that for all $x^\sharp, y^\sharp$ in $\mathcal{D}^\sharp$, $\gamma(x^\sharp) \sqcup \gamma(y^\sharp) \sqsubseteq \gamma(x^\sharp \sqcup^\sharp y^\sharp)$ and $\gamma(x^\sharp) \sqcap \gamma(y^\sharp) \sqsubseteq \gamma(x^\sharp \sqcap^\sharp y^\sharp)$. Moreover, the least upper bound $\top^\sharp$ (resp. greatest lower bound $\bot^\sharp$) of a set of abstract elements, when it exists, approximates the least upper bound $\top$ (resp. greatest lower bound $\bot$) of the corresponding set of concrete elements. A semantic operator is a function over domains. Abstract interpretation often requires computing fixpoints of semantic functions, e.g. in case of loops or recursive functions. They can be computed by iterating an abstract operator over an abstract domain. The convergence of the iteration sequence in the abstract domain is guaranteed by the use of a widening operator $\nabla^\sharp$ [10]. Formally, if $f : \mathcal{D} \mapsto \mathcal{D}$ is a monotonic operator over a complete concrete lattice $(\mathcal{D}, \sqsubseteq)$ and $f^\sharp : \mathcal{D}^\sharp \mapsto \mathcal{D}^\sharp$ is a sound abstraction of f, then the limit of the iteration sequence $x_0^\sharp = \bot^\sharp$, $x_{n+1}^\sharp = x_n^\sharp \nabla^\sharp f^\sharp(x_n^\sharp)$ converges in finite time to a stable iterate $x^\sharp$, which is a sound approximation of the least fixpoint lfp $f \sqsubseteq \gamma(x^\sharp)$. The existence of lfp f and the soundness of the approximation are guaranteed by Tarski's theorem [11, 34]. Moreover, the abstract operator $f^\sharp$ is not required to be monotonic, and does not necessarily admit a least fixpoint. We can settle for an abstract post-fixpoint $f^\sharp(x^\sharp) \sqsubseteq^\sharp x^\sharp$. Moreover, any post-fixpoint is an inductive invariant and the least fixpoint is the least invariant, so the limit of the iteration sequence is also an invariant. Figure 1 presents an interval abstract domain $\mathcal{I}_\mathbb{Z}^\sharp$, which represents values of integers as a pair of bounds, and a widening operator for fast and sound convergence of the iteration sequence.

3 Motivating Example

This section introduces an example that illustrates our contributions. Consider the following (buggy) C program that stores squares' size in an array before computing their area.

$$\mathcal{I}_{\mathbb{Z}}^{\sharp} \triangleq \{\ [a,b] \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{+\infty\}, a \leq b\ \} \cup \{\bot_{\mathcal{I}_{\mathbb{Z}}}^{\sharp}\}$$

$$[a,b]\ \nabla_{\mathcal{I}_{\mathbb{Z}}}^{\sharp}\ [c,d] \triangleq \left[\begin{cases} a & \text{if } a \leq c \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geq d \\ +\infty & \text{otherwise} \end{cases} \right]$$

Figure 1 Interval abstract domain and associated widening operator.

```
1  #define LEN 100
2  int area(int n) { return n + n; /* bug: should be 'n * n' */ }
3  void main() {
4    int sizes[LEN], areas[LEN];
5    for(int i = 0; i < LEN; i++)
6      sizes[i] = i;
7    for(int i = 0; i <= LEN; i++)  /* bug: should be 'i < LEN' */
8      areas[i] = area(sizes[i/2]);
9  }
```

This program contains one functional bug at line 2 and one undefined behavior at line 8 because of an incorrect test at line 7. Any modern abstract interpreter handling C code, such as Astrée [23], the Eva [4, 6] plugin of Frama-C [3, 19], Mopsa [24], or Polyspace [21], is able to analyze such a code instantaneously and reports one alarm. This alarm is raised when writing into the array `areas[100]`. Indeed, the loop at line 7 iterates one time too many, which results in an out-of-bound access. Such modern analyzers often implement function summaries that allow them to avoid analyzing twice a function under the same analysis context. An analysis context of a function is an over-approximation of the memory state at the call site, including the values of the arguments.

In our example, running an interval analysis with a standard widening operator allows the analyzer to converge in four iterations and to only analyze the function `area` twice instead of four times (i.e., one for each loop iteration). Indeed, for each loop iteration, the analysis context of the function call at line 8, in which `sizes` is omitted because it does not change throughout the analysis, is: $\{i \mapsto [0,0]; areas[i] \mapsto uninit; n \mapsto [0,0]\}$ at iteration 1, $\{i \mapsto [0,1]; areas[i] \mapsto [0,0]; n \mapsto [0,0]\}$ at iteration 2, $\{i \mapsto [0,100]; areas[i] \mapsto [0,100]; n \mapsto [0,50]\}$ at iteration 3 (with widening), and $\{i \mapsto [0,100]; areas[i] \mapsto [0,100]; n \mapsto [0,50]\}$ at iteration 4. We observe that the analysis of the function `area` depends on the value of its argument n (which is given by the value of array `sizes` at index `i/2`), and not on the value of the array `areas`. By assuming a sound over-approximation of the variables needed to analyze the function `area`, an analyzer can build a function summary that restricts the values of the state of the analysis before the call to `area` to the values of the variables needed to analyze the function. Such a state is then used to avoid the analysis of the body of the function again if the values of the needed variables are the same. The function summary in this case contains the state of the analysis before the call (denoted as input state $I^{\sharp}$) to function `area`, the state of the analysis after the call (denoted as output state $\mathcal{O}^{\sharp}$) to `area`, and the set (or sets) of variables needed to analyze the function. For example, the function summary for the call at line 8 after the first loop iteration is $\{I^{\sharp} = \{n \mapsto [0,0]\}; \mathcal{O}^{\sharp} = \{areas[i] \mapsto [0,0]\}; R^{\sharp} = \{n\}; W^{\sharp} = \{areas[i]\}\}$, where $R^{\sharp}$ (resp. $W^{\sharp}$) is the set of variables read (resp. written) by the function `area`. When analyzing the same function again during the second loop iteration, the value of the read variable n does not change, so the analyzer can reuse the function summary to avoid

reanalyzing the body of function `area`: It directly computes the state of the analysis after the call to function `area` from the output state $\mathcal{O}^\sharp$ and the written variables $W^\sharp$. Our first contribution consists in formalizing such function summaries for soundly reusing them when possible in order to speed up the analysis.

If such an analyzer is part of a CI/CD process, the developers would easily detect the runtime error at line 7 and fix it by submitting the following patch:

```
- for (i = 0; i <= LEN; i++)
+ for (i = 0; i < LEN; i++)
```

Analyzing the code again will allow them to prove that there is no more out-of-bound access in the program. However, for large programs, this analysis can be time-consuming, which may prevent its use in a CI/CD process. Indeed, it would require to run a new analysis from scratch for every program change, even minimal ones. This paper explains how to reuse previously computed function summaries from a previous analysis in order to prevent abstract interpreters from reanalyzing the body of the function `area` when reanalyzing the (almost) same program after a small patch. In fact, the analysis of the call to `area` was already done prior to the patch for the call contexts $\{n \mapsto [0,0]\}$ and $\{n \mapsto [0,50]\}$. The context $\{n \mapsto [0,0]\}$ can be directly reused when analyzing the patched version, so the analyzer only needs to analyze the call for the context $\{n \mapsto [0,49]\}$.

Later, the functional bug at line 2 will also be fixed by the following commit:

```
- int area(int n) { return n + n; }
+ int area(int n) { return n * n; }
```

Modifying the function `area` results in invalidating all function summaries inferred for this function and for all functions that call it. Therefore, the analyzer will analyze again from scratch the functions `area` and `main`.

Our second contribution consists in reusing previously computed loop invariants to speed up the analysis of both loops in our running example in order to speed up the analysis even when no function summaries can be soundly reused. Indeed, prior to the patch for the functional bug, the analyzer has already computed the loop invariants for the loops at lines 5 and 7. If $arr[a,b] \mapsto [i,j]$ denotes *the array arr contains values in the interval* $[i,j]$ *for any index in the interval* $[a,b]$, these invariants are $\{sizes[0,99] \mapsto [0,99]; areas[0,99] \mapsto uninit; i \mapsto [0,99]\}$ and $\{sizes[0,99] \mapsto [0,99]; areas[0,99] \mapsto [0,198]; i \mapsto [0,99]\}$, respectively. Reusing them will allow the analyzer to converge faster by starting the iteration from the previously computed fixpoint for this loop instead of the least element $\bot^\sharp$. This optimization allows the first loop to converge in one iteration only as the fixpoint has not changed, while the second loop will converge in two iterations as the value of $areas[0,49]$ still needs to stabilize to $[0,2401]$ ($2401 = 49 \times 49$).

Together, our contributions allow analyzers to reduce the analysis time significantly when submitting small patches that fix bugs or introduce new features on large programs (see Section 7). Therefore, they help integrate such analyzers in a CI/CD process.

4 Language and Semantics

Figure 2 presents our language. It is an imperative language containing basic control flows (sequences, conditionals and loops), assignments, function calls and dynamic allocations, which allocate new memory locations of type integer at a given allocation site μ. Expressions consist of integer constants, variables, dereferences, addresses of a variable, and basic n-ary

$$
\begin{array}{lll}
\textbf{expr} & e ::= c & c \in \mathbb{V} \\
& \mid *e & \text{(dereference)} \\
& \mid \&x & \text{(address of } x\text{)},\ x \in \mathbb{X} \\
& \mid \diamondsuit(e_1, \ldots, e_n) & \diamondsuit \in \{+, -, *, /\} \\
\textbf{stmt} & s ::= *e_1 \coloneqq e_2 & \text{(assignment)} \\
& \mid *(\&x) \coloneqq \text{malloc}_\mu & \text{(allocation)},\ \mu \in \mathbb{L} \\
& \mid s_\alpha ; s_\beta & \text{(sequence)} \\
& \mid f(\vec{e}) & \text{(function call)} \\
& \mid \textbf{if}\ (e \bowtie 0)\ \textbf{then}\ s_\alpha\ \textbf{else}\ s_\beta & \text{(conditional)} \\
& \mid \textbf{while}\ (e \bowtie 0)\ \textbf{do}\ s & \text{(loop)} \\
\textbf{fun} & f ::= f(\vec{x})\{s\} & \text{(function definition)} \\
& \bowtie \in \{=, \neq, <, >, \leq, \geq\} & \text{(relational operator)}
\end{array}
$$

Figure 2 Syntax of our language.

arithmetic operators over integers. Functions are defined with a unique name f, a list of formal parameters $\vec{x}$, and a body of statements s. Variables denote integers or memory locations.

Notations

We denote $\mathbb{X}$, the finite set of all variables and formal parameters of functions; $\mathbb{L}$, the finite set of allocation sites; $\mathbb{H} = \mathbb{L} \times \mathbb{N}$, pairs of allocation site and the number of allocations at this site that represents a memory block dynamically allocated in the heap; $\mathbb{A} = \mathbb{X} \uplus \mathbb{H}$, the set of all possible memory locations; $\mathbb{V} = \mathbb{Z} \uplus \mathbb{A}$, the set of all possible values (either an integer or a memory location) assignable to a variable; and $\mathbb{M} : \mathbb{A} \rightharpoonup \mathbb{V}$, a memory state that maps a memory location to some (concrete) value. $\mapsto$ is the update operator that assigns a value to a memory location in a memory state, such that $m[a \mapsto v]$ denotes the memory state $m \in \mathbb{M}$ that maps the location $a \in \mathbb{A}$ to the value $v \in \mathbb{V}$ and other elements $a' \in \text{dom}(m)$ to their previous value $m(a')$. $|$ is the restriction operator that removes the keys of a mapping, such that $m|_A$ is the memory state m restricted to the set A of memory locations. Additionally, we denote $e^\sharp$ an element of an abstract domain. Furthermore, the formal parameters of a function are given by params : $\textbf{fun} \to \mathcal{P}(\mathbb{X})$, the body of the function by body : $\textbf{fun} \to \textbf{stmt}$ and the local variables of the function by locals : $\textbf{fun} \to \mathcal{P}(\mathbb{X})$.

Assumptions and limitations

We do not handle pointer arithmetic: $[\![\diamondsuit]\!]$ is only defined for integers. Assignments can only write a memory location by dereferencing it. So, assigning some expression e to a variable $x \in \mathbb{X}$ is written $*(\&x) = e$. Every malloc_μ expression is assumed to be unique and $\mu \in \mathbb{L}$ identifies the allocation site. A same malloc_μ can be called multiple times during the execution of a program (typically, when written in a loop). In such a case, it generates multiple memory blocks at different addresses. Without loss of generality, all variable names are assumed to be unique, including formal parameters of defined functions. Functions have no return value: They can be simulated by passing to the function a reference to a variable

in which the function stores its result. Recursive function calls are also not supported, we already have functions, unbounded loops and unbounded memory allocation, so the support of recursive calls does not bring theoretical difficulties. It is just a technical limitation of the formalization and is left for future work.

4.1 Concrete Semantics

Figures 3 and 4 define the concrete semantic functions for expressions and statements, respectively. The semantics for expressions, $\mathbb{E}[\![\mathbf{expr}]\!]$, maps a concrete memory state to a concrete value. The semantics for statements, $\mathbb{S}[\![\mathbf{stmt}]\!]$, maps a concrete memory state to another concrete memory state: $\mathbb{S}[\![\mathbf{stmt}]\!]$ computes the state of the program reached at the end of the execution of a statement. The concrete semantics is deterministic and $\mathbb{S}[\![\mathbf{stmt}]\!]$ is a partial function to handle undefined operations such as divisions by zero or reading uninitialized memory locations. A function call is done by binding the value of the evaluation of the arguments to the formal parameters in the memory state, executing the body of the function, removing the formal parameters from the memory state at the end of the function execution, and returning this memory state. The semantics for dynamic allocations relies on a next operator: for a location site $\mu \in \mathbb{L}$ and a memory state $m \in \mathbb{M}$, $\text{next}(\mu, m)$ returns the first free memory block in $\mathbb{H}$ at this call site, which is formally defined as follows:

$$\text{next}(\mu, m) \triangleq \langle \mu, \min\{i \in \mathbb{N} \mid \langle \mu, i\rangle \notin \text{dom}(m)\}\rangle.$$

This definition and the concrete semantics for dynamic allocations imply that the first memory block allocated at μ is $\langle \mu, 0\rangle$, the second one is $\langle \mu, 1\rangle$, and so on.

$$\begin{aligned}
\mathbb{E}[\![\mathbf{expr}]\!] &: \mathbb{M} \to \mathbb{V} \\
\mathbb{E}[\![c]\!]m &\triangleq c,\ c \in \mathbb{Z} \\
\mathbb{E}[\![*e]\!]m &\triangleq m(\mathbb{E}[\![e]\!]m) \\
\mathbb{E}[\![\&x]\!]m &\triangleq x,\ x \in \mathbb{X} \\
\mathbb{E}[\![\diamondsuit(e_1, ..., e_n)]\!]m &\triangleq \diamondsuit(\mathbb{E}[\![e_1]\!]m, ..., \mathbb{E}[\![e_2]\!]m)
\end{aligned}$$

Figure 3 Concrete semantics for expressions.

4.2 Abstract Domains

Value Abstract Domain

We abstract the concrete values $\mathbb{V}$ by an abstract domain $\mathbb{V}^\sharp$, such that $(\mathbb{V}^\sharp, \sqsubseteq^\sharp_{\mathbb{V}})$ is a partial order abstracting the concrete domain $(\mathcal{P}(\mathbb{V}), \subseteq)$ featuring a monotonic concretization function $\gamma_{\mathbb{V}} : \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{V})$ and a widening operator $\nabla^\sharp_{\mathbb{V}}$. To construct such an abstract value domain, we need to abstract both integer values and memory locations.

The abstract domain $(\mathbb{Z}^\sharp, \sqsubseteq^\sharp_{\mathbb{Z}})$ for integer values is left unspecified. We just assume a concretization function $\gamma_{\mathbb{Z}} : \mathbb{Z}^\sharp \to \mathcal{P}(\mathbb{Z})$ and a widening operator $\nabla^\sharp_{\mathbb{Z}}$. For instance, this abstract domain is the interval domain in our running example. The set of variables is finite, so variable sets can be abstracted by its powerset $(\mathcal{P}(\mathbb{X}), \subseteq)$. An abstract heap address represents a set of concrete heap addresses [17]. Consequently, memory updates on abstract heap addresses are done by merging the new value with the previous value, which is called a

$$
\begin{aligned}
\mathbb{S}[\![\mathbf{stmt}]\!] &: \mathbb{M} \rightharpoonup \mathbb{M} \\
\mathbb{S}[\![*e_1 := e_2]\!]m &\triangleq m[\mathbb{E}[\![e_1]\!]m \mapsto \mathbb{E}[\![e_2]\!]m] \\
\mathbb{S}[\![*(\&x) := \mathrm{malloc}_\mu]\!]m &\triangleq \text{let } \langle \mu, n\rangle := \mathrm{next}(\mu, m) \text{ in} \\
&\quad\; m[\mathbb{E}[\![\&x]\!]m \mapsto \langle \mu, n\rangle][\langle \mu, n\rangle \mapsto 0] \\
\mathbb{S}[\![f(\vec{e})]\!]m &\triangleq \text{let } \vec{x} := \mathrm{params}(f) \text{ in} \\
&\quad\; \text{let } \vec{y} := \mathrm{locals}(f) \text{ in} \\
&\quad\; \text{let } m_{\mathrm{ret}} := \mathbb{S}[\![\mathrm{body}(f)]\!](m[\vec{x} \mapsto \mathbb{E}[\![\vec{e}]\!]m]) \text{ in} \\
&\quad\; m_{\mathrm{ret}}|_{\mathrm{dom}(m_{\mathrm{ret}}) \setminus (\vec{x} \cup \vec{y})} \\
\mathbb{S}[\![s_\alpha; s_\beta]\!]m &\triangleq (\mathbb{S}[\![s_\beta]\!] \circ \mathbb{S}[\![s_\alpha]\!])m \\
\mathbb{S}[\![\mathbf{if}\ (e \bowtie 0)\ \mathbf{then}\ s_\alpha\ \mathbf{else}\ s_\beta]\!]m &\triangleq \mathbf{if}\ (\mathbb{E}[\![e]\!]m \bowtie 0)\ \mathbf{then}\ \mathbb{S}[\![s_\alpha]\!]m\ \mathbf{else}\ \mathbb{S}[\![s_\beta]\!]m \\
\mathbb{S}[\![\mathbf{while}\ (e \bowtie 0)\ \mathbf{do}\ s]\!]m &\triangleq \mathrm{lfp}(F)(m) \\
&\quad \text{where } F(f)(m) \triangleq \begin{cases} f(\mathbb{S}[\![s]\!]m) & \text{if } \mathbb{E}[\![e]\!]m \bowtie 0 \\ m & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 4 Concrete semantics for statements.

weak update. The induced precision loss with weak updates can be compensated by using more sophisticated techniques to summarize heap addresses [18]. The idea is to allow some abstract heap addresses to represent a singleton concrete heap address and allow *strong updates* on such addresses. We choose to represent abstract heap addresses $\mathbb{H}^\sharp_k : \mathbb{L} \times \mathbb{N}_{\leq k}$ as a pair of allocation site $\mu \in \mathbb{L}$ and a natural number smaller or equal than k.

Such a technique is based on *k-limiting* [20] and *allocation site* [2]. The natural number k is used to limit the maximum number of generation of strong abstract heap addresses on a given site before over-approximating further allocations with a weak abstract heap address. Such limit k will be referred as $\mathrm{k_{lim}}$ or k for subscript in the rest of the paper. More precisely, we want to represent the first k allocations on a site μ with a dedicated abstract heap address $\langle \mu, k\rangle$ for all $0 \leq k < \mathrm{k_{lim}}$, and all further allocations with an abstract heap address $\langle \mu, \mathrm{k_{lim}}\rangle$. Consequently, when an abstract heap address represents a singleton concrete heap address, the update on this address is a strong update: the new value replaces the previous one. Otherwise, if an abstract heap address potentially represents multiple concrete heap addresses, the update is a weak update: the new value is merged with the previous one, which is the case for $\langle \mu, \mathrm{k_{lim}}\rangle$. Formally, the abstract value domain $\mathbb{V}^\sharp$ is defined as follows:

$$
\begin{aligned}
\mathbb{V}^\sharp &\triangleq \mathbb{Z}^\sharp \times \mathcal{P}(\mathbb{H}^\sharp_k) \times \mathcal{P}(\mathbb{X}) \cup \{\top^\sharp_{\mathbb{V}}, \bot^\sharp_{\mathbb{V}}\} \\
\gamma_{\mathbb{V}} &: \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{V}) \\
\gamma_{\mathbb{V}}(\top^\sharp_{\mathbb{V}}) &\triangleq \mathbb{V} \\
\gamma_{\mathbb{V}}(\bot^\sharp_{\mathbb{V}}) &\triangleq \emptyset \\
\gamma_{\mathbb{V}}(z^\sharp, h^\sharp, x^\sharp) &\triangleq \gamma_{\mathbb{Z}}(z^\sharp) \cup \gamma_{\mathbb{H}}(h^\sharp) \cup x^\sharp \\
&\quad \text{where } \gamma_{\mathbb{H}}(h^\sharp) \triangleq \bigcup \{\gamma_h(\langle \mu, n\rangle) \mid \forall \langle \mu, n\rangle \in h^\sharp\} \\
&\quad \text{where } \gamma_h(\langle \mu, n\rangle) \triangleq \begin{cases} \{\langle \mu, n\rangle\} & \text{if } n < k \\ \{\langle \mu, j\rangle \mid \forall j \geq k\} & \text{if } n = k. \end{cases}
\end{aligned}
$$

The definitions of operators $\sqsubseteq^\sharp_{\mathbb{V}}$, $\sqcup^\sharp$, $\sqcap^\sharp$ and $\nabla^\sharp$ are omitted, but defined pointwise and extended to $\top^\sharp_{\mathbb{V}}$ and $\bot^\sharp_{\mathbb{V}}$ such that the former is the greatest element of $\mathbb{V}^\sharp$ and the latter is its least element for the partial order $\sqsubseteq^\sharp$.

Non-relational Memory Abstract Domain

We derive the abstraction $\mathbb{M}^\sharp$ of $\mathbb{M} : \mathbb{A} \rightharpoonup \mathbb{V}$ from the abstract value domain $\mathbb{V}^\sharp$ by mapping each memory location in the finite set of abstract memory locations $\mathbb{A}^\sharp_k : \mathbb{X} \uplus \mathbb{H}^\sharp_k$ to an abstract value $\mathbb{V}^\sharp$. The abstract memory domain $\mathbb{M}^\sharp$ is thus a partial mapping from abstract memory locations to abstract values. The operations $\sqsubseteq^\sharp, \nabla^\sharp, \sqcup^\sharp, \sqcap^\sharp$ are defined pointwise, and the least element $\bot^\sharp_{\mathbb{V}}$ is used when one of the operands is undefined . The least element $\bot^\sharp$ (resp. the greatest element $\top^\sharp$) maps each memory location in $\mathrm{dom}(\mathbb{M}^\sharp)$ to the least element $\bot^\sharp_{\mathbb{V}}$ (resp. the greatest element $\top^\sharp_{\mathbb{V}}$) of the abstract value domain. The concretization function γ takes into account possible undefined memory locations. Formally, the abstract memory domain is defined as follows:

$$\mathbb{M}^\sharp \triangleq (\mathbb{A}^\sharp_k \rightharpoonup \mathbb{V}^\sharp)$$
$$\gamma : \mathbb{M}^\sharp \to \mathcal{P}(\mathbb{M})$$
$$\gamma(M^\sharp) \triangleq \left\{ m \in \mathbb{M} : \left\{ \begin{array}{ll} m(\langle a, n\rangle) \in \gamma_{\mathbb{V}}(M^\sharp(\langle a, n\rangle)) & (1) \\ m(\langle a, n\rangle) \in \gamma_{\mathbb{V}}(M^\sharp(\langle a, k\rangle)) & (2) \\ m(x) \in \gamma_{\mathbb{V}}(M^\sharp(x)) & (3) \end{array} \right. \middle| \; \forall a : j_a \in \mathbb{N} \right\}$$

where (1) $\forall \langle a, n\rangle \in \mathrm{dom}(M^\sharp)$ and $n < k$ and $n < j_a$
(2) if $\langle a, k\rangle \in \mathrm{dom}(M^\sharp)$ and $k \leq n \leq j_a$
(3) $\forall x \in \mathbb{X}$ and $x \in \mathrm{dom}(M^\sharp)$

4.3 Abstract Semantics

Figures 5 and 6 present the abstract semantics of our language, which computes an over-approximation of the concrete semantics. The abstract semantics for expressions, $\mathbb{E}^\sharp[\![\mathbf{expr}]\!]$, maps an abstract memory to an abstract value, while the abstract semantics for statements, $\mathbb{S}^\sharp[\![\mathbf{stmt}]\!]$, maps an abstract memory to another abstract memory. $\mathbb{S}^\sharp[\![\mathbf{stmt}]\!]$ computes an over-approximation of the possible states reached at the end of the executed statement. Function $\phi_{\mathbb{Z}} : \mathbb{Z} \to \mathbb{Z}^\sharp$ (resp. $\diamondsuit^\sharp$) is assumed to be a sound abstraction of integer constants (resp. unary operators) given by the abstract domain $\mathbb{Z}^\sharp$ for integer values. We assume given a sound abstraction of relational operators $\bowtie^\sharp_e$: $\mathbb{M}^\sharp \to \mathbb{M}^\sharp$ and its negation $\not\bowtie^\sharp_e : \mathbb{M}^\sharp \to \mathbb{M}^\sharp$ such that for all non-relational abstract elements $M^\sharp \in \mathbb{M}^\sharp$ and all concrete memory states $m \in \mathbb{M}$, if $\mathbb{E}[\![e]\!]m \bowtie 0$ then $m \in \gamma(\bowtie^\sharp_e (M^\sharp))$, and its negation if $\mathbb{E}[\![e]\!]m \not\bowtie 0$ then $m \in \gamma(\not\bowtie^\sharp_e(M^\sharp))$.

The abstract semantics of an assignment performs an abstract memory update, which is either a strong update when the written address is precise, or a weak update when it is not. The formal definition of such a memory update is as follows:

▶ **Definition 1** (Memory Update). *Given an abstract memory $M^\sharp \in \mathbb{M}^\sharp$, a set of abstract memory locations $X \in \mathcal{P}(\mathbb{A}^\sharp_k)$ and an abstract value $v^\sharp \in \mathbb{V}^\sharp$, the update of $M^\sharp$ with X is defined as follows:*

$$\forall X \in \mathcal{P}(\mathbb{A}^\sharp_k), \forall v^\sharp \in \mathbb{V}^\sharp :$$
$$M^\sharp[X \rightsquigarrow v^\sharp] \triangleq \begin{cases} M^\sharp[a^\sharp \mapsto v^\sharp] & \text{if } X = \{a^\sharp\} \wedge \\ & (a^\sharp \in \mathbb{X} \vee (a^\sharp = \langle \mu, n\rangle \wedge n < k)) \\ & \text{strong update} \\ M^\sharp[\forall a^\sharp \in X. a^\sharp \mapsto v^\sharp \sqcup^\sharp_{\mathbb{V}} M^\sharp(a^\sharp)] & \text{otherwise weak update} \end{cases}$$

$$
\begin{aligned}
\mathbb{E}^\sharp[\![\mathbf{expr}]\!] &: \mathbb{M}^\sharp \to \mathbb{V}^\sharp \\
\mathbb{E}^\sharp[\![c]\!]M^\sharp &\triangleq \phi_{\mathbb{Z}}(c) \\
\mathbb{E}^\sharp[\![*e]\!]M^\sharp &\triangleq \bigsqcup_{\mathbb{V}}^\sharp \{M^\sharp(a^\sharp) \mid \forall a^\sharp \in \mathbb{E}^\sharp[\![e]\!]M^\sharp\} \\
\mathbb{E}^\sharp[\![\&x]\!]M^\sharp &\triangleq \{x\} \\
\mathbb{E}^\sharp[\![\diamondsuit(e_1,...,e_n)]\!]M^\sharp &\triangleq \diamondsuit^\sharp(\mathbb{E}^\sharp[\![e_1]\!]M^\sharp,\ldots,\mathbb{E}^\sharp[\![e_n]\!]M^\sharp)
\end{aligned}
$$

Figure 5 Abstract semantics for expressions.

$$
\begin{aligned}
\mathbb{S}^\sharp[\![\mathbf{stmt}]\!] &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\mathbb{S}^\sharp[\![*e_1 := e_2]\!]M^\sharp &\triangleq M^\sharp[\mathbb{E}^\sharp[\![e_1]\!]M^\sharp \rightsquigarrow \mathbb{E}^\sharp[\![e_2]\!]M^\sharp] \\
\mathbb{S}^\sharp[\![*(\&x) := \mathrm{malloc}_\mu]\!]M^\sharp &\triangleq \text{let } \langle \mu, n\rangle := \mathrm{next}^\sharp(\mu, M^\sharp) \text{ in} \\
&\qquad M^\sharp[\{x\} \mapsto \langle \mu, n\rangle][\langle \mu, n\rangle \rightsquigarrow Z_0^\sharp] \\
\mathbb{S}^\sharp[\![f(\vec{e})]\!]M^\sharp &\triangleq \text{let } \vec{x} := \mathrm{params}(f) \text{ in} \\
&\qquad \text{let } \vec{y} := \mathrm{locals}(f) \text{ in} \\
&\qquad \text{let } M^\sharp_{\mathrm{ret}} = \mathbb{S}^\sharp[\![\mathrm{body}(f)]\!]M^\sharp[\vec{x} \mapsto \mathbb{E}^\sharp[\![\vec{e}]\!]M^\sharp] \text{ in} \\
&\qquad M^\sharp_{\mathrm{ret}}|_{\mathrm{dom}(M^\sharp_{\mathrm{ret}})\setminus(\vec{x}\cup\vec{y})} \\
\mathbb{S}^\sharp[\![\mathbf{if}\ (e \bowtie 0)\ \mathbf{then}\ s_\alpha\ \mathbf{else}\ s_\beta]\!]M^\sharp &\triangleq (\mathbb{S}^\sharp[\![s_\alpha]\!] \circ \bowtie^\sharp_e)\mathbb{M}^\sharp \sqcup^\sharp (\mathbb{S}^\sharp[\![s_\beta]\!] \circ \bowtie\!\!\!

Soundness Requirements of Abstract Semantic Functions

The concrete semantic functions are implicitly lifted to sets. We require that the abstract semantic functions are sound abstractions of the corresponding concrete semantic functions. This is expressed by the following soundness criterion: $\forall M^\sharp \in \mathbb{M}^\sharp$, $\mathbb{S}[\![\mathbf{stmt}]\!]\gamma(M^\sharp) \subseteq \gamma(\mathbb{S}^\sharp[\![\mathbf{stmt}]\!]M^\sharp)$ and $\mathbb{E}[\![\mathbf{expr}]\!]\gamma(M^\sharp) \subseteq \gamma_{\mathbb{V}}(\mathbb{E}^\sharp[\![\mathbf{expr}]\!]M^\sharp)$.

Set of Abstract Read, Written and Allocated Memory Locations

We define the set of abstract memory locations used in expressions, read in statements, written in statements, and allocated in statements. These sets are used in the definition of function summaries and loop invariants. They are defined inductively over the semantics of statements. These locations are collected and propagated throughout each evaluation of an expression and a statement. We only present here the important cases of these definitions. The others are defined in a straightforward manner.

▶ **Definition 4** (Set of abstract memory locations used in expressions). *The set of abstract memory locations used in an expression* ***expr*** *is the set of memory locations which might change the result of the abstract evaluation of the expression.*

$$
\begin{aligned}
\Upsilon^\sharp[\![\boldsymbol{expr}]\!] &: \mathbb{M}^\sharp \to \mathcal{P}(\mathbb{A}_k^\sharp) \\
\Upsilon^\sharp[\![\&x]\!]M^\sharp &\triangleq \emptyset \\
\Upsilon^\sharp[\![*e]\!]M^\sharp &\triangleq \mathit{let}\ Y := \Upsilon^\sharp[\![e]\!]M^\sharp \cup \mathbb{E}^\sharp[\![e]\!]M^\sharp\ \mathit{in} \\
&\qquad \mathit{let}\ X := \mathbb{E}^\sharp[\![*e]\!]M^\sharp\ \mathit{in} \\
&\qquad\quad \begin{cases} Y & \mathit{if}\ |X| = 1 \\ & \wedge\ (\forall a^\sharp \in X : a^\sharp \in \mathbb{X} \vee (a^\sharp = \langle \mu, n \rangle \wedge n < k)) \\ X \cup Y & \mathit{otherwise} \end{cases} \\
\Upsilon^\sharp[\![\diamondsuit(e_1, ..., e_n)]\!]M^\sharp &\triangleq \bigcup_{\forall i \in [1,n]} \Upsilon^\sharp[\![e_i]\!]M^\sharp
\end{aligned}
$$

▶ **Definition 5** (Set of abstract read memory locations in statements). *An abstract memory location $a^\sharp$ is read by a statement* ***stmt*** *if its abstract value $M^\sharp(a^\sharp)$ is accessed by the computation of $\mathbb{S}^\sharp[\![\boldsymbol{stmt}]\!]M^\sharp$.*

$$
\begin{aligned}
\mathit{Read}^\sharp[\![\boldsymbol{stmt}]\!] &: \mathbb{M}^\sharp \to \mathcal{P}(\mathbb{A}_k^\sharp) \\
\mathit{Read}^\sharp[\![*e_1 := e_2]\!]M^\sharp &\triangleq \Upsilon^\sharp[\![e_1]\!]M^\sharp \cup \Upsilon^\sharp[\![e_2]\!]M^\sharp \\
\mathit{Read}^\sharp[\![*(\&x) := \mathit{malloc}_\mu]\!]M^\sharp &\triangleq \emptyset \\
\mathit{Read}^\sharp[\![f(\vec{e})]\!]M^\sharp &\triangleq \mathit{let}\ M_1^\sharp := M^\sharp[\mathit{params}(f) \mapsto \mathbb{E}^\sharp[\![\vec{e}]\!]M^\sharp]\ \mathit{in} \\
&\qquad \Upsilon^\sharp[\![\vec{e}]\!]M^\sharp \cup \mathit{Read}^\sharp[\![\mathit{body}(f)]\!]M_1^\sharp \setminus \mathit{locals}(f) \setminus \mathit{params}(f)
\end{aligned}
$$

▶ **Definition 6** (Set of abstract memory locations modified in statements). *An abstract memory location $a^\sharp$ is written by a statement* ***stmt*** *if it is a lvalue in an assignment statement.*

$$
\begin{aligned}
\mathit{Write}^\sharp[\![\boldsymbol{stmt}]\!] &: \mathbb{M}^\sharp \to \mathcal{P}(\mathbb{A}_k^\sharp) \\
\mathit{Write}^\sharp[\![*e_1 := e_2]\!]M^\sharp &\triangleq \mathbb{E}^\sharp[\![e_1]\!]M^\sharp \\
\mathit{Write}^\sharp[\![*(\&x) := \mathit{malloc}_\mu]\!]M^\sharp &\triangleq \{x\} \\
\mathit{Write}^\sharp[\![f(\vec{e})]\!]M^\sharp &\triangleq \mathit{let}\ M_1^\sharp := M^\sharp[\mathit{params}(f) \mapsto \mathbb{E}^\sharp[\![\vec{e}]\!]M^\sharp]\ \mathit{in} \\
&\qquad \mathit{Write}^\sharp[\![\mathit{body}(f)]\!]M_1^\sharp \setminus \mathit{params}(f) \setminus \mathit{locals}(f)
\end{aligned}
$$

▶ **Definition 7** (Set of abstract allocated memory locations in statements). *An abstract memory location $a^\sharp$ is allocated by a statement* ***stmt*** *if it is the result of a call to the allocation function.*

$$Alloc^\sharp[\![\textbf{\textit{stmt}}]\!] : \mathbb{M}^\sharp \to \mathcal{P}(\mathbb{A}_k^\sharp)$$
$$Alloc^\sharp[\![*(\&x) := malloc_\mu]\!]M^\sharp \triangleq \{next^\sharp(\mu, M^\sharp)\}$$
$$Alloc^\sharp[\![f(\vec{e})]\!]M^\sharp \triangleq Alloc^\sharp[\![body(f)]\!](M^\sharp[params(f) \mapsto \mathbb{E}^\sharp[\![\vec{e}]\!]M^\sharp])$$

▶ **Example 8.** Given a statement $x := \text{malloc}_\mu; *x := y$ (written in C-like for simplicity) and an abstract memory $M^\sharp$ such that $M^\sharp = \{\langle\mu, 0\rangle \mapsto 0, y \mapsto 1\}$, if $k_{\text{lim}} = 1$ we have:

$$\mathbb{S}^\sharp[\![.]\!]M^\sharp = M^\sharp[x \mapsto \langle\mu, 1\rangle, \langle\mu, 1\rangle \mapsto 1]$$
$$Read^\sharp[\![.]\!]M^\sharp = \{x, y, \langle\mu, 1\rangle\}, Write^\sharp[\![.]\!]M^\sharp = \{x, \langle\mu, 1\rangle\}, Alloc^\sharp[\![.]\!]M^\sharp = \{\langle\mu, 1\rangle\}$$

We have to consider $\langle\mu, 1\rangle$ as read because it is a weak memory location, thus its value is read during a weak update.

▶ **Definition 9** (Projection). *Given a set of memory locations $X^\sharp \in \mathcal{P}(\mathbb{A}_k^\sharp)$ and an abstract memory $M^\sharp \in \mathbb{M}^\sharp$, the projection of $M^\sharp$ on $X^\sharp$ is defined as:*

$$M^\sharp \downarrow X^\sharp \triangleq \lambda x. \textit{if } x \in X^\sharp \textit{ then } M^\sharp(x) \textit{ else } \top^\sharp$$

Memory locations are omitted from the abstract memory after a projection if their value is $\top^\sharp$ for the sake of readability.

5 Incremental Analysis with Function Summaries

This section introduces a summarization technique that allows abstract interpreters to avoid reanalyzing the body of a function when calling it within the same analysis context. We first introduce the program matching process that identifies the unmodified functions across two program versions. We then present and formalize the concept of function summaries and how they can be soundly reused to speed up incremental static analysis.

Given two program versions P_{new} and P_{old}, we aim to identify the functions that have not changed between the two versions. We consider two functions f_{new} and f_{old} to be similar if their ASTs have the same shape, they have the same instructions, the set of global variables they access are the same and have not been modified, and the functions they call are similar. Although simple, this matching process is sufficient for our purpose and allows us to identify a significant number of functions that have not changed between two versions of a program.

▶ **Example 10.** Consider the following program:

```
1 int a = 0;
2 void f() { a = 0; }
3 void g() { a = 1; }
4 void main() { f(); g(); }
```

Let us consider two different patches (a) and (b) that are applied independently to the program:

```
- int a = 0;
+ int a = 1;
```

(a)

```
- void f() { a = 0; }
+ void f() { a = 1; }
```

(b)

Applying patch (a) directly changes the value of the global variable a. Therefore, the functions `f`, `g` and their caller `main` are considered different from their previous versions. In contrast, applying patch (b) only changes the body of the function `f`, which impacts its caller `main` but not the function `g` that is independent of `f`. Therefore, the function `g` is considered similar to its previous version by the matching algorithm. Such a matching process is efficient (with regard to the number of functions that are considered similar) when the changes are localized, i.e., when they do not modify global variables accessed by many functions or when they do not change functions that are called by many other functions. More sophisticated techniques exist to identify similar functions or programs for more complex changes [13, 15], such as detecting variable renaming. However, these are beyond the scope of this work and are left for future research.

After identifying the functions that have not changed between two program versions, we aim to reuse the function summaries for these functions to speed up an incremental analysis. We now introduce the concept of function summaries and how they can be soundly reused to avoid reanalyzing the body of a function. In essence, a function summary is a pair of input/output abstract memory states $(\mathcal{I}^\sharp, \mathcal{O}^\sharp)$ that represent the analysis of a function's body. The input state $\mathcal{I}^\sharp$ is the abstract memory state before executing the body of the function and the output state $\mathcal{O}^\sharp$ is the abstract memory state after executing the body of the function. We consider $\mathcal{I}^\sharp$ as the analysis context of f that contains the abstract values of the function's arguments. By storing $(\mathcal{I}^\sharp, \mathcal{O}^\sharp)$ in a cache, we can reuse the output state $\mathcal{O}^\sharp$ when analyzing the same function body with an equivalent input state $\mathcal{I}^\sharp_{\text{new}}$, such as $\mathcal{I}^\sharp_{\text{new}} = \mathcal{I}^\sharp$. However, such a reusability condition is too restrictive and results in many cache misses because the input abstract memory state $\mathcal{I}^\sharp$ contains memory locations that are not necessarily needed to analyze function f. The following example illustrates this issue.

▶ **Example 11.** Consider the following program:

```
1 void f(int *p) { *p = 1; }
2 void main() {
3   int a = 0;
4   f(&a);
5 }
```

At line 4, the analysis infers $(\mathcal{I}^\sharp_4 = \{a \mapsto 0,\ p \mapsto \{\&a\}\}, \mathcal{O}^\sharp_4 = \{a \mapsto 1, p \mapsto \{\&a\}\})$ as the summary of `f`. A developer then applies the following patch that adds two more calls to `f`:

```
4   f(&a);
5 + f(&a);
6 + f(&a);
```

The incremental analysis of the program identifies that function `f` has not changed between both versions of the program, so it tries to reuse the summary of the first call to function `f` at line 4 when analyzing the second call to function `f` at line 5 and the third call to `f` at line 6. Indeed, the effect of the analysis of function `f` for every call is identical as the value of variable a does not impact the analysis of function `f`. However, the reusability condition $\mathcal{I}^\sharp_{\text{new}} = \mathcal{I}^\sharp$ does not hold at line 5 since $\{a \mapsto 1\} \neq \{a \mapsto 0\}$, the first mapping being part of the input state $\mathcal{I}^\sharp_5$ at line 5 and the second one being part of $\mathcal{I}^\sharp_4$. Consequently, the analyzer should analyze the body of `f` again and compute a new summary for the call at line 5, while it could be soundly reused in this case. Therefore, the above-mentioned reusability condition is too restrictive and leads to a lot of cache misses, although correct. Additionally, the abstract input and output states can be quite large for a real-world program, leading to significant

memory overhead. In this example, it is worth noting that, when analyzing the third call to function `f` at line 6, the reusability condition holds at line 5, so the summary can be reused for this particular call.

We propose a more relaxed condition for reusing function summaries. The new condition needs to verify only (a sound over-approximation of) the read memory locations $R^\sharp$ of the function in order to update the values of (a sound over-approximation of) the written memory locations $W^\sharp$. More precisely, given a statement stmt that does not allocate new memory locations, then its analysis summary is a tuple $(\mathcal{I}^\sharp \downarrow R^\sharp, R^\sharp, \mathcal{O}^\sharp \downarrow W^\sharp, W^\sharp)$: When analyzing stmt with a new input state $\mathcal{I}^\sharp_{\text{new}}$, we only check whether $\mathcal{I}^\sharp_{\text{new}}$ and $\mathcal{I}^\sharp$ are equal for the values restricted to $R^\sharp$. If so, we can reconstruct the output state $\mathcal{O}^\sharp_{\text{new}}$ by merging the values of written memory locations $W^\sharp$ of $\mathcal{O}^\sharp$ with the values of unmodified memory locations $\text{dom}(\mathcal{I}^\sharp_{\text{new}}) \setminus W^\sharp$ of $\mathcal{I}^\sharp_{\text{new}}$. The assumption that stmt does not allocate new memory locations is relaxed later in this section to handle dynamic memory allocation. More formally, if the condition $\mathcal{I}^\sharp_{\text{new}} \downarrow R^\sharp = \mathcal{I}^\sharp \downarrow R^\sharp$ holds before analyzing stmt, then its output state $\mathcal{O}^\sharp_{\text{new}}$ is $\mathcal{O}^\sharp \downarrow W^\sharp \sqcap^\sharp (\mathcal{I}^\sharp_{\text{new}} \downarrow (\text{dom}(\mathcal{I}^\sharp_{\text{new}}) \setminus W^\sharp))$.

▶ **Example 12.** Consider again Example 11. We can infer the following analysis summary of function `f` at line 4, along with its set of read and written memory locations.

$$(\mathcal{I}^\sharp_4 \downarrow R^\sharp_4 = \{p \mapsto \{\&a\}\},\ R^\sharp_4 = \{p\},\ \mathcal{O}^\sharp_4 \downarrow W^\sharp_4 = \{a \mapsto 1\},\ W^\sharp_4 = \{a\}).$$

It is worth noting that $\{a \mapsto 0\}$ is not part of the read input state anymore. When analyzing the call to function `f` at line 5 after having patched the code, we must only check whether $\{a \mapsto 1, p \mapsto \{\&a\}\} \downarrow \{p\}$ is equal to $\{p \mapsto \{\&a\}\}$. By definition of the projection, the condition indeed holds. Therefore, we can build the output state $\{a \mapsto 1\} \sqcap^\sharp (\{a \mapsto 0, p \mapsto \{\&a\}\} \downarrow \{p\})$, which is $\{a \mapsto 1, p \mapsto \{\&a\}\}$.

This relaxed condition allows us to reuse the analysis summary of the first call to function `f` at line 4 when analyzing the second call at line 5. It still allows for reusing it at line 6 as well. Additionally, the memory overhead of the incremental analysis is now reduced since the function summary only stores the projected values of the input and output states.

Extension to Dynamic Memory Allocation

In the case of programs that dynamically allocate memory on the heap, the condition of summary reuse on read memory locations is not sufficient to ensure soundness. Consequently, to preserve soundness, functions that allocate memory dynamically and every caller of these functions will not benefit from summaries.

▶ **Example 13.** Consider for instance the following program:

```
1 void f(int **p) { *p = malloc(sizeof(int)); }
2 void main() {
3     int *a;
4     f(&a);
5     f(&a);
6 }
```

Let μ be the allocation site at the call of malloc_μ in function `f`. We infer the following summary for the analysis of function `f` at line 4:

$$(\mathcal{I}^\sharp_4 \downarrow R^\sharp_4 = \{p \mapsto \{\&a\}\},\ R^\sharp_4 = \{p\},\ \mathcal{O}^\sharp_4 \downarrow W^\sharp_4 = \{a \mapsto \langle \mu, 0 \rangle\},\ W^\sharp_4 = \{a\})$$

When analyzing the call to function `f` at line 5, we check the condition on read memory locations and find that it holds. Therefore, the reconstructed output state is $\{a \mapsto \langle \mu, 0\rangle, \langle \mu, 0\rangle \mapsto Z_0^\sharp\}$. This output state is only correct if we assume that the number of allocations allowed on the site μ is 1, i.e., assuming the call to `f` at line 5 would not generate a new memory location. However, if we assume that $k \geq 2$, the reconstructed output for the call to function `f` at line 5 is incorrect, since the analysis of this call should have generated a new memory location $\langle \mu, 1\rangle$ and updated the value of a to the correct address $\langle \mu, 1\rangle$.

The issue with dynamic memory allocation comes from the condition on read memory locations: It does not capture the allocation of new memory locations. To address this issue, we need to verify that, given an abstract input state $\mathcal{I}^\sharp_{\text{new}}$, we would allocate or reuse the same abstract memory locations (def. 7) as with the saved input state $\mathcal{I}^\sharp$. More formally, we want to verify that the value of the next$^\sharp$ function is the same when applied to $\mathcal{I}^\sharp_{\text{new}}$ and $\mathcal{I}^\sharp$ for each allocation site μ of a given statement. Such a verification is only necessary when a statement stmt dynamically allocates memory, i.e., when $A^\sharp \neq \emptyset$.

▶ **Example 14.** Consider Example 13 again. We can infer the following summary for the analysis of function `f` at line 4, along with the set of read, written and allocated memory locations.

$$(\mathcal{I}_4^\sharp \downarrow R_4^\sharp = \{p \mapsto \{\&a\}\},\ R_4^\sharp = \{p\},\ \mathcal{O}_4^\sharp \downarrow W_4^\sharp = \{a \mapsto \langle \mu, 0\rangle\},\ W_4^\sharp = \{a\},$$
$$\mathcal{O}_4^\sharp \downarrow A_4^\sharp = \{\langle \mu, 0\rangle \mapsto Z_0^\sharp\},\ A_4^\sharp = \{\langle \mu, 0\rangle\})$$

When analyzing the call to function `f` at line 5, we also need to verify whether the input state allocates the same memory locations as the saved input state, i.e., $\forall \langle \mu, _\rangle \in A_4^\sharp,\ \text{next}^\sharp(\mu, \mathcal{I}_4^\sharp) = \text{next}^\sharp(\mu, \mathcal{I}_5^\sharp)$. The above condition holds only if the number of allocations allowed on the site μ is 1 or no allocation has been made on the site μ with the saved input state $\mathcal{I}_4^\sharp$.

The following theorem expresses the soundness property related to function summaries.

▶ **Theorem 15** (Safe approximation of the computation of a statement)**.** *Given a statement* ***stmt****, a previously computed abstract state* $\mathbb{S}^\sharp[\![\textbf{\textit{stmt}}]\!]\mathcal{I}^\sharp_{old} = \mathcal{O}^\sharp_{old}$*, reusing the analysis summary of* ***stmt*** *with a new abstract input state* $\mathcal{I}^\sharp_{new}$ *is a safe approximation of the computation of* ***stmt*** *if* $\mathcal{I}^\sharp_{new}$ *and* $\mathcal{I}^\sharp_{old}$ *agree on the values of the read memory locations and the set of memory locations allocated by* ***stmt*** *with* $\mathcal{I}^\sharp_{old}$ *is the same as the set of memory locations allocated by* ***stmt*** *with* $\mathcal{I}^\sharp_{new}$*. More formally, let* $\mathcal{I}^\sharp_{old}, \mathcal{I}^\sharp_{new} \in \mathbb{M}^\sharp$ *be two abstract input states, and let* $R^\sharp := Read^\sharp[\![\textbf{\textit{stmt}}]\!]\mathcal{I}^\sharp_{old}, W^\sharp := Write^\sharp[\![\textbf{\textit{stmt}}]\!]\mathcal{I}^\sharp_{old}$, $A^\sharp := Alloc^\sharp[\![\textbf{\textit{stmt}}]\!]\mathcal{I}^\sharp_{old}$, *and* $\mathcal{O}^\sharp_{old} := \mathbb{S}^\sharp[\![\textbf{\textit{stmt}}]\!]\mathcal{I}^\sharp_{old}$*:*

$$\begin{aligned} &\textit{if } \mathcal{I}^\sharp_{new} \downarrow R^\sharp = \mathcal{I}^\sharp_{old} \downarrow R^\sharp \textit{ and } \forall \langle \mu, _\rangle \in A^\sharp, next^\sharp(\mu, \mathcal{I}^\sharp_{new}) = next^\sharp(\mu, \mathcal{I}^\sharp_{old}) \\ &\textit{then } \mathbb{S}[\![\textbf{\textit{stmt}}]\!]\ \gamma(\mathcal{I}^\sharp_{new}) \\ &\quad \subseteq \gamma((\mathcal{O}^\sharp_{old} \downarrow (W^\sharp \cup A^\sharp)) \sqcap^\sharp (\mathcal{I}^\sharp_{new} \downarrow (dom(\mathcal{I}^\sharp_{new}) \setminus (W^\sharp \cup A^\sharp)))). \end{aligned}$$

Proof. The interesting part of the proof consists of showing that only the values of written and allocated memory locations are modified by the computation of a statement. This means that the values of memory locations that are neither written nor allocated are left unmodified. Additionally, the values of written and allocated memory locations must over-approximate the values of the corresponding memory locations in the concrete computation. The complete proof is available in the extended version. ◀

6 Incremental Analysis with Loop Invariant Reuse

The previous section has shown that, whenever one of the conditions about statements and input states is not satisfied, we need to recompute the function summaries for a given function. This section now shows how we can reuse loop invariants to accelerate the analysis of loops, even if no function summary is available or is reusable. Such an approach was already used in [23] to accelerate the analysis of loops when analyzing nested loops, but without formal details, nor proofs, nor detailed experimental evaluation. This approach works for incremental and non-incremental analyses. We first introduce the program matching process that identifies the loop invariant for reuse. We then show how such an invariant accelerates the analysis of loops and how we can limit the loss of precision induced by such reuse. Finally, we show the soundness and termination of the approach.

Given two program versions P_{new} and P_{old}, we aim to identify similar loop statements in both versions. To this end, we syntactically sort its loops in every function, and we match the loop of index n in function `f` in P_{new} with the loop of same index n in the function with the same name `f` in P_{old}. This matching might be incorrect, typically when one loop statement has been removed and another one has been added in the same function. However, it suffices to reuse some loop invariants from the previous analysis. Furthermore, an incorrect matching does not impact soundness, as demonstrated later in this section.

▶ **Example 16.** Consider the following program and its associated patch:

```
1  void loop() {
2    int i = 0;
3  - while (i < 10)
4  + while (i < 11)
5      i++;
6  }
7  void main() { loop(); }
```

In this example, the loop statement in function `loop` has been modified by changing its upper bound from 10 to 11. Although the loop's condition on both versions of the program are syntaxically different, we consider them as similar since they appear in the same function in the same order. We now know that such a modification invalidates the summaries computed for functions `loop` and `main`. However, we can still reuse the invariant computed for the loop statement in the original analysis of function `loop` in order to accelerate the analysis of the patched version. We now show how we can reuse its loop invariant. Assume the use of the interval domain, as defined in Fig. 1 for the abstract domain of the integer values, and the use of the standard widening operator over intervals at the loop head in order to accelerate convergence when computing the fixpoint. The analysis of the program before the modification of the loop statement is as follows:

$$\text{let } M^\sharp = \{i \mapsto [0,0]\} \text{ in } \mathbb{S}^\sharp[\![\textbf{while } (i<10) \textbf{ do } i := i+1;]\!]M^\sharp \triangleq \bowtie^\sharp_{i<10}(\text{lim } F^\sharp)$$
$$\text{where } F^\sharp(\mathcal{I}^\sharp) \triangleq \mathcal{I}^\sharp \nabla (M^\sharp \sqcup^\sharp \mathbb{S}^\sharp[\![i := i+1;]\!] \circ \bowtie^\sharp_{i<10} \mathcal{I}^\sharp)$$

We start the iteration with the initial abstract state $\bot^\sharp$, and then iterate the abstract semantics of the loop until reaching a post fixpoint. Here are the obtained abstract states:

$$F^{\sharp 0} \triangleq \bot^\sharp \nabla (\{[0,0]\} \sqcup^\sharp \bot^\sharp) = \{[0,0]\}$$
$$F^{\sharp 1} \triangleq \{[0,0]\} \nabla (\{[0,0]\} \sqcup^\sharp \{[1,1]\}) = \{[0,+\infty]\}$$
$$F^{\sharp 2} \triangleq \{[0,+\infty]\} \nabla (\{[0,0]\} \sqcup^\sharp \{[1,10]\}) = \{[0,+\infty]\}$$

The iteration sequence converges to the invariant $[0, +\infty]$ in three iterations. Now, we apply the patch to the program and analyze the modified version. Our strategy for loop invariant reuse consists in starting the iteration sequence with the invariant previously computed (which we will refer to as $\mathcal{I}^\sharp_{\text{prev}}$) instead of starting it from the initial abstract state $\bot^\sharp$. With such a strategy, the iteration sequence of the analysis of the loop for the modified program is as follows: $F^{\sharp 0} \triangleq (\mathcal{I}^\sharp_{\text{prev}} = \{[0, +\infty]\}) \nabla (\{[0, 0]\} \sqcup^\sharp \{[1, 11]\}) = \{[0, +\infty]\}$. The iteration sequence converges to the invariant $[0, +\infty]$ after a single iteration: We successfully reduced the number of iterations needed to converge to the invariant by reusing $\mathcal{I}^\sharp_{\text{prev}}$. Such a reduction might seem not important at a first glance since it "only" removes two iterations, but their accumulation becomes significant when analyzing large programs as shown in Section 7, since such fixpoint computations are frequent and time-consuming.

We now show the impact of reusing loop invariants on the precision of the analysis. We know that the invariant computed for a loop statement in the previous analysis is a sound over-approximation of the concrete least fixpoint. Consequently, it contains values that are not necessarily used by the loop statement. However, such values might have changed in the new version of the program, and reusing the invariant might propagate imprecise values. To limit such an imprecision, we only reuse the values of memory locations that are used inside the loop. Such memory locations are defined as follows:

▶ **Definition 17** (Set of abstract used memory locations).

$$Used^\sharp[\![\textbf{\textit{stmt}}]\!]M^\sharp \triangleq Read^\sharp[\![\textbf{\textit{stmt}}]\!]M^\sharp \cup Write^\sharp[\![\textbf{\textit{stmt}}]\!]M^\sharp \cup Alloc^\sharp[\![\textbf{\textit{stmt}}]\!]M^\sharp.$$

Given a former invariant $\mathcal{I}^\sharp_{\text{prev}}$, an abstract input state $M^\sharp$ and a loop statement **stmt**, we reduce the invariant $\mathcal{I}^\sharp_{\text{prev}}$ to $\mathcal{I}^\sharp_{\text{red}}$ in the following way:

$$\text{let } U^\sharp = Used^\sharp[\![\textbf{stmt}]\!]M^\sharp \text{ in } \mathcal{I}^\sharp_{\text{red}} \triangleq (\mathcal{I}^\sharp_{\text{prev}} \downarrow U^\sharp) \sqcap^\sharp (M^\sharp \downarrow (\text{dom}(M^\sharp) \setminus U^\sharp)).$$

Reducing $\mathcal{I}^\sharp_{\text{prev}}$ to $\mathcal{I}^\sharp_{\text{red}}$ lets us also reduce the memory space used when storing invariants in the cache. This reduction is important when analyzing large programs.

▶ **Example 18.** Suppose we are analyzing the loop statement **while** $(i < 10)$ **do** $i := i + 1$ with the invariant $\mathcal{I}^\sharp_{\text{prev}} = \{i \mapsto [0, +\infty], j \mapsto [1, 1]\}$. Suppose the initial abstract state before the loop is $M^\sharp = \{i \mapsto [0, 0], j \mapsto [0, 0]\}$, where the value of j differs from $\mathcal{I}^\sharp_{\text{prev}}$. We would like to reuse $\mathcal{I}^\sharp_{\text{prev}}$ to accelerate the analysis of the loop, but we would also like to avoid propagating the value of j, as its value would make the analysis less precise.

Initially, the iteration sequence of the analysis of the loop converges to the invariant $\{i \mapsto [0, +\infty], j \mapsto [-\infty, 1]\}$, since the widening operator would replace the lower bound of j with $-\infty$ because j is unstable and decreasing. We show the part of the iteration sequence that propagates the value of j:

$$F^{\sharp 1}(\{j \mapsto [1, 1]\}) \triangleq \{j \mapsto [1, 1]\} \nabla (\{j \mapsto [0, 0]\} \sqcup^\sharp \{j \mapsto [1, 1]\}) = \{j \mapsto [-\infty, 1]\}.$$

However, by forgetting the value of j from $\mathcal{I}^\sharp_{\text{prev}}$, the iteration sequence of the analysis of the loop converges to the invariant $\{i \mapsto [0, +\infty], j \mapsto [0, 0]\}$. The idea is to reuse only the values of memory locations used by the statement. The reduced invariant is as follows:

$$\begin{aligned}\mathcal{I}^\sharp_{\text{red}} =& (\{i \mapsto [0, +\infty], j \mapsto [1, 1]\} \downarrow \{i\}) \sqcap^\sharp (\{i \mapsto [0, 0], j \mapsto [0, 0]\} \downarrow \{j\}) \\ =& \{i \mapsto [0, +\infty], j \mapsto [0, 0]\}\end{aligned}$$

Nevertheless, some imprecision cannot be avoided when reusing loop invariants. Consider the following example:

▶ **Example 19.** Suppose we analyzed the loop statement **while** $(i < 10)$ **do** $i := i + 1$ where the initial abstract state is $M_1^\sharp = \{i \mapsto [0, 0]\}$ and we obtained the invariant $\mathcal{I}^\sharp_{\text{prev}} = \{i \mapsto [0, +\infty]\}$. We now analyze the same loop with $\mathcal{I}^\sharp_{\text{prev}}$ but with a different initial abstract state $M_2^\sharp = \{i \mapsto [1, 1]\}$, where the value of i differs from the two abstract states. The analysis of the loop converges to the invariant $\{i \mapsto [0, +\infty]\}$ when reusing $\mathcal{I}^\sharp_{\text{prev}}$, which is less precise than the invariant $\{i \mapsto [1, +\infty]\}$ that we would have obtained if we had started the iteration sequence from $\bot^\sharp$. Such an imprecision is inevitable when starting the iteration sequence from a correct but less precise invariant, i.e., $\{i \mapsto [1, +\infty]\} \sqsubseteq^\sharp \mathcal{I}^\sharp_{\text{prev}}$.

Finally, we show the soundness and termination of the approach.

▶ **Theorem 20** (Soundness and Termination)**.** *For any loop statement* ***while*** $(e \bowtie 0)$ ***do*** s*, abstract state* $M^\sharp$ *and invariant* $\mathcal{I}^\sharp$*, the limit of the iteration sequence of the analysis of* ***while*** $(e \bowtie 0)$ ***do*** s *starting from* $\mathcal{I}^\sharp$ *and such that* $\bowtie^\sharp_e(lim\ F^\sharp)$ *where* $F^{\sharp 0} = \mathcal{I}^\sharp \nabla (M^\sharp \sqcup^\sharp \mathbb{S}^\sharp[\![s]\!] \circ \bowtie^\sharp_e \mathcal{I}^\sharp)$ *is sound and always terminates.*

Proof. The iteration sequence starting from an abstract state $\bot^\sharp$ is known to converge to a sound approximation of the concrete least fixpoint. However, we can start the iteration sequence from an arbitrary abstract state $\mathcal{I}^\sharp$: As long as the stability test $F^\sharp(\mathcal{I}^\sharp) \sqsubseteq^\sharp \mathcal{I}^\sharp$ which verifies if the invariant is a post-fixpoint is correct, the invariant $\mathcal{I}^\sharp$ is a sound approximation of the concrete least fixpoint, regardless of the steps of the iteration sequence. ◀

In summary, this approach is complementary to function summaries, as it does not require any condition to be satisfied to reuse the previous invariant. We have shown that reusing the previous invariant makes the analysis faster by reducing the number of iterations before reaching a stable iterate but may lose precision for some cases. Section 7 will show that, in practice, reusing loop invariants significantly accelerates the analysis of loops and does not impact the precision of the analysis.

7 Implementation and Evaluation

This section presents our implementation of the proposed techniques for function summaries and loop invariants, built on top of the Eva [4, 6] plug-in of Frama-C [3, 19]. It is evaluated on a benchmark of open-source C programs. The incremental version of Frama-C and the set of programs used for the evaluation are available as supplementary material.

7.1 Implementation

The Eva plug-in already computes function summaries without support for dynamic memory allocation through the Memexec [37] cache. Moreover, Frama-C already provides a mechanism to store analysis results in a dedicated file [5]. Last, Frama-C also provides an abstract syntax tree (AST) diff tool to compare the ASTs of two relatively similar programs. In our work, we first extended the Memexec cache to support dynamic memory allocation and to compute and store loop invariants. Then, the saved function summaries and loop invariants are reloaded in subsequent analysis runs after comparing the abstract syntax trees of the analyzed programs. Finally, function summaries impacted by the changes are invalidated before running the analysis.

7.1.1 Function Summaries

Eva's Memexec computes and stores function summaries after analyzing a function with a new analysis context. Additionally, Memexec does not bound the number of summaries stored in the cache. We extended Memexec to support dynamic memory allocation by storing the abstract set of memory locations allocated by the function in the summary. However, the reusability condition differs slightly from the one presented in Section 5. In fact, Eva features a mechanism to deallocate memory locations. To do so, Eva removes the memory location from the abstract state when it is freed. More precisely, suppose that we want to analyze a statement $*\&x = \text{malloc}_\mu$ with an abstract input state $M^\sharp$, and we are given a set $A^\sharp$ of memory locations already allocated on the site μ. If Eva finds a memory location $\langle \mu, n \rangle$ in $A^\sharp$ that has been freed, i.e., $\langle \mu, n \rangle \notin \text{dom}(M^\sharp)$, then Eva reuses the memory location $\langle \mu, n \rangle$ instead of generating a new one. This mechanism is particularly useful for avoiding generating new memory locations when the same memory location is allocated and deallocated repeatedly, typically in a loop. It enhances the precision of the analysis by delaying as much as possible the generation of weak memory locations. We leverage this mechanism to reuse function summaries by checking whether the set of allocated memory locations $A^\sharp$ generated by the analysis of the function is reusable or not given a new abstract input state $\mathcal{I}^\sharp_{\text{new}}$, i.e., $A^\sharp \sqcap \text{dom}(\mathcal{I}^\sharp_{\text{new}}) = \emptyset$. This condition is sound with respect to the current handling of dynamic memory allocation in Eva. However, the formal proof of the soundness of this condition is left for future work.

7.1.2 Loop invariants

Contrary to function summaries, Eva had no mechanism to reuse loop invariants. Therefore, we extended it to do so: First, we modified AST Diff to build the correspondence between the loops of the functions of both programs. Then, we added a cache of loop invariants for each loop statements of each function in Memexec. The reusability condition of loop invariants presented in Section 6 states that any loop invariant can be soundly reused. We are left with the task of finding the loop invariant to reuse when multiple invariants are stored in the cache. Finding such an invariant is challenging because, although every choice maintains soundness, it has an impact on the performance and the precision of the analysis. More precisely, we want to find the closest invariant to the one we are looking for. To do so, we decided to store separately new invariants when the analysis of the current function is done with a new analysis context. The analysis context is defined here as the callstack and the abstract values of the argument of the function. When, the same analysis context is encountered, the new invariant is merged using the abstract join $\sqcup^\sharp$ operator with the one stored in the cache.

7.2 Evaluation

Research questions

We evaluate our implementation to address the following research questions:

RQ1: Is incremental analysis more time-efficient than a full analysis?

RQ2: What is the impact of incremental analysis on max memory usage and on-disk storage?

RQ3: What is the impact of incremental analysis on the precision of the analysis?

RQ4: What is the effect of cache loading and AST comparison on analysis time?

Table 1 Program description.

Program	# Commit	LoC	Function	Loop	Diff Summary
PolarSSL	41	28K	818-822	293-302	52 files, 887 (+), 259 (−)
Monocypher	107	7K	660-823	210-244	147 files, 15348 (+), 3395 (−)
Chrony	119	25K	206-265	125-162	109 files, 10754 (+), 1698 (−)

Experimental Setup

Our experiments focus on analyzing three real-world programs: PolarSSL [1], Monocypher [2] and Chrony [3], issued from the Open Source Case Studies (OSCS) [4]. Table 1 shows the total number of commits analyzed, the average number of lines of code (LoC), the minimum and maximum number of functions and loops in the program, and the summary of the diff between the first and last commit. The diff summary shows the total number of files modified, the total number of lines added and removed. All programs, written in C, were already pre-configured and modified to be analyzable with Eva for a classical non-incremental analysis. For our evaluation, we simulate a development scenario where developers modify the source code of these programs and push the changes to the repository. The commits consist of real modifications made by the developers of both projects. Each code commit triggers an analysis run in the continuous integration (CI) pipeline. The analysis process reuses function summaries and loop invariants computed during the previous analysis run. Every commit includes modifications affecting the source code. The code source in OSCS contains annotations made by the maintainers of the repository. These annotations are mandatory to guide the analysis of Eva and ensure it completes in a reasonable time. We extracted these annotations as a patch and applied them to each commit before running the analysis. Since a commit involves changes to the source code, we need to ensure for each commit that the patch applies correctly, that Frama-C can parse the modified code, and that Eva's analysis completes successfully. We took the maximum set of commits that successfully passes these checks for each program. Adding the commits where any of these checks failed would involve changes to the annotations or even the analysis parameters for each additional commit. No additional code annotations were needed for the incremental analysis. Our experiments are executed on a machine with an Intel Core i7-12800H CPU clocked at 2.4 GHz and 64 GB of RAM. The machine runs Manjaro Linux 6.6. We used Frama-C version 28.1 extended with our development for incremental analysis. The analysis parameters were set to the values already used in OSCS for non-incremental analysis. We also set the maximum number of allocation per site k_{lim} to 2 with `-eva-mlevel 2` option and we disabled the possibility of `malloc` to fail with `-eva-no-alloc-returns-null` option.

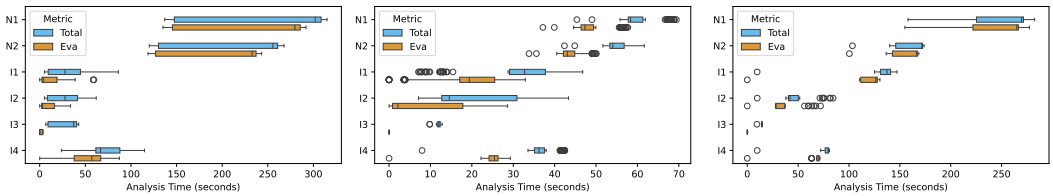
7.3 Results

To answer the research questions RQ1–RQ4, we evaluate the performance and precision of our approach by focusing on criteria such as analysis time, memory consumption, and number of alarms generated on six different analysis settings for each version of PolarSSL,

[1] `https://github.com/Mbed-TLS/mbedtls`
[2] `https://github.com/LoupVaillant/Monocypher`
[3] `https://github.com/mlichvar/chrony`
[4] `https://git.frama-c.com/pub/open-source-case-studies`

Table 2 Analysis settings.

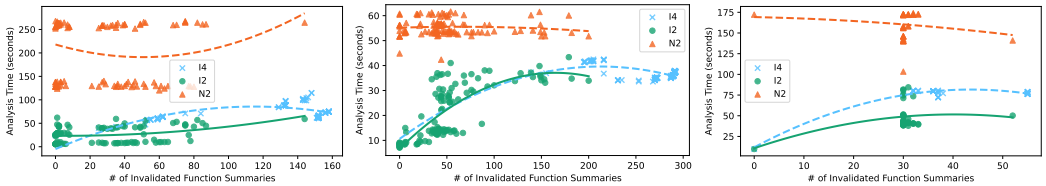
Setting	Type	Version Reused	Approach
N1	Full	None	Summaries
N2	Full	None	Summaries, Invariants
I1	Incremental	Immediate previous	Summaries
I2	Incremental	Immediate previous	Summaries, Invariants
I3	Incremental	Same version	Summaries, Invariants
I4	Incremental	First version	Summaries, Invariants



(a) Monocypher. **(b)** Chrony. **(c)** PolarSSL.

Figure 8 Analysis time measurement results.

Monocypher and Chrony as shown in Table 2. Full Analyses (N1 and N2) are used as a reference for the comparison with Incremental Analyses (I1, I2, I3, and I4). Even though N1 and N2 do not reuse any previous results, they still benefit from summaries and invariants computed during the ongoing analysis. Incremental Analyses (I1 and I2) reuse the results of the immediate previous version of the program when they are available. For the first version of the program, we run a full analysis similar to N1 (resp. N2) for I1 (resp. I2). The first analysis results are then reused by the second version, and so on. The difference between N1 and N2 (resp. I1 and I2) is the reuse of loop invariants. It allows us to evaluate the impact of reusing loop invariants on the analysis results. Incremental Analysis (I3) sets the limit of the maximum efficiency that can be achieved by our approach. We run first a full analysis of each version of the programs and then we incrementally analyze each version with itself. Therefore, no changes are made to the programs and all summaries and invariants are reused. Incremental Analysis (I4) sets the limit of the minimum efficiency that can be achieved by our approach. We run first a full analysis on the first version (v_0) of the programs and then we incrementally analyze each version ($v_1, ..., v_n$) with the results from the first version (v_0). The analysis of v_0 reusing the results of v_0 represents the best case scenario, whereas the analysis of v_n reusing the results of v_0 where n is the last version of the program represents the worst case scenario. Figure 8 shows the analysis time measurement for each program and analysis setting. We separate the total analysis time from Eva's analysis time. The latter is the abstract interpretation per se while the former includes the time spent on parsing the program, loading the cache and computing the impact of changes. Eva's analysis correctly represents the impact of the approach on the analysis time. The difference between the two metrics represents the overhead of incremental analysis and can be further reduced by optimizing cache loading and program matching. Circles represent outliers, i.e., data points that are significantly different from the rest of the data. Outliers such as sub-second Eva analysis time are not removed from the box plot as they are not considered as anomalies. We only remove the analysis of the first versions of each program in I1 and I2 because they cannot reuse any previous results and therefore they are not relevant. Figure 9 shows the



(a) Monocypher. **(b)** Chrony. **(c)** PolarSSL.

Figure 9 Impact of invalidated function summaries on analysis time.

Table 3 Summary of analysis results across programs and analysis settings.

	Analysis Setting	Total Alarms	Total Iter.	Max Mem. (MB)	Cache Load(s)	AST Diff.(s)	Cache Size(MB)
P	N1	29097	839265	1161			89.40
	I1	∼	**1.17x**	0.42x	7	4	0.63x
	N2	∼	1.62x	1.01x			1.01x
	I2	0.99x	4.00x	0.50x	5	4	0.68x
M	N1	25356	14478001	2981			130.42
	I1	0.99x	24.46x	0.64x	7	2	0.65x
	N2	**1.01x**	1.34x	0.94x			0.98x
	I2	**1.01x**	35.22x	0.64x	7	2	0.63x
C	N1	404748	1402787	1249			100.75
	I1	0.99x	1.81x	0.76x	2	4	0.85x
	N2	0.99x	1.6x	0.97x			0.99x
	I2	0.99x	6.9x	0.77x	2	4	0.90x

relationship between the number of invalidated function summaries and the analysis time. It also indicates the impact of the changes made to the program by the amount of invalidated summaries. Table 3 presents a comprehensive comparison between a full analysis and an incremental analysis across several performance metrics. We use N1 as the reference for the comparison with the other analysis settings (I1, N2 and I2). We display the ratio (Full Analysis / Incremental Analysis) for most cases except cache loading and AST Diff time. Cases with a significant difference are highlighted in bold. Cases annotated with ∼ indicate that the value is the same as the reference. Cases without values indicate a non-applicable metric. The first column indicates the program analyzed with **P** for PolarSSL, **C** for Chrony, and **M** for Monocypher. The second column indicates the analysis setting. The third column shows the total number of alarms generated by the analysis of all versions. The fourth column shows the total number of iterations required before convergence of all versions. The fifth column shows the peak memory usage. The sixth and seventh columns show the average time spent on loading the cache and computing the impact of changes, respectively. The last column shows the maximum size of the cache file.

RQ1: Time Efficiency

Figure 8 shows that Incremental Analysis I1 (resp. I2) is faster than Full Analysis N1 (resp. N2) for all programs. The efficiency of reusing function summaries alone is shown by N1 and I1. The further reduction in analysis time when reusing loop invariants is shown by

the differences N2-N1 and I2-I1. Moreover, Table 3 shows a significant reduction of the total iteration required to converge to a fixpoint when reusing loop invariants. Additionally, we observe some cases with Chrony where I1 is slower than N1. This is due to the invalidation of a large number of summaries. Consequently, the analysis does not benefit from previous results and the total analysis time increases due to the overhead of loading the cache and computing the impact of changes. Moreover, we observe that Incremental Analysis I1 is slower than Full Analysis N2 for PolarSSL. The number of iterations required to converge shown in Table 3 is higher for I1 than N2. This indicates that reusing function summaries is not always sufficient and the combination of summaries and loop invariants is necessary to achieve the best performance. We observe in Figure 8 for I3 that Eva's analysis time is instantaneous as no summaries are invalidated. Consequently, the total analysis time is only spent on loading the cache and computing the impact of changes. We observe in Figure 9 for I4 that the analysis time increases as the number of invalidated summaries increases. However, I4 is still more time-efficient than N2 in most cases. Nevertheless, the number of invalidated summaries is still not the best indicator – although better than the raw commit diff – to predict the outcome the analysis time. Indeed, the analysis time of each function can be different depending on its complexity. For example, the analysis of a function with a large number of loops or having deeply nested function calls is more time-consuming than the analysis of a logging function. The significant reduction in analysis time shows the efficiency of our incremental approach, making it highly suitable for use in CI/CD environments where quick feedback on code changes is essential.

RQ2: Max Memory Usage and On-Disk Storage

Table 3 shows that Incremental Analysis uses more memory than Full Analysis for all programs. This increase in memory usage is expected as we trade memory for time efficiency. Additionally, adding summaries and invariants to the cache increases the size of the dedicated file on disk. However, we believe that the increase in memory consumption and on-disk storage of Incremental Analysis is still acceptable and worth the time efficiency gained as they remain stable during the analysis. Moreover, the number of summaries and invariants we store is currently unbounded, thus, it is possible to modify this strategy and dynamically adjust this number in case of memory issues.

RQ3: Precision

We compare the total number of alarms generated by each analysis setting to N1 to assess the precision of our approach. We observe three different cases in Table 3: First, all incremental analysis settings I2 for all programs generated more alarms than N1. This behavior is expected as the reuse of loop invariants might introduce imprecision in the analysis. Although, we observe only a very slight increase in the number of alarms. Second, the number of alarms generated by I1 for Chrony and Monocypher also increases compared to N1. This increase is directly related to the reuse of function summaries involving dynamic memory allocation (summaries of function without dynamic memory allocation are not impacted). More precisely, we found that reusing summaries that contain weak memory locations introduces imprecision in the analysis in some cases because a full Eva analysis could have reused a strong memory location instead. This behavior is related to the definition of the next function which does not take into account memory deallocation, therefore, when the maximum number of allocation per site k_{lim} is reached, the function next always points to the last weak memory location. Third, the number of alarms generated by N2 and I2 for Monocypher decreases compared

to N1. Although unexpected, this behavior is related to the non-monotonicity property of widenings. More precisely, the reuse of loop invariants during the analysis of loops leads to the generation of more precise values. The non-monotonicity property of widening is studied in [9], and we have therefore an experimental evidence of its impact on real-world programs instead of crafted examples. Overall, this indicates that the incremental approach maintains a high level of precision, ensuring that the incremental approach is about as good as the non-incremental approach.

RQ4: Cache Loading and Program Matching

Table 3 shows that the overhead of loading the cache and computing the impact of changes is acceptable. The actual cases where such overhead is significant are when Eva's analysis time is very fast. Nonetheless, there is still room for improvement such as a partial or deferred cache loading and the use of more efficient program matching algorithms.

Summary

Overall, our approach offers substantial benefits in terms of time efficiency, with an acceptable memory usage and on-disk storage cost, and a slight overhead in loading the cache and computing the impact of changes. The analysis remains precise enough and even more precise in some cases. The reuse of loop invariants further enhances performance. These results suggest that incremental analysis is a valuable technique for improving the efficiency of static analysis in CI/CD environments.

Threats to Validity

We identify the following threats to the validity of our results: We reuse mostly the same analysis parameters as the ones already configured for a non-incremental analysis. Therefore, the effects of changing these parameters have not been explored: Our results could vary significantly when modifying them. Even though we conduct our experiments on three real-world programs, the generalizability of our results to other programs remains uncertain. The specific characteristics of these programs could influence the results, making their applicability to other programs unknown.

8 Related work

Incremental static analysis is a technique that has gained significant attention in recent years due to the need for fast and efficient analyses of large growing codebases. This technique speeds up non-incremental analyses by reusing previous analysis results and avoiding redundant work. This section reviews the current state of the art in incremental static analysis by abstract interpretation and highlights the gap that our work fills.

Incremental Modular Analysis. Modular analyses [12], based on summaries [7, 36, 35, 22, 27, 8] enable incremental analysis by analyzing only the modules that are affected by the changes in the program. Work in [35] proposes an approach to make modular static analysis incremental by exploiting dependencies between modules. They show that the dependencies computed during the analysis can be used to bound the impact of a change. Compared to this work our approach is more generic and does not require the original analysis to be modular. Work in [16] proposes a modular incremental analyzer capable of performing fine-grain incremental analysis across modular program partitions. However, their approach

assumes that the program is represented as Constrained Horn clauses and the analysis is done on the Horn clause representation. Compared to this work, our approach is more generic and does not require an intermediate representation far from the original program. Commercial tools like Coverity [22] and Infer [27] support modular incremental analysis on large codebases. However, they do not provide a soundness guarantee, which makes them unsuitable for critical software. In particular, Infer only shows the new alarms that are generated by the changes in the code after the analysis. More precisely, Infer only analyses the functions that are impacted by the changes in the code, and it does not guarantee that the analysis is sound. A shared problem with modularity through summaries is the loss of precision induced by the formalism used. A safe modular analysis implies a summary capable of representing all possible call contexts. However, computing a precise summary is costly, which leads to a trade-off between precision and cost. Our approach is inspired by summaries and guarantees safety while maintaining a high level of precision in the analysis.

Incremental Analysis Framework. Recent work has incrementalized several classes of program analyses. For instance, [1] focuses on dataflow analyses but is limited to distributive lattices, which rules out interval abstract domains. Frameworks such as IncA [33, 31, 32] provide a DSL for defining incremental analyses and algorithms to maintain analysis results. IncA translates the user-defined analysis to an intermediate representation such as graph-patterns and Datalog. It supports incremental control flow and points-to analysis for mbedrr C [33], and strong-update points-to, string, and constant propagation analyses for Java [31, 32]. Code modifications rely on a given AST changes notification, e.g., from IDEs. The incremental strategy maintains results across changes and provides real-time feedback. However, correctness and efficiency depend on the limitations and assumptions of the underlying Datalog solver, which analysis developers must satisfy. For instance, [33] do not support recursive user-defined aggregation, which limits analysis results to powerset lattices only. Later work [31] removes such limitations by providing a new algorithm and the use of widenings for infinite-height lattice, but imposes monotonicity over the aggregation operator and are only efficient for intra-procedural analyses. Work in [32] loosen the monotonicity requirement, although still imposes it in some cases and is efficient for inter-procedural analyses. In contrast, our approach does not target user-defined program analyses, but rather incrementalizes the abstract interpretation framework itself. We do not depend on the underlying abstract domains as long as they are non-relational. Abstract domains only need to be a poset, not a CPO nor a lattice, and abstract operators need not be monotonic – which they are often not, due to widenings and reductions. Moreover, our approach does not incrementally maintain analysis results given a set of changes, but rather runs the analysis of a program with a completely new AST and only reuses previous results when possible. This approach is more suitable for existing whole program analysis frameworks, such as Frama-C [3], Mopsa [24] and Astrée [23], where incrementality has not been considered. Other recent work focuses on incrementalizing existing frameworks. Works in [28, 29] incrementalize a generic local fixpoint solver: They show that the data structures used in the solver can be exploited to limit the part of the program that needs to be reanalyzed. They also support on-demand analyses [14] for which, given a specific query to the analyzer, it only computes the necessary dependencies to answer the query. Work in [30] proposes a new operational semantics that reifies abstract interpretation on a directed acyclic graph. It shows that the dependencies between the nodes in the graph can be exploited to invalidate only the nodes that are affected by the change in the program. Compared to these works, we provide a generic framework that can be built on top of any existing abstract interpreter instead

of incrementalizing a particular framework. Reusing fixpoints for incremental analysis is also a common technique used in incremental analysis. For instance, [26] matches pre- and post-change programs and reuses fixpoints for matched parts. It ensures soundness by reanalyzing the entire program, making it conservative–imprecision from mismatches may propagate. Our method limits fixpoint reuse to loops in modified functions or when function summaries cannot be reused, minimizing imprecision propagation.

Cache Reuse. Caching is pervasive and mandatory in practice for any large scale analysis. Abstract interpreters such as Mopsa [24, 25] and Astrée [23] use caching for function and fixpoint iteration in order to accelerate interprocedural and multi-threaded analyses. The Eva [4] plugin of Frama-C [3] also uses caching for function summaries, which did not support dynamic memory allocation before our extension. However, the reuse of the cache is limited to the ongoing analysis and are not designed to be soundly reused across different analyses. Compared to these works, we provide the conditions under which the cache can be reused or invalidated across different analysis.

9 Conclusion and Future Work

This paper presented a sound, generic approach to incremental static analysis based on reusing results from previous analysis runs. We introduced two complementary approaches: one for computing and reusing function summaries–precise and sound when functions are not affected by changes–and another for reusing loop invariants, which remain sound even when loops are modified, though potentially less precise. Combined, these two approaches can be used to significantly accelerate the analysis of large software. They are beneficial for both ongoing analysis and incremental analysis. They are suitable for critical software and for CI/CD approaches. We have implemented our approach in the Eva plugin of Frama-C and evaluated it on a set of open source programs. Our evaluation shows that our approach can significantly reduce the analysis time of these programs, while maintaining a high level of precision. Our next steps include extending function summaries to handle relational domains. As seen in the semantics of the language, function summaries only compute the set of memory locations inductively from the syntax. However, such a set cannot include by definition the set of memory locations that are related to each other. We already implemented a solution to this problem within Eva. More precisely, we compute the transitive closure of the relation between the read memory locations given by each relational domain used in the analysis. The next step is to formalize this solution and to prove its correctness. Additionally, we plan to extend our formalization to handle more features of the C language, such as pointer arithmetic, recursive function calls and memory deallocations. Moreover, we plan to use more sophisticated techniques for the program matching techniques, allowing us to reuse more analysis results. Finally, we plan to take into account changes in the parameters of the analysis. That is, static analyzers are often used after the initial development phase, typically by a separate team whose goal is to certify the absence of runtime errors. To do so, the analysis parameters are finely tuned incrementally to improve the precision. Since the cost of this process is an order of magnitude higher than the cost of the initial analysis, we aim to adapt our approach to this methodology of incremental tuning of the analysis parameters. However, not only is it crucial to avoid any loss of precision resulting from incrementality, as it currently the case, but we also aim to improve the precision of the analysis.

References

1 Steven Arzt and Eric Bodden. Reviser: Efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *Proceedings of the 36th International Conference on Software Engineering*, pages 288–298, Hyderabad India, May 2014. ACM. doi:10.1145/2568225.2568243.

2 Gogul Balakrishnan and Thomas Reps. Recency-abstraction for heap-allocated storage. In Kwangkeun Yi, editor, *Static Analysis*, pages 221–239, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. doi:10.1007/11823230_15.

3 Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free C programs: the frama-c software analysis platform. *Commun. ACM*, 64(8):56–68, 2021. doi:10.1145/3470569.

4 Sandrine Blazy, David Bühler, and Boris Yakobowski. Structuring Abstract Interpreters Through State and Value Abstractions. In Ahmed Bouajjani and David Monniaux, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 10145, pages 112–130. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-52234-0_7.

5 François Bobot, André Maroneze, Virgile Prevosto, and Julien Signoles. *The Art of Developing Frama-C Plug-ins*, pages 341–401. Springer International Publishing, July 2024. doi:10.1007/978-3-031-55608-1_7.

6 David Bühler, André Maroneze, and Valentin Perrelle. *Abstract Interpretation with the Eva Plug-in*, pages 131–186. Springer International Publishing, July 2024. doi:10.1007/978-3-031-55608-1_3.

7 Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, volume 9058 of *Lecture Notes in Computer Science*, pages 3–11. Springer, 2015. doi:10.1007/978-3-319-17524-9_1.

8 Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, 2011. doi:10.1145/2049697.2049700.

9 Patrick Cousot. *Principles of Abstract Interpretation*. MIT Press, 2022.

10 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages - POPL '77*, pages 238–252, Los Angeles, California, 1977. ACM Press. doi:10.1145/512950.512973.

11 Patrick Cousot and Radhia Cousot. Constructive Versions of Tarski's Fixed Point Theorems. *Pacific Journal of Mathematics*, 82(1), 1979.

12 Patrick Cousot and Radhia Cousot. Modular Static Program Analysis. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and R. Nigel Horspool, editors, *Compiler Construction*, volume 2304, pages 159–179. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. doi:10.1007/3-540-45937-5_13.

13 David Delmas and Antoine Miné. Analysis of Software Patches Using Numerical Abstract Interpretation. In Bor-Yuh Evan Chang, editor, *Static Analysis*, volume 11822, pages 225–246. Springer International Publishing, Cham, 2019. doi:10.1007/978-3-030-32304-2_12.

14 Julian Erhard, Simmo Saan, Sarah Tilscher, Michael Schwarz, Karoliine Holter, Vesal Vojdani, and Helmut Seidl. Interactive Abstract Interpretation: Reanalyzing Whole Programs for Cheap, November 2022. doi:10.48550/arXiv.2209.10445.

15 Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2642937.2642982.

16 Isabel Garcia-Contreras, Jose F. Morales, and Manuel V. Hermenegildo. Incremental and Modular Context-sensitive Analysis. *Theory and Practice of Logic Programming*, 21(2):196–243, March 2021. doi:10.1017/S1471068420000496.

17 Denis Gopan, Frank DiMaio, Nurit Dor, Thomas Reps, and Shmuel Sagiv. Numeric domains with summarized dimensions. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988, pages 512–529, March 2004. doi:10.1007/978-3-540-24730-2_38.

18 Vini Kanvar and Uday P. Khedker. Heap abstractions for static analysis. *ACM Comput. Surv.*, 49(2), June 2016. doi:10.1145/2931098.

19 Nikolaï Kosmatov, Virgile Prevosto, and Julien Signoles. *Guide to Software Verification with Frama-C: Core Components, Usages and Applications*. Springer Cham, July 2024. doi:10.1007/978-3-031-55608-1.

20 William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In Stuart I. Feldman and Richard L. Wexelblat, editors, *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), San Francisco, California, USA, June 17-19, 1992*, PLDI '92, pages 235–248, New York, NY, USA, 1992. ACM. doi:10.1145/143095.143137.

21 MathWorks. Polyspace. https://www.mathworks.com/products/polyspace.html. Accessed: 2024-12-20.

22 Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. Scalable and incremental software bug detection. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, Esec/Fse 2013, pages 554–564, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2491411.2501854.

23 Antoine Miné and David Delmas. Towards an industrial use of sound static analysis for the verification of concurrent embedded avionics software. In *2015 International Conference on Embedded Software (EMSOFT)*, pages 65–74, Amsterdam, Netherlands, October 2015. IEEE. doi:10.1109/EMSOFT.2015.7318261.

24 Antoine Miné, Abdelraouf Ouadjaout, and Matthieu Journault. Design of a Modular Platform for Static Analysis. In *The Ninth Workshop on Tools for Automatic Program Analysis (TAPAS'18)*, 2018.

25 Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Static Type Analysis by Abstract Interpretation of Python Programs. In *Leibniz International Proceedings in Informatics (LIPIcs)*, 2020. doi:10.4230/LIPIcs.ECOOP.2020.17.

26 Lawton Nichols, Mehmet Emre, and Ben Hardekopf. Fixpoint reuse for incremental JavaScript analysis. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 2–7, Phoenix AZ USA, June 2019. ACM. doi:10.1145/3315568.3329964.

27 Peter W. O'Hearn. Continuous Reasoning: Scaling the impact of formal methods. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 13–25, Oxford United Kingdom, July 2018. ACM. doi:10.1145/3209108.3209109.

28 Helmut Seidl, Julian Erhard, and Ralf Vogler. Incremental Abstract Interpretation. In Alessandra Di Pierro, Pasquale Malacaria, and Rajagopal Nagarajan, editors, *From Lambda Calculus to Cybersecurity Through Program Analysis*, volume 12065, pages 132–148. Springer International Publishing, Cham, 2020. doi:10.1007/978-3-030-41103-9_5.

29 Helmut Seidl and Ralf Vogler. Three improvements to the top-down solver. *Mathematical Structures in Computer Science*, 31(9):1090–1134, October 2021. doi:10.1017/S0960129521000499.

30 Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. Demanded abstract interpretation. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, PLDI 2021, pages 282–295, New York, NY, USA, 2021. ACM. doi:10.1145/3453483.3454044.

31 Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. Incrementalizing lattice-based program analyses in datalog. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276509.

32 Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. Incremental whole-program analysis in datalog with lattices. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 1–15, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454026.

33 Tamás Szabó, Sebastian Erdweg, and Markus Voelter. IncA: A DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 320–331, Singapore Singapore, August 2016. ACM. doi:10.1145/2970276.2970298.

34 Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, June 1955. doi:10.2140/pjm.1955.5.285.

35 Jens Van der Plas, Quentin Stievenart, Noah Van Es, and Coen De Roover. Incremental Flow Analysis through Computational Dependency Reification. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 25–36, Adelaide, Australia, September 2020. IEEE. doi:10.1109/SCAM51674.2020.00008.

36 Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: Static race detection on millions of lines of code. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 205–214, Dubrovnik Croatia, September 2007. ACM. doi:10.1145/1287624.1287654.

37 Boris Yakobowski. Fast whole-program verification using on-the-fly summarization. In *Workshop on Tools for Automatic Program Analysis (TAPAS)*, page 2, 2015.

Declarative Dynamic Object Reclassification

Riccardo Sieve
University of Oslo, Norway

Eduard Kamburjan
IT University of Copenhagen, Denmark

Ferruccio Damiani
University of Turin, Italy

Einar Broch Johnsen
University of Oslo, Norway

Abstract

In object-oriented languages, dynamic object reclassification is a technique to change the class binding of an object at runtime. Current approaches express when and how to reclassify inside the program's business code, while maintaining internal consistency. These approaches are less suited for programs that need to be consistent with an external context, such as autonomous systems interacting with a knowledge base. This paper proposes declarative dynamic object reclassification, a novel technique that provides a separation of concerns between a program's business code and its adaptation logic for reclassification, expressed via a knowledge base. We present Featherweight Semantically Reflected Java, a minimal calculus for declarative dynamic object reclassification that enables the programmer to define consistency both internally (using a type system) and externally (using declarative classification queries). We use this calculus to study how internal and external consistency interact for declarative dynamic object reclassification. We further implement the technique by extending SMOL, a language for reflective programming via external knowledge bases.

2012 ACM Subject Classification Software and its engineering → Abstraction, modeling and modularity; Software and its engineering → Object oriented languages

Keywords and phrases Dynamic Object Reclassification, Dynamic Software Updates, Featherweight Java, Knowledge Bases, Semantic Reflection, Type Soundness

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.29

Supplementary Material *Software (Artifact)*: `https://doi.org/10.5281/zenodo.15275245`
Software (ECOOP 2025 Artifact Evaluation approved artifact):
`https://doi.org/10.4230/DARTS.11.2.6`

1 Introduction

In traditional object-oriented programming [54, 56], the behaviour of an object instantiated from a class is defined by the methods and properties of that class. However, in many real-world applications, both the object's state and the context in which it operates can change over time. Dynamic object reclassification [15, 16] and typestate-oriented programming [2, 20] are techniques within class-based object-oriented programming that enable *dynamic* class binding of objects at runtime, allowing objects to change their class while retaining their identity. These powerful programming abstractions are particularly useful when an object's context changes over time, and the object needs to adapt to the new context. When the class-binding of objects is dynamic, affected objects need to remain *consistent* with their expected capabilities. For example, consistency becomes critical for dynamic software updates in asynchronous, IoT-driven ecosystems where the local processing of class re-binding may get delayed [34].

© Riccardo Sieve, Eduard Kamburjan, Ferruccio Damiani, and Einar Broch Johnsen;
licensed under Creative Commons License CC-BY 4.0

39th European Conference on Object-Oriented Programming (ECOOP 2025).
Editors: Jonathan Aldrich and Alexandra Silva; Article No. 29; pp. 29:1–29:31

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Previous approaches to such programming abstractions (e.g., [2, 8, 13–16, 20, 26]) handle the requirements for the reclassification process internally: programs describe classes and reclassification within the same language. This results in (1) a notion of consistency that does not capture the program's context, (2) an operational and low-level view of the conditions for reclassification, and (3) a lack of separation of concerns between describing object behaviour and object reclassification. In contrast, autonomous and self-adaptive systems in, e.g., robotics and digital twins, typically need to adapt to a context represented by external knowledge bases containing information about both the environment and the system itself [7, 42].

This paper proposes a novel approach to dynamic object reclassification, which shifts the reclassification process from ensuring internal consistency, with an operational view of the adaptation logic, to ensuring external consistency, with a declarative view of the adaptation logic. The approach combines three main elements:

Knowledge Base: The context of the program is modelled as an external knowledge base, i.e., a logical representation of facts. This knowledge base evolves independently of the program and can be queried for Boolean results (*"is a certain formula implied by the knowledge base?"*) or retrieval (*"which values satisfy a given formula?"*).

Semantic Reflection expresses that a program can query its own state in context of the knowledge base; this is enabled by a process of lifting the runtime state of a program into the knowledge base [38].

Declarative Object Reclassification is defined via reclassification queries to the knowledge base. The first query defines when an object is *consistent* with a particular class; i.e., it is a *membership* or *classification* query. The second query that defines how to instantiate an object's fields when reclassifying an object into a particular class; i.e., it is a state *retrieval* query. These queries make use of semantic reflection, as they uniformly access both the lifted program state and the context in the knowledge base.

In combination, restrictions on queries and on class inheritance ensure that reclassification preserves type safety. In short, the main contributions of this paper are as follows:

- We propose a declarative dynamic object reclassification technique that uses (1) reflection of the program state into a knowledge base, and (2) the additional context of an external knowledge base, to provide an interface for reclassification queries that encapsulates the adaptation logic from the programmer.
- We formalise our technique in Featherweight Semantically Reflected Java (FSRJ), a novel minimal core calculus in the spirit of Featherweight Java [31], which supports semantic reflection into knowledge bases.
- We prove type soundness for FSRJ and give a precise characterisation of *program coherence*, describing the conditions under which reclassification queries ensure type safety.
- We provide a prototype implementation of declarative dynamic object reclassification as an extension of SMOL [38], a language for self-adaptive digital twins [40] based on Knowledge Graphs [29], including a static check for program coherence.
- We evaluate our prototype implementation by application to GreenhouseDT, an open-source digital twin of a mini-greenhouse [41].

The remainder of this paper is structured as follows: Section 2 illustrates declarative dynamic object reclassification with an example before Section 3 describes Featherweight Semantic Reflected Java, the minimal calculus for reflection into a knowledge base and subsequent object reclassification, as well as the type system. Section 4 describes the implementation in the SMOL language and Section 5 the evaluation on a self-adaptative digital twin. Finally, Section 6 discusses related work and Section 7 concludes with a discussion.

Java

```java
class Plant { int id; String species; }
abstract class Pump {
  int id; int gpioPin; Plant plant;
  void pump(){ ... }; /* uses gpioPin and waters the plant */
}
class NormalPump extends Pump { ... /* methods */ }
class OverheatingPump extends Pump { int maximal; ... /* methods */ }
class Main() {
  Plant pl = new Plant(1, "Ocimum basilicum");
  Pump pu = new NormalPump(2, 7, pl);
  void loop() { while (true) { pu.pump(); System.wait(1); } }
  public static void main(String[] args) { new Main().loop(); }
}
```

Figure 1 Greenhouse digital twin, without domain knowledge and dynamic object reclassification.

2 Overview

This section introduces the programming challenges that we address with declarative dynamic object reclassification, and motivates different technical aspects of our solution. To ease readability in the examples in this section and in the formalisation of Section 3, programs are given with a Java-like syntax and knowledge bases in first-order logic (FOL).

2.1 Programming Challenges

Motivating Scenario. Let us consider as a running example a digital twin of a greenhouse which contains plants and water pumps. These are monitored through sensors measuring their physical *properties*: the height, oxygen level and soil moisture for each plant and the temperature for each pump. Each plant has an associated pump that waters the plant according to a watering profile. The concrete watering profile depends on the stage of the plant (e.g., seedling or mature plant) and the level of functionality of the pump. We first consider a simple configuration of the example in Sections 2 and 3, with one pump, two plants and a temperature monitor for the pump (see Figure 1); we later consider a larger configuration in Section 5, with three pumps, three plants and monitoring more properties.

In the digital twin, each plant is modelled by a `Plant` object and each pump by a `Pump` object. The `Plant` class contains the plant's identifier and scientific `species` name, so further information about the plant can be retrieved from a *domain knowledge base*. The `Pump` class contains the pump's identifier, a reference `pl` to the plant it is watering and a reference `gpioPin` to the output pin needed to operate the physical pump. The `pump` method implements the watering profile for the pump, which depends on the stage of the plant and whether the pump is fully operational or not: A pump is either operating normally (`NormalPump`) or overheated and in need of maintenance (`OverheatingPump`). To avoid overheating, an `OverheatingPump` should run at reduced power. Let us further assume that sensors (not specified here) update a *synchronisation knowledge base* containing the current state of the greenhouse. Thus, our program can retrieve information about the greenhouse from the knowledge base as needed.

The physical entities in the greenhouse evolve over time (e.g., a pump may threaten to overheat). The objects in the digital twin need to adapt their behaviour in accordance with the evolution of the physical entities they are twinning. To address this kind of problem, we propose a dynamic object reclassification technique that is *declarative*; i.e., the adaptation logic for reclassification is derived in the knowledge base. We now introduce a knowledge base for our example, and extend our program with constructs to interact with this knowledge base.

Knowledge Bases and Consistency. A knowledge base is used to organise and make contextual information accessible via an API. To keep the presentation abstract, we here represent knowledge in terms of first-order logic formulas and structure the knowledge base for our digital twin as follows: (1) *domain knowledge* about plants and pumps, see Figure 2a, including the description of the physical pumps and plants in the greenhouse (the so-called domain objects); and (2) *synchronisation knowledge* about the current value of the properties of these domain objects, see Figure 2b. Observe how *consistency* here comes into play: a `Pump` object in the digital twin should be classified in the subclass of `Pump` implementing the watering policy that is appropriate for the current stage of the pump domain object it twins and controls.

We use patterns from ontologies and knowledge representation to model domain knowledge, represent domain objects and domain classes as constants (nullary predicates), and prefix all predicates concerning the domain by `ctx_`. The *membership* of a domain object (e.g., `ctx_o`) in a domain class (e.g., `ctx_C`) is expressed by a binary predicate $\texttt{ctx_in}(\texttt{ctx_o}, \texttt{ctx_C})$. the *value* x of a property `ctx_prop` for a domain object y by the predicate $\texttt{ctx_prop}(y, x)$. For instance, the temperature x of a pump y is expressed by the predicate $\texttt{ctx_temp}(y, x)$.

The formulas **E1**–**E4** in Figure 2a express general knowledge about pumps; the remaining formulas specific knowledge about domain objects (here, one plant and one pump). Formula **E1** states that a pump (`ctx_Pump`) is operating normally (`ctx_NormalPump`) if its temperature is ≤ 50°C, and **E2** that a pump is in danger of overheating (`ctx_OverheatingPump`) if its temperature is > 50°C. Formulas **E3** and **E4** express that the property `ctx_id` uniquely identifies a domain object and that each pump can have only one temperature. The remaining formulas describe the plant and pump in our greenhouse, represented by constants `ctx_plant` and `ctx_pump`; `ctx_Plant` is the domain class of all plants. The last formula states that the pump `ctx_pump` has some temperature. The synchronisation knowledge base, in Figure 2b, contains, for each non-constant property `ctx_prop` of the system, an axiom that states the current value of `ctx_prop`. It contains a formula that states the actual temperature of `ctx_pump`.

To express consistency, we need to relate the `Pump` program objects to the current state of the actual pumps in the greenhouse, as represented by the `ctx_Pump` domain objects. To this aim, we represent the program code and its runtime configuration as knowledge bases. Figure 2c illustrates how the code in Figure 1 can be represented by a *code knowledge base*: predicate $\mathsf{isCls}(x)$ express whether x is a program program class; and $\mathsf{subclass}(x, y)$ express whether x is a subclass of y.

The representation of a program configuration in a knowledge base is known as *semantic lifting* [38], and uses a function defined in terms of the language to serialise runtime configurations into a *lifted-heap* knowledge base, where: $\mathsf{isObj}(x)$ express whether x is a program object (reference); $\mathsf{instOf}(x, y)$ whether x is an object instance of class y; $\mathsf{in}(x, y)$ whether x is an object instance of a subclass of y; and $\texttt{C_f}(x, y)$ express whether x is an object in class a `C` and its field `f` has value y. Figure 2d shows the lifted runtime configuration of the program in Figure 1 after execution of the expression `new Main()` in Line 14 and field initialisation in Lines 11 and 12: here, the program object ι_1, which is stored in `pl`, is an instance of program class `Plant` and ι_2 is an instance of program class `NormalPump`; and $\texttt{Pump_plant}(\iota_2,\iota_1)$ states that the field `plant` in ι_2 of program class `Plant` has value ι_1.

Observe that the fact **D1** $\texttt{ctx_in}(\texttt{ctx_pump}, \texttt{ctx_OverheatingPump})$ can be derived in the knowledge base in Figure 2 (by instantiating the universal quantifier in the axioms **E2** and **E3** with `ctx_plant`), which shows that the lifted runtime configuration in Figure 2d (stating that ι_2 is an instance of `NormalPump`) is *not consistent* with the context. To enable

$(\mathbf{E1})\ \forall x.\ \mathsf{ctx_in}(x, \texttt{ctx_NormalPump}) \Leftrightarrow \big(\mathsf{ctx_in}(x, \texttt{ctx_Pump}) \wedge \exists y.\ \texttt{ctx_temp}(x, y) \wedge y \leq 50\big)$,

$(\mathbf{E2})\ \forall x.\ \mathsf{ctx_in}(x, \texttt{ctx_OverheatingPump}) \Leftrightarrow \big(\mathsf{ctx_in}(x, \texttt{ctx_Pump}) \wedge \exists y.\ \texttt{ctx_temp}(x, y) \wedge y > 50\big)$,

$(\mathbf{E3})\ \forall x, y, z.\ (\texttt{ctx_id}(x, z) \wedge \texttt{ctx_id}(y, z)) \Rightarrow x \doteq y$,

$(\mathbf{E4})\ \forall x, y, z.\ (\texttt{ctx_temp}(x, y) \wedge \texttt{ctx_temp}(x, z)) \Rightarrow y \doteq z$,

$\mathsf{ctx_in}(\mathsf{ctx_plant}, \texttt{ctx_Plant})$, $\mathsf{ctx_in}(\mathsf{ctx_pump}, \texttt{ctx_Pump})$,

$\mathsf{ctx_id}(\mathsf{ctx_plant}, 1)$, $\mathsf{ctx_id}(\mathsf{ctx_pump}, 2)$, $\exists x.\mathsf{ctx_temp}(\mathsf{ctx_pump}, x)$

(a) The *domain knowledge base* for the greenhouse.

$\mathsf{ctx_temp}(\mathsf{ctx_pump}, 52)$

(b) A *synchronisation knowledge base* describing the temperature of the pump.

$\mathsf{isCls}(\texttt{Plant})$, $\mathsf{isCls}(\texttt{Pump})$, $\mathsf{isCls}(\texttt{NormalPump})$, $\mathsf{isCls}(\texttt{OverheatingPump})$, $\mathsf{isCls}(\texttt{Main})$,

$\mathsf{subclass}(\texttt{NormalPump}, \texttt{Pump})$, $\mathsf{subclass}(\texttt{OveheatingPump}, \texttt{Pump})$

(c) A snippet of the *code knowledge base* describing the code in Figure 1.

$\mathsf{isObj}(\iota_1)$, $\mathsf{isObj}(\iota_2)$, $\mathsf{isObj}(\iota_3)$, $\mathsf{instOf}(\iota_1, \texttt{Plant})$, $\mathsf{instOf}(\iota_2, \texttt{NormalPump})$,

$\mathsf{instOf}(\iota_3, \texttt{Main})$, $\texttt{Plant_id}(\iota_1, 1)$, $\texttt{Plant_name}(\iota_1, \texttt{"Ocimum basilicum"})$,

$\texttt{Pump_plant}(\iota_2, \iota_1)$, $\texttt{Pump_id}(\iota_2, 2)$, $\texttt{Pump_gpioPin}(\iota_2, 7)$

(d) The *lifted-heap knowledge base*, for the program's heap after the `Main` object has been initialised.

Figure 2 Knowledge bases for the digital twin: domain, synchronisation, code and lifted heap.

declarative dynamic object reclassification, we need to connect program objects (e.g., ι_1 and ι_2) and classes (e.g., `Pump`, `NormalPump` and `OverheatingPump`) to corresponding domain objects (e.g., `ctx_plant` and `ctx_pump`) and classes (e.g., `ctx_Pump`, `ctx_NormalPump` and `ctx_OverheatingPump`) in the knowledge base. In particular, we face the following challenges:

Ch1: How can we relate program objects with the external knowledge base, and define *external consistency*, i.e., consistency between program and external knowledge base?

Ch2: How can we program reactions to changes in the consistency of the relation between program and knowledge base?

Ch3: How can we ensure that establishing external consistency does not break internal consistency, i.e., the typing of the program?

Knowledge Bases as Interoperability Layers. While our approach is general to dynamic reclassification, it is worth commenting on the role of the knowledge base, which is motivated by digital twins. In digital twins, the focus is not on highly complex low-level control, but on modularity, composition, and semantic interoperability of different components, where knowledge bases are widely used [42, 60]. Often, all communication between the program and the physical component is handled via the knowledge base. The program is acting as a controller *and* a coordinating component that acts on data that has been already ingested into the knowledge base. This approach, which lends to blackboard architectures [25], also motivates our terminology of *external consistency*: The program is consistent with an external component (namely the physical twin), as described by the synchronization knowledge base. From a model-based perspective, this corresponds to *intermodel consistency* (see, e.g., Feichtinger et al. [19]), in contrast to internal, or intramodel, consistency, which corresponds to type soundness.

Extended Java code

```
1  class Plant { int id; String species; }
2  abstract class Pump classifies λself. ctx_in(self, ctx_Pump) {
3    int id; int gpioPin; Plant plant;
4    void pump(){ ... }; /* uses gpioPin and waters the plant */
5  }
6  class NormalPump extends Pump
7    links λself. ctx_in(self, ctx_Pump) ∧ ∀x. Pump_id(self, x) ⇒ ctx_id(self, x)
8    classifies λself. ctx_in(self, ctx_NormalPump) { ... /* methods */ }
9  class OverheatingPump extends Pump
10   links λself. ctx_in(self, ctx_Pump) ∧ ∀x. Pump_id(self, x) ⇒ ctx_id(self, x)
11   classifies λself. ctx_in(self, ctx_OverheatingPump)
12   retrieves λself, maximal. ∃x. ctx_profile(self, x) ∧ ctx_maximalPower(x, maximal)
13   { int maximal; ... /* methods */ }
14 class Main {
15   Plant pl = new Plant(1, "Ocimum basilicum");
16   Pump pu = new NormalPump(2, 7, pl);
17   void loop() { while (true) { adapt(pu);  pu.pump(); System.wait(1); } }
18   public static void main(String[] args) { new Main().loop(); }
19 }
```

(a) The greenhouse digital twin programmatic part with `links`, `classifies`, `retrieves` and `adapt`.

$$\forall y.\ \mathsf{instOf}(y, \mathsf{NormalPump}) \Rightarrow \big(\mathsf{ctx_in}(y, \mathsf{ctx_Pump}) \wedge \forall x.\ \mathsf{Pump_id}(y, x) \Rightarrow \mathtt{ctx_id}(y, x)\big),$$
$$\forall y.\ \mathsf{instOf}(y, \mathsf{OverheatingPump}) \Rightarrow \big(\mathsf{ctx_in}(y, \mathsf{ctx_Pump}) \wedge \forall x.\ \mathsf{Pump_id}(y, x) \Rightarrow \mathtt{ctx_id}(y, x)\big)$$

(b) Linkage knowledge base. Tautologies from the default `links` clauses of `Plant` and `Main` are omitted.

Figure 3 The greenhouse digital twin programmatic part and the linkage knowledge base.

2.2 Declarative Dynamic Object Reclassification

Based on the knowledge base and scenario above, we now illustrate the mechanisms in the programming language to connect the program with its context at runtime, shown in Figure 3. The formalisation and properties for these mechanisms will be explored in Section 3.

Linkage. To address the first part of challenge ***Ch1***, we propose to dynamically add knowledge about the dynamically created objects of the program's runtime state to the knowledge base. Technically, we add to each non-`abstract` class `C` a `links` clause with a single-parameter λ-predicate $\lambda self.\phi$ to express that $\forall x.\mathtt{instOf}(x, \mathtt{C}) \Rightarrow \phi[self := x]$ is added to the knowledge base (the λ-predicate $\lambda self.\mathtt{true}$ here acts as default). At runtime, dynamically created objects are lifted into the lifted-heap knowledge base by adding a fact about their class membership, e.g., $\mathtt{instOf}(\iota, \mathtt{C})$ for some object ι (cf. Section 2.1). In Figure 3 (Section 2.1), the program class `NormalPump` declares its objects to be members of domain class `ctx_NormalPump` and the domain id (`ctx_id`) and program id (`Pump_id`) to be the same. Similarly for `OverheatingPump` (Section 2.1). Linkage connects the objects of `C` to the context, using domain terminology (i.e., predicate symbols). Figure 3b shows the *linkage knowledge base* containing the added axioms. The first states that every program object that is an instance of `NormalPump`, is also a pump (i.e., a `ctx_Pump` in the domain context), and that the ids are the same. The same states the same for `OverheatingPump`. This links the program object to the domain knowledge, as one can use *domain* axioms (such as **E1** and **E2**) to reason about the *program* objects. This is done as follows. From the lifted heap in Figure 2d, we derive that ι_2 is member of `ctx_OverheatingPump` as follows: By lifting, we have $\mathsf{instOf}(\iota_2, \mathtt{NormalPump})$ and by linkage, $\mathtt{ctx_id}(\iota_2, 2)$. Since `ctx_id` uniquely identifies an object (cf. **E3**), $\mathtt{ctx_pump} \doteq \iota_2$. Applying this equality in the formula $\mathtt{ctx_in}(\mathtt{ctx_pump}, \mathtt{ctx_OverheatingPump})$ (**D1**, see above), we obtain $\mathtt{ctx_in}(\iota_2, \mathtt{ctx_OverheatingPump})$.

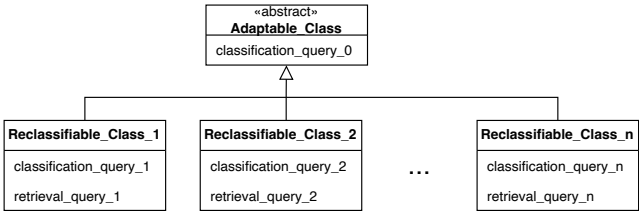


Figure 4 The structure of a class hierarchy for declarative dynamic object classification.

Consistency Declaration. To address the second part of challenge ***Ch1***, we propose to characterise in the knowledge base when an object should belong to a particular class. To this aim, we let objects change between *reclassifiable* subtypes of an *adaptable* (abstract) class. Technically, we add a `classifies` clause with a single-parameter λ-predicate to these classes. If this λ-predicate holds when instantiated for a program object of the given program class, the program object is considered consistent with its context. In our example (see Figure 2), ι_2 is *not* consistent. The only classes with a `classifies` clause are

1. *adaptable* classes, i.e, `abstract` classes that extend classes with no `classifies` clause; and
2. *reclassifiable* classes, i.e., all the subclasses of the adaptable classes (introduced in point 1), that must be non-`abstract` and have no subclasses.

Remark that, as illustrated in Figure 4, each reclassifiable class must (1) directly inherit from an adaptable class; (2) not be inherited from; and (3) have only reclassifiable sibling classes.

Reclassification Queries. Reclassification is the process of changing the program class of an inconsistent program object to a program class that makes the program object consistent. To address challenge ***Ch2***, we provide language support for programming such reclassification. The expression `adapt`(e) (Section 2.1) takes a program object that must be an instance of a reclassifiable class as argument, and checks whether this program object is consistent with the context; if this is not the case, it is reclassified. This is done by searching the knowledge base for a sibling to its current program class in the class hierarchy, that would make the object consistent. For the running example (see Figure 2), this would be the `OverheatingPump` class. Remark that the `adapt` expression is an explicit mechanism to ensure external consistency, but far from the only one. External consistency can also be established implicitly, either lazily, whenever an object is accessed, or periodically in the background, akin to a garbage collector.

To support state transfer for the reclassified object to the fields of its new program class, the programmer adds a `retrieves` clause to each reclassifiable class. This clause provides a λ-predicate with one parameter for the object in question and one for each field declared in the class. We can then infer values from the knowledge base that make the λ-predicate hold. Consider the `OverheatingPump` in Figure 3a: the reclassification query (Section 2.1) first retrieves the profile of the given pump, and then the maximal power it can use when in this stage.

Ensuring Consistency and Type Soundness. To address challenge ***Ch3***, let us observe that it is crucial that the dynamic operations on the knowledge base maintain satisfiability of the knowledge base – otherwise, the program's interactions with the knowledge base would allow incorrect information to be inferred, potentially violating type soundness. Technically, we first ensure that reclassifiable classes are not used as types; e.g., a variable of type `OperationalPump` is not allowed, as it may cause an error when accessed after reclassification. Both this restriction and the structure illustrated in Figure 4 are enforced by the type system; they guarantee that no unsafe access is possible.

Prg ::=	$\mathcal{K}$ $\overline{\texttt{CD}}$ e	Program
$\mathcal{K}$ ::=	$\{\overline{\phi}\}$	Knowledge base
CD ::=	`class` C [`extends` C] [Links] [Adapt] {$\overline{\texttt{FD}}$ $\overline{\texttt{MD}}$}	Class
FD ::=	T f;	Field
T ::=	C | `int`	Type
MD ::=	MH {`return` e;}	Method
MH ::=	T m($\overline{\texttt{T}}$ $\overline{\texttt{x}}$)	Method header
e ::=	x | n | e.f | e.m($\overline{\texttt{e}}$) | `new` C($\overline{\texttt{e}}$) | e.f = e | `null` | `adapt`(e)	Expression
Links ::=	`links` $\lambda z.\phi$	Linkage
Adapt ::=	`classifies` $\lambda z.\phi$ [`retrieves` $\lambda z\overline{z}.\phi$];	Adaptation

Figure 5 FSRJ syntax, where ϕ ranges over first-order logic (FOL) formulas, `C` over class names, `f` over field names, `T` over types (i.e., class names and the basic type `int`), `m` over method names, `x` over local variables (i.e., formal parameters to methods) and z over term variables (so $\lambda z.\phi$ is a unary λ-predicate and $\lambda z\overline{z}.\phi$ a λ-predicate of arity ≥ 1).

To ensure that the interaction between the program and the knowledge base does not introduce errors, we introduce a notion of *program coherence* roughly expressing that

1. every lifted object of a reclassifiable class can be classified as member of the adaptable superclass;
2. every lifted object that is member of an adaptable class can also be classified as member of one of the reclassifiable subclasses; and
3. the `retrieves` λ-predicate associates a value of the correct type to each field.

We say that an object is *hot* to mean that a method invoked on the object has an activation record on the stack, and *cold* otherwise. In the next section we will see that program coherence ensures that reclassification of cold objects is always successful.

3 FSRJ: Featherweight Semantically Reflected Java

This section provides a formal account of programming with semantic reflection in terms of *Featherweight Semantically Reflected Java (FSRJ)*, a minimal core calculus for class-based object-oriented languages with semantic reflection. Our purpose with FSRJ is to formalise the interplay between program execution, semantic reflection and the querying of knowledge bases for dynamic object reclassification. FRSJ is a variant of Featherweight Java (FJ) [31].[1]

3.1 FSRJ Syntax

The syntax of FSRJ is given in Figure 5. We let $\overline{\cdot}$ represent (possibly empty) sequences and $[\cdot]$ represent optional elements. Operations on sequences of pairs are abbreviated in a similar way; e.g., $\overline{\texttt{C}}$ $\overline{\texttt{f}}$ is shorthand for $\texttt{C}_1$ $\texttt{f}_1, \ldots, \texttt{C}_n$ $\texttt{f}_n$. The empty sequence is denoted by $\bullet$ and the length of a sequence $\overline{\texttt{e}}$ is denoted by $\#(\overline{\texttt{e}})$. The set of program variables includes the reserved name `this`, which cannot be used as a name of a field or a method's formal parameter.

In FSRJ, a program Prg consists of a knowledge base $\mathcal{K}$, a list of class definitions $\overline{\texttt{CD}}$ and a main expression e. The knowledge base $\mathcal{K}$, called the *domain knowledge base* of Prg, is set of FOL formulas. A class definition `class C [extends D]` [Links] [Adapt] {$\overline{\texttt{FD}}$ $\overline{\texttt{MD}}$} consists of

[1] FSRJ does not technically extend FJ because cast-expressions are excluded; these are orthogonal to semantic reflection and object reclassification. Further, FSRJ is imperative (it features a field assignment expression), has no implicit root class (`Object`), and classes in FSRJ need not have a superclass and have an implicit constructor which initialises all the fields.

a class name C, an optional clause `extends D` to express inheritance (so D is the immediate superclass of C), an optional linkage clause Links = `links` $\lambda z.\phi$, an optional adaptation clause Adapt = `classifies` $\lambda z.\phi$ [`retrieves` $\lambda z\overline{z}.\psi$] (where, in turn, the `retrieves` part is optional), a list of field definitions $\overline{\texttt{FD}}$ and a list of method definitions $\overline{\texttt{MD}}$. The *knowledge base*, formalising the context in which the program executes, is detailed in Section 3.2. The *linkage clause* expresses that every instance o of C is logically linked by $\lambda z.\phi$; i.e., given the address ι of o, the formula $\phi[z := \iota]$ holds in the context of the program. The *adaptation clause* expresses that if C has the immediate superclass D, then every instance o of an immediate subclass of D is reclassifiable to C by $\lambda z.\phi$ and $\lambda z\overline{z}.\psi$; i.e., given the address ι of o, if the formula $\phi[z := \iota]$ holds in the context of the program then there exist values $\overline{v}$ for the fields of C such that the formula $\psi[z\overline{z} := \iota\overline{v}]$ holds in the context of the program. Types T include names of declared classes and the basic type `int`. Field and method declarations are standard; the fields declared in a class are added to the ones declared in its superclasses, and are assumed to have distinct names (i.e., no field shadowing is allowed). Method names are assumed to be unique in the class; i.e., no method overloading is allowed. It is, however, possible to override methods from the superclass. All fields and methods are public, and each class has an implicit constructor that takes a parameter for each field (inherited or defined) in the class and initialises all the fields.

We assume that the elements of lists of named entities (i.e., class definitions, field definitions, method definitions, method formal parameter declarations) have different names. To simplify the presentation, we use a program Prg as a mapping from class names C to the corresponding class definitions CD; and use a class definition CD as a mapping that maps field and method names to field and method definitions, may map the keyword `extends` to the name of its superclass D, may map each of the keywords `links` and `classifies` to a unary λ-predicate of arity one and the keyword `retrieves` to a λ-predicate of arity $n+1$, where n is the number of fields (defined or inherited) in the class. We assume that programs Prg satisfy the following *well-formedness conditions:*

1. For a class name C appearing anywhere in Prg, we have $\texttt{C} \in \texttt{dom(Prg)}$.
2. There are no cycles in the transitive closure of the immediate `extends` relation.

The *subclassing* relation $<:$ is the reflexive and transitive closure of the immediate `extends` relation. The *subtyping* relation $\leq$ extends $<:$ by adding reflexivity on basic types, i.e., $\texttt{int} \leq \texttt{int}$.

In the sequel, we let the metavariable a range over names and the metavariable AD over definitions. The lookup of the definition of a field or method a in a class C is denoted by `aDef(C)(a)`. For each class in `dom(Prg)`, the function `aDef(C)` is defined as follows:

$$\texttt{aDef(C)(a)} = \begin{cases} \texttt{Prg(C)(a)} & \text{if } \texttt{a} \in \texttt{dom(Prg(C))} \\ \texttt{aDef(D)(a)} & \text{if } \texttt{a} \notin \texttt{dom(Prg(C))} \text{ and } \texttt{Prg(C)(extends)} = \texttt{D} \end{cases}$$

Given a field definition `FD = T f` and a method definition $\texttt{MD} = \texttt{T m } (\overline{\texttt{T}}\ \overline{\texttt{x}})\ \{\cdots\}$, let `signature(FD)` denote the type T of field f and `signature(MD)` the type $\overline{\texttt{T}} \to \texttt{T}$ of method m.

▶ **Example 1** (Greenhouse digital twin in FSRJ). Let us consider the encoding of the Greenhouse example (Figure 3a) in FSRJ. Sequential composition, which is not part of the syntax of FSRJ, can be encoded. For $\texttt{e}_1;\cdots;\texttt{e}_n$ of $n \geq 1$ expressions, define an auxiliary class `Encode` with a method $\texttt{T}_n$ `sqT1_T2_...Tn` $(\texttt{T}_1\ \texttt{x}_1, ..., \texttt{T}_n\ \texttt{x}_n)$ `{ return` $\texttt{e}_n$`;}`, where $\texttt{T}_i$ is the type of $\texttt{e}_i$ $(1 \leq i \leq n)$. Namely, $\texttt{e}_1;\cdots;\texttt{e}_n$ can be expressed as `new Encode().sqT1...Tn(`$\texttt{e}_1, ..., \texttt{e}_n$`)`.

Then, if we ignore the keyword `abstract`, consider `void` as syntactic sugar for `int`, consider the string literal `"Ocinum basilicum"` as syntactic sugar for an integer literal, and consider `System.wait(1)` as syntactic sugar for `new System().wait(1)` where `System` is a

class providing a method `int wait(int x) {...}`, we have that the Java-like code fragment in Figure 3a can be turned into an FSRJ code fragment by replacing the code of the class `Main` (in lines 16-21) by the following FSRJ code:

FSRJ

```
class Main { Plant pl; Pump pu;
 int loop(){ return new Encode().scPump_Plant_int_int_int(pu.plant=this.pl,
                    adapt(pu), pu.pump(), System.wait(1), this.loop(); }
} // Main expression:
new Main(new Plant(1,"Ocinum basilicum"), new NormalPump(2,7,null)).loop()
```

3.2 A FOL Representation of Knowledge Bases

To avoid the technicalities of knowledge representation languages in FSRJ, the knowledge base $\mathcal{K}$ of an FSRJ program is given as a set of closed FOL formulas over a given signature. Formally, signatures and formulas over signatures are defined as follows.

▶ **Definition 2** (Signatures). *A* signature *is a set of predicate symbols* $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2$*, where* Σ_0 *is a set of 0-ary predicate symbols (constants),* Σ_1 *a set of unary predicate symbols, and* Σ_2 *a set of binary predicate symbols (relations). Let* c *range over* Σ_0*,* P *over* Σ_1 *and* R *over* Σ_2*.*

▶ **Definition 3** (FOL formulas and terms over a signature). *Let term variables* z *range over constants in a signature* Σ*. The* formulas ϕ *and* terms t *over* Σ *are defined as follows:*

$$\phi ::= \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \oplus \phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid \exists z.\ \phi \mid \forall z.\ \phi \mid P(t) \mid R(t,t) \mid t \doteq t$$
$$t ::= z \mid c$$

Thus, in a λ*-predicate* $\lambda z\overline{z}.\psi$, $z\overline{z}$ is a (non-empty) list of term variables and ϕ a FOL formula. We denote by $\mathsf{FV}(t)$ the free variables in a term t and by $\mathsf{FV}(\phi)$ the free variables in a formula ϕ. The *substitution* $\phi[z := t]$ replaces all free occurrences of the term variable z by the term t in the formula ϕ (we omit its inductive definition here). Given a knowledge base $\mathcal{K}$ and a formula ψ, we let $\mathcal{K} \Rightarrow \psi$ abbreviate $\big(\wedge_{\phi\in\mathcal{K}}\ \phi\big) \Rightarrow \psi$.

Recall that the semantics of FOL with equality defines the values of terms to be individuals of a domain with an identity-relation, and the values of (well-formed) formulas to be truth values [4]. Thus, the semantics of a formula over a signature Σ is defined relative to an equality-model $\mathcal{M} = \langle \mathcal{D}, \mathcal{I}, \approx \rangle$ and a variable assignment β that maps term variables to elements in the domain $\mathcal{D}$, such that: (1) the interpretation $\mathcal{I} : \Sigma \to \mathcal{D}^n$ maps symbols in Σ_2 to binary relations over $\mathcal{D} \times \mathcal{D}$, symbols in Σ_1 to unary predicates over $\mathcal{D}$, and symbols in Σ_0 to distinct elements in $\mathcal{D}$; and (2) the interpretation $\mathcal{I}(\doteq)$ is the equality-relation $\approx$ on $\mathcal{D}$. In the following definition of satisfiability, we omit the operators $\forall$, $\wedge$, $\oplus$ (which denotes exclusive or), $\Rightarrow$ and $\Leftrightarrow$, since their semantics can be derived by the semantics of the other operators.

▶ **Definition 4** (Satisfiability). *Let* Σ *be signature,* $\mathcal{M} = \langle \mathcal{D}, \mathcal{I}, \approx \rangle$ *an equality-model such that* $\mathcal{I} : \Sigma \to \mathcal{D}$*, and* β *a variable assignment. The values of terms* t *with respect to* β *in* $\mathcal{M}$*, denoted* $[\![t]\!]_{\mathcal{M},\beta}$*, are defined by* $[\![z]\!]_{\mathcal{M},\beta} = \beta(z)$ *and* $[\![c]\!]_{\mathcal{M},\beta} = \mathcal{I}(c)$*. The relation* $\mathcal{M}, \beta \models \phi$*, expressing that a formula* ϕ *is* satisfied *by* $\mathcal{M}$ *and* β*, is defined inductively as follows:*

$$\begin{array}{ll}
\mathcal{M},\beta \models \neg\phi & \textit{iff } \mathcal{M},\beta \not\models \phi \\
\mathcal{M},\beta \models \phi_1 \vee \phi_2 & \textit{iff } \mathcal{M},\beta \models \phi_1 \textit{ or } \mathcal{M},\beta \models \phi_2 \\
\mathcal{M},\beta \models \exists x.\ \phi & \textit{iff } \mathcal{M},\beta[x \mapsto d] \models \phi \textit{ for some } d \in D \\
\mathcal{M},\beta \models P(t) & \textit{iff } [\![t]\!]_{\mathcal{M},\beta} \in \mathcal{M}(P) \\
\mathcal{M},\beta \models R(t_1,t_2) & \textit{iff } \big([\![t_1]\!]_{\mathcal{M},\beta}, [\![t_2]\!]_{\mathcal{M},\beta}\big) \in \mathcal{M}(R)
\end{array}$$

A formula ϕ over Σ is (1) satisfiable *if $\mathcal{M}, \beta \models \phi$ holds for some $\mathcal{M}$ and β, and (2)* valid*, denoted $\models \phi$, when ϕ is satisfiable for every $\mathcal{M}$ and variable assignment.*
A knowledge base $\mathcal{K}$ is satisfiable/valid if $\wedge_{\phi \in \mathcal{K}} \phi$ is satisfiable/valid.
Given a knowledge base $\mathcal{K}$ and a formula ψ, we write $\mathcal{K} \models \psi$ as shorthand for $\models \mathcal{K} \Rightarrow \psi$.

▶ **Example 5** (Domain knowledge base for the Greenhouse digital twin in FSRJ)**.** The domain knowledge base for the FSRJ Greenhouse digital twin is given in Figure 2a.

3.3 Program Typing

In this section we present the type system for FSRJ, detailing especially on the language aspects related to knowledge bases, semantic reflection and dynamic object reclassification.

The function `type`, that extracts the type from a field or method definition, is defined by: `type(T f) = T` and `type(T m(`$\overline{\texttt{T}}$ $\overline{\texttt{x}}$`){···}) =` $\overline{\texttt{T}} \to$ `T`. The lookup of the type of a field or method `a` in the program is defined b:y `aType(C)(a) = type(aDef(C)(a))`. With respect to reclassification, we say that a class `C` ∈ `dom(Prg)` is:

- *standard*, written Std(`C`), if (1) `links` ∈ `dom(Prg(C))`; (2) `classifies` ∉ `dom(Prg(C))`; (3) `retrieves` ∉ `dom(Prg(C))`; and (4) if `C` has a superclass, then this superclass is standard (i.e., if `Prg(C)(extends) = B` for some `B`, then Std(`B`)).
- *adaptable*, written Adp(`C`), if (1) `links` ∉ `dom(Prg(C))`; (2) `classifies` ∈ `dom(Prg(C))`; (3) `retrieves` ∉ `dom(Prg(C))`; and (4) if `C` has a superclass, then this superclass is standard (i.e., if `Prg(C)(extends) = B` for some `B`, then Std(`B`)).
- *reclassifiable*, written Rcl(`C`), if (1) `links` ∈ `dom(Prg(C))`; (2) `classifies` ∈ `dom(Prg(C))`; (3) `retrieves` ∈ `dom(Prg(C))`; and (4) `C` has a superclass which is adaptable (that is, `Prg(C)(extends) = B` and Adp(`B`), for some `B`).

By inspecting the program, it is possible to check that there are no cycles in the transitive closure of the `extends` relation; that, for each class `C` in `dom(Prg)`, the names of the fields in `C` are distinct from the names of inherited fields; that each class is either standard, or adaptable, or reclassifiable; and that each adaptable class has at least a subclass (which is necessarily reclassifiable). We write `sane(Prg)` to express that the program `Prg` satisfies the well-formedness conditions in Section 3.1 and that the following sanity conditions:

1. $\texttt{C}_1 <: \texttt{C}_2$ implies that, for all method names `m`, if $\texttt{aType}(\texttt{C}_2)(\texttt{m})$ is defined, then $\texttt{aType}(\texttt{C}_1)(\texttt{m}) = \texttt{aType}(\texttt{C}_2)(\texttt{m})$.
2. $\texttt{C}_1 <: \texttt{C}_2$ and $\texttt{C}_1 \neq \texttt{C}_2$ imply that, for all field names `f`, if $\texttt{f} \in \texttt{dom}(\texttt{Prg}(\texttt{C}_1))$ then $\texttt{f} \notin \texttt{dom}(\texttt{Prg}(\texttt{C}_2))$.
3. For every `C` ∈ `dom(Prg)`, exactly one of Std(`C`), Adp(`C`) or Rcl(`C`) holds.
4. For every `C` ∈ `dom(Prg)`, if Adp(`C`) then there exists `D` ∈ `dom(Prg)` s. t. `Prg(D)(extends) = C`.
5. Reclassifiable class names are not used as field types, as method parameters or return types.

The typing rules and field lookup rule for FSRJ are given in Figure 6. Rule T-knowledge enforces that the domain knowledge base consists of closed formulas. The typing of expressions is standard, except for rule T-new, which enforces new objects to be typed either by an adaptable or a standard class, and rule T-adapt, which enforces the `adapt`-expression and the argument of `adapt` to be typed by an adaptable class `C`. This allows the argument of `adapt` to be dynamically reclassified to any subclass of `C`.

The typing of `links` clauses ensures that the linkage formula only contains one free variable z, which ensures that the formula can be closed by instantiating z. Similarly, the typing of the `adapts` clause ensures that the free variables of the `classifies`- and `retrieves`-formulas

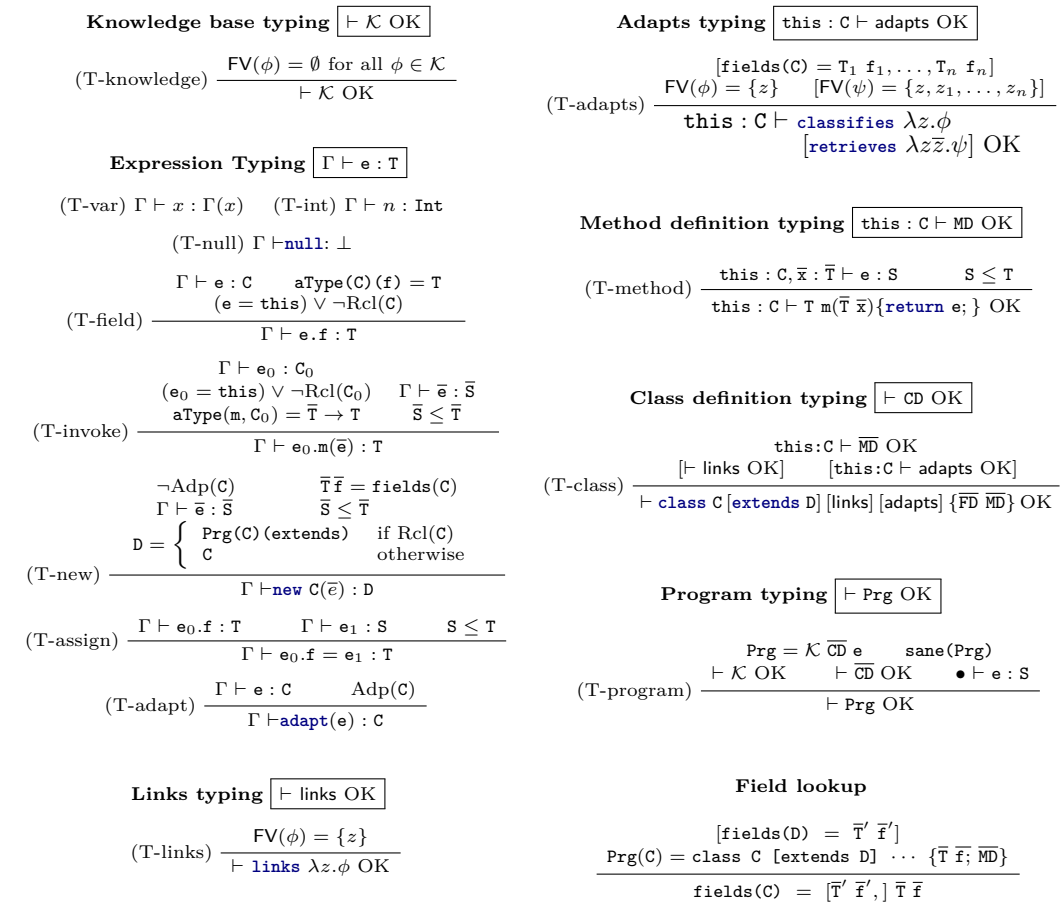


Figure 6 FSRJ: typing rules and field lookup rule.

are known. The typing rules for method and class definitions and for programs are standard recursively checking the different syntactic elements; the latter rule additionally checks that the program is sane. In the type system, the auxiliary function `fields(C)` retrieves all field definitions of a given class `C` in the program `Prg`, including the inherited ones. To type the `null` value, we use the special type $\bot$, which is not a class name, not part of the program and a subtype of any class name. In the sequel, we use the following convention:

- the metavariable `T` denotes either a non-reclassifiable class name or `int`; and
- the metavariable `S` denotes either a non-reclassifiable class name, `int` or $\bot$.

We say that a program `Prg` is well-typed to mean that $\vdash$ `Prg` OK holds.

▶ **Lemma 6** (Syntactic guarantees for well-typed programs). *Let* $\vdash$ *`Prg` OK. Then `sane(Prg)` and if* `new C(...)` *appears in `Prg` then `C` is not adaptable.*

3.4 Code, Linkage and Program Knowledge Bases

The purpose of this section is to explain how to formalise program-specific knowledge bases in FOL. Given a well-typed program $\mathsf{Prg} = \mathcal{K}\,\overline{\mathtt{CD}}\,\mathtt{e}$ with domain knowledge base $\mathcal{K}$, we first define a knowledge base $\mathcal{K}^{\mathsf{code}}$ that reflects part of the code of `Prg` in FOL. The knowledge base $\mathcal{K}^{\mathsf{code}}$ includes general axioms and symbols common to all FSRJ programs, as well as knowledge specifically derived from `Prg`, such as knowledge about declared classes and fields.

$A1$. $\forall x.\ (x \doteq \mathsf{int}) \oplus (x \doteq \mathsf{null}) \oplus \mathsf{isInt}(x) \oplus \mathsf{isObj}(x) \oplus \mathsf{isCls}(x)$,
$A2$. $\forall x.\ \mathsf{isCls}(x) \Leftrightarrow (\mathsf{isStd}(x) \oplus \mathsf{isAdp}(x) \oplus \mathsf{isRcl}(x))$
$A3$. $\mathsf{isInt}(n)$ for all $n \in \mathbb{Z}_k$
$A4$. $\mathsf{isCls}(\mathtt{C})$ for all $\mathtt{C} \in \mathit{Cls}$
$A5$. $\mathsf{isStd}(\mathtt{C})$ for all $\mathtt{C} \in \mathit{Cls}$ such that $\mathrm{Std}(\mathtt{C})$
$A6$. $\mathsf{isAdp}(\mathtt{C})$ for all $\mathtt{C} \in \mathit{Cls}$ such that $\mathrm{Adp}(\mathtt{C})$
$A7$. $\mathsf{isRcl}(\mathtt{C})$ for all $\mathtt{C} \in \mathit{Cls}$ such that $\mathrm{Rcl}(\mathtt{C})$
$A8$. $\forall x.\ \left(\wedge_{n \in \mathbb{Z}_k} \neg(x \doteq n)\right) \Rightarrow \neg\mathsf{isInt}(x)$
$A9$. $\forall x.\ \left(\wedge_{\mathtt{C} \in \mathit{Cls}} \neg(x \doteq \mathtt{C})\right) \Rightarrow \neg\mathsf{isCls}(x)$
$A10$. $\forall x, y.\ \mathsf{subclass}(x, y) \Rightarrow (\mathsf{isCls}(x) \wedge \mathsf{isCls}(y))$
$A11$. $\mathsf{subclass}(\mathtt{C}, \mathtt{C}')$ for all $\mathtt{C}, \mathtt{C}' \in \mathit{Cls}$ such that $\mathtt{C} <: \mathtt{C}'$
$A12$. $\neg\mathsf{subclass}(\mathtt{C}, \mathtt{C}')$ for all $\mathtt{C}, \mathtt{C}' \in \mathit{Cls}$ such that $\mathtt{C} \not<: \mathtt{C}'$
$A13$. $\forall x, y.\ \mathsf{compatible}(x, y) \Leftrightarrow \exists z.\ \mathsf{subclass}(x, z) \wedge \mathsf{subclass}(y, z) \wedge \mathsf{isAdp}(z)$
$A14$. $\forall x, y.\ (\mathsf{instOf}(x, y) \vee \mathsf{in}(x, y)) \Rightarrow (\mathsf{isObj}(x) \wedge \mathsf{isCls}(y))$
$A15$. $\forall x.\ \mathsf{isObj}(x) \Rightarrow \exists y.\ \mathsf{isCls}(y) \wedge \mathsf{instOf}(x, y) \wedge \neg\mathsf{isAdp}(y)$
$A16$. $\forall x, y, z.\ (\mathsf{instOf}(x, y) \wedge \mathsf{instOf}(x, z)) \Rightarrow (y \doteq z)$
$A17$. $\forall x, y.\ \mathsf{in}(x, y) \Leftrightarrow \exists z.\mathsf{instOf}(x, z) \wedge \mathsf{subclass}(z, y)$
$A18$. $\forall x, y.\ \mathsf{hasType}(x, y) \Leftrightarrow (\mathsf{in}(x, y) \vee (\mathsf{isInt}(x) \wedge (y \doteq \mathsf{int})) \vee ((x \doteq \mathsf{null}) \wedge \mathsf{isCls(y)}))$
$A19$. $\forall x.\ \mathsf{in}(x, \mathtt{C}) \Leftrightarrow \exists y.\ \mathtt{C_f}(x, y)$ for all $\mathtt{C} \in \mathit{Cls}$ and $\mathtt{C_f} \in \mathit{Fls}(\mathtt{C})$
$A20$. $\forall x, y, z.\ (\mathtt{C_f}(x, y) \wedge \mathtt{C_f}(x, z)) \Rightarrow (y \doteq z)$ for all $\mathtt{C} \in \mathit{Cls}$ and $\mathtt{C.f} \in \mathit{Fls}(\mathtt{C})$
$A21$. $\forall x, y.\ \mathtt{C_f}(x, y) \Rightarrow \mathsf{in}(x, \mathtt{C}) \wedge \mathsf{hasType}(y, \mathtt{type(C_f)})$ for all $\mathtt{C} \in \mathit{Cls}$ and $\mathtt{C.f} \in \mathit{Fls}(\mathtt{C})$

Figure 7 The code knowledge base.

▶ **Definition 7** (Code knowledge bases). *Given a well-typed FSRJ program* Prg *with domain knowledge base $\mathcal{K}$ over the signature Σ, let $\mathit{Cls} =$ `dom(Prg)` be the set of class names in* Prg*, $\mathit{Fls}(\mathtt{C})$ the set of qualified field names* `C.f` *declared (not inherited) in class* `C`*,* `type(C.f)` *the type of the field* `C.f`*, and $\mathbb{Z}_k = \{z \mid -k \leq z \leq k\}$ (where k is natural number) a set of integers. We denote by Σ^{code} the following signature (where subscripts denote the arity of non-constant symbols):*

$$\mathbb{Z}_k \cup \{\mathsf{int}, \mathsf{null}\} \cup \mathit{Cls} \cup \{\mathtt{C_f}_2 \mid \mathtt{C.f} \in \mathit{Fls}(\mathtt{C})\}$$
$$\cup\{\mathsf{isInt}_1, \mathsf{isObj}_1, \mathsf{isCls}_1, \mathsf{isStd}_1, \mathsf{isAdp}_1, \mathsf{isRcl}_1, \mathsf{subclass}_2, \mathsf{in}_2, \mathsf{compatible}_2, \mathsf{instOf}_2, \mathsf{hasType}_2\}$$

and (without loss of generality) assume that $\Sigma \cap \Sigma^{\mathsf{code}} = \mathbb{Z}_k \cup \{\doteq\}$. The code knowledge base $\mathcal{K}^{\mathsf{code}}$ of Prg *is the set consisting of the formulas over Σ^{code} in Figure 7.*

We briefly discuss the formulas of $\mathcal{K}^{\mathsf{code}}$. Axiom $A1$ expresses that each element in the domain must be either the type name `int`, the value `null`, an integer value, an object or a class; and $A2$ that each class is either standard, adaptable or reclassifiable. Axioms $A3$ declare that each constant $n \in \mathbb{Z}_k$ is an integer and $A4$ declare that each constant $\mathtt{C} \in \mathit{Cls}$ is a class. Axioms $A5$–$A7$ declare the kind (standard, adaptable, reclassifiable) of each constant $\mathtt{C} \in \mathit{Cls}$. Axioms $A8$ express that each integer in the domain is mapped to a constant $n \in \mathbb{Z}_k$ and $A9$ that each class in the domain is mapped to a constant $\mathtt{C} \in \mathit{Cls}$. Axioms $A10$–$A12$ express that the subclass relation relates classes to classes and capture the $<:$ relation of `Prg`. Axiom $A13$ expresses that two (reclassifiable) classes are compatible if and only if they are both subclasses of the same adaptable class. Axiom $A14$ expresses that both the instOf and the in relations relate objects with classes. Axioms $A15$ and $A16$ declare that each object is an instOf of exactly one non-adaptable class. Axiom $A17$ declares that an object is in a class if and only if it is an instOf of a subclass of that class. Axiom $A18$ declares that only null,

integers and objects have types, and which are their types. Axioms $A19$–$A21$ express that the predicate `C_f` models the field `f` of objects of any subclass of `C`, each object has exactly the fields (declared or inherited) in its class, and each field contains exactly one value of the declared type.

▶ **Example 8** (Code knowledge base for the Greenhouse digital twin in FSRJ)**.** A subset of the code knowledge base for the FSRJ program in Example 1 (namely, the subset containing axioms A4 and A11) is given in Figure 2c.

We define the linkage knowledge base to express that each object satisfies the predicate specified by the `links` clause of its class.

▶ **Definition 9** (Linkage knowledge bases)**.** *Let* Prg *be a well-typed FSRJ program with domain knowledge base $\mathcal{K}$ over Σ. Let $Cls = \mathit{dom(Prg)}$. The* linkage knowledge base $\mathcal{K}^{\mathsf{link}}$ of Prg *is the set of formulas over the signature $\Sigma \cup \Sigma^{\mathsf{code}}$ defined by $\forall x.\ \mathsf{instOf}(x, \mathtt{C}) \Rightarrow \phi[z := x]$ for all $\mathtt{C} \in Cls$ and $\mathit{Prg}(\mathtt{C})(\mathtt{links}) = \lambda z.\phi$.*

▶ **Example 10** (Linkage knowledge base for the Greenhouse digital twin in FSRJ)**.** The linkage knowledge base for the FSRJ program of Example 1 is given in Figure 3b.

Observe that by combining Σ and Σ^{code}, predicates in $\mathcal{K}^{\mathsf{link}}$ may express connections between an FSRJ program's runtime values (e.g., instances of a class `C`) and the domain knowledge base. We define a knowledge base that comprises the three knowledge bases defined above:

▶ **Definition 11** (Program knowledge bases)**.** *Let* Prg *be a well-typed FSRJ program with domain knowledge base $\mathcal{K}$ over the signature Σ. The program knowledge base of* Prg *is the knowledge base $\mathcal{K}^{\mathsf{Prg}} = \mathcal{K} \cup \mathcal{K}^{\mathsf{code}} \cup \mathcal{K}^{\mathsf{link}}$ over the signature $\Sigma \cup \Sigma^{\mathsf{code}}$.*

▶ **Lemma 12** (Satisfiability of the program knowledge base)**.** *For every well-typed program* Prg *with domain knowledge base $\mathcal{K}$ over the signature Σ. If $\mathcal{K}$ is satisfiable, then the knowledge base $\mathcal{K}^{\mathsf{Prg}} = \mathcal{K} \cup \mathcal{K}^{\mathsf{code}} \cup \mathcal{K}^{\mathsf{link}}$ over the signature $\Sigma \cup \Sigma^{\mathsf{code}}$ is satisfiable.*

3.5 Program Coherence

This section considers requirements on FSRJ programs `Prg` to ensure that reasoning with the program knowledge base $\mathcal{K}^{\mathsf{Prg}}$ is in fact meaningful for declarative object reclassification. These requirements are needed to ensure type soundness; even with a satisfiable domain knowledge base, linking axioms can make the knowledge base incoherent (e.g., define a linkage predicate to introduce false into the knowledge base). Below, let $\lambda z.\phi^{\mathtt{C}} = \mathtt{Prg}(\mathtt{C})(\mathtt{classifies})$ and $\mathrm{Children}(\mathtt{C}) = \{\mathtt{D} \in \mathtt{dom}(\mathtt{Prg}) \mid \mathtt{Prg}(\mathtt{D})(\mathtt{extends}) = \mathtt{C}\}$ for classes `C`. We say that a well-typed program `Prg` with domain knowledge base $\mathcal{K}$ over a signature Σ is *coherent* if:

Coh1: $\mathcal{K}$ is satisfiable. This requirement ensures that models exist for the domain knowledge base.

Coh2: $\mathcal{K} \cup \mathcal{K}^{\mathsf{code}} \models \mathcal{K}^{\mathsf{links}}$. This requirement ensures that the user-defined linkage axioms exclude no interpretation that satisfies the union of the domain and code knowledge bases $\mathcal{K} \cup \mathcal{K}^{\mathsf{code}}$; thus ensuring that each object satisfies the linkage predicate of its class.

Coh3: In every model satisfying $\mathcal{K}^{\mathsf{Prg}}$, for every object x instance of a reclassifiable subclass of an adaptable class `C`, the classification predicate $\phi^{\mathtt{C}}$ of `C` must hold for x:

$$\mathcal{K}^{\mathsf{Prg}} \models \forall x.\ \mathsf{in}(x, \mathtt{C}) \Rightarrow \phi^{\mathtt{C}}[z := x], \text{ for all } \mathtt{C} \in Cls \text{ such that } \mathsf{isAdp}(\mathtt{C}).$$

This requirement ensures that the classification-predicates in the knowledge base respect the subclass hierarchy in the program.

Coh4: In every model satisfying $\mathcal{K}^{\mathsf{Prg}}$, for every object x in an adaptable class $\mathtt{C}$, classification predicate $\phi^{\mathtt{D}}$ of one of its subclasses $\mathtt{D}$ must hold for x:

$$\mathcal{K}^{\mathsf{Prg}} \models \forall x.\ \mathsf{in}(x,\mathtt{C}) \Rightarrow \bigvee_{\mathtt{D}\in\mathrm{Children}(\mathtt{C})} \phi^{\mathtt{D}}[z:=x],\ \text{for all}\ \mathtt{C} \in \mathit{Cls}\ \text{s.t.}\ \mathsf{isAdp}(\mathtt{C}).$$

This requirement ensures that for every instance of a subclass of $\mathtt{C}$, the classification query of at least one of the subclasses of $\mathtt{C}$ always holds.

Coh5: In every model satisfying $\mathcal{K}^{\mathsf{Prg}}$, for every object x in an adaptable class $\mathtt{C}$ and subclass $\mathtt{D}$ of $\mathtt{C}$, the `retrieves` predicate of $\mathtt{D}$ must hold for x when its `classifies` predicate holds:

$$\mathcal{K}^{\mathsf{Prg}} \models \forall x.\big(\mathsf{in}(x,\mathtt{C}) \wedge \phi^{\mathtt{D}}[z:=x]\big) \Rightarrow \Big(\big(\exists \overline{y}.\psi^{\mathtt{D}}[z\overline{z}:=x\overline{y}]\big) \wedge \big(\forall \overline{y}.\psi^{\mathtt{D}}[z\overline{z}:=x\overline{y}] \Rightarrow \mathsf{hasType}(\overline{y},\overline{\mathtt{T}})\big)\Big),$$

for all $\mathtt{C},\mathtt{D} \in \mathit{Cls}$ such that $\mathsf{isAdp}(\mathtt{C})$ and $\mathsf{subclass}(\mathtt{D},\mathtt{C})$

where $\lambda z,\overline{z}.\psi^{\mathtt{D}} = \mathtt{Prg(D)(retrieves)}$, $\mathtt{fields(D)} = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ (for some $\overline{\mathtt{f}}$), and $\mathsf{hasType}(\overline{y},\overline{\mathtt{T}})$ is shorthand for $\mathsf{hasType}(y_1,\mathtt{T}_1) \wedge \cdots \wedge \mathsf{hasType}(y_n,\mathtt{T}_n)$. This requirement ensures that a reclassified object can be instantiated correctly.

Conditions ***Coh1*** and ***Coh2*** are basic requirements on the domain knowledge base $\mathcal{K}$ and the linkage axioms. They ensure that reasoning over the combined knowledge base is possible. In contrast, ***Coh3***–***Coh5*** are required for type soundness (cf. Theorem 22): first ***Coh3*** ensures that subclassing in the program and the knowledge base are aligned; then ***Coh4*** ensures that reclassification always succeeds in finding a target class $\mathtt{D}$; and finally ***Coh5*** ensures that values for the fields in the target class $\mathtt{D}$ can always be retrieved in the knowledge base and respect the declared types of the fields of class $\mathtt{D}$ in the program.

3.6 Heap Lifting, Synchronisation and Operational Semantics

To model the operational semantics of FSRJ, we introduce the concepts of *address* (of an object in a heap), *stack*, *values*, *object*, *heap*, *lifted-heap knowledge base*, *environment*, *synchronisation knowledge base*, *overall knowledge base*, *runtime expressions* and *runtime configurations*. *Addresses*, ranged over by the metavariable ι, are elements of the denumerable set I. *Stacks* are sequences $\overline{\iota}$, recording the addresses of objects on which methods under execution have been invoked (the address of the object of the most recent invocation is rightmost). *Values*, ranged over by the metavariable v, are either addresses, integers, or the special value `null`. *Objects* are tuples $\langle \mathtt{C}, \overline{\mathtt{f}} = \overline{v}\rangle$, where $\mathtt{C}$ is the class of the object, $\overline{\mathtt{f}}$ the names and $\overline{v}$ the values of the fields. A heap $\mathcal{H}$ maps addresses to objects; the empty heap is denoted by $\emptyset$. A value v occurs in $\mathcal{H}$ if $v \in \{\iota, \overline{v}\}$ for some $\mathcal{H}(\iota) = \langle \mathtt{C}, \overline{\mathtt{f}} = \overline{v}\rangle$, where $\iota \in \mathtt{dom}(\mathcal{H})$.

▶ **Definition 13** (Heaps for well-typed programs). *A heap $\mathcal{H}$ for a well-typed FSRJ program* Prg *is a mapping from addresses to objects such that*

- $\{\iota \mid \iota \text{ occurs in } \mathcal{H}\} = \mathtt{dom}(\mathcal{H})$, *and*
- $\mathcal{H}$ *only contains instances of non-adaptable classes defined in* Prg.

The following definition formalises the concept of *lifting a heap* into a knowledge base.

▶ **Definition 14** (Lifted-heap knowledge base). *Given a heap $\mathcal{H}$ for a well-typed FSRJ program* Prg*, the* lifted-heap knowledge base $\mathcal{K}^{\mathsf{heap}}(\mathcal{H})$*, which lifts $\mathcal{H}$ to FOL, is the set consisting of the following formulas over the signature $\Sigma^{\mathsf{heap}}(\mathcal{H}) = \Sigma^{\mathsf{code}} \cup \mathtt{dom}(\mathcal{H})$:*

*L*1. $\mathsf{instOf}(\iota,\mathtt{C})$ *for all* $\iota \in \mathtt{dom}(\mathcal{H})$ *such that* $\mathcal{H}(\iota) = \langle \mathtt{C}, \mathtt{f}_1 = v_1, \ldots, \mathtt{f}_n = v_n\rangle$

*L*2. $\mathsf{C_f}_j(\iota, v_j)$ *for all* $\iota \in \mathtt{dom}(\mathcal{H})$ *and for all* $\mathsf{C_f}_j$ *such that* $\mathcal{H}(\iota) = \langle \mathtt{C}, \mathtt{f}_1 = v_1, \ldots, \mathtt{f}_n = v_n\rangle$, $\mathtt{C} <: \mathtt{C}_j$, $\mathtt{C}_j.\mathtt{f}_j \in \mathit{Fls}(\mathtt{C}_j)$ *and* $1 \le j \le n$

*L*3. $\forall x.\ \big(\bigwedge_{\iota\in\mathtt{dom}(\mathcal{H})}(x \neq \iota)\big) \Rightarrow \neg\mathsf{isObj}(x)$

The following definition formalises the concept of stating the values of the current properties of the physical assets into a knowledge base.

▶ **Definition 15** (Environment and synchronisation knowledge base)**.** *An environment $\mathcal{E}$ for a well-typed FSRJ program* Prg *is a reference to a mapping that provides suitable current values for the properties of the assets described in the domain knowledge base of* Prg*. The* synchronisation knowledge base $\mathcal{K}^{\mathsf{sync}}(\mathcal{E})$ *of* Prg *provides a witness for each formula* $\exists x.\mathsf{ctx_prop}(\mathsf{ctx_asset}, x)$ *in $\mathcal{K}$ which states that the property* ctx_prop *of the asset* ctx_asset *has some value x.*

We now define a knowledge base that combines the two knowledge bases defined above, which model the dynamic part of the system (the program heap and the current values of the properties physical assets, respectively) with the domain and code knowledge base, which model the static part of the system (domain knowledge and the structure of the code, respectively). The combined knowledge base is operationalised in the operational semantics of FSRJ (see rules R-adapt-y and R-adapt-n in Figure 8).

▶ **Definition 16** (Overall knowledge base)**.** *Given a heap $\mathcal{H}$ and an environment $\mathcal{E}$ for a well-typed FSRJ program* Prg*, the* overall knowledge base $\mathcal{K}^{\mathsf{overall}}(\mathcal{H}, \mathcal{E}) = \mathcal{K}^{\mathsf{Prg}} \cup \mathcal{K}^{\mathsf{heap}}(\mathcal{H}) \cup \mathcal{K}^{\mathsf{sync}}(\mathcal{E})$ *of* Prg*, over the signature $\Sigma^{\mathsf{Prg}} \cup \Sigma^{\mathsf{heap}}(\mathcal{H})$, represents domain knowledge, code, heap and value of the asset properties for the program* Prg *with heap $\mathcal{H}$ and environment $\mathcal{E}$.*

We briefly discuss the formulas of $\mathcal{K}^{\mathsf{heap}}(\mathcal{H})$. The axiom $L1$ introduces individuals ι as instances of classes `C` (Axiom $A14$ of Definition 7 then allows us to infer `isObj(`ι`)`). Axiom $L2$ populates the relation `C_f` for all fields `f` of the classes `C`. Axiom $L3$ states that only addresses from the heap can be considered as objects in the knowledge base $\mathcal{K}^{\mathsf{overall}}(\mathcal{H}, \mathcal{E})$ (thus, $L3$ is similar to the closure axioms $A8$ and $A9$ for integers and classes in Figure 7).

▶ **Example 17** (Lifted-heap and synchronisation knowledge bases for the Greenhouse digital twin in FSRJ)**.** The lifted-heap and synchronisation knowledge bases for the FSRJ program in Figure 3a and Example 1 after the first execution of the `pu.plant=this.pl` expression in the body of the `loop` method, are given in Figures 2b and 2d, respectively.

Recall that (by Lemma 12) the program knowledge base $\mathcal{K}^{\mathsf{Prg}}$ of a well-typed program Prg is satisfiable. The following theorem shows that, if $\mathcal{K}^{\mathsf{Prg}}$ satisfies conditions ***Coh1*** and ***Coh2*** for program coherence (see Section 3.5), then its extension $\mathcal{K}^{\mathsf{overall}}(\mathcal{H}, \mathcal{E})$ with a lifted-heap knowledge base $\mathcal{K}^{\mathsf{heap}}(\mathcal{H})$ and synchronization knowledge base $\mathcal{K}^{\mathsf{sync}}(\mathcal{E})$ for Prg remains satisfiable (thus, reasoning over $\mathcal{K}^{\mathsf{overall}}(\mathcal{H}, \mathcal{E})$ is possible).

▶ **Theorem 18** (Satisfiability of the overall knowledge base)**.** *Let $\mathcal{H}$ be a heap and $\mathcal{E}$ an environment for the well-typed program* Prg *with domain knowledge base $\mathcal{K}$ is over the signature Σ, and let $\mathcal{K}^{\mathsf{Prg}} = \mathcal{K} \cup \mathcal{K}^{\mathsf{code}} \cup \mathcal{K}^{\mathsf{link}}$ be the program knowledge base of* Prg*. If $\mathcal{K}$ is satisfiable and $\mathcal{K} \cup \mathcal{K}^{\mathsf{code}} \models \mathcal{K}^{\mathsf{links}}$, then the overall knowledge base $\mathcal{K}^{\mathsf{overall}}(\mathcal{H}, \mathcal{E})$ is satisfiable.*

Runtime expressions are ranged over by e. Their syntax is obtained from the syntax of expressions (cf. Figure 5) by removing variables `x` (recall that `this` is a variable) and adding addresses ι and `ret`-expressions (of the form `ret`(e), that represent the execution of a method). A *runtime configuration* (*configuration*, for short) is a triple consisting of a heap, a stack and a runtime expression, i.e., $\mathcal{H} \mid \bar{\iota} \mid e$. The reduction relation has the form $\mathcal{H} \mid \bar{\iota} \mid e \to \mathcal{H}' \mid \bar{\iota}' \mid e'$, where the configuration $\mathcal{H} \mid \bar{\iota} \mid e$ gets reduced to $\mathcal{H}' \mid \bar{\iota}' \mid e'$ in one step. A configuration $\mathcal{H} \mid \bar{\iota} \mid e$ that cannot be reduced is called a *normal form*. The initial configuration of a program is $\emptyset \mid \bullet \mid$ `e`, where `e` is the main expression of the program. Remark that configurations only

Computation rules

$$\text{(R-ret)}\ \mathcal{H} \mid \overline{\iota}\iota \mid \texttt{ret}(v) \to \mathcal{H} \mid \overline{\iota} \mid v$$

$$\text{(R-field)}\ \frac{\texttt{f} = v \in \overline{\texttt{f}} = \overline{v} \quad \mathcal{H}(\iota) = \langle \texttt{C}, \overline{\texttt{f}} = \overline{v} \rangle}{\mathcal{H} \mid \overline{\iota} \mid \iota.\texttt{f} \to \mathcal{H} \mid \overline{\iota} \mid v}$$

$$\text{(R-new)}\ \frac{\iota \notin \texttt{dom}(\mathcal{H}) \quad \texttt{fields(C)} = \overline{\texttt{T}}\ \overline{\texttt{f}}}{\mathcal{H} \mid \overline{\iota} \mid \texttt{new C}(\overline{v}) \to \mathcal{H} \cup \left\{ \iota \mapsto \langle \texttt{C}, \overline{\texttt{f}} = \overline{v} \rangle \right\} \mid \overline{\iota} \mid \iota}$$

$$\text{(R-invoke)}\ \frac{\mathcal{H}(\iota) = \langle \texttt{C}, \ldots \rangle \quad \texttt{mBody(m,C)} = (\overline{\texttt{x}}, \texttt{e}_0)}{\mathcal{H} \mid \overline{\iota} \mid \iota.\texttt{m}(\overline{v}) \to \mathcal{H} \mid \overline{\iota}\iota \mid \texttt{ret}([\overline{\texttt{x}} \leftarrow \overline{v}, \texttt{this} \leftarrow \iota]\, \texttt{e}_0)}$$

$$\text{(R-assign)}\ \frac{\mathcal{H}(\iota) = \langle \texttt{C}, \ldots, \texttt{f}_i = v_i, \ldots \rangle}{\mathcal{H} \mid \overline{\iota} \mid \iota.\texttt{f}_i = v \to \mathcal{H}\,[\iota \mapsto \langle \texttt{C}, \ldots, \texttt{f}_i = v, \ldots \rangle] \mid \overline{\iota} \mid v}$$

$$\text{(R-adapt-y)}\ \frac{\begin{array}{c}\iota \notin \overline{\iota} \quad \texttt{D} \neq \texttt{D}' \qquad \mathcal{H}(\iota) = \langle \texttt{D}, \overline{\texttt{f}} = \overline{v} \rangle \\ \texttt{classifies}(\texttt{D}) = \lambda z.\phi \qquad \mathcal{K}^{\text{code}} \models \mathsf{compatible}(\texttt{D}, \texttt{D}') \\ \texttt{classifies}(\texttt{D}') = \lambda z.\phi' \qquad \mathcal{K}^{\text{overall}}(\mathcal{H}, \mathcal{E}) \not\models \phi[x := \iota] \\ \texttt{retrieves}(\texttt{D}') = \lambda z \overline{z}.\psi' \qquad \mathcal{K}^{\text{overall}}(\mathcal{H}, \mathcal{E}) \models \psi'[z\overline{z} := \iota\overline{v}']\end{array}}{\mathcal{H} \mid \overline{\iota} \mid \texttt{adapt}(\iota) \to \mathcal{H}[\iota \mapsto \langle \texttt{D}', \overline{\texttt{f}}' = \overline{v}' \rangle] \mid \overline{\iota} \mid \iota}$$

$$\text{(R-adapt-n)}\ \frac{\begin{array}{c}\iota \notin \overline{\iota} \\ \mathcal{H}(\iota) = \langle \texttt{D}, \overline{\texttt{f}} = \overline{v} \rangle \\ \texttt{classifies}(\texttt{D}) = \lambda z.\phi \\ \mathcal{K}^{\text{overall}}(\mathcal{H}, \mathcal{E}) \models \phi[x := \iota]\end{array}}{\mathcal{H} \mid \overline{\iota} \mid \texttt{adapt}(\iota) \to \mathcal{H} \mid \overline{\iota} \mid \iota}$$

Method body lookup

$$\frac{\texttt{aDef(C)(m)} = \texttt{T m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\ \{\texttt{return e;}\}}{\texttt{mBody(m,C)} = (\overline{\texttt{x}}, \texttt{e})}$$

Evaluation context

$$E ::= [] \mid E.\texttt{f} \mid E.\texttt{m}(\overline{e}) \mid \texttt{v.m}(\overline{v}, E.\overline{e}) \mid \texttt{new C}(\overline{v}, E, \overline{e}) \mid \texttt{v.f} = E \mid \texttt{ret}(E) \mid \texttt{adapt}(E)$$

Congruence rule

$$\frac{\mathcal{H} \mid \overline{\iota} \mid e \to \mathcal{H}' \mid \overline{\iota}' \mid e'}{\mathcal{H} \mid \overline{\iota} \mid E[e] \to \mathcal{H}' \mid \overline{\iota}' \mid E[e']}$$

Figure 8 FSRJ operational semantics: computation rules, method body lookup, evaluation context, and congruence rule.

include stacks to ensure that the operational semantics can model that execution gets stuck when the reclassification of an object on which a method under execution has been invoked is attempted (i.e., that a configuration that is going to perform such a reclassification is a normal form).

The reduction system for FSRJ is given in Figure 8. The reduction rules use the auxiliary function `mBody(m,C)` (for method body lookup) and the *congruence rule* (which uses evaluation contexts) are also given in Figure 8. The rules R-field, R-new and R-assign for field access, object creation and assignment are standard. Rule R-invoke pushes the address of the called object to the stack and a `ret`-expression that stems from method lookup for the called method in the class of the called object, as the new runtime expression. Here, actual parameters and the self-reference are inlined by substitution on the runtime expression. Rule R-ret then pops the object address from the stack, returning the computed value as the new runtime expression.

In FSRJ an object is hot (see Section 2) if and only if its address occurs on the stack. Both rules for dynamic object reclassification, R-adapt-y and R-adapt-n, check whether an object needs to be reclassified and cause the execution to get stuck if an attempt to reclassify a hot object is made. Rule R-adapt-y applies if the object needs to be reclassified (i.e., it cannot be classified as a member of its current class); rule R-adapt-n applies otherwise. We now explain the interaction between execution and the knowledge base in these rules. Both rules check whether the object ι of class `D` is cold and, if so, lift the heap $\mathcal{H}$ to construct the knowledge base $\mathcal{K}^{\text{overall}}(\mathcal{H}, \mathcal{E})$. If the classification predicate ϕ of `D` holds when instantiated with ι, then rule R-adapt-n is applied and the object is not reclassified. Otherwise rule R-adapt-y is applied. Recall from Axiom $A13$ of Definition 7 that two classes are compatible if they have the same superclass. Thus, the premises of rule R-adapt-y identify a class $\texttt{D}'$ that is compatible with `D`, for which the classification predicate ϕ' holds when instantiated with ι. The last premise ensures that the **`retrieves`**-predicate ψ' of $\texttt{D}'$ holds for values $\iota, \overline{v}'$, and the resulting heap on the right-hand side binds ι to an object of class $\texttt{D}'$ in which the fields $\overline{\texttt{f}}$ are bound to $\overline{v}'$.

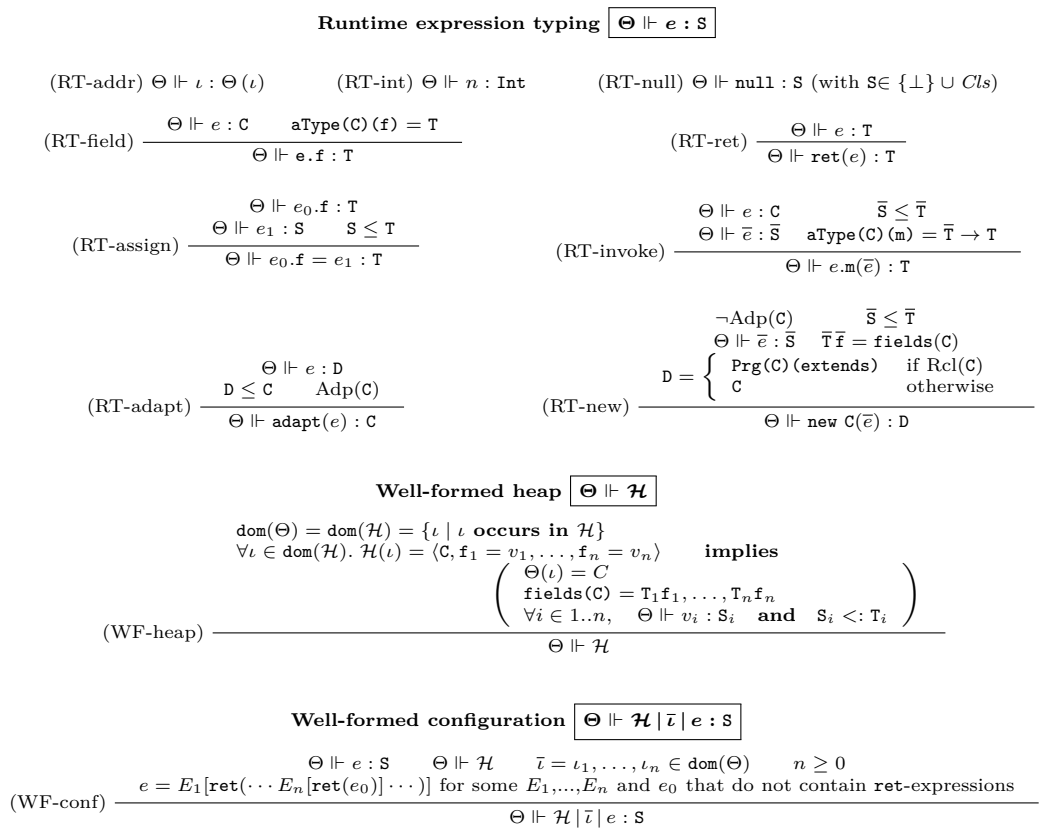


Figure 9 FSRJ run-time typing: runtime expressions, well-formed heaps and configurations.

3.7 Type Soundness for Coherent Programs

In order to prove type soundness for coherent FSRJ programs by a subject reduction theorem and a progress theorem for the small-step operational semantics, we need to formulate a type system for runtime expressions. Expressions containing either a *stupid selection*, i.e., a field selection `null.f` or a method invocation `null.m(...)`, or a *stupid reclassification*, i.e., an object reclassification `adapt(null)`, are not well-typed according to the FSRJ (source level) type system. However, a runtime expression without stupid selections and stupid reclassifications may reduce to a runtime expression containing stupid selections or stupid reclassifications. Therefore, the type system for runtime expressions contains a rule for assigning to the value `null` either the type $\bot$ (like in the source level typing) or any class name.

The type rules for runtime expressions are shown in Figure 9 (left); these rules are of the form $\Theta \Vdash e : \mathtt{S}$, where the environment Θ is a finite (possibly empty) mapping from addresses to class names. Figure 9 (bottom) also presents the notions of *well-formed-heaps* and *well-formed configurations*. The notion of well-formed heap ensures that the environment Θ maps all the addresses in the heap into the class of the corresponding object and that, for every object stored in the heap, the fields of the object contain appropriate values. The notion of well-formed configuration ensures that the heap is well-formed, that the addresses in the stack are defined in the heap, that the runtime expression is well-typed and has a structure that is compatible with the stack – this last check is performed by exploiting evaluation contexts for FSRJ runtime expressions (see Figure 8).

Then, type soundness can be proved by using the standard technique of subject reduction and progress theorems [50]; however, we need to account for declarative object reclassification.

▶ **Lemma 19** (From expression typing to runtime expression typing). *If* $\bullet \vdash e : T$*, then* $\bullet \Vdash e : T$*.*

We now consider subject reduction for the execution of semantically reflected programs. Here, program coherence (Section 3.5) and Theorem 18 (which assumes ***Coh1*** and ***Coh2*** and is independent from ***Coh3***–***Coh5***) are crucial for the case of R-adapt-y:

- an inconsistent (i.e., not satisfiable) knowledge base, or
- a consistent knowledge base that satisfies ***Coh1***–***Coh4*** and does not satisfy ***Coh5*** because the retrieved values for the fields have a wrong type

would allow us to, e.g., infer erroneous values for `retrieves` λ-predicates (see the premise $\mathcal{K}^{\mathsf{overall}}(\mathcal{H}, \mathcal{E}) \models \psi'[z\overline{z} := \iota\overline{v}']$ of rule R-adapt-y), potentially leading to typing errors for runtime configurations after reclassification. Lifting a heap that is not well-typed, could make the knowledge base inconsistent (see Condition L2 in Definition 14 and Axiom A21 in Figure 7). Thus, well-typed runtime configurations ensure consistent overall knowledge bases, which in turn (together with ***Coh5***) ensure that (whenever rule R-adapt-y does not get stuck) dynamic object reclassification produces well-typed runtime configurations.

▶ **Theorem 20** (Subject reduction). *If* $\Theta \Vdash \mathcal{H} \,|\, \overline{\iota} \,|\, e : S$ *and* $\mathcal{H} \,|\, \overline{\iota} \,|\, e \to \mathcal{H}' \,|\, \overline{\iota}' \,|\, e'$ *then there exists* $\Theta' \supseteq \Theta$ *such that* $\Theta' \Vdash \mathcal{H}' \,|\, \overline{\iota}' \,|\, e' : S'$ *for some* S' *such that* $S' \leq S$*.*

Progress relies on program coherence, but not on consistency of the knowledge base. Concretely, ***Coh3***–***Coh5*** guarantee that there exists a class $\mathtt{D}'$ such that the following three premises of rule R-adapt-y hold:

$$\mathcal{K}^{\mathsf{code}} \models \mathsf{compatible}(\mathtt{D}, \mathtt{D}') \qquad \mathcal{K}^{\mathsf{overall}}(\mathcal{H}, \mathcal{E}) \models \phi'[x := \iota] \qquad \mathcal{K}^{\mathsf{overall}}\mathcal{H}, \mathcal{E}) \models \psi'[z\overline{z} := \iota\overline{v}']$$

The last case in the statement of the theorem states that the adaptation of hot objects gets stuck (cf. ***DC3*** in Section 4).

▶ **Theorem 21** (Progress). *Let* $\mathcal{H} \,|\, \overline{\iota} \,|\, e$ *be a well-typed normal form. Then*

1. *either* e *is a value and* $\overline{\iota} = \bullet$*; or*
2. *for some evaluation context* E *we can express* e *as*
   a. *either* $E\,[\mathtt{null.f}]$ *for some* $\mathtt{f}$*, or*
   b. $E\,[\mathtt{null.m}(\overline{v})]$ *for some* $\mathtt{m}$ *and* $\overline{v}$*, or*
   c. $E\,[\mathtt{null.f} = v]$ *for some* $\mathtt{f}$ *and* v*, or*
   d. $E\,[\mathtt{adapt}(\mathtt{null})]$*, or*
   e. $E\,[\mathtt{adapt}(\iota)]$ *for some* $\iota \in \overline{\iota}$*.*

Type soundness follows from Lemma 19 and Theorems 20 and 21. After Theorem 20, this theorem is a second place where consistency (needed only for Theorem 20) and program coherence (needed for Theorem 21) interact: We need the overall knowledge base $\mathcal{K}^{\mathsf{overall}}(\mathcal{H}, \mathcal{E})$ to be consistent so that ***Coh3***–***Coh5*** indeed select the correct class $\mathtt{D}'$ to reclassify.

▶ **Theorem 22** (Type soundness). *Let* $\mathtt{Prg} = \mathcal{K}\,\overline{CD}\,\mathtt{e}_0$ *be well-typed and coherent FSRJ program such that* $\bullet \vdash \mathtt{e}_0 : T_0$*, and* $\emptyset \,|\, \bullet \,|\, e_0 \to^* \mathcal{H} \,|\, \overline{\iota} \,|\, e$ *with* $\mathcal{H} \,|\, \overline{\iota} \,|\, e$ *being a normal form. Then* e *is*

1. *either an integer* n *and* T_0 *is* `int` *and* $\overline{\iota} = \bullet$*; or*
2. $\mathtt{null}$ *and* T_0 *is either* $\bot$ *or a class name and* $\overline{\iota} = \bullet$*; or*
3. *an address* ι *such that* $\mathcal{H}(\iota) = \langle C, \cdots \rangle$ *with* $C \leq T_0$ *and* $\overline{\iota} = \bullet$*; or*
4. *an expression containing either* $\mathtt{null.f}$ *or* $\mathtt{null.m}(\overline{v})$ *or or* $\mathtt{null.f} = v$ *or* $\mathtt{adapt}(\mathtt{null})$ *or* $\mathtt{adapt}(\iota)$ *for some* $\mathtt{f}, \mathtt{m}$*,* $\overline{v}$*,* v *and* ι *such that* $\iota \in \overline{\iota}$*.*

4 Discussion: Design Choices, Implementation and Performance

This section discusses design choices for declarative dynamic object reclassification. We first consider overall design choices for the declarative reclassification mechanism (as reflected in FSRJ), then design choices related to the implementation of declarative dynamic object reclassification (as reflected in our prototype implementation), and finally the implementation itself.

4.1 Overall Design Choices

Our main goal with declarative dynamic object reclassification is to realise a separation of concerns and decoupling between the *application logic*, which describes program behaviour within a given context, and the *adaptation logic*, which describes how the behaviour of the program changes according to context changes. To focus on the basic interaction between adaptation and application logic, we aimed for a simple programming construct for dynamic object reclassification and considered the following design choices:

DC1: The adaptation logic is expressed in a declarative way, leveraging domain knowledge.
DC2: The application logic is expressed by standard class-based object-oriented code.
DC3: Adaptation works on (cold) objects in isolation and hot object adaptation gets stuck.

Concerning ***DC1***, to focus on the general mechanisms of semantic reflection and declarative dynamic object reclassification, FSRJ formalises the knowledge base as a set of FOL formulas. Observe that FSRJ does not impose any restrictions on queries to the knowledge base. For example, `classifies`-predicates may depend on the class of other objects; thus, the reclassification of one object may lead to another object becoming inconsistent. We believe that this flexibility is convenient in a digital twin setting; e.g., a change of requirement monitor for a plant may trigger a change in the controller for the water pump. An alternative design choice would be to restrict queries to, e.g., only depend on the external context (i.e., the domain and synchronisation knowledge). In this case, we avoid the reclassification of one object triggering the reclassification of another, but we could potentially reclassify one object too early, violating the program logic (e.g., changing the watering policy before changing the humidity monitor in the greenhouse). Furthermore, self-adaptation in FSRJ is not deterministic: an object may be reclassifiable to more than one class. If determinism is desired, one can add the additional constraint that the classification queries of its reclassifiable subclasses of all adaptable classes are pairwise disjoint.

Concerning ***DC2***, it can be worth observing that (1) the `links`, `classifies` and `retrieves` clauses could be expressed as, e.g., program annotations or even in a separate file; (2) typing just enforces a programming pattern that treats an adaptable class as an abstract class and its reclassifiable subclasses as final classes that cannot be used as types; and (3) the `adapt` expression could be implemented as a library function.

Concerning ***DC3***, there are several ways to relax this restriction. First, the reclassification of a hot object could be delayed until the object becomes cold (this raises the issue of whether the ordering between delayed adaptations of different objects should be preserved). Second, the `adapt` expression could be replaced or complemented by a class-level adaptation expression to trigger the adaptation of all objects of a class. Third, the `adapt` expression could be externalised; e.g., replaced or complemented by an adaptation triggered by the runtime system, which performs adaptation on all objects of adaptable classes, and delays adaptation for hot objects until they become cold. Relaxing ***DC3*** suggests several interesting directions for further work, including

- a type and effect system [3,57] to ensure that hot object adaptation cannot occur;
- coordinated adaptation of object aggregates (i.e., groups of interconnected objects);
- reclassifiable classes extended by other reclassifiable classes (cf. [15,20]); and
- adaption of hot objects, e.g., the expression `adapt(this)` (cf. [15,20]) or a method defined in a class `C` with an expression `adapt(e)` where `e` has type `C` (cf. [16]).

For the last point, the adaptation logic could depend on the values of fields in the target object; when the object is cold, reclassification might rely on these values satisfying some class invariant (following a program verification methodology, e.g., [1,48]). So, hot object adaptation should require to devise a suitable invariant that the object must satisfy when, e.g., a specific occurrence of the expression `adapt(this)` is executed. For this reason, we decided that even rule R-adapt-n gets stuck on hot objects.

4.2 Implementation and Performance

Implementation. To interact with context in an implementation, we need to be able to check entailment and satisfiability of first-order formulas, and to retrieve witnesses when evaluating the satisfiability of λ-predicates (for `retrieves`-queries). Whereas such functionality is available in, e.g., satisfiability modulo theory solvers, the querying mechanism for declarative dynamic object reclassification would additionally benefit from decidability and availability of formalised domain knowledge. Hence, we consider two possibilities: ontology-based knowledge graphs and logic programming for ***DC2***.

Knowledge Graphs are widely used triple-based data formalisms with numerous variants, of which the RDF/OWL stack is well-suited for reclassification. First, these knowledge graphs are used in numerous industrial ontologies and is an established, core technology for autonomous systems, e.g., in digital twins or robotics [7, 42]. Thus, they enable reuse and provide general knowledge across industries, especially in engineering. Second, these knowledge graphs have a foundation in Description Logics [5], which are decidable fragments of FOL with efficient reasoners and advanced tool-supported pragmatics for non-experts [28].

Logic Programming is widely used and perhaps more rooted in programming and artificial intelligence than knowledge graphs, and also used for autonomous systems [7,32]; numerous implementations are available for knowledge representation and reasoning, e.g., Golog [45]. Logic programming has several variants, such as Prolog-style languages or Answer Set Programming [10,46], which easily produce the witnesses we need for our queries. Logic programming is based on rules that encode domain knowledge, and powerful reasoning engines to achieve high efficiency. However, they are less prevalent for ontologies (i.e., general knowledge with few or no individuals), and seem better suited for concrete planning tasks.

We aim to leverage domain knowledge for self-adaptation in autonomous systems, e.g., digital twins. Observe that FSRJ's knowledge base and queries fit into Description Logic (specifically, the logic $\mathcal{SROIQ}$ underlying OWL 2 [30]). In particular, program coherence (Section 3.5) here maps to *decidable reasoning operations*: ***Coh1*** expresses satisfiability of a $\mathcal{SROIQ}$ knowledge base, ***Coh2*** the entailment of two sets of $\mathcal{SROIQ}$ axioms, and ***Coh3***–***Coh5*** the entailment of a subconcept relation from a set of axioms. In Description Logic syntax, we have $\exists \mathsf{in}.\{\mathtt{C}\} \sqsubseteq \mathsf{cl}_{\mathtt{C}}$ (***Coh3***) and $\exists \mathsf{in}.\{\mathtt{C}\} \sqcap \mathsf{cl}_{\mathtt{C}} \sqsubseteq \bigsqcup_{\mathtt{D}} \mathsf{cl}_{\mathtt{D}}$ (***Coh4***), where $\mathsf{cl}_{\mathtt{C}}$ is the classification concept of class `C`. ***Coh5*** is a special case of type checking knowledge graph access under semantic reflection [39]. Hence, our implementation uses knowledge graphs that connect to existing ontologies.

Further, an implementation of declarative dynamic object reclassification needs to realise semantic lifting to integrate program and knowledge base, and dynamic reclassification via `adapt`-expressions. For these reasons, our implementation is based on SMOL [38], an interpreted research language used to explore connections between programming languages and knowledge graphs, that natively supports heap lifting into a knowledge base. The language is restricted to Description Logics and OWL-based knowledge graphs. The backend uses HermiT [24] for reasoning and Apache Jena (`https://jena.apache.org/`) for graph data management. SMOL was designed to support semantic reflection for the implementation of digital twins [40], but has not yet offered direct programmatic support for reclassification as proposed in this paper.

Our implementation, extending SMOL, goes slightly beyond the design decisions of FSRJ concerning minimality. The main differences are: (1) an explicit constructor that is called after reclassification to establish the classification predicate of the new class; (2) `links`, `classifies` and `retrieves` clauses are inherited; and (3) a more elaborate lifting mechanism, which can be encoded in the one of FSRJ. The constructor ensures a form of reclassification stability, while inheritance of classification predicates ensures a form of behavioural subtyping corresponding to ***Coh3*** (see Section 3.5). Our implementation supports both the `adapt`-expression of FSRJ and an option for reclassification to be triggered periodically by the runtime system. For static program analysis, we exploit the decidability of description logics underlying knowledge graphs to ensure program coherence. Figure 10 gives the SMOL version of our running example, as well as the knowledge graph used as the knowledge base.

Performance. The most expensive operations of FSRJ are on the lifted heap. Experimental results [35] that compare querying a description logic knowledge base with lifted heaps to querying a representation of the same knowledge expressed in program code, show that reasoning in the description logic knowledge base scales significantly better than reasoning with the knowledge represented in the program, both for more complex knowledge bases (i.e., more possible classification targets) and for more elements (i.e., more constants or individuals in the knowledge graph). The direct representation in the program only performs better for simple (<10 classification targets) and small (<2000 individuals) knowledge graphs. Further, remark that heap lifting in SMOL does not generate a full knowledge base every time the lifted heap is accessed. Instead it uses *virtualisation* [59]: the implementation determines which parts of the knowledge base are relevant for a query, and only generates this part.

5 Evaluation of the Prototype Implementation

To evaluate the prototype implementation of declarative dynamic object reclassification, we aim to demonstrate that the reclassification works in practice as expected from the calculus. Concretely, we consider the following research questions:

RQ1: Does derivation in the knowledge base identify the correct class for reclassification?
RQ2: Does derivation in the knowledge base instantiate state correctly during reclassification?

These RQs address essential aspects of the declarative reclassification process, namely that the `classifies`-and `retrieves`-predicates are resolved correctly in the knowledge base at runtime.

Experimental Design and Setup. This evaluation is based on GreenhouseDT [41], an open-source digital twin exemplar of a mini greenhouse developed in SMOL as a sensor-driven system that controls the temperature and humidity on different shelves of a greenhouse, as

OWL

```
ctx:NormalOps EquivalentTo: ctx:Pump and (ctx:temp some xsd:double[<= 50.0])
ctx:Overheating EquivalentTo: ctx:Pump and (ctx:temp some xsd:double[> 50.0])
FunctionalDataProperty(ctx:id)
```

(a) General knowledge as OWL ontology.

SMOL

```
1 abstract class Pump (domain String id, Int GpioPin, Plant plant)
2   links "a ctx:Pump";
3 Unit pump() /* ... */ end end
4 class Normal extends Pump () classifies "<ctx:NormalOps>";
5   /* methods */ end
6 class Overheating extends Pump (Int maximal) classifies "<ctx:Overheating>";
7   retrieves "?this ctx:hasMaintananceProfile [ctx:hasMaximalPower ?maximal]";
8 /* methods */ end
```

(b) SMOL code.

RDF

```
1 ctx:pump1 a ctx:Pump;  ctx:id "2"; ctx:temp 52.5.  run:ob1 a prog:Seedling.
2 run:ob2 a prog:NormalPump; prog:id "2"; prog:GpioPin "7", prog:plant run:ob1.
3 run:ob2 smol:links [a ctx:Pump; ctx:id "1"].
```

(c) Lifting in RDF.

Figure 10 Instantiation of the pump in the Greenhouse digital twin in the SMOL implementation.

well as the soil moisture level of the plants on these shelves. The digital twin uses these sensor readings to determine the control policy of pumps (the actuators of the digital twin) for watering purposes. In our experiments, we focus on the behaviour of the actuators, i.e., the pumps used to water the plants, and use declarative object reclassification to dynamically determine which controller implementation to use for the pumps and to adopt the controller objects accordingly.

We extend GreenhouseDT to capture the following *condition-based maintenance scenario*: to ensure that pumps work properly:

1. the pumps need to operate within a given temperature range, and
2. the pumps should not exceed their expected life time in hours of operation.

If the temperature in an old pump exceeds the temperature thresholds, the pumps could get damaged. For that reason, we implement an extension of the motivating example discussed in Section 2, in which pumps can have one of the following distinct modes of behaviour: normal *operating* mode for pumps that work normally, *overheating* mode for pumps whose temperature is too high, *underheating* mode for pumps whose temperature is too low, and *maintenance* mode for pumps whose temperature is outside the operating range and that have exceeded the expected life span (and therefore need to be stopped). We model the different behavioural modes as reclassifiable subclasses of an adaptable class `Pump`. The exact values for the temperature and age thresholds depend on the type of pump considered; thus, the exact reclassification parameters differ, depending on the specifications of the different pumps. These extensions to the example are realistic as different actuators typically come with different specifications. The extensions also add complexity to the adaptation logic; it is easy to see that as this complexity increases, then a clear separation of concerns between the adaptation logic and the program logic also becomes increasingly attractive, as promoted in the approach of this paper.

Table 1 Pump states based on input temperature and reclassification cycles.

Pump ID	Model	Time (H)	Temperature	Expected Class	Actual Class
After Instantiation					
Pump 1	R385	0	5	Operational	Operational
Pump 2	WPS27	30	80	Overheating	Overheating
Pump 3	R365	3000	94	Maintenance	Maintenance
After First Reclassification					
Pump 1	R385	0	−5	Underheating	Underheating
Pump 2	WPS27	30	41	Overheating	Overheating
Pump 3	R365	2000	50	Operational	Operational
After Second Reclassification					
Pump 1	R385	2500	−5	Maintenance	Maintenance
Pump 2	WPS27	30000	41	Maintenance	Maintenance
Pump 3	R365	300	1	Underheating	Underheating

Our evaluation is performed in terms of two sets of experiments: in the first set, we provide different temperature measurements to evaluate how the system adapts pump controller objects in response to changes in the temperature. We considered scenarios with three pumps with different specifications, and three temperature measurements for each pump: one below the specified temperature range, one above the range and one within the range. In the second set, we additionally let the age of the pumps change to capture a more complex reclassification scenario. In this set, we expanded the previous scenarios with additional age update events that enable the reclassification. In the experiments, we opted to keep the knowledge base invariant; i.e., the actual components that constitute the greenhouse remain the same over time in the domain knowledge base, and only the behaviour of the pump controllers in the virtual layer of the digital twin will change and will require dynamic object reclassification in the digital twin. For ***RQ1***, we then inspected the class targeted by the reclassification in each experiment and, for ***RQ2***, the resulting object state.

Results. Table 1 illustrates the results from the experiments, comparing the expected class of each pump with the actual class after running `adapt`-expressions on objects with different pump specifications in the domain knowledge base in different scenarios. The actual class of each pump in the experiments is retrieved via the API, whereas the expected class of the pumps in the system is determined from the specifications of the different pumps, combined from constraints for the adaptation logic as expressed in the domain knowledge base, and the actual state of the objects in the system after the reclassification process.

To answer ***RQ1***, our experiments used different inputs to adapt the `Pump` objects in the model. When calling `adapt` on an object, the system changes the class of the object depending on the input event, the object state and the domain knowledge base. We checked that the reclassification process worked as formalised in FSRJ by comparing the returned class to the expected ones after running the `adapt` on the `Pump` objects. We ran the two sets of experiments described above on our prototype implementation, in all cases the dynamic object reclassification determined by the `classifies`-predicate selected the correct reclassifiable subclass of the `Pump` class.

To answer ***RQ2***, we ran the same sets of experiments as for ***RQ1***, and inspected the state of the `Pump` objects after the reclassification process, comparing the values of the fields to the expected ones. In all the conducted experiments, the actual state of the `Pump` objects was

consistent with the expected state, demonstrating that the system was able to derive correct values for the `retrieves`-predicates from the domain knowledge base at runtime. The second set of experiments showed that when considering more complex reclassification constraints, our prototype implementation was able to find a correct state.

Threats to Validity. The evaluation of the prototype implementation of declarative dynamic object reclassification has been done on a specific system, GreenhouseDT. Although the adaptation logic involves multiple classes in the knowledge base, the solutions to knowledge base queries have not been very complex. In particular, queries have deterministic solutions in our evaluation setup. We do not foresee particular challenges with non-deterministic queries, as any solution would then be correct (although not necessarily optimal). Furthermore, although the queries in our evaluation involve multiple constraints, they do not involve complex constraint solving (i.e., constraints with many variables). Although well-suited for knowledge representation, knowledge graphs are not particularly strong at complex constraint solving. As our prototype implementation relies completely on the decidable fragments of FOL available in knowledge graphs to answer the queries, the prototype implementation could potentially fail to address ***RQ2*** in cases where solutions might exist. Remark that we do not see this limitation of the prototype as a limitation of the proposed approach, as constraint solvers such as SMT could be integrated in a more refined query-answering mechanism.

6 Related Work

We focus this discussion of related research on two complementary aspects of our work: dynamic object reclassification techniques and programming with external domain knowledge. To the best of our knowledge, the combination of these aspects is novel to our paper.

There is a body of work on dynamic reclassification and related programming mechanisms (e.g., [2, 8, 13–16, 20, 26]), that addresses "self-extension" [13] and extensions of Java and use of the JVM [8, 14, 16, 26], to the design of novel programming languages [2, 20]. This line of work shares the following limitation: the requirements for the reclassification process are handled internally, as part of the program. In contrast, our paper proposes a novel approach to dynamic reclassification, in which we shift the reclassification process from ensuring internal consistency, with an operational description, to external consistency with a declarative description. This way, our work enables a separation of concerns between the adaptation logic and the application logic of the program, that has not been considered in previous work. In contrast to the works mentioned above, we have considered a very basic programming construct for dynamic object reclassification that does not require an advanced type (and effect) system.

Dynamic Software Updating (DSU) [27, 49, 51, 55] addresses runtime changes to the program code, such as software patching and task updates in real-time. DSU leverages techniques such as aspect-oriented programming [52], or changes Java program behaviour [49, 51] at the JVM or Bytecode level by modifying the code base or the instantiated objects, without halting execution. As code elements change, ensuring consistency becomes a major concern: new objects must allow the system to remain consistent with the specification. This can be solved by modifying old objects or by enforcing type renaming [27]. For DSU on asynchronous systems, statically collected type constraints can be enforced at runtime, delaying upgrades until they are type-safe [33]. Compared to reclassification mechanisms, including our work, DSU addresses externally triggered runtime changes to the program

code, while dynamic reclassification programmatically changes the class of objects based on changes in the heap or context, but does not modify the class table. Although software updates are triggered externally, DSU does not consider external consistency as done in our work.

Programming with external domain knowledge is common for self-adaptive systems [11, 44, 58] in, e.g., robotics [7, 9] and digital twins [17], but these systemic approaches generally lack the guarantees that can be provided by language abstractions. Semantic reflection, originally introduced in SMOL [38], has been used for many purposes in digital twins [40], including to repair external consistency by means of self-adaptive techniques [35–37, 40, 41], but so far the adaptation logic had to be programmed by hand. Semantic reflection has also been applied in digital twin architectures [23] to detect internal inconsistencies that trigger reconfigurations, and to drive domain-aware simulations [53]. Whereas this line of work exploits the lifting of runtime states, the self-adaptation has been concerned with the composition of components and not with changing their internal behaviour. In all cases, the adaptation logic was implemented manually without language or type support. Our work overcomes this limitation by means of linguistic support for declarative reclassification.

The adaptation and retrieval queries of our paper can be seen as a form of LINQ [47], used for reflection with knowledge bases for the underlying serialisation. We are not aware of previous applications of such techniques for reclassification. Golog [45] uses FOL to examine and pick elements from its own state and allows decidable model checking when restricted to description logics [6], but has no type or class system. The systems of Fagin et al. [18] and Calvanese et al. [12] include knowledge bases that can be manipulated and updated by the program through epistemic operators, but have no type or class system either. Here, restrictions to description logics gives tractable knowledge base revision [21, 22, 43]. We are not aware of other work on core calculi for semantically reflected programs, as studied in our paper.

7 Conclusion and Future Work

This paper presents declarative dynamic object reclassification and its formalisation in Featherweight Semantically Reflected Java, a minimal core calculus that formalises declarative reclassification in terms of interaction between a program's runtime configurations, domain knowledge and environment. We further show how to implement declarative dynamic object reclassification, and how to use these features in practice for adaptable digital twins. A key advantage of this approach is that developers can specify the reclassification process in a declarative way, disentangled from the business code, using queries to determine when and how an object should be reclassified. This makes it easier to reason about the reclassification process, and to ensure that the system remains consistent with an external context. Unlike previous approaches, declarative reclassification is based on external consistency; i.e., reclassification is driven by external domain knowledge, rather than by the internal state of the system. This makes our approach better suited for applications that must adapt to changes in the environment; it further provides a separation of concerns between the program's business code and its adaptation logic.

Some limitations and possible lines for future work are mentioned in Section 4. This paper does not address the issue of ensuring that the reclassification process is triggered at the right time; instead, reclassification is explicitly invoked. An implicit trigger might extend the range of applications for declarative reclassification; e.g., reclassification could be automatically triggered when certain conditions are met, at fixed time intervals, or during the lifting process.

Such systems would be more dynamic and responsive to changes in the environment. When doing so, it would be crucial to ensure that the system is "free" to act; e.g., the target objects are cold or wait until reclassification is safe. Such constraints point towards DSU and asynchronous update mechanisms; in fact, integrating such mechanisms with a declarative adaptation logic would be an interesting extension of our work. Another interesting line of work is to consider more flexible class hierarchies, e.g., by allowing adaptable subclasses or reclassifiable subclasses of reclassifiable classes, as well as static restrictions to ensure progress, such as type and effect systems to eliminate reclassification attempts on hot objects. Finally, this paper focused on adaptation of cold objects in isolation. In future work we would like to generalise declarative reclassification to reclassification of hot objects and to synchronised reclassification of multiple objects (e.g., all instances of a class or object aggregates) tagged for reclassification, while maintaining the separation of concerns and type soundness of the language.

References

1 Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016. doi:10.1007/978-3-319-49812-6.

2 Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proc. 24th Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA 2009)*, pages 1015–1022. ACM, 2009. doi:10.1145/1639950.1640073.

3 Torben Amtoft, Hanne Riis Nielson, and Flemming Nielson. *Type and effect systems - behaviours for concurrency*. Imperial College Press, 1999.

4 Peter B. Andrews. *An introduction to mathematical logic and type theory: to truth through proof*, volume 27 of *Applied Logic Series*. Kluwer Academic Publishers, 2 edition, 2002. doi:10.1007/978-94-015-9934-4.

5 Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003. doi:10.1017/CBO9780511711787.

6 Franz Baader and Benjamin Zarrieß. Verification of Golog programs over description logic actions. In Pascal Fontaine, Christophe Ringeissen, and Renate A. Schmidt, editors, *Proc. 9th International Symposium on Frontiers of Combining Systems (FroCoS 2013)*, volume 8152 of *Lecture Notes in Computer Science*, pages 181–196. Springer, 2013. doi:10.1007/978-3-642-40885-4_12.

7 Michael Beetz, Daniel Beßler, Andrei Haidu, Mihai Pomarlan, Asil Kaan Bozcuoglu, and Georg Bartels. Know Rob 2.0 - A 2nd generation knowledge processing framework for cognition-enabled robotic agents. In *Proc. International Conference on Robotics and Automation (ICRA 2018)*, pages 512–519. IEEE, 2018. doi:10.1109/ICRA.2018.8460964.

8 Lorenzo Bettini, Sara Capecchi, and Betti Venneri. Extending Java to dynamic object behaviors. In Viviana Bono and Michele Bugliesi, editors, *Proc. Workshop on Object Oriented Developments (WOOD 2003)*, volume 82 (8) of *Electronic Notes in Theoretical Computer Science*, pages 33–52. Elsevier, 2003. doi:10.1016/S1571-0661(04)80801-6.

9 Darko Bozhinoski, Mario Garzon Oviedo, Nadia Hammoudeh Garcia, Harshavardhan Deshpande, Gijs van der Hoorn, Jon Tjerngren, Andrzej Wąsowski, and Carlos Hernández Corbato. MROS: runtime adaptation for robot control architectures. *Advanced Robotics*, 36(11):502–518, 2022. doi:10.1080/01691864.2022.2039761.

10 Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011. doi:10.1145/2043174.2043195.

11 Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger M. Kienle, Marin Litoiu, Hausi A. Müller, Mauro Pezzè, and Mary Shaw. Engineering self-adaptive systems through feedback loops. In Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 48–70. Springer, 2009. doi:10.1007/978-3-642-02161-9_3.

12 Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Actions and programs over description logic knowledge bases: A functional approach. In Gerhard Lakemeyer and Sheila A. McIlraith, editors, *Knowing, Reasoning, and Acting: Essays in Honour of Hector J. Levesque*. College Press, 2011.

13 Alberto Ciaffaglione, Pietro Di Gianantonio, Furio Honsell, and Luigi Liquori. A protoype-based approach to object evolution. *J. Object Technol.*, 20(2):4:1–24, 2021. doi:10.5381/JOT.2021.20.2.A4.

14 Tal Cohen and Joseph Gil. Three approaches to object evolution. In Ben Stephenson and Christian W. Probst, editors, *Proc. 7th International Conference on Principles and Practice of Programming in Java (PPPJ 2009)*, pages 57–66. ACM, 2009. doi:10.1145/1596655.1596665.

15 Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle : Dynamic object re-classification. In *Proc. 15th European Conference, on Object-Oriented Programming (ECOOP 2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 130–149. Springer, 2001. doi:10.1007/3-540-45337-7_8.

16 Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. More dynamic object reclassification: Fickle$_{||}$. *ACM Trans. Program. Lang. Syst.*, 24(2):153–191, 2002. doi:10.1145/514952.514955.

17 Romina Eramo, Francis Bordeleau, Benoit Combemale, Mark van Den Brand, Manuel Wimmer, and Andreas Wortmann. Conceptualizing digital twins. *IEEE Software*, 39(2):39–46, 2021. doi:10.1109/MS.2021.3130755.

18 Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. Knowledge-based programs. *Distributed Comput.*, 10(4):199–225, 1997. doi:10.1007/S004460050038.

19 Kevin Feichtinger, Karl Kegel, Romain Pascual, Uwe Aßmann, Bernhard Beckert, and Ralf H. Reussner. Towards formalizing and relating different notions of consistency in cyber-physical systems engineering. In *Proc. 27th International Conference on Model Driven Engineering Languages and Systems, MODELS Companion 2024*. ACM, 2024. doi:10.1145/3652620.3688565.

20 Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *ACM Trans. Program. Lang. Syst.*, 36(4):12:1–12:44, 2014. doi:10.1145/2629609.

21 Giuseppe De Giacomo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. On the update of description logic ontologies at the instance level. In *Proc. 21st Conference on Artificial Intelligence (AAAI 2006)*, pages 1271–1276. AAAI Press, 2006. URL: http://www.aaai.org/Library/AAAI/2006/aaai06-199.php.

22 Giuseppe De Giacomo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. On the approximation of instance level update and erasure in description logics. In *Proc. 22nd Conference on Artificial Intelligence (AAAI 2007)*, pages 403–408. AAAI Press, 2007. URL: http://www.aaai.org/Library/AAAI/2007/aaai07-063.php.

23 Santiago Gil, Eduard Kamburjan, Prasad Talasila, and Peter Gorm Larsen. An architecture for coupled digital twins with semantic lifting. *Software and Systems Modeling*, November 2024. doi:10.1007/s10270-024-01221-w.

24 Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. Hermit: An OWL 2 reasoner. *J. Autom. Reason.*, 53(3):245–269, 2014. doi:10.1007/S10817-014-9305-1.

25 Barbara Hayes-Roth. A blackboard architecture for control. *Artificial intelligence*, 26(3):251–321, 1985. doi:10.1016/0004-3702(85)90063-3.

26 Chengwan He, Zhijie Nie, Bifeng Li, Lianlian Cao, and Keqing He. Rava: Designing a Java extension with dynamic object roles. In *Proc. International Symposium and Workshop on Engineering of Computer Based Systems (ECBS 2006)*, pages 453–459. IEEE Computer Society, 2006. doi:10.1109/ECBS.2006.57.

27 Michael W. Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In Michael Burke and Mary Lou Soffa, editors, *Proc. Conference on Programming Language Design and Implementation (PLDI 2001)*, pages 13–23. ACM, 2001. doi:10.1145/378795.378798.

28 Pascal Hitzler. A review of the semantic web field. *Commun. ACM*, 64(2):76–83, 2021. doi:10.1145/3397512.

29 Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan F. Sequeda, Steffen Staab, and Antoine Zimmermann. Knowledge graphs. *ACM Comput. Surv.*, 54(4):71:1–71:37, 2022. doi:10.1145/3447772.

30 Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible SROIQ. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors, *Proc. 10th International Conference on Principles of Knowledge Representation and Reasoning (KR 2006)*. AAAI Press, 2006. URL: http://www.aaai.org/Library/KR/2006/kr06-009.php.

31 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. doi:10.1145/503502.503505.

32 Félix Ingrand and Malik Ghallab. Deliberation for autonomous robots: A survey. *Artif. Intell.*, 247:10–44, 2017. doi:10.1016/j.artint.2014.11.003.

33 Einar Broch Johnsen, Marcel Kyas, and Ingrid Chieh Yu. Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. In Ana Cavalcanti and Dennis Dams, editors, *Proc. Second World Congress on Formal Methods (FM 2009)*, volume 5850 of *Lecture Notes in Computer Science*, pages 596–611. Springer, 2009. doi:10.1007/978-3-642-05089-3_38.

34 Einar Broch Johnsen and Ingrid Chieh Yu. Dynamic software updates and context adaptation for distributed active objects. In *Principled Software Development*, pages 147–164. Springer, 2018. doi:10.1007/978-3-319-98047-8_10.

35 Eduard Kamburjan, Nelly Bencomo, Silvia Lizeth Tapia Tarifa, and Einar Broch Johnsen. Declarative lifecycle management in digital twins. In *Proc. 1st International Conference on Engineering Digital Twins (EDTconf 2024)*, MODELS Companion'24, pages 353–363. ACM, 2024. doi:10.1145/3652620.3688248.

36 Eduard Kamburjan and Einar Broch Johnsen. Knowledge structures over simulation units. In *Annual Modeling and Simulation Conference (ANNSIM 2022)*, pages 78–89. IEEE, 2022. doi:10.23919/ANNSIM55834.2022.9859490.

37 Eduard Kamburjan, Vidar Norstein Klungre, Rudolf Schlatte, David Cameron, S. Lizeth Tapia Tarifa, and Einar Broch Johnsen. Digital twin reconfiguration using asset models. In *Proc. 11th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2022)*, volume 13704 of *Lecture Notes in Computer Science*, pages 71–88. Springer, 2022. doi:10.1007/978-3-031-19762-8_6.

38 Eduard Kamburjan, Vidar Norstein Klungre, Rudolf Schlatte, Einar Broch Johnsen, and Martin Giese. Programming and debugging with semantically lifted states. In *Proc. Extended Semantic Web Conference (ESWC 2021)*, volume 12731 of *Lecture Notes in Computer Science*, pages 126–142. Springer, 2021. doi:10.1007/978-3-030-77385-4_8.

39 Eduard Kamburjan and Egor V. Kostylev. Type checking semantically lifted programs via query containment under entailment regimes. In *Description Logics*, volume 2954 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2021. URL: https://ceur-ws.org/Vol-2954/paper-19.pdf.

40 Eduard Kamburjan, Andrea Pferscher, Rudolf Schlatte, Riccardo Sieve, Silvia Lizeth Tapia Tarifa, and Einar Broch Johnsen. Semantic reflection and digital twins: A comprehensive overview. In Mike Hinchey and Bernhard Steffen, editors, *The Combined Power of Research, Education, and Dissemination: Essays Dedicated to Tiziana Margaria on the Occasion of Her 60th Birthday*, volume 15240 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2025. doi:10.1007/978-3-031-73887-6_11.

41 Eduard Kamburjan, Riccardo Sieve, Chinmayi Prabhu Baramashetru, Marco Amato, Gianluca Barmina, Eduard Occhipinti, and Einar Broch Johnsen. GreenhouseDT: An exemplar for digital twins. In *Proc. 19th Intl. Symp. on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'24)*, pages 175–181. ACM, 2024. doi:10.1145/3643915.3644108.

42 Erkan Karabulut, Salvatore F. Pileggi, Paul Groth, and Victoria Degeler. Ontologies in digital twins: A systematic literature review. *Future Gener. Comput. Syst.*, 153:442–456, 2024. doi:10.1016/j.future.2023.12.013.

43 Hirofumi Katsuno and Alberto O. Mendelzon. On the difference between updating a knowledge base and revising it. In James F. Allen, Richard Fikes, and Erik Sandewall, editors, *Proc. 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR 1991)*, pages 387–394. Morgan Kaufmann, 1991.

44 Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003. doi:10.1109/MC.2003.1160055.

45 Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *J. Log. Program.*, 31(1-3):59–83, 1997. doi:10.1016/S0743-1066(96)00121-5.

46 Vladimir Lifschitz. What is answer set programming? In Dieter Fox and Carla P. Gomes, editors, *Proc. 23rd Conference on Artificial Intelligence (AAAI 2008)*, pages 1594–1597. AAAI Press, 2008. URL: http://www.aaai.org/Library/AAAI/2008/aaai08-270.php.

47 Erik Meijer, Brian Beckman, and Gavin M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proc. International Conference on Management of Data*, page 706. ACM, 2006. doi:10.1145/1142473.1142552.

48 Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2 edition, 1997. URL: http://www.eiffel.com/doc/oosc/page.html.

49 Alessandro Orso, Anup Rao, and Mary Jean Harrold. A technique for dynamic updating of Java software. In *Proc. 18th International Conference on Software Maintenance (ICSM 2002)*, pages 649–658. IEEE Computer Society, 2002. doi:10.1109/ICSM.2002.1167829.

50 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

51 Luís Pina, Luís Veiga, and Michael W. Hicks. Rubah: DSU for Java on a stock JVM. In Andrew P. Black and Todd D. Millstein, editors, *Proc. International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA 2014)*, pages 103–119. ACM, 2014. doi:10.1145/2660193.2660220.

52 Susanne Cech Previtali and Thomas R. Gross. Dynamic updating of software systems based on aspects. In *Proc. 22nd International Conference on Software Maintenance (ICSM 2006)*, pages 83–92. IEEE Computer Society, 2006. doi:10.1109/ICSM.2006.23.

53 Yuanwei Qu, Eduard Kamburjan, Anita Torabi, and Martin Giese. Semantically triggered qualitative simulation of a geological process. *Applied Computing and Geosciences*, 21:100152, 2024. doi:10.1016/j.acags.2023.100152.

54 Tim Rentsch. Object oriented programming. *ACM SIGPLAN Notices*, 17(9):51–57, 1982. doi:10.1145/947955.947961.

55 Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. A survey of dynamic software updating. *J. Softw. Evol. Process.*, 25(5):535–568, 2013. doi:10.1002/smr.1556.

56 Bjarne Stroustrup. What is object-oriented programming? *IEEE Softw.*, 5(3):10–20, 1988. doi:10.1109/52.2020.

57 Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Proc. 7th Symposium on Logic in Computer Science (LICS '92)*, pages 162–173. IEEE Computer Society, 1992. doi:10.1109/LICS.1992.185530.

58 Danny Weyns. *An Introduction to Self-Adaptive Systems: A Contemporary Software Engineering Perspective*. John Wiley & Sons, UK, 2020. doi:10.1002/9781119574910.

59 Guohui Xiao, Diego Calvanese, Roman Kontchakov, Domenico Lembo, Antonella Poggi, Riccardo Rosati, and Michael Zakharyaschev. Ontology-based data access: A survey. In *Proc. Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI 2018)*, pages 5511–5519. ijcai.org, 2018. doi:10.24963/IJCAI.2018/777.

60 Jinzhi Lu Xiaochen Zheng and Dimitris Kiritsis. The emergence of cognitive digital twin: vision, challenges and opportunities. *International Journal of Production Research*, 60(24):7610–7632, 2022. doi:10.1080/00207543.2021.2014591.

In-Memory Object Graph Stores

Aditya Thimmaiah
The University of Texas at Austin, TX, USA

Zijian Yi
The University of Texas at Austin, TX, USA

Joseph Kenis
The University of Texas at Austin, TX, USA

Christopher J Rossbach
The University of Texas at Austin, TX, USA

Milos Gligoric
The University of Texas at Austin, TX, USA

Abstract

We present a design and implementation of an in-memory object graph store, dubbed ϵStore. Our key innovation is a storage model – epsilon store – that equates an object on the heap to a node in a graph store. Thus any object on the heap (without changes) can be a part of one, or multiple, graph stores, and vice versa, any node in a graph store can be accessed like any other object on the heap. Specifically, each node in a graph is an object (i.e., instance of a class), and its properties and its edges are the primitive and reference fields declared in its class, respectively. Necessary classes, which are instantiated to represent nodes, are created dynamically by ϵStore. ϵStore uses a subset of the Cypher query language to query the graph store. By design, the result of any query is a table (ResultSet) of references to objects on the heap, which users can manipulate the same way as any other object on the heap in their programs. Moreover, a developer can include (transitively) an arbitrary object to become a part of a graph store. Finally, ϵStore introduces compile-time rewriting of Cypher queries into imperative code to improve the runtime performance. ϵStore can be used for a number of tasks including implementing methods for complex in-memory structures, writing complex assertions, or a stripped down version of a graph database that can conveniently be used during testing. We implement ϵStore in Java and show its application using the aforementioned tasks.

2012 ACM Subject Classification Software and its engineering

Keywords and phrases Object stores, Graph stores, Cypher

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.30

Supplementary Material *Software (Source Code)*: `https://github.com/EngineeringSoftware/eStore`, archived at `swh:1:dir:72a974d02f33141537835dcdb9d1661af263a253`

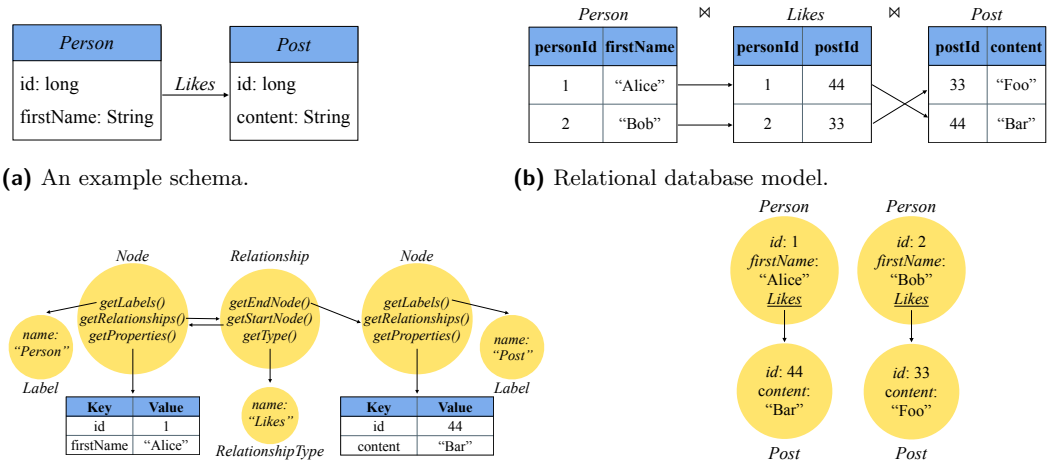
Funding This work is partially supported by the US National Science Foundation under Grant Nos. CCF-2107291, CCF-2217696, CCF-2313027, and CCF-2403036.

Acknowledgements We thank Michael Y. Levin, Zhiqiang Zang, Yu Liu, Nader Al Awar, Jiyang Zhang, Linghan Zhong, Cheng Ding, Ivan Grigorik, Tong-Nong Lin and the anonymous reviewers for their feedback on this work.

1 Introduction

We present a design and implementation of an *in-memory object graph store*, dubbed ϵStore, which enables easy implementation of methods for complex structures, writing assertions over a set of objects on the heap, and substituting (as a lightweight alternative) a graph database in an application during the testing process.

© Aditya Thimmaiah, Zijian Yi, Joseph Kenis, Christopher J Rossbach, and Milos Gligoric;
licensed under Creative Commons License CC-BY 4.0



(a) An example schema.

(b) Relational database model.

(c) Traditional graph database model.

(d) ϵStore model.

Figure 1 Comparison of different storage models for the schema in (a).

ϵStore introduces a novel storage model–epsilon storage–that equates an object on the heap to a node in a graph store. Figure 1 illustrates the differences among well-known storage models and epsilon storage; a schema (or class diagram) that is being instantiated is shown in Figure 1a. In ϵStore, any object on the heap (without changes) can be a part of a graph store (or even multiple stores), and vice versa, any node in a graph store can be accessed like any other object on the heap. Specifically, each node in a graph is an object (i.e., instance of a class). Its properties and its edges are the primitive and reference fields declared in its class, respectively. Necessary classes, which are instantiated to represent nodes, are created dynamically by ϵStore.

ϵStore uses a subset of the Cypher query language [22] to query the graph store; Cypher is a powerful yet concise declarative language popular in the space of graph databases [55]. In our design, the result of any query is a table (ResultSet [51]) of references to objects on the heap, which users can manipulate the same way as any other object on the heap in their programs. Moreover, a developer can include (transitively) an arbitrary object to become a part of a graph store. Processing, e.g., parsing, Cypher queries at runtime can be costly, thus, ϵStore includes support for query rewriting into imperative code at compile-time. Our experiments show query rewrites improving query end-to-end execution times by 5x.

We implemented ϵStore in Java. Our primary focus was to enable novel programming style and propose an efficient storage model. We demonstrate the uniqueness of ϵStore with three use cases. First, we demonstrate the use of ϵStore for concisely expressing complex assertions. Second, we show how ϵStore can be used for implementing various methods. Visualisation of object relations and their fields through graphs can simplify API design. We write methods for a dozen of widely-used data structures from popular libraries (e.g., Guava [21]), as well as methods for H2 [23], an in-memory relational database. We compare ϵStore with OGO [57] in terms of runtime performance. OGO is a framework for Java that allows using Cypher to query the heap. Third, we use ϵStore as a lightweight replacement for graph databases; using an in-memory database (or another form of an object store) is common during testing to save setup cost and runtime cost incurred if a full-blown database is used [6]. At the same time, we note that ϵStore is *not* meant to replace graph databases in production as that is not the primary intent for ϵStore. To estimate benefits of using

ϵStore instead of a graph database, if so desired, we use queries and datasets from the Social Network Benchmark (SNB) [1] of the Linked Data Benchmark Council (LDBC) [56]. LDBC is the most popular benchmark for graph databases. The LDBC SNB was designed to model a snapshot of the activity in a realistic social network. Finally, we evaluate the compile-time code rewriting capability of ϵStore, on the LDBC queries, by comparing its performance with the vanilla version of ϵStore.

Our results show versatility of ϵStore for various tasks and good performance in case a lightweight store is sufficient in the testing process.

The key contributions of this paper include:

- **Idea.** We introduce a novel storage model. Besides being used as a traditional object store (manipulated only via queries), our design enables a unique interoperability between imperative code and objects in a graph store. Results of queries are references to objects in a graph store, thus enabling further imperative processing of the results. Furthermore, any object, which is created by imperative code, can be included into a graph store without any intermediate abstraction and queried for complex relations.
- **Formalization.** We formalize the core of the proposed storage model and the set of API operations supported by ϵStore. We also define a mapping and describe the way instances of any existing class, map to nodes and edges, and can be queried.
- **Implementation.** We implemented ϵStore in Java thus using Java features to dynamically create and load classes that are necessary to represent nodes and their properties. We focus on enabling novel programming models. We also perform compile-time query rewriting into imperative code to reduce the runtime cost. Our implementation is publicly available on GitHub[1].
- **Evaluation.** We performed a three-pronged evaluation. First, we evaluated ϵStore on queries and datasets from the LDBC SNB. Second, we evaluated the power of ϵStore by comparing its execution of library methods of Java data structures, implemented as Cypher queries, with OGO. Finally, we evaluated the compile-time code rewriting capability of ϵStore on the LDBC SNB benchmark queries.

2 Example

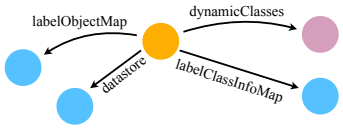
ϵStore is an in-memory graph store. We demonstrate several aspects of ϵStore using an example that showcases the following: (1) creation of a graph store, (2) creation of nodes and edges, (3) querying the graph store, and (4) capturing existing objects into a graph store. We also use this example to provide a brief introduction to the Cypher graph query language [22].

Schema. We use a subset of the LDBC SNB schema shown in Figure 1a to discuss the example. It contains two entities, `Person` and `Post`. The `Person` entity has two properties: `id` of type `long` and `firstName` of type `String`. The `Post` entity has properties: `id` of type `long` and `content` of type `String`. A `Person` can be in a relation (`LIKES`) with a `Post`.

Introduction to Cypher. Cypher is a declarative query language introduced by Neo4j and designed to be expressive when querying graph stores. The labelled property graph (LPG) data model uses nodes and relations to model data. A simple LPG modelling

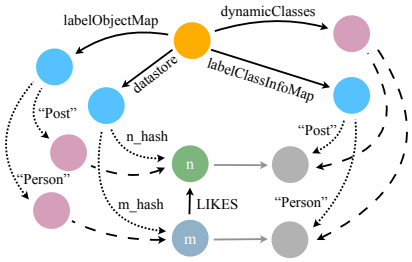
[1] `https://github.com/EngineeringSoftware/eStore`

```
1 /* Instantiate EpsilonStore instance **/
2 EpsilonStore  db = new EpsilonStore ("dbname");
```



(a) Creating an ϵStore instance with name *dbname*.

```
1 /* Add Person and Post nodes to EpsilonStore */
2 db.query(
3  "CREATE (m:`Person` {id: 112, firstName: 'Eve'}"
4  +"-[r:LIKES]->"
5  +"(n:`Post` {id: 481, content: 'About Databases'})"
6 );
```



(b) Insertion of objects without explicit schema (dynamically creating classes `Person` and `Post` at runtime with ASM).

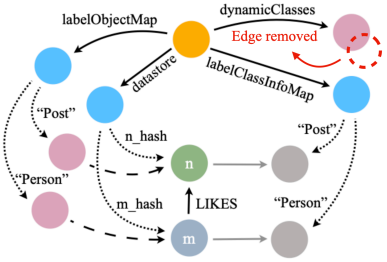
```
1 public class Person {
2     private long id;
3     private String firstName;
4     private Post[] LIKES;
5 }
```

(c) Person.java.

```
1 public class Post {
2     private long id;
3     private String content;
4
5 }
```

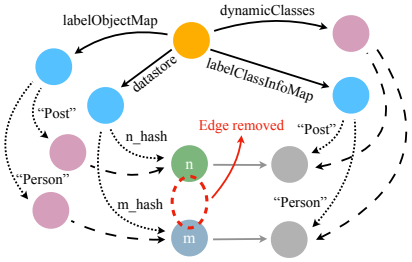
(d) Post.java.

```
1 /* Add Person and Post nodes to EpsilonStore */
2 Post post = new Post(481L, "About Databases");
3 Person person = new Person(112L, "Eve", post);
4 db.captureAll(person);
```



(e) Insertion of objects with pre-defined schema (using existing `Person` and `Post` classes).

```
1 /* Execute Cypher query to remove the LIKES edge */
2 db.query(
3  "MATCH (m:`Person` {id: 112})"
4  +"-[r:LIKES]->"
5  +"(n:`Post` {id: 481}) "
6  +"DELETE r"
7 );
```



(f) Deleting relations using Cypher queries.

Figure 2 An illustrative example to showcase ϵStore for creating a graph store, creating nodes and relations, capturing existing objects, and querying the state of the graph store. The left side of each sub-figure will show the code and the right side will show the corresponding state of the ϵStore instance.

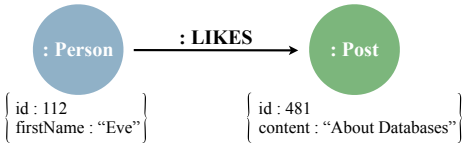


Figure 3 An example labelled property graph.

the schema in Figure 1a is given in Figure 3. In Cypher semantics, the data entities `Person` and `Post` are called *nodes*, and `LIKES` is called a *relation*. Both nodes and relations can have one or more *labels*. Labels function as types, and groups similar nodes and relations. In the example LPG, the `Person` entity has label *Person* and the `Post` entity has label *Post*. The relation between these nodes has the label *LIKES*. Both, nodes and relations, can have properties, which are a set of key-value pairs. The key-value pairs describe the property name and value. The `Person` node has properties *id* with value 112 and *firstName* with value "Eve". A node may have 0 or more relations and can have relations with the same label to different nodes. A relation may be non-directional, uni-directional or bi-directional. We now describe Cypher syntax. A node is generally represented in a Cypher query as `()` and we can optionally specify a variable to reference it in later clauses such as `(n)`. To match and return the node with label `Person`, we would write the Cypher query as `MATCH (n:`Person`{id:112, firstName:"Eve"}) RETURN n`. In a larger context, we can be more specific with the query: `MATCH (n:`Person`{id:112, firstName:"Eve"})-[:LIKES]->(m:`Post`{id:481, content:"About Databases"}) RETURN n`.

Creating the graph store. The ϵStore graph store is created by invoking the constructor of the `EpsilonStore` class with at least one argument to specify the name for the graph store as shown in Figure 2a. The constructor call also instantiates several reference fields of ϵStore. The *datastore* is a map that maps the *ID* of an object present in ϵStore to itself. It can be used to optimize queries through ID based lookups. The *dynamicClasses* is a list that stores the `java.lang.Class` instances created dynamically by ϵStore at runtime. The *labelClassInfoMap* is a map that maps fully qualified class names of classes of objects present in ϵStore to `ClassInfo` instances. The `ClassInfo` class of ϵStore is used to cache primitive and reference field information of a class such as type and name of fields and is used during query execution to access the fields of objects of that class. Finally, the *labelObjectMap* is a map that maps fully qualified names of classes of objects present in ϵStore, to list instances that store objects of the corresponding classes.

Inserting objects without explicit schema. Nodes in ϵStore can be any Java object. Nodes can be created using `CREATE` Cypher queries that instantiate objects of specified classes. If the specified class cannot be found in the classes loaded by the JVM, it is dynamically created by ϵStore. An example of such a query is given in Figure 2b. We refer to such object insertions into ϵStore as insertions *without* explicit schema.

The Cypher query in Figure 2b, as defined by the Cypher grammar, is a *single part query* with an update (`CREATE`) clause. The `CREATE` clause, syntactically, is made up of the token `CREATE` followed by a pattern. The pattern may be a single node pattern (`(n)`) or a multi-node pattern with relations (`(n)-[]->(m)-[]->(p)`) or multiple single node patterns (`(n),(m),(p)`). The pattern in our example is a two node pattern with relations `(:`Person`{...})-[:LIKES]->(:`Post`{...})`. The objective of the `CREATE` clause in ϵStore is to create instances and assign references between them based on the nodes and

relations specified in the pattern. Therefore, the query in Figure 2b creates an instance of type `Person` and an instance of type `Post` with the given properties. It then assigns the reference field `LIKES` of the `Person` instance with the `Post` instance. Assuming that the `Person` and `Post` classes are not present in the classes loaded by the JVM, ϵStore uses the byte code manipulation framework, ASM [7], to first dynamically create these classes.

The *node properties* are mapped to the primitive fields of these classes whereas the relations are mapped to their reference fields. All reference fields, created using class creation, are *Arrays* of type `Object` with the same name as the relation labels. This design decision allows support for one-to-many relations with the same label between nodes.

Inserting objects with explicit schema. ϵStore also supports inserting instances of existing classes using the `captureAll` API method as shown in Figure 2e. `captureAll` captures all references under reflexive transitive closure, and hence, all objects reachable from the captured object are also inserted into the store. The class definitions of `Person` and `Post` are given in Figure 2c and Figure 2d, respectively. The field *dynamicClasses* is empty since the classes of the objects inserted already exist and are not required to be created dynamically.

Data modification through queries. The objects captured into an instance of ϵStore can be modified using Cypher queries. The query (lines 3-6) in Figure 2f deletes the relation between two nodes. It is a single part query with two clauses, a reading (`MATCH`) clause and an update (`DELETE`) clause. The `MATCH` clause identifies a set of objects and their references that matches the specified pattern. The specified pattern is a two node relation pattern specifying the labels, properties and relations for the referrer and referee nodes. Since this pattern exists in our store, the variables `m,r,` and `n`, are mapped to the `Person` instance, the reference field `LIKES` of the `Person` instance, and the `Post` instance respectively. The `DELETE` clause on line 6 deletes all the objects mapped to the variable `r`. Therefore, the field `LIKES` of the referrer node is set to `null` which is shown as the edge being removed in the corresponding graph store state.

3 Graph Store

We first formalize our proposed storage model and the core operations supported by ϵStore (§3.1). We then discuss the mapping between Cypher semantics and Java semantics as implemented by ϵStore (§3.2), followed by a discussion on the API methods provided by ϵStore (§3.3). Finally, we highlight the key implementation details (§3.4).

3.1 Semantics

ϵStore requires no changes to the language syntax, compiler, or execution environment. Thus, we focus on formalizing the core operations of ϵStore such as capturing, deleting, and querying objects. We use big-step operational semantics in our formalization. This section clarifies these operations by providing precise definitions and illustrative examples for each of the rules.

Table 1 shows the key symbols used in our formalization. We define a type (τ) as a (type_name, set_of_fields) pair. Each field is a tuple: (name, type_name), and has a unique name within a type definition. To simplify discussion, we will assume that `int` and `string` are the only primitive types available: (int, $\emptyset$) and (string, $\emptyset$). We define a set type (Set, {}), as an untyped set of values. We also define a metadata type that will be used to describe a type: (Meta, {(name, string)}). We use Γ to denote a set of all available types at runtime.

Table 1 Definitions of key symbols used in our formalization of ϵSTORE operations.

Symbol	Definition
Ξ	The set of all objects available on the heap.
Γ	The set of all available types at runtime.
τ	A type definition ($\tau \in \Gamma$), consisting of a type name and a set of fields.
o	An object on the heap ($o \in \Xi$).
o^τ	A meta-type instance ($o^\tau \in \Xi$).
$\overline{o}$	An instance of ϵSTORE
fields(o)	The set of non-primitive field names belonging to object o.
meta(o)	Retrieves the meta-type instance (o^τ) of the object o.
type(v)	Retrieves the type (primitive) of the primitive value v.
new(τ)	Creates a new object instance of type τ.
ntype(L, f)	Creates a new type named L with a set of fields defined in f which is a tuple of field names and their types ($f = (n_1 : t_1, ..., n_k : t_k)$).
$o[p/v]$	Represents an object o with a set of primitive fields p assigned to a set of values v.
$o.name$	Retrieves the value of the field named $name$ of the object o.
$o.^\star name$	Set of references reachable under transitive closure from the field named $name$ of o via non-primitive fields.
$\Xi(o, f \leftarrow v)$	Assigns the value v to the field f of object o.

We use Ξ to denote all objects available on the heap, i.e., $\Xi = \{o_1, o_2, ..., o_n\}$. Each object ($o$) is an instance of a type (τ) and has a unique identifier. An object has a set of values, each corresponding to one field of the object's type. An access to a field (o.name) returns its current value, and a "star" access to a field ($o.^\star$name) returns a transitive closure, i.e., set of objects reachable via non-primitive fields starting from the given field. We use the following notation to update the field `f` of an object o to value v: $\Xi(o, f \leftarrow v)$.

For each type (τ) in a running program, there is an object (o^τ) on the heap, created by the execution environment, which is an instance of the Meta type (analogous to instance of `java.lang.Class` [47])).

An instance of ϵSTORE is simply an object on the heap ($\overline{o} \in \Xi$). We define the ϵSTORE type as (ϵSTORE, {(store, Set)}). Thus, objects in a graph store are the objects reachable via the store field of an ϵSTORE instance, i.e., $db = \overline{o}.^\star$store, and we have that $db \subset \Xi$. As a result of our design, it is trivial to have any object on the heap inserted into a graph store, to share objects across graph stores, and even to embed one graph store into another.

We define the following helper functions: meta(o) returns the metadata object for the given object; fields(o) returns the set of reference field names for the given object; new(τ) creates a new object (o) of the type τ on the heap and its corresponding set of primitive fields p and their values v is denoted by $o[p/v]$; ntype(L, f) makes a new type (named L) with field names and types defined in the tuple f (tuple of field names mapping to their corresponding types). type(v) returns a name of the primitive type for the given primitive value.

We now formally define the core high-level operations that can be performed on an ϵSTORE instance. In all cases, $o \in \Xi \wedge \overline{o} \in \Xi$ and we use the following configuration:

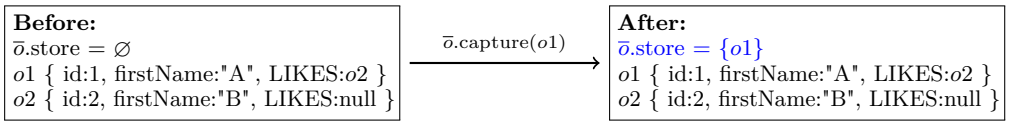
$\langle operation, \Xi, \Gamma \rangle$

For each rule, we first specify its big-step operational semantics, followed by a brief description of the rule, and finally an example showing the state of $\overline{o}$ and other relevant objects, before and after rule application. We assume that all objects used in these examples are instances of type `Person` and that this type exists in Γ unless specified otherwise. For each of the examples, the relevant changes in the after state are highlighted in blue.

▶ **Rule 3.1** (Capture).

$$\frac{\text{store}' = \overline{o}.\text{store} \cup \{o\}}{\langle \overline{o}.capture(o), \Xi, \Gamma\rangle \Downarrow \langle _, \Xi(\overline{o}, \text{store} \leftarrow \text{store}'), \Gamma\rangle}$$

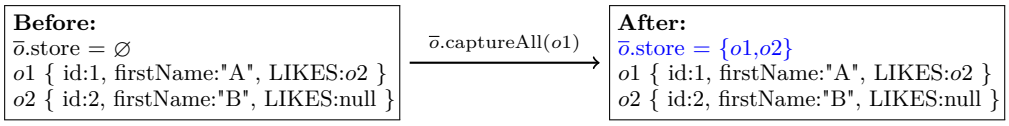
The `capture` rule defines the operation of inserting a single object o (present on the heap Ξ) into an ϵStore instance $\overline{o}$. The union in the premise ensures that if o already exists in $\overline{o}$, then no modifications occur to $\overline{o}$. In the example, we see that the ϵStore instance is empty and there exists an object o on the heap in the before state. After applying the `capture` rule, the ϵStore instance contains the the object $o1$.



▶ **Rule 3.2** (CaptureAll).

$$\frac{C = \overline{o}.\text{store} \cup \{o.^\star f | f \in \text{fields}(o)\} \cup \{o\}}{\langle \overline{o}.captureAll(o), \Xi, \Gamma\rangle \Downarrow \langle _, \Xi(\overline{o}, \text{store} \leftarrow C), \Gamma\rangle}$$

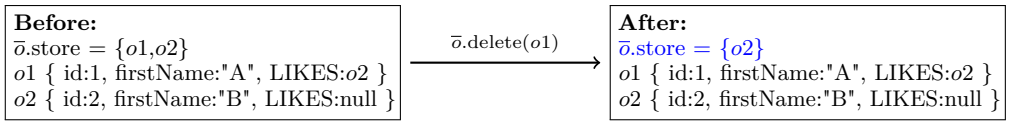
The `captureAll` rule is similar to the `capture` rule but here, the reference fields of the object o being captured are transitively visited and captured into $\overline{o}$ in addition to o. We see that in the premise of the rule, we first collect all the reference fields of o (fields(o)). Next, for every reference field (f), we collect the set of references reachable under transitive closure from o through that field ($o.^\star f$). This set is then unioned with o and the existing store of $\overline{o}$ to get the new store. In the example, we see that the store of $\overline{o}$ is initially empty and there are 2 instances of `Person` ($o1$,$o2$) with one referencing the other. On applying the rule to the referrer instance ($o1$), the store now contains both, the referrer and the referee instances.



▶ **Rule 3.3** (Delete).

$$\frac{\text{store}' = \overline{o}.\text{store} \setminus o}{\langle \overline{o}.delete(o), \Xi, \Gamma\rangle \Downarrow \langle _, \Xi(\overline{o}, \text{store} \leftarrow \text{store}'), \Gamma\rangle}$$

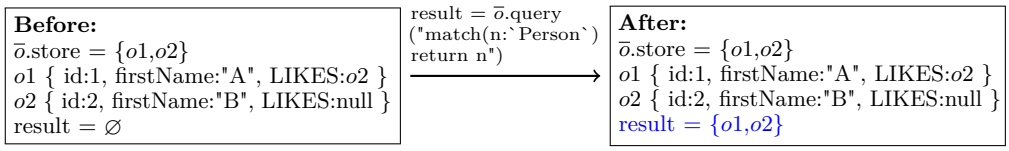
The `delete` rule is used to remove contained objects from ϵStore. In the premise, we remove the object o from the existing store to get the new store (store$'$). This is then used to update the store in the conclusion. If the object does not exist in the store then the store is unmodified after applying the rule. Deleting an object only removes that object from the store while its references that may be part of the store are not removed. In the example, the store initially contains two instances ($o1$,$o2$) with one referencing the other. On applying the rule to the referrer instance, only the referrer instance ($o1$) is removed in the after state of the store.



▶ **Rule 3.4** (Match).

$$\frac{C = \{o \mid o \in \overline{o}.^{\star}\text{store} \wedge \text{meta}(o).\text{name} = L\}}{\langle _ = \overline{o}.query(\text{“match \{a:}L\text{\} return a”}), \Xi, \Gamma\rangle \Downarrow \langle _ = C, \Xi, \Gamma\rangle}$$

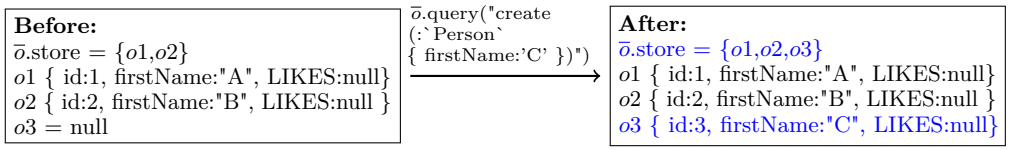
The `match` rule is used to query the store and retrieve stored objects matching one or more specified predicates. Although this rule supports complex predicates, we use a simple predicate to simplify its semantic description. We use the predicate of matching and retrieving all objects in the store whose type matches L. In the premise, we collect the set of all stored objects whose meta-type name matches L. This set is returned as the result of the query in the conclusion. In the example, the store initially contains two instances ($o1$,$o2$) both of type `Person`. The result is initially empty. On applying the rule to retrieve all stored objects of type `Person`, we see that the result now contains $o1$ and $o2$ which are the stored objects of type `Person`.



▶ **Rule 3.5** (Create with pre-defined schema).

$$\frac{L \in \Gamma \qquad o = new(L) \qquad o[p/v]}{\langle \overline{o}.query(\text{“create \{:}L\ \ \{p : v\}\}\text{”}), \Xi, \Gamma\rangle \Downarrow \langle \overline{o}.capture(o), \Xi, \Gamma\rangle}$$

The `create` rule with pre-defined schema is used to create and capture an object of a given existing type into an ϵSTORE instance. This rule uses the `CREATE` clause in the Cypher query. We use a simple example of creating an object of an existing type L with a field p and value v to describe its semantics. In the premise, we first create the object o of type L and update its field named p to value v. In the conclusion, we simply invoke the previously defined `capture` rule on the ϵSTORE instance with o as argument to capture it into the store. In the example below, the store initially contains 2 instances of type `Person` ($o1$, $o2$). After applying the rule, we see that the store now contains 3 instances of type `Person` and $o3$ is now `non-null`. We assign a “name” $o3$ to this created object only for the purpose of showing the before and the after states of applying the rule.

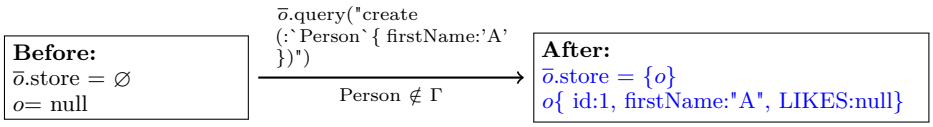


▶ **Rule 3.6** (Create without pre-defined schema).

$$\frac{L \notin \Gamma \qquad \tau = \text{ntype}(L, (\text{“}p\text{”} : type(v))\}) \qquad o = new(\tau) \qquad o[p/v]}{\langle \overline{o}.query(\text{“create \{:}L\ \ \{p : v\}\}\text{”}), \Xi, \Gamma\rangle \Downarrow \langle \overline{o}.capture(o), \Xi, \Gamma \cup \{\tau\}\rangle}$$

The `create` without pre-defined schema is used to create an object of non-existing type and capture it into the ϵSTORE instance. The query used is similar to `create` with pre-defined schema except, now the type L is assumed to not exist in the set of available runtime types

Γ. We use the same example used in create-with-pre-defined-schema rule to describe the semantics. In the premise, we first create a type L with a field named p with type matching that of the primitive value v. Next, we create an instance o of type L and update its field named p with value v. Finally, in the conclusion, we invoke the previously defined `capture` rule on the ϵSTORE instance with o as the argument, to capture it into the store. The newly created type is unioned into the set of available runtime types Γ. This ensures that new objects of this type can now be created by applying the create-with-pre-defined-schema rule instead. In the example below, the store is initially empty. On applying the rule under the assumption that the `Person` type does not yet exist, the store now contains an instance of type `Person` and the object o is `non-null`.



We will now use the operations we formalized to prove 2 properties about ϵSTORE. In the theorems that follow, σ represents the state of the ϵSTORE. $\sigma(o) = \perp$ denotes that the object o is absent from the store. We use the configuration $\langle operation, \sigma, \Gamma\rangle$ in our formal description of the theorems and their proofs.

▶ **Theorem 1** (Idempotency of Insertions). *Repeated insertions of the same object o into an ϵSTORE instance $\overline{o}$ yields an ϵSTORE instance state identical to inserting the object just once. We can formally state the theorem as $\forall\ o$, σ and $\sigma_0(o) \neq \perp$:*

$$\bigwedge_{i=0}^{n} (\langle \overline{o}.capture(o), \sigma_i, \Gamma\rangle \Downarrow \langle _, \sigma_{i+1}, \Gamma\rangle) \implies \sigma_{n+1} = \sigma_0$$

Equality of the states is defined using the standard definition for set equality i.e., $\sigma_{n+1} = \sigma_0 \iff (\sigma_{n+1} \subseteq \sigma_0 \wedge \sigma_0 \subseteq \sigma_{n+1})$.

Proof. We will use mathematical induction to prove this theorem.

1. **Base Case (i = 0):** Prior to capture, o is present in the store, i.e., $\sigma_0(o) \neq \perp$. When a capture operation is executed, the state transitions as follows:

   $$\langle \overline{o}.\text{capture}(o), \sigma_0, \Gamma\rangle \Downarrow \langle _, \sigma_1, \Gamma\rangle \qquad \text{where } \sigma_1 = \sigma_0 \cup \{o\} \text{ by rule } \texttt{capture}$$

   σ_1 denotes the updated state. We know that $\sigma_0 \cup \{o\} = \sigma_0$ since $\sigma_0(o) \neq \perp$. Therefore, $\sigma_1 = \sigma_0$ and the theorem holds for the base case.
2. **Inductive Step (i = n):** Consider an $(n-1)^{th}$ transition:

   $$\langle \overline{o}.\text{capture}(o), \sigma_{n-1}, \Gamma\rangle \Downarrow \langle _, \sigma_n, \Gamma\rangle$$

   Lets assume that the theorem holds for $i = (n-1)$ i.e., $\sigma_n = \sigma_0$. Now for the n^{th} transition, by the semantics defined in rule capture, $\sigma_{n+1} = \sigma_n \cup \{o\}$. However, by our assumption of the theorem holding for $i = (n-1)$, $\sigma_{n+1} = \sigma_0 \cup \{o\}$. By base case, we know that $\sigma_0 \cup \{o\} = \sigma_0$. Therefore, we can claim by mathematical induction that $\sigma_{n+1} = \sigma_0$ as required by the theorem. ◀

▶ **Theorem 2** (Persistence of Inserted Objects). *If an object o is inserted into the store at state σ, then o remains in the store in all subsequent states unless explicitly removed via a deletion operation. We can formally state the theorem as $\forall\ o$, σ and $\sigma_0(o) = \perp$:*

$$(\langle \overline{o}.capture(o), \sigma_0, \Gamma\rangle \Downarrow \langle _, \sigma_1, \Gamma\rangle) \bigwedge_{i=1}^{n} (\langle S_i, \sigma_i, \Gamma_i\rangle \Downarrow \langle S_i', \sigma_{i+1}, \Gamma_{i+1}\rangle) \implies \sigma_{n+1}(o) \neq \perp$$

where S_i (evaluates to S_i') is any ϵSTORE supported operation excluding the `delete` operation.

Table 2 Mapping of Cypher constructs to Java in ϵStore. "—" indicates unsupported features.

Group	Cypher Construct	Cypher Syntax	Class: Dynamically created	Class: Existing
Struct.	Node	`()`	`Object`	`Object`
	Relation	`()-[]-(),()-[]->(),()<-[]-()`	`Object[]`	`Object, Object[]`
		`()<-[]->()`	—	—
Misc.	Node Label	`(:<label>)`	Fully qualified class name	Fully qualified class name
	Relation Label	`()-[:<label>]-()`	Reference field name	Reference field name
	Node Properties	`({<name>:<value>,...})`	Primitive, String fields	Primitive, String fields
	Relation Properties	`()-[{<name>:<value>,...}]-()`	—	—
Property Values	Decimal Literal	`[Long.MIN_VALUE, Long.MAX_VALUE]`	`long`	`byte, short, int, long`
	Float Literal	`[-Double.MAX_VALUE, Double.MAX_VALUE]`	`double`	`float, double`
	Boolean Literal	`TRUE, FALSE`	`boolean`	`boolean`
	String Literal	`"<string>"`	`java.lang.String`	`java.lang.String`

Proof. We will use structural induction on the defined semantic rules to prove this theorem.

1. **Base Case (i = 0):** Prior to capture, o is not present in the store, i.e., $\sigma_0(o) = \bot$. When a capture operation is executed, the state transitions as follows:

$$\langle \overline{o}.\text{capture}(o), \sigma_0, \Gamma \rangle \Downarrow \langle _, \sigma_1, \Gamma \rangle \qquad \text{where } \sigma_1 = \sigma_0 \cup \{o\} \text{ by rule } \texttt{capture}$$

   σ_1 denotes the updated state. Thus, immediately after execution, o is present in the store or in other words $\sigma_1(o) \neq \bot$. Therefore, the theorem holds for the base case.
2. **Inductive Step (i = n):** Consider an $(n-1)^{th}$ transition:

$$\langle S_{n-1}, \sigma_{n-1}, \Gamma_{n-1} \rangle \Downarrow \langle S'_{n-1}, \sigma_n, \Gamma_n \rangle$$

   Lets assume that the theorem holds for $i = (n-1)$ i.e., $\sigma_n(o) \neq \bot$. Now for the n^{th} transition, the arbitrary statement S_n must be chosen from set of operations including capturing, creating or matching as per the requirements of the theorem. Since these rules do not specify any removal condition and were the same choice of operations available for the $(n-1)^{th}$ transition. We can claim by structural induction on the operational semantics, o persists in all transitions unless explicitly removed through deletion. Therefore we can conclude that $\sigma_{n+1}(o) \neq \bot$ as required by the theorem. ◀

3.2 Mappings

The mapping of semantics between Cypher and Java in ϵStore, is given in Table 2. The first column (Group) shows Cypher features [29] supported by ϵStore. The *Cypher Construct* and the *Cypher Syntax* columns describe the Cypher feature and its corresponding syntax when querying. We show the mapping between Cypher features and Java classes in two cases: (1) classes that are dynamically created by ϵStore (i.e., schema absent), and (2) already existing classes written by developers (i.e., pre-defined schema present). These are given by columns *Dynamically created* and *Existing*, respectively.

A Cypher *node* maps to any object in ϵStore for both dynamic and existing classes. If a *label* is specified for the node, then it maps to objects that are instances of the class with a *fully qualified name* matching the label.

For existing classes, a *relation* with a label maps to objects referred to by other objects with the reference field name matching the label. These referee objects can be elements of an array. For dynamic classes, relations always refer to the elements of an array of objects with the array reference field name matching the label. This allows an object to have relations with the same label to different objects. Relationships also support *directionality* which enforces a referrer-referee relation. ϵStore supports non-directional and uni-directional relations but not bi-directional owing to the limitation of its implementation language (Java).

```
1 public void capture(Object obj) throws EpsilonStoreException
2 public void captureAll(Object obj) throws EpsilonStoreException
3 public ResultSet query(String cQuery) throws EpsilonStoreException
```

Figure 4 ϵStore's API available via the `EpsilonStore` class. The `capture` method in Line 1 is used to capture a single object into ϵStore. The `captureAll` method in Line 2 is used to transitively capture all objects reachable from, and including, the argument `obj` into ϵStore. Line 3 shows the method to query ϵStore with the given Cypher query string `cQuery`.

Node properties map to an object's *primitive* or `java.lang.String` fields in ϵStore. The property values can be a decimal, floating point, string or boolean literal. To handle the range of values supported by Cypher, ϵStore defaults to mapping decimal and float properties to `long` and `double` fields for dynamic classes. Whereas for existing classes, these properties may map to `byte`, `short`, `int`, `long`, `float`, and `double` fields.

Finally, for an instance (o) of a class `C` (`C extends A`), we treat all the fields (those from `C` and `A`) in o the same.

3.3 API

ϵStore's API has three methods: (1) `capture`, (2) `captureAll`, and (3) `query` as shown in Figure 4. It is our intentional design choice to keep the interface simple. Furthermore, the architecture of ϵStore and the expressivity of Cypher allows most operations to be performed through queries.

Inserting data. The `captureAll` method given in Figure 4 line 2 is used to insert data into ϵStore by capturing all objects reachable from the given argument object `obj` under reflexive transitive closure. We use a Breadth First Search (BFS) [10] strategy to traverse the graph of object references reachable from the root object `obj`. `capture` method inserts only the given object into ϵStore.

Semantically, to capture an object in ϵStore translates to storing references to the object in the `EpsilonStore` instance's `labelObjectMap` and `datastore` fields as well as to store its primitive and reference field information (name and type) in `ClassInfo` instance present inside the `labelClassInfoMap` field.

The algorithm for capturing objects into ϵStore is given in Algorithm 1. We use a FIFO queue inside a loop to collect all the objects directly and indirectly reachable from the argument object through its reference fields during each iteration. In each iteration, we pop an object o' from the head of the queue (line 4), compute its hash (ID) (line 8) to check if it already exists in the ϵStore instance (lines 9-11), and add it to `datastore` if absent (line 12). We then get the fully qualified class name of the object's class (line 13), check if its mapped to a list in `labelObjectMap`, and append the object to the list if mapped (line 18). If not mapped, we then insert a new empty list for this key in `labelObjectMap` (line 15) and append the object. In addition, we also create a `ClassInfo` instance for the class of this object to cache its field information and insert it into the `labelClassInfoMap` field (line 16). Following this, we get the reference field objects of o' and insert them into the queue (line 22). The loop terminates when the queue is empty.

Querying. The `query` method is shown in Figure 4 line 3 and is used to query the ϵStore instance. It takes the Cypher query string `cQuery` as an argument and returns a `ResultSet` containing the result of the query: references to objects in the ϵStore instance.

Algorithm 1 Capturing objects.

Input: An object o to be inserted into ϵStore
Require: An empty FIFO queue q

```
1:  procedure CAPTUREALL(o)
2:      Append o to q
3:      while q is not empty do
4:          o' ← Pop head of q
5:          if o' is null then
6:              continue
7:          end if
8:          h ← getHashCode(o')
9:          if datastore[h] is not null then
10:             continue
11:         end if
12:         datastore[h] ← o'
13:         c ← getClassName(o')
14:         if labelObjectMap[c] is null then
15:             labelObjectMap[c] ← []
16:             labelClassInfoMap[c] ← new
                ClassInfo(c)
17:         end if
18:         Append o' to labelObjectMap[c]
19:         cInfo ← labelClassInfoMap[c]
20:         for all r := getRefFields(cInfo) do
21:             o'' ← getRefObject(cInfo, r, o')
22:             Append o'' to q
23:         end for
24:     end while
25: end procedure
```

Algorithm 2 Creating objects.

Require: Cypher query string $query$ and list of Java classloaders $cList$

```
1:  for all x := in Nodes of query do
2:      l ← Label of x
3:      pNames ← Property names of x
4:      pValues ← Property values of x
5:      pTypes ← inferTypes(pNames, pValues)
6:      c ← GETORMAKECLASS(l, pNames, pTypes)
7:      o ← CreateAndSetFields(c, pNames, pValues, pTypes)
8:      capture(o)
9:  end for

10: procedure GETORMAKECLASS(l, pNames, pTypes)
11:     if labelClassInfoMap[l] is not null then
12:         return getClass(labelClassInfoMap[l])
13:     else
14:         for all loader := cList do
15:             c ← findClassWithLoader(l, loader)
16:             if c is not null then
17:                 labelClassInfoMap[l] ← new ClassInfo(l)
18:                 return c
19:             end if
20:         end for
21:         c ← ASMCreateClass(l, pNames, pTypes)
22:         labelClassInfoMap[l] ← new ClassInfo(l)
23:         return c
24:     end if
25: end procedure
```

Data can also be inserted into ϵStore through Cypher `CREATE` queries. The algorithm for data insertion through queries is given in Algorithm 2. We describe the case when the Cypher query specifies the creation of a multiple single node pattern (`CREATE (n:label1 {...}), (m:label2), ...`). We start by iterating through all the node definitions in the Cypher query string and collect their labels, property names, property values, and their outgoing edge labels. For each property name and value pair, we infer the property type (line 5) using the mapping given in Table 2. Next, for each node definition, we invoke `GETORMAKECLASS` which either finds a class with a fully qualified name matching that node label or dynamically creates a class with the name matching the label and with its field definitions matching the node's property names and types. The `GETORMAKECLASS` procedure first checks (line 11) the `labelClassInfoMap` field for the class, matching the passed in label argument l and returns it if present (line 12). If absent, we attempt to find the class corresponding to the label by iterating through all the classes loaded into the JVM by all the available *classloaders* [12, 48] and return it if found (line 18). If this also fails, then ϵStore proceeds with dynamic class creation at runtime (line 21). We use the bytecode manipulation and analysis framework ASM [7] to dynamically generate the class. Once the class is found or created, we instantiate it (line 7) and set the fields of the instance to the collected property values. The instance is then captured into ϵStore.

The class instantiation procedure `CreateAndSetFields` checks for consistency of the inferred property types and the field definitions present in the class. An exception is thrown if the checks fail due to a type mismatch.

3.4 Implementation

Field access. Cypher queries may specify patterns in their clauses that require matching a node's properties (`MATCH (n {a:10})`) or its relationships (`MATCH ()-[:label]->()`). Executing these queries requires accessing the fields of objects. To optimize query execution and avoid the overhead of repeatedly retrieving the field name and type for every object,

we cache these field information in ϵStore's `ClassInfo` instances. Since fields are defined in an object's class in Java, it is sufficient to have one `ClassInfo` instance to cache the field information for all objects of that class. These `ClassInfo` instances are stored in ϵStore's `labelClassInfoMap` field. This field is a hashmap that maps a label to its corresponding `ClassInfo` instance. `ClassInfo` is an abstract class. We have two concrete implementations of it based on the approach used to retrieve the field values. We refer to these two implementations of ϵStore as ϵStore^r and ϵStore^u.

- **ϵStore^r** uses Java *reflection* [53] to retrieve the field values. This implementation contains a hashmap mapping field names to their corresponding `java.lang.reflect.Field` [50] instances, obtained using reflection. These `Field` instances are used to retrieve the field values.
- **ϵStore^u** uses Java's *unsafe* [41] API to retrieve the field values. This implementation contains a hashmap mapping field names to field offsets (type `long` values). These field offsets are used to retrieve the field values.

Storage. ϵStore stores inserted objects using their unique ID's and their labels (types).

- **Storing by label**. Every object in Java is an instance of a class which defines its type and the fully qualified name of this class is the *label* of the object. The `labelObjectMap` field is used for storing inserted objects based on their label. This field is a hashmap that maps labels to ordered lists of objects belonging to the corresponding labels. During query execution, if the query string specifies a label for a node to be matched, then since `labelObjectMap` stores objects by their labels, we can use it to efficiently search only a subset of the stored objects.
- **Storing by ID**. The ID is assumed to be unique for every inserted object and is by default computed internally by invoking the `identityHashCode` [46] method provided by `java.lang.System` package on the object. The `datastore` field is used for storing objects based on their ID. This field is a *hashmap* that maps the ID of an inserted object to itself. If the ID of an object being inserted is found to already exist in the `datastore` then the old object reference mapped to that ID in the `datastore` is replaced with the new object reference. During query execution, if the query string specifies a node and its ID then the `datastore` can be used to reduce the search space of stored objects and hence optimize query performance.

Our storage schemes are designed such that the storage structures require minimal update on insertion or removal of objects from ϵStore.

3.5 Code Rewriting

The vanilla version of ϵStore parses input queries and generates a query plan at runtime. We notice the overhead is significant, even multiple times greater than the time to actually execute the query. To reduce overhead, we introduce a code rewriting technique in ϵStore. The basic idea is to parse the query at compile time, then when building the query plan at compile time, we inject the query plan execution code directly into the query call site.

Specifically, we introduce a Java annotation to support this feature. When a method is annotated with this annotation, all the queries (passed as arguments to the `query` API described in Section 3.3) inside the method will be preprocessed by a handler. The handler reuses most part of the ϵStore engine; however, instead of executing the query plan and returning the result, it will aggregate imperative Java code of the query plan needed to be executed for the query and replace the query with corresponding imperative code at the call site. At runtime, only plain imperative Java code is executed. This way, we translate the declarative query into imperative code at compile time, and there is no overhead for parsing and building the query plan at runtime.

```
1 public class LinkedList<E>...{
2     transient Node<E> first; ...
3     private static class Node<E> {
4         ...
5         Node<E> next;
6         Node<E> prev; ...
7     } ...
8 }
```

(a) Snippet of the `LinkedList` class.

```
1 public void testAcyclicity(){
2     List<Long> list = new LinkedList<Long>(); ...
3     EpsilonStore  db = new EpsilonStore ("dbname");
4     db.captureAll(list);
5     assertTrue(db.query(
6       "MATCH (n:‘LinkedList$Node‘)-[:next*]->(n)"
7      +" RETURN COUNT(n) = 0").getBoolean(0));
8 }
```

(b) Checking *acyclicity* invariant on the `LinkedList` with ϵStore.

Figure 5 An example showing complex assertions with ϵStore.

4 Use Cases

We describe 3 use cases that are made possible as the result of our design. These examples showcase the unique programming style of equating objects (instances of classes) and nodes in ϵStore. Our examples include: (1) writing complex assertions for checking structural invariants, (2) implementing methods, and (3) using ϵStore as a lightweight graph store.

4.1 Runtime invariant checking with complex assertions

Structural invariants can be easily checked by writing complex assertions using ϵStore queries. We demonstrate this by checking the *acyclicity* invariant of a linked list.

A snippet of the `java.util.LinkedList` [13] class definition is given in Figure 5a. It contains an inner-class `Node`, whose instances are the `LinkedList` instance's nodes. The fully qualified class name of this inner-class in Java is `LinkedList$Node`. An instance of `Node` has a field `next` that holds a reference to its successor node in the list. The `next` field of a node can be `null` if it is the last node in the list. It also contains a field `prev` that holds a reference to its predecessor node.

The acyclicity invariant of a `LinkedList` imposes the condition that no node can be reachable from itself by strictly following only its successor or predecessor nodes. In other words, a `LinkedList` must be free of cycles. Figure 5b shows how such an invariant can be checked with ϵStore. The `LinkedList` instance `list` is first captured into an ϵStore instance using its `captureAll` API method (line 4). Next, we assert on the result of the Cypher query (lines 6-7), that checks for acyclicity of the captured `list`. The query contains a `MATCH` clause that matches a pattern in the captured graph of objects, where a `Node` instance contains a path to itself through 1 or more `next` edges. The `RETURN` clause returns `true` if such a pattern does not exist.

In this manner, an otherwise complex assertion can be concisely expressed using Cypher queries with ϵStore.

4.2 Implementing methods with Cypher queries

H2 is an open-source, lightweight relational database implemented in Java. H2 supports embedded in-memory mode, where it runs within the same JVM as the application. Thus, all the objects related to an H2 instance are on the heap. As a result, we can insert an instance of H2 into an instance of ϵStore. We can then query anything related to the H2 instance or data within that instance. Here, we show a way to query the metadata of an H2 instance.

Figure 6a shows how to get the schemas in an H2 database using the API provided by JDBC [52, 25]. Figure 6d shows the actual implementation of the `getSchemas` API by H2. It imperatively setups `result`, iterates over the schemas and insert them into the `result`.

```
Connection conn = DriverManager.getConnection(
    "jdbc:h2:mem:h2TestDb",
    "sa",
    ""
);
DatabaseMetaData meta = conn.getMetaData();
ResultSet schemas = meta.getSchemas();
```

(a) Getting *schemas* using JDBC API.

```
EpsilonStore  db = new EpsilonStore ("name");
Class h2db =
    Class.forName("org.h2.engine.Engine");
db.captureAll(h2db);
/* schemas names are the keys of
 * a ConcurrentHashMap
 */
ResultSet schemas = db.query(
  "MATCH (db: `org.h2.engine.Database`)"
  +"-[:schemas]->()-[:table]->()-[:key]->(k)"
  +"RETURN k");
```

(b) Getting *schemas* using ϵStore query.

```
ResultSet users = db.query(
   "MATCH (db: `org.h2.engine.Database`)"
   +"-[:usersAndRoles]->()-[:table]->()"
   +"-[:key]->(k) "
   +"RETURN k");
```

(c) Getting *users* using ϵStore query.

```
public ResultInterface getSchemas() {
  return getSchemas(null, null);
}
public ResultInterface getSchemas(String catalog,
  String schemaPattern) {
  checkClosed();
  SimpleResult result = new SimpleResult();
  result.addColumn(
      "TABLE_SCHEM", TypeInfo.TYPE_VARCHAR);
  result.addColumn(
      "TABLE_CATALOG", TypeInfo.TYPE_VARCHAR);
  if (!checkCatalogName(catalog)) {return result;}
  CompareLike schemaLike = getLike(schemaPattern);
  Collection<Schema> allSchemas =
      session.getDatabase().getAllSchemas();
  Value cValue =
      getString(session.getDatabase().getShortName());
  if (schemaLike == null) {
    for (Schema s : allSchemas)
      result.addRow(getString(s.getName()), cValue);
  } else {
    for (Schema s : allSchemas)
      if (schemaLike.test(s.getName()))
        result.addRow(getString(s.getName()),
        cValue);
  }
  // we ignore sorting for a fair comparison
  // result.sortRows(
  // new SortOrder(session, new int[] { 0 }));
  return result;
}
```

(d) H2 implementation for `getSchemas` JDBC API.

Figure 6 Querying metadata of H2. (a) Querying schemas using JDBC `getSchemas` API, (b) Querying schemas of a captured H2 instance with ϵStore, (c) Querying users of a captured H2 instance with ϵStore, and (d) H2's implementation of `getSchemas`.

Figure 6b shows how to get the same result by inserting the embedded H2 database into ϵStore and querying its metadata using Cypher. The idea here is to show how ϵStore can be easily used to implement some API methods in a concise and readable way, allowing developers to quickly experiment with new ideas and move fast. As another example, figure 6c shows how we can query all the users in an H2 database while JDBC only provide an API for getting current username.

4.3 Lightweight in-memory Graph Store

The ability to insert, delete, update and query objects in ϵStore and the support for the Cypher query language makes ϵStore a good candidate for testing when a graph database is needed. We demonstrate in section 5.4 that ϵStore can be used as a light-weight alternative in-place of graph databases by evaluating it on the LDBC SNB benchmark.

5 Evaluation

We evaluated ϵStore in three ways. First, we benchmarked ϵStore on the LDBC SNB [1] benchmark using Neo4j graph database as reference. Second, we re-implemented a number of imperative library methods of data structures using Cypher in ϵStore and compared its performance with OGO. Finally, we compared the query execution times of ϵStore with and without code rewriting, on the LDBC SNB benchmark. We answer the following questions:

RQ1: How does ϵStore perform as a lightweight graph store?
RQ2: How does ϵStore, when used for implementing methods, compare with OGO?
RQ3: How does ϵStore's code rewriting improve its performance?

We describe environment setup (§5.1), existing systems we use as baselines (§5.2), and the benchmarks (§5.3). Finally, we answer the research questions (§5.4-§5.6).

5.1 Experiment Setup

We built a Docker image for each system used in the evaluation (e.g., OGO) to ensure ease of repeatability of our evaluation experiments. All experiments are run inside Docker containers and averaged over 5 runs. We modified each system used in the evaluation to collect the same profile data. We use a single machine to run the experiments; the machine has an x86_64 11th Gen Intel(R) Core(TM) i7-11700K @ 3.60GHz server with 64GB of RAM and running a 64-bit Ubuntu 20.04.1 operating system. We use Java 17 throughout our experiments.

5.2 Existing Systems

We briefly describe the existing systems that we used in our evaluations.

Neo4j. Neo4j [28] is a graph database and arguably, the most popular one in the industry at the moment. It uses the LPG data model. Neo4j has two modes.

- Server Mode (Neo4j^s): In this mode, Neo4j operates as a database server and runs in a JVM separate from the test JVM which contains the benchmarking queries. We use the Neo4j Java driver [35] version 4.3.3 in the test JVM to send the benchmarking query strings to the Neo4j server. The driver implements the Bolt [30] protocol (similar to JDBC) to communicate with the server. We build Neo4j from source inside docker and load it with the LDBC SNB benchmark dataset. The loading of the datasets (CSVs describing nodes and relations) is done using Neo4j's batch import tool `neo4j-admin` [34].
- Impermanent Mode (Neo4j^i): In this mode, all data inserted into the database are stored in-memory and is non-persistent, and the database runs inside the JVM running the LDBC benchmark queries. This mode is only available in internal test-suites of Neo4j. The Docker image used for evaluation is the same as that built for the server mode. We first create an impermanent database by instantiating `GraphDatabaseService` [31] through dependency injection with `ImpermanentDbmsExtension` [32]. We then insert the LDBC SNB benchmark datasets into the database by using `CREATE` Cypher queries to create the corresponding nodes and relations through database transactions.

We include both, Neo4j^s and Neo4j^i as a point of reference in our evaluation of ϵStore on the LDBC SNB benchmark. We use Neo4j version 5.13.0 and default configuration for all modes of Neo4j in our experiments.

OGO. OGO [57], similar to LINQ [42], combines imperative and declarative (via Cypher) styles of programming. Namely, OGO sees the entire JVM heap (i.e., object graph) as a single graph and enables developers to query the heap (or a subset of it) using queries. We compare it with ϵStore for writing methods using queries on several data structures by replacing existing imperative implementations.

Table 3 shows some major differences between the existing systems we used in our evaluation and ϵStore. We categorize the feature differences into *Programmability* and *Database* features. Programmability features broadly include capabilities such as schema creation, querying runtime program state, manipulating objects on the heap and quickly implementing methods of library classes. These are (partially) supported by OGO and ϵStore. However, most graph databases lack all or most of these features. The database features include some

Table 3 Differences between traditional graph databases, OGO and ϵStore.

	Feature	GDBs	OGO	ϵStore
Programmability	Schema creation	✓	✗	✓
	Query program state	✗	✓	✓
	Heap manipulation	✗	✓	✓
	Method implementation	✗	✓	✓
	Code Rewriting	✗	✗	✓
Database features	Views [43]	✓	✗	✓
	Multi-tenancy [8]	✓	✗	✓
	In-memory	✓	✓	✓
	ACID [24]	✓	✗	✗

features found in traditional databases such as support for multiple views, multi-tenancy, in-memory or non-persistent storage, and ACID compliancy. Most graph databases support all or most of these features, while ϵStore focuses on support for multiple views, in-memory and multi-tenancy features. This table serves to highlight the differences in the design of traditional graph databases and ϵStore and thus their area of applicability.

5.3 Benchmarks

This section provides a brief description of the benchmarks used in our evaluation.

ϵStore as a graph store. To evaluate ϵStore as a graph store and answer RQ1, we use the SNB benchmark from LDBC [56]. LDBC provides both, various sized datasets for its benchmarks and the Cypher queries. The size of a dataset is measured using scale factor which is its uncompressed disk space (e.g., an uncompressed dataset that requires 10GB of disk space would have a scale factor of 10). The LDBC SNB was designed with the aim to model a snapshot of the activity in a realistic social network during a period of time. Table 4 shows the nodes and relationships that appear in the LDBC SNB benchmark and their variation with scale factors used in our evaluation namely, 0.1, 0.3, 1, 3 and 10. Higher scale factors can be supported since ϵStore is only limited by the memory available to the JVM which can be increased with the JVM option `-Xmx`. We observe that the frequency of some nodes (`Comment`) and relationships (`Person Likes Comment`) scale by order of magnitude for an order of magnitude increase in scale factor whereas that of others such as `Tagclass` and `Tagclass IsSubclassOf Tagclass` do not change with scale factor. Query execution time is affected by these different occurrence frequencies depending on the node and relationship labels appearing in it. We use all the queries from the LDBC SNB benchmark that are currently supported by ϵStore. Many queries use Cypher language features which are not implemented in ϵStore yet (§6). Table 5 gives a brief description of the used queries. The name of the query as it appears in the LDBC SNB benchmark documentation is shown in column 1. The queries all contain either a read (`MATCH`) clause or a read and an update (`CREATE`) clause. These read and update clauses contain either single node or two-node patterns. The pattern contained in the queries is given by columns 3, 4 and 5. Finally, a brief description of the queries is given in column 6. Generally, ignoring indexing schemes, we should expect the query execution time to scale with the number of operations performed and the frequency of occurrence of labels in its patterns. For example, Q^G_{SNB} contains the most occurring label (`Comment`) in the benchmark in its patterns and 2 clauses, and would be expected to take more time to execute than Q^D_{SNB} that involves less frequently occurring labels and just 1 clause.

Table 4 Node and relationship statistics for LDBC SNB benchmark across evaluated scale factors.

Type	Name	0.1	0.3	1	3	10
Node	Comment	151043	523222	2052169	6413095	21865475
	Forum	13750	31097	90492	221792	595453
	Person	1528	3514	9892	24328	65645
	Post	135701	324825	1003605	2597141	7435696
	Organisation	7955	7955	7955	7955	7955
	Place	1460	1460	1460	1460	1460
	Tag	16080	16080	16080	16080	16080
	Tagclass	71	71	71	71	71
Relationship	Comment **HasCreator** Person	151043	523222	2052169	6413095	21865475
	Comment **HasTag** Tag	191303	680738	2698393	8426418	28740194
	Comment **IsLocatedIn** Place	151043	523222	2052169	6413095	21865475
	Comment **ReplyOf** Comment	76787	265931	1040749	3251228	11089373
	Comment **ReplyOf** Post	74256	257291	1011420	3161867	10776102
	Forum **ContainerOf** Post	135701	324825	1003605	2597141	7435696
	Forum **HasMember** Person	123268	404952	1611869	4982242	17168614
	Forum **HasModerator** Person	13750	31097	90492	221792	595453
	Forum **HasTag** Tag	47697	108649	309766	767382	2065319
	Person **HasInterest** Tag	35475	81066	229166	569918	1535511
	Person **IsLocatedIn** Place	1528	3514	9892	24328	65645
	Person **Knows** Person	14073	44760	180623	565247	1938516
	Person **Likes** Comment	62225	291590	1438418	5281725	19949360
	Person **Likes** Post	47215	177064	751677	2498139	8839875
	Person **StudyAt** Organisation	1209	2792	7949	19497	52632
	Person **WorkAt** Organisation	3313	7697	21654	53023	143553
	Post **HasCreator** Person	135701	324825	1003605	2597141	7435696
	Post **HasTag** Tag	51118	179499	713258	2229757	7599701
	Post **IsLocatedIn** Place	135701	324825	1003605	2597141	7435696
	Organisation **IsLocatedIn** Place	7955	7955	7955	7955	7955
	Place **IsPartOf** Place	1454	1454	1454	1454	1454
	Tagclass **IsSubclassOf** Tagclass	70	70	70	70	70
	Tag **HasType** Tagclass	16080	16080	16080	16080	16080

ϵStore as a heap manipulation engine. To evaluate ϵStore as an engine to modify objects on the heap and answer RQ2, we use data structures from three sources: Java Collections Framework (JCF) [49], Google Guava [21], and the Eclipse Collections [17] projects. We rewrote on average 2 library methods from each of these data structures to use Cypher queries (rather than the imperative implementation). Simply, for ϵStore, we insert the data structure into an instance of ϵStore and run a query that implements the same functionality as exiting imperative code.

5.4 ϵStore as a Lightweight Graph Store (RQ1)

We use query execution time and memory consumption during benchmarking on LDBC SNB benchmark to motivate ϵStore as a lightweight graph store.

Query execution time. In our early experiments, we noticed substantial variations in query execution times across several runs, which we attribute to the JVM environment. To stabilize the time, we perform the following steps. We first load the dataset for a given scale factor into the systems used in our evaluation. Next, we execute all the chosen LDBC queries for that benchmark in a randomized order. We call this as the 1^{st} set. Following this, we once again execute the same set of queries in another randomized order. We call this as the 2^{nd} set. The sets are executed back-to-back in the same JVM process. The profile data for an evaluation run is collected for every query execution in both the sets. However, when

Table 5 Description of the LDBC SNB benchmark queries used in our evaluation.

Query	Abbrv.	Start Node	Relation	End Node	Description
interactive-delete-query2	Q^A_{SNB}	Person	Likes	Post	Matches a pattern with a *Person* node pointing to a *Post* node through a relation *Likes*, deletes the relation and returns the count of *Likes* relation between the two nodes.
interactive-delete-query3	Q^B_{SNB}	Person	Likes	Comment	Matches a pattern with a *Person* node pointing to a *Comment* node through a relation *Likes*, deletes the relation and returns the count of *Likes* relation between the two nodes.
interactive-delete-query5	Q^C_{SNB}	Forum	HasMember	Person	Matches a pattern with a *Forum* node pointing to a *Person* node through a relation *HasMember*, deletes the relation and returns the count of *HasMember* relation between the two nodes.
interactive-short-query1	Q^D_{SNB}	Person	IsLocatedIn	Place	Matches a pattern with a *Person* node pointing to a *Post* node through a relation *IsLocatedIn* and returns properties of the two nodes.
interactive-short-query5	Q^E_{SNB}	Comment	HasCreator	Person	Matches a pattern with a *Comment* node pointing to a *Person* node through a relation *HasCreator* and returns properties of the *Person* node.
interactive-update-query2	Q^F_{SNB}	Person	Likes	Post	Matches a *Person* and *Post*, creates a relation *Likes* from *Person* to *Post* node and returns the count of *Likes* relation between the two nodes.
interactive-update-query3	Q^G_{SNB}	Person	Likes	Comment	Matches a *Person* and *Comment*, creates a relation *Likes* from *Person* to *Comment* node and returns the count of *Likes* relation between the two nodes.
interactive-update-query5	Q^H_{SNB}	Forum	HasMember	Person	Matches a *Forum* and *Person*, creates a relation *HasMember* from *Forum* to *Person* node and returns the count of *HasMember* relation between the two nodes.
interactive-update-query8	Q^I_{SNB}	Person	Knows	Person	Matches a *Person* and *Person*, creates a relation *Knows* from *Person* to *Person* node and returns the count of *Knows* relation between the two nodes.

reporting the profile data for a query for an evaluation run, we use the data from the 2^{nd} set and discard the 1^{st} set. Figure 7 shows a boxplot of the query execution time for the queries in the 1^{st} and 2^{nd} sets for ϵSTORE^r across 10 evaluation runs. It is observable that profile data for the queries collected from the 2^{nd} set has substantially lower variance than that collected from the 1^{st} set, and, hence, the query profile data are more stable across runs.

The collected profile data breaks down the total query execution time ($\mathbf{T_{tot}}$) into the query parsing time ($\mathbf{t_{Pa}}$), query plan generation time ($\mathbf{t_{Pl}}$), and query plan execution time ($\mathbf{t_{Ex}}$). We compute the sum breakdown time ($\mathbf{t_{Bd}}$) from the profile data as t_{Bd}=t_{Pa}+ t_{Pl}+ t_{Ex}. All times are reported in milliseconds unless otherwise stated.

Table 6 shows the results for LDBC SNB. Column 1 shows the query abbreviation. Column 2 shows the scale factor of the dataset. Columns 3-5 show the results for Neo4j^s, Neo4j^i, and ϵSTORE^r respectively. We initially hypothesized that using reflection to retrieve field values may degrade performance due to excessive runtime type-checking. To test our hypothesis, we decided to implement field access using reflection (ϵSTORE^r) and the unsafe API (ϵSTORE^u). However, we observed no significant performance difference between these two modes of ϵSTORE. Hence, for brevity, we omit showing ϵSTORE^u results in Table 6.

For each system, we show the query parsing time (t_{Pa}), query planning time (t_{Pl}), query plan execution time (t_{Ex}), sum breakdown time (t_{Bd}), and the total query execution (T_{tot}). We use bold text for T_{tot}, and we use gray background to show the best value (smallest T_{tot}) in each row. OOM indicates out of memory (when the physical memory requirement exceeds ~64GB). The T_{tot} of queries in general scales with scale factor of the datasets with the exception of some queries (e.g., Q^A_{SNB},Q^B_{SNB}), that operate on node labels containing very

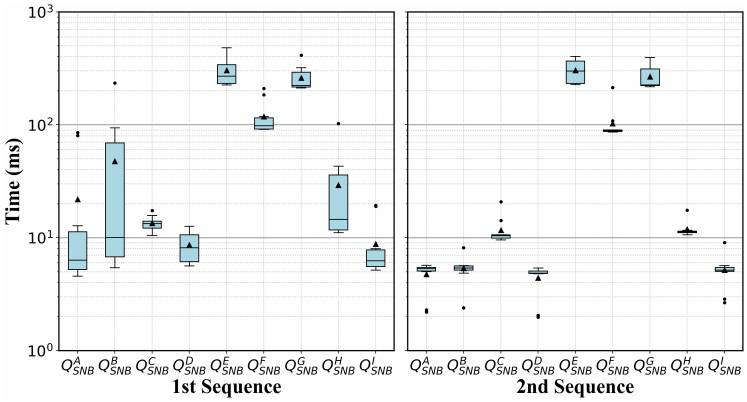
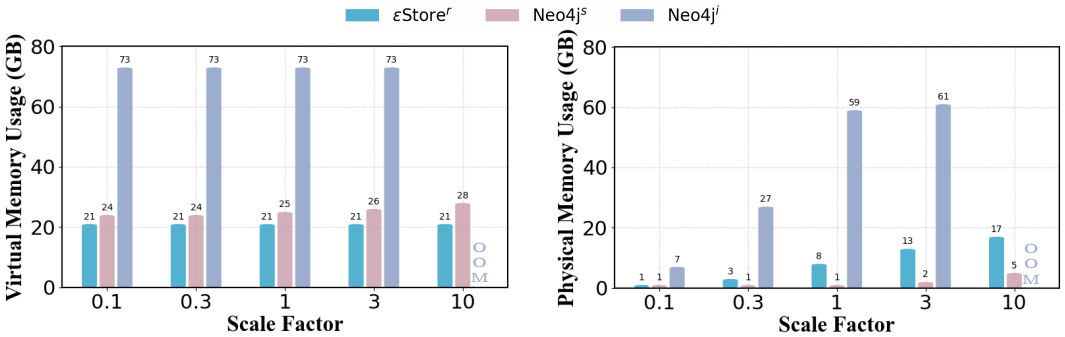


Figure 7 Query execution times for 1st and 2nd randomized sequence runs of ϵSTORE^r.



(a) SNB virtual memory usage in GB.

(b) SNB physical memory usage in GB.

Figure 8 Virtual and Physical memory usage in GB for LDBC SNB evaluation across scale factors. Query failures for exceeding memory are indicated by OOM.

few nodes. We can see that the T_{tot} of ϵSTORE is comparable or better than the production graph database for most of the queries. The graph database outperforms ϵSTORE for query Q^E_{SNB} because, Q^E_{SNB} matches referrer node with label containing the highest amount of nodes in the benchmark (20 million+ for SF 10). This shows that for testing purposes, where datasets are relatively smaller, ϵSTORE can be used as a lightweight graph store instead of a full fledged production graph database.

Memory usage. In addition to time, we also measured memory consumption for all the systems. We used pidstat [39] to collect memory consumption during evaluation runs. Figure 8 shows the peak memory usage for each system (one bar per system). The virtual and physical memory consumption in GB during running all the selected queries on the SNB for different scale factors is shown in Figure 8a and Figure 8b, respectively. We allow Neo4j^i and Neo4j^s to manage their own memory requirements [33], e.g., allocating page cache. We observe that the virtual memory usage of a system does not change significantly across scale factors of the SNB benchmarks. The physical memory usage, on the other hand, increases with increasing scale factors for all the systems. Neo4j^i requires the most physical and virtual memory and is OOM for the SNB scale factor 10 dataset. ϵSTORE^r consumes the least amount of virtual memory across the systems and is second only to Neo4j^s in terms of the least physical memory consumed. This is to be expected since ϵSTORE^r being in-memory,

Table 6 Total query execution time (T_{tot}) and its breakdown in milliseconds, for queries and datasets from the LDBC SNB benchmark. All reported times lower than 0.5 milliseconds are shown as 0. The maximum allocated physical memory for each evaluation run is 63GB.

Query	SF	Neo4j^s					Neo4j^i					ϵStore^r				
		t_{Pa}	t_{Pl}	t_{Ex}	t_{Bd}	T_{tot}	t_{Pa}	t_{Pl}	t_{Ex}	t_{Bd}	T_{tot}	t_{Pa}	t_{Pl}	t_{Ex}	t_{Bd}	T_{tot}
Q^A_{SNB}	0.1	11	29	0	40	**46**	1	33	3	37	**38**	0	0	0	0	**0**
	0.3	11	28	0	39	**44**	1	33	5	39	**40**	0	1	1	2	**3**
	1	10	24	0	34	**38**	0	31	24	55	**55**	0	0	1	1	**1**
	3	12	24	0	36	**40**	2	32	40	75	**76**	0	1	4	4	**5**
	10	11	23	0	35	**39**			OOM			0	0	4	5	**5**
Q^B_{SNB}	0.1	10	24	0	34	**39**	0	32	2	35	**36**	0	0	0	0	**0**
	0.3	12	27	0	39	**43**	0	33	5	38	**39**	0	0	1	1	**2**
	1	11	26	0	37	**41**	1	34	9	44	**45**	0	0	0	1	**1**
	3	10	23	0	33	**37**	0	33	52	86	**87**	0	1	4	5	**6**
	10	11	24	0	35	**39**			OOM			0	0	4	4	**4**
Q^C_{SNB}	0.1	11	25	0	36	**41**	0	32	2	34	**35**	0	0	1	1	**1**
	0.3	10	27	0	37	**41**	0	32	4	37	**37**	0	0	3	3	**4**
	1	10	24	0	34	**38**	0	30	18	49	**50**	0	0	3	3	**3**
	3	10	24	0	34	**38**	0	31	21	52	**53**	0	0	8	8	**9**
	10	11	25	0	36	**41**			OOM			0	0	19	19	**19**
Q^D_{SNB}	0.1	16	30	0	46	**51**	1	34	0	36	**36**	0	0	0	0	**1**
	0.3	15	30	0	45	**50**	1	34	0	35	**36**	0	1	2	2	**4**
	1	13	25	0	38	**43**	0	34	0	34	**35**	0	0	1	1	**1**
	3	16	26	0	43	**47**	3	34	0	37	**38**	0	1	2	3	**4**
	10	15	26	0	42	**46**			OOM			0	0	4	4	**5**
Q^E_{SNB}	0.1	10	23	0	33	**38**	1	30	0	31	**31**	0	0	7	7	**7**
	0.3	11	21	0	32	**36**	2	34	0	36	**37**	0	0	29	29	**30**
	1	11	22	0	34	**38**	4	29	0	33	**33**	0	0	114	115	**115**
	3	11	20	0	31	**36**	6	29	0	36	**37**	0	1	319	320	**321**
	10	11	20	0	31	**35**			OOM			0	0	866	866	**867**
Q^F_{SNB}	0.1	10	15	29	54	**58**	0	24	44	67	**68**	0	0	5	6	**6**
	0.3	10	16	66	92	**97**	1	25	107	133	**134**	0	1	15	16	**16**
	1	9	16	212	237	**241**	1	24	286	311	**312**	0	0	35	35	**35**
	3	8	16	530	554	**559**	0	23	754	777	**778**	0	0	87	87	**88**
	10	11	18	1531	1559	**1564**			OOM			0	0	258	258	**259**
Q^G_{SNB}	0.1	9	16	31	57	**61**	2	26	59	88	**88**	0	0	6	6	**7**
	0.3	9	16	102	128	**133**	0	24	206	230	**231**	0	0	30	31	**31**
	1	10	15	423	448	**453**	1	24	728	753	**754**	0	0	74	75	**75**
	3	9	15	1298	1322	**1326**	3	25	2484	2511	**2512**	0	0	244	244	**245**
	10	10	15	4393	4418	**4422**			OOM			0	0	915	915	**915**
Q^H_{SNB}	0.1	10	16	3	29	**33**	0	24	7	31	**32**	0	0	1	1	**1**
	0.3	10	16	6	33	**38**	0	24	16	40	**41**	0	1	4	4	**5**
	1	9	16	18	44	**48**	0	23	40	63	**63**	0	0	3	3	**3**
	3	9	16	45	70	**74**	0	23	105	128	**129**	0	1	10	11	**11**
	10	10	16	112	137	**143**			OOM			0	0	22	22	**22**
Q^I_{SNB}	0.1	9	17	2	28	**32**	0	24	3	27	**27**	0	0	0	0	**0**
	0.3	10	16	3	29	**34**	0	24	6	30	**31**	0	1	1	2	**2**
	1	9	16	8	33	**38**	0	24	14	38	**39**	0	0	1	1	**1**
	3	10	16	14	39	**43**	0	24	33	57	**57**	0	1	4	5	**5**
	10	9	15	40	64	**69**			OOM			0	0	6	6	**7**

stores all inserted data on RAM whereas Neo4j^s can store part of it on disk and can page it into RAM as and when required. We once again see that for smaller datasets, which is generally the case during testing, ϵStore's memory consumption is comparable or better than a production graph database.

> In summary, the query execution times and memory consumption of ϵStore is on par or better than that of a production graph database for small datasets. Since, smaller datasets are typically the norm in testing environments, ϵStore provides an excellent light-weight alternative to a full fledged graph database for testing.

5.5 Data-structure Performance (RQ2)

We reuse 5 of the data structures from Thimmaiah et al. [57] that were used to evaluate OGO. We also introduce 4 additional data structures from the Eclipse Collections project in our evaluation.

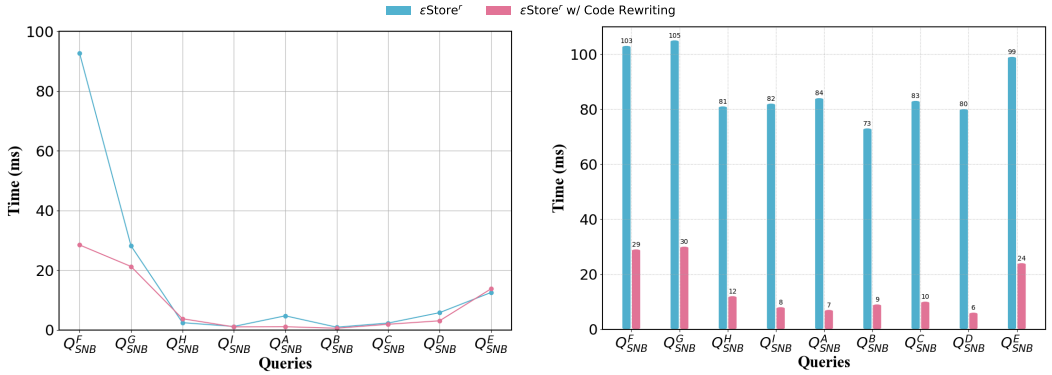
OGO supports two modes of operation, OGO^{Neo} and OGO^{Mem}. In both modes, the first step is to identify a subset of the heap that is relevant to the query. The 2 modes differ in the query engine used. OGO^{Neo} uses an external query engine (Neo4j) and OGO^{Mem} uses an in-memory query engine. We compared the 2 different modes of OGO with ϵStore by executing Cypher queries implementing imperative methods from data-structure libraries. We evaluated these queries for varying workloads (number of elements present inside the data structure). The results are shown in Table 7. We only report the total query execution time T_{tot} (in milliseconds) for each system. The data-structures are given in the first column.

Table 7 Total query execution time (T_{tot}) in milliseconds, for the *contains* or equivalent method reimplemented as Cypher query, on data structures for different modes of OGO and ϵStore. The time out (TO) duration used is 1 minute .

Data-structure	#Elements	OGO^{Neo}	OGO^{Mem}	ϵStore^{r}	ϵStore^{u}
JCF v17.0					
ArrayList	10^2	2231	264	89	81
	10^3	8169	486	97	90
	10^4	40261	888	133	116
	10^5	TO	TO	166	162
ArrayDeque	10^2	1632	399	88	86
	10^3	3377	497	105	89
	10^4	41301	1540	128	120
	10^5	TO	TO	156	166
HashMap	10^2	2391	315	78	77
	10^3	9479	516	88	100
	10^4	TO	2753	136	126
	10^5	TO	TO	356	394
LinkedList	10^2	24831	420	85	79
	10^3	TO	TO	83	88
	10^4	TO	TO	125	133
	10^5	TO	TO	232	216
Guava v32.1.3-jre					
ArrayTable	10^2	1711	358	82	76
	10^3	2742	415	94	90
	10^4	23298	1031	125	140
	10^5	TO	TO	237	228
Eclipse v11.1.0					
UnifiedSet	10^2	1592	514	78	76
	10^3	2916	595	87	90
	10^4	55464	3373	158	149
	10^5	TO	TO	223	239
UnifiedMap	10^2	1997	489	88	83
	10^3	5079	865	102	109
	10^4	TO	TO	111	113
	10^5	TO	TO	210	200
FastList	10^2	1637	453	80	77
	10^3	2750	540	89	87
	10^4	40243	1255	130	143
	10^5	TO	TO	172	161
ArrayStack	10^2	1820	411	78	82
	10^3	3538	741	87	84
	10^4	TO	2461	127	129
	10^5	TO	TO	153	153

The second column shows the workload, and finally, columns three through six give the total query execution times for OGO and ϵStore. We fixed the query execution time out (TO) duration to be 1 minute. We clearly see that both the modes of ϵStore consistently outperform those of OGO by at least an order of magnitude. Larger workloads for most of the considered data structures result in TO for OGO.

OGO^{Neo} is significantly slower than ϵStore because of using an external query engine which incurrs a heavy overhead due to repeated serialization and deserialization of the heap subset. This overhead increases for higher workloads due to increase in the heap subset size. ϵStore is also faster than OGO^{Mem} by an order of magnitude on average. This is primarily due to 3 factors. The first is that both the modes of OGO rely on tagging [45] of objects (assigning a `long` identifier) in the heap to identify those relevant to the query. Since other JVM processes such as the garbage collector (GC) also use tagging, both modes of



(a) Sequence. **(b)** Standalone.

Figure 9 Total query execution time (T_{tot}) for LDBC SNB queries in milliseconds. In (a), each query is executed in sequence inside a single JVM process; in (b), each query is executed in a separate JVM process.

OGO execute every query by first iterating through the entire heap and tagging every object to 0. This tag initialization time grows linearly with the number of objects in the heap. The second factor is, ϵSTORE is implemented purely in Java and thus benefits from Just-In-Time (JIT) compilation whereas OGO^{Mem} uses an in-memory query engine implemented in C++. Finally, the third factor is in the identification of the heap subset in OGO. Both the modes rely on the JVMTI `FollowReferences` [44] method to identify the heap subset. The `FollowReferences` method takes a user provided callback as one of its arguments and visits every object starting from JNI roots following its chain of references, reporting them to the callback. Providing class filters to `FollowReferences` only controls what objects are reported but not what objects are visited. This overhead of visiting objects not relevant to the query increases with increase in heap size. This is unlike ϵSTORE which stores references to the objects that might be queried.

We only compare total query execution time and not lines of code (LOC) since, both OGO and ϵSTORE use Cypher to implement the data-structure methods.

> In summary, ϵSTORE outperforms both the modes of OGO for Cypher queries re-implementing imperative methods from data-structure libraries. OGO is on-average an order of magnitude slower or worse. This is due to ϵSTORE benefitting from JIT compilation due to its pure Java implementation and storing references to all the queryable objects. OGO, on the other hand, incurrs heavy overhead due to its need to identify the heap subset relevant to the query for every query execution.

5.6 Imperative Code Rewriting Performance (RQ3)

We evaluate the performance of the code rewriting using LDBC SNB queries on the smallest scale factor dataset (0.1).

Figure 9 shows the evaluation results. In Figure 9a, we run each query sequentially in the same JVM process. (The order to run these queries is determined by the test runner, we observe the same trend with other orders). We observe that the code rewriting technique speeds up the first two queries; and for the subsequent queries the execution time is similar. There are two main reasons for the results. First, we use ANTLR to parse the query and the start up takes time [4], thus the first query of ϵSTORE^r takes much longer time than others. Second, after the warm up, JIT compilation kicks in and the difference becomes negligible.

We further evaluate the execution time when each query runs in a separate JVM process and the result is shown in Figure 9b. This result is consistent with our previous conclusion.

In summary, parsing and generation of query plans adds nearly an order of magnitude to the execution time. We can significantly speed up query execution by removing this overhead by injecting query plan execution code into the query call site at compile time.

6 Limitations and Future Work

We document potential future directions in this section.

Query optimization. Our current implementation of the query engine only generates the physical plan and directly executes it. In the future, we could introduce a logical plan and construct a physical plan from it. This could allow us to reason about query execution strategies at a more abstract level.

ID collisions. ϵSTORE stores references to captured objects using their class names and their IDs. The object's hash code is used as its ID. However, it is possible for two or more distinct objects to share the same hash code. Currently, ϵSTORE does not support retaining multiple objects with identical hash codes; if a newly captured object shares a hash code with an existing object, the existing reference is overwritten by the new one. Although such situations are theoretically possible, we did not encounter them during our evaluation.

Programming languages. We implemented ϵSTORE in Java due to our familiarity with the language. Modifying types of fields or number of fields in Java is hard. Class redefinition would require bytecode modification and loading in the class with a different classloader and then managing two different versions of the same class within a single JVM. This restricts the types of queries we can support in eStore (e.g., we cannot add new edges, we cannot add new node properties etc.). Other languages like Python or Smalltalk might allow eStore implementations to be more flexible and versatile. We leave design of an in-memory object graph store for other languages as future work.

Concurrency semantics. We defined semantics assuming sequential program execution. It would be interesting to define semantics for concurrent programs when an object might be accessed both inside and outside a store (or multiple stores simultaneously). However, that is outside the scope of the current work.

Query languages. ϵSTORE currently supports a subset of Cypher, which is the most popular query language. Future work could explore supporting other known graph query languages, e.g., GraphQL [19], Gremlin, SPARQL, and AQL [5]. Integration with languages that support both imperative and declarative traversals, such as Gremlin, could be especially well suited for ϵSTORE's data representation.

Cache layer for graph databases. The efficient in-memory graph store model of ϵSTORE makes it suitable to be used as a cache layer for persistent graph databases like Neo4j. The new programming style brought by ϵSTORE can further enrich the interoperability between the applications and the graph databases. We leave for future work the exploration of this direction.

7 Related Work

In this section, we cover the most closely related work, which we organize into: (1) graph databases storage, (2) language integrated queries, (3) object relational/graph mappers.

Graph databases storage. Graph databases [3, 60, 2, 9] are a type of NoSQL database. One of the most popular graph databases is Neo4j, which we discussed in this paper. Many other (proprietary) options are available including TigerGraph [58], Neptune [27], Nebula [37], JanusGraph [38], VelocityDB [36], Kùzu [16] and Memgraph [40]. VelocityDB is an in-memory object database integrated with C# and can be extended as a graph database, but it still introduces extra layer(s) of storage model abstraction and uses a specific set of APIs instead of a query language like Cypher.

Language integrated queries. [11, 42, 20, 54] Microsoft LINQ [42] is an integration of query capabilities directly into C# language. LINQ supports various data sources, including collections (e.g., List), SQL database, XML documents, and streams. Unlike LINQ, ϵSTORE is an in-memory graph backed object store. Including an object into ϵSTORE enables queries on it similar to those on data structures using LINQ. Apache Commons OGNL [18] is an open-source Expression Language (EL) for Java. It provides its own expression syntax to navigate and manipulate Java object graphs. However, it is not designed as a graph store, and does not provide the same level of expressiveness as graph query languages. OGO generalizes the idea behind LINQ's data structure queries and enables querying the entire Java heap. Unlike OGO that supports querying the state of the heap, ϵSTORE focuses on implementing an in-memory graph backed object store.

Object relational/graph mappers. Object-relational mapping (ORM) [59] is used to convert data between a (relational) database and the heap. In a way, object relational mapping techniques create an object database that can be directly manipulated within the program. Example of ORM include Hibernate [26]. There are also Object-graph mappers (OGM) for graph databases, such as Neomodel [15] and Renesca [14] for Neo4j. ϵSTORE is a graph backed object store and thus requires no additional mapping into memory objects.

8 Conclusions

We presented ϵSTORE, the first in-memory graph backed object store. ϵSTORE brings a programming paradigm shift, as it equates nodes in a graph with objects on the heap and relations among nodes with reference fields. It uses dynamic class generation and loading to create necessary schema (classes) to represent nodes and their properties. A subset of Cypher is used for querying the store, and each query returns a table of references. Additionally, ϵSTORE can transitively include an object already on the heap into a store, which enables complex queries for data and relations on already existing object graphs. Our evaluation shows the benefit of our approach. Besides being used as an object graph store, we expect that the combination of graph store features, object store features, implementation of a graph as an object graph, and ability to capture object graphs into a store will introduce new programming styles.

References

1 Renzo Angles, János Benjamin Antal, Alex Averbuch, Altan Birler, Peter Boncz, Márton Búr, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba Pey, Norbert Martínez, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, David Püroja, Mirko Spasić, Benjamin A. Steer, Dávid Szakállas, Gábor Szárnyas, Jack Waudby, Mingxi Wu, and Yuchen Zhang. The ldbc social network benchmark, 2024. arXiv:2001.02299.

2 Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, pages 1–40, 2017.

3 Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys*, pages 1–39, 2008. doi:10.1145/1322432.1322433.

4 Antlr4. Save DFA for faster parser start up. Accessed September 6, 2024 from https://github.com/antlr/antlr4/issues/3682, 2024.

5 ArangoDB. Aql documentation. Accessed March 31, 2024 from https://docs.arangodb.com/3.12/aql/, 2024.

6 Andrea Arcuri, Man Zhang, Asma Belhadi, Bogdan Marculescu, Amid Golmohammadi, Juan Pablo Galeotti, and Susruthan Seran. Building an open-source system test generation tool: lessons learned and empirical analyses with evomaster. *Software Quality Journal*, pages 947–990, 2023. doi:10.1007/S11219-023-09620-W.

7 ASM. A Java bytecode engineering library. Accessed March 10, 2024 from https://asm.ow2.io/publications.html, 2024.

8 Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In *ACM SIGMOD International Conference on Management of Data*, pages 1195–1206, 2008. doi:10.1145/1376616.1376736.

9 Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *ACM Computing Surveys*, pages 1–40, 2023.

10 Alan Bundy and Lincoln Wallen. *Breadth-First Search*, pages 13–13. Springer Berlin Heidelberg, 1984.

11 James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. In *International Conference on Functional Programming*, pages 403–416, 2013. doi:10.1145/2500365.2500586.

12 Shigeru Chiba. Load-time structural reflection in java. In *European Conference on Object-Oriented Programming*, pages 313–336, 2000. doi:10.1007/3-540-45102-1_16.

13 Oracle Corporation.. Linkedlist. Accessed March 30, 2024 from https://github.com/openjdk/jdk/blob/jdk-17%2B0/src/java.base/share/classes/java/util/LinkedList.java, 2019.

14 Felix Dietze, Johannes Karoff, André Calero Valdez, Martina Ziefle, Christoph Greven, and Ulrik Schroeder. An open-source object-graph-mapping framework for neo4j and scala: Renesca. In *Availability, Reliability, and Security in Information Systems*, pages 204–218, 2016. doi:10.1007/978-3-319-45507-5_14.

15 Robin Edwards. Neomodel product website. Accessed March 31, 2024 from https://neomodel.readthedocs.io/, 2024.

16 Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. Kùzu graph database management system. In *Conference on Innovative Data Systems Research*, volume 7, pages 25–35, 2023.

17 Eclipse Foundation. Eclipse collections. Accessed March 10, 2024 from https://github.com/eclipse/eclipse-collections, 2024.

18 The Apache Software Foundation. Apache ognl product website. Accessed March 31, 2024 from https://commons.apache.org/dormant/commons-ognl/, 2013.

19 The GraphQL Foundation. Graphql product website. Accessed March 30, 2024 from https://graphql.org/, 2024.

20 George Giorgidze, Torsten Grust, Alexander Ulrich, and Jeroen Weijers. Algebraic data types for language-integrated queries. In *Workshop on Data Driven Functional Programming*, pages 5–10, 2013. doi:10.1145/2429376.2429379.

21 Google. Guava: Google core libraries for Java. Accessed March 10, 2022 from https://github.com/google/guava, 2024.

22 Alastair Green, Martin Junghanns, Max Kießling, Tobias Lindaaker, Stefan Plantikow, and Petra Selmer. opencypher: New directions in property graph querying. In *International Conference on Extending Database Technology*, pages 520–523, 2018. doi:10.5441/002/EDBT.2018.62.

23 H2. H2 product website. Accessed February 25, 2024 from https://www.h2database.com/html/main.html, 2022.

24 Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, pages 287–317, 1983.

25 Graham Hamilton, Rick Cattell, and Maydene Fisher. *Jdbc Database Access with Java: A Tutorial and Annotated Reference*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1997.

26 Hibernate. Hibernate orm product website. Accessed March 31, 2024 from https://hibernate.org/orm/, 2024.

27 Amazon Web Services Inc. Amazon neptune product website. Accessed March 30, 2024 from https://aws.amazon.com/neptune/, 2024.

28 Neo4j Inc. Neo4j product website. Accessed November 10, 2022 from https://neo4j.com/, 2022.

29 Neo4j Inc. Values and types. Accessed March 10, 2024 from https://neo4j.com/docs/cypher-manual/current/values-and-types/, 2022.

30 Neo4j Inc. Bolt protocol documentation. Accessed February 25, 2024 from https://neo4j.com/docs/bolt/current/, 2024.

31 Neo4j Inc. Graphdatabaseservice. Accessed March 10, 2024 from https://github.com/neo4j/neo4j/blob/5.17/community/graphdb-api/src/main/java/org/neo4j/graphdb/GraphDatabaseService.java, 2024.

32 Neo4j Inc. Impermanentdbmsextension. Accessed March 10, 2024 from https://github.com/neo4j/neo4j/blob/5.17/community/community-it/it-test-support/src/main/java/org/neo4j/test/extension/ImpermanentDbmsExtension.java, 2024.

33 Neo4j Inc. Memory configuration. Accessed March 10, 2024 from https://neo4j.com/docs/operations-manual/current/performance/memory-configuration/, 2024.

34 Neo4j Inc. Neo4j-admin import. Accessed March 10, 2024 from https://neo4j.com/docs/operations-manual/current/tutorial/neo4j-admin-import/, 2024.

35 Neo4j Inc. Neo4j java driver. Accessed February 25, 2024 from https://github.com/neo4j/neo4j-java-driver, 2024.

36 VelocityDB Inc. Velocitydb product website. Accessed March 30, 2024 from https://velocitydb.com/, 2024.

37 Vesoft Inc. Nebula graph product website. Accessed March 30, 2024 from https://www.nebula-graph.io/, 2024.

38 JanusGraph. Janusgraph product website. Accessed March 30, 2024 from https://janusgraph.org/, 2024.

39 Canonical Ltd. pidstat - report statistics for linux tasks. Accessed March 10, 2024 from https://manpages.ubuntu.com/manpages/trusty/man1/pidstat.1.html, 2019.

40 Memgraph Ltd. Memgraph product website. Accessed March 30, 2024 from https://memgraph.com/, 2024.

41 Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. Use at your own risk: The Java unsafe API in the wild. In *Object-oriented Programming, Systems, Languages, and Applications*, pages 695–710, 2015. doi:10.1145/2814270.2814313.

42 Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the. net framework. In *ACM SIGMOD International Conference on Management of Data*, pages 706–706, 2006.

43 Shamkant Navathe, Ramez Elmasri, and James Larson. Integrating user views in database design. *Computer*, 19(01):50–62, 1986. doi:10.1109/MC.1986.1663033.

44 Oracle. Java virtual machine tool interface (JVM TI) - followreferences. Accessed November 10, 2022 from https://docs.oracle.com/en/java/javase/17/docs/specs/jvmti.html#FollowReferences, 2022.

45 Oracle. Java virtual machine tool interface (JVM TI) - settag. Accessed November 10, 2022 from https://docs.oracle.com/en/java/javase/17/docs/specs/jvmti.html#SetTag, 2022.

46 Oracle. Method detail: identityhashcode. Accessed March 10, 2024 from https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/System.html#identityHashCode(java.lang.Object), 2023.

47 Oracle. Class (java platform se 17). Accessed March 10, 2024 from https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html, 2024.

48 Oracle. Classloader. Accessed March 10, 2024 from https://docs.oracle.com/javase/8/docs/api/java/lang/ClassLoader.html, 2024.

49 Oracle. Collections framework overview. Accessed March 10, 2024 from https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html, 2024.

50 Oracle. Field. Accessed March 10, 2024 from https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Field.html, 2024.

51 Oracle. Java interface resultset. Accessed Sep 12, 2024 from https://docs.oracle.com/en/java/javase/17/docs/api/java.sql/java/sql/ResultSet.html, 2024.

52 Oracle. Java jdbc api. Accessed February 25, 2024 from https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/, 2024.

53 Felipe Pontes, Rohit Gheyi, Sabrina Souto, Alessandro Garcia, and Márcio Ribeiro. Java reflection api: Revealing the dark side of the mirror. In *International Conference on the Foundations of Software Engineering*, pages 636–646, 2019. doi:10.1145/3338906.3338946.

54 Alex Potanin, James Noble, and Robert Biddle. Checking ownership and confinement. *Concurrency and Computation: Practice and Experience*, 16(7):671–687, 2004. doi:10.1002/CPE.799.

55 Philipp Seifer, Johannes Härtel, Martin Leinberger, Ralf Lämmel, and Steffen Staab. Empirical study on the usage of graph query languages in open source Java projects. In *International Conference on Software Language Engineering*, pages 152–166, 2019. doi:10.1145/3357766.3359541.

56 Gábor Szárnyas, Brad Bebee, Altan Birler, Alin Deutsch, George Fletcher, Henry A. Gabb, Denise Gosnell, Alastair Green, Zhihui Guo, Keith W. Hare, Jan Hidders, Alexandru Iosup, Atanas Kiryakov, Tomas Kovatchev, Xinsheng Li, Leonid Libkin, Heng Lin, Xiaojian Luo, Arnau Prat-Pérez, David Püroja, Shipeng Qi, Oskar van Rest, Benjamin A. Steer, Dávid Szakállas, Bing Tong, Jack Waudby, Mingxi Wu, Bin Yang, Wenyuan Yu, Chen Zhang, Jason Zhang, Yan Zhou, and Peter Boncz. The linked data benchmark council (ldbc): Driving competition and collaboration in the graph data management space. In *Technology Conference on Performance Evaluation and Benchmarking*, 2023.

57 Aditya Thimmaiah, Leonidas Lampropoulos, Christopher J Rossbach, and Milos Gligoric. Object graph programming. In *International Conference on Software Engineering*, pages 216–228, 2024.

58 TigerGraph. Tigergraph product website. Accessed March 30, 2024 from `https://www.tigergraph.com/`, 2024.

59 Alexandre Torres, Renata Galante, Marcelo S. Pimenta, and Alexandre Jonatan B. Martins. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Information and Software Technology*, pages 1–18, 2017.

60 Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In *Annual Southeast regional conference*, pages 1–6, 2010.

Multiparty Asynchronous Session Types: A Mechanised Proof of Subject Reduction

Dawit Tirore
IT University of Copenhagen, Denmark

Jesper Bengtson
IT University of Copenhagen, Denmark

Marco Carbone
IT University of Copenhagen, Denmark

Abstract

Session types offer a type-based approach to describing the message exchange protocols between participants in communication-based systems. Initially, they were introduced in a binary setting, specifying communication patterns between two components. With the advent of multiparty session types (MPST), the typing discipline was extended to arbitrarily many components. In MPST, communication patterns are given in terms of global types, an Alice-Bob notation that gives a global view of how components interact. A central theorem of MPST is subject reduction: a well-typed system remains well-typed after reduction. The literature contains some formulations of MPST with proofs of subject reduction that have later been shown to be incorrect. In this paper, we show that the subject reduction proof of the original formulation of MPST by Honda et al. contains some flaws. Additionally, we provide a restriction to the theory and show that, for this fragment, subject reduction does indeed hold. Finally, we use subject reduction to show that well-typed processes never go wrong. All of our proofs are mechanised using the Coq proof assistant.

2012 ACM Subject Classification Theory of computation → Process calculi; Theory of computation → Type structures; Theory of computation → Program specifications

Keywords and phrases Session types, Multiparty, Mechanisation, Coq

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.31

Supplementary Material *Software (Source Code)*:
`https://github.com/Tirore96/subject_reduction/tree/ECOOP2025/theories`
archived at `swh:1:dir:00e7a131e4038ba8d267fed7d0de9715b819f9de`
Software (ECOOP 2025 Artifact Evaluation approved artifact):
`https://doi.org/10.4230/DARTS.11.2.3`

Funding This work was supported by DFF-Research Project 1 Grant no. 1026-00385A, from The Danish Council for Independent Research for the Natural Sciences (FNU).

1 Introduction

Session types provide a type-based framework for specifying how participants in a concurrent communication-based system exchange messages. These communication patterns, specified in a protocol language, are used for verifying the programs that implement them. The framework guarantees properties such as type safety (well-typed programs never go wrong), session fidelity (processes behave according to protocol descriptions), and in-session deadlock-freedom (protocols never get stuck). Originally, Honda et al. [30, 31] proposed *binary session types* where types (protocol specifications) describe only how pairs of processes exchange messages. Inspired by choreographic languages [9], Honda et al. later extended this concept to *multiparty session types* [32, 33], which generalise the framework to handle protocols with

© Dawit Tirore, Jesper Bengtson, and Marco Carbone;
licensed under Creative Commons License CC-BY 4.0

arbitrarily many participants and asynchronous communication. This seminal work has driven significant progress in both the theory of communication-based systems [6, 11, 19, 39, 49] and their implementation in mainstream programming languages [1, 2, 20, 38].

Except for some cases [13, 35, 51], the correctness results of the theory of multiparty session types rely on paper proofs, which necessitate the omission of numerous details. Known pitfalls, such as incorrect results caused by issues with binders [21, 54], or unsound projection functions [51], often arise from overlooking technical details that are nearly impossible to catch in lengthy paper proofs.

A common way to make sure that all of these details are found, and to ensure that all proofs are correct, is to use interactive proof assistants like Isabelle [44], Coq [41], or Lean [42], to have all proofs machine-checked by a computer. Proof assistants have proven immensely successful and have been used to guarantee the correctness of a wide variety of foundational results in both mathematics [25, 26, 27, 28] and computer science [5, 36]. Nevertheless, attractive as these mechanisations are, they are also very difficult to implement. The original multiparty session types paper [32], recipient of the Most Influential Paper POPL 2008 award [40], should by all means be a prime candidate for mechanisation but has, because of the difficulty of the proofs, mostly been left untouched.

This paper mechanises the subject reduction theorem for the theory of multiparty asynchronous session types, as stated in the journal version of the original paper [33]. Subject reduction states that well-typed programs remain well-typed throughout their execution, as defined by their operational semantics. Subject reduction is one of the core theorems in the meta-theory, supporting key results such as type safety, session fidelity, and in-session deadlock-freedom. Although the proofs of the original paper are quite detailed, they are also lengthy and intricate. During our mechanisation efforts, however, we discovered that subject reduction, as presented in the original work, does not hold. In this paper, we provide a counterexample to the original theorem and present a new type system for which subject reduction does hold. Our proofs are fully machine-checked using the Coq proof assistant.

Multiparty Session Types. Multiparty session types use *global types*, a protocol specification language for describing the interaction patterns among multiple communicating participants. Interactions feature a sender and a receiver, generally called *roles*. For example, the following global type specifies the communication patterns between roles p, q, and r:

$$G = \mathsf{p} \to \mathsf{q}\colon 1\left\{ \begin{array}{l} l_1\colon \mathsf{q} \to \mathsf{r} : 2\langle \mathsf{bool} \rangle.\mathsf{end} \\ l_2\colon \mathsf{p} \to \mathsf{r} : 2\langle \mathsf{bool} \rangle.\mathsf{end} \end{array} \right\} \tag{1}$$

This global type describes a protocol where role p internally decides and informs role q over channel 1, whether to continue in branch l_1 or in branch l_2. If p picks branch l_1, role q, after receiving this label, must send a boolean to r over channel 2. Conversely, if q picks branch l_2, p must then send a boolean to r over the same channel 2. In both branches, the protocol terminates after p or q communicate with r. A channel in the protocol represents a private FIFO queue, shared between the three roles for communication. A global type describes asynchronous protocols, meaning that sends are non blocking, sent messages are immediately placed on a channel, and the sending process can continue executing while the message is in transit. In this example, role q must first receive label l_1 from p before sending a boolean on channel 2. However, if p chooses branch l_2, it can immediately send a boolean without waiting for q to read from channel 1. Global types are equipped with a *reduction semantics* that captures this asynchronous behaviour.

A key observation, relevant for the remainder of this work, is that, as in the original work, a global type dictates exactly which channels the roles communicate over, as can be seen from the explicit references to channels 1 and 2 in the example above. This practice

is often referred to as *explicit channels*, and contrasts with later work where the way roles use channels is implicit and therefore more restrictive. In the implicit channel setting, each pair of roles shares private channels that no other roles can access. Implicit channels are often used in the literature because they offer a convenient abstraction when full control over channel usage is not needed, and formal proofs are significantly simplified. Explicit channels are, however, more expressive and allow fine-grained control over which channels are actually used. This is particularly useful in applications where only a limited number of channels are available and must be shared among all roles. For example, AMQP [3], the Advanced Message Queuing Protocol, is a widely used message-oriented middleware protocol where communication takes place over named queues (channels), and where efficient use of limited shared resources is critical. Encoding implicit channels using explicit ones is straightforward and just a matter of allocating unique channels to all pairs of roles. This means that our results apply to both explicit and implicit channels, and not just to the explicit setting. It is also worth noting that using implicit channels requires allocating a channel (queue) for each ordered pair of roles, resulting in a quadratic number of channels.

While global types provide a holistic view of a protocol, the behaviour of its individual roles can be described as *local types*. Local types provide a local view of the protocol for each of its roles and can be obtained from global types by an operation known as *projection*. For example, the global type from Equation (1) can be expressed as the following collection of local types:

$$\mathsf{p} : 1 \oplus \left\{ \begin{array}{l} l_1 : \mathsf{end} \\ l_2 : !2\langle \mathsf{bool} \rangle.\mathsf{end} \end{array} \right\} \qquad \mathsf{q} : 1\& \left\{ \begin{array}{l} l_1 : !2\langle \mathsf{bool} \rangle.\mathsf{end} \\ l_2 : \mathsf{end} \end{array} \right\} \qquad \mathsf{r} : ?2\langle \mathsf{bool} \rangle.\mathsf{end} \tag{2}$$

The local type for p says that label l_1 or label l_2 must be communicated over channel 1 and if label l_2 is chosen then a boolean must be sent over channel 2. Dually, role q must be ready to receive label l_1 or l_2 on channel 1, and if label l_1 is selected, a boolean must be sent over channel 2. Finally, role r is ready to receive a boolean from channel 2. Similar to global types, collections of local types such as the one seen in Equation (2), are equipped with a semantics that captures the various communications that can be performed.

Protocols described as global types are executed as *sessions*. A session is an instance of a protocol with private channels where each role is implemented as an executable process, written in the multiparty session calculus, a model of computation for concurrent system based on the π-calculus [43]. The core idea of multiparty session types is that global types can be projected into local types which are then, via a type system, used to verify that a given implementation conforms to the specification provided by the global types. For example, multiparty session types ensure that the following parallel composition of processes can participate in a session that implements the global type from Equation (1):

$$\begin{array}{l} \overline{\mathsf{a}}[\mathsf{p}]_2(\mathfrak{s}).\overbrace{\mathfrak{s}[1] \lhd l_2; \mathfrak{s}[2]!\langle \mathsf{true} \rangle; \mathbf{0}}^{P} \quad | \\ \mathsf{a}[\mathsf{q}](\mathfrak{s}).\underbrace{\mathfrak{s}[1] \rhd \{l_1 : \mathfrak{s}[2]!\langle \mathsf{false} \rangle;\ \mathbf{0},\quad l_2 : \mathbf{0}\}}_{Q} \quad | \quad \mathsf{a}[\mathsf{r}](\mathfrak{s}).\underbrace{\mathfrak{s}[2]?(x); \mathbf{0}}_{R} \end{array} \tag{3}$$

The term above reads as follows: three processes are about to initiate a 3-party protocol on a, where a is a *shared name* denoting the name of the protocol in the calculus. The first process, prefixed with $\overline{\mathsf{a}}[\mathsf{p}]_2(\mathfrak{s})$, is requesting the initiation of a session featuring two queues (instantiating the channels specified in the global type), acting as p while the second and third process are performing a session accept, joining the session as roles q and r respectively. The binder $\mathfrak{s}$ is a placeholder for a session identifier that will be created at runtime. Process P denotes a process that, playing role p, first puts a label l_2 in the queue at channel 1 and

then puts the message true in queue at channel 2. Process Q awaits for a label on queue at channel 1 that determines its continuation. Finally, process R, playing role r, receives a value on the queue at channel 2. Initially, we synchronise over the prefixes $\overline{\mathsf{a}}[\mathsf{p}]_2(\mathfrak{s})$, $\mathsf{a}[\mathsf{q}](\mathfrak{s})$, and $\mathsf{a}[\mathsf{r}](\mathfrak{s})$, yielding the term

$$(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})\left(\begin{array}{lll} \overbrace{\mathsf{s}^{\mathsf{p}}[1] \lhd l_2; \mathsf{s}^{\mathsf{p}}[2]!\langle\mathsf{true}\rangle;\mathbf{0}}^{P[\mathsf{s}^{\mathsf{p}}/\mathfrak{s}]} & | & \overbrace{\mathsf{s}^{\mathsf{q}}[1] \rhd \{l_1 : \mathsf{s}^{\mathsf{q}}[2]!\langle\mathsf{false}\rangle; \mathbf{0}, l_2 : \mathbf{0}\}}^{Q[\mathsf{s}^{\mathsf{q}}/\mathfrak{s}]} \quad | \\ \underbrace{\mathsf{s}^{\mathsf{r}}[2]?(x); \mathbf{0}}_{R[\mathsf{s}^{\mathsf{r}}/\mathfrak{s}]} & | \quad \mathsf{s}[1] :: \emptyset \quad | & \mathsf{s}[2] :: \emptyset \end{array}\right) \tag{4}$$

The operation generates the new session identifier s (the syntax $(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})$ is a restriction binder), which we use to replace the binder $\mathfrak{s}$ with *located session identifiers* s^{p}, s^{q} and s^{r} for roles p, q, and r respectively. The two communication queues, represented by $\mathsf{s}[1]$ and $\mathsf{s}[2]$, where messages can be enqueued and dequeued, are shared between all participants and are initially empty. Role p, implemented by process $P[\mathsf{s}^{\mathsf{p}}/\mathfrak{s}]$, first sends label l_2 over channel 1 and then value true over channel 2, and reduces to the following term:

$$(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})\Big(\mathbf{0} \quad | \quad Q[\mathsf{s}^{\mathsf{q}}/\mathfrak{s}] \quad | \quad R[\mathsf{s}^{\mathsf{r}}/\mathfrak{s}] \quad | \quad \mathsf{s}[1] :: l_2{}^{\mathsf{p}} \cdot \emptyset \quad | \quad \mathsf{s}[2] :: \mathsf{true}^{\mathsf{p}} \cdot \emptyset\Big) \tag{5}$$

where the label $l_2{}^{\mathsf{p}}$ and the value $\mathsf{true}^{\mathsf{p}}$, both annotated with the sending role p, have been enqueued on $\mathsf{s}[1]$ and $\mathsf{s}[2]$ respectively.

A Counter-Example to Subject Reduction. Subject reduction states that any term typed by a collection of local types (referred to as the *typing environment*), derived from a global type through projection, will remain typable after the process reduces. Since local types are equipped with a reduction semantics, the subject reduction theorem also states that the term obtained after reduction can be typed either by the same typing environment or by a modified environment, reflecting a reduction of local types. The original statement of subject reduction by Honda et al. can be informally stated as follows:

> *If P is well typed in a typing environment Δ, and P reduces to P' then either P' is well-typed in the original typing environment Δ, or in another environment Δ' such that Δ can reduce to Δ'*

The reduction on the environment Δ is crucial because it establishes a connection between the reductions performed by the processes and the protocol specification defined by the local types (which are derived from global types). Proving other results, such as session fidelity (processes behave according to protocol descriptions), heavily relies on the relationship between the reductions of processes and types. Unfortunately, the original theory by Honda et al. does not satisfy subject reduction. The key issue lies in the fact that some well-typed processes can reduce to well-typed terms in a way where their typing environments are not the same nor does the initial environment reduce to the latter. We demonstrate this with our running example.

Even though the process in Equation (3) has been reduced to the one in (4) and then to (5), this new process is still typable by the same local types in Equation (2), derived by means of projection from the global type in Equation (1). This is because the semantics of environments tracks interactions, which matches an output action on a channel with an input action on the same channel. So far only outputs have been performed, and the environment is thus unchanged. However, when process $R[\mathsf{s}^{\mathsf{r}}/\mathfrak{s}]$ (which is equal to $\mathsf{s}^{\mathsf{r}}[2]?(x);\mathbf{0}$) retrieves message $\mathsf{true}^{\mathsf{p}}$ from $\mathsf{s}[2]$, the term in Equation (5) reduces to

$$(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})\Big(\mathbf{0} \quad | \quad Q[\mathsf{s}^{\mathsf{q}}/\mathfrak{s}] \quad | \quad \mathbf{0} \quad | \quad \mathsf{s}[1] :: l_2{}^{\mathsf{p}} \cdot \emptyset \quad | \quad \mathsf{s}[2] :: \emptyset\Big) \tag{6}$$

which reveals a problem. Here, the process $\mathsf{s}^{\mathsf{r}}[2]?(x); \mathbf{0}$, acting as role r, has consumed the boolean $\mathsf{true}^{\mathsf{p}}$ from the $\mathsf{s}[2]$ queue while ${l_2}^{\mathsf{p}}$ is still in the $\mathsf{s}[1]$ queue. In other words, the branching interaction between roles p and q described by the global type in Equation (1) has been executed out of order, since q has not yet received l_2, but the communication of a boolean from p to r in the l_2 branch has been completely performed. The problem with the original theory of multiparty session types is that the asynchronous semantics of processes allows $Q[\mathsf{s}^{\mathsf{q}}/\mathsf{s}]$ and $R[\mathsf{s}^{\mathsf{r}}/\mathsf{s}]$ to read from queues $\mathsf{s}[1]$ and $\mathsf{s}[2]$ respectively in any order. However, the semantics of the global type from Equation (1) requires that q (corresponding to process $Q[\mathsf{s}^{\mathsf{q}}/\mathsf{s}]$) reads from channel 1 (corresponding to queue $\mathsf{s}[1]$) before r (corresponding to process $R[\mathsf{s}^{\mathsf{r}}/\mathsf{s}]$) reads from channel 2 (corresponding to queue $\mathsf{s}[2]$).

Below (left) we show a global type that describes the remaining behaviour of the session, and we show (right) the environment derived from it.

$$\mathsf{p} \to \mathsf{q} : 1\left\{ \begin{array}{l} l_1 : \mathsf{end} \\ l_2 : \mathsf{end} \end{array} \right\} \qquad \mathsf{p} : 1 \oplus \left\{ \begin{array}{l} l_1 : \mathsf{end} \\ l_2 : \mathsf{end} \end{array} \right\}, \quad \mathsf{q} : 1 \& \left\{ \begin{array}{l} l_1 : \mathsf{end} \\ l_2 : \mathsf{end} \end{array} \right\}, \quad \mathsf{r} : \mathsf{end} \tag{7}$$

This environment types the process in Equation (6). The problem is that the two environments in Equations (2) and (7) are neither the same, nor can they be related by the semantics of local types, therefore breaking subject reduction.

A reasonable question to ask at this point is why, given how influential the work by Honda et al. is, has this not been noticed before? The answer is most likely twofold. Firstly, the proofs are very complex and have never been mechanised. Secondly, the original formulation of multiparty session types differs from most later work. In particular, they use explicit channels where all roles in a session share queues, as opposed to implicit channels, introduced by Bettini et al. [4], where each ordered pair of roles has a dedicated queue. Our counterexample hinges on the use of explicit channels since it would not have been possible to derive it with implicit channels.

This Work. We develop a theory of multiparty session types that mostly follows the original framework by Honda et al [32]. Our main results are threefold. First, as described above, we identify a counterexample to subject reduction, highlighting a flaw in the original meta-theory of multiparty asynchronous session types. Second, we fix this flaw and prove subject reduction of the amended calculus and its type system. Finally, we mechanise all of our results in the Coq proof assistant which significantly reduces the risk of errors on our part. We gradually introduce the prerequisites of our subject reduction proof throughout this paper.

Section 2 defines a process language that models communicating programs. These programs feature different types of communication operations, all based on the concept of sessions. They can start new sessions and subsequently use session channels to communicate within these sessions. The process language is formally defined as a process algebra, equipped with a standard notion of structural equivalence and reduction semantics.

Next, in Section 3, we define global and local types. Global types give a global abstraction of how a set of processes should behave within a given session. To describe how individual processes within a session should behave, global types can be transformed, by means of projection, into a set of local types (one for each process in the session), where each local type represents the behaviour of a specific process in a session. Both global and local types are equipped with a reduction semantics which specifies the behaviour enforced by the types. Using these reduction semantics, we state a projection theorem that establishes the correctness of the projection from global types to local types. However, due to the flexible definition of global types, some global types are not realisable, meaning that their semantics

does not correspond to that of their projection. To address this, we introduce the linearity property, which characterises those global types that can be correctly realised, similarly to the approach in the original framework [32, 33].

The last part of this article, in Sections 4 and 5, focuses on defining the type system for which we prove subject reduction. This does not follow the original framework precisely. In particular, our typing of queues (processes communicate via FIFO queues, one for each channel) takes a different approach which, despite being equivalent, facilitates the mechanisation work. For subject reduction to hold, we need to have an extra restriction on global types, which we call unstuckness. This restriction is what avoids bad situations such as that in the example above to happen. In addition to subject reduction, we also prove communication safety (type safety), which guarantees linearity – ensuring that there is never a race on a session channel – and error freedom in communications.

All proofs have been machine-checked in the Coq proof assistant, marking this as the first mechanisation of subject reduction for a general framework of asynchronous multiparty session types with explicit channels. Definitions and results in the paper are linked to the Coq code in our online repository. These links are marked with the symbol .

2 Multiparty Asynchronous Session Processes

Our process language is an extension of the π-calculus with the notion of session which forms the foundation for inter-process communication.

Syntax. Let $\mathscr{P}$ be a finite subset of the natural numbers which denotes a set of roles (ranged over by $\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{s}, \mathsf{t}$), let $\mathscr{L}$ be a set of labels (ranged over by l), and let I, J denote index sets of the form $\{1, \ldots, n\}$ for any natural number n. Moreover, let indices in an index set be denoted by i, j. We identify two distinct sets of atoms, *shared names*, ranged over by $\mathsf{a}, \mathsf{b}, \mathsf{c}$, and *session identifiers*, ranged over by $\mathsf{s}, \mathsf{t}, \mathsf{r}$. Note that both shared names and session identifiers are in sans-serif font. Multiparty asynchronous session processes are defined by the following grammar ( ):

$a ::= x \mid \mathsf{a}$	(shared names)	$s ::= \mathfrak{s} \mid \mathsf{s}^{\mathsf{p}}$	(session channels)
$v ::= \mathsf{a} \mid \mathsf{true} \mid \mathsf{false}$	(values)	$e ::= x \mid v \mid e \ \mathsf{and}\ e'$	(expressions)
$h ::= l^{\mathsf{p}} \mid v^{\mathsf{p}} \mid (\mathsf{s}^{\mathsf{q}})^{\mathsf{p}}$	(message)	$D ::= \{X_i(x\tilde{\mathfrak{s}}) = P_i\}_{i \in I}$	(declarations)

$P ::= \overline{a}\,[\mathsf{n}]_k(\mathfrak{s}).P$	(session request)	$\mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$	(conditional branch)
$\mid a[\mathsf{p}](\mathfrak{s}).P$	(session acceptance)	$\mid P \mid Q$	(parallel composition)
$\mid s[k]!\langle e\rangle; P$	(value sending)	$\mid \mathbf{0}$	(inaction)
$\mid s[k]?(x); P$	(value reception)	$\mid (\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})P$	(name hiding)
$\mid s[k]!\langle\langle t\rangle\rangle; P$	(session delegation)	$\mid (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P$	(session hiding)
$\mid s[k]?((\mathfrak{s})); P$	(session reception)	$\mid X\langle e, \tilde{s}\rangle$	(process call)
$\mid s[k] \lhd l; P$	(label selection)	$\mid \mathsf{s}[k] :: \tilde{h}$	(message queue)
$\mid s[k] \rhd \{l_i : P_i\}_{i \in I}$	(label branching)		

A shared name non-terminal $a, b, c, \ldots$ can be either a value variable, denoted by $x, y, z, \ldots$, or a shared name a. Located session non-terminals, denoted by $s, t, r, \ldots$, can be either a located session variable, denoted by $\mathfrak{s}, \mathfrak{t}, \mathfrak{r}, \ldots$, or a located session, denoted by $\mathsf{s}^{\mathsf{p}}, \mathsf{t}^{\mathsf{p}}, \mathsf{r}^{\mathsf{p}}$, where the roles ($\mathsf{p}$, q, and r) denote which role is currently using the session, i.e. which location it is being used at. Figure 1 gives an overview of the various names.

A multi-cast session request $\overline{a}\,[\mathsf{n}]_k(\mathfrak{s}).P$ requests to start a session on a, with roles $\{0, ..., \mathsf{n}\}$ and the requester acting as n. When a session eventually is initiated on a concrete shared name a, a fresh session identifier s is created. The requester acts as n by referencing variable

a	shared name	x	value variable	a	shared name non-terminal
s^{p}	located session	$\mathfrak{s}$	located session variable	s	located session non-terminal
s	session identifier	k	channel		

Figure 1 Naming of key elements.

channel $\mathfrak{s}$ in P which, during the execution, is substituted for a located session s^{n}. The natural number k declares how many channels (queues) will be created when the session is initiated and each queue is addressed as $\mathsf{s}[i]$ for $i \in \{1, ..., k\}$. The term $a[\mathsf{p}](\mathfrak{s}).P$ denotes the dual process that accepts a request on a, playing role p.

The construct $s[k]!\langle e\rangle; P$ denotes the sending of expression e (after its evaluation to a value) to the queue of session s addressed at $\mathsf{s}[k]$. The construct $s[k]?(x); P$ reads from $\mathsf{s}[k]$, with variable x binding the received value in the continuation P. The construct $s[k]!\langle\!\langle t\rangle\!\rangle; P$ denotes sending a session channel, an operation called delegation. The construct $s[k]?((\mathfrak{s})); P$ denotes receiving a session channel, an operation called session reception. In session reception, we bind a located session variable $\mathfrak{s}$ in the continuation P, similarly to the request and accept constructs. Branching communications are captured by $s[k] \lhd l; P$ and $s[k] \rhd \{l_i\colon P_i\}_{i\in I}$. The former selects a branch by communicating label l, while the latter reacts to a branch that another process has chosen (external choice).

The terms $(\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})P$ and $(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P$ respectively bind shared names and session identifiers in P. Free shared names and free session identifiers, denoted by $\mathsf{fn}(Q)$ and $\mathsf{fs}(S)$ respectively, are defined in a standard way. The process call $X\langle e, \tilde{s}\rangle$ invokes procedures stored in the declarations D. Declarations are identified by a name X, ranging over some set of declaration names, and have the form $X(x\tilde{\mathfrak{s}}) = P$, where the parameters x and $\tilde{\mathfrak{s}}$ are instantiated with a value and a list of distinct located sessions, respectively. The body P may include process names, thereby supporting recursion. The terminated process is denoted by $\mathbf{0}$.

A queue addressed at $\mathsf{s}[k]$ with messages $\tilde{h}$ is denoted by $\mathsf{s}[k] :: \tilde{h}$, and a message h is either a label, a value, or a session name. A message contains an annotation of the role that has sent the message, e.g., $\mathsf{s}[k] :: \mathsf{true}^{\mathsf{p}} \cdot \tilde{h}$ contains a message from p. Syntactically, variables cannot be messages in a queue. This is sensible because the semantics introduced next enforces that all variables are substituted for concrete terms before being put on a queue.

We define two types of *substitution*: one on value variables and one on located session variables. Both are denoted by the standard syntax $P[\mathsf{a}/x]$ and $P[\mathsf{s}^{\mathsf{p}}/t]$.

Reduction Semantics. We now give a reduction semantics for processes. First, we must define the structural congruence relation.

▶ **Definition 1** (Structural Congruence 🐓). *Structural congruence, denoted by $P \equiv Q$, is the smallest equivalence relation that satisfies the following rules:*

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

$$(\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})\ \mathbf{0} \equiv \mathbf{0} \qquad (\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})P \mid Q \equiv (\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})(P \mid Q) \quad \textit{when } \mathsf{a} \notin \mathsf{fn}(Q)$$

$$(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})\ \mathbf{0} \equiv \mathbf{0} \qquad (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P \mid Q \equiv (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})(P \mid Q) \quad \textit{when } \mathsf{s} \notin \mathsf{fs}(Q)$$

$$\frac{P \equiv Q}{(\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})P \equiv (\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})Q} \qquad \frac{P \equiv Q}{(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P \equiv (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})Q}$$

$$
\begin{array}{rl}
\text{[Link]} & \Big(\prod_{\mathsf{i}=0}^{\mathsf{n}-1} \mathsf{a}[\mathsf{i}](\mathfrak{z}).P_{\mathsf{i}}\Big) \mid \overline{\mathsf{a}}\,[\mathsf{n}]_k(\mathfrak{z}).P_{\mathsf{n}} \;\to_D\; (\boldsymbol{\nu}\,\mathsf{s})\Big(\prod_{\mathsf{i}=0}^{\mathsf{n}-1} P_{\mathsf{i}}[\mathsf{s}^{\mathsf{i}}/\mathfrak{z}] \mid \prod_{j=1}^{k} \mathsf{s}[j] :: \epsilon\Big) \\
\text{[Send]} & \mathsf{s}^{\mathsf{p}}[k]!\langle e\rangle; P \mid \mathsf{s}[k] :: \tilde{h} \;\to_D\; P \mid \mathsf{s}[k] :: \tilde{h} \cdot v^{\mathsf{p}} \qquad \boxed{e \downarrow v} \\
\text{[Recv]} & \mathsf{s}^{\mathsf{p}}[k]?(x); P \mid \mathsf{s}[k] :: v^{\mathsf{q}} \cdot \tilde{h} \;\to_D\; P[v/x] \mid \mathsf{s}[k] :: \tilde{h} \\
\text{[Deleg]} & \mathsf{s}^{\mathsf{p}}[k]!\langle\!\langle \mathsf{t}^{\mathsf{q}} \rangle\!\rangle; P \mid \mathsf{s}[k] :: \tilde{h} \;\to_D\; P \mid \mathsf{s}[k] :: \tilde{h} \cdot (\mathsf{t}^{\mathsf{q}})^{\mathsf{p}} \\
\text{[SRec]} & \mathsf{s}^{\mathsf{p}}[k]?((\mathfrak{z})); P \mid \mathsf{s}[k] :: (\mathsf{t}^{\mathsf{r}})^{\mathsf{q}} \cdot \tilde{h} \;\to_D\; P[\mathsf{t}^{\mathsf{r}}/\mathfrak{z}] \mid \mathsf{s}[k] :: \tilde{h} \\
\text{[Label]} & \mathsf{s}^{\mathsf{p}}[k] \lhd l; P \mid \mathsf{s}[k] :: \tilde{h} \;\to_D\; P \mid \mathsf{s}[k] :: \tilde{h} \cdot l^{\mathsf{p}} \\
\text{[Branch]} & \mathsf{s}^{\mathsf{p}}[k] \rhd \{l_i \colon P_i\}_{i \in I} \mid \mathsf{s}[k] :: l_j{}^{\mathsf{q}} \cdot \tilde{h} \;\to_D\; P_j \mid \mathsf{s}[k] :: \tilde{h} \qquad \boxed{j \in I} \\
\text{[IfT]} & \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \;\to_D\; P \qquad \boxed{e \downarrow \mathsf{true}} \\
\text{[IfF]} & \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \;\to_D\; Q \qquad \boxed{e \downarrow \mathsf{false}} \\
\text{[Def]} & X\langle e, \tilde{\mathsf{s}}^{\mathsf{p}}\rangle \;\to_D\; P[v/x][\tilde{\mathsf{s}}^{\mathsf{p}}/\tilde{\mathfrak{z}}] \qquad \boxed{e \downarrow v \text{ and } \big(X(x\tilde{\mathfrak{z}}) = P\big) \in D} \\
\text{[Res}_{\boldsymbol{\nu}^{\mathsf{s}}}\text{]} & P \;\to_D\; P' \;\Rightarrow\; (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P \;\to_D\; (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P' \\
\text{[Res}_{\boldsymbol{\nu}^{\mathsf{a}}}\text{]} & P \;\to_D\; P' \;\Rightarrow\; (\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})P \;\to_D\; (\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})P' \\
\text{[Par]} & P \;\to_D\; P' \;\Rightarrow\; P \mid Q \;\to\; P' \mid Q \\
\text{[Str]} & P \equiv P' \text{ and } P' \;\to_D\; Q' \text{ and } Q' \equiv Q \;\Rightarrow\; P \;\to_D\; Q
\end{array}
$$

Figure 2 Reduction semantics for processes ( ).

The first line makes processes a commutative monoid over parallel composition with $\mathbf{0}$ as the unit element. The next two lines give the structural rules for name restriction, allowing the binder to be dropped when its scope is a terminated process, and allowing the scope to be extruded when this is capture-free. The rules in the last line ensure that $\equiv$ is a congruence with respect to the two restriction operators. Whilst this is not strictly necessary for the semantics, it will prove useful when we formulate communication safety in Section 5.

The reduction semantics for processes, denoted by $P \;\to_D\; P'$, is defined by the rules given in Figure 2. Rule [Link] initiates a session between $\{0, \ldots, \mathsf{n}\}$ roles. The request is made by n and the remaining processes accept on the same shared name a, declaring with $[\mathsf{i}]$ which role of the session they will perform as. The requester indicates by k how many queues will be initiated for the session. This information does not need to be shared with the other roles because the type system will ensure queue addresses $\mathsf{s}[k]$ are used correctly. The parallel composition of the accepting processes is defined using $\prod$ to denote parallel composition of a set of processes. We assume that the natural number n is greater than 0, i.e., there are at least two processes engaging in a session.

Rule [Send] puts a value at the end of a queue, annotating the value with the role p of the located session s^{p} it was sent on, and [Recv] reads from the front of the queue. Rules [Deleg] and [SRec] are the corresponding rules for delegation and session reception while [Label] and [Branch] are the corresponding rules for labels. Rules [IfT] and [IfF] are standard semantics for if-statements. Rule [Def] handles recursive calls by name by replacing the process variable X with its declaration in D, substituting x for the evaluation of e, and substituting $\tilde{\mathfrak{z}}$ for a list of located sessions. Rules [Res$_{\boldsymbol{\nu}^{\mathsf{s}}}$], [Res$_{\boldsymbol{\nu}^{\mathsf{a}}}$] and [Par] allow reductions to occur under restriction binders and parallel composition. Rule [Str] handles structural congruence.

▶ Remark 2 (Differences with the original work). We annotate a session identifier s with a role p, yielding a located session s^{p}. Such annotations have become a common practice in the literature. They were introduced as *polarities* for binary session types [22, 54], which address unintended limitations of the semantics. As in previous work [4, 17], we annotate messages with the sending role to indicate the sender of the message. This allows us to compare the specification of p against the messages it has enqueued and its remaining actions.

3 Global Types, Local Types, and Projection

Syntax. The syntax of global and local types is given as follows (🐓):

$$
\begin{array}{rcll}
G & ::= & \mathsf{p} \to \mathsf{q} : k\langle U\rangle.G \mid \mathsf{p} \to \mathsf{q} : k\{l_j : G_j\}_{j\in J} \mid \mu\mathbf{t}.G \mid \mathbf{t} \mid \mathsf{end} & \\
T & ::= & !k\langle U\rangle.T \mid ?k\langle U\rangle.T \mid k \oplus \{l_i : T\}_{i\in I} \mid k \,\&\, \{l_i : T_i\}_{i\in I} \mid \mu\mathbf{t}.T \mid \mathbf{t} \mid \mathsf{end} & \\
U & ::= & \mathsf{bool} \mid \mathsf{int} \mid G \mid T & \qquad \hat{\Delta} ::= \emptyset \mid \hat{\Delta},\ \mathsf{p} : T
\end{array}
$$

The global type $\mathsf{p} \to \mathsf{q} : k\langle U\rangle.G$ specifies a session with an interaction where p sends a message of type U to q via channel k, and G specifies the remaining session. The type $\mathsf{p} \to \mathsf{q} : k\{l_j : G_j\}_{j\in J}$ specifies a branching interaction where p sends a label l_i to q via k, and G_i specifies the remaining session. The constructs $\mu\mathbf{t}.G$ and $\mathbf{t}$ are used to write tail-recursive specifications such as $\mu\mathbf{t}.\mathsf{p} \to \mathsf{q} : k\langle U\rangle.\mathbf{t}$. Finally, end specifies the terminated session. A message type U is either a boolean, an integer[1], a global type or a local type. Global types are used as message types when communicating shared names and local types are used when communicating session channels.

The local type $!k\langle U\rangle.T$ (resp. $?k\langle U\rangle.T$) specifies a role that sends (resp. receives) a message of type U over channel k, with T representing the remaining actions. The type $k \oplus \{l_i : T\}_{i\in I}$ (resp. $k \,\&\, \{l_i : T_i\}_{i\in I}$) specifies sending (resp. receiving) a label l_i, with T_i being the subsequent actions. As for global types, $\mu\mathbf{t}.T$ and $\mathbf{t}$ allow tail-recursive specifications. Lastly, end is a terminated role.

We deal with recursive variables in a standard way and write capture-avoiding substitution as $G_1[G_2/\mathbf{t}]$ (resp. $T_1[T_2/\mathbf{t}]$). We assume that types are closed and contractive. Global and local types are contractive if, for any of its sub-expressions with shape $\mu\mathbf{t}_0.\mu\mathbf{t}_1...\mu\mathbf{t}_n.\mathbf{t}$, the body $\mathbf{t}$ is not $\mathbf{t}_0$ [47].

A *local type environment* $\hat{\Delta}$ is a map from roles to local types. For every p in the domain of $\hat{\Delta}$ (denoted by $\mathsf{dom}(\hat{\Delta})$), there is a unique entry $\mathsf{p} : T$ for some T.

Projection. Projection is a partial function which, given a global type and a role, attempts to generate a local type that captures the specification of this role in the global type. When the global type specifies behaviour that the role cannot implement, projection is undefined. Our mechanisation uses the projection function $\mathsf{proj}_{\mathsf{p}}(G)$ proposed by Tirore et al. [51], which is defined in Figure 3. The function $\mathsf{proj}_{\mathsf{p}}(G)$ is defined in terms of two auxiliary definitions: a translation function, denoted by $\mathsf{trans}_{\mathsf{p}}(G)$, and a projectability predicate, denoted by $\mathsf{projectable}_{\mathsf{p}}(G)$. The translation function is a total function which produces a local type representing the local behavior of the translated role. The projectability predicate checks that the global type specifies behavior that is possible to implement by the projected role.

[1] Our mechanization excludes the integer message type, as the expressiveness of the expression language is orthogonal to our focus. We include the integer type here solely to simplify examples with distinct message types.

$$\mathsf{gVar}(\mathsf{t}, G) \stackrel{\text{def}}{=} \begin{cases} \mathsf{gVar}(\mathsf{t}, G_1) \ \wedge \ \mathbf{t} \neq \mathbf{t}' & \text{if } G = \mu \mathbf{t}'.G_1 \\ \mathbf{t} \neq \mathbf{t}' & \text{if } G = \mathbf{t}' \\ \mathsf{True} & \text{otherwise} \end{cases}$$

$$\mathsf{trans}_{\mathsf{p}}(\mathsf{p}_1 \to \mathsf{p}_2 : k\langle U \rangle.G) \stackrel{\text{def}}{=} \begin{cases} !k\langle U \rangle.(\mathsf{trans}_{\mathsf{p}}(G)) & \text{if } \mathsf{p} = \mathsf{p}_1 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ ?k\langle U \rangle.(\mathsf{trans}_{\mathsf{p}}(G)) & \text{if } \mathsf{p} = \mathsf{p}_2 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ \mathsf{trans}_{\mathsf{p}}(G) & \text{if } \mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \end{cases}$$

$$\mathsf{trans}_{\mathsf{p}}(\mathsf{p}_1 \to \mathsf{p}_2 : k\{l_j : G_j\}_{j \in J}) \stackrel{\text{def}}{=} \begin{cases} k \oplus \{l_j : \mathsf{trans}_{\mathsf{p}}(G_j)\}_{j \in J} & \text{if } \mathsf{p} = \mathsf{p}_1 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ k \ \& \ \{l_j : \mathsf{trans}_{\mathsf{p}}(G_j)\}_{j \in J} & \text{if } \mathsf{p} = \mathsf{p}_2 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ \mathsf{trans}_{\mathsf{p}}(G_1) & \text{otherwise} \end{cases}$$

$$\mathsf{trans}_{\mathsf{p}}(\mu \mathbf{t}.G) \stackrel{\text{def}}{=} \begin{cases} \mu \mathbf{t}.(\mathsf{trans}_{\mathsf{p}}(G)) & \text{if } \mathsf{gVar}(\mathsf{t}, G) \\ \mathsf{end} & \text{otherwise} \end{cases}$$

$$\mathsf{trans}_{\mathsf{p}}(\mathbf{t}) \stackrel{\text{def}}{=} \mathbf{t} \qquad \mathsf{trans}_{\mathsf{p}}(\mathsf{end}) \stackrel{\text{def}}{=} \mathsf{end}$$

$$\mathsf{proj}_{\mathsf{p}}(G) \stackrel{\text{def}}{=} \begin{cases} \mathsf{trans}_{\mathsf{p}}(G) & \text{if } \mathsf{projectable}_{\mathsf{p}}(G) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathsf{proj_roles}(G, \ \{\mathsf{p}_1, \ \cdots, \ \mathsf{p}_n\}) \stackrel{\text{def}}{=} \begin{cases} \mathsf{p}_1 : \mathsf{proj}_{\mathsf{p}_1}(G), \ \cdots, & \text{if } \forall i \in \{1, \ \cdots, \ n\}. \\ \mathsf{p}_n : \mathsf{proj}_{\mathsf{p}_n}(G) & \mathsf{proj}_{\mathsf{p}_i}(G) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathsf{full_proj}(G) \stackrel{\text{def}}{=} \mathsf{proj_roles}(G, \ \mathsf{roles}(G))$$

Figure 3 Projection ( ).

Translation is defined such that $\mathsf{trans}_{\mathsf{p}}(\mathsf{p}_1 \to \mathsf{p}_2 : k\langle U \rangle.\ G)$ produces the output $!k\langle U \rangle$ (resp. input $?k\langle U \rangle$) when the translated role is p_1 (resp. p_2), and translates the remaining global type as $\mathsf{trans}_{\mathsf{p}}(G)$. If the translated role does not occur in the interaction, translation simply ignores the interaction proceeds as $\mathsf{trans}_{\mathsf{p}}(G)$. The translation of a branching global type $\mathsf{trans}_{\mathsf{p}}(\mathsf{p}_1 \to \mathsf{p}_2 : k\{l_j : G_j\}_{j \in J})$ is similar when p occurs in the interaction. When p does not occur, translation chooses the first branch j_1 and proceeds as $\mathsf{trans}_{\mathsf{p}}(G_1)$. The auxiliary predicate $\mathsf{gVar}(\mathsf{t}, G)$ is used in the translation of $\mu \mathbf{t}.G$, and checks if $\mu \mathbf{t}$ can safely be added to $\mathsf{trans}_{\mathsf{p}}(G)$, without producing non-contractive local types. In particular, this predicate ensures that trans never produces non-contractive types like $\mu \mathbf{t}.\mathbf{t}$. Finally, $\mathsf{trans}_{\mathsf{p}}(\mathbf{t})$ (resp. $\mathsf{trans}_{\mathsf{p}}(\mathsf{end})$) returns $\mathbf{t}$ (resp. end).

On its own, the translation function is too permissive. E.g., the global type

$$\mathsf{p} \to \mathsf{q} \colon 1 \left\{ \begin{array}{l} l_1 \colon \mathsf{r} \to \mathsf{s} : 2\langle \mathsf{bool} \rangle.\mathsf{end} \\ l_2 \colon \mathsf{r} \to \mathsf{s} : 2\langle \mathsf{int} \rangle.\mathsf{end} \end{array} \right\}$$

$$\frac{}{\mathsf{p} \to \mathsf{q} : k\langle U\rangle.G \xrightarrow{\mathsf{p}\to\mathsf{q}:k\langle U\rangle} G}\ [\text{GR1}] \qquad \frac{j \in I}{\mathsf{p} \to \mathsf{q} : k\{l_i : G_i\}_{i\in I} \xrightarrow{\mathsf{p}\to\mathsf{q}:k\langle l_j\rangle} G_j}\ [\text{GR2}]$$

$$\frac{G_1 \xrightarrow{\ell} G_2 \qquad \mathsf{q} \notin \ell}{\mathsf{p} \to \mathsf{q} : k\langle U\rangle.G_1 \xrightarrow{\ell} \mathsf{p} \to \mathsf{q} : k\langle U\rangle.G_2}\ [\text{GR3}]$$

$$\frac{\forall i \in I.\ G_i \xrightarrow{\ell} G_i' \qquad \mathsf{q} \notin \ell}{\mathsf{p} \to \mathsf{q} : k\{l_i : G_i\}_{i\in I} \xrightarrow{\ell} \mathsf{p} \to \mathsf{q} : k\{l_i : G_i'\}_{i\in I}}\ [\text{GR4}] \qquad \frac{G[\mu \mathbf{t}.G/t] \xrightarrow{\ell} G'}{\mu \mathbf{t}.G \xrightarrow{\ell} G'}\ [\text{GR5}]$$

Figure 4 Semantics for global types ( ).

specifies that p sends l_1 or l_2 to q and depending on this choice, r has to send a boolean or an integer to s. The problem is that role r cannot know which choice was made, and its specification is thus not possible to implement. The projection of role r should therefore be undefined for this global type. As a total function, translation naively ignores this and translates the l_1 branch, producing the local type $!2\langle \mathsf{bool}\rangle.\mathsf{end}$. If role p chooses l_2, role r will incorrectly behave as if l_1 was chosen.

The problem in the example above is solved by letting the projection function, which we denote with $\mathsf{proj}_{\mathsf{p}}(G)$, be defined only when the projectability predicate $\mathsf{projectable}_{\mathsf{p}}(G)$ holds. In such case, the returned local type is computed as $\mathsf{trans}_{\mathsf{p}}(G)$. This predicate depends on a coinductive specification of projection that is not covered in this paper. For the definition of the projectable predicate and the coinductive specification we refer the reader to the paper where they were first introduced [51].

The bottom of Figure 3 defines the function full_proj which projects a global type over all its roles, and generates a local type environment $\hat{\Delta}$.

▶ Remark 3 (Differences with the original work.). Projection is notoriously complex to define correctly. Our mechanisation uses the projection by Tirore et al. [51], which, unlike that of Honda et al., handles μ-binders correctly and has been proven sound and complete in Coq with respect to the coinductive projection specification by Ghilezan et al. [24].

Semantics of Global Types, Local Types, and Environments. We define the semantics of global types as a labeled transition system, denoted by $G \xrightarrow{\ell} G'$. A label ℓ, called *interaction*, is formally defined as:

$$\mathcal{U} ::= U \mid l \qquad\qquad \ell ::= \mathsf{p} \to \mathsf{q} : k\langle \mathcal{U}\rangle$$

The syntactic category $\mathcal{U}$ is either a message type U or a label l, and an interaction $\mathsf{p} \to \mathsf{q} : k\langle \mathcal{U}\rangle$ is either the exchange of a message type or a label.

The semantics of global types is defined by the rules given in Figure 4. Rules [GR1] and [GR2] allow a global type to reduce by its first interaction. Rules [GR3] and [GR4] allow a global type to reduce by an interaction ℓ that occurs nested in one of its continuations, which makes the semantics asynchronous by allowing p to occur in ℓ. On the process level, this corresponds to p putting a message on the queue to be read by q, followed by p completing interaction ℓ with a different role before q receives the message. The completed interaction ℓ may not involve q because inputs are blocking. Rule [GR5] unfolds a μ-binder.

$$\frac{}{!k\langle U\rangle.T \xrightarrow{!k\langle U\rangle} T}\ [\text{LR1}] \qquad \frac{}{?k\langle U\rangle.T \xrightarrow{?k\langle U\rangle} T}\ [\text{LR2}]$$

$$\frac{j \in I}{k \oplus \{l_i : T\}_{i\in I} \xrightarrow{!k\langle l_j\rangle} T_j}\ [\text{LR3}] \qquad \frac{j \in I}{k \,\&\, \{l_i : T_i\}_{i\in I} \xrightarrow{?k\langle l_j\rangle} T_j}\ [\text{LR4}]$$

$$\frac{T \xrightarrow{\zeta} T' \quad \mathsf{ch}(\zeta) \neq k}{!k\langle U\rangle.T \xrightarrow{\zeta} !k\langle U\rangle.T'}\ [\text{LR5}] \qquad \frac{\forall i \in I.\ T_i \xrightarrow{\zeta} T_i' \quad \mathsf{ch}(\zeta) \neq k}{k \oplus \{l_i : T\}_{i\in I} \xrightarrow{\zeta} k \oplus \{l_i : T_i'\}_{i\in I}}\ [\text{LR6}]$$

$$\frac{T[\mu \mathbf{t}.T/\mathbf{t}] \xrightarrow{\zeta} T'}{\mu \mathbf{t}.T \xrightarrow{\zeta} T'}\ [\text{LR7}] \qquad \frac{T_1 \xrightarrow{!k\langle U\rangle} T_1' \qquad T_2 \xrightarrow{?k\langle U\rangle} T_2'}{\hat{\Delta},\ \mathsf{p} : T_1,\ \mathsf{q} : T_2 \xrightarrow{\mathsf{p}\to\mathsf{q}:k\langle U\rangle} \hat{\Delta},\ \mathsf{p} : T_1',\ \mathsf{q} : T_2'}\ [\text{LEnv}]$$

Figure 5 Semantics for local types ( ) and local type environments ( ).

The semantics of local types is a two-level labeled transition system: the first level gives semantics to local types T, denoted by $T \xrightarrow{\zeta} T'$, and the second level gives semantics to local type environments $\hat{\Delta}$, denoted by $\hat{\Delta} \xrightarrow{\ell} \hat{\Delta}'$. A label ζ, called *action*, is formally defined as:

$$\zeta ::= !k\langle U\rangle \mid ?k\langle U\rangle$$

An action is the sending or receiving of a message type or a label. We write $\mathsf{ch}(\zeta)$ (resp. $\mathsf{ch}(\ell)$) to extract the channel of a local label ζ (resp. global label ℓ).

Formally, we define the semantics of local types and typing environments by the rules given in Figure 5. Rules [LR1], [LR2], [LR3] and [LR4] allow a local type to reduce by its first action. Similar to the global type rules [GR3] and [GR4], rules [LR5] and [LR6] allow a local type to reduce by an action ζ that occurs nested in one of its continuations. These rules make the local type semantics asynchronous. A condition for reducing by a nested action ζ, is that ζ uses a different channel than the first action, i.e., $\mathsf{ch}(\zeta) \neq k$. The intuition is that the first action, which is an output, already has occurred and it has resulted in a value being put on channel k; but before this value has been received by another participant, another interaction has in the meantime completed on another channel k'. Interactions between local types is captured by the semantics of local type environments $\hat{\Delta} \xrightarrow{\ell} \hat{\Delta}'$, given by the single rule [LEnv]. An environment may reduce by $\mathsf{p} \to \mathsf{q} : k\langle U\rangle$ when it maps p and q to local types that can reduce by $!k\langle U\rangle$ and $?k\langle U\rangle$.

▶ Remark 4 (Differences with the original work). We define the semantics of local types differently from Honda et al. Unlike their semantics, ours includes the asynchronous rules [LR5] and [LR6]. Instead, Honda et al. achieve asynchrony by defining a relation that relates local types with permuted outputs; for example, $!k\langle U\rangle.!k'\langle U'\rangle.T$ is related to $!k'\langle U'\rangle.!k\langle U\rangle.T$. Similar to our asynchronous rules, this permutation is only allowed if $k \neq k'$. However, that by Honda et al. is a more restrictive semantics, as it misses cases where an output is preceded by a nested input on a different channel, thereby permitting the input to be placed in front of the output. For instance, with the types $!k\langle U\rangle;?k'\langle U'\rangle;T$ and $?k'\langle U'\rangle;!k\langle U\rangle;T$, for $k \neq k'$, our rule [LR5] with $\zeta = ?k'\langle U\rangle$ allows the output to be preceded by the input. Preceding an output with an input is essential for proving the equivalence between the semantics of global types and that of local type environments, a property known as the *Projection Theorem*. Consequently, their restrictive semantics means that their projection theorem (Lemma 5.11 [33]) does not hold. At the end of this section, we present a projection theorem for our semantics.

Linearity. The Honda et al. formulation of multiparty session types began an entire line of research, but to the best of our knowledge, all future work has adopted the *implicit* channels introduced by Bettini et al. [4]. Although they are more expressive, explicit channels are more complicated, allowing global types to contain races. For example, the global type:

$$\mathsf{p} \to \mathsf{q} : k\langle \mathsf{int} \rangle.\ \mathsf{r} \to \mathsf{q} : k\langle \mathsf{bool} \rangle.\mathsf{end} \tag{8}$$

is problematic because the two interactions $\mathsf{p} \to \mathsf{q} : k\langle \mathsf{int} \rangle$ and $\mathsf{r} \to \mathsf{q} : k\langle \mathsf{bool} \rangle$ share the same explicit channel k which introduces a race between p and r. The problem is that roles p, q, and r are freely using channel k without coordination. Hence, if r writes on channel k before p, it wins the race and a type error occurs when q receives bool while expecting int. Linearity of global types (not to be confused with standard linear types) is a property that rules out global types with interactions that introduce races. The main intuition behind linearity is that those interactions of a global type that share the same channel must have some causal dependency that avoids races. In the example, there is no causal dependency between the output of p and the output of q. However, the global type

$$\mathsf{p} \to \mathsf{q} : k\langle \mathsf{int} \rangle.\ \mathsf{p} \to \mathsf{r} : k'\langle \mathsf{string} \rangle.\ \mathsf{r} \to \mathsf{q} : k\langle \mathsf{bool} \rangle.\mathsf{end} \tag{9}$$

enforces a dependency between the output of p and the output of r: p first sends int on channel k and then sends string on k' while r has to receive string on channel k' before sending bool on channel k. Linearity checks that all execution paths of a global type enforce dependencies between interactions on the same channel, ensuring that no race can occur.

We first introduce the notion of a trace as:

▶ **Definition 5** (Trace 🐓). *A trace ρ of a global type G is a (possibly empty) sequence of interactions such that*

$$\frac{\mathsf{trace}\ \rho\ G}{\mathsf{trace}\ (\mathsf{p} \to \mathsf{q} : k\langle U \rangle \cdot \rho)\ (\mathsf{p} \to \mathsf{q} : k\langle U \rangle.G)} \qquad \frac{\mathsf{trace}\ \rho\ (G[\mu \mathbf{t}.G\ \mathsf{t}])}{\mathsf{trace}\ \rho\ (\mu \mathbf{t}.G)}$$

$$\frac{\mathsf{trace}\ \rho\ G_i}{\mathsf{trace}\ (\mathsf{p} \to \mathsf{q} : k\langle l_i \rangle \cdot \rho)\ (\mathsf{p} \to \mathsf{q} : k\{l_i : G_i\}_{i \in I})} \qquad \frac{}{\mathsf{trace}\ \epsilon\ G}$$

Intuitively, a trace records the interactions of a finite walk through the global type, starting from the first interaction and unfolding μ-binders when they occur. For example, the global type G in Equation (1) has traces $\mathsf{p} \to \mathsf{q} : 1\langle l_1 \rangle \cdot \mathsf{q} \to \mathsf{r} : 2\langle \mathsf{bool} \rangle \cdot \epsilon$ and $\mathsf{p} \to \mathsf{q} : 1\langle l_2 \rangle \cdot \mathsf{p} \to \mathsf{r} : 2\langle \mathsf{bool} \rangle \cdot \epsilon$. However, since a trace is a finite walk, also $\mathsf{p} \to \mathsf{q} : 1\langle l_1 \rangle \cdot \epsilon$ and $\mathsf{p} \to \mathsf{q} : 1\langle l_2 \rangle \cdot \epsilon$ are traces of G. We say $\mathsf{p} \to \mathsf{q} : k$ happens before $\mathsf{r} \to \mathsf{s} : k'$ in G if, for some ρ and ρ', we can derive:

$$\mathsf{trace}\ (\rho \cdot \mathsf{p} \to \mathsf{q} : k\langle \mathcal{U} \rangle \cdot \rho' \cdot \mathsf{r} \to \mathsf{s} : k'\langle \mathcal{U}' \rangle)\ G \tag{10}$$

We omit $\mathcal{U}$ and $\mathcal{U}'$ from the interactions as they provide no causal information. Given two interactions ℓ_1 and ℓ_2, we say that ℓ_1 and ℓ_2 are ordered if ℓ_1 happens before ℓ_2 or vice versa. Linearity asserts that ordered interactions that share the same channel must have dependencies between their inputs and outputs, which then guarantees the absence of races. Formally, we first define dependencies as:

▶ **Definition 6** (Input and Output Dependencies). *The interaction dependencies* $\mathsf{IO}, \mathsf{II}, \mathsf{OO}, \mathsf{IOOO} \subseteq \ell \times \ell$ (🐓) *and judgments* input dep ρ (🐓) *and* output dep ρ (🐓) *are defined as:*

$$\mathsf{p} \to \mathsf{q} : k\ \mathsf{IO}\ \mathsf{s} \to \mathsf{r} : k' \stackrel{\mathsf{def}}{=} (\mathsf{q} = \mathsf{s}) \qquad \mathsf{p} \to \mathsf{q} : k\ \mathsf{II}\ \mathsf{r} \to \mathsf{s} : k' \stackrel{\mathsf{def}}{=} (\mathsf{q} = \mathsf{s})$$

$$\mathsf{p} \to \mathsf{q} : k\ \mathsf{OO}\ \mathsf{s} \to \mathsf{r} : k' \stackrel{\mathsf{def}}{=} (\mathsf{p} = \mathsf{s}) \wedge (k = k') \qquad \ell_1\ \mathsf{IOOO}\ \ell_2 \stackrel{\mathsf{def}}{=} \ell_1\ \mathsf{IO}\ \ell_2 \vee \ell_1\ \mathsf{OO}\ \ell_2$$

$$\text{[I-Base]}\ \frac{\ell_1\ \mathsf{II}\ \ell_2}{\mathsf{input\ dep}\ \ell_1 \cdot \ell_2} \qquad \text{[I-Skip]}\ \frac{\mathsf{input\ dep}\ \ell_1 \cdot \rho \cdot \ell_3}{\mathsf{input\ dep}\ \ell_1 \cdot \ell_2 \cdot \rho \cdot \ell_3} \qquad \text{[I-Tail]}\ \frac{\mathsf{input\ dep}\ \ell_2 \cdot \rho \cdot \ell_3 \qquad \ell_1\ \mathsf{IO}\ \ell_2}{\mathsf{input\ dep}\ \ell_1 \cdot \ell_2 \cdot \rho \cdot \ell_3}$$

$$\text{[O-Base]}\ \frac{\ell_1\ \mathsf{IOOO}\ \ell_2}{\mathsf{output\ dep}\ \ell_1 \cdot \ell_2} \qquad \text{[O-Skip]}\ \frac{\mathsf{output\ dep}\ \ell_1 \cdot \rho \cdot \ell_3}{\mathsf{output\ dep}\ \ell_1 \cdot \ell_2 \cdot \rho \cdot \ell_3} \qquad \text{[O-Tail]}\ \frac{\mathsf{output\ dep}\ \ell_2 \cdot \rho \cdot \ell_3 \qquad \ell_1\ \mathsf{IOOO}\ \ell_2}{\mathsf{output\ dep}\ \ell_1 \cdot \ell_2 \cdot \rho \cdot \ell_3}$$

Interaction dependencies relate interactions by asserting different ways the same role can occur in both interactions, such as IO (input-output) meaning the input role of the first interaction is also the output role of the second interaction (cf. [32]). Likewise we have OO (output-output) and II (input-input) dependencies. Input and output dependencies are obtained by composing interaction dependencies. An input dependency is a sequence of IO dependencies that ends in II while an output dependency is a sequence of IOOO dependencies. We use these chains to show causal orderings between the roles in the first and last interactions of the chain. For the sake of presentation of rules [O-Base] and [O-Tail], we have defined the interaction dependency IOOO which simply states that there can be either an IO or an OO dependency.

We begin by examining each rule of input dependency to understand why they assert and preserve the ordering of inputs. The shortest input dependency is established by [I-Base], where the two inputs originate from the same role and are therefore ordered. Rule [I-Skip] reflects that chains do not need to include all interactions in the trace. Finally rule [I-Tail] extends an existing input dependency between ℓ_2 and ℓ_3 by adding ℓ_1 in front of ℓ_2 when it satisfies $\ell_1\ \mathsf{IO}\ \ell_2$. We can assume that the inputs of ℓ_2 and ℓ_3 are ordered and this is also the case between ℓ_1 and ℓ_3 if it is the case for ℓ_1 and ℓ_2. If the channels of ℓ_1 and ℓ_2 are different then the inputs are ordered because the input of ℓ_2 cannot intercept the output of ℓ_1. If the channels are the same, then there must exist input and output dependencies between ℓ_1 and ℓ_2, ensuring in particular that their inputs are ordered. More generally, input and output dependencies show that a specific pair of ordered interactions, with the same channel, have ordered inputs and outputs, by relying on this property to hold for all other ordered interactions with the same channel.

Next, we consider why the rules of output dependency preserve ordering of outputs. We start by the shortest output dependency built by [O-Base], relating the interactions by either OO or IO. The former ensures the ordering by having the same role in both outputs. IO preserves output dependency by blocking the output of the second interaction. Rule [O-Skip] is the corresponding skip. Finally, rule [O-Tail] allows output dependencies to be extended either by OO or IO, extending the ordering of the outputs between ℓ_2 and ℓ_3 by adding ℓ_1 in front, as long as the outputs of ℓ_1 and ℓ_2 are ordered as well. The argument for why output ordering is preserved is similar to that of rule [O-Base].

Linearity asserts dependencies between any ordered interactions with the same channel. Unlike Honda et al., we define linearity in terms of a weaker property, $\mathsf{linearHead}(G)$, which simplifies the decision procedure. This formulation only asserts the existence of dependencies when the first of the two interactions is also the first interaction in the trace.

We define linearity of a global type, $\mathsf{linear}(G)$, in terms of $\mathsf{linearHead}(G)$ in Figure 7. Intuitively, $\mathsf{linearHead}(G)$ checks if the first interaction is free of races and $\mathsf{next}(G)$ proceeds to the continuation of the global type. Due to global types having only finitely many distinct subterms, a consequence of μ-types being regular [47], only finitely many applications of the coinductive rule of linear is necessary before a previously encountered global type inevitably reappears, at which point the proof can circularly be closed.

$$|G| \stackrel{\text{def}}{=} \begin{cases} 1 + |G_1| & \text{if } G = \mu \mathbf{t}.G_1 \\ 0 & \text{otherwise} \end{cases} \qquad \mathsf{unfold_once}(G) \stackrel{\text{def}}{=} \begin{cases} G_1[\mu \mathbf{t}.G_1/t] & \text{if } G = \mu \mathbf{t}.G_1 \\ G & \text{otherwise} \end{cases}$$

$$\mathsf{unf}(G) \stackrel{\text{def}}{=} \mathsf{unfold_once}^{|G|}(G)$$

Figure 6 Calculating μ-height ( ) and unfolding global types ( ). Corresponding operations exist on local types (respectively and).

$$\mathsf{linearHead}(G) \stackrel{\text{def}}{=} \forall \ell_1\ \rho\ \ell_2.\ \ \mathsf{trace}\ (\ell_1 \cdot \rho \cdot \ell_2)\ G\ \rightarrow\ \mathsf{ch}(\ell_1) = \mathsf{ch}(\ell_2)\ \rightarrow \\ \mathsf{input\ dep}\ \ell_1 \cdot \rho \cdot \ell_2 \wedge \mathsf{output\ dep}\ \ell_1 \cdot \rho \cdot \ell_2$$

$$\mathsf{next}(G) \stackrel{\text{def}}{=} \begin{cases} \{G_1\} & \text{if } \mathsf{unf}(G) = \mathsf{p}_1 \to \mathsf{p}_2 : k\langle U \rangle.G_1 \\ \{G_i \mid i \in I\} & \text{if } \mathsf{unf}(G) = \mathsf{p}_1 \to \mathsf{p}_2 : k\{l_i : G_i\}_{i \in I} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\frac{\mathsf{linearHead}(G) \quad \forall G' \in \mathsf{next}(G).\ \mathsf{linear}(G')}{\mathsf{linear}(G)}$$

Figure 7 Linearity of global types ( ).

The definition of the function $\mathsf{next}(G)$ uses the unfold operation $\mathsf{unf}(G)$ which we define in Figure 6. The definition uses μ-height, denoted by $|G|$, which is the number of top-level μ-binders occurring before another construct appears. For example, $|\mu \mathbf{t}.\mathsf{end}| = 1$ and $|\mathsf{p} \to \mathsf{q} : k\langle U \rangle.G| = 0$. We create an auxiliary function $\mathsf{unfold_once}(\cdot)$ which unfolds a single binder by turning $\mu \mathbf{t}.G$ into $G[\mu \mathbf{t}.G/\mathbf{t}]$ and define the unfolding operation for global types, denoted by $\mathsf{unf}(G)$, as the repeated application of $\mathsf{unfold_once}(G)$ μ-height times. The operation $\mathsf{unf}(T)$ can be similarly defined on local types and we elide its definition.

▶ **Theorem 7** (Decidability of Linearity). *The predicate* $\mathsf{linear}(G)$ *is decidable.*

Decidability of linearity can be reduced to a reachability problem on a graph induced by a global type with $\mathsf{next}(\cdot)$ as an adjacency list.

Projection Theorem. We have seen that interactions in a protocol can be described by $G \xrightarrow{\ell} G'$ and $\hat{\Delta} \xrightarrow{\ell} \hat{\Delta}'$. Naturally, we want to establish a correspondence between these two semantics with respect to the projection operation. Intuitively, if $\hat{\Delta}$ and $\hat{\Delta}'$ are derived from G and G', respectively, then $G \xrightarrow{\ell} G'$ holds if and only if $\hat{\Delta} \xrightarrow{\ell} \hat{\Delta}'$. However, this result is too strong due to recursion, which may unfold some terms during reduction, making projection slightly different. The issue lies in syntactic equality on local types being too restrictive to prove the projection theorem. Local types, however, can be more loosely related by a coinductive equality on their continuously unfolded terms:

▶ **Definition 8** (Coinductive equality). *Coinductive equality is a relation on local types defined by the following rules:*

$$\frac{\mathit{unf}(T) = \mathsf{end} \quad \mathit{unf}(T') = \mathsf{end}}{T \approx T'} \text{ [Eq-End]}$$

$$\frac{\mathit{unf}(T) = !k\langle U\rangle.T'' \quad \mathit{unf}(T') = !k\langle U\rangle.T''' \quad T'' \approx T'''}{T \approx T'} \text{ [Eq-!]}$$

$$\frac{\mathit{unf}(T) = ?k\langle U\rangle.T'' \quad \mathit{unf}(T') = ?k\langle U\rangle.T''' \quad T'' \approx T'''}{T \approx T'} \text{ [Eq-?]}$$

$$\frac{\mathit{unf}(T) = k \oplus \{l_j : T_j\}_{j\in J} \quad \mathit{unf}(T') = k \oplus \{l_j : T'_j\}_{j\in J} \quad \forall j \in J.\ T_j \approx T'_j}{T \approx T'} \text{ [Eq-}\oplus\text{]}$$

$$\frac{\mathit{unf}(T) = k \,\&\, \{l_j : T_j\}_{j\in J} \quad \mathit{unf}(T') = k \,\&\, \{l_j : T'_j\}_{j\in J} \quad \forall j \in J.\ T_j \approx T'_j}{T \approx T'} \text{ [Eq-\&]}$$

This relation checks the shape of the unfolded local types, and checks whether the same actions are specified, before proceeding to the continuations. Because of unfolding, it might require a circular proof to derive the equality. With an abuse of notation, $\hat{\Delta}_1 \approx \hat{\Delta}_2$ is the point-wise extension of $\approx$ to local type environments. Coinductive equality on continuously unfolded μ-types is decidable [47], and this has been mechanised [18]. Our mechanisation also includes a proof of the decidability of the relation $\approx$, which, to the best of our knowledge, is the first mechanisation for session types. Using this relation, we can state the following:

▶ **Theorem 9** (Projection). *Let G be a global type such that* $\mathsf{linear}(G)$ *holds and* $\mathsf{full_proj}(G)$ *is defined, then*

1. (🐓) *if $G \xrightarrow{l} G'$ then $\hat{\Delta}$ exists such that* $\mathsf{full_proj}(G) \xrightarrow{l} \hat{\Delta}$ *and* $\mathsf{proj_roles}(G', \mathsf{roles}(G)) \approx \hat{\Delta}$
2. (🐓) *if* $\mathsf{full_proj}(G) \xrightarrow{l} \hat{\Delta}$ *then G' exists such that $G \xrightarrow{l} G'$ and* $\mathsf{proj_roles}(G', \mathsf{roles}(G)) \approx \hat{\Delta}$.

▶ Remark 10 (Differences with the original work). The projection theorem of Honda et al. does not take into account that $\mathsf{roles}(G')$ may be a strict subset of $\mathsf{roles}(G)$. The set of roles in a global type can decrease after reduction if the reduced interaction involves a role that only occurs in that interaction. This results in an invalid environment reduction because the domain shrinks, exemplifying an innocent error that is hard to catch on paper.

4 Queue Types, Environment Decomposition, and Coherence

Queue types. Honda et al. handle the typing of queues using type contexts $\mathcal{T}$, which are local types with a hole $[\cdot]$. They decompose a local type T into a context $\mathcal{T}$ and another local type T', such that the applied context $\mathcal{T}[T']$ is related to T by a subtyping relation (see Definition 5.1 [33]). The type checking of a process against an environment $\hat{\Delta}$, may involve multiple of these decompositions. Mechanising this approach is non-trivial, and for convenience, we take a different approach: we type processes with an environment that contains local types that are decomposed *a priori*. We refer to this as a decomposed environment, denoted by $\hat{\Delta}; \hat{\mathcal{Q}}$. We formally define queue environments $\hat{\mathcal{Q}}$, and the queue types Q it contains, as:

▶ **Definition 11** (Queue type and Queue Environment). Queue types (🐓), *denoted by* $\mathtt{Q}$, *and* queue environments (🐓), *denoted by* $\hat{\mathcal{Q}}$ *, are defined as:*

$$\mathtt{Q} ::= \emptyset \mid (k, \mathcal{U}) \cdot \mathtt{Q} \qquad \hat{\mathcal{Q}} ::= \emptyset \mid \hat{\mathcal{Q}},\ \mathtt{p} : \mathtt{Q}$$

A queue environment $\hat{\mathcal{Q}}$ is a map from roles to queue types, and a queue type is a sequence of pairs. In each pair, the left component is always a channel, and the right component is either a message type U or a label l. A pair $(k, \mathcal{U})$ in the queue type models the presence of a message $\mathcal{U}$ in the queue at channel k, and the order of these pairs indicates the order in which these messages were sent. Queue types are given with respect to roles in the environment $\hat{\mathcal{Q}}$, and this environment describes the content of all queues in the session. Queue types are derived from local types using the following notion of path:

▶ **Definition 12** (Path 🐓). *A path, denoted by σ, is a (possibly empty) sequence of elements from the set $\mathcal{L} \cup \{\bot\}$, where $\mathcal{L}$ is the set of labels defined in Section 2.*

A path is similar to a trace, it represents a walk through the μ-type. Unlike traces, we use paths to decompose local types.

▶ **Definition 13** (Decomposition of Local Types 🐓). *The decomposition of local types, denoted by* $\mathsf{decomp} : \sigma \to T \to T \times Q$*, is defined as:*

$$\mathsf{decomp}_\sigma\ T \stackrel{\mathsf{def}}{=} \begin{cases} (T', (k, U) \cdot \mathtt{Q}) & \mathsf{unf}\ T = !k\langle U\rangle.T_1 \ \wedge\ \sigma = \bot \cdot \sigma' \ \wedge\ \mathsf{decomp}_\sigma\ T_1 = (T', \mathtt{Q}) \\ (T', (k, l_j) \cdot \mathtt{Q}) & \mathsf{unf}\ T = k \oplus \{l_i : T\}_{i \in I} \ \wedge\ \sigma = l_j \cdot \sigma' \ \wedge \\ & \mathsf{decomp}_\sigma\ T_j = (T', \mathtt{Q}) \quad \wedge\ j \in I \\ (\mathsf{unf}\ T, \emptyset) & \sigma = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

With respect to a path, decomposition splits a local type into a pair consisting of a residual local type and a queue type. We overload notation, denoting the point-wise extension of decomposition onto environments as $\mathsf{decomp}_f\ \hat{\Delta}$, which splits the environment $\hat{\Delta}$, with respect to a path function $f : \mathscr{P} \to \sigma$, into a residual environment $\hat{\Delta}'$ and a queue environment $\hat{\mathcal{Q}}$.

▶ **Definition 14** (Decomposition of Environments 🐓). *Let the path function f be such that $\forall i \in \{1, \ldots, n\}.\ \mathsf{decomp}_{f(\mathsf{p}_i)}\ T_i = (T_i', \mathtt{Q}_i)$. Environment decomposition, denoted by* $\mathsf{decomp} : (\mathscr{P} \to \sigma) \to \hat{\Delta} \to \hat{\Delta} \times \hat{\mathcal{Q}}$*, is defined as:*

$$\mathsf{decomp}_f\ (\mathsf{p}_1 : T_1, ..., \mathsf{p}_n : T_n) \quad \stackrel{\mathsf{def}}{=} \quad \mathsf{p}_1 : T_1', ..., \mathsf{p}_n : T_n';\ \mathsf{p}_1 : \mathtt{Q}_1, ..., \mathsf{p}_n : \mathtt{Q}_n$$

▶ **Example 15.** The path function tracks, within a session, the messages that are in transit. Recall that the initial state of our running example from Section 1, Equation (4), has two empty queues. Because the queues are empty, the decomposition of the local types that type the three roles, should all produce the empty queue type $\emptyset$. This is achieved with the path function: $f(\mathsf{p}) = \emptyset$. After p has put a message on each of the two queues in Equation (5), the local type of p must be decomposed in a way that reflects the two outputs have been performed, and that the remaining roles have performed no outputs. We achieve this with the path function: $f(\mathsf{p}') = \mathsf{if}\ \mathsf{p}' = \mathsf{p}\ \mathsf{then}\ l_1 \cdot \bot\ \mathsf{else}\ \emptyset$. The decomposition of $G\!\upharpoonright_{\mathsf{p}}$ with respect to $f(\mathsf{p})$, where G is the global type in our running example, yields the residual local type end and queue type $(1, l_2) \cdot (2, \mathsf{bool})$.

Coherence. Honda et al. use a well-formedness condition on global types called coherence, which they define as a global type being linear and projectable, i.e., its projection is defined for all roles. Coherence is key to the soundness of Honda et al.'s work and other approaches based on global types, as it ensures that local behaviours are compatible [11, 49, 12] and correctly follow the global specification. However, coherence is not sufficient to rule out our counterexample. For this reason, we introduce an additional property to our definition of coherence, called unstuckness and denoted by $\mathsf{unstuck}\ (G)$.

$$\frac{}{\mathsf{p} \to \mathsf{q} : k\langle U\rangle.G \downarrow^1 \mathsf{p} \to \mathsf{q} : k\langle U\rangle}\ [\text{GB-1}] \qquad \frac{j \in I}{\mathsf{p} \to \mathsf{q} : k\{l_i : G_i\}_{i\in I} \downarrow^1 \mathsf{p} \to \mathsf{q} : k\langle l_j\rangle}\ [\text{GB-2}]$$

$$\frac{\mathsf{q} \notin \ell \quad G \downarrow^1 \ell}{\mathsf{p} \to \mathsf{q} : k\langle U\rangle.G \downarrow^1 \ell}\ [\text{GB-3}] \qquad \frac{\mathsf{q} \notin \ell \quad j \in I \quad G_j \downarrow^1 \ell}{\mathsf{p} \to \mathsf{q} : k\{l_i : G_i\}_{i\in I} \downarrow^1 \ell}\ [\text{GB-4}] \qquad \frac{G[\mu \mathbf{t}.G/t] \downarrow^1 \ell}{\mu \mathbf{t}.G \downarrow^1 \ell}\ [\text{GB-5}]$$

Figure 8 Barb relation for global types ( ).

The idea behind unstuckness is the following. The global type which specifies the counterexample is stuck because the only relevant rule, [GR4], is not applicable due to its restrictive premise, which requires universal quantification $\forall i \in I.\ G_i \xrightarrow{\ell} G_i'$. If this premise were relaxed to existential quantification $\exists i \in I.\ G_i \xrightarrow{\ell} G_i'$, then the global type would not be stuck. It is however not an option to relax the rule because it would break the Projection theorem. Instead, we define $\mathsf{unstuck}\ (G)$, which captures global types where the universally quantified premise implies the existentially quantified premise. The global type in the counterexample does not satisfy this property and is thus ruled out.

In order to define the unstuck predicate, we use the notion of barb, denoted by $G \downarrow^1 \ell$, formally defined by the rules in Figure 8. Intuitively, $G \downarrow^1 \ell$ holds whenever it is possible to derive that G can reduce by ℓ, in the relaxed setting where the existentially quantified premise is used in [GR4]. This lets us quantify over the labels for which reduction derivations must exist. This property must also be preserved by reduction, so we additionally assert that the reduced global type G' satisfies the property as well. For recursive global types, a derivation will not be finite, and the predicate is therefore coinductively defined:

▶ **Definition 16** (Unstuck). *We say a global type is unstuck if it satisfies the predicate* $\mathsf{unstuck}\ (G)$*, coinductively defined as:*

$$\frac{\forall \ell.\ G \downarrow^1 \ell \implies \exists G'.G \xrightarrow{\ell} G' \wedge \mathsf{unstuck}\ (G')}{\mathsf{unstuck}\ (G)}$$

We can finally introduce our stricter notion of coherence:

▶ **Definition 17** (Coherence). *The coherence of global types, local types and decomposed environments, denoted respectively by* $\mathsf{coherent}(G)$*,* $\mathsf{coherent}(L)$ *and* $\mathsf{coherent}(\hat{\Delta};\hat{\mathcal{Q}})$*, is defined as:*

- ( ) $\mathsf{coherent}(G)$ *holds iff* G *is projectable, linear and unstuck*
- ( ) $\mathsf{coherent}(\hat{\Delta})$ *holds iff there exists a coherent* G *whose full projection is* $\hat{\Delta}$
- ( ) $\mathsf{coherent}(\hat{\Delta};\hat{\mathcal{Q}})$ *holds iff there exists a coherent* $\hat{\Delta}'$ *and a function* f *such that* $\mathsf{decomp}_f\ \hat{\Delta}' = \hat{\Delta};\hat{\mathcal{Q}}$*.*

5 Typing System and Subject Reduction

This section introduces the type system for processes, connecting the concepts we have covered so far. This is formalised by the following typing judgement ( ):

$$\Gamma \vdash_{\mathcal{D}} P \triangleright_{\mathcal{C}} \mathcal{Q};\ \Delta$$

The intuitive meaning of the judgement is that process P is well-typed with values typed by the unrestricted environment Γ, including shared names, which Γ maps to global types. Ongoing sessions are typed by the two linear environments Δ and $\mathcal{Q}$. Process variables

are typed by $\mathcal{D}$, and the linear environment $\mathcal{C}$ ensures that each queue address is unique. Each entry $\mathsf{s}[k]$ in $\mathcal{C}$ acts as a unique token associated to a queue. We use $\mathcal{C}$ to ensure that queue addresses are unique across a process, e.g., $\mathsf{s}[k] :: \emptyset \mid \mathsf{s}[k] :: \emptyset$. Formally, we define these environments as

$$\text{(Unrestricted)} \quad \Gamma ::= \emptyset \mid \Gamma, a : G \mid \Gamma, x : U \qquad \mathcal{D} ::= \emptyset \mid \mathcal{D}, X : (U, \vec{T})$$

$$\text{(Linear)} \quad \mathcal{C} ::= \emptyset \mid \mathcal{C}, \mathsf{s}[k] \qquad \Delta ::= \emptyset \mid \Delta, s : T \qquad \mathcal{Q} ::= \emptyset \mid \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : Q$$

The environment $\mathcal{Q}$ associates a queue type with each located session, recording what actions have been performed, while Δ associates a local type with each session channel, indicating what actions remain. $\mathcal{C}$ ensures that all queue addresses are disjoint. Note that the environments $\mathcal{Q}$ and Δ may specify multiple sessions, while $\hat{\mathcal{Q}}$ and $\hat{\Delta}$ specifies a single session.

▶ **Definition 18** (Session Filtering and Lifting). *The filtering of Δ by session identifier s, denoted by $\Delta(\mathsf{s})$, is the environment $\hat{\Delta}$ containing the local types of session s in Δ. Similarly, the filtering of $\mathcal{Q}$ by session identifier s, denoted by $\mathcal{Q}(\mathsf{s})$, is the environment $\hat{\mathcal{Q}}$ containing the local types of session s in $\mathcal{Q}$. These operations are always defined and return the empty environment when s is not present.*
Conversely, we define a lifting operation on $\hat{\Delta}$ and $\hat{\mathcal{Q}}$, denoted respectively by $[\hat{\Delta}]_{\mathsf{s}}$ and $[\hat{\mathcal{Q}}]_{\mathsf{s}}$, which yields environments Δ and $\mathcal{Q}$, corresponding to the initial environments where roles p have been replaced with located sessions s^{p}.

Using session filtering, we can extend the definition of coherence to $\Delta; \mathcal{Q}$.

▶ **Definition 19** (Coherence of $\Delta; \mathcal{Q}$ 🐓). *The coherence of the environment $\Delta; \mathcal{Q}$, denoted by $\mathsf{coherent}(\Delta; \mathcal{Q})$, holds iff for any session identifier s, $\Delta(\mathsf{s}); \mathcal{Q}(\mathsf{s})$ is coherent.*
When we, in stating the coherence of $\Delta; \mathcal{Q}$, want to refer to the underlying coherent environment Δ', whose decomposition yields $\Delta; \mathcal{Q}$, we write $\mathsf{coherent}(\Delta; \mathcal{Q})$ as Δ'.

The environments Δ, $\mathcal{Q}$ and $\mathcal{C}$ are linear, and this means different things for each. For Δ and $\mathcal{C}$, linearity carries the standard meaning: each entry in the environment must be used exactly once. In other words, entries cannot be duplicated, such that a local type in Δ is used to type two different processes. The linearity of $\mathcal{Q}$ is based on the partitioning of a queue type, which we explain next:

▶ **Example 20** (Partitioning a queue). Consider two queues with messages from the same p

$$\mathsf{s}[k] :: \mathsf{true}^{\mathsf{p}} \mid \mathsf{s}[k'] :: \mathsf{true}^{\mathsf{p}}$$

The queue type of p is $(k, \mathsf{bool}) \cdot (k', \mathsf{bool})$. We partition the queue type into two queue types, one for each queue, namely (k, bool) and (k', bool).

In the following definition, let $\mathsf{filter}\ f\ Q$ denote the queue type obtained by filtering out all those entries in the channels of Q that do not satisfy the predicate $f : k \to \{\mathsf{True}, \mathsf{False}\}$. Partitioning is then defined as:

▶ **Definition 21** (Partitioning 🐓). *Partitioning of a queue type by a predicate $f : k \to \mathsf{bool}$, denoted by $\mathsf{partition} : (k \to \{\mathsf{True}, \mathsf{False}\}) \to Q \to Q \times Q$, is defined as:*

$$\mathsf{partition}^f(Q) = (\mathsf{filter}\ f\ Q, \mathsf{filter}\ (\lambda k.\ \neg f(k))\ Q)$$

We write $(Q_0, Q_1) \hookrightarrow Q$ when there exists a predicate f such that $\mathsf{partition}^f(Q) = (Q_0, Q_1)$ and we point-wise extend this to environments, abusing notation by writing $(\mathcal{Q}_0, \mathcal{Q}_1) \hookrightarrow \mathcal{Q}$.

$$\frac{\Gamma(x) = S}{\Gamma \vdash x : S} \qquad \Gamma \vdash b : \mathsf{bool} \qquad \frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash e' : \mathsf{bool}}{\Gamma \vdash e \text{ and } e' : \mathsf{bool}}$$

Figure 9 Typing rules for expressions ( ).

The terminated process $\mathbf{0}$ and the empty queue $\mathsf{s}[k] :: \emptyset$ are typed using the following definitions for terminated environments:

▶ **Definition 22** (Terminated Environments). *A local type environment is terminated, denoted by* $\mathsf{ends}(\Delta)$ *iff* $\forall(\mathsf{p} : T) \in \Delta.\ \mathit{unf}(T) = \mathsf{end}$. *A queue environment is terminated, denoted by* $\mathsf{ends}(\mathcal{Q})$ *iff* $\forall(\mathsf{s}^{\mathsf{p}} : Q) \in \mathcal{Q}.\ Q = \epsilon$

Typing Rules. The rules defining our type system can be found in Figures 9, 10, and 11. Figure 9 contains the typing rules for expressions. The typing rules for processes are split into two parts: the first part is presented in Figure 10, where all rules include the premise $\mathsf{ends}(\mathcal{Q})$, which is omitted in the figure to conserve space. [T-MCast] and [T-MAcc] type session request and accept, respectively. Roles are totally ordered, and the largest role of G is n which is assigned to the requesting process. The unfolding of its projection is inserted into Δ. [T-Send] and [T-Rcv] are the rules for sending and receiving. The session channel s is mapped to a local type in Δ and checked to correspond with the action of the process. In the typing of the subterm P, the s entry in Δ is updated to the unfolded continuation $\mathsf{unf}\ T$. [T-Deleg] and [T-SRec] type session delegation and reception. Similarly to the rules for send and receive, they perform the delegation of session channel s', which requires the type T_2 for s' in the environment to be coinductively equal to the carried type T_1 in the type of s. Here, coinductive equality is necessary to ensure the type system is closed under $\approx$. The rules [T-Sel] and [T-Branch] are similar to [T-Send] and [T-Rcv]. [T-IF] is the rule for if statements, which has no effect on the type environments. [T-Inact] and [T-QNil] are the rules for inaction and the empty queue, using the predicates $\mathsf{ends}(\Delta)$ and $\mathsf{ends}(\mathcal{Q})$ to check that the respective environments are terminated.

The remaining rules are in Figure 11, where the premises of all rules are explicitly stated. [T-Conc] types parallel composition and implements the split of the environments as discussed above. [T-NRes] and [T-CRes] are the rules for name and channel restriction. [T-NRes] introduces a coherent global type into Γ. [T-CRes] adds the decomposed environment $\hat{\Delta}; \hat{\mathcal{Q}}$ into $\Delta; \mathcal{Q}$ and extends $\mathcal{C}$ by a disjoint set of unique tokens $\{\mathsf{s}[k_i]\}_{i \in I}$. The tokens in $\mathcal{C}$ are used to ensure the uniqueness of queue addresses such that if $\mathsf{s}_k \in \mathsf{dom}\ \mathcal{C}$, then exactly one queue of shape $\mathsf{s}[k] :: \tilde{h}$ exists in the typed process. [T-QVal], [T-QSess], and [T-QSel] are type queues. A message is annotated by the role p that sent the message, which is used to identify the queue type in $\mathcal{Q}$ associated with the located session s^{p}, checking that k and $\mathcal{U}$ in the queue type correspond to the channel of the queue and the message it contains. [T-Var] is the rule for procedure call, calling procedure X with an expression and a list of distinct session channels. Here, we write $\tilde{s} : \tilde{T}$ to mean then environment with entries $(\tilde{s}_i : \tilde{T}_i)$ for $i \in \{1, .., |\tilde{s}|\}$, where $|\tilde{s}| = |\tilde{T}|$.

Our semantics uses a fixed set of definitions D in $\rightarrow_D$. We now show how to type them.

▶ **Definition 23** (Typing of declarations). *Declarations D are typed by the environment $\mathcal{D}$ whenever* $\mathsf{dom}\ D = \mathsf{dom}\ \mathcal{D}$ *and*

$$\forall X(x\tilde{\jmath}) = P \in D. \quad X : (U, \tilde{T}) \in \mathcal{D} \text{ implies } x : U \vdash_{\mathcal{D}} P \triangleright_{\emptyset} \tilde{\jmath} : \tilde{T};\ \emptyset$$

$$\frac{\Gamma \vdash a : G \quad \begin{array}{l}\mathsf{roles}(G) = \{0, \ldots, \mathsf{n}\} \\ \mathsf{channels}(G) = \{1, \ldots, k\}\end{array} \quad \Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, \mathfrak{s} : \mathsf{unf}(G\restriction_{\mathsf{n}});\ \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} \overline{a}\,[\mathsf{n}]_k(\mathfrak{s}).P \triangleright_{\emptyset} \Delta;\ \mathcal{Q}}\ [\text{T-MCast}]$$

$$\frac{\Gamma \vdash a : G \quad \mathsf{p} \in \mathsf{roles}(G) \quad \Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, \mathfrak{s} : \mathsf{unf}(G\restriction_{\mathsf{p}});\ \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} a[\mathsf{p}](\mathfrak{s}).P \triangleright_{\emptyset} \Delta;\ \mathcal{Q}}\ [\text{T-MAcc}]$$

$$\frac{\Gamma \vdash e : U \quad \Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, s : \mathsf{unf}(T);\ \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} s[k]!\langle e\rangle; P \triangleright_{\emptyset} \Delta, s :! k\langle U\rangle.T;\ \mathcal{Q}}\ [\text{T-Send}] \qquad \frac{\Gamma, x : U \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, s : \mathsf{unf}(T);\ \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} s[k]?(x); P \triangleright_{\emptyset} \Delta, s :? k\langle U\rangle.T;\ \mathcal{Q}}\ [\text{T-Rcv}]$$

$$\frac{T_1 \approx T_2 \quad s \neq s' \quad \Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, s : \mathsf{unf}(T);\ \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} s[k]!\langle\langle s'\rangle\rangle; P \triangleright_{\emptyset} \Delta, s :! k\langle T_1\rangle.T, s' : T_2;\ \mathcal{Q}}\ [\text{T-Deleg}]$$

$$\frac{\Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, s : \mathsf{unf}(T), t : \mathsf{unf}(T');\ \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} s[k]?((t)); P \triangleright_{\emptyset} \Delta, s :? k\langle T'\rangle.T;\ \mathcal{Q}}\ [\text{T-SRec}]$$

$$\frac{i \in I \quad \Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, s : \mathsf{unf}(T_i);\ \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} s[k] \lhd l_i; P \triangleright_{\emptyset} \Delta, s : k \oplus \{l_j : T_j\}_{j\in I};\ \mathcal{Q}}\ [\text{T-Sel}]$$

$$\frac{\forall i \in I \quad \Gamma \vdash_{\mathscr{D}} P_i \triangleright_{\emptyset} \Delta, s : \mathsf{unf}(T_i);\ \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} s[k] \rhd \{l_j : P_j\}_{j\in I} \triangleright_{\emptyset} \Delta, s : k \,\&\, \{l_j : T_j\}_{j\in I};\ \mathcal{Q}}\ [\text{T-Branch}]$$

$$\frac{\Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta;\ \mathcal{Q} \quad \Gamma \vdash_{\mathscr{D}} Q \triangleright_{\emptyset} \Delta;\ \mathcal{Q} \quad \Gamma \vdash e : \mathsf{bool}}{\Gamma \vdash_{\mathscr{D}} \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleright_{\emptyset} \Delta;\ \mathcal{Q}}\ [\text{T-If}]$$

$$\frac{\mathsf{ends}(\Delta_0) \quad \Gamma \vdash e : U \quad X : (U, \tilde{T}) \in \mathscr{D} \quad \Delta_1 \approx \tilde{s} : \tilde{T}}{\Gamma \vdash_{\mathscr{D}} X\langle e, \tilde{s}\rangle \triangleright_{\mathcal{C}} \Delta_0, \Delta_1;\ \mathcal{Q}}\ [\text{T-Var}] \qquad \frac{\mathsf{ends}(\Delta)}{\Gamma \vdash_{\mathscr{D}} \mathbf{0} \triangleright_{\emptyset} \Delta;\ \mathcal{Q}}\ [\text{T-Inact}]$$

Figure 10 Typing rules (🏃): all rules implicitly include the premise $\mathsf{ends}(\mathcal{Q})$.

▶ **Example 24** (Typing process definitions). Let D contain the single declaration $X(x\mathfrak{s}) = \mathfrak{s}[k]!\langle x\rangle; X\langle x, \mathfrak{s}\rangle$, which is typed by the environment $X : \mathsf{bool} \times \mathsf{unf}(T)$, where $T = \mu\mathbf{t}.!k\langle\mathsf{bool}\rangle.\mathbf{t}$. We name this environment $\mathscr{D}$ and to assert that D is typed by $\mathscr{D}$, we must show:

$$x : \mathsf{bool} \vdash_{\mathscr{D}} \mathfrak{s}[k]!\langle x\rangle; X\langle x, \mathfrak{s}\rangle \triangleright_{\emptyset} \mathfrak{s} : \mathsf{unf}(T);\ \emptyset$$

Which is derivable by [T-Send] and and [T-Var].

The example above highlights an important aspect of the type system. All local types in Δ are unfolded. Note for example that we put the unfolded local type in $\mathscr{D}$. This is necessary because all rules assume local types are unfolded, and we preserve this property by only introducing unfolded local types into Δ.

Subject Reduction. We now state the properties of the mechanised type system. We start with the substitution lemma, which is required by the subject reduction theorem to derive typings for value substituted processes $P[v/x]$ and located session substituted processes $P[\mathsf{s}^{\mathsf{p}}/t]$.

▶ **Lemma 25** (Substitution 🏃).
1. *If* $\Gamma, x : U \vdash_{\mathscr{D}} P \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}$ *and* $\Gamma \vdash v : U$ *then* $\Gamma \vdash_{\mathscr{D}} P[v/x] \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}$.
2. *If* $\Gamma \vdash_{\mathscr{D}} P \triangleright_{\mathcal{C}} \Delta, \mathfrak{s} : T;\ \mathcal{Q}$ *then* $\Gamma \vdash_{\mathscr{D}} P[\mathsf{s}^{\mathsf{p}}/\mathfrak{s}] \triangleright_{\mathcal{C}} \Delta, \mathsf{s}^{\mathsf{p}} : T;\ \mathcal{Q}$.

$$\frac{\Gamma \vdash_{\mathscr{D}} P \triangleright_{c_0} \Delta_0;\ \mathcal{Q}_0 \quad \Gamma \vdash_{\mathscr{D}} Q \triangleright_{c_1} \Delta_1;\ \mathcal{Q}_1 \quad (\mathcal{Q}_0, \mathcal{Q}_1) \hookrightarrow \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} P \mid Q \triangleright_{c_0, c_1} \Delta_0, \Delta_1;\ \mathcal{Q}} \text{ [T-Conc]}$$

$$\frac{\mathsf{ends}(\mathcal{Q}) \quad \mathsf{ends}(\Delta)}{\Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: \emptyset \triangleright_{\mathsf{s}[k]} \Delta;\ \mathcal{Q}} \text{ [T-QNil]} \qquad \frac{\mathsf{coherent}(G) \quad \Gamma, \mathsf{a} : G \vdash_{\mathscr{D}} P \triangleright_C \Delta;\ \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} (\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a}) P \triangleright_{\emptyset} \Delta;\ \mathcal{Q}} \text{ [T-NRes]}$$

$$\frac{\mathsf{coherent}(\hat{\Delta}; \hat{\mathcal{Q}}) \quad \Gamma \vdash_{\mathscr{D}} P \triangleright_{(c, \{\mathsf{s}[k_i]\}_{i \in I})} \Delta, [\hat{\Delta}]_{\mathsf{s}};\ \mathcal{Q}, [\hat{\mathcal{Q}}]_{\mathsf{s}}}{\Gamma \vdash_{\mathscr{D}} (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s}) P \triangleright_c \Delta;\ \mathcal{Q}} \text{ [T-CRes]}$$

$$\frac{\Gamma \vdash v : U \quad \Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: \tilde{h} \triangleright_c \Delta;\ \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : Q}{\Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: v^{\mathsf{p}} \cdot \tilde{h} \triangleright_c \Delta;\ \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : (k, U) \cdot Q} \text{ [T-QVal]}$$

$$\frac{\Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: \tilde{h} \triangleright_c \Delta;\ \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : Q}{\Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: l^{\mathsf{p}} \cdot \tilde{h} \triangleright_c \Delta;\ \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : (k, l) \cdot Q} \text{ [T-QSel]}$$

$$\frac{T_0 \approx T_1 \quad \Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: \tilde{h} \triangleright_c \Delta;\ \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : Q}{\Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: (\mathsf{t}^{\mathsf{q}})^{\mathsf{p}} \cdot \tilde{h} \triangleright_c \Delta, \mathsf{t}^{\mathsf{q}} : T_0;\ \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : (k, T_1) \cdot Q} \text{ [T-QSess]}$$

Figure 11 Typing rules (🐓): there are no implicit premises for these rules.

The lemma states that processes remain well-typed after variables are substituted for values and located session variables are substituted for located session identifiers. This lemma is necessary for proving the cases of subject reduction dealing with value and session reception.

We are now ready to state subject congruence and subject reduction:

▶ **Theorem 26** (Subject congruence and reduction). *Let P, P', and Q be processes, and Γ, Δ, Δ_0, and $\mathcal{Q}$ environments. Then,*

1. (🐓) *if $P \equiv Q$ and $\Gamma \vdash_{\mathscr{D}} P \triangleright_c \Delta;\ \mathcal{Q}$ then $\Gamma \vdash_{\mathscr{D}} Q \triangleright_c \Delta;\ \mathcal{Q}$*
2. (🐓) *if $\Gamma \vdash_{\mathscr{D}} P \triangleright_c \Delta;\ \mathcal{Q}$, $\mathsf{coherent}(\Delta; \mathcal{Q})$ as Δ_0, and $P \ \rightarrow_D \ P'$ then there exists $\Delta_1, \Delta', \mathcal{Q}'$ s.t. $\mathsf{coherent}(\Delta'; \mathcal{Q}')$ as Δ_1 and $\Gamma \vdash_{\mathscr{D}} P' \triangleright_c \Delta';\ \mathcal{Q}'$ and either $\Delta_0 = \Delta_1$ or there exists ℓ s.t. $\Delta_0 \xrightarrow{\ell} \Delta_1$*
3. (🐓) *if $\Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \emptyset;\ \emptyset$ and $P \ \rightarrow_D \ P'$ then $\Gamma \vdash_{\mathscr{D}} P' \triangleright_{\emptyset} \emptyset;\ \emptyset$*

The theorem contains three statements: (1) typing is preserved by congruence; (2) typing is preserved by process reductions, relating environments either by equality or reduction; and, (3) typing is preserved by closed processes with no queues , which is a special case of (2). The proof of (1) proceeds by structural induction on the congruence relation, while the proof of (2), the core subject reduction result, is by induction on process reductions.

▶ Remark 27 (Differences with the original work). The type system enforces linear resource use by defining smaller resources as splits of larger ones, contrasting with Honda et al., who define larger resources as compositions of smaller ones. Our focus on splitting rather than composing is evident in several aspects of the type system. Firstly, the environment $\Delta; \mathcal{Q}$ provides an intuitive temporal split: queue types in $\mathcal{Q}$ indicate completed actions, while residual local types in Δ indicate remaining actions. Keeping these concepts distinct simplifies the definition of environment splitting, as Δ and $\mathcal{Q}$ split orthogonally. Secondly, queue types and residual local types are obtained by decomposition. Our definition allows us to avoid using the subtyping relation, unlike Honda et al. This is significant because introducing subtyping through a non-structural subsumption rule complicates reasoning about type derivations, particularly in proof assistants.

$$\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]?(x); P \mid \mathsf{s}^{\mathsf{q}}[k]?(x); Q) \qquad \mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]!\langle e\rangle; P \mid \mathsf{s}^{\mathsf{q}}[k]!\langle e'\rangle; Q)$$
$$\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]?(x); P \mid \mathsf{s}^{\mathsf{q}}[k] \rhd \{l_j\colon Q_j\}_{i\in J}) \qquad \mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]!\langle e\rangle; P \mid \mathsf{s}^{\mathsf{q}}[k] \lhd l; Q)$$
$$\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]?(x); P \mid \mathsf{s}^{\mathsf{q}}[k]?((\jmath)); Q) \qquad \mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]!\langle e\rangle; P \mid \mathsf{s}^{\mathsf{q}}[k]!\langle\!\langle t\rangle\!\rangle; Q)$$
$$\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k] \rhd \{l_i\colon P_i\}_{i\in I} \mid \mathsf{s}^{\mathsf{q}}[k] \rhd \{l_j\colon Q_j\}_{j\in J}) \qquad \mathbf{error}(\mathsf{s}^{\mathsf{p}}[k] \lhd l;_1 P \mid \mathsf{s}^{\mathsf{q}}[k] \lhd l;_2 Q)$$
$$\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k] \rhd \{l_i\colon P_i\}_{i\in I} \mid \mathsf{s}^{\mathsf{q}}[k]?((\jmath)); Q) \qquad \mathbf{error}(\mathsf{s}^{\mathsf{p}}[k] \lhd l; P \mid \mathsf{s}^{\mathsf{q}}[k]!\langle\!\langle t\rangle\!\rangle; Q)$$
$$\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]?((\jmath)); P \mid \mathsf{s}^{\mathsf{q}}[k]?((\jmath)); Q) \qquad \mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]!\langle\!\langle t\rangle\!\rangle; P \mid \mathsf{s}^{\mathsf{q}}[k]!\langle\!\langle u\rangle\!\rangle; Q)$$

Figure 12 The **error** relation (🐓): violation of linearity (race on the same channel).

$$\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]?(x); P \mid \mathsf{s}[k] :: l_j{}^{\mathsf{q}} \cdot \tilde{h})$$
$$\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]?(x); P \mid \mathsf{s}[k] :: (\mathsf{t}^{\mathsf{r}})^{\mathsf{q}} \cdot \tilde{h})$$
$$\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k] \rhd \{l_i\colon P_i\}_{i\in I} \mid \mathsf{s}[k] :: v^{\mathsf{q}} \cdot \tilde{h})$$
$$\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k] \rhd \{l_i\colon P_i\}_{i\in I} \mid \mathsf{s}[k] :: (\mathsf{t}^{\mathsf{r}})^{\mathsf{q}} \cdot \tilde{h})$$
$$\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]?((\jmath)); P \mid \mathsf{s}[k] :: v^{\mathsf{q}} \cdot \tilde{h})$$
$$\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]?((\jmath)); P \mid \mathsf{s}[k] :: l_j{}^{\mathsf{q}} \cdot \tilde{h})$$

$$\frac{\mathbf{error}(P)}{\mathbf{error}((\nu^{\mathsf{a}}\mathsf{a})P)} \qquad \frac{\mathbf{error}(P)}{\mathbf{error}((\nu^{\mathsf{s}}\mathsf{s})P)} \qquad \frac{\mathbf{error}(P)}{\mathbf{error}(P \mid Q)}$$

Figure 13 The **error** relation (🐓): communication error and structural rules.

Communication Safety. We conclude this section by presenting communication safety, the property that states that a well-typed process never goes wrong. We first define a notion of error, in order to capture processes behaviour that we consider erroneous:

▶ **Definition 28** (**error**$_{\equiv}$ 🐓). *The relation* **error** *on processes is the minimal relation satisfying the rules in Figure 12 and Figure 13. Then,* $\mathbf{error}_{\equiv}(P)$ *is defined as* $\exists Q.\ P \equiv Q \wedge \mathbf{error}(Q)$.

The **error** relation captures situations in which processes cause a run-time error. Following the original work, we have categorised errors into two groups. The rules in Figure 12 address situations where linearity is violated. In session types, linearity ensures that there is never a race condition on a session channel. Such an error situation can arise in different ways, notably when two inputs or outputs competing on the same channel are ready to be executed (recall the notion of linearity of global types). The second type of error is a communication error, where a process reads from a queue a message of the wrong kind, e.g., it reads a label when a delegated session is expected. These cases are presented in Figure 13, along with three structural rules that permit lifting the relation under restriction or parallel composition.

The predicate $\mathbf{error}_{\equiv}(P)$ lifts the error relation to structural congruence in order to capture different structurally congruent processes. This design choice (as opposed to including a structural congruence rule in the definition of **error**) was made to simplify our mechanised proof of communication safety. Note that we can always rewrite a process into the form $(\nu x_1 \ldots x_n)(P \mid Q \mid R)$, where P and Q are the components to be checked for errors. Restriction rules are then applied to eliminate binders, and parallel rules are used to strip away the R component. We formally state communication safety as:

▶ **Theorem 29** (Communication Safety 🐓). *If* $\Gamma \vdash_{\mathcal{D}} P \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}$ *and* $\mathsf{coherent}(\Delta; \mathcal{Q})$ *as* Δ_0 *then it is never the case that* $\mathbf{error}_{\equiv}(P)$.

From the previous theorem and subject reduction, we obtain the following:

▶ **Corollary 30** (Safety Preservation 🐓). *If* $\Gamma \vdash_{\mathcal{D}} P \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}$, coherent$(\Delta;\mathcal{Q})$ *as* Δ_0, *and* $P \ \rightarrow_D \ P'$ *then it is never the case that* $\mathbf{error}_{\equiv}(P')$.

▶ Remark 31 (Differences with the original work). Our approach to defining communication safety differs from that of the original work by Honda et al. While we structurally define what constitutes an erroneous process and negate this property to derive safety, they define safety directly in terms of available redexes in the processes. In their approach to safety, when a redex is present in a session, session channels must occur linearly, and the messages read from queues must conform to the expectations of the receiving process. We departed from their non-structural approach in favour of an error predicate, as this proved more convenient for mechanisation, allowing us to perform induction on the predicate.

Given that the original formulation is not entirely formal, we have not mechanised the original definition nor demonstrated whether the two approaches are equivalent. However, it is worth noting that the use of an error predicate for defining safety is fairly standard and, for example, was employed in the original work on binary session types by Honda et al. [31].

6 Related Work and Discussion

Related Work. We discuss the most relevant related work, with a particular focus on mechanisations of session types.

Mechanisation of multiparty session types. Castro-Perez et al. [13] mechanise in Coq a trace equivalence between processes, coinductive local types and coinductive global types. Their processes, equipped with a trace semantics and no structural congruence, use implicit channels and are single-session without delegation; therefore, there is no multi-cast session request nor session acceptance and a process is always typed by a single global type. As a consequence, subject reduction is replaced by showing that processes and types have the same traces. Their process language is a domain specific language embedded in Coq. This allows a user to write a correct-by-construction process in Coq for each role of a session, and use Coq's extraction feature to extract executable OCaml code. The generated code has a transport API that allows interaction with external code. While their setting is more applied than ours, they also propose an elegant approach to representing session types as coinductive types that are coinductively related to unfolded μ-types. Castro-Perez et al. prove their definition of projection sound with respect to a coinductive specification of projection by Ghilezan et a. [24]. For a similar setting but with explicit channels, Tirore et al. [51] propose a projection for which both soundness and completeness hold: this is the projection that we adopt in our mechanisation.

Jacobs et al. [35] mechanise multiparty GV in Coq using the Iris framework [37]. Multiparty GV is an extension of the linear lambda calculus with multiparty asynchronous session types with implicit channels. Their language, equipped with a thread pool semantics and no structural congruence, features an n-ary fork operator that combines session initiation and process spawning, ensuring an acyclic topology, therefore ensuring progress [8]. This is unlike our setting where already existing processes can initiate sessions on shared names. Their methodology is significantly different from ours, and is based on separation logic [45, 46] and connectivity graphs [34]. They define an invariant on runtime configurations, ensuring well-typedness and buffer consistency. They then prove preservation of this invariant under operational semantics and show that configurations satisfying the invariant cannot deadlock.

Mechanisation of binary session types. Using the Iris framework, Hinrichsen et al. [29] introduce semantic typing for a variant of binary session types using logical relations. Gay et al. [23] compare duality definitions in an Agda mechanisation of binary session types.

Duality is the binary notion of compatibility used for binary session types, that is a special case of the multiparty notion of coherence. Gay et al. demonstrate that some definitions are unsound when type variables can appear in messages. Like them, our work does not take the equi-recursive view and we explicitly state when syntactic equality of μ-types is used and thus when to trigger unfoldings and when we use coinductive equality modulo unfolding. Castro-Perez et al. [14] mechanise in Coq a subject reduction theorem for the binary session type meta-theory given by Yoshida and Vasconcelos [54]. They use the locally nameless representation for variables by Charguéraud [15]. Like us, Castro-Perez et al. distinguish between variable and value session channels. Unlike our setting, theirs does not include recursion nor process call and definitions. Instead of process call, they use replication $!P$. Their definition of structural congruence lacks commutativity and transitivity which, although it makes some simple reductions impossible, simplifies substantially the subject reduction proof. Recently, Sano et al. [48] mechanised a version of binary session types that are in a Curry-Howard correspondence with linear logic [52]. They use linearity predicates, which enable the separate checking of linearity properties in the type system, allowing the typing judgement to focus solely on non-linear contexts.

Session types on paper. There is a vast literature on multiparty session types that builds on the original paper. An introduction was given by Yoshida and Gheri [53] and a comprehensive overview was given by Scalas and Yoshida [49] where they argue the inconvenience of using global types to define compatibility. It should be noted that, to the best of our knowledge, all later work uses implicit channels and, therefore, the counter example does not invalidate these results. However, as argued in the introduction, explicit channels are more expressive and particularly useful in applications where only a limited number of channels are available, e.g., AMQP [3]. A binary version of session types has also been applied to type a choreographic language, i.e., a synchronous process-level language with global interactions that features explicit channels [9, 10]. Like for global types, choreographies must undergo a well-formedness check, which resembles a more restrictive linearity analysis for global types.

Discussion. We discuss our contributions and potential further developments.

On the Mechanisation in Coq. Our mechanisation in Coq consists of over 20K lines of code and features several design choices to address the complexity of the theory of multiparty session types. For handling binders, we employed the *Autosubst2* library, developed in Stark's PhD thesis [50]. From a user-defined signature in higher-order abstract syntax (HOAS), Autosubst2 generates Coq code with an inductive definition that uses a de Bruijn representation, along with substitution operations on this datatype. We reuse the definitions in the work by Tirore et al [51] for representing global types, local types and the projection function. We reused their code off-the-shelf, significantly streamlining our development.

For representing typing environments, we adopted a straightforward approach using lists of pairs $(\text{name}, \text{type})$. Other common forms of representation include lists with indices or functions from names to types. We found our approach more convenient to use for parallel substitutions, which is the kind of substitution operation that Autosubst2 is based on. In particular, while low-level, it simplified context splitting for parallel composition rules, requiring only standard checks for linearity of name usage, as dictated by linear type systems. We believe this representation offers an accessible starting point for researchers aiming to mechanise session types. Finally, our work heavily relies on a combination of *inductive* and *coinductive* definitions, which we implemented using the Paco library [16], which proved invaluable for developing our mechanisation.

On Communication Safety, Session Fidelity, and Progress. Subject reduction serves as a critical internal lemma that can be used to prove many key results, including communication safety (also known as type safety), session fidelity, and progress. The primary goal of this paper is to prove subject reduction and uncover the issues in the original theorem by Honda et al., demonstrating its incorrectness; establishing communication safety, which ensures that well-typed processes do not exhibit wrong behaviour, was an additional achievement. On the other hand, other results, though highly interesting, lie beyond the intended scope of this work. Session fidelity expresses that communications within a session progress according to the global type. While we have not mechanised session fidelity as a formal theorem, addressing this remains a promising avenue for future research. Proving session fidelity would require augmenting the reduction semantics with labels to specify the communication performed and the channels involved, enabling a finer-grained analysis of interaction behaviours.

Another key result from Honda et al. is progress, which asserts that a well-typed single-session process never deadlocks. In our work, linearity of global types plays a crucial role in ensuring that interleavings of send and receive operations adhere to the global type. However, our counterexample raises questions about the implications of explicit channels and causality on progress. Investigating these implications and formally proving progress for our system would further strengthen our meta-theoretical framework.

Finally, our results offer a foundation for extending our work to mechanise more advanced properties of session-typed systems, particularly in multi-session scenarios. Explicit channels, as demonstrated by our counterexample, introduce non-trivial distinctions in causality that deserve further exploration.

Alternative Semantics for Global and Local Types. The counterexample highlighted the restrictiveness of the global type semantics. Later work on multiparty session types [49, 13] introduces a more flexible global type semantics that, in the context of implicit channels, is unrelated to the counterexample. This work suggests semantic definitions that could allow the unstuck predicate to be removed from the meta-theory.

Other Observations. The mechanisation revealed some inconveniences in the meta-theory. For example, the representation of a finished session channel is ambiguous; it can be represented either as not being contained in the environment or as being contained but mapped to `end`. We conjecture that it would simplify proofs to resolve this ambiguity by choosing only the non-presence representation in conjunction with a closure operation on sessions, as done by Caires et al. [7] and Wadler [52].

Another inconvenience is the use of global types as the basis for multiparty compatibility. Scalas and Yoshida [49] pointed out that more general notions of compatibility exist beyond global types. This broader perspective may not only provide a more encompassing definition of compatibility but also simplify proofs by eliminating the need for two specification languages in the meta-theory.

References

1 Scribble: Describing Multy Party Protocols. URL: `http://www.scribble.org`.

2 Session Types in Programming Languages: A Collection of Implementations. URL: `http://groups.inf.ed.ac.uk/abcd/session-implementations.html`.

3 Advanced Message Queuing Protocol. `http://www.iona.com/opensource/amqp/`, 2015.

4 Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In Franck van Breugel and Marsha Chechik, editors, *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings*, volume 5201 of *Lecture Notes in Computer Science*, pages 418–433. Springer, 2008. `doi:10.1007/978-3-540-85361-9_33`.

5 Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. A formally-verified C compiler supporting floating-point arithmetic. In Alberto Nannarelli, Peter-Michael Seidel, and Ping Tak Peter Tang, editors, *21st IEEE Symposium on Computer Arithmetic*, pages 107–115. IEEE Computer Society, 2013. doi:10.1109/ARITH.2013.30.

6 Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. Undecidability of asynchronous session subtyping. *Inf. Comput.*, 256:300–320, 2017. doi:10.1016/J.IC.2017.07.010.

7 Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Math. Struct. Comput. Sci.*, 26(3):367–423, 2016. doi:10.1017/S0960129514000218.

8 Marco Carbone and Søren Debois. A graphical approach to progress for structured communication in web services. In Simon Bliudze, Roberto Bruni, Davide Grohmann, and Alexandra Silva, editors, *Proceedings Third Interaction and Concurrency Experience: Guaranteed Interaction, ICE 2010, Amsterdam, The Netherlands, 10th of June 2010*, volume 38 of *EPTCS*, pages 13–27, 2010. doi:10.4204/EPTCS.38.4.

9 Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In Rocco De Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2007. doi:10.1007/978-3-540-71316-6_2.

10 Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8:1–8:78, 2012. doi:10.1145/2220365.2220367.

11 Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, volume 59 of *LIPIcs*, pages 33:1–33:15, Germany, 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CONCUR.2016.33.

12 Marco Carbone, Sonia Marin, and Carsten Schürmann. A logical interpretation of asynchronous multiparty compatibility. In Robert Glück and Bishoksan Kafle, editors, *Logic-Based Program Synthesis and Transformation - 33rd International Symposium, LOPSTR 2023, Cascais, Portugal, October 23-24, 2023, Proceedings*, volume 14330 of *Lecture Notes in Computer Science*, pages 99–117. Springer, 2023. doi:10.1007/978-3-031-45784-5_7.

13 David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. In *Proceedings of PLDI*, pages 237–251. ACM, 2021. doi:10.1145/3453483.3454041.

14 David Castro-Perez, Francisco Ferreira, and Nobuko Yoshida. EMTST: engineering the meta-theory of session types. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*, volume 12079 of *Lecture Notes in Computer Science*, pages 278–285. Springer, 2020. doi:10.1007/978-3-030-45237-7_17.

15 Arthur Charguéraud. The locally nameless representation. *J. Autom. Reason.*, 49(3):363–408, 2012. doi:10.1007/S10817-011-9225-2.

16 Chung-Kil, Georg Neis Georg, Derek Dreyer, and Victor Vafeiadis. The power of parameterization in coinductive proof. *SIGPLAN Notices*, 48(1):193–206, 2013. doi:10.1145/2480359.2429093.

17 Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *MSCS*, 760:1–65, 2015.

18 Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, declaratively. In Claude Bolduc, Josée Desharnais, and Béchir Ktari, editors, *Mathematics of Program Construction, 10th International Conference, MPC 2010, Québec City, Canada, June 21-23, 2010. Proceedings*, volume 6120 of *Lecture Notes in Computer Science*, pages 100–118. Springer, 2010. doi:10.1007/978-3-642-13321-3_8.

19 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 194–213. Springer, 2012. doi:10.1007/978-3-642-28869-2_10.

20 Simon Fowler. An erlang implementation of multiparty session actors. In Massimo Bartoletti, Ludovic Henrio, Sophia Knight, and Hugo Torres Vieira, editors, *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016*, volume 223 of *EPTCS*, pages 36–50, 2016. doi:10.4204/EPTCS.223.3.

21 Simon J. Gay and Malcolm Hole. Types and subtypes for client-server interactions. In S. Doaitse Swierstra, editor, *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*, volume 1576 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 1999. doi:10.1007/3-540-49099-X_6.

22 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005. doi:10.1007/S00236-005-0177-Z.

23 Simon J. Gay, Peter Thiemann, and Vasco T. Vasconcelos. Duality of session types: The final cut. In Stephanie Balzer and Luca Padovani, editors, *Proceedings of the 12th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2020, Dublin, Ireland, 26th April 2020*, volume 314 of *EPTCS*, pages 23–33, 2020. doi:10.4204/EPTCS.314.3.

24 Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. *Journal of Logical and Algebraic Methods in Programming*, 104:127–173, 2019. doi:10.1016/j.jlamp.2018.12.002.

25 Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur, editor, *Computer Mathematics, 8th Asian Symposium, ASCM*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007. doi:10.1007/978-3-540-87827-8_28.

26 Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP*, volume 7998 of *LNCS*, pages 163–179. Springer, 2013. doi:10.1007/978-3-642-39634-2_14.

27 Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason M. Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the kepler conjecture. *CoRR*, abs/1501.02155, 2015. arXiv:1501.02155.

28 Jesse Michael Han and Floris van Doorn. A formal proof of the independence of the continuum hypothesis. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 353–366. ACM, 2020. doi:10.1145/3372885.3373826.

29 Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. Machine-checked semantic session typing. In Catalin Hritcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 178–198. ACM, 2021. doi:10.1145/3437992.3439914.

30 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

31 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

32 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of POPL*, pages 273–284. ACM, 2008. doi:10.1145/1328438.1328472.

33 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of the ACM*, 63(1):9:1–9:67, 2016. doi:10.1145/2827695.

34 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Connectivity graphs: a method for proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang.*, 6(POPL):1–33, 2022. doi:10.1145/3498662.

35 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proceedings of the ACM on Programming Languages*, 6(ICFP):466–495, 2022. doi:10.1145/3547638.

36 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an os kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220. ACM, 2009. doi:10.1145/1629575.1629596.

37 Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 205–217. ACM, 2017. doi:10.1145/3009837.3009855.

38 Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in go using behavioural types. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1137–1148. ACM, 2018. doi:10.1145/3180155.3180157.

39 Julien Lange and Nobuko Yoshida. On the undecidability of asynchronous session subtyping. In Javier Esparza and Andrzej S. Murawski, editors, *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10203 of *Lecture Notes in Computer Science*, pages 441–457, 2017. doi:10.1007/978-3-662-54458-7_26.

40 SIPLAN selection committee. Most Influential POPL Paper Award (2018). https://www.sigplan.org/Awards. Accessed: July 2024.

41 The Coq development team. The Coq Proof Assistant. https://coq.inria.fr. Accessed: July 2024.

42 The Lean development team. The Lean Proof Assistant. https://lean-lang.org. Accessed: July 2024.

43 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.

44 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-45949-9.

45 Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bull. Symb. Log.*, 5(2):215–244, 1999. doi:10.2307/421090.

46 Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001. doi:10.1007/3-540-44802-0_1.

47 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

48 Chuta Sano, Ryan Kavanagh, and Brigitte Pientka. Mechanizing session-types using a structural view: Enforcing linearity without linearity. *Proc. ACM Program. Lang.*, 7(OOPSLA2):374–399, 2023. doi:10.1145/3622810.

49 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, 2019. doi:10.1145/3290343.

50 Kathrin Stark. *Mechanising Syntax with Binders in Coq*. Phd thesis, Universität des Saarlandes, Fakultät für Mathematik und Informatik, Saarbrücken, Germany, 2019.

51 Dawit Legesse Tirore, Jesper Bengtson, and Marco Carbone. A sound and complete projection for global types. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland*, volume 268 of *LIPIcs*, pages 28:1–28:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.ITP.2023.28.

52 Philip Wadler. Propositions as sessions. In *Proceedings of ICFP*, pages 273–286. ACM, 2012. doi:10.1145/2364527.2364568.

53 Nobuko Yoshida and Lorenzo Gheri. A very gentle introduction to multiparty session types. In Dang Van Hung and Meenakshi D'Souza, editors, *Distributed Computing and Internet Technology - 16th International Conference, ICDCIT 2020, Bhubaneswar, India, January 9-12, 2020, Proceedings*, volume 11969 of *Lecture Notes in Computer Science*, pages 73–93. Springer, 2020. doi:10.1007/978-3-030-36987-3_5.

54 Nobuko Yoshida and Vasco Thudichum Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. In Maribel Fernández and Claude Kirchner, editors, *Proceedings of the First International Workshop on Security and Rewriting Techniques, SecReT@ICALP 2006, Venice, Italy, July 15, 2006*, volume 171(4) of *Electronic Notes in Theoretical Computer Science*, pages 73–93. Elsevier, 2006. doi:10.1016/J.ENTCS.2007.02.056.

Compositional Static Value Analysis for Higher-Order Numerical Programs

Milla Valnet
LIP6, Sorbonne Université, F-75005, Paris, France

Raphaël Monat
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Antoine Miné
LIP6, Sorbonne Université, F-75005, Paris, France

Abstract

Static analyzers have been successfully developed to detect runtime errors in many languages. However, the automatic analysis of functional languages remains a challenge due to their recursive functions, recursive algebraic data types, and higher-order functions. Classic type systems provide compositional methods that are in general not precise enough to prove the absence of runtime errors such as assertion failures. At the other end of the spectrum, deductive methods are more expressive but may require user guidance to prove invariants.

Our work describes a static value analysis by abstract interpretation for a higher-order pure functional language. This analysis provides a sound and automatic approach to discover invariants and prevent assertion and match failures. We have designed a compositional analysis: functions are analyzed only once, at their definition site, generating a summary of their behavior. The summaries can be viewed as input-output relations expressed with relational abstract domains. We present two new abstract domains. A first abstract domain summarizes recursive algebraic data types. A second abstract domain lifts existing disjunctive relational summaries to higher-order by formalizing them as domains able to abstract higher-order functions. Both abstractions are parameterized by the abstractions of basic types (strings, integers, ...). Thanks to this parametric nature, both domains can be combined, allowing the analysis of higher-order functions manipulating algebraic data types and, conversely, algebraic data types using functions as first-class values.

We have implemented this analysis in the open-source MOPSA platform. Preliminary evaluation confirms the precision of our approach on a set of 40 handwritten toy programs as well as 20 programs from the state-of-the-art Salto analyzer benchmark.

2012 ACM Subject Classification Software and its engineering → Automated static analysis; Theory of computation → Program analysis; Software and its engineering → Functional languages

Keywords and phrases Static Value Analysis, Functional Programming, Abstract Interpretation

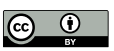
Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.32

Related Version *Full Version*: `https://hal.science/hal-05047369`

Supplementary Material *Software (ECOOP 2025 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.11.2.5`

1 Introduction

Thanks to strong static typing, functional programming languages such as OCaml are safer than traditional imperative languages, as they "reduce the number of runtime errors [...] found in Java and C#" [29]. Still, well-typed functional programs can occasionally go wrong upon encountering runtime errors, such as out-of-bounds array accesses, arithmetic overflows, non-exhaustive pattern matching or uncaught exceptions. In this work, we aim at statically detecting such errors by developing a *compositional static value analysis* and targeting higher-order pure functional languages. Our approach is rooted in the abstract interpretation


© Milla Valnet, Raphaël Monat, and Antoine Miné;
licensed under Creative Commons License CC-BY 4.0

framework, and leverages *relational abstract domains* to yield a high expressivity. In particular, we explore a new compositional value analysis that can yield better precision than classic type systems. The analysis is fully automated, contrary to more expressive approaches traditionally based on deductive verification.

Value analyses based on abstract interpretation have been mostly developed to target imperative programming languages [6, 9]. An ongoing research direction focuses on developing *compositional* (or function-modular) analyses. These analyses consist in analyzing functions once, at their definition site, to infer a summary; the summary is then applied to analyze each function call. These methods significantly improve scalability when a function is called multiple times, as summary applications are computationally cheaper than reanalyzing a function at every calling context. One difficulty faced by compositional analyses is precision loss: analyses need to produce a unique summary, correct for every possible input of the analyzed function. A key ingredient to achieve interesting precision levels is to rely on *relational abstract domains* which are able to express relationships between variables. Relational domains have been used extensively in previous works to create input-output relational summaries [26, 14, 17, 25]. However, relational domains are not always sufficient to make up for precision loss, for example when very distinct function behaviors are approximated as a single input-output relation. A second ingredient is the addition of partitioning [30, 7] to create a set of separated summaries for each function, each summary expressing different results on different inputs. Boutonnet and Halbwachs [8] developed a method to compute disjunctive relational summaries that express precise properties of first-order, numeric functions.

The static analysis of functional programs raises new challenges: analyzers need to handle features such as user-defined algebraic data types (ADTs), higher-order functions, partial application, recursive functions and polymorphism. This work presents a compositional analysis which leverages relational abstract domains, exploring a new precision-scalability tradeoff. The analysis supports a pure, monomorphic subset of OCaml, including recursive ADTs, as well as both recursive and higher-order functions.

We present in Figure 1 a motivating example. This program defines a custom algebraic data type, with one constructor containing functions. `to_fun` is a higher-order function, as its result is a function. The definitions of values `f1` and `f2` are performed through partial application. In the case of the program in Figure 1, our approach analyzes function `to_fun` once, at its definition site, and deduces a summary for this function. Under the hood, our compositional analysis relies on two new, key abstract domains: one abstracts user-defined algebraic data types, while the other abstracts higher-order functions. We rely on a cooperation between these two abstract domains to infer a precise, disjunctive [8], summary for `to_fun`. Handling partial applications is seamless, and our analysis is able to infer the following summaries for functions `f1` and `f2`: $\texttt{f1} : x \to 5, \texttt{f2} : x \to x + 4$. The analysis can then perform further applications to conclude that $\texttt{r1} = 5, \texttt{r2} = 9$ fully automatically. Note that both abstract domains can leverage arbitrary abstractions for leaf data types (integers, strings, ...). In particular, the summary of `f2` is expressed precisely thanks to the relational polyhedra domain [15].

In previous work, Lermusiaux and Montagu [27] implemented a value analysis for a large subset of OCaml, which includes recursive ADTs and higher-order constructs. However, this analysis is neither compositional nor relational. This simplifies the analysis of higher-order functions, which can be triggered by analyzing functions only when their parameters are known, at call site. In our example, a non-compositional analysis would analyze `to_fun` twice, when computing the abstract values of `r1` and `r2`. Bautista et al. [5] defined a compositional, relational analysis for a toy imperative language extended with non-recursive ADTs. In particular, their approach does not support recursive functions, nor higher-order ones.

```
type cstorfun = Cst of int | Fun of (int -> int)
let to_fun a =
  match a with
  | Cst n -> fun x -> n
  | Fun f -> f

let f1 = to_fun (Cst 5)
let f2 = to_fun (Fun (fun x -> x + 4))
let r1 = f1 4
let r2 = f2 5
```

Figure 1 Motivating OCaml program with key features from functional programming paradigm.

Contributions. This paper defines the following contributions:

- We introduce a domain summarizing recursive algebraic data types (ADTs). This domain is fully parametric in the abstraction of its base types (integers, strings, ...). Our domain supports relational abstractions.
- Independently, we define an abstract domain for higher-order functions, lifting previous work on disjunctive relational summaries [8]. This allows the creation of a compositional analysis for higher-order functions.
- We illustrate how the two abstract domains can mutually benefit from each other on an example, and provide a soundness statement about the overall analysis.
- We have implemented these domains in the MOPSA platform [22]. Thanks to this implementation, a preliminary evaluation confirms the precision of our approach on a set of 60 toy OCaml programs (handwritten or from previous work [27]).

Outline. Section 2 presents the syntax of a functional language with standard semantics. Section 3 provides background on compositional, relational analyses for the first-order case. Section 4 proposes an abstract domain for recursive algebraic data types, and Section 5 presents our method to analyze higher-order functions. Section 6 highlights the cooperation achieved by combining both abstract domains thanks to their parametricity. Section 7 presents our implementation, benchmarks, and experimental results. Section 8 presents related work and Section 9 concludes.

2 Syntax of the Considered Functional Language

In this section, we present the syntax of a generic functional language, on which we will define our analysis. Its key features are recursion, algebraic data types, and higher-order. We limit ourselves to a monomorphic higher-order language without side-effects. This language is described in Figure 2. $\mathbb{V}$ is the set of variables, $\oplus$ any operator of the language ($+$, $-$, $\times$, $/$, $=$, etc.). $\mathcal{E}$ is the set of expressions. $fv(e)$ is the set of free variables of expression e. Π, the set of patterns, contains the wildcard _, variables, integers, `when` clauses, and type constructors. Free variables from the same pattern are required to be disjoint, as is the case in OCaml: the same variable cannot be bound twice. Figure 3 describes the semantic domain of the language. $\mathbb{C}$ is the set of all possible constructors. Semantic values of the language $\mathcal{V}$ are integers, type constructors containing values, error ω and continuous functions from values to values, to which we add $\perp$ (for non-termination). Finally, Σ is the set of concrete

$$
\begin{array}{rll}
\mathcal{E} ::= & x \in \mathbb{V} & \\
& \mid n \in \mathbb{Z} & \mid e_1 \oplus e_2 \\
& \mid e_0\ e_1\ \ldots\ e_n & \mid \text{fun } x_1 \ldots x_n \to e \\
& \mid \text{let } x = e_1 \text{ in } e_2 & \mid \text{let rec } x = e_1 \text{ in } e_2 \\
& \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 & \mid \text{assert } e \\
& \mid C(e_1, \ldots, e_n) & \mid \text{match } e_0 \text{ with } p_1 \to e_1 \mid \cdots \mid p_m \to e_m \\
\Pi ::= & _ \mid\ x \in \mathbb{V} \mid n \in \mathbb{Z} \mid p_1 \texttt{ when } e_1 \mid C(p_1, \ldots, p_n) & \\
& \text{where } \forall i \neq j, fv(p_i) \cap fv(p_j) = \emptyset &
\end{array}
$$

Figure 2 Syntax of the functional language.

$$\mathcal{V} ::= \mathbb{Z}^\perp \cup \{\ C(v_1, \ldots, v_n) \mid C \in \mathbb{C}, v_i \in \mathcal{V}\ \}^\perp \cup \{\omega\}^\perp \cup \bigcup_{n \in \mathbb{N}} [\mathcal{V}^n \to \mathcal{V}]^\perp$$

$$\Sigma = \mathbb{V} \to \mathcal{V}$$

$$\mathbb{E}[\![\cdot]\!] : \mathcal{E} \times \Sigma \to \mathcal{V}$$

Figure 3 Semantic domain of the functional language.

environments, associating a value to each variable. The concrete semantics of an expression e is $\mathbb{E}[\![e]\!] \in \Sigma \to \mathcal{V}$. It associates a semantic value to an expression in an environment. This semantics follows the standard for eagerly evaluated functional languages (such as OCaml). It is standard, and not detailed here.

We work under the hypothesis that programs are well-typed. The type inference works as usual on functional languages [20, 31]. This way, some situations where an expression evaluates to the error ω are statically detected. However, well-typedness is not enough to prevent all possible kinds of errors, such as assertion failures, or match failures in the presence of `when` clauses. Our analysis focuses on those. During the analysis, we use type information, as well as user-defined type definitions, to select relevant abstract domains.

To simplify presentation, `int` type represents mathematical integers: working on machine integers and statically detecting overflows would only require adding assertions when performing arithmetic operations. Similarly, even though the language is pure, we can still handle array bound checks with a simple translation of programs with arrays to array-less programs: array accesses are replaced with assertions that indices are within the array range, abstracting away the content of the array.

3 Compositional and Relational Analysis at First-Order

Since the concrete semantics is uncomputable, we propose an abstract semantics for our language. This section provides background on relational analyses, and compositional interprocedural analyses for first-order programs with integers and pairs. We start by introducing an abstract semantics compatible with numerical relational analyses through the introduction of symbolic expressions (Section 3.1), and then describe how this abstract semantics can be lifted to perform compositional analysis of recursive functions (Section 3.2).

The analyses we describe feature two notable strengths: their definition is *domain-modular* – i.e. fully parametric in the underlying abstract domains – and supports precise *relational* inference – allowing us to express relationships between variables. We believe that this approach, originally formulated by the authors of the MOPSA framework [22], provides greater expressivity than traditional static analysis definitions. However, this generic and domain-modular approach requires additional formalization efforts to define domains.

3.1 Relational Analysis

We define a computable abstract semantics $\mathbb{E}^\sharp[\![e]\!]$ of an expression e, based on abstract domains. It delegates the abstraction of an object of type τ (e.g. integers, pairs, and after Section 4, ADTs) to the abstract domain $\mathcal{D}_\tau$. We denote the union of those domains as $\mathcal{D}$. Usually, such a semantics takes as parameter an expression e and an abstract environment (abstracting the possible values of all the variables), and returns an abstract value representing the set of possible values of the expression (in the abstract domain corresponding to the type of the expression). However, this formulation cannot easily express a relationship between the value of the expression and those of the variables. Indeed, let us consider the program in Figure 4. A numerical domain such as the interval domain can abstract `x` and `y` independently, as $\texttt{x} \in [1, 10]$ and $\texttt{y} \in [2, 11]$, but cannot infer the relation `y = x + 1`. To overcome this limitation, we use an abstract numerical domain able to store relations between variables – e.g. the polyhedra domain [15]. However, expressing relations between numeric variables from the program, that is, `x` and `y`, is not sufficient to leverage those relational information to `pair`, since we have no way to express a relation between the second field of `pair` and `y`.

Following the works of Chevalier and Feret [10], Journault et al. [22], we introduce *intermediate variables* (also named ghost variables), in addition to the initial program variables. Here, we introduce two intermediate variables p_1 and p_2, which respectively represent the first and second fields of `pair`. In practice, we keep separate the set of program variables $\mathbb{V}$ and the set $\mathbb{V}_\mathbb{Z}$ of intermediate variables on which a (possibly relational) numerical domain will express semantic constraints. We then keep a map from program variables in $\mathbb{V}$ to their *abstract values* in $\mathcal{V}^\sharp$, which can either be an element of an abstract domain, or an intermediate variable. In our example, $\mathbb{V}_\mathbb{Z}$ is $\{x, y, p_1, p_2\}$ whereas $\mathbb{V} = \{\texttt{x}, \texttt{y}, \texttt{pair}\}$. The map is then $m = \texttt{x} \to x, \texttt{y} \to y, \texttt{pair} \to (p_1, p_2)$. The possible values of variables in $\mathbb{V}_\mathbb{Z}$ are then represented as an abstract element from a (possibly relational) abstract domain $\mathcal{D}_\mathbb{Z}$. If the chosen numerical domain $\mathcal{D}_\mathbb{Z}$ is the polyhedra domain – which can express conjunctions of linear constraints – we can infer: $d = 1 \leq x \leq 10 \wedge y = x + 1 \wedge p_1 = y \wedge 1 \leq p_2 \leq y \in \mathcal{D}_\mathbb{Z}$. Note that the relation between the second field of `pair` and `y` was not expressible without introducing p_2. Such intermediate variables will be especially useful when abstracting algebraic data types in Section 4.

An abstract environment in $\Sigma^\sharp$ is then the pair of a map $m \in \mathbb{V} \to \mathcal{V}^\sharp$ and an element of the possibly relational numerical domain $d \in \mathcal{D}_\mathbb{Z}$. Thus, $\sigma^\sharp = (m, d) \in \Sigma^\sharp$.

As a last step, we define *abstract expressions* $\mathcal{E}^\sharp$, a lifting of abstract values where numerical intermediate variables can instead be *numeric expressions* $\mathcal{E}_\mathbb{Z}$, that is, symbolic operations between intermediate variables and integers. The abstract semantics of an expression e is then: $\mathbb{E}^\sharp[\![e]\!] : \Sigma^\sharp \to \mathcal{E}^\sharp \times \Sigma^\sharp$. It evaluates an expression e in an environment and returns an abstract expression into a new environment. Let p be the program in Figure 4, and $\sigma_0^\sharp$ a starting environment with no information on program variables. The analysis of p is:

$$\mathbb{E}^\sharp[\![p]\!]\sigma_0^\sharp = (p_1, p_2), (\texttt{x} \to x, \texttt{y} \to y, \texttt{pair} \to (p_1, p_2), \\ 1 \leq x \leq 10 \wedge y = x + 1 \wedge p_1 = y \wedge 1 \leq p_2 \leq y)$$

```
1 let x = random 1 10 in
2 let y = x + 1 in
3 let pair = (y, random 1 y) in
4  pair
```

Figure 4 Example program motivating relational analysis.

Set of program variables	$\mathbb{V}$
Set of intermediate variables	$\mathbb{V}_{\mathbb{Z}}$
Union of non-numerical domains	$\mathcal{D}$
Numerical domain	$\mathcal{D}_{\mathbb{Z}}$
Abstract values	$\mathcal{V}^\sharp = \mathcal{D} \cup \mathbb{V}_{\mathbb{Z}}$
Abstract environments	$\Sigma^\sharp = (\mathbb{V} \to \mathcal{V}^\sharp) \times \mathcal{D}_{\mathbb{Z}}$
Concrete numerical environments	$\Sigma_{\mathbb{Z}} = \mathbb{V}_{\mathbb{Z}} \to \mathbb{Z}$
Numeric expressions	$\mathcal{E}_{\mathbb{Z}} ::= n \in \mathbb{Z} \mid v \in \mathbb{V}_{\mathbb{Z}} \mid e_1 \oplus e_2$, where $e_1, e_2 \in \mathcal{E}_{\mathbb{Z}}$
Abstract expressions	$\mathcal{E}^\sharp = \mathcal{D} \cup \mathcal{E}_{\mathbb{Z}}$
Abstract semantics of expr. e	$\mathbb{E}^\sharp[\![e]\!] : \Sigma^\sharp \to \mathcal{E}^\sharp \times \Sigma^\sharp$

Figure 5 Formalization of the abstract semantics.

The abstract expression (p_1, p_2) is a symbolic pair of intermediate variables $p_1, p_2 \in \mathbb{V}_{\mathbb{Z}}$, therefore $(p_1, p_2) \in \mathcal{E}_{\mathbb{Z}}$. Figure 5 summarizes all abstract sets.

Assignments. Our abstract environments are made of two components $(m, d) \in \Sigma^\sharp$: a map m from variables to abstract values and an element d of the relational domain. To define assignments in such an environment, we write:

- $m[\mathtt{x} \to v]$ the map $m \in (\mathbb{V} \to \mathcal{V}^\sharp)$ where variable $\mathtt{x} \in \mathbb{V}$ is now bound to the value $v \in \mathcal{V}^\sharp$;
- $\mathbb{E}^\sharp_{\mathbb{Z}}[\![x \to v]\!]d \in \mathcal{D}_{\mathbb{Z}}$ the result of the assignment in an element $d \in \mathcal{D}_{\mathbb{Z}}$ of variable $x \in \mathbb{V}_{\mathbb{Z}}$ to a value or a numeric expression v.

▶ **Definition 3.1** (Abstract assignment). *The abstract assignment of a program variable* $\mathtt{x} \in \mathbb{V}$ *with abstract value* $v \in \mathcal{V}^\sharp$ *in environment* $(m, d) \in \Sigma^\sharp$ *is then:*

$$(m, d)[\mathtt{x} \to v] = \begin{cases} \left(m[\mathtt{x} \to x], \mathbb{E}^\sharp_{\mathbb{Z}}[\![x \to v]\!]d\right) & \text{if } x : \texttt{int} \\ (m[\mathtt{x} \to v], d) & \text{otherwise} \end{cases}$$

Intuitively, the assignment is delegated to the numerical domain for integer variables, and it is a simple map update otherwise.

Join. We denote the join between two environments as $\sqcup^\sharp$, which lifts the join operators from underlying components. The join is defined pointwise between two maps in $\mathbb{V} \to \mathcal{V}^\sharp$: each variable maps to a join of its abstract values from both maps in the relevant domain. We rely on the join of the numerical abstraction for the second component of the abstract environment. Note that this join can be *heterogeneous* [24], if both elements are not defined on the same set of variables (this will be further developed in Section 4). The meet $\sqcap^\sharp$ is defined similarly.

Concretization. The concretization of an abstract environment is based on the concretization $\gamma_{\mathbb{Z}}$ of the numerical domain, and on the concretizations γ_τ of the domains $\mathcal{D}_\tau$ abstracting objects of type τ. Those domains can be relational, so the concretization of a value can depend on other variables' concretization, following a construction defined by Monat [32]. Consequently, γ_τ takes as parameter an element of the domain $\mathcal{D}_\tau$ *together with a concrete numerical environment* $\sigma_{\mathbb{Z}} \in \Sigma_{\mathbb{Z}}$.

▶ **Definition 3.2** (Environment concretization). *Given* $\sigma^\sharp = (m, d) \in \Sigma^\sharp$*, its concretization is:*

$$\gamma_{\Sigma^\sharp}(m, d) = \{\sigma \in \Sigma \mid \exists \sigma_{\mathbb{Z}} \in \gamma_{\mathbb{Z}}(d), \forall x \in \mathbb{V}, x : \tau, \sigma(x) \in \gamma_\tau(m(x), \sigma_{\mathbb{Z}})\}$$

This concretization thus depends on the concretizations for ground types (such as integers, strings, etc.), but also on the concretizations for functions and algebraic data types. Those will be defined in Section 4 and Section 5.

▶ Remark 3.3 (Implicit concretization signature). Even though the concretization γ_τ takes as parameter a *concrete environment*, we can lift this definition so that it takes as parameter an *abstract environment* by writing:

$$\gamma_\tau(e^\sharp, (m, d)) = \bigcup_{\sigma_{\mathbb{Z}} \in \gamma_{\mathbb{Z}}(d)} \gamma_\tau(e^\sharp, \sigma_{\mathbb{Z}})$$

We simply join concretizations for every concrete numerical environment abstracted by d. Our abstract semantics takes as input an abstract environment and returns an abstract value in an abstract environment. With this notation, given $e : \tau$ and $\sigma^\sharp \in \Sigma^\sharp$, we can write the concretization of the abstract semantics $\mathbb{E}^\sharp[\![e]\!]\sigma^\sharp \in \mathcal{D}_\tau \times \Sigma^\sharp$, and we can write $\gamma_\tau(\mathbb{E}^\sharp[\![e]\!]\sigma^\sharp)$. In particular, this will be useful when stating the soundness theorem (Section 6.2).

Error propagation. To simplify the presentation, errors are implicitly propagated: if a sub-expression evaluates to the error ω, then so does the whole expression.

3.2 Compositional Function Analysis

Since an important contribution of this article is to design a *compositional* analysis, we first introduce how we analyze first-order functions in a compositional fashion, before extending our analysis in Section 5 to higher-order.

Compositional analysis aims at over-approximating all the possible behaviors of a function at definition site. The function is analyzed once and for all, and we then store the result of the analysis, its *summary*. Since the language is pure, the function does not modify the environment (no side-effects). Its behavior can then be represented by a relation between its inputs and its output, using a relational domain.

Let $\mathcal{F}^\tau$ be the set of functions of type $\tau = \tau_1 \to \cdots \to \tau_{n+1}$. Since we are restricted to first-order in this section, τ_i are base types such as integers or pairs. Given a set V of variables for inputs and output, we denote $\mathcal{R}(V)$ as the abstract domain chosen to represent relations between those. It abstracts $\Sigma|_V$, i.e. concrete environments restricted to V. It is provided with its concretization $\gamma_r : \mathcal{R}(V) \to \Sigma|_V$, with a lifting $\lambda_r : \mathcal{R}(V) \to \Sigma^\sharp$ and with a projection operator proj_r such that $\text{proj}_r(\sigma^\sharp) \in \mathcal{R}(V)$ only keeps information on variables in V.

We denote the domain chosen for functions of type τ as $\mathcal{D}_\tau = \mathbb{V}^n \times \mathbb{V} \times \mathcal{R}(V)$. This way, a function is abstracted as: the names $x_1, \ldots, x_n$ of its formal inputs, a result variable, and a relation between those variables.

▶ **Definition 3.4** (Functions concretization). *The concretization* $\gamma_\tau : \mathcal{D}_\tau \times \Sigma_\mathbb{Z} \to \mathcal{F}^\tau$ *is:*

$$\gamma_\tau((x_1,\dots,x_n,r,p),_) = \{f : \tau \mid \forall(a_1,\dots,a_n) : \tau_1 \times \cdots \times \tau_n, \\ [x_1 = a_1] \cdots [x_n = a_n][r = f(a_1,\dots,a_n)] \in \gamma_r(p)\}$$

A function f is abstracted as $(x_1,\dots,x_n,r,p)$ if the relation between its inputs and its output can be described by p. $_ \in \Sigma_\mathbb{Z}$ is not used by the concretization, as the summary relies on its own relational abstract element p.

We now describe the transfer functions used by the compositional analysis.

Function definition. Intuitively, the analysis of functions needs to generate a summary valid for all inputs. Thus, the body is analyzed with the most general inputs: the inputs are initialized at $\top$, the domain maximum (meaning we have no information on their values). We project the result on input and output variables. This way, we get the most general input-output relation, usable at any call site. Note that relationality is critical there, enabling us to discover relevant information on the function behavior despite making no hypothesis on the values of the inputs.

$$\mathbb{E}^\sharp[\![\text{fun } x_1 \cdots x_n \to \mathit{body}]\!]\sigma^\sharp = \text{let } e^\sharp, \sigma_0^\sharp = \mathbb{E}^\sharp[\![\mathit{body}]\!]\sigma^\sharp[x_1 \to \top,\dots,x_n \to \top] \text{ in} \\ ((x_1,\dots,x_n), r, \text{proj}_r(\sigma_0^\sharp[r \to e^\sharp])), \sigma^\sharp$$

Recursive functions. For recursive functions, the concrete semantics has to compute a least fixpoint of the function concrete semantics F:

$$\mathbb{E}[\![\text{let rec } f = e_1 \text{ in } e_2]\!]\sigma = \text{ let } F(v) = \mathbb{E}[\![e_1]\!]\sigma[f \to v] \text{ in } \mathbb{E}[\![e_2]\!]\sigma[f \to \text{lfp}(F)]$$

We use a standard technique from abstract interpretation [12] to obtain a computable abstract semantics, in three steps. First, thanks to Kleene's theorem, we rewrite the least fixpoint as the union of iterations of the analysis of the function, starting from $f \to \bot$, i.e. $\text{lfp}(F) = \bigcup_{k\in\mathbb{N}} F^k(\bot)$. Second, we move to the abstract semantics by using an iteration from $f \to \bot_\tau$, but evaluate the function body in the abstract domain with F. Third, we use a widening operator ∇_τ instead of a join to enforce the convergence in finite time. The widening operator is a component-wise lift, similar to the definition of the join operator on abstract environments in Section 3.1.

$$\mathbb{E}^\sharp[\![\text{let rec } f = e_1 \text{ in } e_2]\!]\sigma^\sharp = \text{let } \bot_\tau = ((x_1,\dots,x_n), r, [r \to \bot][x_i \to \top]_{i\le n}) \text{ in} \\ \text{let } F : \mathcal{D}_\tau \to \mathcal{D}_\tau = v^\sharp \to \text{fst}(\mathbb{E}^\sharp[\![e_1]\!]\sigma^\sharp[f \to v^\sharp]) \text{ in} \\ \text{let } s = \nabla_\tau \bigcup_{k\in\mathbb{N}} F^k(\bot_\tau) \text{ in} \\ \mathbb{E}^\sharp[\![e_2]\!]\sigma^\sharp[f \to s]$$

This reasoning extends to mutually recursive functions by iterating on a vector of function abstractions.

Function application. When applying a function e_0 to inputs of abstract values $e_i^\sharp$, we intuitively combine the current environment and the summary relation p. Before the combination, both abstract environments need to be defined on the same set of variables. We rely on two auxiliary functions: `add_vars` : $\Sigma^\sharp \times \mathcal{P}(\mathbb{V}_\mathbb{Z}) \to \Sigma^\sharp$ which extends the definition domain of an abstract environment with a set of variables, and `dom` : $\Sigma^\sharp \to \mathcal{P}(\mathbb{V}_\mathbb{Z})$ which

provides the definition domain of an abstract environment. The combination also requires additional equality constraints $[x_i = e_i^\sharp]$ between formal arguments x_i and values $e_i^\sharp$. These equality constraints are delegated to the filter function[1] of the relevant abstract domain.

$$\begin{aligned}\mathbb{E}^\sharp[\![e_0 \ \dots \ e_n]\!]\sigma_0^\sharp = \ & \text{let } ((x_1,\dots,x_n),r,p),\sigma_0^\sharp = \mathbb{E}^\sharp[\![e_0]\!]\sigma^\sharp \text{ in}\\ & \text{let } \forall i \in [\![0,n-1]\!] \ e_{i+1}^\sharp,\sigma_{i+1}^\sharp = \mathbb{E}^\sharp[\![e_{i+1}]\!]\sigma_i^\sharp \text{ in}\\ & \text{let } \sigma_n^\sharp = \texttt{add_vars}(\sigma_n^\sharp,\texttt{dom}(p)) \text{ in}\\ & \text{let } p = \texttt{add_vars}(\lambda_r(p),\texttt{dom}(\sigma_n^\sharp)) \text{ in}\\ & r,(\sigma_n^\sharp \sqcap^\sharp p)[x_1 = e_1^\sharp,\dots,x_n = e_n^\sharp]\end{aligned}$$

Compositionality. A key observation on our abstract semantics is that the function is analyzed at *definition site*. Consequently, we analyze it only once. At any call site, we only instantiate the resulting summary with input values. At first-order, Boutonnet and Halbwachs [8] proved that such a *compositional analysis* leads to a more efficient analysis for a function used at many call sites. Therefore, it can help the analysis to scale up.

Note that the purity hypothesis is necessary here: abstracting functions as input-output relations is correct only because the function does not modify the memory state. To handle side-effects, we would need to modify this abstraction to track mutability of the memory state, e.g., using extra input-output arguments to functions to model the mutable part of the state. Keeping compositionality while overcoming this limitation is left for future work.

In conclusion, this section defined a compositional analysis for first-order functions, both relational and fully parametric in the choice of abstract domains used to represent ground types. These features will be at the core of our abstractions for ADTs and higher-order functions.

4 Abstracting Recursive Algebraic Data Types

Algebraic objects are basic blocks of functional languages. Lists, trees, abstract syntax trees, etc. are defined using algebraic data types. To analyze them, we need a specific domain for objects of user-defined type:

$$\texttt{type } \tau = C_1 \text{ of } \tau_{1,1} * \cdots * \tau_{1,m_1} \mid \cdots \mid C_n \text{ of } \tau_{n,1} * \cdots * \tau_{n,m_n}$$

This section defines an abstraction for user-defined algebraic data types which is fully parametric in the abstraction of base types (integers, strings, etc). We illustrate our construction on integer base types, as these are very common, and many numerical abstractions already exist to serve as parameter. It also allows us to show that our analysis can be relational when parameterized with relational base domains. This relationality is a crucial factor to make compositional analyses precise, as exemplified in Section 4.3.

4.1 Parametric Relational Domain with Symbolic Variables

4.1.1 Domain

Let n be the number of constructors and m_i the number of fields of constructor C_i. Let $\mathbb{C}_\tau = \{C_i \mid 1 \leq \mathtt{i} \leq n\}$ be the set of constructors of type τ. Let $\mathbb{T}_\tau$ be the set of finite concrete objects of type τ. For recursive types, $\mathbb{T}_\tau$ can be infinite. To provide a computable

[1] The filter function models the effect of a test or a guard in the domain, by selecting environments satisfying a conditional – here, $[x_i = e_i^\sharp]$

abstract semantics, we chose to summarize (or "smash") together all the elements of a given field that are nested in an abstract data type. For instance, when abstracting a list of integers, a single abstract integer variable will be used to represent all the possible values of all the elements of the list. More generally, infinite sets of unbounded data-structures can be abstracted using a finite number of dimensions – here, one per field.

We construct systematically an abstraction for values of type τ from existing abstractions for values of types $\tau_{i,j}$ used in the definition of τ. However, in order to avoid cyclic definitions in the case of recursive types, we start from given domains $\mathcal{D}^\sharp_{\tau_{i,j}}$ defined only when $\tau_{i,j} \neq \tau$. The case of recursive types is well-founded, as is the case of user-defined types using fields of previously defined user-defined types. For simplicity, this definition omits the case of mutually recursive types, which does not pose additional theoretical issues.

▶ **Definition 4.1** (ADTs domain). *We derive domains $\mathcal{D}^\sharp_{i,j}$ used to abstract the value of the j-th field of the i-th constructor as follows:*

$$\mathcal{D}^\sharp_{i,j} = \begin{cases} \mathcal{P}(\mathbb{C}_\tau) & \text{if } \tau_{i,j} = \tau \\ \mathbb{V}_\mathbb{Z} & \text{if } \tau_{i,j} = \texttt{int} \\ \mathcal{D}^\sharp_{\tau_{i,j}} & \text{otherwise} \end{cases}$$

Finally, values of type τ are abstracted in $\mathcal{D}^\sharp_\tau$, defined as follows:

$$\mathcal{D}^\sharp_\tau = \prod_{\substack{1 \leq i \leq n \\ 1 \leq j \leq m_i}} (\mathcal{D}^\sharp_{i,j})^\perp \times \mathcal{P}(\mathbb{C}_\tau)$$

Intuitively, $\mathcal{D}^\sharp_\tau$ abstracts an object x of type τ as $((g_{i,j})_{i,j}, \mathscr{C})$ such that:

- $g_{i,j} \in (\mathcal{D}^\sharp_{i,j})^\perp$ abstracts every j-th field of C_i nested in x;
- $\mathscr{C} \in \mathcal{P}(\mathbb{C}_\tau)$ is the set of possible constructors C_i for x.

$\mathcal{D}^\sharp_\tau$ delegates the abstraction of field (i,j) to $\mathcal{D}^\sharp_{i,j}$. If this field is recursive, $\mathcal{D}^\sharp_{i,j} = \mathcal{P}(\mathbb{C}_\tau)$, meaning that we only keep track of its possible start constructors. If this is an integer field, $\mathcal{D}^\sharp_{i,j} = \mathbb{V}_\mathbb{Z}$, meaning that we create a symbolic variable $x.i.j \in \mathbb{V}_\mathbb{Z}$, to represent every field (i,j) nested in x. The numerical domain can then infer relations between them. When the field is neither recursive nor an integer, we delegate the abstraction to the domain chosen to abstract objects of type $\tau_{i,j}$. In this case, $\mathcal{D}^\sharp_{i,j} = \mathcal{D}^\sharp_{\tau_{i,j}}$.

By smashing every constructor field in one summary variable, we use an abstraction of fixed size to represent unbounded objects of recursive type. This approach shares similarities with the approach of Gopan et al. [18]. They use a fixed number of dimensions to over-approximate numeric values in unbounded collections, to analyze e.g. dynamic arrays or heap-allocated structures. Summarization variables consequently abstract more than one object. Note that smashing is done at the variable allocation site: each variable of type τ defines a set of numerical variables – the abstraction is called *object-sensitive* [39].

▶ **Example 4.2.** Let us illustrate this abstraction on lists of integers, defined as:

```
type list = Cons of int * list | Nil
```

We use the interval domain (i.e., $\mathcal{D}_\mathbb{Z} = \mathbb{I}$) to abstract integers. The set of possible constructors is $\mathbb{C}_{\texttt{list}} = \{\texttt{Cons}, \texttt{Nil}\}$. The constructor `Cons` has two fields. The first one, containing integers, is represented by $\mathcal{D}^\sharp_{1,1} = \mathbb{V}_\mathbb{Z}$, that is, a numeric variable, abstracted in $\sigma^\sharp$ as an interval. The second, recursive, one is represented by $\mathcal{D}^\sharp_{1,2} = \mathcal{P}(\mathbb{C}_{\texttt{list}})$. The constructor `Nil` has no field to be represented. The global abstraction for lists of integers is then:

$$\mathcal{D}^\sharp_{\texttt{list}} = \left((\mathcal{D}^\sharp_{1,1})^\perp \times (\mathcal{D}^\sharp_{1,2})^\perp\right) \times \mathcal{P}(\mathbb{C}_{\texttt{list}}) = \left(\mathbb{V}^\perp_\mathbb{Z} \times \mathcal{P}(\mathbb{C}_{\texttt{list}})^\perp\right) \times \mathcal{P}(\mathbb{C}_{\texttt{list}})$$

4.1.2 Concretization

Let $\gamma_{\tau_{i,j}}$ be the concretization for $\mathcal{D}^\sharp_{\tau_{i,j}}$ when $\tau_{i,j} \neq \tau$. It is provided by the domain $\mathcal{D}^\sharp_{\tau_{i,j}}$.

▶ **Definition 4.3** (ADTs concretization). *We define* $\gamma_\tau : \mathcal{D}^\sharp_\tau \times \Sigma_\mathbb{Z} \to \mathcal{P}(\mathbb{T}_\tau)$. *For* $(g, \mathscr{C}) \in \mathcal{D}^\sharp_\tau$, $\sigma_\mathbb{Z} \in \Sigma_\mathbb{Z}$*:*

$$\gamma_\tau((g,\mathscr{C}),\sigma_\mathbb{Z}) = \left\{ x : \tau \;\middle|\; \exists i,\, x = C_i(x_{i,1},\ldots,x_{i,m_i}) \;\wedge\; C_i \in \mathscr{C} \;\wedge\; \forall j, x_{i,j} \in \begin{cases} \gamma_\tau((g,g_{i,j}),\sigma_\mathbb{Z}) & \text{if } \tau_{i,j} = \tau \\ \gamma_{\tau_{i,j}}(g_{i,j},\sigma_\mathbb{Z}) & \text{otherwise} \end{cases} \right\}$$

Since type τ is finite, recursion is well-founded ($x_{i,j}$ contains strictly less constructors than x). An object $x : \tau$ is abstracted as $(g, \mathscr{C})$ if:

- it starts with $C_i \in \mathscr{C}$,
- and every j-th field of C_i accessible in x
  - either starts with constructor in $g_{i,j} \subseteq \mathbb{C}_\tau$, in the recursive case ($\tau_{i,j} = \tau$),
  - or can be abstracted as $g_{i,j}$ otherwise.

$g_{i,j}$ represents every j-th field from C_i nested in x, while $\Sigma_\mathbb{Z}$ keeps track of numeric variables.

▶ **Example 4.4.** Following Example 4.2, we consider the integer list abstract value $(g, \mathscr{C}) = ((r, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\})$ with numerical environment $\sigma_\mathbb{Z} = [r \to [1, 10]]$. Its concretization $\gamma_{\texttt{list}}((g, \mathscr{C}), \sigma_\mathbb{Z})$ is the set of non-empty lists containing integers between 1 and 10:

$$\{\, x : \texttt{int list} \mid x = \texttt{Cons}(h, q) \wedge h \in [1, 10] \wedge q \in \gamma_{\texttt{list}}(((r, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\}), \sigma_\mathbb{Z})\}$$

Let us consider $l = \texttt{Cons}(1, \texttt{Cons}(4, \texttt{Cons}(10, \texttt{Nil})))$. We have $l \in \gamma_{\texttt{list}}((g, \mathscr{C}), \sigma_\mathbb{Z})$, meaning that $(g, \mathscr{C})$ is a correct over-approximation of l.

4.1.3 Lattice Operators

We now define lattice operators. Since our domain is relational, operators are defined on environments. As seen in Section 3, environments are pairs, composed of a map from program variables to objects from their abstract domains, together with an element from the numerical relational domain. We denote lattice operators for the numerical relational domain as $(\sqsubseteq_r, \cup_r, \cap_r, \nabla_r)$. We denote operators for $\mathcal{D}^\sharp_{i,j}$ as $(\sqsubseteq_{\mathcal{D}_{i,j}}, \cup_{\mathcal{D}_{i,j}}, \cap_{\mathcal{D}_{i,j}}, \nabla_{\mathcal{D}_{i,j}})$. We define an intermediate inclusion between two ADT abstractions, interpreted outside their abstract environment. We simply lift inclusions $\sqsubseteq_{\mathcal{D}_{i,j}}$.

▶ **Definition 4.5** (Environment-free inclusion). *For* $(g_1, \mathscr{C}_1)$, $(g_2, \mathscr{C}_2) \in \mathcal{D}^\sharp_\tau$ *abstract elements:*

$$(g_1,\mathscr{C}_1) \subseteq^\sharp_\tau (g_2,\mathscr{C}_2) \iff \mathscr{C}_1 \subseteq \mathscr{C}_2 \wedge \forall i,j, g^1_{i,j} \sqsubseteq_{\mathcal{D}_{i,j}} g^2_{i,j}$$

$(g_1, \mathscr{C}_1) \subseteq^\sharp_\tau (g_2, \mathscr{C}_2)$ then means that the non-numerical fields of $(g_1, \mathscr{C}_1)$ are more precise than those of $(g_2, \mathscr{C}_2)$. To compare numerical fields, we need to check inclusion on environments:

▶ **Definition 4.6** (Inclusion). *We define* $\sqsubseteq^\sharp : \Sigma^\sharp \to \Sigma^\sharp$*:*

$$(m_1,d_1) \sqsubseteq^\sharp (m_2,d_2) \iff \forall v : \tau, m_1(v) \subseteq^\sharp_\tau m_2(v) \wedge d_1 \sqsubseteq_r d_2$$

An environment (m_1, d_1) is more precise than (m_2, d_2) if for every program variable, the abstraction of this variable is more precise in m_1 than in m_2, and its abstraction for numerical variables, d_1, is more precise than d_2.

▶ **Example 4.7.** Following Example 4.2 on integer lists, we have for instance:

$$((x.1.1, \{\texttt{Nil}\}), \{\texttt{Cons}\}) \subseteq^\sharp_{\texttt{list}} ((x.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\})$$

So we can deduce:

$$\begin{aligned}&(x \to ((x.1.1, \{\texttt{Nil}\}), \{\texttt{Cons}\}), \{3 \leq x.1.1 \leq 6\})\\ \sqsubseteq^\sharp\ &(x \to ((x.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\}), \{2 \leq x.1.1 \leq 7\})\end{aligned}$$

To define the meet, join and widening operators, we also start by defining those operators on the environment-free ADT abstractions, that is, ignoring their numerical part. Similarly, we lift operators from domains $\mathcal{D}^\sharp_{i,j}$. Those operators are defined between objects giving the same name to their summarization variables.

▶ **Definition 4.8** (Environment-free operators). *For* $\square \in \{\cup, \cap, \nabla\}$, *for* $(g_1, \mathscr{C}_1), (g_2, \mathscr{C}_2) \in \mathcal{D}^\sharp_\tau$*:*

$$(g_1, \mathscr{C}_1)\ \square^\sharp_\tau\ (g_2, \mathscr{C}_2) = (g, \mathscr{C}_1\ \square\ \mathscr{C}_2) \text{ where } g \text{ is defined by } \forall i, j, g_{i,j} = g^1_{i,j}\ \square_{\mathcal{D}_{i,j}}\ g^2_{i,j}$$

$(g_1, \mathscr{C}_1)\ \square^\sharp_\tau\ (g_2, \mathscr{C}_2)$ performs a join (resp. meet or widening) between non-numerical fields by delegating the operation to the relevant domains. Afterwards, to perform the operation on numerical fields too, we need to perform it on environments:

▶ **Definition 4.9** (Operators). *For* $\square \in \{\cup, \cap, \nabla\}$,*we build* $\square^\sharp : \Sigma^\sharp \to \Sigma^\sharp$*:*

$$(m_1, d_1)\ \square^\sharp\ (m_2, d_2) = (v : \tau \to m_1(v)\ \square^\sharp_\tau\ m_2(v),\ d_1\ \square^\sharp_r\ d_2)$$

Performing an operation between two environments consists in performing it on each variable abstraction in both maps, pointwise, and between both numerical abstract elements.

Note that the inclusion test, meet, join, and widening for the numerical relational domain operate on *heterogeneous* environments, that is, environments that may be defined on different sets of variables. We rely on the technique from Journault et al. [24] to lift classic relational domains (such as polyhedra) to the case of heterogeneous environments.. The soundness proof of our operators is a direct consequence of their soundness proofs [21].

4.1.4 Constructor Transfer Function

We then define the transfer function of constructors. Given an abstract environment $\sigma^\sharp_0$ and a concrete expression $C_k(e_1, \ldots, e_n)$, we compute a sound abstraction $(g, \mathscr{C})$ in a new abstract environment:

$$\begin{aligned}
\mathbb{E}^\sharp[\![\ C_k(e_1, \ldots, e_n)\ ]\!]\sigma^\sharp_0 = {}& \text{let } \forall i \in [\![1, n]\!]\ v_i, \sigma^\sharp_i = \mathbb{E}^\sharp[\![e_i]\!]\sigma^\sharp_{i-1} \text{ in} && (1)\\
& \text{let } (g^0, _), \sigma^\sharp = \mathbf{fold}\left((g^0, \emptyset) \leftarrow (v_j)_{\tau_{k,j}=\tau}\right)\sigma^\sharp_n \text{ in} && (2)\\
& \text{let } g^0, \sigma^\sharp = \bigcirc_{j, \tau_{k,j} \neq \tau}\left(\mathbf{fold}(g^0 \leftarrow_{k,j} v_j)\right)\ \sigma^\sharp \text{ in} && (3)\\
& \text{let } g = \left(\begin{cases} g^0_{k,j} \cup \mathrm{snd}(v_j) & \text{if } i = k \wedge \tau_{k,j} = \tau \\ g^0_{i,j} & \text{otherwise} \end{cases}\right)_{i,j} \text{ in} && (4)\\
& (g, \{C_k\}), \sigma^\sharp
\end{aligned}$$

The abstract semantics of $C_k(e_1, \ldots, e_n)$ is an element of $\mathcal{D}^\sharp_\tau$. The only possible start constructor is C_k. We abstract its fields as g, which is computed as follows. First, we recursively compute abstractions for e_i in v_i (1). We then define $g^0 = (g^0_{i,j})_{i,j}$. We use the

fold operator, defined by Gopan et al. [18] for summarization variables of arrays and lifted there for ADTs. It folds inside every $g^0_{i,j}$ all abstractions for the j-th fields of C_i nested inside every recursive field abstraction v_j (2). After the first folding, $g^0_{i,j}$ abstracts possible values in the j-th field of C_i nested at a depth of at least 1 in x. Then, we need to update it with every possible values in j-th fields of C_i nested at *any depth* in x. First, we add non-recursive information at depth 0, that is, updating $g^0_{k,j}$ with j-th field of C_k at depth 0, v_j (3). To do so, we compose ($\bigcirc$), for every non-recursive field (k, j), the **fold** operator to add v_j as a possible abstract values for field $g^0_{k,j}$. Finally, the possible start constructors of e_j, that is, $\mathrm{snd}(v_j)$, are added as possible start constructors for the j-th field of C_k (4).

▶ **Example 4.10.** Following previous examples on integer lists, given $\sigma^\sharp \in \Sigma^\sharp$:

$$\mathbb{E}^\sharp[\![\texttt{Nil}]\!]\sigma^\sharp = ((r.1.1, \bot), \{\texttt{Nil}\}), \sigma^\sharp[r.1.1 \to \bot]$$
$$\mathbb{E}^\sharp[\![\texttt{Cons}(10, \texttt{Nil})]\!]\sigma^\sharp = ((r.1.1, \{\texttt{Nil}\}), \{\texttt{Cons}\}), \sigma^\sharp[r.1.1 \to \bot \cup_{\mathbb{I}} [10, 10]]$$
$$\mathbb{E}^\sharp[\![\texttt{Cons}(1, \texttt{Cons}(10, \texttt{Nil}))]\!]\sigma^\sharp = ((r.1.1, \{\texttt{Nil}, \texttt{Cons}\}), \{\texttt{Cons}\}), \sigma^\sharp[r.1.1 \to [1, 10]]$$

Note that we rely on type information to delegate abstraction of a given field to the relevant domain. We discuss related implementation details in Section 7.1.

4.1.5 Abstraction Precision

▶ **Example 4.11** (Abstraction of trees)**.** We start with $\sigma^\sharp_0$ an empty environment. As an illustration for this abstract domain, consider:

```
type tree = Node of tree * int * tree | Leaf of int
let x = Node(Node(Leaf(250), 100, Leaf(251)), 1, Leaf(252))
```

Here, $\mathbb{E}^\sharp[\![\texttt{Node}(\texttt{Node}, (\texttt{Leaf}(250), 100, \texttt{Leaf}(251)), 1, \texttt{Leaf}(252))]\!]\sigma^\sharp_0 = ((g, \mathscr{C}), \sigma^\sharp)$ with:

$$(g, \mathscr{C}) = ((\{\texttt{Node}, \texttt{Leaf}\}, x.1.2, \{\texttt{Leaf}\}, x.2.1), \{\texttt{Node}\})$$
$$\sigma^\sharp = ([x.1.2 \to v_{x.1.2}], [x.2.1 \to v_{x.2.1}], [v_{x.1.2} \to [1, 100]][v_{x.2.1} \to [250, 252]])$$

$$\gamma_{\texttt{tree}}((g, \mathscr{C}), \sigma^\sharp) = \{x \mid x = \texttt{Node}(t_1, n, t_2) \ \wedge \ n \in [1, 100] \\ \wedge\, t_1 \in \gamma_{\texttt{tree}}((g, \{\texttt{Node}, \texttt{Leaf}\}))t_2 \in \gamma_{\texttt{tree}}(g, \{\texttt{Leaf}\})\}$$

It abstracts the set of trees starting with `Node`, only growing to the left, where the content of `Leaf` fields are between 250 and 252, and the content of `Node` between 1 and 100.

Note that our summarization abstraction can quickly lose precision, even when the structure size is known statically. To limit this, we could keep track of `x`'s exact content and summarize only when needed for convergence (recursive call), or for a certain depth of `x`. Such features are not yet implemented. Note that though of identical type, `Node` and `Leaf`'s integer fields are summed up separately.

▶ **Example 4.12** (Relationality)**.** Consider this program manipulating integer variables `a` and `y` of unknown value:

```
let z = Cons(a, Nil) in
let x = if y <= 2*a + 1 then Cons(y, z) else Cons(2 * a + 1, z)
```

With the polyhedra domain of Cousot and Halbwachs [15], the analysis is able to infer relational properties:

$$
\begin{aligned}
z \to & ((z.1.1, \{\texttt{Nil}\}), \{\texttt{Cons}\}), [z.1.1 = a] \\
x \to & (((x.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\}), [x.1.1 = z.1.1 \cup y][y \leq 2a+1][z.1.1 = a]) \cup^\sharp_{\texttt{list}} \\
& (((x.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\}), [x.1.1 = z.1.1 \cup 2a+1][y > 2a+1][z.1.1 = a]) \\
\to & ((x.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\}), [x.1.1 < 2a+1][z.1.1 = a]
\end{aligned}
$$

We can then prove: $x_{1,1} \leq 2a+1$, that is, every element of the list is smaller than $2a+1$, without assuming any bound on a. This relational aspect is crucial to infer input-output function summaries for generic inputs.

4.2 Pattern-Matching Abstract Semantics

Pattern-matching is a key construct to manipulate ADTs. We define their abstract semantics. We use as intermediate function $\texttt{match}^\sharp : \mathcal{V}^\sharp \times \Sigma^\sharp \times \Pi \to \Sigma^\sharp \times \Sigma^\sharp$. Given an abstract value $v^\sharp$, an abstract environment $\sigma^\sharp$ and a pattern p, it returns two abstract environments. The first is an over-approximation of environments from $\sigma^\sharp$, in which p and $v^\sharp$ can match. The second is an over-approximation of environments in which they cannot. We rely on a filter function $\mathcal{F}^\sharp$ from the numerical domain. It restricts an abstract environment to keep only those where a given boolean predicate evaluates to true. It is useful to model the `when` guards appearing in patterns:

$$
\begin{aligned}
\texttt{match}^\sharp(v^\sharp, \sigma^\sharp, p \texttt{ when } e_1) = \ & \text{let } \sigma^\sharp_0, \sigma^\sharp_{\neg,0} = \texttt{match}^\sharp(\sigma^\sharp, v^\sharp, p) \text{ in} \\
& \text{let } e^n_1, \sigma^\sharp_1 = \mathbb{E}^\sharp[\![e_1]\!]\sigma^\sharp_0 \text{ in} \\
& \left(\mathcal{F}^\sharp[\![e^n_1 \neq 0]\!]\sigma^\sharp_1\right), \left(\sigma^\sharp_{\neg,0} \cup^\sharp \mathcal{F}^\sharp[\![e^n_1 = 0]\!]\sigma^\sharp_1\right)
\end{aligned}
$$

Thus, $v^\sharp$ matches p `when` e_1 in environments where $v^\sharp$ and p match, and when e^n_1 is non-zero. The other cases are simpler.

$$
\begin{aligned}
\mathbb{E}^\sharp[\![\text{match } e_0 \text{ with } p_1 \to e_1 \mid \cdots \mid p_m \to e_m]\!]\sigma^\sharp = \ & \\
& \text{let } \sigma^\sharp_1, \sigma^\sharp_{\neg,1} = \texttt{match}^\sharp(\mathbb{E}^\sharp[\![e_0]\!]\sigma^\sharp, p_1) \text{ in} \\
& \text{let } e'_1 = \text{match } e_0 \text{ with } p_2 \to e_2 \mid \cdots \mid p_m \to e_m \text{ in} \\
& \begin{cases} \mathbb{E}^\sharp[\![e_1]\!]\sigma^\sharp_1 \cup^\sharp \mathbb{E}^\sharp[\![e'_1]\!]\sigma^\sharp_{\neg,1} & \text{if } m \geq 1 \\ \{\omega\}, \sigma^\sharp & \text{if } m = 0 \wedge \sigma^\sharp \neq \bot \end{cases}
\end{aligned}
$$

The abstract semantics of pattern matching is the abstract union of all interpretations of branches e_i, in an environment in which e_0 matches the pattern p_i but not the previous patterns p_j for $j < i$. We add $\{\omega\}$ (match failure) if some part of the environment did not match any pattern. Our analysis can thus target match failure, i.e. when no pattern caught the value of e_0. This is the case if $\sigma^\sharp \neq \bot$ when no pattern is left. To the best of our knowledge, this is the first value analysis handling `when` clauses in pattern-matching exploiting value information to precisely detect non-exhaustive pattern-matching. The OCaml compiler does warn on non-exhaustive pattern-matching but, since its analysis is not value-sensitive, it is more conservative. For the following program, it will issue a "non-exhaustive pattern matching" warning, whereas our analysis, knowing that `l` starts with `Cons`, can prove that no case is left unmatched:

```
let l = Cons(1, Nil) in let x = match l with Cons(h,Nil)-> h.
```

```
let rec filter_le inf l =
  match l with
  | Cons(h, q) when h > inf -> filter_le inf q
  | Cons(h, q) when h <= inf -> Cons(h, filter_le inf q)
  | Nil -> Nil
in
let x = Cons(0, Cons(5, Cons(11, Nil))) in
let y =  filter_le 4 x in
let hd = match y with Cons(h,q) -> h | Nil -> 0 in
assert(hd <= 4)
```

Figure 6 A recursive function manipulating recursive algebraic objects.

4.3 Example

Before defining a domain for higher-order functions, we show with an example how our compositional function analysis works together with our ADT summarization domain and the abstract semantics of pattern-matching. We consider the program in Figure 6. It defines a function `filter_le`. This function of type $\tau = \texttt{int} \to \texttt{list} \to \texttt{list}$ takes as inputs an integer *inf* and a list l and keeps only the elements of l that are lower than *inf*.

Let *body* be the body of the function `filter_le`. We analyze it with unknown inputs, in $\sigma^\sharp = [l \to \top][inf \to \top]$. Its abstract semantics is $F : v^\sharp \to \text{fst}(\mathbb{E}^\sharp[\![body]\!]\sigma^\sharp[f \to v^\sharp])$. To get the abstract semantics of the recursive function, we compute the abstract fixpoint of the body semantics (Section 3.2). Starting with `filter_le` at $\bot_\tau = ((inf, l), r, \sigma^\sharp[r \to \bot])$, we compute the fixpoint $\nabla_{n\in\mathbb{N}} F^n(\bot_\tau)$:

$$\begin{aligned}
F(\bot_\tau) &= \mathbb{E}^\sharp[\![\text{fun } l \ inf \ \to body]\!]\sigma^\sharp[\texttt{filter_le} \to \bot_\tau] \\
&= ((inf, l), ((r.1.1, \bot), \{\texttt{Cons}\}, \sigma^\sharp[r.1.1 \to \bot])) \cup^\sharp_\tau \bot \\
&= (\bot_\tau, \{\texttt{Cons}, \texttt{Nil}\}, \sigma^\sharp[r.1.1 \to \bot]) \\
F^2(\bot_\tau) &= F(((inf, l), (r.1.1, \bot), \{\texttt{Cons}, \texttt{Nil}\}, \sigma^\sharp[r.1.1 \to \bot])) \\
&= (((inf, l), (r.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\}), \sigma^\sharp[r.1.1 \leq inf]) \\
F^3(\bot_\tau) &= F^2(\bot_\tau) \\
\nabla_{n\in\mathbb{N}} F^n(\bot_\tau) &= (((inf, l), (r.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\}), \sigma^\sharp[r.1.1 \leq inf])
\end{aligned}$$

By iteratively computing the summary of the function, we are finally able to infer that the elements in the output list ($r.1.1$) are lower than input *inf*. We denote this result as V_1, assign it to `filter_le` and continue the analysis:

$$\mathbb{E}^\sharp[\![\text{let rec } \texttt{filter_le} = body \text{ in } e]\!]\sigma^\sharp = \text{ let } \sigma_1^\sharp = \sigma^\sharp[\texttt{filter_le} \to V_1] \text{ in } \mathbb{E}^\sharp[\![e]\!]\sigma_1^\sharp$$

We abstract the assignment into x:

$$\begin{aligned}
\mathbb{E}^\sharp[\![\texttt{Cons}(0, \texttt{Cons}(5, \texttt{Cons}(11, \texttt{Nil})))]\!]\sigma_1^\sharp = \ & \text{let } \sigma_2^\sharp = \sigma_1^\sharp[x.1.1 \to 0 \cup_\mathbb{Z} 5 \cup_\mathbb{Z} 11] \text{ in} \\
& ((x.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\}), \sigma_2^\sharp = V_2, \sigma_2^\sharp \\
\mathbb{E}^\sharp[\![\text{let } x = \texttt{Cons}(0, \texttt{Cons}(5, \texttt{Cons}(11, \texttt{Nil}))) \text{ in } e_2]\!]\sigma_3^\sharp = \ & \text{let } \sigma_3^\sharp = \sigma_2^\sharp[x \to V_2] \text{ in } \mathbb{E}^\sharp[\![e_2]\!]\sigma_3^\sharp
\end{aligned}$$

We then abstract the assignment into y by applying `filter_le`.

$$\mathbb{E}^\sharp[\![\texttt{filter_le}\ 4\ x]\!]\sigma_3^\sharp = ((r.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\}),$$
$$(\sigma^\sharp[r.1.1 \leq inf])[inf = 4][l = \sigma_3^\sharp(x)]$$
$$((r.1.1, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\}), \sigma^\sharp[r.1.1 \leq 4] = V_3, \sigma_4^\sharp$$

$$\mathbb{E}^\sharp[\![\text{let } y = \texttt{filter_le}\ 4\ x \text{ in } e_2]\!]\sigma_4^\sharp = \text{ let } \sigma_5^\sharp = \sigma_4^\sharp[x.1.1 \to 0 \cup_{\mathbb{Z}} 5 \cup_{\mathbb{Z}} 11][y \to V_3] \text{ in}$$
$$\mathbb{E}^\sharp[\![e_2]\!]\sigma_5^\sharp$$

We then evaluate $e_1 =$ match y with $\mid \texttt{Cons}(h, q) \to h \mid \texttt{Nil} \to 0$. We assign the result to hd.

$$\mathbb{E}^\sharp[\![e_1]\!]\sigma_1^\sharp = h, \sigma_5^\sharp[hd \leq 4]$$

Since $hd \leq 4$, the assertion is proven. More generally, the summary for `filter_le` proves that for any list l and integer inf, `filter_le` inf l is a list whose content is lower than inf. The analysis also proves that the pattern-matching inside the function body is exhaustive.

5 Higher-Order, Disjunctive Relational Summaries

In a higher-order setting, functions can be inputs or output of other functions. This is a key concept of functional programming, making programs harder to analyze. We consider the program from Figure 1 and highlight the benefits of compositional analyses on this example.

Non-compositional analysis. Different choices were made in the literature to analyze higher-order functions. A method giving precise results [27] is simply to defer the analysis of `to_fun`'s body until we know its parameters, or an abstraction of them. Consequently, the body analysis is performed at every call site. Thus, the definitions of `f1` and `f2` only trigger a store of their definition code. When we compute `r1` and `r2`, all inputs are known, yielding: $\texttt{r1} = 5, \texttt{r2} = 9$. This method is able to infer precise results for `r1` and `r2`, with the drawback that the function body is reanalyzed at each call site. This is costly, and hinders scalability.

Compositional analysis. To circumvent this cost, we can perform a compositional analysis, where a function is analyzed only once, at its definition site. This is how `filter_le` was analyzed in Section 4.3, at first-order. Since functions are abstracted as points in the polyhedra space, intuitively, manipulating them as first class values for higher order does not add further complexity. The function domain is simply based on the polyhedra one. We can analyze `to_fun` once and for all, supposing that we do not know anything about `f`'s behavior. We get $\texttt{r1} = \top, \texttt{r2} = \top$. This loss of information is due to a join of both possible cases from `to_fun`'s behavior. However, this is not inherent to compositional analysis, since the specification of `to_fun` is:

$$\texttt{to_fun}(c, x) = \begin{cases} n & \text{if } c = \texttt{Cst}(n) \\ f(x) & \text{if } c = \texttt{Fun}(f) \end{cases}$$

It is thus possible to describe `to_fun`'s behavior without a priori information on possible values for c, as long as *disjunctions* on those possible values are possible. The precision loss mentioned above was indeed due to the impossibility to express disjunctions when abstracting higher-order functions. This is why we focus in this section on lifting partitioning methods to higher-order.

5.1 Partitioning Function Summaries

Even at first-order, a similar precision loss occurs on very simple programs:

```
let binary x = if x > 0 then 10 else -10
```

By conducting a compositional non-relational analysis with the interval domain, we are only able to infer that the output is in $[-10, 10]$, a quite weak result. This comes from the fact that `binary` has two very distinct behaviors depending on the input value. Joining them induces a precision loss. The issue can be overcome by partitioning on the input [7], then keeping separate those behaviors: $\lambda x.(x > 0 \wedge res = 10) \vee (x \leq 0 \wedge res = -10)$. Note that relationality cannot, on its own, overcome this cause of imprecision. The polyhedra domain gives the same result as intervals on `binary`. However, Boutonnet and Halbwachs [8] successfully use relationality and partitioning together. Consider the following example:

```
let max x y = if x >= y then x else y
```

With the interval domain, even with partitioning, we cannot express anything interesting about `max`, whereas the polyhedra domain alone can only infer: $\lambda xy.x \leq res \wedge y \leq res$. But with *disjunctive relational summaries*, we can separate disjunct behaviors, then summarizing the maximum function with: $\lambda xy.(x < y \wedge res = y) \vee (x \geq y \wedge res = x)$. Maintaining multiple partitions is costly, so a key point of partitioning is to decide when to partition and when to merge different behaviors. Various heuristics were developed on that topic. Those methods are compatible with recursion. Indeed, Bourdoncle [7] and Boutonnet and Halbwachs [8] are able to give precise summaries for McCarthy 91 function.

We want to benefit from this precision improvement at higher-order. In a higher-order setting however, functions are first class values: they can be input or output of other functions, etc. As first class objects, they should be represented as abstract values. Compared to previous works at first-order, we then need to define the *abstract domain of functions*. To the best of our knowledge, this is the first presentation of disjunctive summaries as a domain abstracting functions complete with all its lattice operators, including join and widening.

5.2 Disjunctive Relational Summaries as a Domain on Functions

As in Section 3.2, we choose $(\mathcal{R}(V), \sqsubseteq_r^\sharp, \sqcup_r^\sharp, \sqcap_r^\sharp, \nabla_r^\sharp, \gamma_r, \mathrm{proj}_r, \lambda_r)$ a relational domain on V. This way, the formalization is independent of the chosen domain.

▶ **Definition 5.1** (Function domain). *Given a function type $\tau = \tau_1 \to \cdots \tau_n \to \tau_r$, we define $\mathcal{F}_\tau^\sharp$ the set of elements abstracting functions of type τ. Let n_i be the number of variables necessary to abstract an object of type τ_i.*

$$\mathcal{F}_\tau^\sharp = \{(V, r, (p_i)_{[\![1,m]\!]}) \mid V = (x_1, \ldots, x_n), \forall i, x_i \in \mathbb{V}^{n_i}, \forall i, j, i \neq j \Rightarrow x_i \neq x_j, r \in \mathbb{V}, \\ \forall i \in [\![1, m]\!], p_i \in \mathcal{R}(V \cup \{r\})\}$$

A function is thus abstracted as a triplet of: a n-tuple of input variables, an output variable, and a collection of m relations abstracting different cases in the function's behavior. We denote $(x_1, \ldots, x_n), r, (p_1, \ldots, p_m)$ as $\lambda x_1 \ldots x_n. \bigvee_{i=1}^{m} p_i(x_1, \ldots, x_n, r)$. For example, `let add x y = x + y` is abstracted as $((x, y), r, (r = x + y))$, written $\lambda xy.r = x + y$.

▶ **Definition 5.2** (Function concretization). *For* $\tau = \tau_1 \to \cdots \tau_n \to \tau_r$, $f = ((x_i)_{i\in[\![1,n]\!]}, r, (p_i)_{i\in[\![1,m]\!]}) \in \mathcal{F}^\sharp_\tau$, *the concretization is:*

$$\gamma_\tau(f, _) = \{f : \tau \mid \forall (a_1, \ldots, a_n) : \tau_1 \times \cdots \times \tau_n, \\ [x_1 = a_1] \cdots [x_n = a_n][r = f(a_1, \ldots, a_n)] \in \bigvee_{i=1}^{m} \gamma_r(p_i)\}$$

This concretization is simply an update of Definition 3.4, with a disjunction of relations instead of a unique one.

▶ **Example 5.3.** For $f^\sharp = \lambda x_1 x_2.(x_1 \leq x_2 \wedge r > 0) \vee (x_1 > x_2 \wedge r < 0)$, $\tau =$ `int` $\to$ `int` $\to$ `int`:

$$\gamma_\tau(f^\sharp, _) = \{f : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \mid \forall x_1, x_2 \in \mathbb{N}, \\ x_1 \leq x_2 \implies f(x_1, x_2) > 0 \wedge x_1 > x_2 \implies f(x_1, x_2) < 0\}$$

That is, $f^\sharp$ represents all functions whose result is lower than 0 when the first input is the smallest, and greater than 0 otherwise.

We need to define comparison, join, meet, and widening operators on functions. Since we consider only well-typed programs, those operators will be defined for functions of identical type: in particular, they will have the same number of parameters. These definitions build upon relational domain operators.

Note that renaming the inputs or the output of the function does not change its concretization. Without loss of generality, we assume when comparing two functions that they use the same names for inputs or outputs (this can always be ensured by renaming). We then define abstract inclusion, meet, join, and widening operators on the functional domain:

▶ **Definition 5.4** (Inclusion). *For* $f = V, r, (f_i)_{[\![1,n_1]\!]}$, $g = V, r, (g_i)_{[\![1,n_2]\!]}$, *the inclusion* $\subseteq^\sharp$ *on* $\mathcal{P}^\sharp_n$ *is:* $f \subseteq^\sharp g \iff \forall i \in [\![1, n_1]\!], \exists j \in [\![1, n_2]\!], f_i \sqsubseteq^\sharp_r g_j$

f is included in g if every disjunct of f is included in a disjunct of g. This order is the classic order for disjunctive completion domains [1].

▶ **Example 5.5.** With this definition, given two functions of type `int -> int`:

$$x, r, \begin{cases} x > 1 \wedge r = 10 \\ x = 0 \wedge r = 10 \\ x < 0 \wedge r = \top \end{cases} \quad \subseteq^\sharp \quad x, r, \begin{cases} x \geq 1 \wedge r = 10 \\ x < 1 \wedge r = \top \end{cases}$$

▶ **Definition 5.6** (Meet). *Given* $f = V, r, (f_i)_{i\in[\![1,m_1]\!]}$ *and* $g = V, r, (g_i)_{i\in[\![1,m_2]\!]}$, *the abstract intersection of functions is defined as* $\sqcap^\sharp_f$*:*

$$f \sqcap^\sharp_f g = V, r, \left(\bigsqcup_{j=1}^{m_2} (f_i \sqcap^\sharp_r g_j) \right)_{i\in[\![1,n_1]\!]}$$

That is, given a disjunct f_i, we compute its meet with every disjunct g_j and then join them together, keeping n_1 partitions. Indeed, for performance reasons, we do not want to maintain all (quadratic) combinations of pairwise intersections. This method ensures that we keep the number of partitions used in the right argument. This abstraction of the meet is then not commutative. An alternative definition would partition with regard to the abstraction with the highest number of disjuncts, which would likely be more precise. An experimental study, left for future work, could be led to determine the best version.

▶ **Example 5.7.** Given two functions of type `int -> int` with different input partitioning:

$$n, r, \begin{cases} n \geq 0 \wedge 1 \leq r \leq 3 \\ n < 0 \wedge -3 \leq r \leq -1 \end{cases} \quad \sqcap_f^\sharp \quad n, r, \begin{cases} n \geq 1 \wedge 2 \leq r \leq 4 \\ n = 0 \wedge r = 3 \\ n < 0 \wedge -4 \leq r \leq -2 \end{cases}$$

$$= n, r, \begin{cases} (n \geq 1 \wedge 2 \leq r \leq 3) \sqcup_p^\sharp (n = 0, r = 3) \sqcup_p^\sharp \bot_p \\ \bot_p \sqcup_p^\sharp \bot_p \sqcup_p^\sharp n < 0 \wedge -3 \leq r \leq -2 \end{cases}$$

$$= n, r, \begin{cases} n \geq 0 \wedge 2 \leq r \leq 3 \\ n < 0 \wedge -3 \leq r \leq -2 \end{cases}$$

▶ **Definition 5.8** (Join). *Given $f = V, r, (f_i)_{i \in [\![1, n_1]\!]}$ and $g = V, r, (g_i)_{i \in [\![1, n_2]\!]}$, the abstract union of functions is defined as $\sqcup_f^\sharp$. We write $G_i = \{g_j \mid 1 \leq j \leq n_2 \wedge proj_r(g_j) \sqsubseteq proj_r(f_i)\}$ the set of g_j whose constraints on the inputs are included in the constraints of f_i, and $G^\neg = \{g_j \mid 1 \leq j \leq n_2 \wedge \forall 1 \leq i \leq n_1, g_j \notin G_i\}$ the set of g_j not included in any G_i.*

$$f \sqcup_f^\sharp g = V, r, (f_i \bigsqcup_{g_j \in G_i} g_j)_{i \in [\![1, n_1]\!]} \vee (g_j)_{j \in [\![1, n_2]\!] \wedge g_j \in G^\neg}$$

That is, we keep every disjunct f_i, and join them with disjuncts of g respecting their conditions on the input. We add disjuncts g_j that are compatible with no f_i. Similarly to the meet operator, we do not want to simply keep every disjunct, so as to limit their number.

▶ **Example 5.9.** Given two functions of type `int -> int`:

$$n, r, \begin{cases} n \geq 0 \wedge 1 \leq r \leq 3 \\ n < 0 \wedge -3 \leq r \leq -1 \end{cases} \quad \sqcup_f^\sharp \quad n, r, \begin{cases} n \geq 1 \wedge 2 \leq r \leq 4 \\ n = 0 \wedge r = 3 \\ n < 0 \wedge -4 \leq r \leq -2 \end{cases}$$

$$= n, r, \begin{cases} (n \geq 0 \wedge 1 \leq r \leq 3) \sqcup_p^\sharp (n \geq 1 \wedge 2 \leq r \leq 4) \sqcup_p^\sharp (n = 0 \wedge r = 3) \\ (n < 0 \wedge -3 \leq r \leq -1) \sqcup_p^\sharp (n < 0 \wedge -4 \leq r \leq -2) \end{cases}$$

$$= n, r, \begin{cases} n \geq 0 \wedge 1 \leq r \leq 4 \\ n < 0 \wedge -4 \leq r \leq -1 \end{cases}$$

Widening is defined the same way as join, replacing $\sqcup_p^\sharp$ with $\nabla_p^\sharp$. However, to ensure convergence, we also limit the number of disjuncts to a user-controlable constant. When performing widening, we join disjuncts when necessary to respect this limit, before delegating widening to the underlying domain.

In conclusion, we defined the set $\mathcal{F}_\tau^\sharp$ alongside its concretization, inclusion, join, meet, and widening operators, as an abstract domain for functions of type τ. We now explain how to apply a computed summary in $\mathcal{F}_\tau^\sharp$ at a call site, to evaluate the effect of a function call.

5.3 Function Application with Partitioning

Function summaries being now disjunctive, we need to update the semantics from Section 3.2:

$$\mathbb{E}^\sharp[\![e_0 \dots e_k]\!]\sigma^\sharp = \text{let } (x_1, \dots, x_n, r, (p_i)_{i\in[\![1,m]\!]}), \sigma_0^\sharp = \mathbb{E}^\sharp[\![e_0]\!]\sigma^\sharp \text{ in} \tag{5}$$

$$\text{let } \forall i \in [\![0, k-1]\!]\ e_{i+1}^\sharp, \sigma_{i+1}^\sharp = \mathbb{E}^\sharp[\![e_{i+1}]\!]\sigma_i^\sharp \text{ in} \tag{6}$$

$$\text{let } \forall i \in [\![1, m]\!]\ \sigma_{k,i}^\sharp = \texttt{add_vars}(\sigma_k^\sharp, \texttt{dom}(p_i)) \text{ in} \tag{7}$$

$$\text{let } \forall i \in [\![1, m]\!]\ p_i = \texttt{add_vars}(\lambda_r(p_i), \texttt{dom}(\sigma_n^\sharp)) \text{ in} \tag{8}$$

$$\text{let } \forall i \in [\![1, m]\!]\ p_i' = (p_i \sqcap^\sharp \sigma_{k,i}^\sharp)[x_j = e_j^\sharp]_{j\in[\![1,k]\!]} \text{ in} \tag{9}$$

$$\text{let } X = \{x_{k+1}, \dots, x_n\} \text{ in} \tag{10}$$

$$\begin{cases} \left(X, r, (\text{proj}_r(p_i'))_{i\in[\![1,m]\!]\wedge p_i'\neq\bot}\right), \sigma^\sharp & \text{if } k < n \\ r, (p_i')_{i\in[\![1,m]\!]\wedge p_i'\neq\bot} & \text{otherwise} \end{cases} \tag{11}$$

First, we compute the semantics of the function e_0 (5). Then we compute sequentially the semantics of its argument expressions e_1, …, e_k (6). As in Section 3.2, we extend p_i and $\sigma_k^\sharp$ to the same definition domain (7,8). We combine them and add to each function disjunct p_i the equality constraint between formal argument x_j and value $e_j^\sharp$ (9). X is the set of free formal arguments (not bound to an actual argument) in case of partial application (10).

Partial application comes naturally (case $k < n$ in (11)). When partially applying a function, we project the result on the set of remaining variables. The result is then a set of relations consisting in a disjunctive summary. It abstracts the function resulting from the partial application. When the application is total (otherwise in (11)), there is no remaining parameters: we simply change the format of the result. In both cases, we remove empty disjuncts $p_i' = \bot$, which correspond to unsatisfiable constraints on the inputs.

▶ **Example 5.10.** The summary of the function `max` is $(x, y), r, (x > y \wedge r = x) \vee (x \leq y \wedge r = y)$. Then partially applying it with one argument a, we get the abstraction:

$$\mathbb{E}^\sharp[\![\texttt{max}\ a]\!]\{a \geq 5\} = y, r, \{(a > y \wedge r = a \wedge a \geq 5) \vee (a \leq y \wedge r = y \wedge a \geq 5)\}$$

This way, we deduce that the function returns a result which is always greater than 5, and is y if it is greater than a and a otherwise.

5.4 Function Analysis with Partitioning

We update function analysis semantics from Section 3.2 to allow a disjunctive result:

$$\mathbb{E}^\sharp[\![\text{fun } x_1 \cdots x_n \to body]\!]\sigma^\sharp = \text{let } c_1, \dots, c_p = \texttt{get_preconditions}(body) \text{ in} \tag{12}$$

$$\text{let } \forall i \in [\![1, p]\!],\ e_i^\sharp, \sigma_i^\sharp = \mathbb{E}^\sharp[\![body]\!](\sigma^\sharp \wedge c_i)[x_j \to \top] \tag{13}$$

$$\text{and } s_i = \text{proj}_r(\sigma_i^\sharp[r \to e_i^\sharp]) \text{ in} \tag{14}$$

$$\left((x_1, \dots, x_n), r, (s_i)_{i\in[\![1,p]\!]}\right), \sigma^\sharp \tag{15}$$

The `get_preconditions` helper function scans the body to get a set of predicates $(c_1, \dots, c_p)$ suitable for partitioning (12). Then, we analyze the body of the function with each precondition c_i (13) and get a set of possible behaviors s_i (14). The function is then abstracted as the disjunction of those behaviors for the function (15). Note that we do not keep partitioned states *inside* the analysis of a recursive function, to avoid the risk of diverging

sets of partitions. Consequently, when applying a function inside the body of the recursive function, we immediately merge cases. also note that this definition is independent of the chosen `get_preconditions` partitioning heuristics.

We chose to use a heuristics, so called *local reachability*, introduced by Boutonnet and Halbwachs [8]. We try to partition regarding conditions controlling reachability to certain key points – branches in conditional branching or pattern matching. Note that finding those preconditions is a pre-analysis, performed just before analyzing the function itself. Consequently, when a function body we are either analyzing or pre-analyzing performs a function call, the abstract summary of the called function is already available (or it is set to bottom if this is the first iteration of a recursive call).

To sum up the method, the body of the function is pre-analyzed to decide a partitioning on the inputs. It starts with a precondition $I^\sharp$, which can be $\top$, and follows those steps:

1. Analyze the function body with an interprocedural analysis and note $r_i^\sharp$ the result at each reachable control point;
2. Choose a control point for which $r_i^\sharp \neq \bot$;
3. Choose an abstract value $s^\sharp$ which is complementable, i.e. its complementary can be expressed in the domain (note that $x \leq y$ as well as $x > y$ tests are always complementable for polyhedra on integers) such that $\text{proj}_r(r_i^\sharp) \sqsubseteq^\sharp s^\sharp$. We get disjuncts $I^\sharp \sqcap s^\sharp$ and $I^\sharp \sqcap \bar{s}^\sharp$.

We may iterate this algorithm to get more disjuncts and gain more precision.

6 Combining both Domains into an Analysis

Sections 4 and 5 presented, respectively, independent extensions of a first-order numeric analysis to algebraic data types and to higher-order functions. Thanks to their parametricity, we can combine these two extensions to analyze our target language from Section 2:

- Algebraic data types can contain higher-order values (i.e., functions). Indeed, the parametricity of the ADT abstraction supports functions as leaf data types, themselves represented by the higher-order part of our work.
- Function summaries from the higher-order section can express precise properties on ADTs they manipulate (including the case of recursive functions manipulating recursive ADT). Indeed, the abstract domain representing functions is parametric in the domain chosen to express relations between inputs and outputs (e.g., polyhedra).

We show how the resulting analysis performs on our example function `to_fun` and state its soundness theorem.

6.1 Analysis Cooperation Example

We choose as relational domain the polyhedra domain, combined with a domain stating the equality of functions. We analyze our example from the introduction. We recall its code:

```
1  let to_fun a =
2    match a with
3    | Cst n -> (fun x -> n) (* p1 *)
4    | Fun f -> f (* p2 *)
```

Pre-analysis by local reachability. We analyze the function with a classic relational analysis. We denote the control point after the first branch of the match as p_1 and the one after the second branch of the match as p_2. We denote possible constructors of a as a_c.

The analysis in p_1 gives us $r_1^\sharp = (x, r, r = n), [a_c = \{\texttt{Cst}\} \wedge a.1.1 = n] \neq \bot$. We see that $proj(\{a_c, a.1.1, a.2.1\}, r_1^\sharp) = [a_c = \{\texttt{Cst}\}]$ is complementable, of complementary $[a_c = \{\texttt{Fun}\}]$. Consequently, we get two disjuncts: $a_c = \{\texttt{Cst}\}$ and $a_c = \{\texttt{Fun}\}$.

Full analysis. Let *body* be the body of `to_fun`.

$$\mathbb{E}^\sharp[\![\text{fun } a \to body]\!]\sigma^\sharp = (a_c, a.1.1, a.2.1), r, \begin{cases} \mathbb{E}^\sharp[\![body]\!]\sigma^\sharp[a_c = \{\texttt{Cst}\}] \\ \mathbb{E}^\sharp[\![body]\!]\sigma^\sharp[a_c = \{\texttt{Fun}\}] \end{cases}$$

We analyze the first case. We have $\texttt{match}^\sharp(\sigma^\sharp[a_c = \{\texttt{Cst}\}], a, p_1) = \sigma^\sharp[a_c = \{\texttt{Cst}\}], \bot$ so:

$$\begin{aligned} \mathbb{E}^\sharp[\![body]\!]\sigma^\sharp[a_c = \{\texttt{Cst}\}] &= \mathbb{E}^\sharp[\![\text{fun } x \to n]\!]\sigma^\sharp[a_c = \{\texttt{Cst}\}] \cup^\sharp \bot^\sharp \\ &= (x, r', r' = n), [a_c = \{\texttt{Cst}\} \wedge a.1.1 = n] \end{aligned}$$

Similarly, for the second case:

$$\begin{aligned} \mathbb{E}^\sharp[\![body]\!]\sigma^\sharp[a_c = \{\texttt{Fun}\}] &= \mathbb{E}^\sharp[\![\text{fun } x \to n]\!]\sigma^\sharp[a_c = \{\texttt{Fun}\}] \cup^\sharp \bot^\sharp \\ &= f, [a_c = \{\texttt{Fun}\} \wedge a.2.1 = f] \end{aligned}$$

In the end, the semantics of `to_fun` is:

$$\mathbb{E}^\sharp[\![\text{fun } a \to body]\!]\sigma^\sharp = (a_c, a.1.1, a.2.1), r, \begin{cases} [a_c = \{\texttt{Cst}\} \wedge r = (x, r', r' = a.1.1)] \\ [a_c = \{\texttt{Fun}\} \wedge a.2.1 = f] \end{cases}$$

We denote this summary as s and bind `to_fun` to s in the environment. Note that this summary is as precise as in the concrete. We can evaluate `f_1 = to_fun Cst`(5) and `f_2 = to_fun Fun`$(\text{fun } x \to n)$:

$$\begin{aligned} \mathbb{E}^\sharp[\![\texttt{to_fun Cst}(5)]\!]\sigma^\sharp &= r, \sigma^\sharp[a_c = \{\texttt{Cst}\} \wedge r = (x, r', r' = a.1.1)] \\ &\quad [a_c = \{\texttt{Cst}\} \wedge a.1.1 = 5] \\ &= r, \sigma^\sharp[a_c = \{\texttt{Cst}\} \wedge a.1.1 = 5 \\ &\quad \wedge r = (x, r', r' = a.1.1)] \\ \mathbb{E}^\sharp[\![\texttt{to_fun Fun}(\text{fun } x \to n)]\!]\sigma^\sharp &= r, \sigma^\sharp[a_c = \{\texttt{Fun}\} \wedge a.2.1 = f] \\ &\quad [a_c = \{\texttt{Fun}\} \wedge a.2.1 = (x, r', r' = x + 1)] \\ &= r, \sigma^\sharp[a_c = \{\texttt{Fun}\} \wedge a.2.1 = (x, r', r' = x + 1) \\ &\quad \wedge r = a.2.1] \end{aligned}$$

In both cases, one of the summary disjuncts has an empty intersection with the environment, so it does not appear in the final state. Applying the summaries, we are able to infer $r_1 : 5, r_2 : 9$. In the end, we are able to recover the same precision as a non-compositional method (Section 5) while staying compositional. This way, we do not have to re-analyze `to_fun`'s body when computing r_1 and r_2, instead using the summaries inferred at call site. Additionally, we get a precise contract for `to_fun`, which is valid for every input.

6.2 Analysis Soundness

The analysis defined in this article is sound, i.e. the analysis of a program P over-approximates the reachable states of P. Section 3.1 defined the concretization of abstract environments (Definition 3.2) and the concretization of the abstract semantics of an expression (Remark 3.3).

▶ **Theorem 6.1.** *The abstract semantics* $\mathbb{E}^\sharp$ *is sound, i.e.:*

$$\forall \sigma \in \Sigma, \forall \sigma^\sharp \in \Sigma^\sharp, \forall e : \tau \in \mathcal{E}, \sigma \in \gamma_{\Sigma^\sharp}(\sigma^\sharp) \implies \mathbb{E}[\![e]\!]\sigma \in \gamma_\tau(\mathbb{E}^\sharp[\![e]\!]\sigma^\sharp)$$

Proof. The proof is by induction over the syntax of expression e. ◀

7 Experimental Evaluation

7.1 Implementation

The methods described in the article were implemented in MOPSA [22], an open source and multi-language platform to ease the development of abstract analyzers. Similarly to other static analyzers by abstract interpretation, such as Astrée [6], Frama-C [9], Infer [16], or Julia [40], MOPSA is based on a composition of several abstract domains. Compared to them, it goes further in terms of modularity, enforcing a finer level of granularity of abstractions. It encourages relational abstractions for all types and features powerful domain communication mechanisms (such as reduced products, cartesian products, expression rewriting) and supports the reuse of abstractions in the analysis of widely different languages (such as C [37, 23] and Python [33, 34, 35]). The core of the platform is composed of 22 000 LoC of OCaml.

We added 3 000 LoC to support OCaml analysis and to implement the algebraic data types and higher-order domains described in the article. Our analysis is fully automatic: it leverages typing information from the frontend – handled by the OCaml compiler – to automatically associate abstract domains to values, based on their types. The user sets a global precision level by selecting more or less expressive domains for base types (e.g., polyhedra or intervals for integers). Choosing the best precision/performance trade-off is a long-standing problem for the analysis of any language. It is not addressed by this paper.

In MOPSA, a configuration file describes how domains are combined to define an analysis. A configuration for our analysis is shown in Figure 7. Sequences delegate the analysis of an expression to the right-hand side domain when relevant. Reduced products perform reductions between both domain results. Cartesian products implement collaborations between domains. `ML.program` is the frontend, handling the import from the OCaml compiler. `ML.intraproc` handles conditionals and assertions. `ML.let_rec` computes an abstract fixpoint for recursive definitions of variables. Domains `ML.functions` (defined in Section 5.2) and `ML.adts` (defined in Section 4.1) depend on each other because of their parametricity. They share underlying domains for ground types, here strings and integers. MOPSA includes a Universal language, implementing ready-to-use abstractions for basic constructs. Here, integers are abstracted by the reduced product of the polyhedra domain `U.polyhedra` and the congruence domain `U.congruences`. `U.string` abstracts strings. The configuration could be defined using other domains for ground types as well. Our implementation is parametric in those domains.

On a technical note, our ADT domain required support for heterogeneous operations, that is, operations on environments defined on different sets of variables [24], which is the case when joining two match cases. Our functions' domain required adapting the already existing trace partitioning [30] for C-like languages on the platform. Sound operations on summarization variables from Gopan et al. [18] were already part of the platform and were leveraged when handling summarization variables of recursive algebraic data types.

7.2 Experiments

We tested our implementation on the examples from this article as well as 40 handwritten programs highlighting the precise handling of recursive functions manipulating algebraic data types, functions partitioning on algebraic constructors, partial application, algebraic data

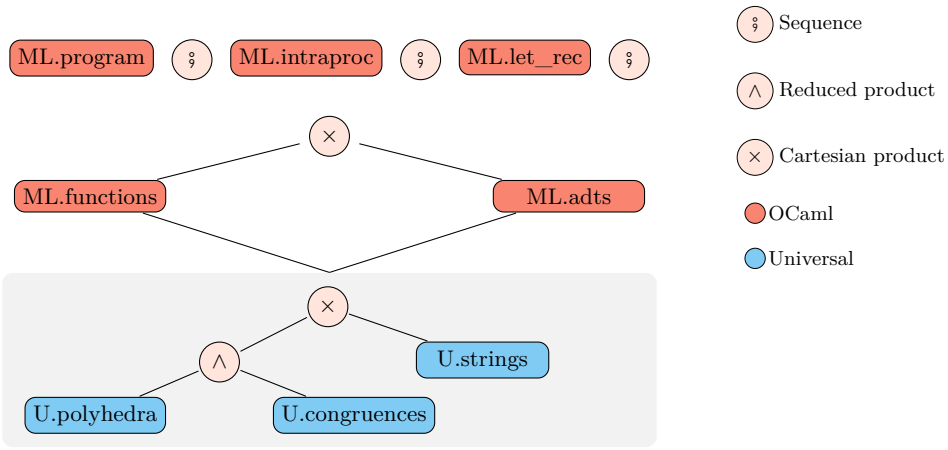


Figure 7 OCaml configuration in MOPSA.

Program name	Analysis (ms)	Program name	Analysis (ms)
`numeric_loop3b.ml`	12	`filter_le.ml`	131
`binomial.ml`	400	`make_list.ml`	131
`mc91.ml`	30	`match_when.ml`	9
`tak.ml`	923	`is_exhaustive.ml`	10
`abs.ml`	15	`partial_app.ml`	15
`xor.ml`	36	`embed_fun.ml`	9
`rec_add.ml`	44	`to_fun.ml`	11
`non_terminate.ml`	6	`f_from_g.ml`	11

Figure 8 Benchmarks for the OCaml analysis.

types containing functions, etc. In particular, it inferred correct summaries for `filter_le` and `to_fun` functions. Additionally, we selected 20 programs from Salto's benchmarks [27] that are compatible with the current limitations of our approach, i.e. they do not contain modules, nor imperative features, nor polymorphism. Figure 8 displays the analysis time for an extract of our 60 programs. The first five programs are from Salto's benchmarks. The programs we consider are small: they consist in around a dozen lines each.

Precision. For most numerical recursive functions (e.g. `binomial`, `ackerman`, `mc91`), our relational compositional analysis gives results at least as precise as Salto's non-compositional and non-relational approach, but with only one analysis of the body. This illustrates how relationality and disjunctions can recover precision lost by compositionality, thus enhancing performance when the analyzed function is analyzed multiple times. Besides, for some functions, no precise invariant can be discovered without relationality – e.g. `rec_add`, or `numeric_loop3b` from Salto's benchmark. Our analysis was therefore of the same precision or better than Salto on 14 out of those 16 examples. Finally, as explained in Section 4.2 our analysis is the first to support `when` clauses, and to detect non-exhaustive pattern-matching.

Performance. All our tests were performed on a Intel(R) Core(TM) i7-8565U CPU 1.80GHz with 16 GB of RAM. The analysis for those small programs is lower than a second. The slowest analysis was for the `tak` function, making three recursive calls in its body. Those

runtimes are comparable to the analysis performed by Salto – less than a second. The runtime difference on the (small) benchmarks available is not significant enough to deduce that one analysis is faster, and to evaluate which part of our method results in a speed-up and which part in a slowdown. Moreover, Salto and MOPSA feature different abstractions beside compositionality that may impact the analysis time as well. A more thorough evaluation of runtime trade-offs is thus left for future work. Previous work by Boutonnet and Halbwachs [8] at first-order concluded that summary construction time is often negligible with regard to total analysis time. Besides, in our implementation, maintaining a summary is equivalent to storing it (they are immutable, as functions are currently pure). We optimistically hope that future work would draw a similar conclusion on higher-order programs.

Note that our method's interest goes beyond performance: it automatically generates *contracts* for higher-order functions. Therefore, it plays a role in proving not only the absence of runtime errors, but also refined properties on the program behavior, being able to automatically discover functions specifications.

8 Related Work

Type systems. To prove properties on functional programs, type systems are widely used. The simplest ones already prevent some errors, such as adding a function to an integer. More expressive type systems, used for program verification, include dependent types, and in particular refinement types. They have enjoyed a steady popularity over the years [44, 43], relying under-the-hood on SMT solvers to reason on properties, mainly numeric ones. Although formulated within the framework of abstract interpretation, we believe our work shares similarities, as it provides a compositional analysis and infers numeric invariants.

Deductive methods. Deductive methods, such as Cameleer [38] or F* [41], can prove precise properties, such as correctness with regard to a specification, but need often both user annotations and SMT solvers. It differs from our goal, a fully-automated and solver-free method, to infer semantic properties.

Non-value abstract interpretation analysis. Whereas many works approach static analysis of functional languages, they often focus on control flow analysis, which, as precised in Liang and Might [28], does not suffice to keep track of values through flow. Cousot and Cousot [13] define higher-order functions abstractions, e.g. as relations or as sets. They also formulate refinements, e.g. disjunctive completion. They are however mainly interested in comportment analysis such as strictness or termination. We use similar abstractions and refinements, but to define a value analysis. Besides, we support algebraic data types with an abstraction fully parametric in possibly relational domains. Montagu and Jensen [36] develop methods to infer a form of frame condition for pure functional higher-order languages, i.e. identifying equality relations between parts of algebraic values. They suffer from the same lack of information when applying an input function as we do. They are however limited to non-recursive types, and cannot handle complex numeric relations.

Abstract interpretation-based value analysis. In our experimental evaluation, we compared to Salto [27], a static value analyzer verifying OCaml programs. It supports a wider subset of the language, such as side-effects and modules. Their method differs from ours, being non-relational and non-compositional. Our compositional approach could be more scalable. Journault et al. [24] propose a relational abstract domain for trees but are limited to this

specific data structure. Bautista et al. [3], Bautista et al. [4] describe a relational abstraction for numeric algebraic values and infer structural equalities between non-numeric fields, but are limited to non-recursive objects. Valnet et al. [42] support recursion, but not higher-order.

Compositional analysis. Input-output relational analysis has been long studied, first applied to `while` programs [26]. Our method can be seen as an extension of symbolic relational separate analysis from Cousot and Cousot [14], extended to support ADTs and higher-order. This is a long-known method [14] to improve analysis scalability. Compositionality has been used in multiple settings: Farzan and Kincaid [17] use them to analyze independently decomposed parts of a program and Kincaid et al. [25] for inter-procedural analysis, analyzing procedures independently of their calling context, once and for all. Codish et al. [11] develop a compositional analysis for logic programs, therefore proving it useful in a declarative context. Bautista et al. [2, 5] define an ADT domain in an input-output analysis for non-recursive imperative programs. In the MOPSA platform [22], a prototype modular analysis was implemented for C programs manipulating strings [23]. To improve precision, Bourdoncle [7] expresses non-relational disjunctive summaries. Boutonnet and Halbwachs [8], on which we built upon, permit relationality. All those methods are however limited to first-order.

9 Conclusion and Future Work

This article presents two abstract domains, one tailored to handle recursive ADTs, and the other one to represent functions used as first-class values, as well as higher-order functions. Thanks to the parametricity of both domains, each domain can leverage the other, e.g. to abstract functions manipulating ADTs (and conversely). The combination of these two domains yields a compositional and relational analysis for a pure functional programming language. This analysis has been implemented into the MOPSA framework to analyze a pure subset of the OCaml language. Our preliminary evaluation shows the precision of our approach on 60 programs, including 20 benchmarks from Salto [27].

We now review the limitations of our approach and discuss future work. The language we have studied here is pure; extending our analysis to support references will be challenging. Our analysis focuses on a monomorphic functional programming language; we could rely, in future work, on the polymorphic equality domain from Montagu and Jensen [36] to handle polymorphism. Our current analysis does not detect arithmetic overflows: we believe this is an important, yet orthogonal concern. Our analysis is technically able to infer ranges of numerical variables and can thus detect overflows. However, our OCaml analysis does not support the wraparound semantics of overflows for now. To the best of our knowledge, there are no compositional value analysis currently tackling this issue, even in the case of first-order imperative languages. Previous work on compositional static analysis for first-order languages (e.g. Boutonnet and Halbwachs [8]) proved that compositionality is useful for scaling up. We postulate that similar speed-ups could be achieved in a higher-order setting with compositional analyses. While this article provides a contribution towards this goal by proposing a precise and compositional analysis, we have not evaluated its scalability in our preliminary experiments. We believe this work is a first step towards automatically proving functional properties (e.g., sorting [19]) in a functional setting with higher-order functions.

References

1 Samson Abramsky and Achim Jung. *Domain theory.* Oxford University Press, 1994.
2 Santiago Bautista. *Static Analysis of Algebraic Data Types and Arrays.* PhD thesis, ENS Rennes, 2023.

3 Santiago Bautista, Thomas Jensen, and Benoît Montagu. Numeric domains meet algebraic data types. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains*, pages 12–16, 2020. doi:10.1145/3427762.3430178.

4 Santiago Bautista, Thomas Jensen, and Benoît Montagu. Lifting numeric relational domains to algebraic data types. In *International Static Analysis Symposium*, pages 104–134. Springer, 2022. doi:10.1007/978-3-031-22308-2_6.

5 Santiago Bautista, Thomas Jensen, and Benoît Montagu. An input–output relational domain for algebraic data types and functional arrays. *Formal Methods in System Design*, pages 1–74, 2024.

6 Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, Xavier Rival, et al. Static analysis and verification of aerospace software by abstract interpretation. *Foundations and Trends® in Programming Languages*, 2(2-3):71–190, 2015. doi:10.1561/2500000002.

7 François Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4):407–435, 1992.

8 Rémy Boutonnet and Nicolas Halbwachs. Disjunctive relational abstract interpretation for interprocedural program analysis. In *Verification, Model Checking, and Abstract Interpretation: 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13–15, 2019, Proceedings 20*, pages 136–159. Springer, 2019. doi:10.1007/978-3-030-11245-5_7.

9 David Bühler. *Structuring an abstract interpreter through value and state abstractions: eva, an evolved value analysis for Frama-C*. PhD thesis, Université de Rennes 1, 2017.

10 Marc Chevalier and Jérôme Feret. Sharing ghost variables in a collection of abstract domains. In *VMCAI*, volume 11990 of *Lecture Notes in Computer Science*, pages 158–179. Springer, 2020. doi:10.1007/978-3-030-39322-9_8.

11 Michael Codish, Saumya K Debray, and Roberto Giacobazzi. Compositional analysis of modular logic programs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 451–464, 1993. doi:10.1145/158511.158703.

12 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977. doi:10.1145/512950.512973.

13 Patrick Cousot and Radhia Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94)*, pages 95–112. IEEE, 1994.

14 Patrick Cousot and Radhia Cousot. Modular static program analysis. In *International Conference on Compiler Construction*, pages 159–179. Springer, 2002. doi:10.1007/3-540-45937-5_13.

15 Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, 1978. doi:10.1145/512760.512770.

16 Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W O'Hearn. Scaling static analyses at Facebook. *Communications of the ACM*, 62(8):62–70, 2019. doi:10.1145/3338112.

17 Azadeh Farzan and Zachary Kincaid. Compositional recurrence analysis. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*, pages 57–64. IEEE, 2015. doi:10.1109/FMCAD.2015.7542253.

18 Denis Gopan, Frank DiMaio, Nurit Dor, Thomas Reps, and Mooly Sagiv. Numeric domains with summarized dimensions. In *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29-April 2, 2004. Proceedings 10*, pages 512–529. Springer, 2004. doi:10.1007/978-3-540-24730-2_38.

19 Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 339–348. ACM, 2008. doi:10.1145/1375581.1375623.

20 Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

21 Matthieu Journault. *Precise and modular static analysis by abstract interpretation for the automatic proof of program soundness and contracts inference*. PhD thesis, Sorbonne Université, 2019.

22 Matthieu Journault, Antoine Miné, Raphaël Monat, and Abdelraouf Ouadjaout. Combinations of reusable abstract domains for a multilingual static analyzer. In *Verified Software. Theories, Tools, and Experiments: 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13–14, 2019, Revised Selected Papers 11*, pages 1–18. Springer, 2020. doi:10.1007/978-3-030-41600-3_1.

23 Matthieu Journault, Antoine Miné, and Abdelraouf Ouadjaout. Modular static analysis of string manipulations in C programs. In *SAS*, volume 11002 of *Lecture Notes in Computer Science*, pages 243–262. Springer, 2018. doi:10.1007/978-3-319-99725-4_16.

24 Matthieu Journault, Antoine Miné, and Abdelraouf Ouadjaout. An abstract domain for trees with numeric relations. In *European Symposium on Programming*, pages 724–751. Springer, 2019. doi:10.1007/978-3-030-17184-1_26.

25 Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps. Compositional recurrence analysis revisited. *ACM SIGPLAN Notices*, 52(6):248–262, 2017. doi:10.1145/3062341.3062373.

26 Dexter Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(3):427–443, 1997. doi:10.1145/256167.256195.

27 Pierre Lermusiaux and Benoît Montagu. Detection of uncaught exceptions in functional programs by abstract interpretation. In *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II*, volume 14577 of *LNCS*, pages 391–420. Springer, 2024. doi:10.1007/978-3-031-57267-8_15.

28 Shuying Liang and Matthew Might. Entangled abstract domains for higher-order programs. In *Proceedings of the 2013 Workshop on Scheme and Functional Programming, Washington, DC*, 2013.

29 Anil Madhavapeddy and Yaron Minsky. *Real World OCaml: Functional Programming for the Masses*. Cambridge University Press, 2 edition, 2022.

30 Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *European Symposium on Programming*, pages 5–20. Springer, 2005. doi:10.1007/978-3-540-31987-0_2.

31 Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978. doi:10.1016/0022-0000(78)90014-4.

32 Raphaël Monat. *Static type and value analysis by abstract interpretation of Python programs with native C libraries*. PhD thesis, Sorbonne Université, 2021.

33 Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Static type analysis by abstract interpretation of Python programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, pages 17–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

34 Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Value and allocation sensitivity in static python analyses. In *SOAP@PLDI*, pages 8–13. ACM, 2020. doi:10.1145/3394451.3397205.

35 Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. A multilanguage static analysis of python programs with native C extensions. In *SAS*, volume 12913 of *Lecture Notes in Computer Science*, pages 323–345. Springer, 2021. doi:10.1007/978-3-030-88806-0_16.

36 Benoît Montagu and Thomas Jensen. Stable relations and abstract interpretation of higher-order programs. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–30, 2020. doi:10.1145/3409001.

37 Abdelraouf Ouadjaout and Antoine Miné. A library modeling language for the static analysis of C programs. In *SAS*, volume 12389 of *Lecture Notes in Computer Science*, pages 223–247. Springer, 2020. doi:10.1007/978-3-030-65474-0_11.

38 Mário Pereira and António Ravara. Cameleer: A deductive verification tool for OCaml. In *International Conference on Computer Aided Verification*, pages 677–689. Springer, 2021. doi:10.1007/978-3-030-81688-9_31.

39 Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–30, 2011. doi:10.1145/1926385.1926390.

40 Fausto Spoto. The Julia static analyzer for Java. In *Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings 23*, pages 39–57. Springer, 2016. doi:10.1007/978-3-662-53413-7_3.

41 Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. Dependent types and multi-monadic effects in f. In *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–270, 2016.

42 Milla Valnet, Raphaël Monat, and Antoine Miné. Analyse statique de valeurs par interprétation abstraite de programmes fonctionnels manipulant des types algébriques récursifs. In *JFLA 2023-34èmes Journées Francophones des Langages Applicatifs*, pages 211–242, 2023.

43 Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement types for Haskell. In *ICFP*, pages 269–282. ACM, 2014. doi:10.1145/2628136.2628161.

44 Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, pages 249–257. ACM, 1998. doi:10.1145/277650.277732.

Mono Types – First-Class Containers for Datalog

Runqing Xu
JGU Mainz, Germany

David Klopp
JGU Mainz, Germany

Sebastian Erdweg
JGU Mainz, Germany

Abstract

We propose mono types, a new abstraction for programming Datalog. Mono types behave like first-class containers that can be stored in relations and to which elements can be added decentrally. But, mono types are more than just containers: They provide a `read` operation that can yield any result as long as it monotonically grows with each added element and is independent of the order in which elements are added to the container. This design permits a wide range of mono types (e.g., sets, maps, and aggregations), yet guarantees mono types can be integrated into Datalog without jeopardizing Datalog's least fixed-point semantics. We develop a theory for mono types, which includes constructions for complex mono types, equivalence relation for mono types, and properties about semantics-preserving mono-type transformations. This theory ensures sound Datalog integration and justifies crucial compiler optimizations for mono types. Together, these techniques demonstrate that mono types provide abstraction without regret: We demonstrate in two case studies that programs become easier to write with mono types, while their performance also improves drastically.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Datalog, compiler optimization

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.33

Supplementary Material *Software (Source Code)*:
`https://gitlab.rlp.net/plmz/artifacts/mono-types-ecoop25` [18]

Funding This work is supported by the German Research Foundation (DFG) (Project numbers 451545561 & 508316729) and ERC grant AutoInc (`https://doi.org/10.3030/101125325`).

Acknowledgements We thank the anonymous reviewers for their effort and helpful suggestions.

1 Introduction

Datalog is a programming language for processing structured and cyclic data. Datalog's distinguishing feature is its efficient fixed-point semantics: A program runs until no more deductions are possible, exploiting that programs behave monotonically. Therefore, when processing graphs (as in data-flow or social-network analysis), Datalog programmers can forgo data structures for graph traversals (e.g., visited nodes, queues, stacks) and explicit termination conditions. Instead, the termination is managed by the fixed-point semantics. However, this does not mean all Datalog programs are equally efficient.

Consider a Datalog program that collects the variables shared between two syntax trees:

```
sharedVar(v) :- e1 = Plus(Plus(Var("x1"),Var("x2")),Var("y")),
                e2 = Plus(Var("y"),Plus(Var("z1"),Var("z2"))),
                hasVar(e1, v), hasVar(e2, v).
hasVar(e, v) :- ?Var(e,v).
hasVar(e, v) :- ?Plus(e,lhs,rhs), hasVar(lhs,v).
hasVar(e, v) :- ?Plus(e,lhs,rhs), hasVar(rhs,v).
```

© Runqing Xu, David Klopp, and Sebastian Erdweg;
licensed under Creative Commons License CC-BY 4.0

Predicate `sharedVar` constructs two syntax trees `e1` and `e2`, computes their variables using `hasVar`, and returns those variables `v` that occur in both expressions. Predicate `hasVar` traverses the syntax tree to collect variables. We write `?Var(e,v)` to pattern match `e` as a `Var`, extracting the variable name `v`. In the end, `hasVar` computes a relation that associates each expression with each variable contained in the expression. Thus, `hasVar` will derive 8 tuples for each of `e1` and `e2`, even though they only contain 3 variables each. In general, an expression tree of size n will contain at most $(n+1)/2$ variables, yet `hasVar` computes up to $((n+1) \cdot log\,(n+1))/2$ tuples. That is, our program performs asymptotically worse than necessary.

The problem of our program is that it propagates variables along the tree, generating a separate tuple for an intermediate node. But the problem is more general than that. For example, consider a graph analysis of a social network, where we want to find Alice's most popular friend.

```
mostPopFriend(p, max((p, n))) :- n = count(connectedWith(p,_)).
mostPopFriend(p1, max((p, n))) :- connectedWith(p1, p2), mostPopFriend(p2, (p, n)).
main(p) :- mostPopFriend("Alice", (p, _)).
```

This program uses recursive aggregation to select the person `p` with the most connections `n`. Moreover, since `connectedWith` may span a cyclic graph, the program relies on Datalog's fixed-point semantics, producing a single most popular person in the end. Again this program derives many more tuples than necessary. Initially, `mostPopFriend` contains tuples `(p, (p,n))` for each person `p` and only gradually will the most connected person become known throughout the graph.

In this paper, we propose a Datalog extension that not only eliminates this overhead but also improves the programmability of Datalog. Specifically, we introduce *first-class containers for Datalog* that enable a new programming pattern: Rather than redundantly propagating information from one node to another, we can propagate a single container to collect all information. Conceptually, our containers are mutable collections that can grow over time. Let us revisit the first example:[1]

```
sharedVar(v) :- e1 = Plus(Plus(Var("x1"),Var("x2")),Var("y")),
                e2 = Plus(Var("y"),Plus(Var("z1"),Var("z2"))),
                hasVar(e1, v), hasVar(e2, v).
hasVar(e, v) :- col = new set[string], collectVar(e,col), read(col) = v.
collectVar(e, col) :- ?Var(e,v), col += v.
collectVar(e, col) :- ?Plus(e,lhs,rhs),collectVar(lhs,col),collectVar(rhs,col).
```

Our extension comprises three new constructs. First, we can create distinct containers such as `new set[string]` in `hasVar`. Second, we can add elements to the container such as `col += v` in `collectVar`. Third, we can read from a container `read(col)`. When executing this program, only the container needs to be propagated but not the variables, leading to considerably fewer tuples. But, actually, our containers are much more flexible than this example may suggest.

Our extension supports a wide variety of "containers", which only need to adhere to a single monotonicity property for a user-defined partial order $\sqsubseteq$.

$\forall c.\, \forall i.\, \mathsf{read}(c) \sqsubseteq \mathsf{read}(\mathsf{c} \mathrel{+}= \mathsf{i})$

[1] Datalog requires range-restrictedness: Each variable in the head of a rule must be positively bound in the body of the rule. In this paper, example Datalog programs sometimes violate range-restrictedness when a relation's head variable is intended to receive input from the relation's call-sites. This is the case for head variable `col` of `collectVar`, which is bound by the call in `hasVar`. Our Datalog code can be systematically rewritten to satisfy range-restrictedness using the magic set transformation. We take the liberty of showing the "wrong" code for readability.

That is, adding elements i to a container c will only increase the subsequent reads. This property is sufficient for ensuring that containers are compatible with Datalog's fixed-point semantics. For the `set` container from above, the monotonicity property holds because `read` produces the set of added elements. But we do not need to collect elements, we may also aggregate them. For example, a sum aggregation of natural numbers satisfies the monotonicity property: `read` produces the sum, which grows with every addition. Since monotonicity is the only constraint, we call our container extension *mono types*. Mono types not only can improve performance, they also provide a new form of abstraction. For example, in our social-network program, we can add people with their friendship count to a mono type, and separately decide which mono-type definition to use:

```
collectFriend(p, col) :- col += (p, count(connectedWith(p,_))).
collectFriend (p1, col) :- connectedWith(p1, p2), collectFriend(p2, col).
mostPopFriend(p, pop) :- max = new retain_max, collectFriend(p, max), read(max) = pop.
leastPopFriend(p, pop) :- min = new retain_min, collectFriend(p, min), read(min) = pop.
```

We first demonstrate the algebraic theory of mono types. A well-founded theory of mono types is crucial because violations of monotonicity will lead to undefined Datalog behavior. In particular, we develop mono-type constructions that enable us to develop complex mono types based on simpler ones. For example, a map mono `Map[K,V]` is parametric in a mono type for values `V` and combines all values that have the same key. This way we can construct mono types for multi maps `new map[K, Set[V]](new set)` or for grouped maxima `new map[K, int](new max)`. Our theory ensures that all map constructions yield valid mono types. We have formalized the algebraic theory of mono types and verified the correctness of mono-type constructions in the proof assistant Rocq, and the code is available open-source.[1]

We then show how mono types can be integrated into Datalog systems. Technically, we compile mono types to Datalog aggregations. However, this induces a considerable overhead for mono types that are containers, such as sets and maps. Therefore, we develop the theory of mono-type equivalence and mono-type functors, allowing us to convert between equivalent mono-type definitions. We use this theory to prove certain optimizations sound, and add them to our Datalog integration. Finally, we evaluate the programmability and performance of mono types using case studies. In summary, we present the following contributions:

- We develop and mechanize the algebraic theory of mono types and mono-type constructions in Rocq.
- We integrate mono types into Datalog by translation to user-defined aggregations.
- We formalize the theory of mono-type equivalence and mono-type functors and use it to develop sound compiler optimizations for the set and map mono types.
- We show that mono types improve Datalog's programmability and evaluate their performance.

2 A Theory of Mono Types

In the introduction, we have seen some motivating examples for using mono types: to collect values and to aggregate them. Since mono types can occur as part of Datalog's fixed-point semantics, it is important to ensure Datalog remains well-founded. To this end, we develop and formalize a specification for mono types and instantiate the specification with various examples. We then present generic constructions for mono types, allowing us to build up complex mono types from simpler ones. We have mechanized the theory of mono types and the constructions in Rocq and use Rocq's notation in our presentation.

[1] `https://gitlab.rlp.net/plmz/artifacts/mono-types-ecoop25`

2.1 Mono Types

Conceptually, mono types are stateful containers. However, mono-type containers are restricted in three important ways: First, elements can only be added to a mono type but never removed or replaced. Second, the content of a mono type can only be observed monotonically. And third, the observation of a mono type is independent with the order of added elements. We represent the interface of mono types as a type-level record in Rocq:

```
Record MonoAPI M I O `{PartialOrder O} : Type :=
{ create : M I O ;  add : M I O → I → M I O ;  read : M I O → O
; mono_law : forall m i, read m ⊑ read (add m i)
; ord_indep : forall m i1 i2, read (add (add m i1) i2) = read (add (add m i2) i1)
}.
```

Mono instances have type `M I O`, where `I` is the type of added inputs, and `O` is the type of the read observations. Mono instances can only be created through functions `create` and `add`, which means all mono instances are sequences:

$$\mathtt{m}_0 = \mathtt{create}, \qquad \mathtt{m}_1 = \mathtt{add}\ \mathtt{m}_0\ \mathtt{i}_1, \qquad \cdots \qquad \mathtt{m}_n = \mathtt{add}\ \mathtt{m}_{n-1}\ \mathtt{i}_n$$

Users can observe the state of a mono instance at any time using `read`, but with two restrictions:

(i) Reads on a later mono instance produce larger values than reads on an earlier mono instance from the same sequence:

$$\forall\, i, j.\quad i < j \implies \mathtt{read}\ \mathtt{m}_i \sqsubseteq \mathtt{read}\ \mathtt{m}_j$$

In the interface of mono types above, we ensure this property by requiring a monotonicity law `mono_law`. This law states that each individual `add` increases the observable output with respect to a fixed partial order for `O`.

(ii) Swapping the order of inputs does not affect the updated value of a mono instance:

$$\forall\, i_1, i_2.\quad \mathtt{read}\ (\mathtt{add}\ (\mathtt{add}\ \mathtt{m}_j\ i_1)\ i_2) = \mathtt{read}\ (\mathtt{add}\ (\mathtt{add}\ \mathtt{m}_j\ i_2)\ i_1)$$

We ensure the order independence through the law `ord_indep` in the interface.

The monotonicity law, order independence law, and the lack of any removal operations are the reason that mono types integrate well with Datalog as we will discuss below.

One notable omission from `MonoAPI` is the internal state of mono instances. While it is clear that mono types must use internal state to carry information from an `add` to a `read`, the internal state cannot be observed by users directly. Instead, the internal state is only accessible while defining a mono type, for which we provide a dedicated interface:

```
Record Mono ST I O `{PartialOrder O} : Type :=
{ init : ST ; add : ST → I → ST ; read : ST → O
; mono_law : forall (st : ST) (i : I), read st ⊑ read (add st i)
; ord_indep : forall s a1 a2, result (add (add s a1) a2) = result (add (add s a2) a1)
}.
```

In contrast to `MonoAPI`, interface `Mono` has an additional type parameter `ST`, which exposes the type of the internal state. Otherwise, the two interfaces are almost identical and each `Mono` gives rise to an implementation of `MonoAPI` by setting `M I O = ST` and using `init` for `create`. All our examples and constructions use the `Mono` interface, but it is important to remember that the internal state will not be visible to users of `MonoAPI`, which means we can exchange the implementation internals of a mono type without affecting users.

As first example, consider a mono type `nat_sum_mono` that computes the sum over naturals:

```
Definition nat_sum_mono : Mono nat nat nat :=
  {| init := 0 ; add s n := s + n ; read s := s |}.
```

For `nat_sum`, the internal state, input, and output all are of type `nat`. We start with an initial state `0`, add each new input to it, and allow the sum to be observed directly. This mono type satisfies the `mono_law` for the natural order of `nat`.

Our second example is `retain_max_mono`, which we used in Section 1 for the most popular person in a social network:

```
Definition retain_max_mono A : Mono (option (A, nat)) (A, nat) (option (A, nat)) :=
{| init := None
 ; add s i := match s, i with
   | None, _ ⇒ Some i
   | Some (a1,n1), (a2,n2) ⇒ if n1 >? n2 then Some (a1,n1) else Some (a2,n2)
   end
 ; read s = s
|}.
```

This mono type is more interesting and requires distinct types for input and output. The input is a pair `(A,nat)`, which can be understood as a weighted `A`. The mono type retains that input `A` with the highest weight. However, initially no `A` value is known yet, which is why we employ an option type for the internal state and output. Mono type `retain_max` satisfies the `mono_law` for the ordering where `None` $\sqsubseteq$ `Some` (a, n) for all a and n, and `Some` $(a_1, n_1) \sqsubseteq$ `Some` (a_2, n_2) if $n_1 \leq n_2$.

In both previous examples, the `read` function was a simple identity. We include `read` in our design because it allows us to isolate a post-processing step. For example, we can define a set mono that collects and produces a set of all inputs. However, internally this mono type uses a list representation for constant-time `add` and only converts the list to a set in `read`:

```
Definition set_mono1 A : Mono (list A) A (set A) :=
{| init := nil ; add s a := a :: s ; read s := set_from_list s |}.
```

The internal state has no obvious ordering, but the output set satisfies the `mono_law` based on the partial order $(\subseteq)$ of set inclusion. Technically, it is not necessary to isolate the `read` projection, but it makes mono types easier to define and reason about. In principle, we can merge the post-processing done by `read` into the other functions if we use the internal state to also retain the post-processed output:

```
Definition set_mono2 A : Mono (list A, set A) A (set A) :=
{| init := (nil, set_from_list nil)
 ; add st a := let (l,_) := st in (a :: l, set_from_list (a :: l))
 ; read st := let (_,s) := st in s
|}.
```

Here, the internal state contains both the list of inputs and the projected set representation. We update the projected set after each `add`, but only use it in `read`. For this particular mono type, we can actually improve the definition by fusing `add` and `set_from_list`:

```
Definition set_mono A : Mono (set A) A (set A) :=
  {| init := empty_set ; add s a := set_add a s ; read s := s |}.
```

This is our final mono type for sets (*set mono* for short). We start with an empty set and add input elements to that set directly. While we could have started with this mono type, note this: All three mono types implement the same `MonoAPI`. Semantically, the three mono types are interchangeable because they only differ in their internal state, which is not observable. We will exploit such interchangeability in Section 4 to optimize usages of mono types.

The set mono is actually a special case of an entire class of mono types. A bounded semi-lattice has a bottom element and a least upper bound operator `lub`. Each bounded semi-lattice uniquely determines a partial order:

$$\mathtt{a}_1 \sqsubseteq \mathtt{a}_2 \iff \mathtt{lub}\ \mathtt{a}_1\ \mathtt{a}_2 = \mathtt{a}_2$$

Each bounded semi-lattice gives rise to mono type, which satisfies the `mono_law`:

```
Definition lattice_mono A `(BoundedSemiLattice A) : Mono A A A :=
  {| init := bottom ; add := lub ; read a := a |}.
```

This is an important class of mono types because lattices are heavily exercised in Datalog. In particular, lattice-based aggregation [8, 13] allows Datalog programs to implement data-flow analyses over custom lattices. The lattice mono can also easily accomodate widening, for example, using a unary widening operator in `read`.

Lastly, we want to provide an example where the internal state of a mono type is not monotone, but the output is. Consider the following mono type where the internal state changes in accordance with the collatz function:

```
Definition collatz_fun n : {n : nat | 0 < n} := if is_even n then n / 2 else 3 * n + 1.
Definition collatz_mono A (n : {n : nat | 0 < n}) : Mono nat A bool :=
{| init := n ; add st _ := if st =? 1 then 1 else collatz_fun st ; read st := st =? 1 |}.
```

The `collatz_mono` starts at a given positive number `n`. At each call of `add`, we forward the state to the next collatz number, unless we already reached `1`. The output of the `collatz_mono` is `true` if the number of calls to `add` is equal to or exceeds the total stopping time of `n`, that is, if the sequence has reached `1` yet. For example, `collatz_mono 12` will exercise the following sequence for the first 9 calls of `add`: $12, 6, 3, 10, 5, 16, 8, 4, 2, 1$. Any subsequent call of `add` will keep the state at 1. Nonetheless, `add` is not monotone: the state fluctuates up and down indeterminately. However, the output of mono type satisfies `mono_law` for the order `false` $\sqsubseteq$ `true`.

2.2 Mono-Type Constructions

We have seen a number of mono types in the previous subsection. However, it is not necessary to construct mono types individually, including the necessary proof of `mono_law` . Instead, users can use generic constructions that we discovered for creating complex mono types from simpler ones. We start with constructions that compose multiple mono types.

Sum mono. In Section 1, we used a mono type to collect all variables in an expression tree. Consider we also want to collect the integer literals the tree contains. Of course, we could traverse the tree twice, once collecting variables and once collecting literals, but this entails a memory and run-time overhead. It is more efficient to use a single mono type that can collect both kinds of data. We can construct such a mono type using the sum construction:[2]

```
Definition sum_mono (m1 : Mono ST1 I1 O1) (m2 : Mono ST2 I2 O2)
  : Mono (ST1 * ST2) (I1 + I2) (O1 * O2) :=
{| init := (init m1, init m2)
 ; add s a := let (s1, s2) := s in match a with
              | inl a1  ⇒  (add m1 s1 a1, s2)
              | inr a2  ⇒  (s1, add m2 s2 a2)
              end
 ; read s := let (s1, s2) := s in (read m1 s1, read m2 s2)
|}.
```

A sum mono is parametric in mono types `m1` and `m2`, and it retains both of their internal states `ST1 * ST2`. The sum mono's input is the sum type `I1 + I2` and the `add` function dispatches to either `m1` and `m2`. We can use the sum mono in Datalog as follows:

[2] Note the record accessors in Rocq: `init m1`, `add m1`, and `read m1` refer to the respective fields of the mono-type record `m1`.

```
main(e) :- col = new sum(new set[string], new set[int]), collectInfo(e, col).
collectInfo(e,col) :- ?Var(e,v), col += (inl v).
collectInfo(e,col) :- ?Num(e,n), col += (inr n).
collectInfo(e,col) :- ?Plus(e,lhs,rhs),collectInfo(lhs,col),collectInfo(rhs,col).
```

Product mono. While the sum mono dispatches its input, a product mono obtains inputs for both contained mono types. For example, in a social network, we may want to find the most popular person while also collecting all people in a set. The product mono `new prod(new retain_max, new set[Person])` can deliver this behavior:

```
Definition prod_mono (m1 : Mono ST1 I1 O1) (m2 : Mono ST2 I2 O2)
  : Mono (ST1 * ST2) (I1 * I2) (O1 * O2) :=
{| init := (init m1, init m2)
 ; add s a := let (s1, s2) := s in let (a1, a2) := a in (add m1 s1 a1, add m2 s2 a2)
 ; read s := let (s1, s2) := s in (read m1 s1, read m2 s2)
|}.
```

Note how each input pair `(a1,a2)` is split up and forwarded to `m1` and `m2` in `add`. The definitions `init` and `read` are identical to the sum mono.

Input functors. We can also construct various mono types through functors. The simplest class of functors only affects the input of a mono type:

```
Definition fmap_input_mono (f : I2 → I1) (m : Mono ST I1 O) : Mono ST I2 O :=
  {| init := init m ; add s a := add m s (f a) ; read s := read m s |}.
```

This functor wraps a mono type and applies `f` to all inputs prior to processing. For example, we can use the functor to define mono types that fork their input:

```
Definition fork_mono (m1: Mono ST1 I O1) (m2: Mono ST2 I O2): Mono (ST1 * ST2) I (O1 * O2)
  := fmap_input_mono (fun a ⇒ (a,a)) (prod_mono m1 m2).
```

The fork mono is like a product mono, but both contained mono types expect the same input type `I`. We can use an input functor to duplicate a given input and use the standard product mono to do the composition.

Map mono. Finite maps are used ubiquitously in computing, and program analysis applications in Datalog are no exception. For example, even a flow-insensitive analysis usually tracks abstract values for each variable `map[string,V]`, where `V` forms a lattice and multiple entries for the same variable are joined. A flow-sensitive analysis computes such a map for each control-flow node `map[Node,map[string,V]]`. To support this important class of Datalog applications, we developed a construction for map monos:

A map mono accepts input pairs `K * V`, where `K` is a key and `V` is a value. However, it does not store the values for keys directly. Instead, all values associated with the same key are added to a mono type `m` with input `V`. The map mono then stores the state of `m` for each key:

```
Definition map_mono `{Bounded O} (m: Mono ST V O): Mono (map K ST) (K * V) (map K O) :=
{| init := empty_map
 ; add s ka := let (k, a) := ka in
               map_add_with s k (fun st ⇒ add m st a) (add m (init m) a)
 ; read s := map_map_values (read m) s
|}.
Definition map_add_with {K V} : map K V → K → (V → V) → V → map K V.
Definition map_map_values {K V1 V2} : (V1 → V2) → map K V1 → map K V2.
```

$$
\begin{array}{rrll}
p & \in & \mathit{Prog} & ::= \overline{r} \\
r & \in & \mathit{Rule} & ::= P(\overline{t}) \coloneq \overline{a}. \mid P(\overline{t}, \alpha(\overline{t})) \coloneq \overline{a}. \\
a & \in & \mathit{Atom} & ::= P(\overline{t}) \mid t = t \mid t \mathrel{+}= t@\overline{t} \\
t & \in & \mathit{Term} & ::= c \mid x \mid \textbf{new}\ M(\overline{m})@\overline{T} \mid \textbf{read}(t) \\
m & \in & \mathit{MonoTerm} & ::= \textbf{new}\ M(\overline{m}) \mid c
\end{array}
$$

$c \in \textbf{Val} \quad x \in \textbf{Var} \quad M \in \textbf{MonoType} \quad \alpha \in \textbf{AggOp}$

Figure 1 Datalog syntax extended with constructs for creating, adding to, and reading from a mono type.

Initially, the state of a map mono is the empty map; we initialize the state for new keys on demand in `add`. To this end, `add` uses a helper function `map_add_with s k f v` that adds key `k` to map `s`. If the key already exists in `s`, `map_add_with` uses `f` to update the old value using `add` of `m`. But if the key is new `s`, `map_add_with` inserts its value `v`. This is where `add` lazily initializes the entry for `k` with `init m` before adding `a` to it. The result of the map mono is another map, which contains the outputs of `m` for each key in the map.

To prove that `map_mono` produces valid mono types that satisfy the `mono_law`, we need to introduce a partial order on maps. The constraint `Bounded V` means `V` has a partial order and a bottom element.

```
Instance map_order {K V} `{Bounded V}: PartialOrder (map K V) :=
  { s1 ⊑ s2 := forall k, map_find_default k bottom s1 ⊑ map_find_default k bottom s2 }.
```

Function `map_find_default k v s` yields the value of `k` in `s`, or `v` if `s` does not contain `k`. This means we compare two maps by treating all undefined keys as being assigned to bottom. Thus, `s1 ⊑ s2` holds if for all keys `k` in `s1`, either `s1(k) = bottom` or `s2` also contains `k` and `s1(k) ⊑ s2(k)`. Based on this order, we prove `mono_law` for `map_mono` by induction on its internal state. We can now use the map mono to construct various useful mono types, such as:

```
Definition multi_map_mono K V := map_mono K (set_mono V).
Definition var_sign_mono := map_mono string (lattice_mono sign).
Definition stmt_var_interval_mono :=
  map_mono stmt (map_mono string (lattice_mono interval)).
```

The `var_sign_mono` can be used for a flow-insensitive sign analysis, tracking the sign of each variable. The `stmt_var_interval_mono` nests two map monos to track the range of each variable for each statement separately. These mono types can improve the programmability of Datalog.

3 Integrating Mono Types into Datalog

The theory from the previous section establishes that reading from a mono type yields monotonically growing observations about its state. In the present section, we explain how this property enables us to integrate mono types into Datalog. Syntactically, the integration is straightforward as shown in Figure 1 (you can ignore the parts in blue for now). This follows our examples from the paper: We add an atom $t \mathrel{+}= t$ to add a value to a mono type, **new** M to create a new mono instance, and **read**(t) to read the result of a mono type. The present section explains how these constructs can be integrated into Datalog semantically.

Our strategy for integrating mono types into Datalog is to treat them as a generalized form of aggregation. In Datalog, aggregation is often written as an annotation on the rule head:

```
R(x, α(y)) :- P(x, y).
P("a",1). P("a",2). P("b",1). P("b",3). P("c",10).
```

Here, α is an aggregation operator, `α(y)` is the aggregation results, and all bindings for `y` in the body of `R` are the aggregation inputs. Note that we obtain a separate aggregation result for each `x` in `R(x,α(y))`. For example, if α implements a sum aggregation, we obtain:

```
R("a",3). R("b",4). R("c",10).
```

This form of aggregation is prevalent in existing Datalog systems and we want to reuse it for mono types. However, mono types are first-class containers, and aggregation happens within each container. To embed mono types in Datalog, we have to solve 3 conceptual challenges first:

- The above example uses strings to distinguish aggregation results. Mono instances also need to be distinguishable, so that each one can provide their own aggregation result.
- The above example collects all aggregation inputs in a relation `P`. Mono instances also need to collect their inputs, even though they are provided through scattered calls of `add`.
- The above example uses a simple aggregation operator $\alpha : 2^{\texttt{int}} \to \texttt{Y}$ that takes a set of input elements of type `int` and yields a value of some type `Y`. Mono types use `add` and `read` functions to produce aggregation results, which does not fit the signature of α.

In the remainder of this section, we describe these challenges in detail and propose concrete solutions. In the end, we present translation rules that compile mono types to Datalog and give a declarative semantics for mono types.

3.1 Representing Mono Instances

Each instance of a mono type represents a distinct container. Adding an element to one mono instance may not affect other mono instances. That is, mono instances need to be identifiable and distinguishable. Consider the following example, reproduced from Section 1:

```
sharedVar(v) :- e1 = Plus(Plus(x1,x2),y), e2 = Plus(y,Plus(z1,z2)), hasVar(e1,v), hasVar(
    e2,v).
hasVar(e, v) :- m = new set[int], collectVar(e,m), read(m) = v.
collectVar(e, m) :- ?Var(e,v), m += v.
collectVar(e, m) :- ?Plus(e,lhs,rhs), collectVar(lhs,m), collectVar(rhs,m).
```

In our example, `sharedVar` invokes `hasVar` twice, once for `e1` and once for `e2`. Accordingly, `hasVar` must create two instances with `new set[int]`, one for each expression. In contrast to object-oriented programming languages, Datalog does not have side-effects, which is necessary for creating unique object IDs. We need a different approach.

Our solution is based on an observation: Predicate `hasVar` creates a distinct mono instance for each expression `e` provided as `input`. By default, the atoms that occur before `new set[int]` in the body of `hasVar` determine how many mono instances we need: one for each unique binding of variable `e`. In general, if a rule $P(\overline{t}) :- a_1, \ldots, a_n.$ has an atom a_i that contains **new** M, then we need a distinct instance of M for each substitution produced by $a_1, \ldots, a_{i-1}$. For each substitution $\sigma = \{X_1 \mapsto c_1, \ldots, X_k \mapsto c_k\}$ produced by $a_1, \ldots, a_{i-1}$, we uniquely represent the associated mono instance as a tuple $(c_1, \ldots, c_n)$. We also support a more explicit form **for** $\overline{t}$: **new** $M(\overline{m})$, which lists relevant variables explicitly and does not rely on the order of atoms in the rule. Note that nested mono constructors such as the two sets in `new sum(new set[string], new set[int])` are arguments to the surrounding `sum` mono; they are not values in Datalog and do not require a Datalog representation.

3.2 Collecting Mono-Type Inputs

Datalog aggregation is defined over relations: A relation collects all relevant aggregands to be processed by an aggregation operator. In contrast, the aggregands of mono types are scattered throughout the program: the invocations of `+=` determine the relevant aggregands, but there is no central relation that collects them. For example, consider again the following program from Section 2, where we add values to a mono type `m` in two different places:

```
main(e) :- m = new sum(new set[string], new set[int]), collectInfo(e, m).
collectInfo(e,m) :- ?Var(e,v), m += (inl v).
collectInfo(e,m) :- ?Num(e,n), m += (inr n).
collectInfo(e,m) :- ?Plus(e,lhs,rhs),collectInfo(lhs,m),collectInfo(rhs,m).
```

To enable aggregation for mono types, we must collect the aggregands in a relation. We found that we can use a magic set transformation to achieve this. A magic set transformation rewrites the Datalog program to collect the relevant arguments of a relation [16]. If we treat `m += a` as a call to a synthesized relation `Aggregand(m, a)`, we can use a magic set transformation to obtain the following rules:

```
Aggregand(m, a) :- ?Var(e,v), a = inl v.
Aggregand(m, a) :- ?Num(e,n), a = inr n.
```

These rules are derived from the previous program by copying the relevant atoms for each `m += a`. This way, the `Aggregand` relation enumerates the added aggregand values `a` for each mono instance `m`.

By collecting all added values in a relation, we implicitly submit to Datalog's set semantics. Specifically, even when executing `m += a` multiple times, relation `Aggregand` will only enumerate `(m,a)` $\in$ `Aggregand` at most once; multiple occurrences of the same tuple are collapsed. While this is fine for the set monos used in the example above, other mono types are less permissive. For example, consider we want to infer how often each variable occurs in an expression:

```
main(e) :- m = new map[string,nat](new nat_sum), collectInfo(e, m).
collectInfo(e,m) :- ?Var(e,v), m += (v,1).
collectInfo(e,m) :- ?Plus(e,lhs,rhs),collectInfo(lhs,m),collectInfo(rhs,m).
```

Even if an expression contains the same variable or numeric literal multiple times $x + y + x$, the `Aggregand` relation will only contain two tuples: `(m, ("x",1))` and `(m,("y", 1))`. That is, the second insertion of `(m, ("x",1))` does not show up, because Datalog employs its set semantics for the relation. As a consequence, we would incorrectly compute that both `"x"` and `"y"` only occur once in the expression.

This issue is well-known for aggregations in Datalog, and we adapt the standard solution [9]. To distinguish different occurrences of aggregands, we introduce auxiliary columns that help distinguish them. Figure 1 showed the necessary syntactic constructs in blue font. For our example, we can use the IDs of expressions for this purpose:

```
main(e) :- m = new map[string,nat](new nat_sum)@Exp, collectInfo(e, m).
collectInfo(e,m) :- ?Var(e,v), m += (v,1)@e.
collectInfo(e,m) :- ?Plus(e,lhs,rhs),collectInfo(lhs,m),collectInfo(rhs,m).
Aggregand(m, kv, id) :- ?Var(e,v), kv = (v,1), id = e.id.
```

Note how we augmented the mono construction with the type and the `add` call with the value of the auxiliary columns. For $x + y + x$ we obtain the following aggregands, assuming each variable has a distinct ID `Var#i`.

```
Aggregand(m, ("x",1), Var#1). Aggregand(m, ("y",1), Var#2). Aggregand(m, ("x",1), Var#3).
```

Importantly, aggregation will ignore the auxiliary columns. It will query `Aggregand(m, kv, _)` and call `add` for each individual entry. The next subsection explains how aggregation works in detail.

3.3 Mono Types as Generalized Recursive Aggregation

Conceptually, Datalog's aggregation operators are defined over sets of inputs and have the signature $\alpha : 2^{\mathtt{X}} \to \mathtt{Y}$. That is, given a set of inputs of type X, aggregation yields a single output of type Y. In practice, Datalog systems compute aggregation results iteratively and require aggregation operators of the following form: $\beta : \mathtt{Y} \times (\mathtt{X} \to \mathtt{Y} \to \mathtt{Y})$. An aggregation operator β defines an initial result β_1 that corresponds to $\alpha(\emptyset)$ and an update function β_2 that incorporates another input. Then, for $X = \{x_1, \ldots, x_n\}$, we have $\alpha(X) = \beta_2(x_n, (\ldots, \beta_2(x_1, \beta_1)))$.

As a first try, we could integrate mono types as iterative aggregation operators $\beta = (\mathtt{init}, \mathtt{add})$. The `read` function is then only applied after obtaining the mono-type state through aggregation. This leads to code of the following form, where `x` are mono-type inputs, `y` are mono-type states, and `z` are mono-type outputs:

```
Aggregand(m, v) :- ... .
Aggregate(m, β(x)) :- Aggregand(m, x).
P(z) :- Aggregate(m,y), read(y) = z.
```

But this approach faces a semantic issue: it does not work for recursive aggregation.

If recursive aggregation occurs the Datalog program (indirectly) feeds the output y of an aggregation back as input x. Many interesting use cases of aggregation require recursive aggregation, including program analysis and network analysis. For example, consider again the social-network analysis from Section 1, which finds the most popular friend:

```
mostPopFriend(p, max((p, n))) :- n = count(connectedWith(p,_)).
mostPopFriend(p1, max((p, n))) :- connectedWith(p1, p2), mostPopFriend(p2, (p, n)).
main(p) :- mostPopFriend("Alice", (p, _)).
```

This program uses recursive aggregation, because the second rule obtains an aggregation output `(p,n)` of `p2` and uses it as aggregation input for `p1`. Recursive aggregation is only well-defined if the aggregation operator is monotone: $\forall X.\, \forall Y.\, \alpha(X) \sqsubseteq \alpha(X \cup Y)$ for traditional aggregation, or $\forall x.\, \forall y.\, y \sqsubseteq \beta_2(x, y)$ for iterative aggregation.[3] Monotonicity of aggregation is important because new aggregation results *replace* prior aggregation results (in contrast to regular columns, which accumulate all derived values in a set). But our mono types do not guarantee monotonicity for the states aggregated by `add`; we only ensure monotonicity for the resulting outputs. This renders the above approach incorrect because the second column of `Aggregate(m,β(x))` does not grow monotonically. Consequently, we lose the property that every Datalog program has a minimal model, and a fixed-point does not need to exist.

We can fix the situation by considering mono types as generalized aggregation operators

$$\gamma : \mathtt{Y} \times (\mathtt{X} \to \mathtt{Y} \to \mathtt{Y}) \times (\mathtt{Y} \to \mathtt{Z}).$$

This directly corresponds to the interface of mono types, with $\gamma = (\mathtt{init}, \mathtt{add}, \mathtt{read})$ and extends iterative aggregation by incorporating the output projection `read`. While this may seem negligible, it significantly alters the semantics of Datalog programs using such aggregation:

[3] Note that the initial result β_1 does not need to be a least element because Datalog yields *no result* when there are no aggregands. Internally, Datalog embeds aggregations results in `option` Y and yields `None` initially, but `Some` y subsequently.

```
Aggregand(m, v) :- ... .
Aggregate(m, γ(x)) :- Aggregand(m, x).
P(z) :- Aggregate(m,z).
```

Compared to the first approach, `Aggregate` now contains output values `z` rather than internal states `y`. And since mono types guarantee monotonicity of the projected outputs through the `mono_law`, we can use them as a generalized form of aggregation. For example, this generalized aggregation is necessary to correctly embed the `collatz_mono` into Datalog, because its output is monotone even though its state fluctuates. We have used generalized aggregations in an existing Datalog system (see Section 5) and it was easy to replace β-aggregation by γ-aggregation.

3.4 Formal Translation and Mono Dispatch

The previous subsections explained the ideas for encoding mono types in Datalog. To make these encodings precise and reproducible, we provide a formal translation from Datalog with mono types to Datalog with aggregation. Figure 2 shows the translation rules, which we explain one by one subsequently. The reason we define the semantics of mono types through translation to recursive aggregation is that the semantics of recursive aggregation is well understood [9] and supported by modern Datalog engines.

A program consists of rules $r_1 \cdots r_n$, which we translate compositionally $[\![r_1]\!] \cdots [\![r_n]\!]$. However, to support mono types, we must also generate additional rules for each mono construction **new** $M(\overline{m})@\overline{U}$ in the program. Note that $\overline{U}$ are the types of the auxiliary columns used to distinguish different occurrences of added inputs as described in Subsection 3.2. For each mono construction, we derive three new rules:

- Aggregand: Collects all aggregands of a mono type, added through invocations of $m \mathrel{+}= t$. As explained in Subsection 3.2, we rely on a magic-set transformation to synthesize the rules for this predicate; our translation only indicates the head of these rules.
- $\mathsf{Aggregate}_{M(\overline{m})}$: Performs the aggregation for the fully instantiated mono type $M(\overline{m})$ in the head of the rule. The body of the rule draws the aggregation inputs from Aggregand, ignoring the auxiliary columns.
- Aggregate: Since different mono types can occur in the same program, we need to carefully select the correct aggregation operation (an issue we have ignored so far). To this end, we use Aggregate to *dispatch* over the current mono type. Specifically, if m is a mono instance of $M(\overline{m})$, we dispatch to $\mathsf{Aggregate}_{M(\overline{m})}$. Note that these dispatch alternatives must be generated statically to identify the predicate named $\mathsf{Aggregate}_{M(\overline{m})}$.

The rest of the translation is compositional; Figure 2 only shows the rules for rewritten atoms. First, consider the construction of a new mono instance of $M(\overline{m})$. We represent mono instances as algebraic data in Datalog: $m = !\mathsf{MonoVal}(M(\overline{m}), \overline{t})$ constructs a mono instance and $?\mathsf{MonoVal}(m, M(\overline{m}), \overline{t})$ deconstructs m again. Each mono instance contains its fully instantiated mono type $M(\overline{m})$ and a list of values computed by $\overline{t}$ that uniquely identifies each mono instance as explained in Subsection 3.1.

An add operation $m \mathrel{+}= t@\overline{u}$ translates to a call of relation Aggregand. The magic set transformation will use this call to generate a rule for Aggregand that enumerates the arguments of the call. Here, the terms $\overline{u}$ serve to distinguish different occurrences of t. Finally, we read the aggregation result of a mono instance by calling relation Aggregate.

This concludes the integration of mono types into Datalog. In the translated program, the aggregation operator $M(\overline{m})$ is both order-independent and monotonic under bag inclusion, based on the algebraic properties of mono types. Thus, our integration does not jeopardize

$$
\begin{array}{lcl}
[\![r_1 \cdots r_n]\!]_{prog} & = & [\![r_1]\!]_{rule} \cdots [\![r_n]\!]_{rule} \cdot [\![\textbf{new } M(\overline{m})@\overline{T} \in (r_1 \cdots r_n)]\!]_{agg} \\
[\![\textbf{new } M(\overline{m})@(U_1, \ldots, U_k)]\!]_{agg} & = & \mathsf{Aggregand}(m, t, (u_1, \ldots, u_k)) :- \ \textit{by magic set trans.} \\
 & & \mathsf{Aggregate}_{M(\overline{m})}(m, M(\overline{m})(t)) :- \mathsf{Aggregand}(m, t, _). \\
 & & \mathsf{Aggregate}(m, t) :- \ ?\mathsf{MonoVal}(m, M(\overline{m}), _), \\
 & & \qquad\qquad \mathsf{Aggregate}_{M(\overline{m})}(m, t). \\
[\![a_0 :- a_1, \ldots, a_n]\!]_{rule} & = & a_0 :- [\![a_1]\!]_{atom}, \ldots, [\![a_n]\!]_{atom} \\
[\![m = \textbf{for } \overline{t} : \textbf{new } M(\overline{m})@\overline{U}]\!]_{atom} & = & m = !\mathsf{MonoVal}(M(\overline{m}), \overline{t}) \\
[\![m \mathrel{+}= t@\overline{u}]\!]_{atom} & = & \mathsf{Aggregand}(m, t, \overline{u}) \\
[\![x = \textbf{read}(m)]\!]_{atom} & = & \mathsf{Aggregate}(m, x)
\end{array}
$$

Figure 2 Formal translation of Datalog with mono types to Datalog with generalized aggregation.

the least-fixed point semantics of Datalog with recursive aggreation [13], as long as there is no infinite ascending chain in the aggregation domain. We can use our translation to execute Datalog programs with mono types in any Datalog engine that supports generalized aggregation, such as Viatra/IncA [17], Ascent [10], and Flix [8].

3.5 An Example of Mono-type Translation

Even with above translation rules, understanding the execution of Datalog programs with mono types can be challenging, especially for readers who are unfamiliar with recursive aggregation. To aid in comprehension, we present a concrete example of the translation and the execution of a Datalog program with mono types.

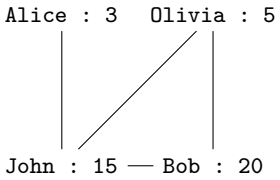
Consider a program that determines the number of followers for the most-followed user (we call it *maximum follower count* in the follows) in Alice's social network. This program uses a map mono, which associates each user's name with a `nat_max` mono, to track the maximum follower count for each individual.

```
visit(m) :- followers(p, n), m += (p,n).
visit(m) :- connected(p, q), visit(q,m), map = read(m), m += (p,map(q)).
main(f) :- m = new map[string, nat](new nat_max), visit(m), f = read(m)("Alice").
```

Here is the translation of above program. For simplicity, we use ... to leave the body generated by magic set transformation, and inline the $\mathsf{Aggregate}_{M(\overline{m})}$ rule as there is only one mono-type instance created.

```
visit(m) :- followers(p, n), Aggregand(m,p,n).
visit(m) :- connected(p, q), visit(q,m), Aggregate(m,map), Aggregand(m,p,map(q)).
main(f) :- m = !MonoVal(map_mono(nat_max)), visit(m), Aggregate(m,d), f = d("Alice").
Aggregand(m,p,n) :- ...
Aggregate(m,map_mono(nat_max)(p,n)) :- Aggregand(m,p,n).
```

We illustrate the execution of the translated program using a simple example in Figure 3. Figure 3a depicts a sample social network that includes Alice, where `Alice : 3` indicates that Alice has three followers. We detail the execution trace for both the `Aggregand` and `Aggregate` rules in Figure 3b. The aggregated argument of the `Aggregate` relation is a map that associates each user with the maximum follower count in her network. In the second iteration, John's maximum user count is updated by 20, which triggers the tuple `Aggregand(m, "Alice", 20)` derived in the third iteration. Besides, we can observe that the number of followers for each user in the map mono grows monotonically. We do not show the fourth iteration because no



(a) An instance of social-network.

	Aggregand(m,p,n)	Aggregate(m, d)
Iter. 1	(m, "Alice", 3), (m, "John", 15)*) (m, "Olivia", 5), (m, "Bob", 20)	(m, Map("Alice" -> 3, "John" -> 15, "Olivia" -> 5, "Bob" -> 20))
Iter. 2	(m, "Alice", 15), (m, "John", 20) (m, "John", 5), (m, "John", 3) (m, "Olivia", 15), (m, "Olivia", 20) (m, "Bob", 5), (m, "Bob", 15)	(m, Map("Alice" -> 15, "John" -> 20, "Olivia" -> 20, "Bob" -> 20))
Iter. 3	(m, "Alice", 20)	(m, Map("Alice" -> 20, "John" -> 20, "Olivia" -> 20, "Bob" -> 20))

*) We use `m` to represent the mono-type value `map_mono(nat_max)` for brevity.

(b) The execution trace of the translated program.

Figure 3 A concrete example of the execution of Datalog programs with mono types.

new tuples are derived and the program terminates. This example shows that performing mono add after mono read will lead to recursive aggregation in the translated program, but mono types abstract from the complications this involves.

4 Mono-Type Equivalence and Optimization

Mono-type aggregation can have a considerable performance overhead in Datalog systems because we are leaving the realm of Datalog's highly optimized relational algebra. In this section, we develop a theory for the equivalence of mono types that allows us to replace mono types without changing the semantics of a Datalog program. We then explain how this theory establishes optimizations for mono types and how they can reduce the need for aggregation in important cases.

4.1 Mono-type Equivalence

Intuitively, we consider two mono types to be equivalent if they produce the same outputs given the same sequence of inputs. There is no need to compare the states of mono types since they are invisible in the Datalog programs. For example, we introduced different version of the set mono in Section 2 (shown below): `set_mono1` uses a list internally and converts it to a set on `read`, while `set_mono` maintains a set of added elements internally and yields that set on `read`. These two mono types are equivalent because they produce the same sets as output, independent of states. However, sometimes we want to compare mono types that have different output types. For example, we would expect a multimap mono `map_mono K (set_mono V)` is equivalent to set of pairs `set_mono (K * V)`, because their output types `map K (set V)` and `set (K * V)` are isomorphic. Accordingly, we formalize the equivalence on mono types through a bidirectional embedding. A function f embeds mono type m_1 into m_2 if (i) f maps the initial output of m_1 to the initial output of m_2 and (ii) f is closed under `add`. That is, instead of running m_2, we can run m_1 and embed its outputs to obtain results equivalent to m_2. Two mono types then are *equivalent* if there exist embeddings in both directions. We formalized mono-type equivalence in `mono_equiv` below and proved it is a reflexive, symmetric, and transitive relation in Rocq:

```
Definition reachable (m : Mono ST I O) (st : ST) :=
  exists (l : list I), List.fold_left (add m) l (init m) = st.
Definition mono_embed (m1 : Mono ST1 I O1) (m2 : Mono ST2 I O2) (f : O1 → O2) : Prop :=
  f (read m1 (init m1)) == read m2 (init m2) ∧
  forall (s1 : ST1) (i : I), reachable m1 s1 → exists (s2 : ST2),
    reachable m2 s2 ∧ f (read m1 s1) == read m2 s2 ∧
    f (read m1 (add m1 s1 i)) == read m2 (add m2 s2 i).
Definition mono_equiv (m1 : Mono ST1 I O1) (m2 : Mono ST2 I O2)
    (f : O1 → O2) (g : O2 → O1) : Prop := mono_embed m1 m2 f ∧ mono_embed m2 m1 g.
```

We showed `mono_equiv` holds for the multimap mono `map_mono K (set_mono V)` and `set_mono (K * V)`. To this end, we defined embedding functions for their outputs and verified they satisfy `mono_embed`.

4.2 Mono-type Functors

With mono-equivalence in place, we can now study how to transform between equivalent mono types. If we inspect the specification of mono types and their equivalence relation, we can find two significant ways to transform between equivalent mono types. First, we can post-process the result of `read` to produce an alternative output. And second, we can post-process the result of `add` to produce (and maintain) an alternative state. We call both kinds of transformations *mono-type functors* and provide sufficient criteria to ensure they preserve equivalence. This means that no manual equivalence proof will be necessary for mono types where one is derived from the other through mono-type functors.

We design output functors to change the output type and value of a mono type. To this end, we specify `OutputFunctor`, which employs a function `f` to adjust the `read` operation and leave `init` and `add` untouched. The result of `OutputFunctor` is a new mono type whose output changed from type `O1` to `O2`. In order to preserve the `mono_law`, we require `f` to be a monotone function.

```
Definition OutputFunctor (f : O1 → O2) (_ : monotone f) (m : Mono ST I O1)
  : Mono ST I O2 := {| init := init m ;  add := add m ; read s := f (read m s) |}.
```

For example, we can derive a flattened version of the multimap mono:

```
Definition flatten_multimap {A B} : map A (set B) → set (A * B).
Definition flat_multimap_mono : Mono (map K (set V)) (K * V) (set (K * V)) :=
  OutputFunctor flatten_multimap _ (map_mono K (set_mono V)).
```

Observe how `flat_multimap_mono` yields outputs of type `set (K * V)` rather than `map K (set V)`. This way, output functors allow us to create many interesting mono types by projection of the output. For example, we can use output functors to obtain a mono that only contains the unique keys of a multimap, or only those entries that satisfy a filter function.

```
Definition multimap_keys_mono : Mono (map K (set V)) (K * V) (set K) :=
  OutputFunctor map_keys _ (map_mono K (set_mono V)).
Definition multimap_size_mono : Mono (map K (set V)) (K * V) nat :=
  OutputFunctor set_size _ flat_multimap_mono.
```

Here, the last example shows that we can stack applications of `OutputFunctor`: We use a flattened multimap and take its size. This way, `multimap_size_mono` computes the number of key-value pairs.

But, when are the resulting mono types equivalent to the original ones? We have proved that, if `f` has an inverse function `g`, the mono type generated from an output functor is equivalent to the original mono type. That is, the effect of `OutputFunctor` must be reversible:

```
Theorem ob_functor_equiv : forall (m: Mono ST I O1) f (f_mono: monotone1 f) g,
    bijective f g → mono_equiv m (OutputFunctor f f_mono m) f g.
```

Not all uses of `OutputFunctor` satisfy this property. But, we can use this lemma to guarantee a multimap mono is equivalent to its flattened version:

```
Definition unflatten_multimap : set (K * V) → map K (set V).
Lemma bij_flatten_multimap : bijective flatten_multimap unflatten_multimap.
Lemma flat_multimap_equiv : mono_equiv (map_mono K (set_mono V)) flat_multimap_mono.
```

State functor. We formulate state functors to derive new mono types that operate on a modified state. Unlike output functors, state functors use a pair of functions `f` and `g` that convert between `ST1` and `ST2` and back. This is necessary because the state is both argument and result of `add`. While `add` operates on `ST1` in the original mono type, we aim to derive `add` that operates on `ST2`. To this end, we map the argument state from `ST2` to `ST1` to apply the original `add`, and then map its result back to `ST2`. To ensure the resulting definition satisfies the `mono_law`, we require function `f ∘ g` to be both extensive and increasing in domain `ST1`. Moreover, we impose another restriction `indep` to guarantee the transformed mono types meet the `ord_indep` law.

```
Definition extensive (f : ST1 → ST1) (m : Mono ST1 I O) :=
  forall s : ST1, result m s ⊑ result m (f s).
Definition increasing (f : ST1 → ST1) (m : Mono ST1 I O) :=
  forall s1 s2 : ST1, result m s1 ⊑ result m s2 → result m (f s1) ⊑ result m (f s2).
Definition ord_indep_abs (f : ST2 → I → ST2) (g : ST2 → O) :=
  forall s a1 a2, g (f (f s a1) a2) = g (f (f s a2) a1).
Definition StateFunctor (f : ST1 → ST2) (g : ST2 → ST1) (m : Mono ST1 I O)
  (_ : extensive (f ∘ g) m) (_ : increasing (f ∘ g) m)
  (indep : ord_indep_abs (fun s a ⇒ f (add m (g s) a)) (read m ∘ g) : Mono ST2 I O :=
  {| init := f (init m) ; add s a := f (add m (g s) a) ; read s := (read m ∘ g) s |}.
```

The state functor derives mono types that operate on different states internally. For example, we can transform the multimap mono to use flattened sets as states:

```
Definition multimap_flat_state_mono : Mono (set (K * V)) (K * V) (map K (set V)) :=
  StateFunctor flatten_multimap unflatten_multimap _ _ (map_mono K (set_mono V)) _.
```

Note how the derived mono type uses sets of pairs internally, although its output is still a multimap for now. But is the resulting mono type also equivalent to the original one? We have proved that, if `f` is a bijection and `g` is its inverse, `StateFunctor` preserves the equivalence of mono types:

```
Theorem st_functor_equiv {ST1 ST2 I O} `{PartialOrder O, PartialOrder ST1,
  PartialOrder ST2} : forall (m: Mono ST1 I O) (f : ST1 → ST2) (g : ST2 → ST1),
    bijective f g → mono_equiv m (StateFunctor f g _ _ m _) id id.
```

Composing functors. We can compose mono-type functors by stacking them. The order in which we compose multiple `StateFunctor` or multiple `OutputFunctor` instances matters, because one transforms the output of the other. We have seen an example of this in `multimap_size_mono`, which stacks two `OutputFunctor`: The first converts a multimap to a set of pairs, the second counts the number of entries. Consequently, the order of their application cannot be changed.

The story is different when combining a `StateFunctor` with an `OutputFunctor`: One changes the state, the other changes the output. In that sense, we could try to apply them in any order. Indeed, we were able to prove that the order of their application is irrelevant for `mono_equiv`. Figure 4 shows the corresponding commuting diagram.

4.3 Optimization by Mono-Type Replacement

The equivalence relation `mono_equiv` indicates that we can use one mono type as a drop-in replacement for another mono type. Thus, if one implementation is more efficient than the other, we may opt for the better one. Mono-type functors help us establish such equivalences.

We illustrate the mono-type replacement for the multimap mono. So far, we have established a number of mono types that are provably equivalent to the multimap mono.

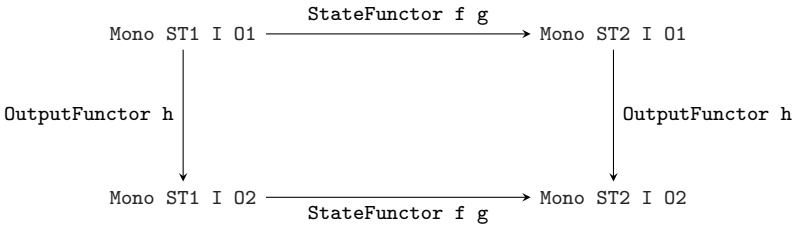


Figure 4 Commuting diagram for `OutputFunctor` and `StateFunctor` under `mono_equiv`.

```
Definition flat_multimap_mono : Mono (map K (set V)) (K * V) (set (K * V)) :=
  OutputFunctor flatten_multimap _ (map_mono K (set_mono V)).
Definition multimap_flat_state_mono : Mono (set (K * V)) (K * V) (map K (set V)) :=
  StateFunctor flatten_multimap unflatten_multimap _ _ (map_mono K (set_mono V)) _.
Definition flat_multimap_flat_state_mono : Mono (set (K * V)) (K * V) (set (K * V)) :=
  OutputFunctor flatten_multimap _ multimap_flat_state_mono.
```

The last mono type shown above composes the state functor used in `multimap_flat_state_mono` with an output functor. The implementation of this mono type is not efficient, because it is converting back and forth between maps and sets all the time. But its signature suggests a way forward: it uses a set of pairs both as state and output. This is interesting, because we have seen another mono type that admits this type:

```
Definition set_pair_mono : Mono (set (K * V)) (K * V) (set (K * V)) := set_mono (K * V).
```

Since the state and output functor have done most of the heavy lifting, it is easy to prove that the set mono on pairs is equivalent to the last multimap mono from above:

```
Lemma set_multimap_equiv : mono_equiv flat_multimap_flat_state_mono set_pair_mono id id.
```

Hence, we can replace any multimap mono by a set mono in Datalog programs. We can follow the same approach to prove further equivalences and use them to opt for more efficient mono types.

4.4 Optimization by Mono-Type Elimination

This paper introduces mono types as a new abstraction for Datalog programmers. However, not all mono types are necessary at run time because *Datalog provides some mono types as built-ins*. We can eliminate those mono types that are built into Datalog from the program.

First, most Datalog systems support a number of primitive aggregations functions, such as `count`, `max`, or `sum`. Of course, we can reuse them instead of using `nat_sum`, for example. But this is unlikely to affect the performance of the program much, because the built-in aggregations have to do the exact same work as our mono types and the aggregations resulting from them.

The second class of built-in mono types is much more important: Datalog has a number of relational mono types built-in. For example, Datalog relations essentially implement the `set_mono`: they collect tuples monotonically. If we can adopt such built-in relational mono types, we can eliminate the aggregation that the standard semantics of mono types imposes. Aggregation can have a considerable performance overhead in Datalog systems because we are leaving the realm of Datalog's highly optimized relational algebra. So this is likely to have a large performance impact.

Since the set mono is built into Datalog, we can optimize the translation rule:

if `mono_equiv` $M(\overline{m})$ (`set_mono` A) for some A:
$[\![\textbf{new}\ M(\overline{m})@(U_1, \ldots, U_k)]\!]_{agg} = \mathsf{Aggregate}_{M(\overline{m})}(m, t) :\!-\ \mathsf{Aggregand}(m, t, _).$
else:
$[\![\textbf{new}\ M(\overline{m})@(U_1, \ldots, U_k)]\!]_{agg} = \mathsf{Aggregate}_{M(\overline{m})}(m, M(\overline{m})(t)) :\!-\ \mathsf{Aggregand}(m, t, _).$

If $M(\overline{m})$ is equivalent to a set mono, the aggregation can be skipped: Note how the head of the generated rule simply forwards the aggregands. In contrast, other mono types have to aggregate the aggregands using $M(\overline{m})$. For example, we have established that the mono type for multimaps is equivalent to the mono type for sets. Therefore, multimaps do not require aggregation using the optimized translation rules.

We found another important relational mono type that is built into Datalog and which we call *functional relations*. A functional relation is a finite relation (i.e., set of tuples) with a functional dependency, meaning that some columns uniquely determine other columns. Datalog allows functional relations to aggregate the uniquely determined columns. Therefore, we can faithfully capture functional relations as the following mono type:

```
Definition fun_dep (s : set (A * B)) :=
  forall (a : A) (b b' : B), set_In (a, b) s → set_In (a, b') s → b = b'.
Definition funrel A B := {s : set (A * B) | fun_dep s}.
Definition funrel_mono K (m : Mono ST I O) : Mono (funrel K ST) (K * I) (funrel K O) :=
{| init := empty_set
 ; add s ka := let (k, a) := ka in
               funrel_add_with s k (fun st ⇒ add m st a) (add m (init m) a)
 ; read s := funrel_map_values (read m) s
|}.
```

Here, `funrel_add_with` uses the first function argument if `k` already is present in `s`, and the second the function argument for new `k`. Functional relations still require aggregation, but only on the uniquely determined columns rather than on all columns of the relation. We can exploit this in a dedicated translation rule:

if `mono_equiv` $M(\overline{m})$ (`funrel_mono` m') for some A:
$[\![\textbf{new}\ M(\overline{m})@(U_1, \ldots, U_k)]\!]_{agg} = \mathsf{Aggregate}_{M(\overline{m})}(m, k, m'(t)) :\!-\ \mathsf{Aggregand}(m, (k, t), _).$

Note how the generated rule simply propagates the first half of the input k, and only requires aggregation of the remainder t. But this only holds for mono types $M(\overline{m})$ that are equivalent to a `funrel_mono`. We were able to establish such equivalence for the map mono:

```
Theorem map_funrel_mono_equiv : forall (m : Mono ST I O)
  mono_equiv (map_mono K m) (funrel_mono K m) map_funrel funrel_map.
```

We prove this property by first applying output and state functors to `map_mono`, such that the resulting mono type has the same signature as `funrel_mono`. The remainder of the proof is manageable and only took a few hours to complete.

In conclusion, Datalog programs that use a map mono only need to use aggregation on the values of that map. Moreover, since `map_mono K1 (map_mono K2 m)` is equivalent to `map_mono (K1 * K2) m`, actually only the most-nested value type has to be aggregated. This has a significant performance impact as we will evaluate in the subsequent section.

5 Implementation, Case Studies, and Performance

We have implemented mono types as an extension of the IncA Datalog system [14] and made the code available open-source.[2] IncA supports generalized recursive aggregations over user-defined functions, which makes it well-suited for mono types. In particular, we added built-in support for set monos, map monos, and a number of primitive monos on strings and numbers. We also provide an interface for user-defined mono types, which we used to integrate lattice-based mono types for example. We integrated mono types as described in Section 3 by translating into Datalog with aggregation. The translation for set monos and map monos is configurable to activate or deactivate the optimization described in Subsection 4.3 and 4.4. Overall, the integration of mono types required 550 LoC for IncA.

Next, we evaluate the usability and performance of mono types using two case studies: dependency analysis and interval analysis, each with and without using mono types. Program analysis is a common use case for Datalog because of its least fixpoint semantics. We show that mono types significantly improve the performance of these analysis compared to pure Datalog code.

5.1 Dependency Analysis

We want to analyze the dependencies of a simple language with inter-dependent definitions. A program of the language consists of definitions that assign a name to an expression. Expressions are either numbers, additions, or variables:

$$\begin{array}{lll} \text{(programs)} & p & ::= \overline{d} \\ \text{(definitions)} & d & ::= \texttt{def}\ x = e \\ \text{(expressions)} & e & ::= n \mid x \mid e + e \end{array}$$

We represent the analyzed program as an abstract syntax tree (AST) which is encoded with algebraic data types in Datalog. Each variable refers to exactly one definition, as illustrated with subscripts in the following example:

```
def a1 = c3; def b2 = 2; def c3 = a1; def main4 = c3
```

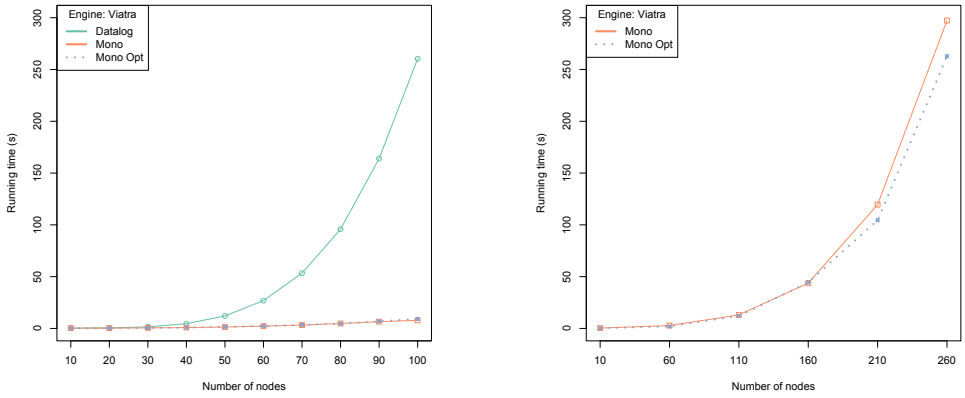
For a given definition, we compute all definitions it transitively depends on. In our example, `main`$_4$, `a`$_1$, and `c`$_3$ all transitively depend on `a`$_1$ and `c`$_3$, whereas `b`$_2$ is not used.

We implement this analysis in pure Datalog using three central relations:

```
edgesDefs(defs, from, to) :- ?Cons(defs, hd, tl), edgesDef(defs, hd, from, to).
edgesDefs(defs, from, to) :- ?Cons(defs, hd, tl), edgesDefs(tl, from, to).
edgesDef(defs, def, from, to) :- ?Def(def, _, e), refersTo(defs, e, to), from = def.
edgesDef(defs, def, from, to) :- ?Def(def, _, e), refersTo(defs, e, other),
                                 edgesDef(defs, other, from, to).
refersTo(defs, e, def) :- ?Var(e, name), findDef(defs, name, def).
refersTo(defs, e, def) :- ?Add(e, e1, e2), refersTo(defs, e1, def).
refersTo(defs, e, def) :- ?Add(e, e1, e2), refersTo(defs, e2, def).
```

We first traverse the list of all definitions in `edgesDefs`. For each definition, we compute all definitions it depends on using the two rules of `edgesDef`: A definition `def` depends on another definition `to` if (i) `def` contains a variable referencing `to`, or (ii) `def` contains a variable

[2] https://gitlab.rlp.net/plmz/inca-scala/-/tree/layered-ir/inca-ir/src/main/scala/inca/ir/extension/mono



(a) Viatra: Pure Datalog vs set mono.

(b) Viatra: Set mono vs optimized set mono.

Figure 5 Performance results for the dependency analysis.

reference to `other`, which in turn depends on `to`. An expression `e` refers to a definition `def` if `e` contains a variable with the name of `def`. We accomplish this using the `refersTo` relation to recursively traverses an expression.

We now rewrite this program with mono types. Instead of collecting and propagating edges in `edgesDefs` and `edgesDef`, we create a single mutable mono set and pass it around. That way, `edgesDef` can directly write reachable pairs of definitions into the mono set when they are discovered:

```
main(from, to) :- m = new set[(Def, Def)], edgesDefs(defs, m), read(m) = (from, to).
edgesDefs(defs, m) :- ?Cons(defs, hd, tl), edgesDef(defs, hd, m), edgesDefs(tl, m).
edgesDef(defs, def, m) :- ?Def(def, _, e), refersTo(defs, e, trg),
                          m += (def, trg), edgesDef(defs, trg, m).
```

The changes required to the original dependency analysis are minimal. And the program became simpler. For example, we now only need a single rule for `edgesDef` that registers a dependency edge `(def, trg)` and also recurs. Similarly, `edgesDefs` only needs one rule that asks for the dependencies of `hd` and recurs on `tl` at once. But, most importantly for this case study, mono types improved the performance significantly.

In Figure 5, we show the execution time for the dependency analysis for different sized cyclic graphs on Viatra [17]. Figure 5a reveals that using a set mono reduces the running time considerably compared to pure Datalog. This is because the analysis using pure Datalog code has to propagate edges in `edgesDefs` and `edgesDef`, which produces orders of magnitudes more tuples than the analysis using mono types. Using a mono set eliminates this overhead. To highlight the difference between the optimized (Mono opt) and non-optimized set mono (Mono), we ran additional tests with bigger input graphs. As shown in Figure 5b, the optimization of set monos yields around 10% performance gains. For graphs exceeding 200 nodes this difference becomes more evident. The optimization does not have a bigger impact because, internally, the state of the non-optimized set mono is represented by an immutable Scala set. The aggregation operation needs to add elements to this internal state, which is a constant time operation. Since we are only reading the set mono once, in the `main` relation, this aggregation only happens a single time. We have additionally translated the dependency analysis program into Soufflé [5] and Ascent [11]. Since neither of them supports user-defined aggregation, we can only compare the performance between plain Datalog and optimized set mono on these two engines. We found that the experimental results are almost identical

to Viarta, optimized set mono's running time is significantly less than plain Datalog. This experiment result confirms mono types are an effective and general optimization technique for Datalog.

5.2 Interval Analysis

We also implement a flow-sensitive interval analysis for the While language in Datalog as idiomatically as possible. We then rewrite it to use nested map monos. Both implementations share common relations to compute the control flow graph and abstractly evaluate expressions.

Based on the control flow graph, the pure Datalog version uses two mutually recursive relations `intervalBefore` and `intervalAfter`. These relations define the interval value for a variable before and after the execution of a specific statement. To compute `intervalAfter`, we distinguish two different cases. If the current statement is an assignment of the current variable `var`, then `aeval` computes the value of `var` after the assignment. In all other cases, we propagate the information from `intervalBefore`. Note how `aeval` requires a reference to the current statement so that it can query `intervalBefore` for the value of referenced variables in `exp`. To compute `intervalBefore` for a variable `var`, we join the intervals of `var` that are valid after any predcessor statement.

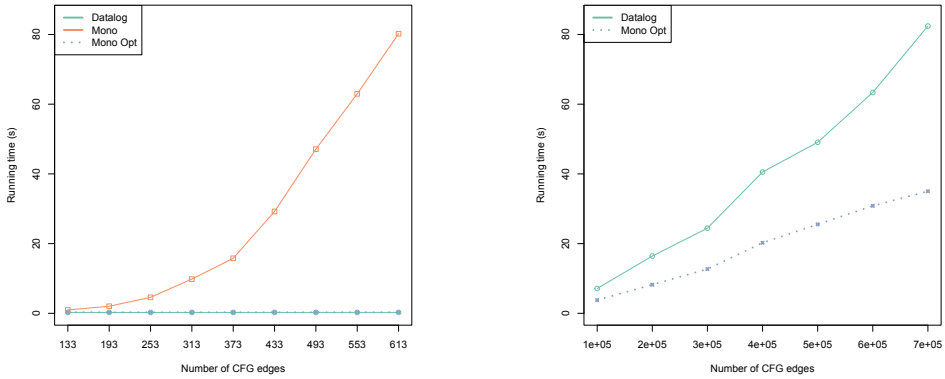
```
intervalAfter(stmt, var, iv) :- assignment(stmt, var, exp), aeval(stmt, exp, iv).
intervalAfter(stmt, var, iv) :- nonassignedVars(stmt, var), intervalBefore(stmt, var, iv).
aeval(stmt, exp, iv) :- ?Var(exp, v), intervalBefore(stmt, v, iv).
aeval(stmt, exp, iv) :- ...
predecessorIntervals(stmt, var, pred, iv) :- cflow(pred, stmt),
                                             intervalAfter(pred, var, iv).
intervalBefore(stmt, var, lub(iv)) :- predecessorIntervals(stmt, var, _, iv).
```

To implement the same analysis using mono types, we propagate a single nested mono map while traversing the control flow graph. The mono map yields a result of type `map[Stmt, map[Var, Interval]]`, which is the standard abstract domain used in flow-sensitive data-flow analysis: For each statement, we compute the abstract value of each variable. With this mono type, we do not need explicit aggregation in the Datalog code anymore. We initialize the nested mono map for the first statement in the analyzed code and then propagate it to all other statements. Note that we use the upper-case `Interval` for the type of interval values and the lower-case `interval` for the interval mono type.

```
traverse(stmt, m) :- isFirst(stmt), m = for stmt : new map[Stmt, map[Var, Interval]](
                                        new map[Var, Interval](new interval)].
traverse(stmt, m) :- cflow(pred, stmt), traverse(pred, m).
```

Similar to before, we need to handle two different cases when we process a statement. If the statement is an assignment of a variable `var` to some expression `exp`, we evaluate `exp` under an environment `env`. We obtain `env` from the map mono by first reading the mono type, which yields the full flow-sensitive analysis result as a nested map, and then performing a map lookup at `stmt`, which yields a variable map. This way, `aeval` does not depend on `intervalBefore` or the like, it simply reads variable values from the given environment. If the analyzed statement does not assign `var`, then we simply propagate its value to all successor statements:

```
transfer(stmt) :- traverse(stmt, m), assignment(stmt, var, exp), cflow(stmt, succ),
                  env = read(m)(stmt), aeval(env, exp, iv), m += (succ, (var, iv)).
transfer(stmt) :- traverse(stmt, m), nonassignedVars(stmt, var), cflow(stmt, succ),
                  env = read(m)(stmt), iv = env(var), m += (succ, (var, iv)).
aeval(env, exp, iv) :- ?Var(exp, v), iv = env(v).
aeval(env, exp, iv) :- ...
```



(a) Pure Datalog vs map mono.

(b) Pure Datalog vs optimized map mono.

Figure 6 Performance results for the interval analysis.

In our opinion, mono types enable a much cleaner definition of the interval analysis that is very close to the definitions found in program-analysis textbooks. In fact, the definition of `transfer` is most declarative: it describes equations that must hold for each statement. And since `transfer` is neither directly nor indirectly recursive, it can be understood as a specification without even considering a fixed-point semantics. Of course, solving this analysis problem still requires a fixed-point computation due to loops in the analyzed program. This is evident in the analysis definition implicitly: The values written to the map mono depend on the values read from the map mono, which will be translated into recursive aggregation. The mono type protected the programmer from this complication. But, is there a performance penalty for this abstraction?

Figure 6 shows the performance results for the interval analysis. On the left, we can see that the non-optimized mono version is orders of magnitude slower than using pure Datalog. But these are very small programs with only a few hundred CFG edges. We suspect that this slow down is due to the explicit map data structures that needs to be maintained by the non-optimized map mono. The optimization for map monos removes this overhead, by flattening the nested map mono and converting it to relations. Not only does this eliminate the overhead of the mono type abstraction, the optimized program actually scales much better to analyzing large programs. For example, to analyze programs with $\sim$ 700K CFG edges, the optimized map mono requires less than 40s whereas the pure Datalog version requires more than 80s. This clearly shows that mono types provide abstraction without regret.

6 Related Work

We develop and mechanize the theory of mono types and their properties in Rocq. To the best of our knowledge, we are the first to introduce a mutable, monotonic container abstraction for Datalog. However, in the context of distributed computing and parallel programming, monotone data types have been studied in the past, which we discuss below. Besides a thorough theoretical exploration, we also integrate and evaluate mono types using the IncA framework for Datalog. In recent years, numerous extensions and frontends for Datalog have been proposed in the literature. Some of these have features that are related to mono types, as the following discussion shows.

6.1 Datalog Dialects

Compared to all existing Datalog dialects, mono types bring three novel features. First, mono types generalize existing styles of aggregation, making aggregation more expressive. Second, mono types have a mechanized theory in Rocq, including constructions, functors, and an equivalence relation. And third, mono types behave like first-class containers that can be propagated and filled anywhere, which improves programmability and performance. In our discussion below, we focus on the first part, namely the expressiveness of aggregation. This is because no existing Datalog system has a mechanized aggregation theory nor supports anything like first-class containers.

Datafun [3] is a typed higher-order functional programming language that can express Datalog-style programs. Datafun supports aggregation over relations via the join operator of semi-lattices. Our mono types are a generalization of semi-lattices and permit `add` operations that are not computing a least upper bound, such as required for summation. Datafun's key feature is to track monotonicity with types by differentiating discrete and monotone variables. In contrast, we choose to develop and prove the theory of mono types in Rocq and use this foundation to let users design custom mono types. This is a classic trade-off between automation and expressiveness. Specifically, Datafun's type system has limited capabilities for establishing monotonicity and therefore only supports a few built-in semi-lattices, whereas our implementation supports user-defined mono types. Finally, Datafun ensures the termination of fixed-point computations by constraining them to finite semi-lattice types, which are tracked by its type system. With mono types, we do not guarantee termination but instead require the user to prevent infinite ascending chains, which is in conformance with lattice-based aggregation in existing Datalog systems. It would be interesting to study if Datafun can be faithfully extended to support mono types in place of semi-lattices while retaining its metatheory.

Flix [8] and IncA [13] are Datalog frameworks that support recursive aggregation over user-defined aggregation operators. While most aggregation operators in their case studies are semi-lattices, they can also express other monotonic operations, such as summation [15]. Mono types are more general still, because they support a post-processing through the `read` operation, and only the result of `read` must be monotonic. Since we deviated from the textbook semi-lattice aggregation, we established a theory of mono types formally in Rocq. Flix supports validating the monotonicity of user-defined aggregation operators using an SMT solver [7]. However, given the complexity of properties we needed to establish for mono types, it seems highly unlikely that an automated approach can establish the same guarantees. In particular, to enable optimizations of mono types, we had to prove a number of equivalences between mono types. These optimizations have a significant performance impact, as our evaluation shows. Existing Datalog systems cannot optimize aggregations.

BYODS [10] is a Datalog engine and dialect written in Rust with support for custom data structures to back Datalog relations. This not only improves performance, but also allows the user to extend the semantics of Datalog. Namely, BYODS represents relation as a triple $(D, \mathit{inj}, \gamma)$, where D is a lattice that represents the state of the custom relation. The injection function $\mathit{inj} : T \to D$ converts a tuple T to D, which can then be joined with the current state of the custom relation. And the concretization function $\gamma : D \to \mathcal{P}(T)$ extracts a set of tuples from the state of the custom relation. They have proved that if γ is a monotonic function, the custom relation will not break the fixed point semantics of Datalog. The custom relation is similar to mono types, as D can be seen as the state type, inj and γ are close to the `add` and `read` operation respectively. However, mono types do not require the state type to be ordered and the output of mono types is not limited to powersets. In

addition, BYODS can also be used to support user-defined lattices and recursive aggregations, which give rise to performing aggregation on map lattices. But they do not explore how to optimize aggregations, which requires an equivalence relation over user-defined types like ours. A promising future work is to combine BYODS with mono types to enable additional performance improvements after our optimizations.

Dedalus [2] introduces state to Datalog to reason about distributed systems. In particular, Dedalus introduces a notion of time to Datalog by annotating relations with a monotonically increasing timestamp. Each change happens at a specific point in time. To persist a fact over time, it is rederived in each subsequent iteration. Effectively, Dedalus evaluates Datalog over time, which enables strong updates by revoking or updating facts. While our mono types also introduce a notion of mutable state to Datalog, they do not permit strong updates. Even though we enforce no restriction on the state of a mono type, we require their output to grow monotonically to guarantee a unique minimal model. It is not obvious how we could support strong updates instead of monotone updates without conflicting with Datalog's fixed-point semantics. In general, mono types and Dedalus serve different purposes. Mono types provide a general purpose abstraction for Datalog to increase programmability and performance. Dedalus on the other hand, evaluates Datalog over time to reason about distributed systems.

6.2 Application of Monotonic Data Types

Conflict-free replicated data types (CRDTs) [12] is a framework for maintaining eventually consistent values in distributed systems. There are two types of CRDTs, operation-based CRDTs and state-based CRDTs. Mono types are more closely related to state-based CRDTs. Specifically, a state-based CRDT object consists of an initial state, a query method to read the current state, an update method to modify state, and a merge method to compute the least upper bound between local state and remote states. CRDT requires the state to grow monotonically after applying an update or merge method. Except for the merge method, we can find that state-based CRDTs and mono types exploit a similar design pattern. However, the monotonicity property serves different purposes in CRDTs and mono types. CRDTs require monotonicity to guarantee that all of the replicas reach the same state eventually. In contrast, mono types use monotonicity to ensure the well-definedness of recursive aggregation in Datalog [13].

Nevertheless, the similarity between CRDTs and mono types leads to a promising direction for future work: extending mono types to Datalog dialects that are used in the distributed system, such as Dedalus [2], Bloom [1], or Bloom^L [4]. Notably, Bloom^L allows passing lattice-based stateful objects across relations, and adopts similar ideas as mono types to collect the inputs of objects. However, mono types provide a more generalized form of lattice-based aggregation that differentiates between state and output, which is more flexible than Bloom^L. In addition, Bloom^L does not provide optimization for sets and maps.

LVars [6] (lattice vars) is a model for deterministic-by-construction parallel programming. Its interface only exposes a put operation (compute the `lub`) and a get operation (to read the state) to programmers. To guarantee determinism, LVars require the get operation to only return a state from a predefined threshold set Q, in which the `lub` of two distinct elements is $\top$ (the error state). With mono types, we do not need to associate a threshold set since we can rely on the eventual consistency of the fixed-point computation.

7 Conclusion

We propose mono types, a monotonic data structure for Datalog to improve its programmability and performance. Mono types support three essential operations: (1) we can instantiate them, (2) we can add elements to them, and (3) we can read monotonically increasing outputs from them. To guarantee well-defined behaviour for Datalog programs, we mechanize and prove the theory of mono types in Rocq. Given this theoretical foundation, we integrate mono types into Datalog by translating them to generalized aggregations over relations. We identify two central container mono types: set mono and map mono. Using a new concept of mono equivalence, we show that these containers are native to Datalog in the sense that we can profit from Datalog's set semantics to implement them. To validate the usability and performance improvements of mono types, we implement two case studies, a dependency analysis and an interval analysis. We show that mono types significantly improve the performance of these analysis compared to pure Datalog.

References

1 Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in bloom: a CALM and collected approach. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260. www.cidrdb.org, 2011. URL: `http://cidrdb.org/cidr2011/Papers/CIDR11_Paper35.pdf`.

2 Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. Dedalus: Datalog in time and space. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, pages 262–281, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

3 Michael Arntzenius and Neelakantan R. Krishnaswami. Datafun: A functional Datalog. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 214–227. ACM, 2016. `doi:10.1145/2951913.2951948`.

4 Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In Michael J. Carey and Steven Hand, editors, *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*, page 1. ACM, 2012. `doi:10.1145/2391229.2391230`.

5 Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing.

6 Lindsey Kuper and Ryan R. Newton. Lvars: Lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing*, FHPC '13, pages 71–84, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2502323.2502326`.

7 Magnus Madsen and Ondrej Lhoták. Safe and sound program analysis with flix. In Frank Tip and Eric Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 38–48. ACM, 2018. `doi:10.1145/3213846.3213847`.

8 Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. From Datalog to Flix: A declarative language for fixed points on lattices. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 194–208. ACM, 2016. `doi:10.1145/2908080.2908096`.

9 Kenneth A. Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. In Moshe Y. Vardi and Paris C. Kanellakis, editors, *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, 1992, San Diego, California, USA*, pages 114–126. ACM Press, 1992. doi:10.1145/137097.137852.

10 Arash Sahebolamri, Langston Barrett, Scott Moore, and Kristopher K. Micinski. Bring your own data structures to datalog. *Proc. ACM Program. Lang.*, 7(OOPSLA2):1198–1223, 2023. doi:10.1145/3622840.

11 Arash Sahebolamri, Thomas Gilray, and Kristopher K. Micinski. Seamless deductive inference via macros. In Bernhard Egger and Aaron Smith, editors, *CC '22: 31st ACM SIGPLAN International Conference on Compiler Construction, Seoul, South Korea, April 2 - 3, 2022*, pages 77–88. ACM, 2022. doi:10.1145/3497776.3517779.

12 Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2011. doi:10.1007/978-3-642-24550-3_29.

13 Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. Incrementalizing lattice-based program analyses in Datalog. *Proc. ACM Program. Lang.*, 2(OOPSLA):139:1–139:29, 2018. doi:10.1145/3276509.

14 Tamás Szabó, Sebastian Erdweg, and Markus Voelter. IncA: a DSL for the definition of incremental program analyses. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 320–331. ACM, 2016. doi:10.1145/2970276.2970298.

15 Tamás Szabó, Edlira Kuci, Matthijs Bijman, Mira Mezini, and Sebastian Erdweg. Incremental overload resolution in object-oriented programming languages. In Julian Dolby, William G. J. Halfond, and Ashish Mishra, editors, *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA 2018, Amsterdam, Netherlands, July 16-21, 2018*, pages 27–33. ACM, 2018. doi:10.1145/3236454.3236485.

16 K. Tuncay Tekle and Yanhong A. Liu. Precise complexity analysis for efficient datalog queries. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, pages 35–44, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1836089.1836094.

17 Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Softw. Syst. Model.*, 15(3):609–629, 2016. doi:10.1007/S10270-016-0530-4.

18 Runqing Xu, David Klopp, and Sebastian Erdweg. Mono Types – First-Class Containers for Datalog. Software (visited on 2025-06-13). URL: https://gitlab.rlp.net/plmz/artifacts/mono-types-ecoop25, doi:10.4230/artifacts.23600.

Taming and Dissecting Recursions Through Interprocedural Weak Topological Ordering

Jiawei Yang*
The Hong Kong Polytechnic University, Hong Kong

Xiao Cheng*
University of New South Wales, Sydney, Australia

Bor-Yuh Evan Chang†
University of Colorado Boulder, Boulder, CO, USA
Amazon, Boulder, CO, USA

Xiapu Luo
The Hong Kong Polytechnic University, Hong Kong

Yulei Sui
University of New South Wales, Sydney, Australia

Abstract

Abstract interpretation provides a foundational framework for approximating program semantics by interpreting code through abstract domains using semantic functions over ordered sets along a program's control flow graph (CFG). To facilitate fixpoint computation in abstract interpretation, weak topological ordering (WTO) is an effective strategy for handling loops, as it identifies strategic control points in the CFG where widening and narrowing operations should be applied. However, existing abstract interpreters still face challenges when extending WTO computation in the presence of recursive programs. Computing a precise whole-program WTO requires full context-sensitive analysis which is not scalable for large programs, while context-insensitive analysis introduces spurious cycles that compromise precision. Current approaches either ignore recursion (resulting in unsoundness) or rely on conservative approximations, sacrificing precision by adopting the greatest elements of abstract domains and applying widening at function boundaries without subsequent narrowing refinements. These can lead to undesired results for downstream tasks, such as bug detection.

To address the above limitations, we present RecTopo, a new technique to boost the efficiency of precise abstract interpretation in the presence of recursive programs through interprocedural weak topological ordering (IWTO). Rather than pursuing an expensive whole-program WTO analysis, RecTopo employs an on-demand approach that strategically decomposes programs at recursion boundaries and constructs targeted IWTOs for each recursive component. RecTopo dissects and analyzes (nested) recursions through interleaved widening and narrowing operations. This approach enables precise control over interpretation ordering within recursive structures while eliminating spurious recursions through systematic correlation of control flow and call graphs.

We implemented RecTopo and evaluated its effectiveness using an assertion-based checking client focused on buffer overflow detection, comparing it against three popular open-source abstract interpreters (IKOS, Clam, CSA). The experiments on 8312 programs from the NIST dataset demonstrate that, on average, RecTopo is 31.99% more precise and achieves a 17.49% higher recall rate compared to three other tools. Moreover, RecTopo exhibits an average precision improvement of 46.51% and a higher recall rate of 32.98% compared to our baselines across ten large open-source projects. Further ablation studies reveal that IWTO reduces spurious widening operations compared to whole-program WTO, resulting in a 12.83% reduction in analysis time.

*These authors contributed equally to this work.
†Bor-Yuh Evan Chang holds concurrent appointments at the University of Colorado Boulder and as an Amazon Scholar. This paper describes work performed at the University of Colorado Boulder and is not associated with Amazon.

© Jiawei Yang, Xiao Cheng, Bor-Yuh Evan Chang, Xiapu Luo, and Yulei Sui;
licensed under Creative Commons License CC-BY 4.0
39th European Conference on Object-Oriented Programming (ECOOP 2025).
Editors: Jonathan Aldrich and Alexandra Silva; Article No. 34; pp. 34:1–34:31

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases Abstract interpretation, recursion, weak topological ordering

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.34

Supplementary Material *Software (Source Code)*: `https://github.com/JoelYYoung/RecTopo`

Funding This research is supported by Australian Research Council Grants DP250101396 and FT220100391.

1 Introduction

Abstract interpretation [24, 22, 25] offers a framework for approximating program semantics by analyzing code through abstract domains and applying semantic functions over ordered sets along the program's control flow graph (CFG). The analysis converges to a conservative solution at a post-fixpoint, which is an over-approximation of the least fixpoint of the program's semantics. This least fixpoint is defined by a system of equations $a_1 = \Phi_1(a_1, a_2, \ldots, a_n),\ \ldots,\ a_n = \Phi_n(a_1, a_2, \ldots, a_n)$, where each a_i denotes the abstract states at control point i, and Φ_i represents the corresponding semantic function of a program.

For programs with acyclic control flows, fixpoint computation typically converges after a few iterations over the CFG. However, the presence of cycles introduces cyclic dependencies between program states, complicating the analysis when working with abstract domains characterized by unbounded or very deep lattice heights [27, 62]. In such domains, such as the standard interval domain [17], the iterative computation may require a prohibitively large number of steps to reach a post-fixpoint. Even worse, termination is not guaranteed, as a post-fixpoint may not be reached within a finite number of steps. To facilitate fixpoint computation in these domains, widening techniques [22, 24, 13] compute a safe (upper) approximation of the least fixpoint (a post-fixpoint, hereafter referred to as a fixpoint for simplicity). However, widening operations must be judiciously applied only at cycle points, as spurious widening at acyclic locations introduces precision loss due to over-approximation, which cannot be recovered through subsequent narrowing [11]. Bourdoncle's algorithm [13] provides an effective solution through weak topological ordering (WTO) for intraprocedural analysis, which efficiently identifies program loops and their nested structures while requiring only one widening point per loop. The WTO framework further enables sophisticated interleaved widening and narrowing strategies [10], where narrowing compensates precision that may have been lost during widening by incorporating invariants such as loop bounds and branching constraints. These advantages have made WTO a standard method for handling intraprocedural loops in many current abstract interpreters [15, 26, 19, 18].

While state-of-the-art abstract interpreters effectively leverage Bourdoncle's algorithm for loop analysis, they face difficulties when determining WTO for recursive function calls. Several approaches have been proposed for taming recursions and interprocedural abstract interpretation. For non-recursive functions, function inlining enables precise interprocedural analysis. However, this approach becomes infeasible for recursive functions due to potentially infinite inlining sequences in unbounded recursion. To address this, some tools adopt strong assumptions, employing either under- or over-approximation approaches. For example, Astrée [37] completely excludes the analysis of unbounded recursion, while IKOS [15, 40] conservatively assigns top values ($\top$) to variables in recursive contexts, effectively bypassing the analysis of recursive functions. To improve precision, tools such as Clam [34, 19] and CSA [18] analyze recursive function bodies but introduce many spurious widening operations

at function boundaries. Their effectiveness remains limited, as their WTO computations are confined to individual function scopes, preventing the creation of an effective interprocedural order for efficient fixpoint computation and subsequent narrowing to enhance precision. Consequently, both under-approximation and over-approximation approaches can produce undesirable outcomes for downstream tasks, such as bug detection.

Addressing these limitations necessitates an approach that incorporates both the analysis of recursive function bodies and guarantees termination properties. This requires an interprocedural weak topological ordering (IWTO) that establishes an efficient computation order across function boundaries, enabling minimal widening points and allowing narrowing strategies for precise handling of recursions. The primary challenge lies in accurately identifying the IWTO within the control flow graph, where multiple call sites of the same function can generate spurious recursive paths. These paths arise when call sites connect to prior invocations of the same function, potentially introducing unnecessary widening points and thereby compromising the precision of the analysis.

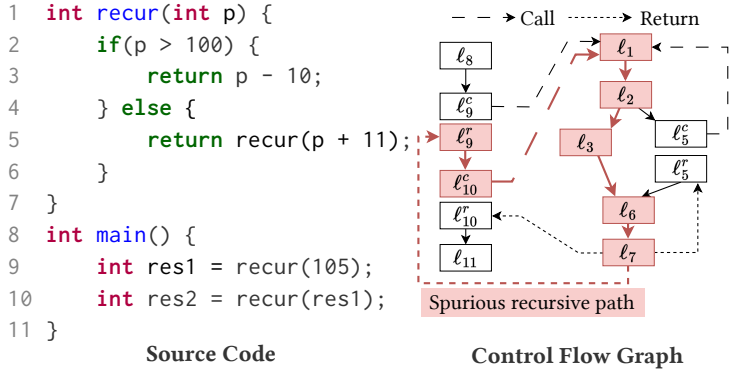


Figure 1 Spurious recursive path when analyzing McCarthy 91 function [43] (a library of mathematical theory of computation).

Figure 1 illustrates this challenge in the context of the McCarthy 91 function [43]. The function `recur` ($\ell_1 - \ell_7$) is a recursive function that takes an integer `p` as input and recursively invokes itself at ℓ_5 to increase `p`'s value until `p` is greater than 100. The main function invokes `recur` twice at ℓ_9 and ℓ_{10} respectively. In the control flow graph, each function invocation at location ℓ_n is decomposed into two distinct nodes: a call site ℓ_n^c and a corresponding return site ℓ_n^r (e.g., the call at ℓ_9 in the source program manifests as ℓ_9^c and ℓ_9^r in the control flow graph). The control flow edges are established such that each call site connects to its callee's entry point (e.g., $\ell_5^c \dashrightarrow \ell_1$, $\ell_9^c \dashrightarrow \ell_1$, and $\ell_{10}^c \dashrightarrow \ell_1$), while return edges link the callee's exit point to the corresponding return sites (e.g., $\ell_7 \dashrightarrow \ell_{10}^r$, $\ell_7 \dashrightarrow \ell_9^r$, and $\ell_7 \dashrightarrow \ell_5^r$). Note that, a spurious recursive path highlighted in red in the figure ($\ell_9^r \rightarrow \ell_{10}^c \dashrightarrow \ell_1 \rightarrow \ell_2 \rightarrow \ell_3 \rightarrow \ell_6 \rightarrow \ell_7 \dashrightarrow \ell_9^r$) arises due to the consecutive call to the function `recur` at ℓ_9 and ℓ_{10}, which may lead to a redundant widening operation at the spurious recursion and compromises analysis precision. While a fully context-sensitive analysis could theoretically eliminate such spurious paths through precise call-return matching, the exponential complexity of such an approach renders it impractical for real-world program analysis.

We present RecTopo, a new interprocedural fixpoint computation technique that computes interprocedural weak topological ordering (IWTO) in the presence of recursive programs, enabling recursions to be handled as efficiently and precisely as loops. Our approach

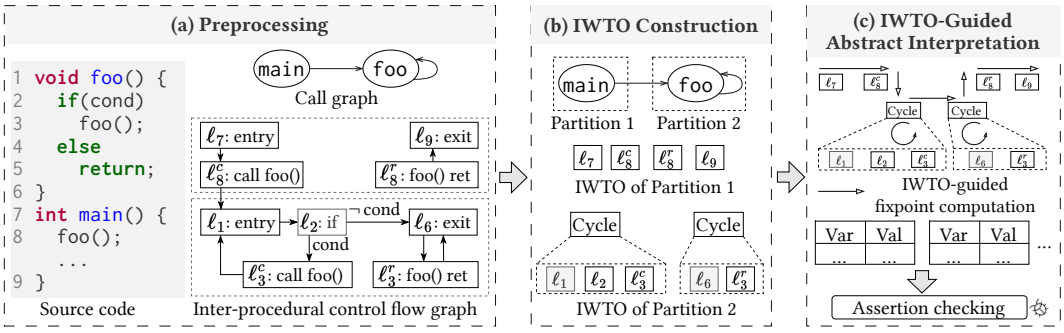


Figure 2 An overview of RecTopo framework.

systematically dissects programs into non-recursive and recursive components, avoiding the complexity of whole-program WTO construction. The framework first leverages Tarjan's strongly connected component algorithm [61] on the call graph to identify recursive function calling chains. Subsequently, on the CFG, it coalesces call sites of functions within the same recursive chain and assigns each recursion a unified IWTO. This principled decomposition enables the seamless integration of an efficient worklist algorithm with interleaved widening and narrowing operations for precise recursion analysis. Our approach also eliminates spurious interprocedural cycle, thus preventing spurious widening operations that would otherwise arise from distinct call sites of the same function.

Figure 2 illustrates the three-phase architecture of RecTopo, demonstrating how our framework systematically addresses the challenges of interprocedural analysis in the presence of recursions:

(a) Preprocessing: The framework begins by transforming the input program into LLVM's intermediate representation (LLVM-IR) [41]. This standardized representation provides a unified foundation for constructing both the interprocedural control-flow graph (ICFG) and the corresponding call graph.

(b) IWTO Construction: At the core of our approach lies a novel two-step IWTO construction process. First, we employ Tarjan's algorithm [61] on the call graph to identify and partition functions based on their recursive relationships. This partitioning is crucial as it enables us to isolate and group functions within the same recursive calling chain. Then, guided by these function partitions, we construct their corresponding IWTOs on the ICFG through iterative applications of Tarjan's algorithm. This strategic decomposition yields a precise IWTO for each recursion, providing an effective foundation for subsequent analysis phases while avoiding the complexity of whole-program WTO construction.

(c) IWTO-Guided Abstract Interpretation: The framework leverages the constructed IWTOs to perform precise abstract interpretation. By following the computed order, our analysis ensures effective convergence through strategic application of widening and narrowing operations at carefully selected points. This principled approach yields highly precise abstract states that serve as a foundation for various client analyses, such as buffer overflow detection, while maintaining analysis efficiency.

In summary, this paper makes the following major contributions:

- We introduce RecTopo, a new fixpoint computation approach for precise interprocedural abstract interpretation with enhanced efficiency. Our approach manages recursions precisely by leveraging interprocedural weak topological ordering (IWTO) and an interleaved strategy of widening and narrowing operations.

Table 1 Analysis domains and LLVM-like statement set.

ℓ		$\in \mathcal{L}$	Statements
$\mathtt{c}, \mathtt{fld}$		$\in \mathcal{C}$	Constants
$\mathtt{p}, \mathtt{q}, \mathtt{idx}$		$\in \mathcal{S}$	Stack virtual registers
$\mathtt{g}$		$\in \mathcal{G}$	Global variables
$\mathtt{f}$		$\in \mathcal{F} \subseteq \mathcal{G}$	Program functions
$\mathtt{p}, \mathtt{q}, \mathtt{idx}, \mathtt{g}$		$\in \mathcal{P} = \mathcal{S} \cup \mathcal{G}$	Top-level variables/pointers
$\mathtt{o}$		$\in \mathcal{O}$	Abstract objects
$\mathtt{v}$		$\in \mathcal{V} = \mathcal{P} \cup \mathcal{O}$	Program variables
ℓ	$::=$	$\mathtt{p} = \mathtt{c} \mid \mathtt{p} = \&\mathtt{o}$	Const/Addr
	$\mid$	$\mathtt{p} = \&(\mathtt{q} \rightarrow \mathtt{fld}) \mid \mathtt{p} = \&\mathtt{q}[\mathtt{idx}]$	Gep
	$\mid$	$\mathtt{p} = *\mathtt{q} \mid *\mathtt{p} = \mathtt{q}$	Load/Store
	$\mid$	$\mathtt{p} = \mathtt{q}$	Copy
	$\mid$	$\mathtt{p} = \phi(\mathtt{p}_1, \mathtt{p}_2, \dots)$	Phi
	$\mid$	$\mathtt{r} = \mathtt{p} \odot \mathtt{q}$	Binary
	$\mid$	$\mathtt{p} = \neg\mathtt{q}$	Unary
	$\mid$	$\mathtt{r} = \mathtt{f}(\mathtt{p}_1, \dots, \mathtt{p}_n)$	Call
	$\mid$	$\mathtt{ret}_\mathtt{f}\ \mathtt{q}$	Return
$\odot \in \{+, -, *, /, \%, <<, >>, <, >, \&, \&\&, <=, >=, \equiv, \sim, \mid, \wedge\}$			

- We introduce a scalable algorithm for constructing an IWTO for each recursion, designed to circumvent the high-cost construction of a whole-program weak topological ordering. This is accomplished by applying Tarjan's strongly connected component algorithm [61] on the call graph, which in turn guides the construction of IWTOs on the control flow graph.
- We comprehensively evaluate RecTopo's performance using 8312 programs from the NIST dataset and 10 real-world open-source projects. Experimental results show that RecTopo outperforms our baselines by achieving 31.99% higher precision and a 17.49% increase in recall rate on average. Furthermore, RecTopo shows an average precision improvement of 46.51% and a higher recall rate of 32.98% compared to baselines tools on ten open-source projects.

2 Background

This section provides the foundational knowledge of the language on which our abstract interpretation is based (see §2.1), as well as discussions on abstract interpretation (see §2.2) and weak topological ordering (see §2.3).

2.1 Language

Table 1 presents the analysis domains and LLVM-like statement set used in abstract interpretation. The set of program variables $\mathcal{V}$ is split into two disjoint subsets: $\mathcal{O}$, comprising *address-taken variables*, and $\mathcal{P}$, containing all *top-level variables*. Top-level variables $\mathtt{p}, \mathtt{q}, \mathtt{idx}, \mathtt{g} \in \mathcal{P}$ adhere to static single assignment (SSA) form and are subject to exactly one definition. Address-taken objects $\mathtt{o} \in \mathcal{O}$ can only be accessed through pointer dereferencing operations at Load/Store instructions. Constant values are initially bound to top-level variables through Cons instructions ($\mathtt{p} = \mathtt{c}$) that assign constants $\mathtt{c}$. Addr statements,

denoted as `p = &o`, allocate address-taken objects `o`, which may represent stack variables, global variables, or abstract heap objects. GEP enables structured access to composite objects: struct fields are accessed via constant offsets `fld`, while array elements use variable offsets `idx`. For field-sensitive analysis, RECTOPO employs a field-index-based strategy [12, 52], distinguishing struct fields through unique indices. COPY denotes a simple assignment of top-level variables/pointers. The PHI instruction, fundamental to SSA form, consolidates variable values at control flow merge points. BINARY and UNARY instructions represent arithmetic and bitwise operations, respectively. CALL and RETURN represents parameter passing and return value handling in function invocations.

2.2 Abstract Interpretation

Abstract interpretation offers a formalized framework for computing program semantics at each control point, represented as an abstract state specific to that point. The collection of all abstract states constitutes an abstract trace, denoted as σ (see Definition 2). The analysis iteratively updates σ by applying a semantic function – assumed to be monotonic to simplify the exposition – until reaching a fixpoint. This fixpoint yields a sound over-approximation of the program's behaviors.

▶ **Definition 1** (Concrete and Abstract Domains). *Let $\mathbb{C} = 2^{\mathcal{V}\to S}$ be the concrete domain, where S represents all possible runtime values. The abstract domain $\mathbb{A}$ forms a lattice $\langle \mathbb{A}, \sqsubseteq, \sqcap, \sqcup, \bot, \top \rangle$ with partial order $\sqsubseteq$. A Galois connection $\mathbb{C} \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} \mathbb{A}$ formalizes their relationship through abstraction function $\alpha : \mathbb{C} \to \mathbb{A}$ and concretization function $\gamma : \mathbb{A} \to \mathbb{C}$.*

▶ **Definition 2** (Abstract Trace $\sigma \in \mathbb{L} \to \mathbb{A}$). *An abstract trace maps control points to abstract states, where σ_L denotes the abstract state at control point $L \in \mathbb{L}$. Function $\sigma_L(x)$ yields variable x's abstract value at L. For a program statement ℓ, $\sigma_{\overline{\ell}}$ and $\sigma_{\underline{\ell}}$ represent the pre- and post-abstract states, respectively.*

Widening. The widening technique [24, 22] is introduced to ensure fixpoint termination and accelerate convergence by approximating infinite loop behavior. It involves choosing a subset of control points $\mathcal{L}' \subseteq \mathcal{L}$ as widening points. For each widening point $\ell_i \in \mathcal{L}'$, we use equation $\sigma_{\underline{\ell_i}} = \sigma_{\underline{\ell_i}} \nabla \Phi_i(\sigma_{\underline{\ell_1}}, \sigma_{\underline{\ell_2}}, \ldots, \sigma_{\underline{\ell_n}})$ instead of $\sigma_{\underline{\ell_i}} = \Phi_i(\sigma_{\underline{\ell_1}}, \sigma_{\underline{\ell_2}}, \ldots, \sigma_{\underline{\ell_n}})$ for fixpoint computation. In this equation, ∇ is a widening operator, which is formally defined on a poset $(\mathbb{A}, \sqsubseteq)$ that satisfies the following properties: (1) $\forall a, a' \in \mathbb{A}, a \sqcup a' \sqsubseteq a \nabla a'$ and (2) for an increasing chain $a_1 \sqsubseteq a_2 \sqsubseteq ... \sqsubseteq a_m$, the increasing chain $a'_1 \sqsubseteq a'_2 \sqsubseteq ... \sqsubseteq a'_m$ where $a'_1 = a_1$ and $a'_m = a'_{m-1} \nabla a_m$ eventually stabilizes. For example, the widening function on the integer interval domain $\mathcal{I} = \{[l, u] | l, u \in \mathbb{Z}\}$ is defined as: $[l_0, u_0] \nabla [l_1, u_1] = [\text{if } l_1 < l_0 \text{ then } -\infty \text{ else } l_0, \text{if } u_0 < u_1 \text{ then } +\infty \text{ else } u_0]$.

Narrowing. After the widening iteration converges to a fixpoint, a subsequent narrowing iteration, as described in [15, 24, 11, 39], aims to enhance the post solution through a downward fixpoint iteration. The narrowing operator Δ satisfies the following properties: (1) $\forall a, a' \in \mathbb{A}, a' \sqsubseteq a \implies a \Delta a' \sqsubseteq a$ and (2) for a decreasing chain $a_1 \sqsupseteq a_2 \sqsupseteq ... \sqsupseteq a_m$, the decreasing chain $a'_1 \sqsupseteq a'_2 \sqsupseteq ... \sqsupseteq a'_m$ where $a'_1 = a_1$ and $a'_m = a'_{m-1} \Delta a_m$ eventually stabilizes. For example, the narrowing operator on the integer interval domain $\mathcal{I}$ is defined as: $[l_0, u_0] \Delta [l_1, u_1] = [\text{if } l_0 = -\infty \text{ then } l_1 \text{ else } l_0, \text{if } u_0 = +\infty \text{ then } u_1 \text{ else } u_0]$.

2.3 Weak Topological Ordering (WTO)

WTO provides an arbitrary update of abstract states rather than strictly following the order of control points. This arbitrary update (also called *chaotic iteration* [20]) is proven not to affect the precision of fixpoint computation while having the potential to accelerate the convergence to the fixpoint [20, 21]. We first give some basic concepts of WTO and then delve into how WTO can be leveraged to enhance the precision of widening and accelerate the *chaotic iteration algorithm*.

▶ **Definition 3** (Hierarchical Ordering). *A hierarchical ordering of a set of elements V is a well-parenthesized permutation of V that does not contain consecutive "(".*

A hierarchical ordering of V induces a total order $\preceq$, a binary relation over the elements of V, where for any $v, w \in V$, either $v \preceq w$ or $w \preceq v$ holds, establishing a linear sequence. A *hierarchical component* is formed by a matching pair of "(" and ")", along with the sub-components or elements in V between them in the specified order. The hierarchical order could also be viewed as a tree, with leaf nodes representing the elements of V, and sub-trees (with non-leaf roots) representing hierarchical components. The first element of a hierarchical component is denoted as *component head*, which is highlighted using double underlining as $\underline{\underline{v}}$. The set of heads of the components containing the element v is denoted by $\omega(v)$. The depth of an element $v \in V$ is defined as $\delta(v) = |\omega(v)|$, which represents the depth of the leaf node corresponding to v in the tree.

▶ **Example 4.** "($\underline{\underline{1}}$ ($\underline{\underline{2}}$ 3))" is a valid hierarchical ordering of elements $\{1, 2, 3\}$. Both "($\underline{\underline{1}}$ ($\underline{\underline{2}}$ 3))" and "($\underline{\underline{2}}$ 3)" are hierarchical components with 1 and 2 being their respective component heads. The orders like "(1 ((2 3))" or "((1 2) 3)" are invalid hierarchical orders. The former is not *well-parenthesized*, and the latter contains consecutive "("s. Components have a hierarchical (nested) structure, meaning that one component can be a sub-component of another. For instance, "($\underline{\underline{2}}$ 3)" is a sub-component of "($\underline{\underline{1}}$ ($\underline{\underline{2}}$ 3))", thus $\omega(3) = \{1, 2\}$ and $\delta(3) = 2$.

▶ **Definition 5** (Weak Topological Ordering). *A weak topological ordering (WTO) of a directed graph $G = (V, E)$ is a hierarchical ordering of control points V such that for each edge $u \to v \in E$, $u \prec v \wedge v \notin \omega(u)$ or $v \preceq u \wedge v \in \omega(u)$. If an edge $u \to v$ satisfies $v \preceq u \wedge v \in \omega(u)$, $u \to v$ is called a back edge.*

▶ **Example 6.** One possible WTO of the intraprocedural control flow graph in Figure 3 is "1 ($\underline{\underline{2}}$ ($\underline{\underline{3}}$ 4) 5) 6". 2 and 3 are the WTO heads and $5 \to 2$ and $4 \to 3$ are back edges. It is clear that WTO precisely captures the nested loop structures within the graph. The inside loop is captured by "($\underline{\underline{3}}$ 4)" while the outside loop is "($\underline{\underline{2}}$ ($\underline{\underline{3}}$ 4) 5)".

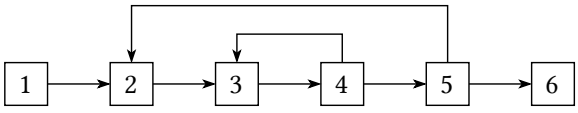


Figure 3 An example of control flow graph.

***WTO*-Guided Chaotic Iteration.** Given the explicit WTO of the control flow points, the chaotic iteration recursively computes a fixed point of the sub-components of each component in WTO every time the component reaches a fixed point [13]. WTO also provides an admissible set of widening points (the heads of WTO components) such that every loop in the program is cut by at least one widening point.

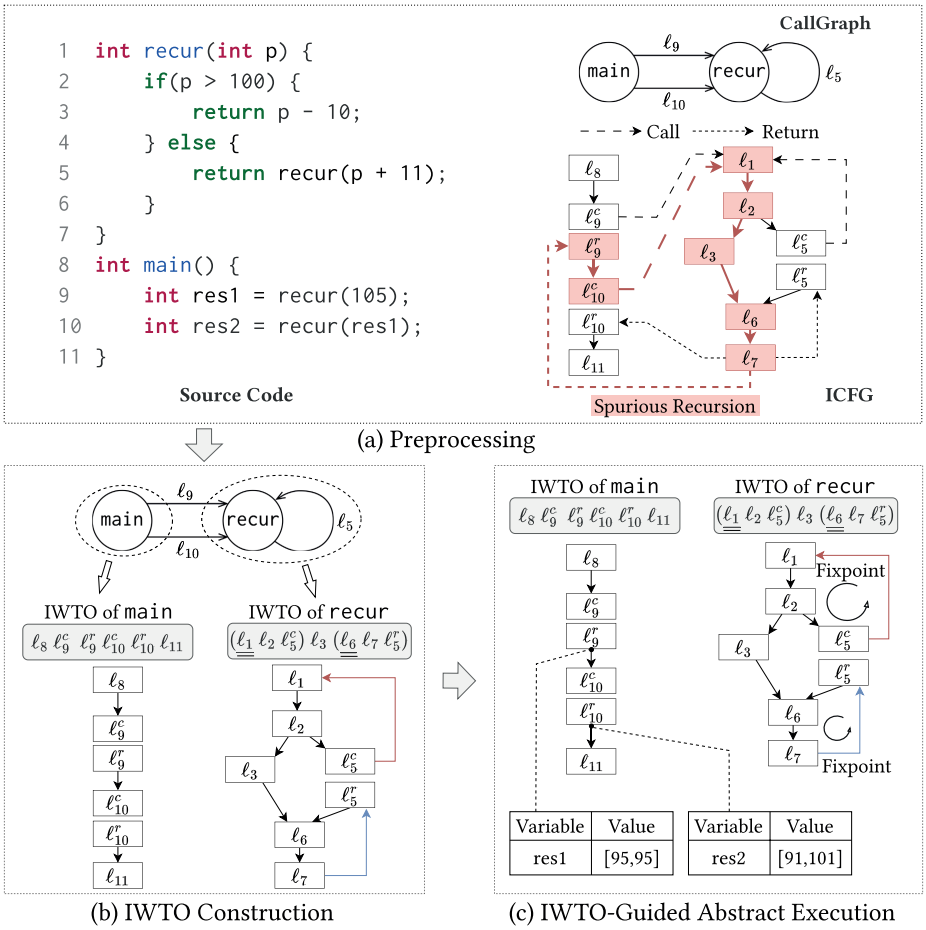


Figure 4 A motivating example by revisiting the example in Figure 1. It provides a detailed illustration of the phases of our framework, as shown in Figure 2.

***WTO* Construction.** WTO is constructed based on Bourdoncle's algorithm [32], which utilizes Tarjan's algorithm [61] to identify all strongly connected components (SCCs) in a directed graph. The algorithm repeatedly applies Tarjan's algorithm on the graph to break each found SCC into smaller SCCs until no further subdivision is possible. Each time an SCC is divided, smaller SCCs are ordered within the larger one in the WTO. The hierarchical structure of these components reflects the partial ordering of the WTO. Current abstract interpretation frameworks [15, 19, 18] compute WTO at the function level, applying the analysis intraprocedurally. However, the interprocedural fixpoint computation order remains implicit, as it is not guided by an interprocedural WTO structure.

3 Motivating Example

Figure 4 illustrates how RecTopo precisely determines the interprocedural fixpoint computation order and widening points by revisiting the example in Figure 1. We use the integer interval domain [17] to reason about its behaviors. The state-of-the-art analyzers, IKOS [15], Clam [19, 34] and CSA [18], all yield an imprecise top value $[-\infty, \infty]$ for `recur`'s return variable `res1` at ℓ_9 and `res2` at ℓ_{10}. In comparison, RecTopo derives a more precise result of $[95, 95]$ for `res1` and $[91, 101]$ for `res2`, demonstrating its precise interprocedural fixpoint computation order and widening strategies.

(a) Preprocessing. As shown in Figure 4(a), we first generate the call graph and interprocedural control flow graph (ICFG) of the analyzed program. In the call graph, we identify the function `recur` as a recursive function, and `main` as the caller. In the recursive function `recur`, we observe two distinct recursive calling paths: one from ℓ_1 to ℓ_5^c and another from ℓ_6 to ℓ_5^r. Note that, the cyclic path highlighted in red in the figure ($\ell_9^r \rightarrow \ell_{10}^c \dashrightarrow \ell_1 \rightarrow \ell_2 \rightarrow \ell_3 \rightarrow \ell_6 \rightarrow \ell_7 \dashrightarrow \ell_9^r$) is a spurious recursive path arising due to the consecutive call to the function `recur` at ℓ_9 and ℓ_{10}.

(b) IWTO Construction. Initially, we apply Tarjan's algorithm to the call graph, identifying two distinct function partitions: `main` and `recur`. Leveraging these partitions, we divide the original ICFG into two sub-ICFGs: one for `main` and another for `recur`. This division eliminates interprocedural edges between `main` and `recur`, as they are not part of the same recursive calling chain, according to the call graph's structure. Consequently, this approach removes the spurious recursion previously depicted in Figure 4(a) and prevents the redundant widening of this spurious recursion, thereby maintaining the precision as inlining-based methods for non-recursive functions. Furthermore, the true recursion within `recur` is comprehensively analyzed, resulting in an interprocedural weak topological ordering (IWTO) of "$(\underline{\underline{\ell_1}}\ \ell_2\ \ell_5^c)\ \ell_3\ (\underline{\underline{\ell_6}}\ \ell_7\ \ell_5^r)$" (where $\underline{\underline{\ell_1}}$ and $\underline{\underline{\ell_6}}$ are WTO heads). This analysis contrasts with approaches like IKOS [15], which typically omit the function body in recursive analyses, or Clam [19, 34] and CSA [18] which only perform widening on ℓ_1 and ℓ_7 without narrowing.

(c) IWTO-Guided Abstract Interpretation. The abstract interpretation illustrated in Figure 4(c) follows the order outlined in the IWTO constructed for the program. The execution begins at the `main` function and progresses through the call sites ℓ_9^c and ℓ_{10}^c. An interleaved widening and narrowing approach is employed to precisely approximate the behaviors of the two IWTO cycles within `recur`. Specifically, at the call site ℓ_9^c, the formal parameter `p` is initialized with the interval $[105, 105]$. As execution enters the first cycle $(\underline{\underline{\ell_1}}\ \ell_2\ \ell_5^c)$, a fixpoint is quickly reached with `p` remaining at $[105, 105]$, given that the path condition for the branch from ℓ_2 to ℓ_5^c is infeasible. Subsequently, at ℓ_3, the return value is calculated as $[95, 95]$. This value is then carried over to the second IWTO cycle $(\underline{\underline{\ell_6}}\ \ell_7\ \ell_5^r)$, where a fixpoint is again promptly achieved, resulting in a final return value of $[95, 95]$ for `res1`, much more precise compared to $[-\infty, \infty]$ for existing state-of-the-art tools [15, 19, 18]. Following this, the second invocation at ℓ_{10}^c uses the value of `res1`, $[95, 95]$, as the input for `p`, and re-engages with the IWTO of `recur`. In the first cycle, the initial widening operation expands `p` to $[95, \infty]$, followed by narrowing that refines the value to $[95, 111]$, which is subsequently propagated to ℓ_3 as $[101, 111]$. Consequently, upon returning to ℓ_{10}^r, the value of `res2` is determined to be $[91, 101]$, demonstrating consistently greater precision than IKOS, Clam and CSA.

4 RecTopo Approach

In this section, we introduce our approach for constructing interprocedural weak topological ordering in §4.1, followed by the algorithm for IWTO-guided abstract interpretation in §4.2. Additionally, we provide the proof of termination for our algorithm in §4.3.

4.1 IWTO Construction

The construction of IWTO consists of two stages. In the first stage (see §4.1.1), we perform program partitioning to divide the program into several function partitions (Definition 7). The second stage (see §4.1.2) involves constructing the IWTO on the ICFG for each resulting function partition from the first stage.

```
1  int main() {
2      recur1(20);
3      recur2(15);
4  }
5  int recur1(int i) {
6      if(i > 10)
7          return recur2(i - 1);
8      else
9          return i;
10 }
```

```
11 int recur2(int j) {
12     if(j > 10)
13         return recur1(j - 1);
14     else
15         return id(j);
16 }
17 // identity function
18 int id(int k) {
19     return k;
20 }
```

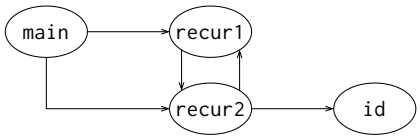


Figure 5 A recursion example and its call graph.

4.1.1 Program Partitioning

This stage aims to decompose the input whole program into several function partitions (see Definition 7) and identify all function partition entries (see Definition 8). The process of function partitioning segments the whole program analysis into manageable parts while ensuring that the functions within each recursive call are analyzed collectively and precisely. The identification of function partition entries is critical to guarantee that each IWTO is accurately constructed starting from the entry point as dictated by the program's call sites.

▶ **Definition 7** (Function Partition). *A function partition, denoted as* `FPar`*, consists of a set of functions that are directly or indirectly reachable from each other on the call graph.*

We apply Tarjan's algorithm [61] to the pre-built call graph of the analyzed program, identifying all strongly connected components (SCCs). Each SCC on the call graph corresponds to a distinct function partition. Non-recursive functions, being single-node SCCs, each forms an individual partition. For recursive functions, those within the same calling chain are aggregated into a single partition. Note that if a recursive function calls a non-recursive function, they are placed in separate partitions because the non-recursive function does not call back, meaning that they do not meet the criteria of mutual reachability required for placement in the same partition (Definition 7).

▶ **Definition 8** (Function Partition Entry). *For a function partition* `FPar`*, a function* $\texttt{f} \in \texttt{FPar}$ *is considered a function partition entry if and only if there exists a function* $\texttt{f}' \notin \texttt{FPar}$ *such that* $\texttt{f}'$ *calls* `f`*. In other words,* `f` *is an entry point to the partition* `FPar` *if it is invoked by at least one function outside of the partition.*

Function Partition Entries Identification. For each function `f` within a function partition `FPar`, we examine the predecessors of `f` on the call graph. If any predecessor $\texttt{f}'$ is found not to belong to `FPar`, then `f` is designated as a partition entry for that partition.

▶ **Example 9.** We illustrate the process of program partitioning with an example and its call graph in Figure 5. This example includes a mutual recursion between the functions `recur1` and `recur2`, invoked by `main` at locations ℓ_2 and ℓ_3, respectively. Additionally, the function `id` is called by `recur2` at location ℓ_{15}. Applying Tarjan's algorithm to this call graph results

Algorithm 1 IWTO construction algorithm.

```
   Input: G(V, E): ICFG of the program
          EMap: map from function partition to its partition entries
   Output: IMap: map from partition entry to IWTO
1  Function constructIMap(EMap, G):
2      IMap := {};
3      foreach (FPar → FParEns) ∈ EMap do
4          constructIWTO(FPar, FParEns, G, IMap);
5      return IMap;
6  Function constructIWTO(FPar, FParEns, G, IMap):
7      G_FPar(V_FPar, E_FPar) := ICFGPartition(G, FPar);          // Step 1
8      foreach v ∈ V_FPar do                                      // Step 2
9          if v is a non-recursive call site then
10             E_FPar := E_FPar ∪ (v, v.returnSite());
11     foreach entry ∈ FParEns do                                 // Step 3
12         entryICFGNode := Entry ICFG Node of entry;
13         IMap[entry] := Bourdoncle(G_FPar, entryICFGNode);
```

in three SCCs: {main}, {recur1, recur2}, and {id}. Each SCC corresponds to a distinct function partition. For the function partition {recur1, recur2}, both functions are identified as partition entries, as they are separately called by main at different locations.

4.1.2 IWTO Construction

For each function partition entry, we establish an interprocedural weak topological ordering (IWTO). As shown in Algorithm 1, the main function constructIMap (Lines 1-5) processes all function partitions of the analyzed program, alongside EMap, which maps each function partition to its function partition entries and aims to produce IMap, which maps each function partition entry to its respective IWTO. The subroutine constructIWTO defined at Lines 6-13 aims to construct IWTO for a specific function partition FPar and its associated entries FParEns. constructIWTO first extracts the subgraph corresponding to FPar on the ICFG, and then completes the extracted subgraph by connecting non-recursive calls and returns (Definition 10). Finally, based on the completed subgraph, it constructs an IWTO for each function partition entry and stores the results in IMap. The detailed steps of constructIWTO are outlined as follows:

Step 1: ICFG Partitioning. Given a function partition FPar, this step (Line 7) aims to extract the induced sub-ICFG corresponding to FPar from the ICFG of the entire input program. The resulting sub-ICFG is composed exclusively of ICFG nodes that are contained within a specific function from FPar.

▶ **Definition 10** (Recursive and Non-Recursive Call/Return Site). *A call site* $\ell^c\langle$caller, callee$\rangle$ *in the ICFG is defined as a recursive call site iff.* caller *and* callee *are in the same function partion. Else,* ℓ^c *is classified as a non-recursive call site. Each* ℓ^c *is associated with a corresponding return site* ℓ^r*, which facilitates the return from the callee function.*

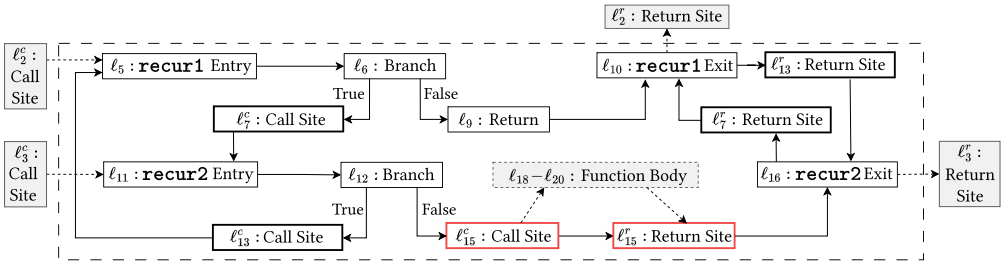


Figure 6 The sub-ICFG of function partition `recur1` and `recur2` by revisiting the example in Figure 5.

▶ **Example 11.** Figure 6 shows the sub-ICFG of function partition {`recur1`, `recur2`} for the code example in Figure 5. The call sites ℓ_7^c and ℓ_{13}^c are *recursive call sites* because their callee functions are recursive, and these calls occur within recursive functions (`recur1` and `recur2`). By comparison, the call sites ℓ_2^c and ℓ_3^c are considered *non-recursive* as they are located in the `main` function, which is not part of the recursion involving `recur1` or `recur2`. ℓ_{15}^c is also a *non-recursive call site* because its callee, `id`, is not a recursive function.

Step 2: Sub-ICFG Completion. This step (Lines 8-10) ensures that every node within the sub-ICFG extracted in Step 1 remains reachable from the entry of its corresponding function. During the ICFG partitioning in Step 1, all non-recursive return sites (Definition 10) become unreachable. This occurs because each non-recursive return site is initially linked to its corresponding call site via the callee function body; however, the callee function is excluded from the sub-ICFG because it does not belong to the current function partition. Therefore, we introduce an additional edge from each non-recursive call site directly to its respective return site.

Step 3: *IWTO* Construction. In this step (Lines 11-13), we employ Bourdoncle's algorithm [32] to build the IWTO for the completed sub-ICFG derived from Step 2. For each partition entry, denoted as `entry`, Bourdoncle's algorithm begins at the function entry node associated with the function partition entry, referred to as `entryICFGNode` in Algorithm 1. It then constructs an IWTO across the entire sub-ICFG, incorporating its interprocedural edges.

▶ **Example 12.** Let us revisit the example in Figure 5 and illustrate the three steps in Algorithm 1 used to construct the IWTO of the function partition {`recur1`, `recur2`}. In Step 1, we obtain an induced sub-ICFG shown in Figure 6, where call and return sites are delineated with bold outlines. Non-recursive call/return sites are denoted by ▭, whereas recursive call/return sites are indicated by ▭. In Step 2, since the ICFG nodes of `id` are excluded from the sub-ICFG in Step 1, the return site ℓ_{15}^r becomes unreachable from its call site. Consequently, we add an edge from the call site ℓ_{15}^c to its return site, ensuring reachability from the function entry within the sub-ICFG. Finally, in Step 3, we apply Bourdoncle's algorithm to the partition entries `recur1` and `recur2`, starting from their respective entry ICFG nodes, ℓ_5 and ℓ_{11}. Specifically, initiating from ℓ_5, we establish the IWTO as: "$(\underline{\underline{\ell_5}}\ \ell_6\ \ell_7^c\ \ell_{11}\ \ell_{12}\ \ell_{13}^c)\ \ell_9\ \ell_{15}^c\ \ell_{15}^r\ (\underline{\underline{\ell_{10}}}\ \ell_{13}^r\ \ell_{16}\ \ell_7^r)$". The IWTO starting from ℓ_{11} is outlined as: "$(\underline{\underline{\ell_{11}}}\ \ell_{12}\ \ell_{13}^c\ \ell_5\ \ell_6\ \ell_7^c)\ \ell_9\ \ell_{15}^c\ \ell_{15}^r\ (\underline{\underline{\ell_{16}}}\ \ell_7^r\ \ell_{10}\ \ell_{13}^r)$".

4.2 IWTO-Guided Abstract Interpretation

This section first presents the overall algorithm for fixpoint computation in §4.2.1, followed by an example set of feasible node-level interpretation rules that can be applied during the fixpoint computation in §4.2.2.

4.2.1 Abstract Interpretation Algorithm

Algorithm 2 presents the pseudo-code for IWTO-guided abstract interpretation, initialized by the `chaoticIteration` function (Line 1). This algorithm takes two inputs: a map linking each partition entry to its corresponding IWTO (`IMap`), as described in §4.1.2, and the program's entry function, `root` (e.g., the `main` function). Starting from `root`, which is a function partition entry (Definition 8), the function `handlePartition` iterates through the elements in its IWTO (Lines 5-9), processing each as either a single node (Line 7) or a cycle (Line 9), where cycles in the IWTO may represent either intraprocedural loops or recursive calls. For handling cycles, we apply the interleaved widening and narrowing strategy [10] to compute a fixpoint that considers both efficiency, achieved through widening, and precision, refined by narrowing.

The subroutine `handleNode`, detailed from Line 10 to Line 13 in Algorithm 2, processes individual nodes. At Line 11, the procedure begins by updating the abstract state node-wise, based on the semantics of the ICFG node. If this node is identified as a non-recursive call site (as defined in Definition 10), `handlePartition` is recursively invoked to process the corresponding callee function. Recursive call and return sites are treated similarly to normal intraprocedural nodes, as their analysis is integrated within the IWTO.

The algorithm for computing a fixpoint for a cycle is shown in the `stabilizeCycle` subroutine, delineated from Line 14 to Line 32 in Algorithm 2. It employs a while loop to continuously process the IWTO elements constituting the cycle. As mentioned in §2.3, each iteration of the loop begins by processing the component head of the IWTO using either widening or narrowing operations, outlined in Lines 18-27, followed by the process other IWTO components (Lines 28-32). The cycle commences with iterations where the widening operation is applied first (Lines 19-23) to the head node, aiming to accelerate the stabilization process. Once the widening has reached a fixpoint, where no changes in the abstract state are observed from one iteration to the next, the algorithm transitions to the narrowing phase (Lines 24-27). This phase refines the analysis's precision, following the methodologies proposed in [10]. The loop concludes when the narrowing phase also achieves a fixpoint, determined at Line 26, where a consistent abstract state is maintained post-narrowing from one loop iteration to the next.

4.2.2 Node-Level Interpretation Rules

Figure 7 demonstrates an example set of rules for handling each IWTO node, which can be applied at Line 11 in Algorithm 2. Note that these rules can be customized to perform other types of abstract interpretations, such as using different abstract domains or object sensitivity.

The interpretation of each node encompasses two fundamental steps: first, aggregating post-abstract states from predecessors to derive its pre-abstract state; second, transforming the pre-abstract state according to program semantics to obtain the post-abstract state. For interprocedural analysis, abstract states are aggregated via CallFlow and RetFlow rules, while intraprocedural states are gathered through the Sequence rule. Notably, CallFlow employs selective aggregation of abstract states from call sites: it blocks propagation from mismatching non-recursive call sites while allowing propagation from recursive call sites.

Algorithm 2 IWTO-guided abstract interpretation.

```
Input: IMap: map from partition entry to IWTO
       root: program entry function
Output: Abstract states of the program
1  Function chaoticIteration():                                  // Algorithm entry
2      handlePartition(dummyCall⟨_, root⟩);
3  Function handlePartition(callSite⟨caller, callee⟩):
4      IWTO := IMap[callee];
5      foreach elem ∈ IWTO do
6          if elem is a single node then
7              handleNode(callSite, elem);
8          else
9              stabilizeCycle(callSite, elem);
10 Function handleNode(pEntryCallsite, node):
11     Update abstract state of node;
12     if node is a non-recursive call site then
13         handlePartition(node);                                 // Inline
14 Function stabilizeCycle(pEntryCallsite, cycle):
15     head := cycle.getHead();                                   // Cycle head
16     increasing := true;
17     while true do
18         handleNode(pEntryCallsite, head);
19         if increasing then
20             Widen abstract state of head;                      // Widening
21             if abstract state of head not increasing then
22                 increasing := false;
23                 continue;
24         else
25             Narrow abstract state of head;                     // Narrowing
26             if abstract state of head not decreasing then
27                 break;
28         foreach elem ∈ cycle.getBody() do
29             if elem is a single node then
30                 handleNode(pEntryCallsite, elem);
31             else
32                 stabilizeCycle(pEntryCallsite, elem);
```

$$[\textsc{CallFlow}]\ \frac{\ell@\texttt{caller} \hookrightarrow \ell'@\texttt{callee}}{\sigma_{\overline{\ell'}} \sqsupseteq \texttt{filterCall}(\sigma_{\underline{\ell}}, \ell)} \qquad [\textsc{RetFlow}]\ \frac{\ell@\texttt{caller} \hookleftarrow \ell'@\texttt{callee}}{\sigma_{\overline{\ell}} \sqsupseteq \sigma_{\underline{\ell'}}}$$

$$[\textsc{Sequence}]\ \frac{\ell \overset{\texttt{cond}}{\hookrightarrow} \ell'}{\sigma_{\overline{\ell'}} \sqsupseteq \texttt{refineSeq}(\sigma_{\underline{\ell}}, \texttt{cond})} \qquad [\textsc{Cons}]\ \frac{\ell : \mathtt{p} = \mathtt{c}}{\sigma_{\underline{\ell}} := \sigma_{\overline{\ell}}[\mathtt{p} \mapsto \langle \alpha(\{\mathtt{c}\}), \top\rangle]}$$

$$[\textsc{Copy}]\ \frac{\ell : \mathtt{p} = \mathtt{q}}{\sigma_{\underline{\ell}} := \sigma_{\overline{\ell}}[\mathtt{p} \mapsto \sigma_{\overline{\ell}}(\mathtt{q})]} \qquad [\textsc{Phi}]\ \frac{\ell : \mathtt{r} = \phi(\mathtt{p}_1, \mathtt{p}_2, ..., \mathtt{p}_n)}{\sigma_{\underline{\ell}} := \sigma_{\overline{\ell}}[\mathtt{r} \mapsto \bigsqcup_{i=1}^{n} \sigma_{\overline{\ell}}(\mathtt{p}_i)]}$$

$$[\textsc{Unary}]\ \frac{\ell : \mathtt{p} = \neg \mathtt{q}}{\sigma_{\underline{\ell}} := \sigma_{\overline{\ell}}[\mathtt{p} \mapsto \neg^{\#} \sigma_{\overline{\ell}}(\mathtt{q})]} \qquad [\textsc{Binary}]\ \frac{\ell : \mathtt{r} = \mathtt{p} \odot \mathtt{q}}{\sigma_{\underline{\ell}} := \sigma_{\overline{\ell}}[\mathtt{r} \mapsto \sigma_{\overline{\ell}}(\mathtt{p}) \odot^{\#} \sigma_{\overline{\ell}}(\mathtt{q})]}$$

$$[\textsc{Addr}]\ \frac{\ell : \mathtt{p} = \&\mathtt{o}}{\sigma_{\underline{\ell}} := \sigma_{\overline{\ell}}[\mathtt{p} \mapsto \langle \top, \{o_i\}\rangle]}$$

$$[\textsc{Gep}]\ \frac{\ell : \mathtt{p} = \&(\mathtt{q} \to \mathtt{idx}) \mid \mathtt{p} = \&\mathtt{q}[\mathtt{idx}]}{\sigma_{\underline{\ell}} := \sigma_{\overline{\ell}}[\mathtt{p} \mapsto \bigsqcup_{o \in \gamma(\sigma_{\overline{\ell}}(\mathtt{q}))} \bigsqcup_{j \in \gamma(\sigma_{\overline{\ell}}(\mathtt{idx}))} \langle \top, \{\mathtt{o.fld}_j\}\rangle]}$$

$$[\textsc{Load}]\ \frac{\ell : \mathtt{p} = *\mathtt{q}}{\sigma_{\underline{\ell}} := \sigma_{\overline{\ell}}[\mathtt{p} \mapsto \bigsqcup_{o \in \gamma(\sigma_{\overline{\ell}}(\mathtt{q}))} \sigma_{\overline{\ell}}(o)]}$$

$$[\textsc{Store}]\ \frac{\ell : *\mathtt{p} = \mathtt{q}}{\sigma_{\underline{\ell}} := (\{o \mapsto \sigma_{\overline{\ell}}(\mathtt{q}) | o \in \gamma(\sigma_{\overline{\ell}}(\mathtt{p}))\} \sqcup \sigma_{\overline{\ell}} \setminus \mathsf{kill}(\ell))}$$

$$\mathsf{filterCall}(\sigma_{\underline{\ell}}, \ell) := \begin{cases} \sigma_{\underline{\ell}} & \text{if } \ell \text{ is recursive call site} \vee \ell = \texttt{pEntryCallsite} \\ \bot & \text{otherwise} \end{cases}$$

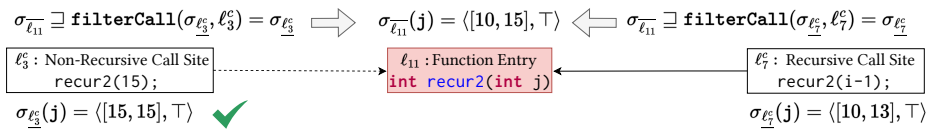
$$\mathsf{refineSeq}(\sigma_{\underline{\ell}}, \texttt{cond}) := \sigma_{\underline{\ell}}[\mathtt{p}_1 \mapsto \sigma_{\underline{\ell}}(\mathtt{p}_1) \sqcap \alpha(\texttt{cond}(\mathtt{p}_1)), \ldots, \mathtt{p}_n \mapsto \sigma_{\underline{\ell}}(\mathtt{p}_n) \sqcap \alpha(\texttt{cond}(\mathtt{p}_n))]$$

$$\mathsf{kill}(\ell : *\mathtt{p} = \mathtt{q}) := \begin{cases} \{\mathtt{o} \mapsto _ \mid \mathtt{o} \in \gamma(\sigma_{\overline{\ell}}(\mathtt{p}))\} & \text{if } \sigma_{\overline{\ell}}(\mathtt{p}) = \langle \top, \{\mathtt{o}\}\rangle \wedge \mathtt{o} \text{ is singleton} \\ \{\mathtt{o} \mapsto _ \mid \mathtt{o} \in \mathcal{O}\} & \text{if } \sigma_{\overline{\ell}}(\mathtt{p}) = \langle \top, \varnothing\rangle \\ \varnothing & \text{otherwise} \end{cases}$$

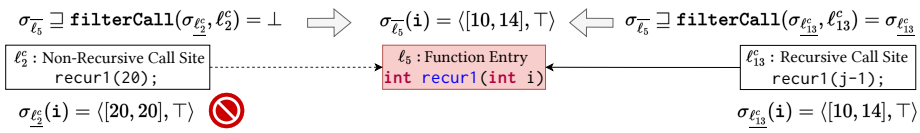
Figure 7 Rules for interpreting each node during abstract interpretation (Line 11 in Algorithm 2). $\ell@\texttt{caller} \hookrightarrow \ell'@\texttt{callee}$ and $\ell@\texttt{caller} \hookleftarrow \ell'@\texttt{callee}$ represent interprocedural control flows. The former indicates a flow from the CALL statement ℓ in `caller` to the entry ℓ' of the `callee` function, while the latter denotes a flow from the exit ℓ' of the `callee` back to the RETURN statement ℓ in `caller`. $\ell \overset{\texttt{cond}}{\hookrightarrow} \ell'$ denotes an intraprocedural control flow with path condition `cond`, whose value is null for unconditional flows. $a \sqsupseteq b$ represents $a := a \sqcup b$. $\mathtt{f}^{\#}$ is the abstract operator concerning the concrete operator `f`. $\alpha(\texttt{cond}(\mathtt{p}_i))$ abstracts the concrete values of $\mathtt{p}_i$ restricted by sequence condition `cond` based on Definition 1. $\mathtt{o.fld}_j$ represents the j^{th} field of the base object `o`.

▶ **Example 13.** Figure 8 illustrates this selective aggregation using the example in Figure 6, demonstrating the application of the CALLFLOW rule in deriving pre-abstract states for ℓ_{11} and ℓ_5. Consider the scenario where abstract interpretation progresses to invoking $\texttt{handlePartition}(\ell_3^c\langle\texttt{main}, \texttt{recur2}\rangle)$ to analyze the function partition $\{\texttt{recur1}, \texttt{recur2}\}$ from partition entry `recur2`. When deriving the pre-abstract state of ℓ_{11}, as shown in Figure 8a, both $\sigma_{\underline{\ell_7^c}}$ and $\sigma_{\underline{\ell_3^c}}$ are permitted to propagate. This is because ℓ_7^c is a recursive call site, whereas ℓ_3^c, while non-recursive, matches the current partition entry's call site, denoted as `pEntryCallsite` in Algorithm 2 at Line 10. Conversely, as shown in Figure 8b, when deriving the pre-abstract state of ℓ_5, $\sigma_{\underline{\ell_2^c}}$ is blocked because ℓ_2^c does not match `pEntryCallsite`, ℓ_3^c, whereas $\sigma_{\underline{\ell_{13}^c}}$ propagates due to the recursive nature of ℓ_{13}^c.

The RETFLOW rule facilitates the propagation of abstract states from a callee function's exit node to the corresponding return site. The SEQUENCE rule refines abstract values based on control flow edge conditions. Figure 9 demonstrates this refinement process for deriving the pre-abstract state of ℓ_{13}^c. Since the incoming edge from ℓ_{12} requires condition `i > 10` to hold, we refine the abstract value of `i` in $\sigma_{\underline{\ell_{12}}}$ by computing its meet with $[11, \infty]$.



(a) Deriving the pre-abstract state $\sigma_{\overline{\ell_{11}}}$ during running $\texttt{handlePartition}(\ell_3^c\langle \texttt{main}, \texttt{recur2} \rangle)$.



(b) Deriving the pre-abstract state $\sigma_{\overline{\ell_5}}$ during running $\texttt{handlePartition}(\ell_3^c\langle \texttt{main}, \texttt{recur2} \rangle)$.

Figure 8 Illustration of the CallFlow rule for deriving pre-abstract state of ℓ_{11} and ℓ_5 by revisiting the example in Figure 6.

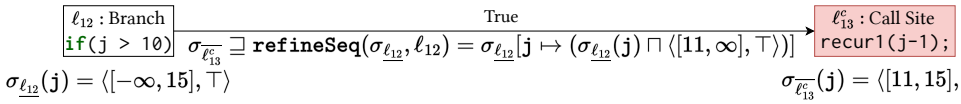


Figure 9 Illustration of the Sequence rule for deriving the pre-abstract state $\sigma_{\overline{\ell_{13}^c}}$ by revisiting the example in Figure 6.

The post-abstract state $\sigma_{\underline{\ell}}$ of node ℓ is derived from its pre-abstract state $\sigma_{\overline{\ell}}$ and the relevant statement. For pointer-free statements (Cons, Copy, Phi, Unary, Binary, Call and Return), we calculate the abstract value of the left-hand-side variable based on the right-hand-side values and relevant operators. For instance, given a Binary statement $\ell : \texttt{r} = \texttt{p} + \texttt{q}$ and $\sigma_{\overline{\ell}}(\texttt{p}) := \langle [1, 3], \top \rangle, \sigma_{\overline{\ell}}(\texttt{q}) := \langle [4, 7], \top \rangle$, we derive $\sigma_{\underline{\ell}}(\texttt{r}) := \langle [5, 10], \top \rangle$. Particularly, since Call and Return statements involve parameter and return value passing – essentially top-level variable assignments – we handle each assignment using the Copy rule for translation. For pointer-related statements (Gep, Load and Store), we utilize abstract values across interval and address domains to derive the post-abstract state. To elaborate, for Gep, we join all potential address values formed by adding a base address $\texttt{o} \in \gamma(\sigma_{\overline{\ell}}(\texttt{p}))$ with an offset $\texttt{j} \in \gamma(\sigma_{\overline{\ell}}(\texttt{idx}))$ to derive the conservative address value for the left-hand-side variable. As for Load, we join the abstract values $\sigma_{\overline{\ell}}(\texttt{o})$ of all possible addresses $\texttt{o} \in \gamma(\sigma_{\overline{\ell}}(\texttt{q}))$ to form the conservative abstract value of the left-hand-side variable. Lastly, for Store, we update the abstract value of every possible address $\texttt{o} \in \gamma(\sigma_{\overline{\ell}}(\texttt{p}))$. Particularly, when there are multiple addresses $\texttt{o} \in \gamma(\sigma_{\overline{\ell}}(\texttt{p}))$, we join the abstract value $\sigma_{\overline{\ell}}(\texttt{q})$ with each existing abstract value $\sigma_{\overline{\ell}}(\texttt{o})$ to form o's new abstract value conservatively. In contrast, when there is only one $\texttt{o} \in \gamma(\sigma_{\overline{\ell}}(\texttt{p}))$, it is safe to directly use $\sigma_{\overline{\ell}}(\texttt{q})$ as o's new abstract value, because the store must occur at o.

▶ **Example 14.** We revisit the example in Figure 5 to illustrate the IWTO-guided abstract interpretation. It starts with $\texttt{handlePartition}(\ell_0\langle _, \texttt{main} \rangle)$. The IWTO of main function is outlined as "$\ell_1\ \ell_2^c\ \ell_2^r\ \ell_3^c\ \ell_3^r\ \ell_4$". Both ℓ_2^c and ℓ_3^c are non-recursive call sites, which require invoking handlePartition to handle their callee functions. Take ℓ_3^c as an example. $\texttt{handlePartition}(\ell_3^c\langle \texttt{main}, \texttt{recur2} \rangle)$ follows the IWTO of function partition {recur1, recur2}, which is outlined as "$(\underline{\underline{\ell_{11}}}\ \ell_{12}\ \ell_{13}^c\ \ell_5\ \ell_6\ \ell_7^c)\ \ell_9\ \ell_{15}^c\ \ell_{15}^r\ (\underline{\underline{\ell_{16}}}\ \ell_7^r\ \ell_{10}\ \ell_{13}^r)$", to perform abstract interpretation. The first element of the IWTO, "$(\underline{\underline{\ell_{11}}}\ \ell_{12}\ \ell_{13}^c\ \ell_5\ \ell_6\ \ell_7^c)$", is a cycle, with ℓ_{11} being the head. To handle the cycle, the function stabilizeCycle is

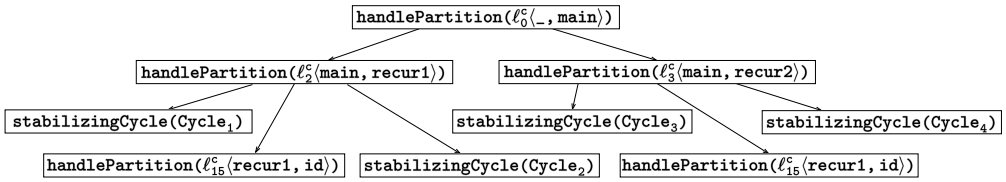


Figure 10 Recursion tree of algorithm 2 executing on the program presented in Figure 5.

invoked, which iteratively updates the abstract states of the nodes within the cycle. The iteration involves two stages, the widening stage and the narrowing stage. We elaborate on the widening and narrowing stages using the updating process of $\sigma_{\overline{\ell_{11}}}(\mathtt{j})$, as ℓ_{11} is the only widening point of the cycle. During the widening stage, it is initialized as $\langle[15, 15], \top\rangle$ in the first iteration. In the second iteration, it is first updated to $\langle[13, 15], \top\rangle$, and then widened to $\langle[-\infty, 15], \top\rangle$ (refer back to §2.2). The iteration then proceeds to the narrowing stage, during which $\sigma_{\overline{\ell_{11}}}(\mathtt{j})$ is updated to $\langle[10, 15], \top\rangle$, refining the value obtained in the widening stage. This refinement benefits from the adjustments made in the SEQUENCE rule while deriving the pre-abstract state of ℓ_7^c. With the conclusion of the narrowing stage, `stabilizeCycle` finalizes, and the abstract interpreter continues to address the remainder of the IWTO. After finalizing its handling of the entire IWTO, the abstract interpreter proceeds to addressing ℓ_3^r and concludes that the return value of this recursion, entering from ℓ_3^c, is exactly 10. This exact value also benefits from the refinement conducted by the SEQUENCE rule while deriving the pre-abstract state of ℓ_{15}^c.

4.3 Termination Analysis

This section aims to prove that Algorithm 2 is guaranteed to terminate, even in the presence of nested recursions and loops in the input program. Algorithm 2 begins with `chaoticIteration`, which calls `handlePartition` to handle the dummy call site to `root`, the program entry function. It then enters a mutual recursion among three functions: `handlePartition`, `handleNode` and `stabilizeCycle`. Due to the simplicity of `handleNode`, we inline it at the invoking sites in Lines 7 and 30 for the convenience of the proof. In order to model the recursion described in Algorithm 2, we introduce the concept of *recursion tree*.

▶ **Definition 15** (Recursion Tree). *The recursion tree, denoted as a triple $T = (N, E, r)$, models the invocation process of a running algorithm. The set N is non-empty and consists of nodes, where each $n \in N$ represents an instance of a function invocation with specific parameters during the recursive process. The set $E \subseteq N \times N$ comprises edges, where an edge $(n_1, n_2) \in E$ indicates that function n_1 calls n_2 during the recursion. The element $r \in N$ denotes the root node of T, representing the initial function that starts the recursive calls.*

▶ **Example 16.** Figure 10 demonstrates the recursion tree of Algorithm 2 as it executes on the program presented in Figure 5. The root node indicates that the recursion of Algorithm 2 begins with `handlePartition`($\ell_0^c\langle$_, `main`$\rangle$), where $\ell_0^c\langle$_, `main`$\rangle$ denotes a dummy call to the program entry, `main`. Additionally, the two edges of the root node indicates that it invokes each child subroutine once during execution.

Since all operations in the functions `handlePartition` and `stabilizeCycle` are trivial and guaranteed to terminate, the overall termination of the algorithm can be reduced to the termination of the recursive process. This is equivalent to stating that the recursion tree of Algorithm 2 contains a finite number of nodes for any input.

▶ **Lemma 17.** *The recursion tree of Algorithm 2 on any input has finite height.*

Proof. Consider a path (sequence of nodes) starting from the root node in the recursion tree. We will show that the number of nodes along this path is bounded by analyzing the invocations of two specific functions: `handlePartition` (referred as P nodes) and `stabilizeCycle` (reffered as S nodes). These are the only types of nodes present on the path, as we have inlined all `handleNode` invokations for simplicity.

First, we examine the subsequence of `handlePartition` calls, denoted as:

$$P_1(\ell_1^c\langle \texttt{caller}_1, \texttt{callee}_1\rangle), \ldots, P_m(\ell_m^c\langle \texttt{caller}_m, \texttt{callee}_m\rangle).$$

We use $FP(\texttt{f})$ to denote the unique function partition to which function `f` belongs. It follows that for all $i \in [1, m-1]$, $FP(\texttt{callee}_i) = FP(\texttt{caller}_{i+1})$. Consequently, for all $i < j$, $\texttt{caller}_j$ is reachable from $\texttt{caller}_i$ on the call graph. We aim to prove the following sub-conclusions. **Sub-conclusion 1**: For all $i \leq j$, $FP(\texttt{caller}_i) \neq FP(\texttt{callee}_j)$. Suppose this were not the case. Then, there would exists $i \leq j$ such that $FP(\texttt{caller}_i) = FP(\texttt{callee}_j)$, implying that $\texttt{caller}_i$ is reachable from $\texttt{callee}_j$. Furthermore, it follows that $\texttt{caller}_j$ would also be reachable from $\texttt{callee}_j$, which contradicts the definition of $\texttt{v}_j\langle \texttt{caller}_j, \texttt{callee}_j\rangle$ as a non-recursive call site (according to Definition 10). **Sub-conclusion 2**: For all $i < j$, $FP(\texttt{caller}_i) \neq FP(\texttt{caller}_j)$. This follows directly from Sub-conclusion 1 and $FP(\texttt{callee}_i) = FP(\texttt{caller}_{i+1})$. If $FP(\texttt{caller}_i) = FP(\texttt{caller}_j)$ for some $i < j$, it would contradict Sub-conclusion 1. **Sub-conclusion 3**: $m \leq |FPars|$, where $|FPars|$ denotes the number of function partitions in the program. If this were false, then there would exist at least one pair $i < j$ such that $FP(\texttt{caller}_i) = FP(\texttt{caller}_j)$, which would contradict Sub-conclusion 2.

Second, we examine a *continuous* subsequence of `stabilizeCycle` invocations,

$$S_1(\texttt{cycle}_1), \ldots, S_n(\texttt{cycle}_n).$$

For this sequence, the inequality $n < \texttt{cycle}_1\texttt{.depth}$ is established, where $\texttt{cycle}_1\texttt{.depth}$ denotes the depth of the $\texttt{cycle}_1$. This holds because $\texttt{cycle}_{i+1}$ is a sub-IWTO component of $\texttt{cycle}_i$. Furthermore, the number of continuous S subsequences is limited to length of the longest P subsequence on the path, which, as previously concluded, is $|FPars|$. Consequently, the total number of S nodes on the path is limited to $|FPars| \times maxCycleDepth$, where $maxCycleDepth$ is the maximum depth of all IWTO cycles.

Finally, since the path consists solely of P and S nodes, the length of the path is less than $|FPars| \times (maxWTOdepth + 1)$. ◀

▶ **Lemma 18.** *The recursion tree of Algorithm 2 for any input has finite degree.*

Proof. To establish this, we consider two cases. **Case 1:** For a $P(\ell^c\langle \texttt{caller}, \texttt{callee}\rangle)$ node, its degree is limited by the number of elements in `callee`'s corresponding IWTO. **Case 2:** For a $S(\texttt{cycle})$ node, its child nodes represent the invoked subroutines within the double-nested loop outlined in Algorithm 2 from Lines 17 to 32. The outer loop is guaranteed to terminate within h rounds [13], where h represents the height of the lattice when applying the widening operation. The inner loop is bounded by the number of elements in `cycle`. Thus, the degree of an S node is also finite.

In both cases, the degree of the nodes are limited, confirming that the recursion tree has finite degree. ◀

▶ **Theorem 19** (Termination)**.** *Algorithm 2 is guaranteed to terminate.*

Table 2 Statistics of 10 open-source projects. *#LOI* denotes the number of lines of LLVM instructions. *#Method*, *#Call* and *#Obj* are the numbers of functions, method calls and memory objects, respectively. $|V|$ and $|E|$ are the numbers of ICFG nodes and ICFG edges.

Project	*#LOI*	*#Method*	*#Call*	*#Obj*	$\|V\|$	$\|E\|$
`paste`	8,416	53	758	510	9,395	9,922
`md5sum`	11,483	63	881	606	12,494	13,064
`YAJL`	20,592	151	561	208	9,253	9,922
`8cc`	30,609	480	2,349	3,267	15,984	25,294
`libplist`	35,324	342	1,743	1,635	13,697	20,978
`MP4v2`	39,178	601	610	1,991	15,595	16,733
`RIOT`	54,597	579	1,614	951	20,176	20,843
`darknet`	210,117	1,004	9,861	2,614	110,424	140,213
`tmux`	446,626	1,967	22,369	3,879	162,879	178,924
`Teeworlds`	529,737	2,306	20,701	5,754	251,356	246,029
Total	1,386,679	7,546	61,447	21,415	621,253	681,922

Proof. By Lemmas 17 and 18, both the height and the breadth of the recursion tree are limited. Consequently, the recursion tree contains a finite number of nodes, which implies the termination of Algorithm 2. ◀

5 Evaluation

In this section, we evaluate the performance of RecTopo, focusing on its precision improvement in abstract interpretation and its scalability when applied to large-scale programs. We conduct a comparative analysis between RecTopo and three state-of-the-art abstract interpreters, IKOS [15], Clam [19, 34] and CSA [18], in an assertion-checking client, buffer overflow detection.

Additionally, we perform an ablation study to assess the influence of the IWTO implementation on the overall effectiveness of RecTopo.

Our empirical evaluation addresses the following research questions:

RQ1 **Analysis Effectiveness:** How effective is RecTopo in detecting buffer overflow vulnerabilities compared to state-of-the-art tools when evaluated on standard NIST benchmarks?

RQ2 **Real-world Applicability:** To what extent can RecTopo scale to analyze real-world applications while maintaining its precision in bug detection?

RQ3 **Impact of IWTO:** What is the contribution of the IWTO technique to RecTopo's overall performance, particularly regarding precision improvements and computational overhead?

5.1 Datasets and Implementation

Datasets. We evaluate RecTopo using (1) a benchmark composed of 8312 programs from NIST [46], which includes comprehensive test cases of buffer overflow vulnerabilities, with 800 cases containing recursions, and (2) 10 popular open-source C/C++ projects across various application domains, with their statistics detailed in Table 2. These projects include `paste` [16] (file merger), `md5sum` [16] (file verifier), `YAJL` [4] (JSON parsing library),

Table 3 Comparison of tools using the NIST benchmark, with the recall and precision in percentages (%). "NIST (entire)" refers to the entire dataset, while "NIST (recursion)" specifies the test cases that have recursions.

Dataset	Metric	IKOS	Clam	CSA	RecTopo
NIST (entire)	Recall (%)	53.90	55.29	80.83	**80.83**
	Precision (%)	46.38	37.18	78.38	**85.97**
NIST (recursion)	Recall (%)	100.00	100.00	100.00	**100.00**
	Precision (%)	50.00	50.00	55.56	**95.24**

`8cc` [7] (C compiler), `libplist` [8] (Property List file handler), `MP4v2` [2] (MP4 file library), `RIOT` [3] (IoT operating system), `darknet`[1] (neural network framework), `tmux` [5] (terminal multiplexer) and `Teeworlds` [9] (online multiplayer game).

Implementation. The experiments are conducted on an Ubuntu 22.04 server with an eight-core 2.60GHz Intel Xeon CPU and 128 GB memory. The interprocedural control and value flow graphs are built upon LLVM-IR (with LLVM version 14.0.0) using the SVF library (version 2.9). The LLVM-IR represents program functions as global variables, further modeled as address-taken variables. The call graph is built by considering the points-to information to resolve indirect calls. Program loops/recursions are captured by IWTO, which is constructed using the algorithm described in Section 4.1 (Algorithm 1). The abstract value representation is implemented using Z3 expressions [45]. We utilize the interleaved widening and narrowing strategy as described in [10] to precisely handle recursions and intraprocedural loops. We detect buffer overflows at each `gep` instruction by comparing the offset size and the memory size, both of which are calculated based on interval and points-to information from abstract interpretation. For the baselines IKOS, Clam and CSA, we directly use their open-source implementations and their default settings (e.g., interval analysis) for detecting buffer overflows.

5.2 RQ1: NIST Benchmark

Table 3 presents the true positive rates and precision of RecTopo along with three baseline abstract interpreters in detecting buffer overflows [28, 29, 30] using the pre-labeled NIST benchmark, NIST (complete), and its subset containing test cases that includes recursive functions, NIST (recursion).

Comparison Results. For the complete NIST dataset, RecTopo consistently surpasses the baseline tools. It identifies, on average, 26.24% more buffer overflows than IKOS and Clam, while also achieving an average precision improvement of 44.19% over these tools. Its precision is also 7.59% higher than CSA. This performance underscores the robustness of RecTopo in interprocedural abstract interpretation. Moreover, the advantages of RecTopo are particularly notable in scenarios involving recursive functions, where it achieves a precision rate of 95.24% in detecting buffer overflows. In contrast, the average precision of IKOS, Clam and CSA in these test cases is only approximately 51.85%, 43.39% lower than RecTopo.

Result Analysis. The improved recall can be attributed to RecTopo's adoption of CSA's robust handling of core program features, such as loop handling, and precise external API modeling [18]. Here, we focus on analyzing the precision improvements in handling recursion

```c
int mc91(int p) {
    if(p > 100) {
        return p - 10;
    } else {
        int p1 = p + 11;
        int p2 = mc91(p1);
        int res = mc91(p2);
        return res;
    }
}
int main() {
    char buf[200];
    int idx = mc91(100);
    buf[idx] = 0;
}
```

(a) Recursive function.

```c
int sum = 0;
void adder(int num) {
    sum += num;
}
int main() {
    char buf[10];

    // call adder twice
    int m = 1;
    adder(m);
    int n = 2;
    adder(n);

    buf[sum] = 0;
}
```

(b) Multiple call sites.

Figure 11 False buffer overflows eliminated by RecTopo.

```c
int modNb(int n) {
    if (n - 9 >= 0)
        return n;
    else
        return modNb(n + 1);
}
int main() {
    char buf[10];
    int idx = modNb(0);
    buf[idx] = 0;
}
```

(a) Complex path condition.

```c
int sum(int n, int m) {
    if (n <= 0)
        return m + n;
    else
        return sum(n - 1,m + 1);
}
int main() {
    char buf[10];
    int idx = sum(9, 0);
    buf[idx] = 0;
}
```

(b) Variable correlations.

Figure 12 False positives of RecTopo.

through two representative code scenarios shown in Figure 11. A key contributing factor is RecTopo's construction of IWTO for recursive functions, which effectively guides the narrowing process by leveraging program path conditions. As demonstrated in Figure 11(a), the program accesses the `buf` buffer at ℓ_{14} with an offset `idx`, which is obtained from the return value of the `mc91(100)` function call at ℓ_{13}. RecTopo precisely determines `mc91`'s return value in the range $[91, 101]$. Consequently, it concludes that the buffer access at ℓ_{14} is safe. In contrast, all our baseline analyses report an overflow alarm here, incorrectly indicating that the value of `mc91(100)` could be anywhere within $[-\infty, \infty]$.

Another reason is that the IWTO construction approach effectively eliminates spurious recursions (widening points) arising from multiple call sites of the same function, such as the double call to `adder` in Figure 11(b). RecTopo accurately differentiates between these call sites, preventing misclassification of program points, $\ell_{12}, \ell_2 - \ell_4, \ell_{10}, \ell_{11}$, as part of a spurious recursion. This precision prevents RecTopo from widening the value of `sum` at the `adder` entry point (ℓ_2) to $[0, \infty]$. Consequently, RecTopo derives the precise value of `sum` at ℓ_{14} as $[3, 3]$, thereby eliminating a false overflow alarm.

Table 4 Comparing RecTopo with three open-source tools (IKOS, Clam and CSA), using ten popular applications. #TP and #FP are true positive and false positive, respectively. Time (secs) is running time.

Project	IKOS			Clam			CSA			**RecTopo**		
	Report		Time	Report		Time	Report		Time	Report		Time
	#TP	#FP	(secs)	#TP	#FP	(secs)	#TP	#FP	(secs)	#TP	#FP	(secs)
`paste`	3	21	512	2	15	589	3	0	9	3	0	12
`md5sum`	2	35	986	1	28	1146	4	1	8	4	2	10
`YAJL`	1	1625	2895	1	1483	3061	3	0	5	3	0	9
`8cc`	2	1893	3244	1	683	3824	3	7	36	3	5	38
`libplist`	1	1206	3312	1	835	4178	5	14	45	5	8	49
`MP4v2`	1	965	3684	2	681	4375	1	0	13	1	0	16
`RIOT`	2	1325	5216	2	1206	6387	8	6	27	8	4	31
`darknet`	14	1265	9531	9	1324	10936	21	10	3507	21	7	3725
`tmux`	4	1632	11325	2	1359	12894	12	10	824	12	5	983
`Teeworlds`	2	529	13569	4	1436	14734	15	8	2886	15	8	3032
Total	32	10496	54274	25	9050	62124	75	56	7360	**75**	**39**	7905

False Positives of *RecTopo*. We further analyze RecTopo's false positives and identify two primary causes, as demonstrated in the examples in Figure 12. First, RecTopo compromises precision for efficiency in scenarios involving complex path conditions. A case in point is its failure to accurately interpret the path condition at ℓ_2 in Figure 12(a), resulting in an imprecise narrowing of the range of `n` after widening. This imprecision triggers the overflow alarm at ℓ_{10}. This issue primarily stems from the limitations of the abstract interpretation solver, which, although not a key contribution of RecTopo, affects its precision. Enhancing the solver's capability for precise path condition interpretation could rectify this. Another source of precision loss is RecTopo's inability to track relationships between variables. This limitation is illustrated in Figure 12(b), where the parameters `m` and `n` have a constraint that their sum at each call site remains constant. However, RecTopo's interval domain representation only allows for the independent storage of their abstract states, thus failing to capture their interrelation. As a result, when the parameter `m` is widened to $[0, \infty]$, RecTopo over-approximates `idx`'s range as $[0, \infty]$. Adopting a relational domain to articulate the constraints between program variables could overcome this limitation.

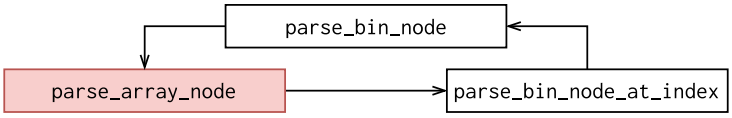
5.3 RQ2: Real-World Projects

Table 4 presents the experimental results for true positives, false positives and running time of RecTopo compared to the three existing tools (IKOS, Clam and CSA) on ten real-world projects.

Comparison Results and Analysis. RecTopo demonstrates superior performance across all metrics, successfully identifying 75 real bugs with a precision rate of 65.79%, representing 79.78% of all identified vulnerabilities. In contrast, IKOS and Clam achieve an average recall rate of only 30.32% among discovered bugs. The precision rates of the baseline tools average 19.28%, less than one-third of RecTopo's precision. Notably, RecTopo achieves these significant improvements while maintaining computational efficiency comparable to CSA.

```
1  static plist_t parse_array_node(struct bplist_data *bplist,
2                                  const char** bnode, uint64_t size)
3  {
4      plist_data_t data = plist_new_plist_data();
5      const char *index1_ptr = NULL;
6      ...
7      for (uint64_t j = 0; j < size; j++) {
8          index1_ptr = (*bnode) + j * bplist->ref_size;
9  +       if (index1_ptr < bplist->data ||
10 +           index1_ptr + bplist->ref_size >=
11 +           bplist->data + bplist->size) {
12 +               plist_free(node);
13 +               return NULL;
14 +       }
15         index1 = UINT_TO_HOST(index1_ptr, bplist->ref_size);
16     ...
17 }
```

Patch



Call Graph

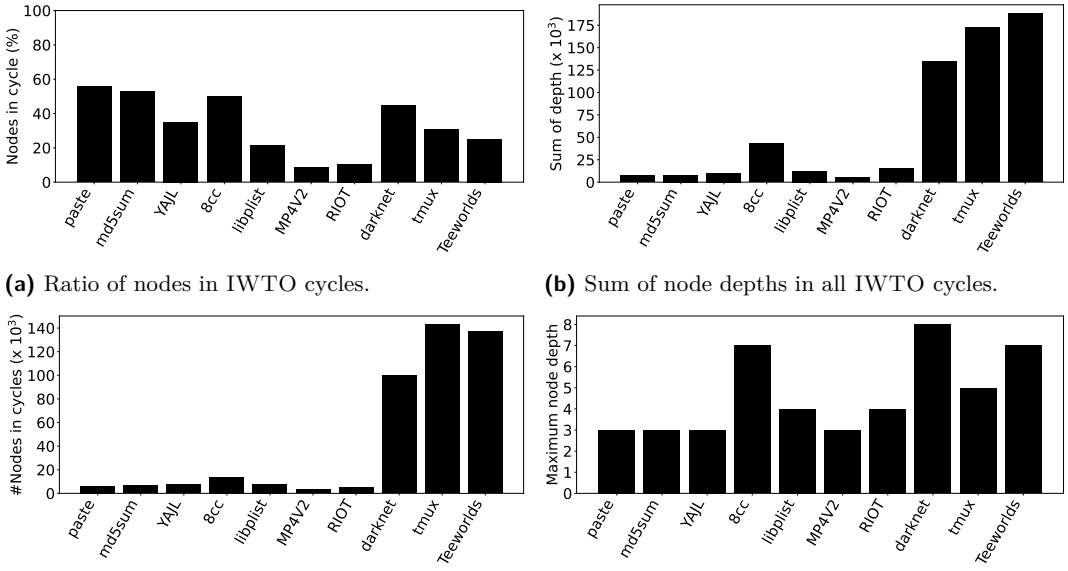
Figure 13 The patch and call graph of a real-world buffer overflow in `libplist` [8] found by our tool.

The markedly lower false positive rate of RecTopo can be attributed to its sophisticated abstract interpretation framework, guided by the IWTO approach. This methodology enables precise derivation of abstract states within recursive contexts through carefully orchestrated interleaved widening and narrowing operations on recursions.

Case Study. Figure 13 illustrates the patch [6] and call graph for a real-world buffer overflow identified by RecTopo in `libplist` [8], a portable C library to handle Property List files [53]. RecTopo and CSA initially find this bug, which is missed by Clam and IKOS. However, after the bug is patched, CSA erroneously continue to report it as a buffer overflow – resulting in false positives. By comparison, RecTopo accurately eliminates the false alarm after the bug is fixed.

The bug resides in the `parse_array_node` function, within a recursive calling chain as depicted in Figure 13. It is triggered at line ℓ_{15}, where the function `UINT_TO_HOST` accesses the address pointed by `index1_ptr` and reads an extra `bplist→ref_size` bytes. This bug arises because `bplist→ref_size`, the buffer access variable in the recursion, is not properly bounded, potentially allowing reads past the allocated buffer of `index1_ptr`, leading to a buffer overflow. When the issue is addressed by introducing a boundary check before ℓ_{15}, RecTopo stops reporting the problem, while CSA still signal a bug. This discrepancy is attributed to our utilization of widening and narrowing techniques within the recursive function partition, which precisely leverage path conditions to refine the abstract value of `bplist→ref_size`, thereby providing more accurate results and effectively eliminating the false positive.

Statistics. We present detailed statistics of IWTO across the ten real-world projects. Figure 14a illustrates the proportion of nodes residing in IWTO cycles for each program. The data reveals a substantial presence of nodes within IWTO cycles, ranging from 8.4% to



(a) Ratio of nodes in IWTO cycles.

(b) Sum of node depths in all IWTO cycles.

(c) Number of nodes in IWTO cycles.

(d) Maximum node depth in the IWTO.

Figure 14 Statistics of IWTO in real-world programs.

55.9%, with an average of 33.5%. Figure 14b depicts the aggregate depth of nodes (as defined in §2.3) across all IWTO cycles in each program. The data exhibits a near-linear growth pattern relative to program size. Given that the time complexity for cycle stabilization is linear with respect to the sum of node depths [13], this suggests that the overall time complexity for stabilizing IWTO cycles scales approximately linearly with program size. Figure 14c presents the number of nodes within IWTO cycles per program, while Figure 14d shows the maximum node depth in each program's IWTO. The data demonstrates that while the number of nodes in cycles increases linearly with program scale, the maximum depth remains relatively stable. This relationship explains the linear growth pattern observed in the aggregate node depth measurements.

5.4 RQ3: Ablation Analysis

This section presents a comprehensive ablation analysis of RecTopo, with a focus on evaluating its precision enhancements and scalability.

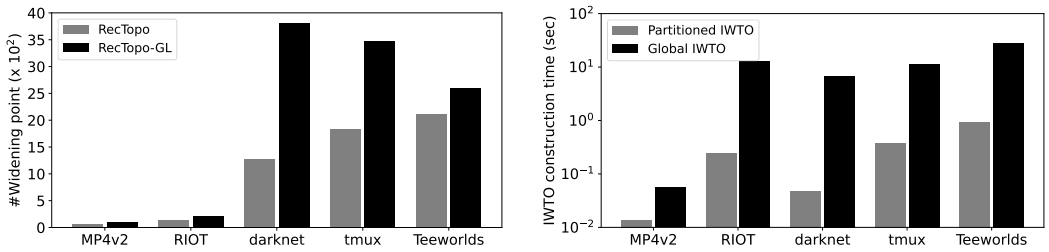
We compare RecTopo with two distinct variants: RecTopo-NR, which omits recursive calls by assigning a default top value; and RecTopo-GL, which integrates a whole-program global IWTO. This comparative analysis underscores the critical contributions of IWTO in enhancing both precision and scalability.

Table 5 presents the true positives, false positives and running time of RecTopo compared to its two variants (RecTopo-NR and RecTopo-GL) on real-world popular applications. The results demonstrate that the IWTO employed in RecTopo significantly enhances precision while incurring only a moderate increase in computational overhead.

Comparison with *RecTopo*-NR. RecTopo exhibits 60.41% higher precision than RecTopo-NR. This superior precision stems from RecTopo's ability to more precisely calculate the values of variables within recursive functions. In contrast, RecTopo-NR simplifies the process by assigning a generic top value to these variables.

Table 5 Comparing RecTopo with its two variants, RecTopo-NR and RecTopo-GL (described in §5.4), using ten popular applications. #TP and #FP are true positive and false positive, respectively. Time (secs) is running time.

Project	RecTopo-NR			RecTopo-GL			**RecTopo**		
	Report		Time	Report		Time	Report		Time
	#TP	#FP	(secs)	#TP	#FP	(secs)	#TP	#FP	(secs)
`paste`	3	13	8	3	32	15	3	0	12
`md5sum`	4	9	6	4	36	13	4	2	10
`YAJL`	3	62	3	3	97	12	3	0	9
`8cc`	3	85	30	3	108	44	3	5	38
`libplist`	5	119	39	5	230	52	5	8	49
`MP4v2`	1	131	11	1	193	23	1	0	16
`RIOT`	8	126	22	8	285	35	8	4	31
`darknet`	21	217	3168	21	914	4211	21	7	3725
`tmux`	12	253	727	12	1322	1247	12	5	983
`Teeworlds`	15	304	2491	15	1641	3416	15	8	3032
Total	75	1319	6505	75	4858	9068	**75**	**39**	7905



(a) (b)

Figure 15 (a) Comparing the number of widening points of RecTopo and RecTopo-GL on real-world programs and (b) Comparing IWTO construction time of partitioned and global IWTO on real-world programs.

This overly conservative approaches of RecTopo-NR results in broader intervals that frequently trigger false overflow alarms during buffer access. In terms of performance efficiency, RecTopo-NR leads in execution speed as it bypasses the body of recursive functions entirely.

However, the slight time advantage of RecTopo-NR does not compensate for the substantial precision gains achieved by RecTopo, reinforcing its effectiveness in balancing precision with performance.

Comparison with *RecTopo*-GL. RecTopo-GL implements a global whole-program IWTO that establishes connections between all call sites and their corresponding callee functions. The abstract interpretation within RecTopo-GL is managed by this singular IWTO, with each cycle processed with precision comparable to that of RecTopo. Despite this, RecTopo-GL demonstrates a reduction in precision by 64.27%, and it also operates 1.15× longer compared to RecTopo.

The reduced precision in RecTopo-GL can be primarily attributed to its redundant widening operations. As depicted in Figure 15a, RecTopo achieves a significant decrease in the number of widening points, with an average reduction of 58.99% and a peak reduction of

up to 81.56%. This substantial decrease in unnecessary widening throughout various program procedures directly contributes to the enhanced precision of RecTopo. Regarding scalability, the comparison of the time required to construct IWTO in RecTopo and RecTopo-GL highlights significant differences. Figure 15b clearly shows that constructing a whole-program IWTO in RecTopo-GL demands considerably more time – nearly 36 times longer – than the IWTO approach used in RecTopo. This comparison underscores the efficiency and practicality of the partitioned approach in handling scalability.

6 Related Work

Our discussion centers on areas closely related to RecTopo, specifically focusing on abstract interpretation, fixpoint computation, and recursion handling in program analysis.

Abstract Interpretation. Abstract interpretation [24, 27] serves as the theoretical foundation for numerous static analysis techniques, encompassing program verification [31, 15, 42, 66, 67], data-flow analysis [50, 51], and static bug detection [59, 60, 18, 55, 56, 63, 48]. These techniques are fundamentally designed to ensure both state over-approximation and analysis termination. Some approaches [48, 36] utilize sparse abstract interpretation framework that propagates abstract states through def-use chains instead of control flows. In contrast, our work operates within the context of control-point-based analysis. Note that this assumption is a design choice made to align with classical abstract interpretation formulations [13, 18, 20, 40, 47], though our approach is not fundamentally restricted to this scope. Weiss et al. [65] proposes database-backed analysis with graph algorithms to enhance both scalability and precision. More recently, Cheng et al. [18] developed cross-domain abstract interpretation to track correlations across different abstract domains. While significant, these approaches are orthogonal to our work.

Fixpoint Computation. Fixpoint computation, originally conceptualized by Cousot et al. [24], forms the theoretical cornerstone of abstract interpretation by providing conservative approximations of program runtime behaviors. For acyclic control flows, fixpoint computation typically terminates after a few iterations over the CFG. However, in the presence of cyclic control flows, the fixpoint computation may require an impractically large number of iterations to converge, or in the worst case, may never converge, especially when using abstract domains characterized by infinite or very large lattice heights. To address this issue, widening operators [23] are introduced to accelerate the fixpoint computation and ensure its termination. Since its introduction, Bourdoncle's algorithm [13] has remained the predominant approach for efficient fixpoint computation in abstract interpretation. Subsequent research has largely built upon this foundation, exploring various refinements such as interleaved widening and narrowing strategies to enhance precision [10, 11, 35, 33] and parallel computation techniques [40, 64, 44]. These advances, while valuable, are orthogonal to our focus on interprocedural fixpoint computation. Current interprocedural methods typically either inline callee functions [15], overlooking opportunities for interprocedural topological ordering, or disregard varying calling contexts [48, 47], potentially introducing superfluous widening operations.

Recursion Handling. In interprocedural static analysis, recursive program structures present significant challenges by potentially generating unbounded call stacks that can affect analyzer termination. Existing approaches address this challenge through various strategies: some

employ fixed-depth recursive call unrolling [59, 55, 56], while others explicitly bound the analysis of recursive calls [26, 37]. These approaches offer practical trade-offs between precision and analysis feasibility. More conservative methods focus on computing sound approximations of values within recursive structures [15, 18], prioritizing soundness while potentially sacrificing precision when analyzing recursions. Oh et al. [49] proposed techniques for eliminating spurious interprocedural cycles, though not specifically targeting recursion handling. Rival et al. [54] developed specialized methods for handling recursion in shape analysis, focusing primarily on precise approximation of recursive heap structures – an approach complementary to our work. Keidel et al. [38, 14] introduces a terminable big-step abstract interpretation approach even in the presence of loops and recursions. It achieves this by identifying recurrent call traces of the abstract interpreter itself during runtime. However, while this dynamic approach allows the application of widening on recursions, it currently does not support further narrowing to improve precision. Späth et al. [57] proposes a synchronized pushdown system that improves time complexity for recursive data structures. It operates within a pushdown framework, which differs from our abstract interpretation methodology. Stein [58] introduces a versatile framework for incremental and demand-driven tabulation-based abstract interpretation, utilizing dynamic summarization graphs to support sound analysis of recursion. While effective, it does not incorporate Bourdoncle's strategy, which could enhance efficiency in handling cycles.

7 Conclusion

This paper introduces RecTopo, a novel approach to fixpoint computation for precise interprocedural abstract interpretation. RecTopo utilizes interprocedural weak topological ordering to steer chaotic iterations across program procedures effectively. This strategy not only incorporates an interleaved widening and narrowing approach for accurately addressing recursions but also eliminates redundant widening caused by interprocedural dependencies. RecTopo demonstrates an average precision improvement of 46.51% and an increased recall rate of 32.98% when compared to baselines tools on ten open-source projects.

References

1 Darknet: Open source neural networks in C. `https://github.com/pjreddie/darknet`, 2023.
2 MP4v2: A C/C++ library to create, modify and read MP4 files. `https://github.com/enzo1982/mp4v2/`, 2023.
3 RIOT: The friendly OS for IoT. `https://github.com/RIOT-OS/RIOT`, 2023.
4 YAJL: A fast streaming JSON parsing library in C. `https://github.com/lloyd/yajl`, 2023.
5 Tmux: Tmux source code. `https://github.com/tmux/tmux`, 2023.
6 A buffer overflow patch in libplist. `https://github.com/libimobiledevice/libplist/commit/4765d9a60ca4248a8f89289271ac69cbffcc29bc`, 2024.
7 8cc: C compiler. `https://github.com/rui314/8cc`, 2024.
8 libplist: A small portable C library to handle Apple Property List files in binary, XML, JSON, or OpenStep format. `https://github.com/libimobiledevice/libplist`, 2024.
9 Teeworlds: A retro multiplayer shooter. `https://teeworlds.com/`, 2024.
10 Gianluca Amato and Francesca Scozzari. Localizing widening and narrowing. In *International Static Analysis Symposium*, pages 25–42. Springer, 2013. `doi:10.1007/978-3-642-38856-9_4`.
11 Gianluca Amato, Francesca Scozzari, Helmut Seidl, Kalmer Apinis, and Vesal Vojdani. Efficiently intertwining widening and narrowing. *Science of Computer Programming*, 120:1–24, 2016. `doi:10.1016/j.scico.2015.12.005`.

12 George Balatsouras and Yannis Smaragdakis. Structure-sensitive points-to analysis for C and C++. In *SAS '16*, 2016. doi:10.1007/978-3-662-53413-7_5.

13 François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications: International Conference Academgorodok, Novosibirsk, Russia June 28–July 2, 1993 Proceedings*, pages 128–141. Springer, 2005. doi:10.1007/BFb0039704.

14 Katharina Brandl, Sebastian Erdweg, Sven Keidel, and Nils Hansen. Modular abstract definitional interpreters for webassembly. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, pages 5–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

15 Guillaume Brat, Jorge A Navas, Nija Shi, and Arnaud Venet. IKOS: A framework for static analysis based on abstract interpretation. In *Software Engineering and Formal Methods: 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings 12*, pages 271–277. Springer, 2014. doi:10.1007/978-3-319-10431-7_20.

16 Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224. USENIX Association, 2008. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

17 Liqian Chen, Antoine Miné, Ji Wang, and Patrick Cousot. Interval polyhedra: An abstract domain to infer interval linear relationships. In *International Static Analysis Symposium*, pages 309–325. Springer, 2009. doi:10.1007/978-3-642-03237-0_21.

18 Xiao Cheng, Jiawei Wang, and Yulei Sui. Precise sparse abstract execution via cross-domain interaction. In *46th International Conference on Software Engineering*, ICSE '24. ACM/IEEE, 2024. doi:10.1145/3597503.3639220.

19 Clam. Clam: LLVM front-end for Crab, 2024. URL: https://github.com/seahorn/clam.

20 Patrick Cousot. Asynchronous iterative methods for solving a fixpoint system of monotone equations. Technical report, Research Report IMAG-RR-88, Université Scientifique et Médicale de Grenoble, 1977.

21 Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes*. PhD thesis, Institut National Polytechnique de Grenoble-INPG; Université Joseph-Fourier . . . , 1978. URL: https://tel.archives-ouvertes.fr/tel-00288657.

22 Patrick Cousot. Semantic foundations of program analysis. In *Program flow analysis: theory and applications*, pages 303–342. Prentice Hall, 1981.

23 Patrick Cousot. Abstracting induction by extrapolation and interpolation. In Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 19–42, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi:10.1007/978-3-662-46081-8_2.

24 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977. doi:10.1145/512950.512973.

25 Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, 1992. doi:10.1093/logcom/2.4.511.

26 Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The Astrée analyzer. In *Programming Languages and Systems: 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings 14*, pages 21–30. Springer, 2005. doi:10.1007/978-3-540-31987-0_3.

27 Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. A²I: Abstract² Interpretation. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290355.

28 CWE-121: Stack-based buffer overflow. https://cwe.mitre.org/data/definitions/121.html, 2024.

29 CWE-122: Heap-based buffer overflow. https://cwe.mitre.org/data/definitions/122.html, 2024.

30 CWE-126: buffer over-read. https://cwe.mitre.org/data/definitions/126.html, 2024.

31 Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, PLDI '02, pages 57–68, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/512529.512538.

32 Matt Elder. Bourdoncle components, 2010.

33 Denis Gopan and Thomas Reps. Lookahead widening. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, pages 452–466, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. doi:10.1007/11817963_41.

34 Arie Gurfinkel and Jorge A. Navas. Abstract interpretation of LLVM with a region-based memory model. In *Software Verification: 13th International Conference, VSTTE 2021, New Haven, CT, USA, October 18–19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18–19, 2021, Revised Selected Papers*, pages 122–144, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-95561-8_8.

35 Nicolas Halbwachs and Julien Henry. When the decreasing sequence fails. In Antoine Miné and David Schmidt, editors, *Static Analysis*, pages 198–213, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-33125-1_15.

36 Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 289–298. IEEE, 2011. doi:10.1109/CGO.2011.5764696.

37 Daniel Kästner, Stephan Wilhelm, Stefana Nenova, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, Xavier Rival, et al. Astrée: Proving the absence of runtime errors. *Proc. of Embedded Real Time Software and Systems (ERTS2 2010)*, 9, 2010.

38 Sven Keidel, Sebastian Erdweg, and Tobias Hombücher. Combinator-based fixpoint algorithms for big-step abstract interpreters. *Proceedings of the ACM on Programming Languages*, 7(ICFP):955–981, 2023. doi:10.1145/3607863.

39 Sol Kim, Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Widening with thresholds via binary search. *Software: Practice and Experience*, 46(10):1317–1328, 2016. doi:10.1002/spe.2381.

40 Sung Kook Kim, Arnaud J. Venet, and Aditya V. Thakur. Deterministic parallel fixpoint computation. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371082.

41 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004. doi:10.1109/CGO.2004.1281665.

42 Jacob Laurel, Rem Yang, Gagandeep Singh, and Sasa Misailovic. A dual number abstraction for static analysis of Clarke Jacobians. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. doi:10.1145/3498718.

43 Zohar Manna. *Mathematical theory of computation*. Dover Publications, Inc., 2003.

44 David Monniaux. The parallel implementation of the Astrée static analyzer. In Kwangkeun Yi, editor, *Programming Languages and Systems*, pages 86–96, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. doi:10.1007/11575467_7.

45 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

46 NIST datasets. https://samate.nist.gov/SARD/test-suites/116, 2023.

47 Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for C-like languages. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 229–238, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2254064.2254092.

48 Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. The Sparrow static analyzer, 2012. URL: `https://opam.ocaml.org/packages/sparrow/`.

49 Hakjoo Oh and Kwangkeun Yi. An algorithmic mitigation of large spurious interprocedural cycles in static analysis. *Software: Practice and Experience*, 40(8):585–603, 2010. `doi:10.1002/spe.969`.

50 Komal Pathade and Uday P. Khedker. Computing partially path-sensitive MFP solutions in data flow analyses. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 37–47, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3178372.3179497`.

51 Komal Pathade and Uday P. Khedker. Path sensitive MFP solutions in presence of intersecting infeasible control flow path segments. In *Proceedings of the 28th International Conference on Compiler Construction*, CC 2019, pages 159–169, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3302516.3307349`.

52 D.J. Pearce, P.H.J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis of C. *ACM TOPLAS*, 30(1):4–es, 2007. `doi:10.1145/1290520.1290524`.

53 Property list. `https://en.wikipedia.org/wiki/Property_list`, 2024.

54 Xavier Rival and Bor-Yuh Evan Chang. Calling context abstraction with shapes. *SIGPLAN Not.*, 46(1):173–186, January 2011. `doi:10.1145/1925844.1926406`.

55 Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 693–706, 2018. `doi:10.1145/3192366.3192418`.

56 Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Path-sensitive sparse analysis without path conditions. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 930–943, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3453483.3454086`.

57 Johannes Späth, Karim Ali, and Eric Bodden. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019. `doi:10.1145/3290361`.

58 Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. Interactive abstract interpretation with demanded summarization. *ACM Transactions on Programming Languages and Systems*, 46(1):1–40, 2024. `doi:10.1145/3648441`.

59 Yulei Sui, Ding Ye, and Jingling Xue. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA '12, pages 254–264. ACM, 2012. `doi:10.1145/2338965.2336784`.

60 Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Trans. Software Eng (TSE '14).*, 40(2):107–122, 2014. `doi:10.1109/TSE.2014.2302311`.

61 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972. `doi:10.1137/0201010`.

62 Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

63 Oskar Haarklou Veileborg, Georgian-Vlad Saioc, and Anders Møller. Detecting blocking errors in go programs using localized abstract interpretation. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3551349.3561154`.

64 Arnaud Venet and Guillaume Brat. Precise and efficient static array bound checking for large embedded C programs. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 231–242, New York, NY, USA, 2004. Association for Computing Machinery. `doi:10.1145/996841.996869`.

65 Cathrin Weiss, Cindy Rubio-González, and Ben Liblit. Database-backed program analysis for scalable error propagation. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 586–597, 2015. doi:10.1109/ICSE.2015.75.

66 Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. Program analysis via efficient symbolic abstraction. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi:10.1145/3485495.

67 Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. Grapple: A graph system for static finite-state property checking of large-scale systems code. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, EuroSys '19. ACM, 2019. doi:10.1145/3302424.3303972.

Chain of Grounded Objectives: Concise Goal-Oriented Prompting for Code Generation

Sangyeop Yeo
Electronics and Telecommunications Research Institute, Daejeon, South Korea

Seung-Won Hwang
Department of Computer Science and Engineering, Seoul National University, South Korea

Yu-Seung Ma[1]
Electronics and Telecommunications Research Institute, Daejeon, South Korea

Abstract

The use of Large Language Models (LLMs) for code generation has gained significant attention in recent years. Existing methods often aim to improve the quality of generated code by incorporating additional contextual information or guidance into input prompts. Many of these approaches adopt process-oriented reasoning strategies, mimicking human-like step-by-step thinking; however, they may not always align with the structured nature of programming languages. This paper introduces **Chain of Grounded Objectives (CGO)**, a concise goal-oriented prompting approach that embeds functional objectives into prompts to enhance code generation. By focusing on precisely defined objectives rather than explicit procedural steps, CGO aligns more naturally with programming tasks while retaining flexibility. Empirical evaluations on HumanEval, MBPP, their extended versions, and LiveCodeBench show that CGO achieves accuracy comparable to or better than existing methods while using fewer tokens, making it a more efficient approach to LLM-based code generation.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence

Keywords and phrases Artificial Intelligence, Natural Language Processing, Prompt Design, Large Language Models, Code Generation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.35

Related Version *ArXiv Version*: `https://arxiv.org/abs/2501.13978`

Supplementary Material *Software (Source Code)*: `https://github.com/DDIDUs/Chain-of-Grounded-Objectives` [31]; archived at `swh:1:dir:315704008a9f9a515047943266a959518485cbb0`

Funding This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2022-0-00995, Automated reliable source code generation from natural language descriptions)

Acknowledgements We appreciate the anonymous reviewers' constructive feedback that improved our paper.

1 Introduction

The advancement of Large Language Models (LLMs) has significantly transformed software development, particularly in natural language-driven code generation [35, 12]. These models enable developers to generate code from textual descriptions, reducing manual effort and expanding accessibility for non-experts. However, fully leveraging their potential requires addressing challenges related to output accuracy, motivating researchers to explore more structured and detailed prompts. In response, recent research has focused on enhancing the input to LLMs by incorporating additional context or guidance, and these methods can be classified according to the type of supplementary information provided:

[1] corresponding author

© Sangyeop Yeo, Seung-Won Hwang, and Yu-Seung Ma;
licensed under Creative Commons License CC-BY 4.0

First, **example-based guidance** approaches involve supplying sample codes or problem-solution examples to guide LLMs; with few-shot prompting [4] being a typical example.

Second, **process-oriented reasoning**, focuses on guiding the model through the procedural steps required to solve a problem, essentially explaining "how" to reach a solution. Prompts like Chain of Thought [30] and self-planning [16] exemplify this approach by decomposing the problem-solving process into a sequence of logical steps. Based on our preliminary experiments, while effective, process-oriented reasoning can introduce unnecessary complexity in straightforward tasks, potentially leading to increased inference time and token consumption.

Third, **goal-oriented reasoning** focuses on specifying "what" needs to be achieved rather than prescribing "how" to achieve it. This includes prompting with input-output examples, specifications, or constraints, leaving the model to infer the optimal approach. Test-driven prompting [25, 21] is a representative example: the problem goal is defined by test cases, but such methods require code execution for validation, incurring computational overhead.

However, **is complex reasoning always necessary for effective code generation?** Instead of guiding LLMs through intricate reasoning steps, could a simpler approach – one that provides only clear and direct goals – lead to better results?

To explore this, we propose **Chain of Grounded Objectives (CGO)**, a concise goal-oriented reasoning approach that succinctly captures the objectives of a coding problem. CGO first prompts the LLM to analyze the given problem and autonomously generate a compact set of objectives outlining the functional requirements of the solution. These objectives are then incorporated as supplemental input to guide the final code generation step. Notably, they are expressed in a style similar to code comments, a format LLMs have extensively encountered during training. By presenting objectives in this grounded manner, CGO enhances compatibility with programming languages while prioritizing computational interpretability. Furthermore, since CGO introduces minimal additional information, it consumes fewer tokens than verbose, process-oriented prompts, making it a more efficient alternative.

By examining CGO's effectiveness and efficiency in comparison to existing reasoning techniques, we aim to determine whether concise goal-driven guidance can serve as a more practical alternative to complex procedural reasoning in code generation.

The key contributions of this study are as follows:

- **Chain of Grounded Objectives (CGO)** is introduced as a concise goal-oriented reasoning prompt technique that employs functional objectives to guide code generation.
- **Higher accuracy**: Experimental results on HumanEval and MBPP show that CGO outperforms existing prompting techniques.
- **Real-world applicability**: CGO also exhibits competitive performance on LiveCode Bench, a benchmark reflecting more realistic coding scenarios.
- **Efficiency**: Despite requiring fewer tokens, CGO maintains high-quality code generation relative to other prompt techniques.

2 Related Work

2.1 Code Generation Using LLMs

The development of pretrained language models has significantly influenced research in code generation. Models such as CodeT5 [29] and UnixCoder [8], which learn from both textual and code data, have demonstrated strong performance across various code generation

and understanding tasks. Furthermore, models like Codex, CodeGen, AlphaCode, and CodeGeeX have shown that incremental increases in model size can lead to improved performance, highlighting the importance of scaling model parameters in code generation models [6, 22, 19, 34].

Recently, open-source models have also made significant progress. Models such as LLaMA and Mistral [26, 15] provide researchers with accessible yet high-performing alternatives, while CodeLLaMA [27], specifically designed for code generation, has achieved strong performance in code-related tasks. Moreover, the emergence of small open-weight models like Deepseek and Qwen has further improved the efficiency of code generation models. Qwen [12] has released models optimized not only for general natural language processing (NLP) tasks but also for code generation, demonstrating performance comparable to some large proprietary models. Deepseek introduced various architectures, including Deepseek-R1 [9], which enhances Deepseek-V3 with efficient reinforce learning, excelling in benchmarks and competitive programming.

Additionally, proprietary models such as the GPT series, Gemini, and Claude continue to achieve state-of-the-art performance in code generation, demonstrating ongoing advancements in code understanding and generation capabilities [1, 24].

2.2 Prompting Strategies for Code Generation

The simplest way to generate code with LLMs is to input the coding problem directly and let the model produce a solution. However, as effective prompting has been shown to improve LLM performance, various studies have explored prompting techniques for code generation.

Example-based guidance approaches, such as few-shot prompting [4], provide example problem-solution pairs but have shown limited impact on code quality, according to our preliminary experiments. Process-oriented reasoning methods decompose the problem-solving process into a sequence of manageable steps. CodeCoT adapts CoT for code generation by refining intermediate steps to align with coding tasks [11], Structured Chain of Thought (SCoT) uses pseudocode as intermediate steps [18]. Self-Planning [16] prompts the model to create a detailed plan before generating code, effectively breaking complex problems into smaller, manageable subtasks. Goal-oriented reasoning includes methods like Test-Case-Driven Chain of Thought (TCoT) [25]. TCoT focuses on defining goals in the form of test cases to validate generated code through execution and iterative refinement. However, its reliance on execution-based feedback loop incurs computational costs.

3 Proposed Technique: CGO Prompting

In this study, we propose Chain of Grounded Objectives (CGO) prompting, a concise goal-oriented reasoning approach designed to enhance the performance of LLMs in automated code generation tasks. The core idea behind CGO is to provide functional objectives as additional input to guide the LLM. These objectives are presented in the form of comment-level representations that are likely familiar to the LLM based on its training data, such as structured natural language descriptions commonly found alongside code.

3.1 Motivation

CGO is grounded in the principle that learning-aligned prompting has been shown to be effective [28], and that providing flexible guidance rather than overly specific context can lead to improved generation performance [2]. Specifically, we adopt comment-style objectives

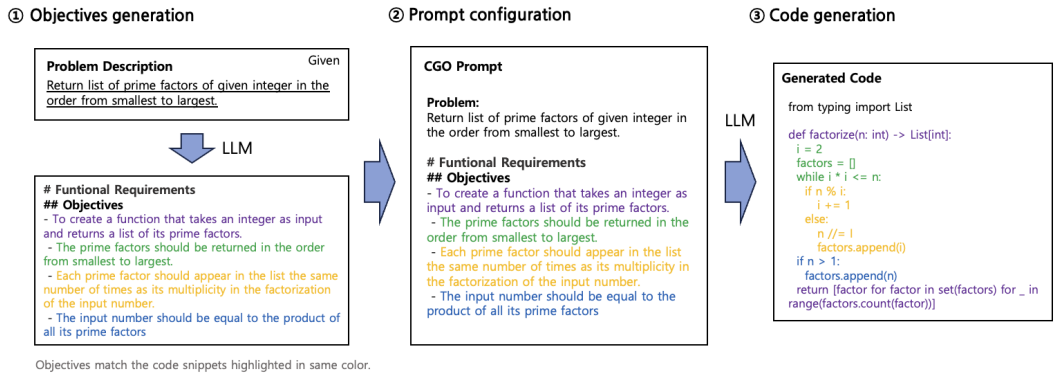


Figure 1 CGO's Workflow.

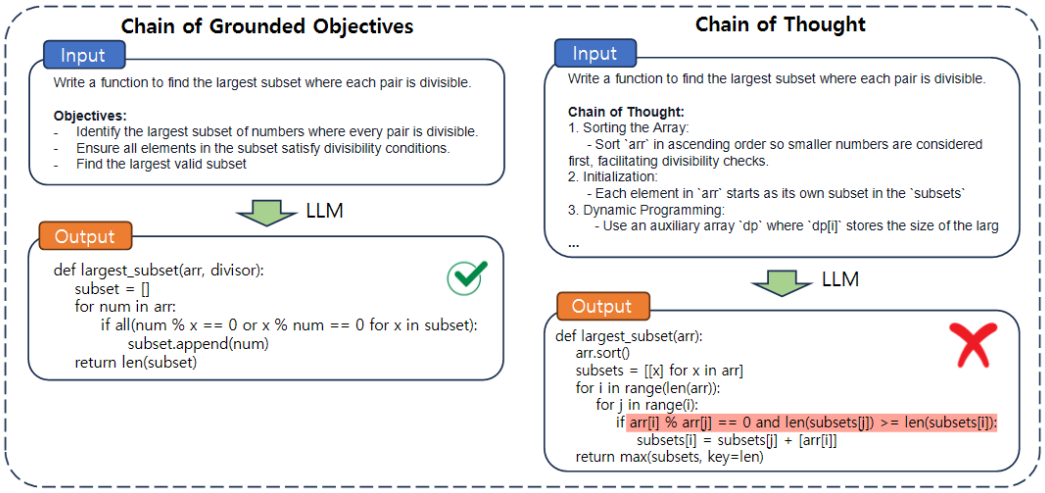


Figure 2 Code example generated with the Chain of Grounded Objectives and Chain of Thought.

based on the intuition that these resemble structured explanations commonly found in the training data of LLMs. Such structured explanations, including comments and documentation, generally describe intended functionality in a way that abstracts implementation details. By incorporating such familiar patterns, we expect that CGO prompting:

- Provides clarity to the LLM about the task requirements.
- Improves the model's ability to map functional objectives to code implementations.
- Leverage its training familiarity with structured descriptions to improve the overall quality of code generation.

Figure 1 illustrates the workflow of the CGO-based code generation process consists of two stages: (1) the "Objective Generation Stage", where the LLM generates functional objectives from the problem description, and (2) the "Code Generation Stage", where these objectives are used as additional context to guide the LLM in generating the code.

The CGO-based code generation process can be formally described as follows (detailed examples are provided in Tables 8):

(1) Objective Generation Stage

In the first stage, the LLM generates a set of functional objectives $O = \{o_1, o_2, \ldots, o_n\}$ based on the given problem description P.

$$O = \text{GenObj}(P) \tag{1}$$

Here, GenObj is a function that maps the problem description P to a set of objectives O, where each objective o_i represents a specific sub-goal, such as a functional requirement, that the final code is expected to achieve. The LLM is guided by a specifically designed prompt, such as *"Write **objectives** for the problem."* This step ensures that the model explicitly understands the key elements of the problem before generating code.

By framing functional objectives to reflect the developer's intent, this approach bridges natural language and code, using the LLM's reasoning to align with the developer's thought process.

(2) Code Generation Stage

In the second stage, the LLM synthesizes the final code C using both the problem description P and the objectives O as input. This process is represented by the function GenCode:

$$C = \text{GenCode}(O \mid P) \tag{2}$$

The notation $O \mid P$ indicates that while the objectives O are derived from the problem P, they are treated as additional context that supplements the original problem description. The function GenCode leverages this enriched context to generate code that not only solves the problem but also adheres to the specific goals outlined in O. This supports the LLM in developing a clear understanding of the problem's requirements, resulting in code that is both functionally correct and contextually appropriate.

The entire CGO-based code generation process can be represented by combining the two stages:

$$C = \text{GenCode}(\text{GenObj}(P) \mid P) \tag{3}$$

4 Evaluation

4.1 Experimental Setup

4.1.1 Benchmarks

Experiments were conducted on five datasets: HumanEval, MBPP-sanitized, their extended versions – HumanEval+ and MBPP+ – which include additional edge cases, and LiveCodeBench.

While HumanEval and MBPP-sanitized provide high-quality coding problems for evaluating LLMs, they primarily consist of standalone function implementations, making them relatively simple benchmarks. HumanEval+ and MBPP+ improve robustness assessment by incorporating more test cases, while LiveCodeBench better reflects real-world coding challenges by requiring a deeper understanding of problem contexts and constraints. These datasets support the evaluation of both fundamental code generation capabilities and performance in practical settings.

- **HumanEval**: A benchmark of 164 coding problems introduced by OpenAI, each comprising a natural language description, example function signatures, and an average of 7.7 test cases per problem.

- **MBPP-sanitized** [3]: A refined subset of MBPP, containing verified problems with a problem description, a reference solution, and three test cases for each instance.
- **HumanEval+ and MBPP+** [20]: Extended versions of HumanEval and MBPP with 80x and 35x more test cases, focusing on edge case robustness.
- **LiveCodeBench** [13] : A contamination-free and continuously evolving benchmark for evaluating LLMs on code generation. Each problem in LiveCodeBench includes a structured problem description, example inputs and outputs, and explicit constraints, making it more reflective of real-world programming tasks.

In this experiment, since MBPP-sanitized does not provide function signatures, the input prompt was adjusted to follow the **HumanEval-style** for consistency by extracting function headers from the ground-truth code. Examples of this formatting are shown in Table 7.

4.1.2 Baselines

We compared the proposed method with the following baseline approaches (detailed examples are provided in Tables 9 to 12):

- **Direct prompting**: Generates code solely from the problem description.
- **Few-shot**: Includes example problems and solutions to guide the model.
- **CodeCoT**: Utilizes Chain of Thought (CoT) with a one-shot example to generate intermediate steps and final code in two stages [11].
- **Zero-shot CoT**: A prompt that generates a CoT Reasoning process to solve the given problem without providing example CoT and incorporates it into code generation [17].
- **Self-Planning**: Extracts sub-tasks from the problem and generates a plan, which guides code generation in two stages [16].
- **Self-Pseudo**: Reflecting the design phase of the software development process, first define the problem requirements and then generate pseudocode based on them. Subsequently, use the generated pseudocode to write the final code in three stage.

4.1.3 Performance Evaluation

We compared the proposed CGO against six baseline reasoning methods in Section 4.1.2 using five LLMs: LLaMA3-8B-Instruct, LLaMA3-70B-Instruct, LLaMA3.1-8B-Instruct, LLaMA3.1-70B-Instruct, and GPT-3.5-Turbo. For all methods, both the intermediate reasoning steps and the final code generation were performed using *greedy sampling* (*temperature* = 0, *top-p* = 1), producing deterministic outputs at each stage. Further details of the experimental setup are provided in Appendix A.

Performance evaluation was conducted using *pass@k*, a widely adopted metric, along with *pass-ratio@n*, which provides a more fine-grained analysis of code generation performance. Further details regarding the metrics bellow are provided in Appendix B.

- *pass@k*: The probability of generating at least one correct solution among k attempts [6].
- *pass-ratio@n*: The average squared ratio of passed test cases across n generated solutions, emphasizing higher accuracy, as defined in [32].

Although *greedy sampling* typically produces the same output for a given prompt, minor variations may still occur due to factors such as floating-point precision errors and non-deterministic execution environments. To account for this, we repeated the experiment 10 times per problem. As a result, we set $k = 1$ for *pass@k*, while for *pass-ratio@n*, we used $n = 10$, averaging results across multiple runs.

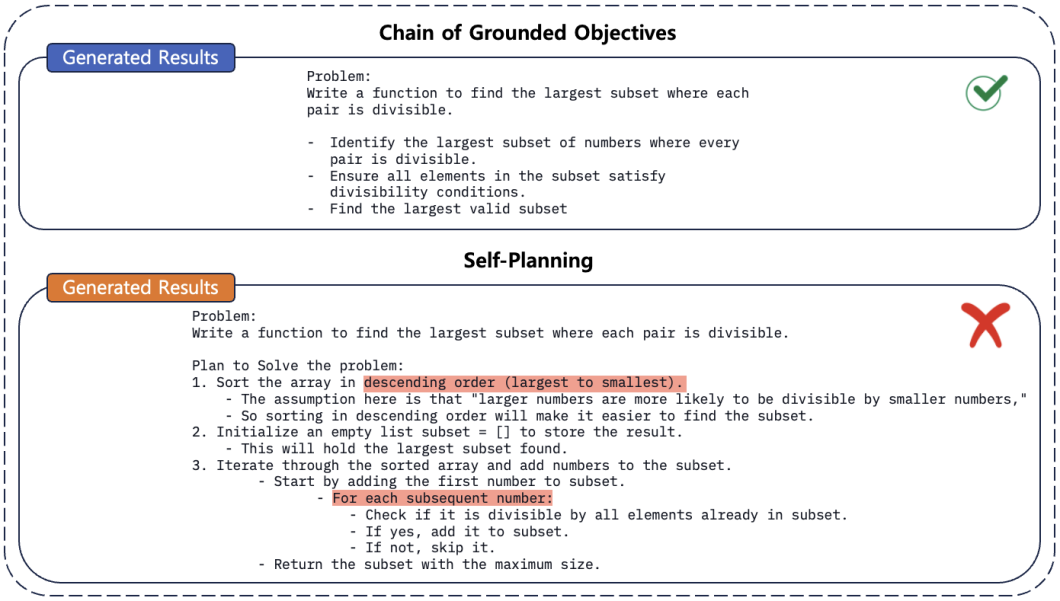


Figure 3 Context example generated with the Chain of Grounded Objectives and Self-Planning.

4.2 RQ 1: How Does CGO Perform Compared to Baselines?

In this section, we evaluate the correctness of generated code across four benchmarks: HumanEval, MBPP-sanitized, and their extended versions – HumanEval+ and MBPP+. Tables 1, 2, and 3 present the *pass@1* (for $n = 10$) and *pass-ratio@10* scores, comparing CGO with various baseline reasoning methods using three different LLMs: LLaMA3-8B-Instruct, LLaMA3-70B-Instruct, and GPT-3.5-Turbo.

LLaMA3-8B-Instruct (Medium-scale LLM)

Table 1 reports the results for LLaMA3-8B-Instruct, showing the accuracy of code generated using nine different methods: Direct prompting, 1-shot, 2-shot, 3-shot, CodeCoT, zero-shot CoT, self-planning, self-pseudo, and CGO. The results demonstrate that CGO outperformed all baseline reasoning methods across all benchmarks, achieving the highest *pass@1* and *pass-ratio@10* scores.

Interestingly, despite its widespread use in LLM prompting, few-shot prompting yielded surprisingly suboptimal performance, with inconsistent results across different benchmarks. While it showed improvements in MBPP-sanitized and MBPP+, it underperformed compared to direct prompting on the HumanEval benchmarks. Overall, it did not demonstrate a clear advantage over other reasoning methods, suggesting that few-shot prompting may be less effective for code generation than anticipated.

CodeCoT improved over few-shot prompting, demonstrating the benefits of leveraging CoT, but still fell 4.9 points behind CGO in *pass@1* (57.5 vs. 62.4) and 4.1 points in *pass-ratio@10* (61.3 vs. 65.4) on HumanEval. Across the evaluated benchmarks, Zero-shot CoT often outperformed CodeCoT; however, it still lagged behind CGO, suggesting that leveraging the model's inherent reasoning abilities may be more effective than manually constructing CoT examples for code generation.

While CGO and CoT share the general principle of decomposition, they differ in their underlying reasoning processes. CoT emulates a human-like, step-by-step reasoning, whereas CGO adopts a machine-oriented reasoning approach. This mechanistic approach facilitates a

Table 1 Comparison of the CGO prompt with various baselines using the LLaMA3-8B-Instruct across four benchmarks (HumanEval, HumanEval+, MBPP-sanitized, MBPP+), evaluated using *pass*@1(n=10) and *pass-ratio*@10.

Prompts	HumanEval		MBPP-sanitized		HumanEval+		MBPP+	
	pass@1	*pass-ratio*@10	*pass*@1	*pass-ratio*@10	*pass*@1	*pass-ratio*@10	*pass*@1	*pass-ratio*@10
Direct prompting	54.4	57.3	53.5	56.7	48.5	49.9	43.2	46.6
1-shot	47.8	50.8	60.4	61.4	42.9	43.6	50.3	51.1
2-shot	47.2	50.5	64.4	65.4	42.9	43.7	54.5	55.1
3-shot	45.4	48.7	64.8	65.4	41.5	42.1	54.9	55.6
CodeCoT	57.5	61.3	63.7	64.6	51.1	52.2	52.3	53.3
Zero-shot CoT	59.8	63.4	63.8	64.7	53.7	54.3	54.0	55.6
Self-Planning	60.1	63.8	67.9	**68.8**	52.8	53.5	56.9	57.4
Self-Pseudo	52.4	55.2	48.4	49.2	45.7	46.6	39.7	40.4
CGO	**62.4**	**65.4**	**68.1**	**68.8**	**56.2**	**56.9**	**57.9**	**58.4**

Table 2 Comparison of the CGO prompt with various baselines using the LLaMA3-70B-Instruct across four benchmarks (HumanEval, HumanEval+, MBPP-sanitized, MBPP+), evaluated using *pass*@1(n=10) and *pass-ratio*@10.

Prompts	HumanEval		MBPP-sanitized		HumanEval+		MBPP+	
	pass@1	*pass-ratio*@10	*pass*@1	*pass-ratio*@10	*pass*@1	*pass-ratio*@10	*pass*@1	*pass-ratio*@10
Direct prompting	76.9	80.2	72.7	74.0	71.3	72.6	60.1	63.1
1-shot	73.0	75.6	75.3	75.7	68.4	69.5	63.9	64.6
2-shot	73.9	77.3	76.4	76.8	70.0	70.9	65.0	65.8
3-shot	74.4	77.7	73.8	74.3	70.2	71.3	63.1	64.0
CodeCoT	76.2	79.6	82.3	82.9	68.2	69.8	69.7	70.7
Zero-shot CoT	79.3	82.8	80.6	81.1	74.4	74.9	69.0	69.8
Self-Planning	79.5	82.3	81.5	82.0	71.8	72.7	67.4	68.4
Self-Pseudo	79.9	81.7	82.3	82.8	73.8	75.0	69.3	70.6
CGO	**80.3**	**83.2**	**83.3**	**83.7**	**74.6**	**75.7**	**70.4**	**71.1**

more direct alignment with automated code-generation tasks. Figure 2 presents an example of the intermediate outputs generated by CGO (objectives) and CoT (step-by-step reasoning), clearly demonstrating that CGO generates more concise intermediate results. While CoT's detailed, explanatory style is beneficial to some extent, Figure 3 shows that its emphasis on human-readable explanations introduces unnecessary verbosity, diverting focus from capturing essential functional objectives for code generation.

Among the evaluated baseline methods, self-planning achieved the highest performance, coming closest to CGO. While planning, like CoT, emphasizes procedural steps, it places less emphasis on strict step-by-step sequencing, which may contribute to its stronger performance.

Self-Pseudo follows a three-stage reasoning process, sequentially generating requirements, pseudo-code, and then the final implementation, aligning with typical software development procedures. However, its effectiveness was limited with LLaMA3-8B-Instruct, consistently recording lower scores than direct prompting (DP) across all four benchmarks. This suggests that incorporating software development process elements into code generation is less effective for an 8B-scale model, likely due to its limited capacity to process complex contextual dependencies.

Overall, CGO consistently outperformed all other methods, demonstrating that its concise, goal-oriented reasoning approach effectively enhances code generation accuracy across diverse benchmarks.

Table 3 Comparison of the CGO prompt with various baselines using the GPT-3.5-Turbo across four benchmarks (HumanEval, HumanEval+, MBPP-sanitized, MBPP+), evaluated using *pass*@1(n=10) and *pass-ratio*@10.

Prompts	HumanEval		MBPP-sanitized		HumanEval+		MBPP+	
	pass@1	*pass-ratio*@10	*pass*@1	*pass-ratio*@10	*pass*@1	*pass-ratio*@10	*pass*@1	*pass-ratio*@10
Direct prompting	71.6	75.3	84.3	84.8	67.7	68.5	71.2	71.2
1-shot	70.4	74.8	84.9	85.3	65.5	65.8	73.3	73.8
2-shot	72.0	73.9	83.6	84.1	67.7	68.0	73.3	73.7
3-shot	64.6	66.9	85.2	85.7	61.6	61.6	73.3	73.7
CodeCoT	74.2	77.2	85.6	86.3	67.8	68.6	72.2	72.6
Zero-shot CoT	72.0	74.6	85.2	85.7	67.1	67.6	71.7	72.5
Self-Planning	72.7	76.1	82.6	83.4	67.3	67.7	68.1	68.3
Self-Pseudo	**75.4**	77.1	85.4	85.5	**69.1**	69.0	72.2	73.1
CGO	74.6	**77.4**	**86.0**	**86.5**	68.5	**69.1**	**73.7**	**74.0**

LLaMA3-70B-Instruct and GPT-3.5-turbo (Large-scale LLMs)

Tables 2 and 3 show the results for LLaMA3-70B-Instruct and GPT-3.5-Turbo, both of which have greater computational capacity than LLaMA3-8B-Instruct. With these larger models, the accuracy of all methods improved compared to LLaMA3-8B-Instruct, while the overall performance trends remained similar.

As in the previous results, CGO generally outperformed other methods across benchmarks, with self-planning achieving strong performance among baselines. Zero-shot CoT continued to perform better than CodeCoT in HumanEval, but unlike in LLaMA3-8B-Instruct, CodeCoT outperformed Zero-shot CoT in some MBPP-sanitized benchmarks, suggesting that its effectiveness may depend on the characteristics of the benchmark.

A notable difference from previous results is the performance of self-pseudo, which showed improvement with larger models. While it performed worse than direct prompting in LLaMA3-8B-Instruct, it achieved better results in HumanEval with larger models, in some cases approaching CGO. This suggests that larger models are better able to integrate multi-step reasoning, making the use of pseudo-code more effective. However, CGO still outperformed Self-Pseudo in terms of *pass-ratio@10*, indicating that while generating pseudo-code can be beneficial, CGO remains a more efficient and reliable approach.

Overall, CGO demonstrated strong performance across different model sizes, showing its effectiveness as a concise, goal-oriented reasoning approach for code generation.

4.3 RQ 2. Does CGO Effective in Real-World Programming Task?

To evaluate CGO's effectiveness in real-world programming scenarios, we compared its performance with baseline reasoning-based prompting techniques on LiveCodeBench. Table 4 presents the results based on *pass@1* and *pass-ratio@10* across LLaMA3.1-8B-Instruct, LLaMA3.1-70B-Instruct, and GPT-3.5-Turbo.

In this experiment, few-Shot prompting was excluded due to the longer problem descriptions in LiveCodeBench. Including additional examples for few-Shot prompting would exceed the context length limit of LLaMA3.1 models, making it impractical for evaluation.

Experimental results show that while CodeCoT outperformed Direct Prompting (DP), its performance improvement was not particularly significant compared to other baseline methods. This observation became more pronounced as the model size increased. These findings suggest that as the diversity of problem increases, incorporating appropriate examples into the code generation process becomes increasingly important.

Table 4 Comparison of the CGO prompt with baseline methods on LiveCodeBench, using 3 LLMs. Results is evaluated based on *pass*@1(n=10) and *pass-ratio*@10.

Prompts	LLaMA3.1-8B-Instruct		LLaMA3.1-70B-Instruct		GPT-3.5-turbo	
	pass@1	*pass-ratio*@10	*pass*@1	*pass-ratio*@10	*pass*@1	*pass-ratio*@10
Direct Prompting	26.0	32.1	41.5	48.8	35.5	44.1
CodeCoT	28.0	36.1	44.7	52.4	35.7	43.6
Zero-shot CoT	29.0	36.6	47.0	54.3	39.0	48.3
Self-Planning	**30.0**	**38.1**	48.3	56.0	38.3	47.6
Self-Pseudo	25.5	33.0	48.0	55.7	39.5	48.1
CGO	28.0	34.7	**49.0**	**56.1**	**40.75**	**50.2**

Table 5 Performance comparison of baselines and CGO across difficulty levels based on *pass*@1 and *pass-ratio*@10, using the GPT-3.5-Turbo.

Prompts	easy		medium		hard	
	pass@1	*pass-ratio*@10	*pass*@1	*pass-ratio*@10	*pass*@1	*pass-ratio*@10
Direct Prompting	60.56	67.83	22.61	32.73	20.00	28.00
Self-Planning	60.56	70.92	**28.57**	39.97	16.66	25.26
CodeCoT	60.56	67.79	25.00	34.71	16.66	22.12
Zero-shot CoT	**65.49**	**73.78**	27.97	39.02	17.77	25.99
Self-Pseudo	61.97	71.29	**28.57**	40.44	15.55	25.85
CGO	**65.49**	73.48	**28.57**	**40.55**	**24.44**	**31.58**

In contrast, zero-shot CoT demonstrated greater performance improvements compared to CodeCoT, and this tendency was consistently observed across all three models. This suggests that as problem complexity and diversity increase, leveraging the model's inherent capabilities to construct a Chain of Thought (CoT) reasoning process before code generation can be a more effective approach.

Conversely, self-planning recorded the highest performance on LLaMA3.1-8B-Instruct, achieving 30.0 in *pass@1* (30.0 vs. 28.0) and 38.1 in *pass-ratio@10* (38.1 vs. 34.7), outperforming CGO by 2.0 and 4.7 points. However, on LLaMA3.1-70B-Instruct and GPT-3.5-Turbo, Self-Planning showed lower performance compared to CGO.

Self-pseudo showed lower performance than DP on LLaMA3.1-8B-Instruct, consistent with the results observed in RQ 1. However, for the other two models, it demonstrated performance comparable to or higher than that of self-planning. This variation suggests that the effectiveness of the self-pseudo approach may depend on model size and capability. These findings indicate that incorporating the software development process into the code generation pipeline remains a valid approach, even in environments with diverse problem types such as LiveCodeBench.

CGO showed lower performance than self-planning and zero-shot CoT when using the LLaMA3.1-8B-Instruct model. However, in experiments using LLaMA3.1-70B-Instruct and GPT-3.5-Turbo, CGO achieved the highest performance compared to the baseline prompts. Notably, in GPT-3.5-Turbo experiments, CGO recorded 40.75 in *pass@1* (40.75 vs. 35.5) and 50.2 in *pass-ratio@10* (50.2 vs. 44.1), 4.25 and 6.1 points higher than DP, demonstrating significant improvements.

These results suggest that CGO's goal-oriented reasoning approach is also effective in real-world problem-solving when applied to larger models such as LLaMA3.1-70B-Instruct and GPT-3.5-Turbo, rather than smaller 8B-scale models. Furthermore, it confirms CGO's advantage over baseline prompts.

Table 6 Performance comparison of baselines and CGO across difficulty levels based on *pass*@1 and *pass-ratio*@10, using the LLaMA3.1-70B-Instruct.

Prompts	easy		medium		hard	
	pass@1	*pass-ratio*@10	*pass*@1	*pass-ratio*@10	*pass*@1	*pass-ratio*@10
Direct Prompting	69.71	72.69	30.95	40.31	16.66	22.60
Self-Planning	**80.99**	**85.33**	38.10	47.09	15.56	26.52
CodeCoT	79.58	83.32	32.74	42.02	12.22	23.19
Zero-shot CoT	79.58	83.17	33.93	43.60	**20.00**	**28.83**
Self-Pseudo	78.16	82.72	38.28	47.90	16.88	28.32
CGO	80.22	82.59	**39.88**	**49.46**	17.78	26.58

4.3.1 Performance Analysis of CGO Across Difficulty Levels

This subsection provides a more detailed analysis of Table 4, examining the performance of CGO and baseline prompts across different problem difficulty levels as defined by LiveCodeBench. Table 5 presents the results, categorizing problems into three difficulty levels: easy, medium, and hard. The results in Table 5 were obtained using GPT-3.5-Turbo, while Figure 4 provides a visual representation of these findings for better readability.

Easy problems showed relatively modest performance gains across most baselines. The self-planning approach matched the baseline in *pass@1* but demonstrated improvements in *pass-ratio@10*. Zero-shot CoT achieved one of the highest scores among all baselines, along with CGO, significantly outperforming DP. Other prompting techniques showed performance levels similar to DP, with only minor enhancements.

As problem complexity increased, the performance gap among baselines became more pronounced. On medium-level problems, self-planning achieved 28.57 in *pass@1* (28.57 vs. 22.61), outperforming DP by 4.95 points. Zero-shot CoT also demonstrated improvements over DP, reinforcing the advantages of autonomous reasoning before code generation. In contrast, CodeCoT exhibited the lowest performance, indicating the need for a more refined Chain of Thought (CoT) strategy.

Several methods, including CGO, self-planning, and self-pseudo, attained the same *pass@1* score (28.57). However, in terms of *pass-ratio@10*, CGO recorded the highest value (40.55), followed by self-pseudo (40.44) and self-planning (39.97). These results suggest that while different approaches may achieve similar *pass@1* scores, *pass-ratio@10* provides additional insight into solution quality by reflecting the proportion of generated code that successfully passes more test cases, thereby offering a more comprehensive measure of correctness.

Performance degradation was more apparent in hard problems, where most baselines underperformed compared to DP. Self-planning and CodeCoT showed lower *pass@1* scores compared to DP, and this pattern was consistent in *pass-ratio@10* as well. While zero-shot CoT led among the baselines, it still fell short of DP, revealing limitations in handling higher complexity tasks. self-pseudo saw the most significant performance drop, indicating its ineffectiveness in solving more challenging problems. In contrast, CGO achieved the highest performance across all baselines, with *pass@1* and *pass-ratio@10* scores of 24.44 and 31.58, respectively. These results represent a 4.44 and 3.58 improvement over DP, demonstrating CGO's effectiveness not only in lower difficulty levels but also in more complex problem-solving scenarios.

Table 6 shows the performance of the prompting methods, including CGO, across difficulty levels using the LLaMA3.1-70B-Instruct. CGO achieved strong performance on medium-difficulty problems – which constitute the majority of the benchmark – achieving the highest

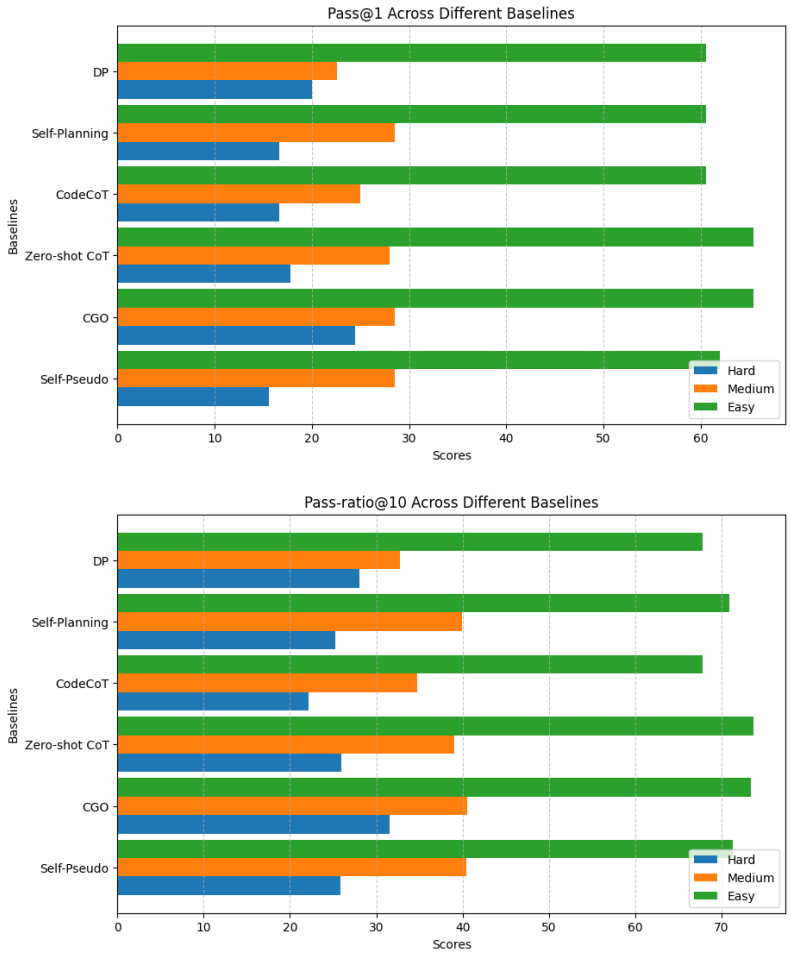


Figure 4 *pass@1* and *pass-ratio@10* scores by difficulty level on the GPT-3.5-turbo experiment.

scores of 39.88 for *pass@1* and 49.46 for *pass-ratio@10*, indicating its effectiveness in addressing problems of intermediate complexity. On easy and hard problems, although CGO did not attain the highest scores, it still ranked second in terms of *pass@1*, closely behind self-planning on easy problems and zero-shot CoT on hard problems, indicating a stable level of performance across different complexities.

The results show that CGO performs similarly to other baselines on easy, medium difficulty problems, while achieving better results on hard problems. This suggests that CGO's objective-driven approach provides useful guidance, particularly for more complex tasks. Further studies could investigate whether integrating CGO with other reasoning approaches improves performance, particularly on hard problems where additional structured reasoning may be advantageous.

4.4 RQ 3. Does CGO Resource-Efficient Compared to the Baselines?

All prompting methods examined in this study follow a multi-step reasoning process, where the LLM first autonomously generates intermediate supplementary content (e.g., objectives, plans, chain-of-thought, or pseudo-code) before generating the final code. While this additional content helps guide the code generation process, it also increases token generation, thereby

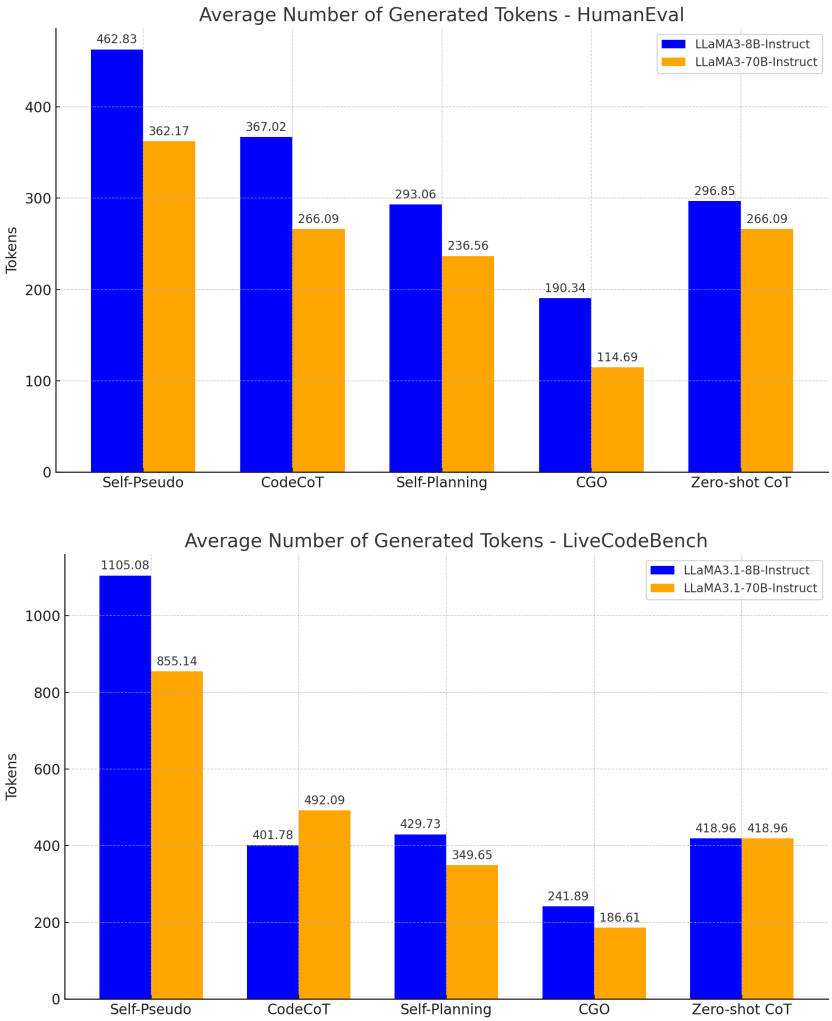


Figure 5 Average number of generated tokens by LLaMA3.1-8B-Instruct and LLaMA3.1-70B-Instruct on the HumanEval and LiveCodeBench across 5 baseline prompts (lower is better).

raising the computational cost of using LLMs. Since LLM inference cost scales with the number of generated tokens, it is essential to evaluate the trade-off between enhanced guidance and resource efficiency.

In this section, we evaluate the resource efficiency of each baseline by measuring the token count of the intermediate supplementary content generated by the LLM. Figure 5 presents the average token count across baselines on the HumanEval and LiveCodeBench benchmarks, serving as a proxy for computational cost. The results show that CGO consistently generates the fewest tokens, making it the most resource-efficient approach among the compared baselines.

On HumanEval, CGO produces 114.69 tokens on average for LLaMA3-70B-Instruct and 190.34 tokens for LLaMA3-8B-Instruct, significantly lower than other baselines. In contrast, self-pseudo requires the most tokens (462.83 for LLaMA3-8B-Instruct and 362.17 for LLaMA3-70B-Instruct), followed by CodeCoT and self-planning.

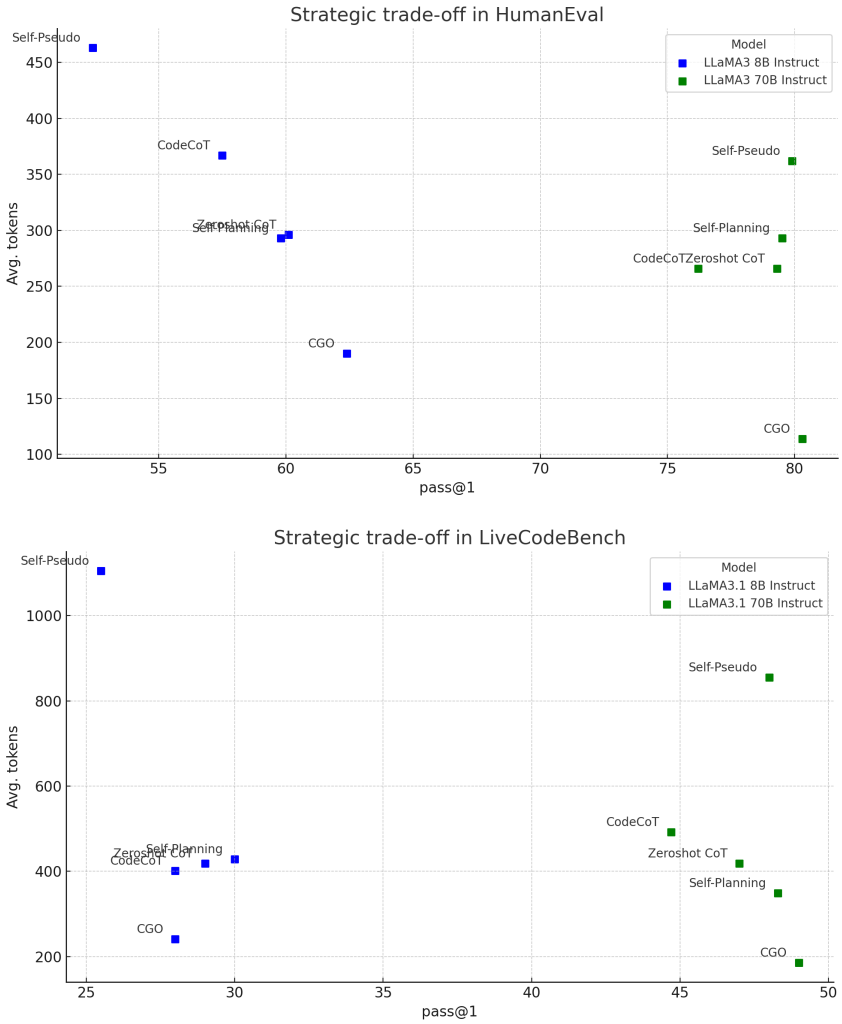


Figure 6 Strategic trade-off on the HumanEval and LiveCodeBench across 5 baseline prompts.

This pattern is observed on LiveCodeBench, where CGO remains the most efficient, generating 186.61 tokens for LLaMA3.1-70B-Instruct and 241.89 tokens for LLaMA3.1-8B-Instruct. self-pseudo again exhibits the highest token consumption (1105.08 and 855.14, respectively), with CodeCoT and self-planning also showing significantly higher token usage than CGO.

Interestingly, for most cases, the LLaMA3.1-8B-Instruct generates more tokens than the LLaMA3.1-70B-Instruct under the same benchmark and prompting technique. A possible explanation is that the larger LLaMA3.1-70B-Instruct, having a greater capacity to process and distill information, tends to generate more concise outputs. Consequently, it may require fewer tokens to convey the necessary details for code generation.

Overall, these findings highlight CGO as the most resource-efficient approach among the evaluated baselines. By emphasizing concise functional objectives, CGO achieves an effective balance between token usage and guidance quality, thereby minimizing computational overhead without compromising performance, as further demonstrated in Figure 6. Its primary

Table 7 Performance comparison of CGO and requirement-based prompting with the LLaMA3.1-70B-Instruct, evaluated on HumanEval and LiveCodeBench using *pass*@1 and *pass-ratio*@10.

Prompts	HumanEval		LiveCodeBench	
	pass@1	*pass-ratio*@10	*pass*@1	*pass-ratio*@10
CGO-original	**80.3**	83.2	**49.0**	56.1
CGO-variant	79.27	**83.9**	48.5	**56.19**

advantage lies in its ability to generate only the essential contextual elements required for code generation, making it particularly well-suited for scenarios where computational efficiency is a priority.

4.5 Wording Variations in CGO Prompting

The CGO approach does not strictly require the use of the word "objective" in its prompts. For example, terms such as "objective" and "requirement" convey similar meanings within the context of coding. To evaluate whether minor wording differences could significantly influence the effectiveness of the CGO approach, we conducted an experiment.

In this experiment, we compared two variations of prompt wording used during the intermediate output generation stage:

- Original Prompt: *"Write **objectives** to solve the problem"*
- Variant Prompt: *"Write **requirements** to solve the problem"*

We assessed the performance of code generation on the HumanEval and LiveCodeBench benchmarks. The experimental results are summarized in Table 7.

The results reveal negligible differences between the two prompt variants, with variations within approximately 1% in both *pass@1* and *pass-ratio@10* scores. These findings indicate that CGO's effectiveness is consistently good regardless of minor semantic differences in prompt wording, allowing developers the flexibility to choose synonymous terms without compromising performance.

4.6 Limitation

Our study demonstrates the efficiency and effectiveness of CGO, but certain aspects remain unexplored:

- **Model Selection**: The evaluation was conducted on five LLMs (LLaMA3-series, and GPT-3.5-Turbo), providing insights into CGO's effectiveness but not covering the full range of recent models. Future research will extend experiments to Qwen, DeepSeek, and other emerging LLMs to assess CGO's broader applicability.
- **Sampling Strategy**: We used greedy sampling for all experiments, ensuring deterministic outputs but not considering sampling-based approaches such as nucleus sampling. Since different sampling strategies can influence both output diversity and efficiency, future work will explore their impact on CGO.
- **Objective Generation**: CGO generates objectives using a zero-shot approach, which may lead to variations in format, detail, or clarity across different instances. Employing methods such as few-shot prompting or structured guidance could improve consistency and standardization in the objectives produced. Additionally, systematic evaluations involving human or automated approaches (e.g., "LLM-as-judge" [7]) would further help assess the quality and appropriateness of generated objectives.

- **Evaluation Methodology**: Our evaluation primarily focused on functional correctness and performance metrics. Additional criteria such as readability, maintainability, and robustness against edge cases need to be considered.

Addressing these aspects will further refine CGO and enhance its adaptability to different models and inference settings.

5 Conclusion

This paper introduced Chain of Grounded Objectives (CGO), a concise goal-oriented reasoning approach that succinctly captures the objectives of a coding problem. CGO autonomously generates comment-style objectives, aligning well with programming language conventions while offering clear guidance without excessive verbosity.

Empirical evaluations on HumanEval, MBPP-sanitized, and their extended versions (HumanEval+ and MBPP+) demonstrate that CGO consistently achieves high accuracy, outperforming baseline prompting techniques in *pass@1* and *pass-ratio@10*. These results indicate that CGO enables LLMs to generate correct solutions without requiring intricate reasoning steps.

On LiveCodeBench, CGO also demonstrated competitive performance, particularly on medium and hard difficulty problems, underscoring its applicability to real-world programming tasks. In addition to accuracy, CGO exhibited the highest efficiency, often reducing the number of intermediate tokens generated by more than half compared to baseline prompts. This reduction minimizes inference costs while sustaining competitive performance, making CGO a cost-effective approach for LLM-based code generation.

Overall, CGO presents a practical and efficient approach to guiding LLMs in code generation. Future work could investigate applying CGO to specialized reasoning models such as DeepSeek-R1 or OpenAI o3 to further evaluate and enhance its effectiveness.

References

1 Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint*, 2023. arXiv:2303.08774.

2 Eshaan Agarwal, Joykirat Singh, Vivek Dani, Raghav Magazine, Tanuja Ganu, and Akshay Nambi. Promptwizard: Task-aware prompt optimization framework. *arXiv preprint*, 2024. arXiv:2405.18369.

3 Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint*, 2021. arXiv:2108.07732.

4 Tom B Brown. Language models are few-shot learners. *arXiv preprint*, 2020. arXiv:2005.14165.

5 Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022 . doi:10.48550/arXiv.2207.10397.

6 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. arXiv:2107.03374.

7 Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Saizhuo Wang, Kun Zhang, Yuanzhuo Wang, Wen Gao, Lionel Ni, and Jian Guo. A survey on llm-as-a-judge, 2025. arXiv:2411.15594.

8 Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022. doi:10.48550/arXiv.2203.03850.

9 Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

10 Hojae Han, Jaejin Kim, Jaeseok Yoo, Youngwon Lee, and Seung-won Hwang. Archcode: Incorporating software requirements in code generation with large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13520–13552, 2024. doi:10.18653/V1/2024.ACL-LONG.730.

11 Dong Huang, Qingwen Bu, and Heming Cui. Codecot and beyond: Learning to program and test like a developer. *arXiv preprint arXiv:2308.08784*, 2023. doi:10.48550/arXiv.2308.08784.

12 Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint*, 2024. arXiv:2409.12186.

13 Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024. doi:10.48550/arXiv.2403.07974.

14 Shashank Mohan Jain. Hugging face. In *Introduction to transformers for NLP: With the hugging face library and models to solve problems*, pages 51–67. Springer, 2022.

15 Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023. doi:10.48550/arXiv.2310.06825.

16 Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30, 2024. doi:10.1145/3672456.

17 Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners, 2023. doi:10.48550/arXiv.2205.11916.

18 Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 2023.

19 Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022. doi:10.1126/science.abq1158.

20 Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 2024.

21 Noble Saji Mathews and Meiyappan Nagappan. Test-driven development and llm-based code generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pages 1583–1594. ACM, October 2024. doi:10.1145/3691620.3695527.

22 Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint*, 2022. arXiv:2203.13474.

23 Zhihong Shao, Yeyun Gong, Yelong Shen, Minlie Huang, Nan Duan, and Weizhu Chen. Synthetic prompting: Generating chain-of-thought demonstrations for large language models. In *International Conference on Machine Learning*, pages 30706–30775. PMLR, 2023. URL: https://proceedings.mlr.press/v202/shao23a.html.

24 Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint*, 2023. arXiv:2312.11805.

25 Zhao Tian, Junjie Chen, and Xiangyu Zhang. Test-case-driven programming understanding in large language models for better code generation. *arXiv preprint*, 2023. arXiv:2309.16120.

26 Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023. doi:10.48550/arXiv.2302.13971.

27 Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023. doi:10.48550/arXiv.2307.09288.

28 Prashant Trivedi, Souradip Chakraborty, Avinash Reddy, Vaneet Aggarwal, Amrit Singh Bedi, and George K Atia. Align-pro: A principled approach to prompt optimization for llm alignment. *arXiv preprint*, 2025. doi:10.48550/arXiv.2501.03486.

29 Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint*, 2021. arXiv:2109.00859.

30 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

31 Sangyeop Yeo, Seung-Won Hwang, and Yu-Seung Ma. Chain-of-Grounded-Objectives. Software, version 1.0., swhId: swh:1:dir:315704008a9f9a515047943266a959518485cbb0 (visited on 2025-06-10). URL: https://github.com/DDIDUs/Chain-of-Grounded-Objectives, doi:10.4230/artifacts.23374.

32 Sangyeop Yeo, Yu-Seung Ma, Sang Cheol Kim, Hyungkook Jun, and Taeho Kim. Framework for evaluating code generation ability of large language models. *ETRI Journal*, 46(1):106–117, 2024.

33 Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493*, 2022. doi:10.48550/arXiv.2210.03493.

34 Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x, 2024. arXiv:2303.17568.

35 Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint*, 2024. arXiv:2406.11931.

A Experimental Details

In our experiments, we employed LLaMA-3 and LLaMA-3.1 based models, and GPT-3.5-Turbo-16k-0106. Both objective generation and code generation were carried out using greedy sampling with a temperature of 0 and a top-p of 1, following prior work [10, 5].

The MBPP prompts were standardized to align with the HumanEval format. To ensure consistency, function signatures were extracted directly from the ground-truth code. Examples of the standardized formatting are presented in Table 7.

To increase the diversity of outputs, we generated $n = 10$ objective samples per problem, which were subsequently used to guide the code generation process.

For the implementation of *Chain-of-Thought (CoT)* [23, 33] and *Self-Planning* strategies, reasoning chains were constructed based on code examples provided in the original Self-Planning approach. Instead of directly reusing existing reasoning sequences, we manually curated new examples tailored to our experimental configuration.

Few-shot prompting was realized by randomly selecting example problems and their corresponding solutions from the same dataset. Each problem description was labeled as *Input* and the corresponding solution as *Output*, which were incorporated into the prompt as reference examples.

The execution of the generated code was configured to run in Docker sandbox environment.

All experiments were conducted using the Huggingface [14] and LangChain, LangGraph libraries. All resources are publicly available on GitHub.

B Evaluation Details

We evaluated the experimental results using *pass@k* and *pass-ratio@n* metrics.

B.1 *pass@k*

The *pass@k* metric assesses the performance of an LLM in code generation by evaluating the execution outcomes of the generated code. For a given coding problem, *pass@k* indicates whether at least one of the k code solutions produced by the LLM successfully passes all associated test cases. A problem is deemed solved by the LLM if any of the k solutions satisfy this condition.

$$pass@k := \mathop{\mathbb{E}}_{problems}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{4}$$

B.2 *pass-ratio@n*

For a given solution indexed by i $(0 < i \leq n)$, $pass\text{-}ratio_i$ is computed as the square of the ratio between the number of passed test cases for solution i and the total number of test cases. Squaring this value assigns greater weight to solutions with higher pass ratios, thereby emphasizing solutions with greater accuracy.

$$pass\text{-}ratio_i = \left(\frac{\text{Number of passed test cases } i}{\text{Total number of test cases}}\right)^2 \tag{5}$$

The *pass-ratio@n* is calculated using the formula below. It represents the average *pass-ratio* of the n generated codes.

$$pass\text{-}ratio@n = \frac{\sum_{i=1}^{n} pass\text{-}ratio_i}{n} \tag{6}$$

HumanEval Prompt Example

```
def separate_paren_groups(paren_string: str) -> List[str]:
    """ Input to this function is a string containing multiple groups
        of nested parentheses.
    >>> separate_paren_groups('( ) (( )) (( )( ))')
    ['()', '(())', '(()())']
    """
```

MBPP-sanitized Prompt Example

```
def first_repeated_char(str1):
    '''Write a python function to find the first repeated character
       in a given string.'''
```

Figure 7 Examples of HumanEval-style Prompt (HumanEval, MBPP-sanitized).

Table 8 Examples of Objective and Code Generation Using CGO.

Problem Description Example

```
def separate_paren_groups(paren_string: str) -> List[str]:
    """ Input to this function is a string containing multiple
        groups of nested parentheses. Your goal is to
    separate those group into separate strings and return the
        list of those.
    Separate groups are balanced (each open brace is properly
        closed) and not nested within each other
    Ignore any spaces in the input string.
    >>> separate_paren_groups('( ) (( )) (( )( ))')
    ['()', '(())', '(()())']
    """
```

Objectives Generation Prompt

Write objectives to solve the problem.

Objectives Generation Examples

- Separate groups of nested parentheses should be identified and extracted from the input string.
- Ignore any spaces in the input string.
- Ensure that each extracted group is balanced (each open brace is properly closed).
- Extracted groups should not be nested within each other.

Generated Code

```
def separate_paren_groups(paren_string: str) -> List[str]:
    result = []
    stack = []
    current_group = ""

    for char in paren_string:
        if char == '(':
            stack.append('(')
            current_group += '('
        elif char == ')':
            stack.pop()
            current_group += ')'
            if not stack:
                result.append(current_group)
                current_group = ""

    return result
```

Table 9 Examples of Code Generation Using Few-shot.

Problem Description Example

```
def separate_paren_groups(paren_string: str) -> List[str]:
    """ Input to this function is a string containing multiple
        groups of nested parentheses. Your goal is to
    separate those group into separate strings and return the
        list of those.
    Separate groups are balanced (each open brace is properly
        closed) and not nested within each other
    Ignore any spaces in the input string.
    >>> separate_paren_groups('( ) (( )) (( )( ))')
    ['()', '(())', '(()())']
    """
```

Few-shot Examples

```
### Input
def minSubArraySum(nums):
    """
    Given an array of integers nums, find the minimum sum of any
        non-empty sub-array of nums.
    Examples:
        minSubArraySum([2, 3, 4, 1, 2, 4]) == 1
        minSubArraySum([-1, -2, -3]) == -6
    """

###Output
def minSubArraySum(nums):
    max_sum = 0
    s = 0
    ...
```

Few-shot Code Generation Prompt

Write python code for the problem.

Generated Code

```
groups = []
current_group = ""
for char in paren_string:
    if char == '(':
        current_group += char
    elif char == ')':
        current_group += char
        groups.append(current_group)
        current_group = ""
return groups
```

Table 10 Examples of Code Generation Using CodeCoT.

Problem Description Example

```
def separate_paren_groups(paren_string: str) -> List[str]:
    """ Input to this function is a string containing multiple
        groups of nested parentheses. Your goal is to
    separate those group into separate strings and return the
        list of those.
    Separate groups are balanced (each open brace is properly
        closed) and not nested within each other
    Ignore any spaces in the input string.
    >>> separate_paren_groups('( ) (( )) (( )( ))')
    ['()', '(())', '(()())']
    """
```

CoT Generation Prompt

Generate detailed comments for this code.

CoT Generation Examples

- The input string contains groups of nested parentheses separated by spaces.
- Split the input string by spaces to handle each group individually.
- Remove any extra spaces to ensure proper processing.
- Assume each group is balanced and valid.
- Iterate through groups, filtering out empty or invalid ones.
- Collect remaining valid groups into a list.
- Return the filtered list.
- This method assumes no nested validation beyond splitting.

Generated Code

```
groups = paren_string.split(' ')
filtered_groups = [group.strip() for group in groups if group.
    strip()]
return filtered_groups

print(separate_paren_groups('( ) (( )) (( )( ))'))  # Output:
    ['()', '(())', '(()())']
```

Table 11 Examples of Code Generation Using Self-Planning.

Problem Description Example

```
def separate_paren_groups(paren_string: str) -> List[str]:
    """ Input to this function is a string containing multiple
        groups of nested parentheses. Your goal is to
    separate those group into separate strings and return the
        list of those.
    Separate groups are balanced (each open brace is properly
        closed) and not nested within each other
    Ignore any spaces in the input string.
    >>> separate_paren_groups('( ) (( )) (( )( ))')
    ['()', '(())', '(()())']
    """
```

Plan Generation Prompt

Write plan for the problem.

Plan Generation Examples

(1) Preprocess the input string:
- **(a)** Remove all spaces to simplify processing.

(2) Initialize variables:
- **(a)** List `groups` to store the final balanced groups.
- **(b)** String `temp` to construct the current group.
- **(c)** Integer `balance` to track open and close parentheses.

(3) Iterate through the cleaned string:
- **(a)** Add the character to `temp`.
- **(b)** Adjust `balance` based on whether the character is `(` or `)`.
- **(c)** If `balance == 0`, add `temp` to `groups` and reset `temp`.

(4) Return the list `groups` containing all balanced groups.

Generated Code

```
cleaned_string = paren_string.replace(" ", "")
groups = []
temp = ""
balance = 0
for char in cleaned_string:
    temp += char
    if char == '(':
        balance += 1
    elif char == ')':
        balance -= 1
        if balance == 0:
            groups.append(temp)
            temp = ""

return groups
```

Table 12 Examples of Code Generation Using Pseudo-Code.

Problem Description Example

```
def separate_paren_groups(paren_string: str) -> List[str]:
    """ Input to this function is a string containing multiple
        groups of nested parentheses. Your goal is to
    separate those group into separate strings and return the
        list of those.
    Separate groups are balanced (each open brace is properly
        closed) and not nested within each other
    Ignore any spaces in the input string.
    >>> separate_paren_groups('( ) (( )) (( )( ))')
    ['()', '(())', '(()())']
    """
```

Pseudo-code

```
Function separate_paren_groups(input_string):
    Strip spaces from input_string
    Initialize result as empty list
    Initialize current_group as empty string
    Set balance = 0
    For each char in input_string:
        Append char to current_group
        If char is '(':
            increment balance
        If char is ')':
            decrement balance
        If balance == 0:
            Append current_group to result
            Reset current_group to empty string
    Return result
```

Generated Code

```
cleaned_string = paren_string.replace(" ", "")
groups = []
temp = ""
balance = 0
for char in cleaned_string:
    temp += char
    if char == '(':
        balance += 1
    elif char == ')':
        balance -= 1
        if balance == 0:
            groups.append(temp)
            temp = ""
return groups
```

Efficient Neural Network Verification via Order Leading Exploration of Branch-and-Bound Trees

Guanqin Zhang
University of New South Wales, Sydney, Australia
CSIRO's Data61, Sydney, Australia

Kota Fukuda
Kyushu University, Fukuoka, Japan

Zhenya Zhang
Kyushu University, Fukuoka, Japan
National Institute of Informatics, Tokyo, Japan

H.M.N. Dilum Bandara
CSIRO's Data61, Sydney, Australia
University of New South Wales, Sydney, Australia

Shiping Chen
CSIRO's Data61, Sydney, Australia
University of New South Wales, Sydney, Australia

Jianjun Zhao
Kyushu University, Fukuoka, Japan

Yulei Sui
University of New South Wales, Sydney, Australia

Abstract

The vulnerability of neural networks to adversarial perturbations has necessitated formal verification techniques that can rigorously certify the quality of neural networks. As the state-of-the-art, branch-and-bound (`BaB`) is a "divide-and-conquer" strategy that applies off-the-shelf verifiers to sub-problems for which they perform better. While `BaB` can identify the sub-problems that are necessary to be split, it explores the space of these sub-problems in a naive "first-come-first-served" manner, thereby suffering from an issue of inefficiency to reach a verification conclusion. To bridge this gap, we introduce an order over different sub-problems produced by `BaB`, concerning with their different likelihoods of containing counterexamples. Based on this order, we propose a novel verification framework Oliva that explores the sub-problem space by prioritizing those sub-problems that are more likely to find counterexamples, in order to efficiently reach the conclusion of the verification. Even if no counterexample can be found in any sub-problem, it only changes the order of visiting different sub-problems and so will not lead to a performance degradation. Specifically, Oliva has two variants, including Oliva$^{\mathsf{GR}}$, a greedy strategy that always prioritizes the sub-problems that are more likely to find counterexamples, and Oliva$^{\mathsf{SA}}$, a balanced strategy inspired by simulated annealing that gradually shifts from exploration to exploitation to locate the globally optimal sub-problems. We experimentally evaluate the performance of Oliva on 690 verification problems spanning over 5 models with datasets `MNIST` and `CIFAR-10`. Compared to the state-of-the-art approaches, we demonstrate the speedup of Oliva for up to 25× in `MNIST`, and up to 80× in `CIFAR-10`.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Software and its engineering → Software testing and debugging

Keywords and phrases neural network verification, branch and bound, counterexample potentiality, simulated annealing, stochastic optimization

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.36

Supplementary Material *Software*: `https://github.com/DeepLearningVerification/Oliva`
Software (ECOOP 2025 Artifact Evaluation approved artifact):
`https://doi.org/10.4230/DARTS.11.2.11`

Funding *Guanqin Zhang*: CSIRO's Data61 PhD Scholarship
Kota Fukuda: JST BOOST Grant No. JPMJBS2406
Zhenya Zhang: JST BOOST Grant No. JPMJBY24D7 and JSPS Grant No. JP25K21179
Jianjun Zhao: JSPS KAKENHI Grant No. JP23H0337
Yulei Sui: Australian Research Council Grants No. FT220100391 and No. DP250101396

© Guanqin Zhang, Kota Fukuda, Zhenya Zhang, H.M.N. Dilum Bandara, Shiping Chen, Jianjun Zhao, and Yulei Sui;
licensed under Creative Commons License CC-BY 4.0

39th European Conference on Object-Oriented Programming (ECOOP 2025).
Editors: Jonathan Aldrich and Alexandra Silva; Article No. 36; pp. 36:1–36:29

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The rapid advancement of artificial intelligence (AI) has propelled the state-of-the-art across various fields, including computer vision, natural language processing, recommendation systems, etc. Recently, there is also a trend that adopts AI products in safety-critical systems, such as autonomous driving systems, in which neural networks are used in the perception module to perceive external environments. In this type of application, unexpected behaviors of neural networks can bring catastrophic consequences and intolerable social losses; given that neural networks are notoriously vulnerable to deliberate attacks or environmental perturbations [28, 18], effective quality assurance techniques are necessary in order to expose their defects timely before their deployment in the real world.

Formal verification is a rigorous approach that can ensure the quality of target systems by providing mathematical proofs on conformance of the systems with their desired properties. In the context of neural networks, formal verification aims to certify that a neural network, under specific input conditions, will never violate a pre-defined specification regarding its behavior, such as safety and robustness, thereby providing a rigorous guarantee that the neural network can function as expected in real-world applications. With the growing emphasis on AI safety, neural network verification has emerged as a prominent area of research, leading to the development of innovative methodologies and tools.

As a straightforward approach, exact encoding solves the neural network verification problem by encoding the inference process and specifications to be logical constraints and applying off-the-shelf or dedicated solvers, such as MILP solvers [47] and SMT solvers [23], to solve the problem. However, this approach can be very slow due to the non-linearity of neural network inferences and so they are not scalable to neural networks of large sizes. In contrast, approximated methods [44, 41, 49] that over-approximate the reachable region of neural networks are more efficient: once the over-approximation satisfies the specification, the original output must also satisfy; however, if the over-approximation violates the specification, it does not indicate that the original output also violates, and in this case it may raise a *false alarm*, i.e., a specification violation that actually does not exist.

To date, *Branch-and-Bound* (`BaB`) [5] is the state-of-the-art verification approach that overarches multiple advanced verification tools, such as $\alpha\beta$-Crown [51] and Marabou [52]. It is essentially a "divide-and-conquer" strategy, and is often jointly used with off-the-shelf approximated verifiers due to their great efficiency. Given a verification problem, it first applies an approximated verifier to the problem, and splits it to sub-problems if the verifier raises a false alarm, until all the sub-problems are verified or a real counterexample is detected, as a witness of specification violation. As application of approximated verifiers to sub-problems leads to less over-approximation, `BaB` can thus resolve the issue of false alarms of approximated verifiers and outperform their plain application to the original problem.

Motivations

As a "divide-and-conquer" strategy, `BaB` can produce a large space that consists of quantities of sub-problems, especially for verification tasks that are reasonably difficult. However, when exploring this space, the classic `BaB` adopts a naive "first come, first served" strategy, which ignores the importance of different sub-problems and thus is not efficient to reach a verification conclusion. Notably, different sub-problems produced by `BaB` are not equally important – with a part of the sub-problems, it is more likely to find a real counterexample that can show the violation of the specification, and thereby we can reach a conclusion for the verification efficiently without going through the remaining sub-problems.

Contributions

In this paper, we propose a novel verification approach Oliva, which is an **o**rder **l**eading **i**ntelligent **v**erification technique for **a**rtificial neural networks.

We first define an order called *counterexample potentiality* over the different sub-problems produced by BaB. Our order estimates *how likely* a sub-problem is to contain a counterexample, based on the following two attributes: **1.** the level of problem splitting of the sub-problem, which implies how much approximation the approximated verifier needs to perform. The less approximation there is, the more likely the verifier can find a real counterexample; **2.** a quantity obtained by applying approximated verifiers to the sub-problem, which is an indicator of *the degree* to which the neural network satisfies/violates the specification. By these two attributes, we define the counterexample potentiality order over the sub-problems, as a proxy to suggest their likelihood of containing counterexamples and serve as guidance for our verification approach.

Then, we devise our approach Oliva that exploits the order to explore the sub-problem space. In general, Oliva favors the sub-problems that are more likely to contain counterexamples. Once it can find a real counterexample, it can immediately terminate the verification and return with a verdict that the specification can be violated; even if it cannot find such a counterexample, after visiting all the sub-problems, it can still manage to certify the neural network without a significant performance degradation.

To exemplify the power of our order-guided exploration, we propose two variants of Oliva:

- Oliva^{GR} employs a greedy exploitation strategy, which always prioritizes the sub-problems that have a higher likelihood of containing counterexamples. This approach focuses on the most suspicious regions of the sub-problem space, and is likely to quickly expose the counterexamples such that the verification can be concluded quickly;
- To avoid overfitting of our proposed order, we also devise Oliva^{SA} inspired by simulated annealing [24], the famous stochastic optimization technique. The approach works similarly to simulated annealing: it maintains a variable called *temperature* that keeps decreasing throughout the process, which can control the trade-off between "exploitation" and "exploration". At the initial stage, the temperature is high, and so the algorithm allows more chances of exploring the sub-problems that are not promising; as the temperature goes down, it converges to the optimal sub-problem. As the exploration of the search space has been done at the early stage of the algorithm, it is likely to converge to the globally optimal sub-problems and thereby find counterexamples. In this way, we mitigate the potential issue of being too greedy, and aim to strike a balance between "exploitation" and "exploration" in the search for suspicious sub-problems.

Evaluation

To evaluate the performance of Oliva, we perform a large scale of experiments on 690 problem instances spanning over 5 neural network models associated with the commonly-used datasets `MNIST` and `CIFAR-10`. By a comparison with the state-of-the-art verification approaches, we demonstrate the speedup of Oliva, for up to 25× in `MNIST`, and up to 80× in `CIFAR-10`. Moreover, we also show the breakdown results of Oliva for problem instances that are finally certified and the instances that are finally falsified. Experimental results show that Oliva is particularly efficient for those instances that are falsified, which demonstrates the effectiveness of our approach.

For a verification problem whose result is unknown beforehand, it is always desired to reach the conclusion as quickly as possible. Given that verification of neural networks is typically time- and resource-consuming, our approach provides a meaningful way to accelerate

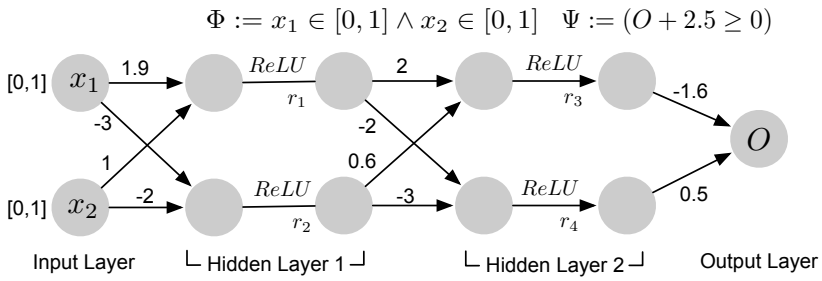


Figure 1 A neural network N and specification.

the verification process. While performing verification of neural networks with the aim of finding counterexamples sounds similar to approaches like testing or adversarial attacks, our approach differs fundamentally from those approaches, in the sense that, while those approaches deal with a single input each time and so they can never exhaust the search space, our approach deals with sub-problems that are finitely many, and so we can finally provide rigorous guarantees for specification satisfaction of neural networks.

Paper organization

The rest of the paper is organized as follows: §2 overviews our approach by using an illustrative example; §3 introduces the necessary technical background; §4 presents the proposed incremental verification approach; §5 presents our experimental evaluation results; §6 discusses related work. Conclusion and future work are presented in §7.

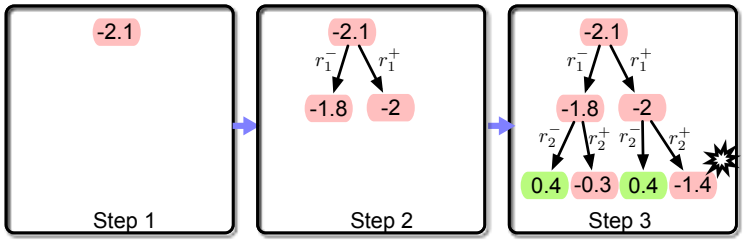
2 Overview of The Proposed Approach

In this section, we use an example to illustrate how the proposed approach solves neural network verification problems.

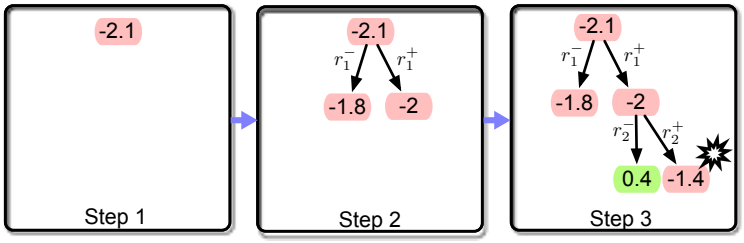
2.1 Verification Problem and BaB Approach

Fig. 1 depicts a (feed-forward) neural network N. It has an input layer, an output layer, and two *hidden layers* that are *fully-connected*, namely, the output of each hidden layer is computed by taking the weighted sum of the output of the previous layer, and applying the ReLU activation function. The output O of the neural network is computed by taking only the weighted sum of the output of the second hidden layer (without activation function). The weights of each layer are as labeled in Fig. 1. This neural network N is expected to satisfy such a specification: for any input $(x_1, x_2) \in [0,1] \times [0,1]$, it should hold that the output $O + 2.5 \geq 0$. Verification aims to give a formal proof to certify that N indeed satisfies the specification, or give a counterexample instead if N does not satisfy the specification.

As the state-of-the-art neural network verification approach, Branch-and-Bound (`BaB`) [5] has overarched several famous verification tools, such as $\alpha\beta$-Crown [51]. The application of `BaB` to neural network verification often relies on the combination with off-the-shelf verifiers, and a common choice involves the family of approximated verifiers. These verifiers can efficiently decide whether the neural network satisfies the specification, by constructing a convex over-approximation of neural network outputs: if the over-approximated output satisfies the specification, the original output must also satisfy; however, if not, it does not indicate that the original output also violates the specification, and so it may raise a *false alarm* that reports a specification violation that is actually not existent.



(a) BaB's process for verification.



(b) Oliva's process for verification.

Figure 2 Neural network verification problem and solution via BaB (BaB *vs.* Oliva).

BaB is introduced to solve this problem. It is essentially a "divide-and-conquer" strategy that splits the problem when necessary and applies approximated verifiers to sub-problems to reduce the occurrences of false alarms. In this context, a problem can be split by predicating over the sign of the input of a ReLU function (i.e., by allowing the ReLU's input to be positive or negative only), such that it can be decomposed to two linear functions each of which is easier to handle. BaB decides whether a (sub-)problem is needed to be split further, based on a value $\hat{p}$ returned from the approximated verifier. Intuitively, $\hat{p}$ indicates *how far* the specification is from being violated by the over-approximation of a (sub-)problem. If $\hat{p}$ is positive, it implies that problem has been verified and so there is no need to split it; otherwise, it implies that the problem cannot be verified, and so BaB needs to check whether it is a false alarm, by checking whether the counterexample reported by the verifier is a spurious one. Fig. 2a illustrates how BaB solves the verification problem in Fig. 1:

Step 1 – BaB first applies a verifier to the original verification problem, which returns a negative $\hat{p} = -2.1$. By validating the counterexample reported by the verifier, BaB identifies that it is a false alarm and decides to split the problem;

Step 2 – BaB splits the problem identified by the root node, and applies the verifier to the two sub-problems respectively. Again, it identifies that the verifier raises a false alarm for each of the sub-problems. Therefore, it needs to split both sub-problems;

Step 3 – Similarly, BaB applies the verifier in turn to the newly expanded sub-problems. It manages to verify two sub-problems (i.e., the two nodes both with $\hat{p} = 0.4$), and also identifies a false alarm (i.e., the node with $\hat{p} = -0.3$). In the node with $\hat{p} = -1.4$, BaB finds that the counterexample associated with this sub-problem is a real one; because having one such real counterexample suffices to show that the specification can be violated, BaB terminates the verification at this point with a verdict that the specification is violated.

2.2 The Proposed Approach

For a verification problem that is considerably difficult, BaB can often produce a space that contains a large number of sub-problems. However, as shown in Fig. 2a, the classic BaB explores these sub-problems by a naive "first come, first served" order, which ignores the difference of the sub-problems in terms of their importance, and thus can be very inefficient.

To bridge this gap, we propose an approach that explores the space of sub-problems guided by an order of importance of different sub-problems. Specifically, we identify the importance of different sub-problems by their likelihood of containing counterexamples, in the sense that the more likely a sub-problem contains counterexamples, it should be more prioritized in the exploration of the sub-problem space – once we can find a counterexample there, we can immediately terminate the verification and draw a conclusion. Even if it does not manage to find a counterexample in any problem, it just visits the sub-problems in a different order from the original BaB, so it still would not be much slower.

Fig. 2b demonstrates how our approach solves the same verification problem in Fig. 1.

- In early stages, Oliva works the same as BaB, i.e., it visits the (sub-)problems and obtains the same feedback $\hat{p}$ from the verifier, and based on that, it decides to split the (sub-)problems and apply verifiers to sub-problems;
- The difference comes from the rightmost plot of Fig. 2b, in which Oliva prioritizes the sub-problem that has $\hat{p} = -2$ rather than its sibling that has $\hat{p} = -1.8$ to expand. This is because $\hat{p}$ is a indicator that reflects how far the sub-problem is from being violated (based on the over-approximation performed by the verifier), and in this case $\hat{p} = -2$ signifies that this sub-problem has been violated *more* than the other and so it is more likely to find a counterexample there. Indeed, Oliva manages to find a real counterexample in one of its sub-problems, thereby terminating the verification immediately after that.

By this strategy, we manage to save two visits to sub-problems compared to BaB, each of which consists of an expensive problem solving process, and therefore, it reaches the conclusion of the problem more efficiently than BaB.

A variant of our approach inspired by simulated annealing

Essentially, our proposed approach changes the naive "breadth-first" exploration of the sub-problem space in the classic BaB to an intelligent way guided by the likelihood of finding counterexamples in different sub-problems. However, as this guidance consists of a total order over different branches, it may lose some chances of finding counterexamples in some branches that initially seem not promising.

To mitigate this issue, we also propose a variant of our approach, inspired by simulated annealing [24]. It works as follows: throughout the process, we maintain a temperature that keeps decreasing slowly, which is used to control the trade-off between "exploitation" and "exploration". Initially, when the temperature is high, our algorithm tends to explore the sub-problem space by assigning a considerably high probability to accept a sub-problem that is not promising; as temperature goes down, it converges to the optimal sub-problem; due to the exploration of the search space at the initial stage, the algorithm is likely to converge to the globally optimal sub-problems. In this way, we can mitigate the side-effect of being too greedy, and strike a balance between "exploitation" and "exploration".

3 Preliminaries

In this section, we first introduce the neural network verification problem, and then introduce the state-of-the-art verification approach called *branch-and-bound* (BaB).

3.1 Neural Network Verification Problem

In this paper, we consider feed-forward neural networks, as depicted in Fig. 1.

▶ **Definition 1** (Neural networks). *A (feed-forward) neural network $N : \mathbb{R}^n \to \mathbb{R}^m$ maps an n-dim input to an m-dim output (see Fig. 1 for an example). It accomplishes the mapping by alternating between affine transformations (parametrized by weight matrix W_i and bias vector B_i) and non-linear activation functions σ; namely, the output $\boldsymbol{x}_i$ of layer i ($i \in \{1, \ldots, L\}$) is computed based on the output $\boldsymbol{x}_{i-1}$ of layer $i-1$, as follows: $\boldsymbol{x}_i = \sigma(W_i \boldsymbol{x}_{i-1} + B_i)$. Here, the computation unit that computes each dimension of $\boldsymbol{x}_i$ is called a* neuron*; in each layer i, the number of neurons is equivalent to the number of dimensions of $\boldsymbol{x}_i$. Specially, $\boldsymbol{x}_0$ is the input and $\boldsymbol{x}_L$ is the output of the neural network. There could be different choices for the non-linear activation functions, such as ReLU, sigmoid and tanh; following many existing works [25], we adopt Rectified Linear Unit (ReLU) (i.e., $ReLU(x) = \max(0, x)$) as the activation function in our neural networks.*

Specifications are logical expressions that formalize users' desired properties about neural networks. In this paper, we adopt the following notation in Def. 2 as our specification formalism, and later we show that it can be used to formalize commonly-used properties, such as *local robustness* of neural networks.

▶ **Definition 2** (Specification). *We denote by a pair (Φ, Ψ) a* specification *for neural networks, where Φ is called an input specification that predicates over the input region, and Ψ is called an output specification that predicates over the output of a neural network. Specifically, we denote the output specification Ψ as follows: $f(N(\boldsymbol{x}_0)) > 0$, where $f : \mathbb{R}^m \to \mathbb{R}$ is a function that maps an m-dim vector to a real number.*

▶ **Definition 3** (Verification problem). *A verification problem aims to answer the following question: given a neural network N and a specification (Φ, Ψ), whether $\Psi(N(\boldsymbol{x}))$ holds, for any input $\boldsymbol{x}$ that holds $\Phi(\boldsymbol{x})$. A verifier is a tool used to answer a verification problem: it either returns* true*, certifying N's satisfaction to the specification, or returns* false *with a* counterexample *$\hat{\boldsymbol{x}}$, which is an input that holds $\Phi(\hat{\boldsymbol{x}})$ but does not hold $\Psi(\hat{\boldsymbol{x}})$.*

We now explain how our notations can be used to formalize a neural network verification problem against local robustness, which is a property often considered in the domain of image classification. Local robustness requires that a neural network classifier should make consistent classification for two images $\boldsymbol{x}$ and $\boldsymbol{x}_0$, where $\boldsymbol{x}$ is produced by adding small perturbations to $\boldsymbol{x}_0$. Formally, given a reference input $\boldsymbol{x}_0$, $\Phi(\boldsymbol{x})$ requires that the input $\boldsymbol{x}$ must stay in the region $\{\boldsymbol{x} \mid \|\boldsymbol{x} - \boldsymbol{x}_0\|_\infty \leq \epsilon\}$, where $\|\cdot\|_\infty$ denotes the ℓ^∞-norm distance metric and ϵ is a small real value, and $\Psi(N(\boldsymbol{x}))$ requires that $\min_{1 \leq i \leq m, i \neq i_l} (N(\boldsymbol{x})_{i_l} - N(\boldsymbol{x})_i) > 0$, where $N(\boldsymbol{x})_i$ denotes the i-th component of the output vector $N(\boldsymbol{x})$, and i_l is the label of $\boldsymbol{x}_0$ inferred by the neural network, i.e., $i_l = \arg\max_{1 \leq i \leq m} N(\boldsymbol{x}_0)_i$.

3.2 Branch-and-Bound (BaB) – State-of-the-Art Verification Approach

There have been various approaches proposed to solve the neural network verification problem. Exact encoding [23, 47, 7] formalizes the inference process and the specification of a neural network to be an SMT or optimization problem and solves them by dedicated [23] or off-the-shelf [47, 7] solvers. These approaches are sound and complete, but due to the non-linearity of neural network inferences, they suffer from severe scalability issues and cannot handle neural networks of large sizes. To resolve this issue, approximated approaches [42, 41, 44] over-approximate the output of neural networks by linear relaxation, and they can certify the satisfaction of specification if the over-approximated output satisfies the specification. These approaches are typically efficient and sound; however, they are not complete, i.e., they

may raise false alarms with spurious counterexamples, if the over-approximated output does not satisfy the specification. As the state-of-the-art, Branch-and-Bound (BaB) [5] employs approximated verifiers and splits the problem when necessary (i.e., if a false alarm arises in a problem), because by solving sub-problems it can improve the precision of approximation and reduce the occurrence of false alarms. BaB keeps problem splitting until all of the sub-problems are certified (such that the original problem can be certified) or a real counterexample is detected with a sub-problem (such that the original problem is falsified).

In the following, we explain the necessary ingredients and detailed workflow of BaB.

Approximated verifiers

Approximated verifiers, denoted as LpVerifier, are a class of verifiers that solve a verification problem by computing an over-approximation of the output region of neural networks – if the over-approximation satisfies the specification, it implies that the original output region must also satisfy it. Typically, LpVerifier can return a tuple $\langle \hat{p}, \hat{\boldsymbol{x}} \rangle$, where $\hat{p} \in \mathbb{R}$ is called a *verifier assessment*, which is a quantity that indicates *the extent* to which the over-approximated output satisfies the specification. Formally, given an LpVerifier, it can construct a region $\hat{\Omega} \supseteq \{N(\boldsymbol{x}) \mid \Phi(\boldsymbol{x}) = \mathsf{true}\}$, which over-approximates the output region of a neural network N; then, LpVerifier computes $\hat{p}$ as follows: $\hat{p} := \min_{\boldsymbol{y} \in \hat{\Omega}} f(\boldsymbol{y})$, where f is as defined in Def. 2. If $\hat{p}$ is positive, it implies that the original output also satisfies the specification, and so it can certify the satisfaction of neural networks. Conversely, if $\hat{p}$ is negative, LpVerifier deems that the specification is violated and provides a counterexample input $\hat{\boldsymbol{x}}$ as an evidence. However, due to the over-approximation, this reported violation of LpVerifier could be a false alarm and the counterexample $\hat{\boldsymbol{x}}$ may be a spurious one, i.e., the corresponding output $N(\hat{\boldsymbol{x}})$ of $\hat{\boldsymbol{x}}$ actually satisfies the specification. In this case, just based on the result of LpVerifier, we cannot infer whether N holds or violates the specification. This situation is referred to as the *completeness issue* of approximated verifiers.

There have been various approximation strategies that can be used to implement approximated verifiers, e.g., *DeepPoly* [44] and *ReluVal* [50], and mostly, the over-approximation is accomplished by using linear constraints to bound the output range of the non-linear ReLU functions. In particular, we assume that our adopted approximated verifiers hold the *monotonicity* property, namely, for two output regions of neural networks that hold $\Omega_1 \subset \Omega_2$, if we over-approximate them by using the same approximated verifier LpVerifier, the obtained over-approximation $\hat{\Omega}_1$ and $\hat{\Omega}_2$ hold that $\hat{\Omega}_1 \subseteq \hat{\Omega}_2$. While there can be different options of over-approximation strategies, our proposed approach is orthogonal to them, so we can adopt any approximated verifiers that hold our assumption about monotonicity.

Branch-and-Bound (BaB)

BaB is the state-of-the-art neural network verification approach, and has been adopted in several advanced verification tools, such as $\alpha\beta$-Crown [51] and Marabou [52]. It is essentially a "divide-and-conquer" strategy, that divides a verification problem adaptively and applies off-the-shelf verifiers (such as approximated verifiers) to solve the sub-problems. Because approximated verifiers often achieve better precision on sub-problems, BaB can thus overcome the weaknesses of the plain application of approximated verifiers to the original problem and resolve the issues of false alarms.

Before looking into the details of BaB, we first introduce ReLU specification, which is an important notion in the algorithm of BaB.

Algorithm 1 Branch-and-Bound (`BaB`) [5].

Require: A neural network N, an input specification Φ, an output specification Ψ, an approximated verifier `LpVerifier`$(\cdot)$, and a ReLU selection heuristic $\mathsf{H}(\cdot)$.
Ensure: A *verdict* $\in \{\mathsf{true}, \mathsf{false}\}$.

```
1:  Q ← {⊤}
2:  T ← ∅
3:  function BaB(N, Φ, Ψ, Q)
4:      if Empty(Q) then
5:          return true
6:      Γ ← Pop(Q)
7:      ⟨p̂, x̂⟩ ← LpVerifier(N, Φ, Ψ, Γ)
8:      T ← T ∪ {⟨Γ, p̂⟩}
9:      if p̂ < 0 then
10:         if Valid(x̂, N, Ψ) then
11:             return false
12:         else
13:             r_k ← H(Γ)
14:             for a ∈ {r_k^+, r_k^-} do
15:                 Push(Q, Γ ∧ a)
16:     return BaB(N, Φ, Ψ, Q)
```

▶ **Definition 4** (ReLU specification). *Let N be a neural network consisting of K neurons, and $\hat{x}_i \in \mathbb{R}$ $(i \in \{1, \ldots, K\})$ be the input for the ReLU function in the i-th neuron. An* atomic proposition `AP` *w.r.t. the i-th neuron is defined as either $\hat{x}_i \geq 0$ (written as r_i^+) or $\hat{x}_i < 0$ (written as r_i^-). Then, a ReLU specification Γ is defined as the conjunction of a sequence $\mathsf{Set}(\Gamma) := \{\mathtt{AP}_1, \ldots, \mathtt{AP}_{|\Gamma|}\}$ of atomic propositions, where each $\mathtt{AP} \in \mathsf{Set}(\Gamma)$ is defined w.r.t. a distinct neuron. $|\Gamma|$ is the number of atomic propositions in $\mathsf{Set}(\Gamma)$; specially if $|\Gamma| = 0$, Γ is denoted as $\top$. Moreover, we define a* refinement relation $\prec$ *over ReLU specifications, namely, given two ReLU specifications Γ_i and Γ_j, we say Γ_j refines Γ_i (denoted as $\Gamma_i \prec \Gamma_j$) if and only if $\mathsf{Set}(\Gamma_i) \subset \mathsf{Set}(\Gamma_j)$.*

By selecting a neuron i, we can split the ReLU function into two linear functions, each identified by a predicate r_i^+ or r_i^- over the input of ReLU. Thereby, a neural network verification problem boils down to two sub-problems, for each of which `LpVerifier` does not need to perform over-approximation for the ReLU function in the i-th neuron.

The workflow of `BaB` is presented in Alg. 1. In Alg. 1, we allow `LpVerifier` to take an additional argument, namely, a ReLU specification Γ, which identifies a sub-problem by adding the constraints in $\mathsf{Set}(\Gamma)$ to constrain the inputs of a number of selected ReLU functions. It uses a FIFO queue Q to maintain the problem to be solved, which is initialized to be a set that consists of $\top$ only, identifying the original verification problem.

i) First, it applies `LpVerifier` to the original problem (Line 7): if `LpVerifier` returns a positive $\hat{p}$, or a negative $\hat{p}$ with a valid counterexample $\hat{\boldsymbol{x}}$ (Line 10), then verification can be terminated with a verdict returned accordingly;

ii) In the case it returns a negative $\hat{p}$ with a spurious counterexample (Line 12), it divides the verification problem into two sub-problems. This is achieved by first selecting a neuron (i.e., a ReLU) in the network according to a pre-defined ReLU selection heuristics H (Line 13), and then identifying two sub-problems each identified by an additional constraint on the input of the selected ReLU function (Line 15).

iii) It applies `LpVerifier` to the new sub-problems respectively, and decides whether it needs to further split the sub-problem, following the same rule in Step ii. In Alg. 1, this is implemented by recursive call of the BaB function in Line 16.

In `BaB`, the ReLU selection heuristics H involves an order over different neurons such that it can select the next ReLU based on an existing ReLU specification. There has been a rich body of literature that proposes different ReLU selection strategies, such as DeepSplit [21] and FSB [10]. However, our proposed approach in §4 is orthogonal to these strategies and so it can work with existing strategies. In this work, we follow an existing neural network verification approach [48] and we adopt the state-of-the-art ReLU selection strategy in [21].

As a result of Alg. 1, it forms a binary tree $\mathcal{T}$ (see Line 2 and Line 8) that records the history of problem splitting during the `BaB` process. In this tree, each node $\langle \Gamma, \hat{p} \rangle$ denotes a sub-problem, identified by a ReLU specification Γ and the verifier assessment $\hat{p}$ returned by `LpVerifier` that signifies the satisfaction of the sub-problem.

▶ **Lemma 5** (Soundness and completeness). *The* `BaB` *algorithm is sound and complete.*

Proof. Soundness requires that if `BaB` returns `true`, the neural network must satisfy the specification. The soundness of `BaB` relies on that: 1) `LpVerifier` is sound; 2) the ReLU specifications identified by the leaf nodes of the `BaB` tree cover all the cases about the input conditions of the split ReLUs in the neural network.

Completeness requires that if `BaB` returns `false`, the neural network must violate the specification. The completeness of `BaB` relies on the fact that the counterexamples $\hat{\boldsymbol{x}}$ reported by `BaB` are all validated and so they must be real. Moreover, in the worst case if all neurons are split, the sub-problem will be linear and so a $\hat{\boldsymbol{x}}$ must be a real one if it exists.

Finally, `BaB` is guaranteed to return either `true` or `false` within a finite time budget, because the number of neurons in a neural network is finite. ◀

4 Oliva: The Proposed Verification Approach

As a "divide-and-conquer" strategy, `BaB` can produce a huge space that consists of quantities of sub-problems; however, as shown in Alg. 1, the existing `BaB` approach explores this space in a naive "first come, first served" manner (implemented by the *first-in-first-out* queue in Alg. 1 that stores the (sub)-problems to be solved), which can be very inefficient to exhaust the sub-problem space.

Our proposed approach involves exploring the `BaB` tree in an intelligent fashion, guided by the severity of different tree nodes. In §4.1, we showcase such a severity order, defined by the probability of finding counterexamples with a sub-problem. The intuition behind this order is that, by prioritizing sub-problems that are more likely to find counterexamples, we may quickly find a counterexample and thereby immediately terminate the verification. If we cannot find it after visiting all sub-problems, we achieve certification of the problem.

4.1 Counterexample Potentiality Order

We introduce an order over different sub-problems based on their probability of containing counterexamples. Given a node in `BaB` tree that identifies a sub-problem, we can infer this probability using the following two attributes:

Node depth. In a `BaB` tree, the depth of a node signifies the levels of problem splitting, and for more finely-split sub-problems, `LpVerifier` introduces less over-approximation. Because of this, if a node Γ with a greater depth is still deemed by `LpVerifier` as violating the specification (i.e., the verifier assessment $\hat{p}$ is negative), it is more likely that Γ indeed contains real counterexamples.

Verifier assessment. Given a node in BaB tree that identifies a sub-problem, its $\hat{p}$ returned by `LpVerifier` can be considered as a quantitative indicator of *how far* the sub-problem is from being violated. Due to our assumption about the monotonicity of `LpVerifier` in performing over-approximation, given a fixed `LpVerifier`, $\hat{p}$ is correlated to the original output region of the neural network. Therefore, in the case $\hat{p}$ is negative, the greater $|\hat{p}|$ is, there is a higher possibility that the original output region is closer to violation of the specification, and so it is more likely that the sub-problem contains a real counterexample.

Based on the above two node attributes, we define the suspiciousness of a BaB tree node, and then define a Counterexample POtentiality (CePO) order over different nodes.

▶ **Definition 6** (Suspiciousness of sub-problems). *Let Γ be a node of the BaB tree that has a verifier assessment $\hat{p}$ (with a counterexample $\hat{\boldsymbol{x}}$ if $\hat{p} < 0$). The suspiciousness $\mathsf{susp}(\Gamma) \in [0,1] \cup \{+\infty, -\infty\}$ of the node Γ maps Γ to a real number as follows:*

$$\mathsf{susp}(\Gamma) := \begin{cases} -\infty & \text{if } \hat{p} > 0 \\ +\infty & \text{if } \hat{p} < 0 \text{ and } \mathtt{valid}(\hat{\boldsymbol{x}}) \\ \lambda \frac{\mathtt{depth}(\Gamma)}{K} + (1-\lambda)\frac{\hat{p}}{\hat{p}_{\min}} & \text{otherwise} \end{cases}$$

where $\lambda \in [0,1]$ is a parameter that controls the weights of the two attributes, and K is the total number of neurons (i.e., ReLUs) in the network.

Intuitively, the suspiciousness of a node encompasses a heuristic that estimates the probability of the relevant sub-problem violating the specification. It is particularly meaningful in the case when $\hat{p} < 0$ and the counterexample is spurious, and set to be $+\infty$ if the sub-problem is provably violated, and $-\infty$ if the sub-problem is certified. By their suspiciousness, we define the CePO order over different nodes as follows:

▶ **Definition 7** (Counterexample potentiality (CePO) order). *Let Γ_1 and Γ_2 be two nodes in a BaB tree, and $\hat{p}_1$ and $\hat{p}_2$ be verifier assessments for Γ_1 and Γ_2, respectively. We define a CePO order $\sqsubset$ between the two nodes as follows:*

$$\Gamma_1 \sqsubset \Gamma_2 \quad \textit{iff} \quad \mathsf{susp}(\Gamma_1) < \mathsf{susp}(\Gamma_2)$$

The CePO order allows us to sort the nodes in BaB tree,[1] and so it can serve as a guidance to our verification approach in §4.2 and §4.3.

4.2 Oliva$^{\mathsf{GR}}$: Greedy Exploration of BaB Tree

The Oliva$^{\mathsf{GR}}$ algorithm, presented in Alg. 2, implements a greedy strategy that explores the space of BaB tree nodes guided by the CePO order. Compared to the classic BaB, it always selects the most suspicious node to expand, such that it can maximize the probability of finding counterexamples and thereby conclude the verification problem efficiently.

The algorithm begins with applying the approximated verifier `LpVerifier` to the original verification problem, identified by the ReLU specification ε (Line 2), which returns a tuple $\langle \hat{p}, \hat{\boldsymbol{x}} \rangle$ containing a verifier assessment $\hat{p}$, possibly followed by a counterexample $\hat{\boldsymbol{x}}$. The suspiciousness $\mathsf{susp}(\varepsilon)$ of the root node is then computed based on this assessment. At this point, if $\hat{p} > 0$ or if $\hat{p} < 0$ with a valid counterexample $\hat{\boldsymbol{x}}$, the algorithm can be immediately

[1] In the case if two nodes have the same suspiciousness, we simply impose a random order; in the case both are $-\infty$ or $+\infty$, the comparison is meaningless and so we do not need to sort them.

Algorithm 2 Oliva$^{\mathsf{GR}}$: The proposed greedy algorithm.

Require: A neural network N, input and output specification Φ and Ψ, an approximated verifier $\texttt{LpVerifier}(\cdot)$, a ReLU selection heuristic $\mathsf{H}(\cdot)$, and a hyperparameter λ.
Ensure: A *verdict* $\in \{\mathsf{true}, \mathsf{false}, \texttt{timeout}\}$

1: $\mathcal{T} \leftarrow \{\varepsilon\}$
2: $\langle \hat{p}, \hat{\boldsymbol{x}} \rangle \leftarrow \texttt{LpVerifier}(N, \Phi, \Psi, \varepsilon)$
3: $\mathsf{R}(\varepsilon) \leftarrow \mathsf{susp}(\varepsilon)$ ▷ a metric for child selection
4: **if** $\hat{p} < 0$ and $\texttt{not valid}(\hat{\boldsymbol{x}})$ **then**
5: **while** not reach termination condition **do**
6: GreedyBaB$(\varepsilon, N, \Phi, \Psi)$
7: **return** $\begin{cases} \mathsf{true} & \text{if } \mathsf{R}(\varepsilon) = -\infty \\ \mathsf{false} & \text{if } \mathsf{R}(\varepsilon) = +\infty \\ \texttt{timeout} & \text{otherwise} \end{cases}$
8: **else**
9: **return** $\begin{cases} \mathsf{true} & \text{if } \hat{p} > 0 \\ \mathsf{false} & \text{if } \hat{p} < 0 \text{ and } \texttt{valid}(\hat{\boldsymbol{x}}) \end{cases}$

10: **function** GreedyBaB(Γ, N, Φ, Ψ)
11: $r_k \leftarrow \mathsf{H}(\Gamma)$
12: **if** $\Gamma \cdot r_k^+ \in \mathcal{T}$ **then** ▷ check existence of children
13: $\Gamma^* \leftarrow \Gamma \cdot a^*$ s.t. $a^* \leftarrow \underset{a \in \{r_k^+, r_k^-\}}{\arg\max}\, \mathsf{R}(\Gamma \cdot a)$
14: GreedyBaB$(\Gamma^*, N, \Phi, \Psi)$
15: **else**
16: **for** $a \in \{r_k^+, r_k^-\}$ **do** ▷ $\mathcal{T}$ expansion via BaB.
17: $\langle \hat{p}, \hat{\boldsymbol{x}} \rangle \leftarrow \texttt{LpVerifier}(N, \Phi, \Psi, \Gamma \cdot a)$ ▷ apply `LpVerifier` with $\Gamma \wedge a$
18: $\mathsf{R}(\Gamma \cdot a) \leftarrow \mathsf{susp}(\Gamma \cdot a)$ ▷ compute R
19: $\mathcal{T} \leftarrow \mathcal{T} \cup \{\Gamma \cdot a\}$ ▷ add to the tree of $\Gamma \wedge a$
20: $\mathsf{R}(\Gamma) \leftarrow \underset{a \in \{r_k^+, r_k^-\}}{\arg\max}\, \mathsf{R}(\Gamma \cdot a)$

terminated with a conclusive verdict (Line 9). Otherwise, the algorithm enters its main loop where it iteratively calls GreedyBaB to split and verify the sub-problems until a termination condition is reached (Line 5-7). We elaborate on the termination condition later.

The GreedyBaB function implements the main process of tree exploration and problem splitting. By each of its execution, it selects the maximal sub-problem in terms of the CePO order in the tree and applies `LpVerifier` to its subsequent sub-problems with the aim of finding counterexamples. The function begins with selecting a ReLU r_k as a successor of the current node (Line 11) using the pre-defined ReLU selection heuristics H (see §3.2), and then it recursively calls GreedyBaB until it reaches the greatest node Γ, in terms of the CePO order, whose children are not expanded yet (Line 12-14). After reaching such a node Γ, it expands the children of Γ, by applying `LpVerifier` respectively to the two sub-problems identified by the children of Γ (Line 16-17). It also computes and records the suspiciousness of the newly expanded children (Line 18), and updates $\mathcal{T}$ that keeps track of the tree (Line 19). Lastly, the greater suspiciousness over the children are propagated backwards to ancestor nodes until the root (Line 20), serving as a reference for future node selection.

Termination condition

The main loop in Line 5 of Alg. 2 can be terminated on the satisfaction of any of the following three conditions:

- $\mathsf{R}(\varepsilon) = -\infty$: This implies that all leaf nodes have been verified successfully (i.e., $\hat{p} > 0$ for all leaves, otherwise by Line 20 $\mathsf{R}(\varepsilon)$ cannot be $-\infty$), allowing the algorithm to return a true verdict that certifies the specification satisfaction;
- $\mathsf{R}(\varepsilon) = +\infty$: This implies that a valid counterexample has been found in some leaf node, and so by Line 20 its suspiciousness of $+\infty$ can be back-propagated to the root node; in this case, the algorithm can be terminated with a false verdict;
- `timeout`: This occurs when the algorithm exceeds its allocated time budget without reaching either of the above conclusive verdicts.

The three termination conditions correspond to the three cases of return in Line 7.

4.3 Oliva$^{\mathsf{SA}}$: Simulated-Annealing-Style Exploration of BaB Tree

The greedy strategy in §4.2 exploits the CePO order in the exploration of `BaB` tree, so it can efficiently move towards the sub-problems that are more likely to find counterexamples. However, as it is a greedy strategy, in the case if the CePO order is not sufficiently precise, the verification process can be easily trapped into a local optimum, and miss the chances of finding counterexamples in other branches than the one suggested by the CePO order.

To bridge this gap, we propose an approach that adapts the classic framework of *simulated annealing*, which not only follows the CePO order, but also takes other branches into account. As a consequence, during the verification, it strikes a balance between "exploitation" of the suspicious branches and "exploration" of the less suspicious branches.

"Hill climbing" vs. "Simulated annealing"

Hill climbing is a stochastic optimization technique that aims to find the optimum of a black-box function. Due to the black-box nature of the objective function, it relies on sampling in the search space and selects only the samples that optimize the objective function as the direction to move. Therefore, this is a greedy strategy, similarly to our proposed approach Oliva$^{\mathsf{GR}}$ in §4.2. While Oliva$^{\mathsf{GR}}$ does not need to sample the search space, it can select the sub-problem that is more promising, and thereby move towards the direction that is more likely to achieve the objective. As a consequence, they both suffer from the issue of "local optima".

In the field of stochastic optimization, *simulated annealing* [24] is an effective approach to mitigate the issue of "local optima" in hill climbing. It is inspired by the process of *annealing* in metallurgy, in which the temperature is initially high but slowly decreases such that the physical properties of metals can be stabilized. In simulated annealing, there is also a temperature that slowly decreases throughout the process: when the temperature is initially high, it tends to explore the search space and assigns a considerably high probability to accept a sample that is not the optimal; as temperature goes down, it converges to the optimum that is likely to be the global one, thanks to the exploration of the search space at the initial stage.

Our proposed approach Oliva$^{\mathsf{SA}}$ adapts simulated annealing to our problem setting, and the core idea involves that, at the initial stage, we allow more chances of exploring the branches that are less promising. Then, after we have comprehensively explored the search space and obtained the information about the suspiciousness of different branches, we tend to exploit the branches that are more suspicious, namely, that are more likely to contain counterexamples.

Algorithm 3 Oliva$^{\mathsf{SA}}$: The proposed simulated annealing algorithm.

Require: A neural network N, input and output specification Φ and Ψ, an approximated verifier $\texttt{LpVerifier}(\cdot)$, a ReLU selection heuristic $\mathsf{H}(\cdot)$, and a hyperparameter λ.
Ensure: A *verdict* $\in \{\mathsf{true}, \mathsf{false}, \texttt{timeout}\}$

1: $\mathcal{T} \leftarrow \{\varepsilon\}$
2: $\langle \hat{p}, \hat{\boldsymbol{x}} \rangle \leftarrow \texttt{LpVerifier}(N, \Phi, \Psi, \varepsilon)$
3: $\mathsf{R}(\varepsilon) \leftarrow \mathsf{susp}(\varepsilon)$ ▷ a metric for child selection
4: $T \leftarrow T_{max}$ ▷ temperature
5: **if** $\hat{p} < 0$ and $\texttt{not valid}(\hat{\boldsymbol{x}})$ **then**
6: **while** not reach termination condition **do**
7: $T \leftarrow \alpha \cdot T$ ▷ T is decreased by $(1-\alpha)T$ in each iteration
8: $\text{SABaB}(\varepsilon, N, \Phi, \Psi)$
9: **return** $\begin{cases} \mathsf{true} & \text{if } \mathsf{R}(\varepsilon) = -\infty \\ \mathsf{false} & \text{if } \mathsf{R}(\varepsilon) = +\infty \\ \texttt{timeout} & \text{otherwise} \end{cases}$
10: **else**
11: **return** $\begin{cases} \mathsf{true} & \text{if } \hat{p} > 0 \\ \mathsf{false} & \text{if } \hat{p} < 0 \text{ and } \texttt{valid}(\hat{\boldsymbol{x}}) \end{cases}$

12: **function** $\text{SABaB}(\Gamma, N, \Phi, \Psi)$
13: $r_k \leftarrow \mathsf{H}(\Gamma)$
14: **if** $\Gamma \cdot r_k^+ \in \mathcal{T}$ **then**
15: $\Delta p \leftarrow \exp\left(\frac{\min \mathsf{R}(\Gamma \cdot a) - \max \mathsf{R}(\Gamma \cdot a)}{T}\right)$ s.t. $a \in \{r_k^+, r_k^-\}$
16: $\Gamma^* \leftarrow \Gamma \cdot a^*$ s.t. $a^* \leftarrow \begin{cases} \text{randomly choose } r_k^+ \text{ or } r_k^- & \text{if } \texttt{rand}(0,1) < \Delta p \\ \underset{a \in \{r_k^+, r_k^-\}}{\arg\max}\, \mathsf{R}(\Gamma \cdot a) & \text{otherwise} \end{cases}$
17: $\text{SABaB}(\Gamma^*, N, \Phi, \Psi)$
18: **else**
19: **for** $a \in \{r_k^+, r_k^-\}$ **do** ▷ $\mathcal{T}$ expansion via BaB.
20: $\langle \hat{p}, \hat{\boldsymbol{x}} \rangle \leftarrow \texttt{LpVerifier}(N, \Phi, \Psi, \Gamma \cdot a)$ ▷ apply `LpVerifier` with $\Gamma \wedge a$
21: $\mathsf{R}(\Gamma \cdot a) \leftarrow \mathsf{susp}(\Gamma \cdot a)$ ▷ compute R
22: $\mathcal{T} \leftarrow \mathcal{T} \cup \{\Gamma \cdot a\}$ ▷ add to the tree of $\Gamma \wedge a$
23: $\mathsf{R}(\Gamma) \leftarrow \underset{a \in \{r_k^+, r_k^-\}}{\arg\max}\, \mathsf{R}(\Gamma \cdot a)$

Algorithm details

The Oliva$^{\mathsf{SA}}$ algorithm is presented in Alg. 3. It also starts with checking the original verification problem (Line 2), and enters the loop of tree exploration if the original problem cannot be solved by `LpVerifier` (Line 5). However, compared to Oliva$^{\mathsf{SA}}$, it has a notable difference about the adoption of a temperature T (Line 4) which is a global variable that keeps decreasing in each loop of the algorithm (Line 7).

In the function SABaB, the temperature T is used when selecting the nodes to proceed. In the case if the children of the current node Γ have been expanded (Line 14), unlike Oliva$^{\mathsf{GR}}$ that always prefers the most suspicious child, Oliva$^{\mathsf{SA}}$ selects a child according to the policy adapted from the original simulated annealing:

- It first computes an acceptance probability Δp (Line 15), by which it determines whether the selection of a child that is less promising is acceptable. This probability Δp is decided by both the difference of "energy" (defined by the suspiciousness difference between two children) and the temperature T;
- In the original simulated annealing, a sample that is more promising can be accepted in any case, and a sample that is less promising can be accepted with the probability Δp. In our context, we adapt this policy as follows (Line 16):
  - If a random value in $[0, 1]$ is less than Δp, we randomly select a child from the two children of Γ with the same probability, despite the CePO order over the two children;
  - Otherwise, we select the more suspicious child following the CePO order.

The selected child will be used as the argument for the recursive call of SABaB, in order to proceed towards a node whose children are not expanded. After achieving such a node Γ, it expands the children of Γ (Lines 19-22) by applying `LpVerifier` to each child of Γ, recording their suspiciousness (Line 21) and updating the `BaB` tree (Line 22). Lastly, it back-propagates the greater suspiciousness over the children until the root (Line 23), similarly to Oliva$^{\mathsf{SA}}$.

This gradual transition from exploration to exploitation helps Oliva$^{\mathsf{SA}}$ to avoid premature convergence while ensuring an eventual focus on the promising branches. The termination conditions remain the same as in Oliva$^{\mathsf{GR}}$, checking for full verification ($\mathsf{R}(\varepsilon) = -\infty$), counterexample discovery ($\mathsf{R}(\varepsilon) = +\infty$), or timeout. However, by pursuing a higher coverage of the sub-problem space before being greedy, Oliva$^{\mathsf{SA}}$ increases the possibility of discovering counterexamples that might be missed by the purely greedy strategy of Oliva$^{\mathsf{GR}}$.

▶ **Theorem 8.** *Both* Oliva$^{\mathsf{GR}}$ *and* Oliva$^{\mathsf{SA}}$ *are sound and complete.*

Proof. The proofs of soundness and completeness of our approaches are similar to that of `BaB` in Lemma 5, so we skip the details. Intuitively, our approaches only introduce an order of visiting different nodes in `BaB` tree, but hold all the conditions that are necessary for the soundness and completeness of `BaB`. ◀

5 Experimental Evaluation

5.1 Experiment Settings

Baselines and Metrics

We compare the two versions of Oliva, namely, Oliva$^{\mathsf{GR}}$ and Oliva$^{\mathsf{SA}}$, with three state-of-the-art neural network verification tools $\alpha\beta$-Crown [59, 51], NeuralSAT [11], and `BaB`-baseline, as presented in §3.2. The settings of baseline approaches are detailed below:

- `BaB`-baseline is implemented based on the ERAN framework [41, 44], which employs LP-based triangle relaxation for bounding the output of ReLU functions, and utilizes DeepSplit [21] as the ReLU selection heuristics for selecting ReLU function to split, i.e., H in Alg. 1, Alg. 2, and Alg. 3.
- $\alpha\beta$-Crown [59, 51] is applied with its default branch-and-bound settings. Moreover, a balanced strategy kFSB from FSB [10] is selected as the ReLU selection heuristics for selecting the ReLU function to split.
- NeuralSAT [11] is applied with its default DPLL(T) framework and we select the stabilized optimization as the neuron stability heuristics for ReLU activation pattern search. Additionally, it is registered with a random attack [8, 56] to reject the easily detected violation instances at the early stage of verification processes.

Table 1 Benchmark details for the evaluation of verification.

Model	Architecture	Dataset	#Activations	# Instances	#Images
$\texttt{MNIST}_{\texttt{L2}}$	2 × 256 linear	MNIST	512	100	70
$\texttt{MNIST}_{\texttt{L4}}$	4 × 256 linear	MNIST	1024	78	52
$\texttt{OVAL21}_{\texttt{BASE}}$	2 Conv, 2 linear	CIFAR-10	3172	173	53
$\texttt{OVAL21}_{\texttt{WIDE}}$	2 Conv, 2 linear	CIFAR-10	6244	196	40
$\texttt{OVAL21}_{\texttt{DEEP}}$	4sub Conv, 2 linear	CIFAR-10	6756	143	53

For our approaches, the hyperparameters are set as follows: $\lambda = 0.5$ (see Def. 6), $T_{max} = 1$ (see Line 4 of Alg. 3) and $\alpha = 0.99$ (see Line 7 of Alg. 3); in RQ3, we study the impact of these hyperparameters on the performance of our approaches.

We apply each of the five approaches (including two proposed approaches and three baseline approaches) to each of the verification problems, and we adopt 1000 secs as our time budget, following the VNN-COMP competition [30]. If an approach manages to solve the problem (either verifies the problem or reports a real counterexample), we deem this run as a *solved* verification process, and record the time cost for reaching the verification conclusion. In particular, our $\mathsf{Oliva}^{\mathsf{SA}}$ is a stochastic approach, namely, its performance is subject to randomness. To compare its performance with other approaches, in Table 2, Table 3 and Fig. 5, we only show the performance of one random run; in Fig. 6, we particularly study the influence of randomness to $\mathsf{Oliva}^{\mathsf{SA}}$, by repeating each verification process for 5 times.

Our evaluation metrics include the number of instances solved by an approach and the time costs for each successful verification process. For comparison of two different approaches, we also compute the speedup rate, which is the ratio of the time costs of the two approaches.

Moreover, in order to understand whether our counterexample potentiality order works, we compare the performances of our tools with the baseline approach, respectively for the problems that are finally certified and the problems that are finally falsified. We further study the influence of the hyperparameters (including λ in Def. 6 and α that decides the change rate of temperature in Line 7 of Alg. 3) to the performances of our approaches. In the implementation of Oliva, we adopt the same approximated verifiers and ReLU selection heuristics as the BaB-baseline. The code and experimental data of Oliva are available online[2].

Datasets and Neural Networks

Our experimental evaluation uses two well-known datasets: MNIST, featuring images of handwritten digits for classification, and CIFAR-10, featuring images of various real-world objects like airplanes, cars, and animals, with networks tasked to identify each class. These datasets are standard benchmarks that have been widely used in the neural network verification community and adopted in the VNN-COMP [30], an annual competition in the community for comparing the performances of different verification tools. We evaluate two networks trained on MNIST that have fully-connected layers only, and three neural networks trained on CIFAR-10 that have both convolutional layer and fully-connected layers, with different network architectures, following common evaluation utilized in ERAN [32, 51] and OVAL21 [4, 5] benchmarks. More details of these benchmarks are presented in Table 1.

[2] https://github.com/DeepLearningVerification/Oliva

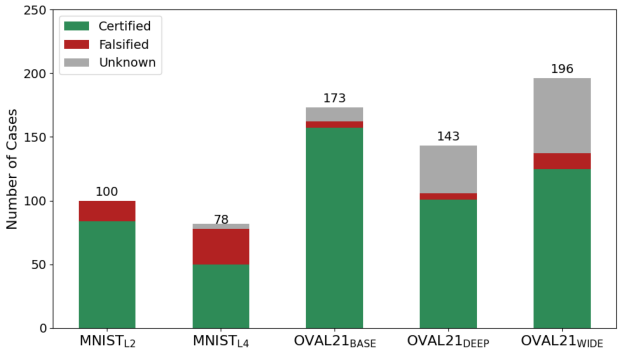


Figure 3 The distribution of Certified/Falsified/Unknown cases generated by `BaB`-baseline.

Specifications

In our experiments, all our neural network models are used for image classification tasks, and all the specifications adopted are about local robustness of the neural network models. Each input specification defines an ϵ as the threshold for perturbation; and each output specification requires the expected label to be matched by the original input of the model. In total, we collected 690 problem instances. Table 1 under the column "# Instances" displays the number of instances in each model, and "# Images" shows the number of different images covered by the different instances. Fig. 3 demonstrates the distribution of verification verdicts across the five models in our benchmark set, according to our experimental results. The green portion denotes tasks eventually verified ("Certified"), the red portion denotes tasks that were shown to violate the property ("Falsified"), and the gray portion corresponds to tasks that remained inconclusive ("Unknown").

In order to present a meaningful comparison, we need to avoid the verification problems that are too simple, for which the problem can be solved within a small number of problem splitting and so there is no need to expand the `BaB` tree too much. To that end, we perform a selection of parameters (i.e., ϵ in Def. 2) of input specifications, from a range of 0/255 to 16/255 under the L_∞ norm. Our approach involves a binary search-like algorithm to determine the proper perturbation values for each image, as follows:

1. Initially, we set 0/255 as the lower bound l and 16/255 as the upper bound u, and calculates the midpoint by taking $m = \frac{l+u}{2}$;
2. We apply `BaB`-baseline to the verification problem with m as the ϵ, and check the number of nodes in the `BaB` tree. If the tree size is greater than 1, we accept it as a candidate parameter and move to the next image; otherwise, based on the results of verification, we proceed to update m as like Step 1 and repeat Step 2 as follows:
   - If the specification is violated, it implies that ϵ is too large and so we set u to be m, and update m accordingly;
   - If the specification is satisfied, it implies that ϵ is too small and so we set l to be m, and update m accordingly.

This process continues until a pre-defined budget is exhausted.

Fig. 4 shows the distribution of node counts across all models for the verification instances. The distribution confirms that all instances involve multiple tree nodes, ensuring meaningful sub-problem selection. Notably, more than half of the tree sizes fall within the range of 100-1000 nodes, highlighting the importance of effective sub-problem selection. The complexity of most instances necessitates careful choice in verification instances.

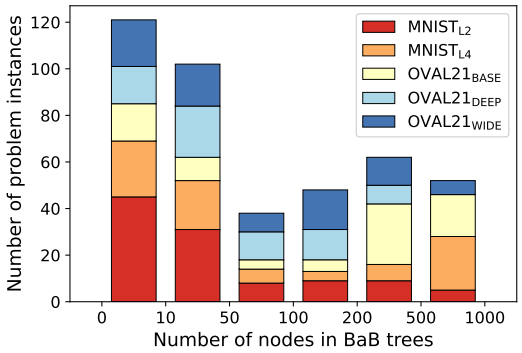


Figure 4 The distribution of the tree sizes generated by BaB-baseline.

Software and Hardware Setup

All experiments were conducted on an AWS EC2 instance running a Linux system with 16GB of memory and an 8-core Xeon E5 2.90 GHz CPU. All tools are developed in Python 3.9, and we used the GUROBI solver 9.1.2 [19] for the LP-based optimization. For each verification instance, we set a time budget as 1000 seconds which is consistent with VNN-COMP [30].

5.2 Evaluation

RQ1: Is Oliva more efficient than existing approaches?

In Table 2, we compare the performance of our approach with the state-of-the-art baseline approaches, BaB-baseline, $\alpha\beta$-Crown and NeuralSAT, across all our verification problems.

We observe that, Oliva shows evident improvement for benchmarks including OVAL21$_{BASE}$, OVAL21$_{DEEP}$ and OVAL21$_{WIDE}$. On the OVAL21$_{BASE}$, Oliva^{SA} solves 159 instances in an average of 155.29 seconds, far surpassing BaB-baseline (42 instances in 770.7 seconds) and $\alpha\beta$-Crown (58 instances in 641.7 seconds). For OVAL21$_{DEEP}$, Oliva^{GR} solves 92 instances, compared to 33 by BaB-baseline and 55 by $\alpha\beta$-Crown, while using less time (250.72 seconds compared to 552.24 seconds). The difference is even more pronounced for OVAL21$_{WIDE}$, where Oliva^{GR} solves 131 instances in an average of 240.16 seconds, greatly exceeding both BaB-baseline (40 instances in 733.73 seconds) and $\alpha\beta$-Crown (63 instances in 557.51 seconds).

For MNIST$_{L2}$ benchmarks that have simpler architectures compared to OVAL21, BaB-baseline performs well with 96 instances solved, while Oliva^{GR} achieves 99 solved instances with a faster average time (57.76 seconds). As the size of the network increases to MNIST$_{L4}$, $\alpha\beta$-Crown declines to 44 instances, while Oliva^{GR} achieves 67 solved instances with an average time of 146.31 seconds. These results highlight the effectiveness and efficiency of Oliva, compared to the baseline approaches.

Table 3 presents a pairwise comparison, in which each cell indicates the number of problem instances solved by the approach listed in the row, but not solved by the approach listed in the column. Namely, we can perform pairwise comparison between two approaches by comparing the values in a pair of cells symmetric to the diagonal of Table 3. Compared to baseline approaches, our approaches demonstrate superior performances, as indicated by the green area that shows the number of additional problems solved by our approaches by not by baseline approaches; in comparison, the numbers in the red area, that includes the number of problems solved by baseline approaches, but not solved by our approaches, are much less. By comparing the performances between Oliva^{GR} and Oliva^{SA}, we find that Oliva^{GR} slightly

Table 2 RQ1 – Overall comparison of different verification approaches, in terms of the number of solved problem instances and the time costs (in secs).

Model	BaB-baseline		$\alpha\beta$-Crown		Neuralsat		Oliva^{GR}		Oliva^{SA}	
	Solved	Time	Solved	Time	Solved	Time	Solved	Time	Solved	Time
$\texttt{MNIST}_{\texttt{L2}}$	96	126.41	87	51.32	99	32.37	95	96.79	99	57.76
$\texttt{MNIST}_{\texttt{L4}}$	65	194.74	44	428.03	54	392.04	67	146.31	53	142.72
$\texttt{OVAL21}_{\texttt{BASE}}$	42	770.7	58	641.7	70	621.21	154	184.96	159	155.29
$\texttt{OVAL21}_{\texttt{DEEP}}$	33	694.74	55	552.24	55	539.59	92	250.72	87	261.68
$\texttt{OVAL21}_{\texttt{WIDE}}$	40	733.73	63	557.51	65	533.01	131	240.16	112	288.0

Table 3 RQ1 – Pairwise comparison on the number of additional solved problem instances from all verification tasks. The number in each cell implies the number of problem instance solved by the approach of the row, but not solved by the approach of the column.

	BaB-Baseline	$\alpha\beta$-Crown	NeuralSAT	Oliva^{GR}	Oliva^{SA}
BaB-Baseline	0	80	59	8	13
$\alpha\beta$-Crown	111	0	11	23	39
NeuralSAT	126	47	0	30	46
Oliva^{GR}	271	255	226	0	40
Oliva^{SA}	247	242	213	11	0

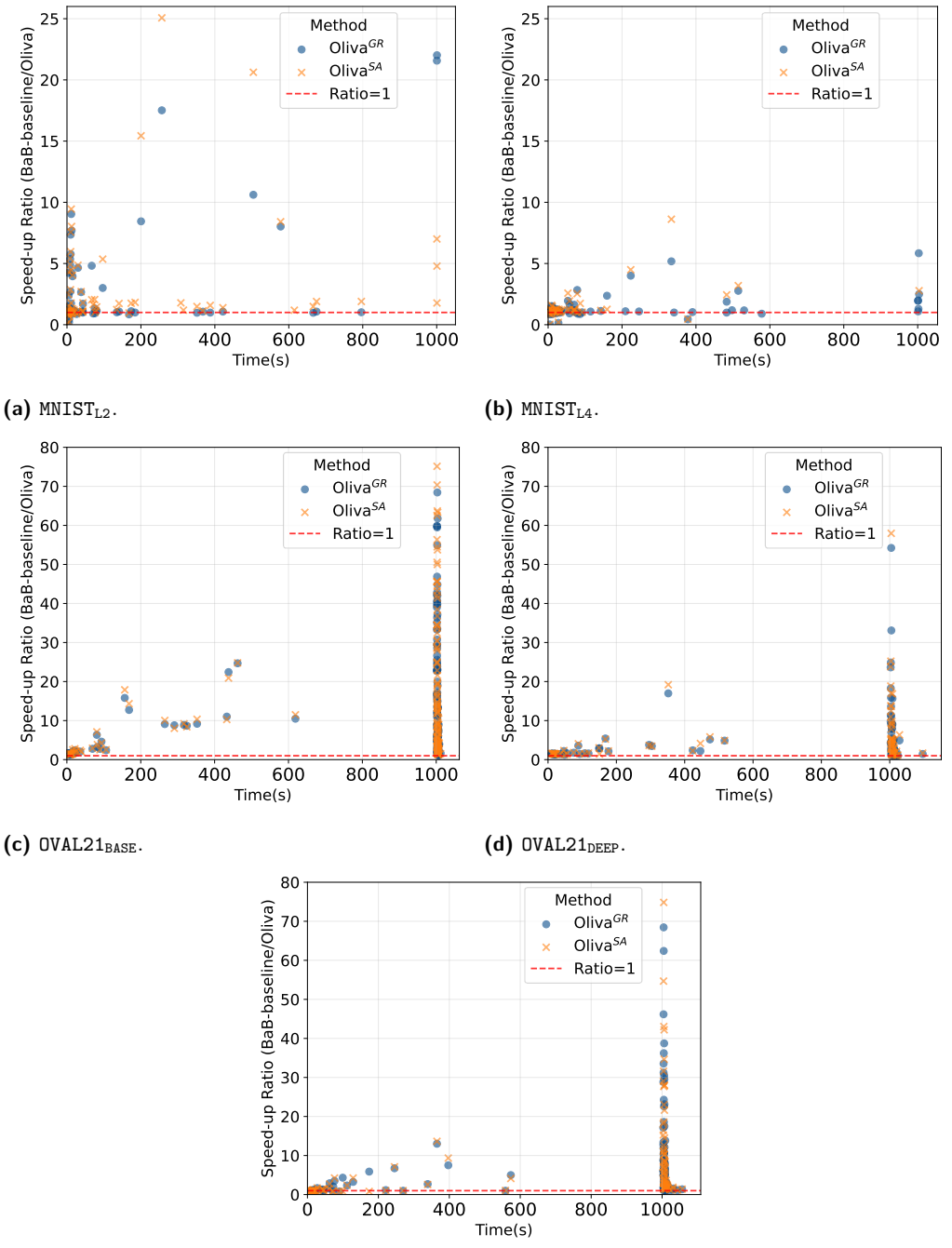
outperforms Oliva^{SA}, in that it solves 40 additional problems than Oliva^{SA}, although there are 11 problems Oliva^{SA} solves but Oliva^{GR} does not do. That said, it also demonstrates the complementary strengths between different approaches, namely, there is no one global optimal approach that can solve all the problems, and so it is worthwhile to try different approaches when one approach does not work.

In Fig. 5, we take a further comparison between Oliva and BaB-baseline, and draw a scatter plot to show the individual instances for which Oliva outperform BaB-baseline. The x-axis represents the time taken by the BaB-baseline method in seconds, while the y-axis shows the speedup ratio of Oliva over BaB-baseline. This ratio is calculated as $\tau_{\text{BaB}}(i)/\tau_{\text{Oliva}}(i)$, where $\tau_{\text{BaB}}(i)$ and $\tau_{\text{Oliva}}(i)$ denote the time taken by BaB-baseline and Oliva on the instance i, respectively. The blue dots and orange crosses represent all of the individual problem instances. The red line is the threshold at $1\times$ speedup, above which the ratio distinguishes that Oliva outperforms than BaB-baseline.

Across all the models, it can be seen that a significant number of blue dots (Oliva^{GR}) and orange crosses (Oliva^{SA}) appear above the red line, indicating that Oliva outperforms the BaB-baseline. Notably, around the 1000-second mark on the x-axis, there are multiple instances where the speedup ratio is up to $80\times$, meaning that Oliva can solve these problems more than 80 times faster than the BaB-baseline. This trend is particularly evident in OVAL21 models ($\texttt{OVAL21}_{\texttt{BASE}}$, $\texttt{OVAL21}_{\texttt{DEEP}}$, and $\texttt{OVAL21}_{\texttt{WIDE}}$) that have relatively complex architectures.

In Table 4, we summarize the statistical information of speedup ratios of our approaches over BaB-baseline. Overall, the two variants of Oliva achieve consistent performance improvements, with median speedups exceeding 2 times and mean gain values around 7 times.

Similar to simulated annealing, our tool Oliva^{SA} is a stochastic approach that is subject to randomness. To analyze the impact of randomness on the performance of Oliva^{SA}, we conduct five independent attempts with all the verification problems and compare them with the deterministic Oliva^{GR}. Since Oliva^{SA} incorporates a simulated annealing approach that makes probabilistic decisions during sub-problem exploration, different runs may explore the verification space in different orders, potentially leading to variations in performance.



(a) $\texttt{MNIST}_{\texttt{L2}}$. **(b)** $\texttt{MNIST}_{\texttt{L4}}$.

(c) $\texttt{OVAL21}_{\texttt{BASE}}$. **(d)** $\texttt{OVAL21}_{\texttt{DEEP}}$.

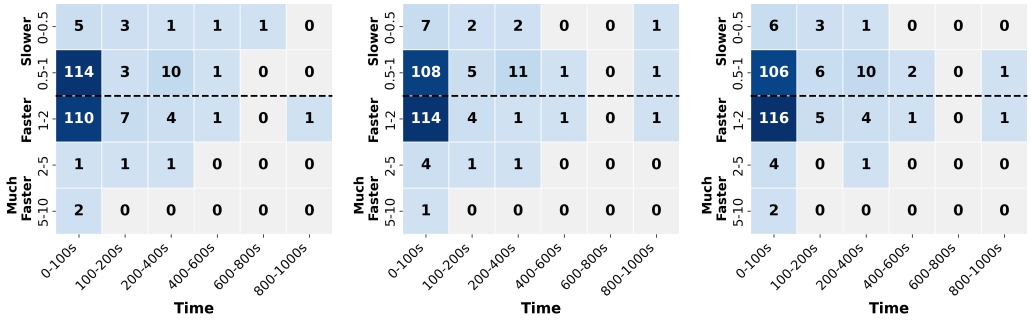
(e) $\texttt{OVAL21}_{\texttt{WIDE}}$.

Figure 5 RQ1 – Comparison of time cost and speedup ratio of the two variants $\mathsf{Oliva}^{\mathsf{GR}}$ and $\mathsf{Oliva}^{\mathsf{SA}}$ of our proposed approach Oliva over BaB-baseline.

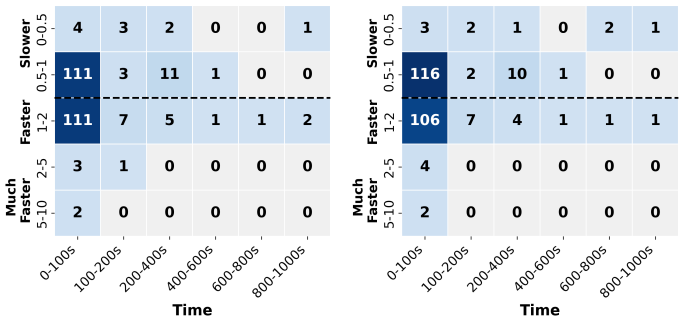
The results are presented in Fig. 6, in which we only present the problems that are either solved by $\mathsf{Oliva}^{\mathsf{GR}}$ or $\mathsf{Oliva}^{\mathsf{SA}}$. Overall, the performance comparisons between $\mathsf{Oliva}^{\mathsf{GR}}$ and $\mathsf{Oliva}^{\mathsf{SA}}$ are consistent across different attempts, and are concentrated in the 0.5-2× speedup bracket. This result suggests that the randomness in the simulated annealing approach does

Table 4 RQ1 – Statistics of speedup ratios of two variants of Oliva over BaB-baseline.

Model	Tool name	Min	Max	Median	Mean
Overall	Oliva^{GR}	0.02	80.97	2.21	7.27
	Oliva^{SA}	0.03	75.13	2.18	7.57
$MNIST_{L2}$	Oliva^{GR}	0.11	22.02	1.04	2.49
	Oliva^{SA}	0.11	25.07	1.12	2.50
$MNIST_{L4}$	Oliva^{GR}	0.02	5.84	1.10	1.38
	Oliva^{SA}	0.03	8.62	1.17	1.47
$OVAL21_{BASE}$	Oliva^{GR}	1.07	80.97	6.93	13.73
	Oliva^{SA}	1.28	75.13	7.02	13.96
$OVAL21_{DEEP}$	Oliva^{GR}	1.01	54.23	2.80	5.37
	Oliva^{SA}	1.03	57.96	2.34	5.31
$OVAL21_{WIDE}$	Oliva^{GR}	0.81	68.44	2.95	7.48
	Oliva^{SA}	0.78	74.83	2.55	7.60



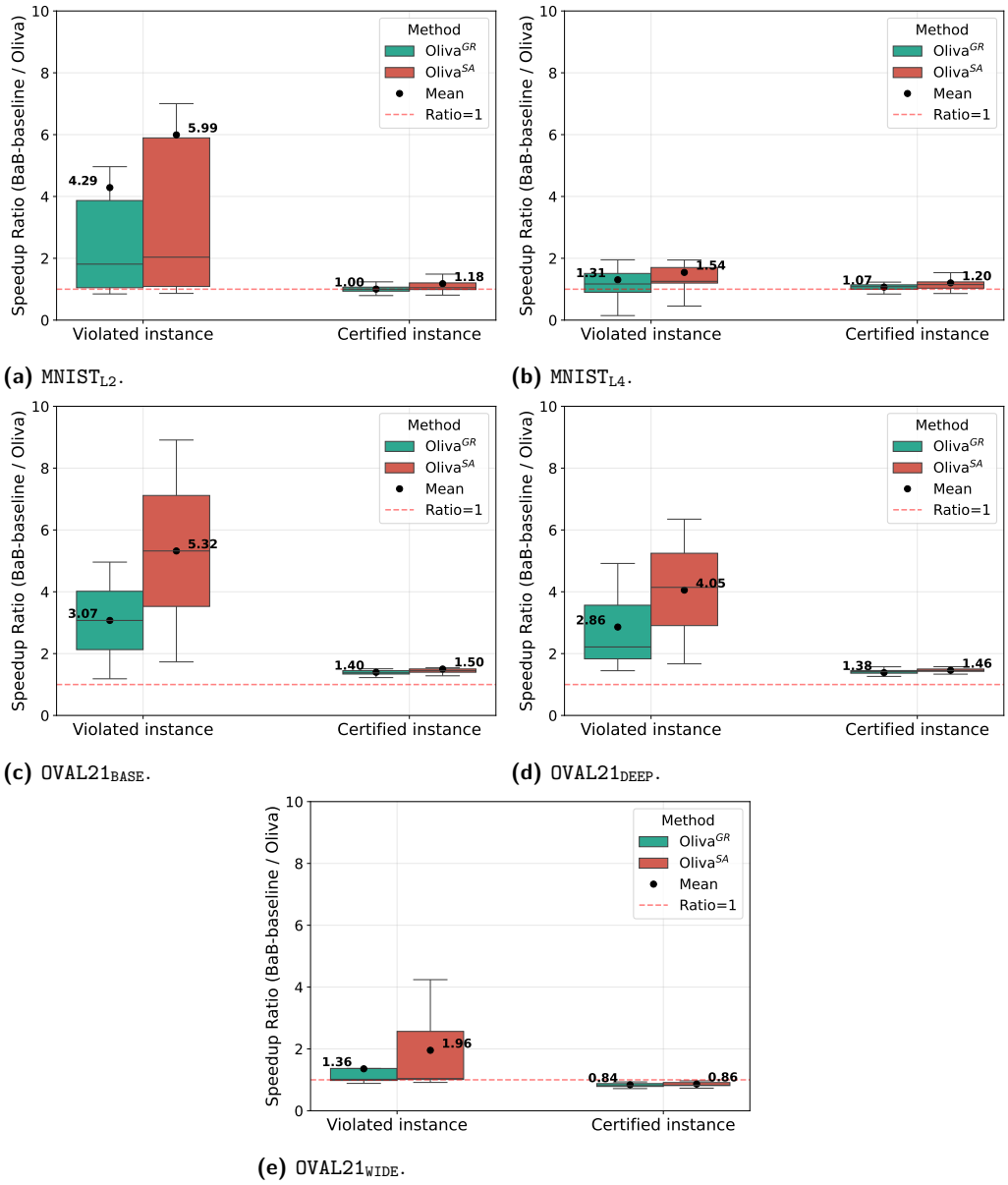
(a) SA attempt 1. **(b)** SA attempt 2. **(c)** SA attempt 3.



(d) SA attempt 4. **(e)** SA attempt 5.

Figure 6 The distributions of conclusive instances by the variant of Oliva^{SA}, in different speedups and time costs compared with Oliva^{GR}. The black dashed line indicates the performance of Oliva^{GR}: above the black dashed line, Oliva^{GR} outperforms Oliva^{SA}; below it, Oliva^{SA} performs better.

not substantially impact the effectiveness of Oliva^{SA}. Notably, in Fig 6c and Fig 6d, with the nature of the randomness, Oliva^{SA} exhibits the strength in finding solutions in the time cost of 100-600s that might be difficult to solve by the deterministic approach of Oliva^{GR}.



(a) $\texttt{MNIST}_{\texttt{L2}}$. **(b)** $\texttt{MNIST}_{\texttt{L4}}$.

(c) $\texttt{OVAL21}_{\texttt{BASE}}$. **(d)** $\texttt{OVAL21}_{\texttt{DEEP}}$.

(e) $\texttt{OVAL21}_{\texttt{WIDE}}$.

Figure 7 RQ2 – Comparison between BaB-baseline and Oliva for violated and certified verification problem instances.

RQ2: How effective is Oliva in handling violated and certified problem instances, respectively?

Fig. 7 shows the performance advantage of Oliva^{GR} and Oliva^{SA} over BaB-baseline across different verification tasks. For violated instances (i.e., confirmed counterexample by BaB-baseline and Oliva), Oliva^{GR} and Oliva^{SA} consistently demonstrate superior efficiency, evidenced by lower median time costs and smaller interquartile ranges across all models.

In Fig. 7a, Oliva shows notably lower median time costs and compressed interquartile ranges compared to BaB-baseline, indicating more consistent and faster execution. In $\texttt{OVAL21}_{\texttt{BASE}}$ and $\texttt{OVAL21}_{\texttt{DEEP}}$ (Fig. 7c and Fig. 7d), Oliva^{GR} and Oliva^{SA} sub-problem selection

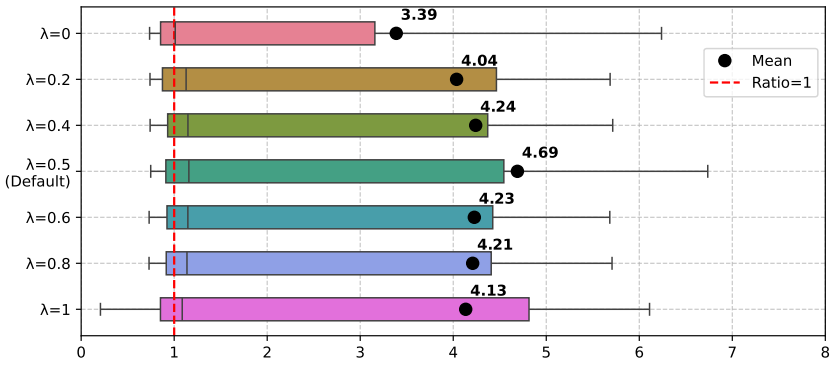


Figure 8 RQ3 – Speedup over BaB-baseline under different hyperparameter λ values on verification tasks with $\texttt{OVAL21}_{\texttt{WIDE}}$.

strategy is highly effective in problem instances where counterexamples exist. For $\texttt{OVAL21}_{\texttt{BASE}}$, Oliva^{GR} achieves a median speed-up ratio of approximately 2.5× for violated instances, and Oliva^{SA} with reaching up to 4× faster execution than BaB-baseline. Similarly, for $\texttt{OVAL21}_{\texttt{DEEP}}$, the median speed-up ratio is around 3×, with peak performance showing up to 5× acceleration.

In contrast, for certified instances (i.e., confirmed robust), two variants of Oliva are generally on par with the BaB-baseline, indicating that in these cases, the performance of Oliva is comparable with BaB-baseline, which does not introduce overhead costs. This is expected, because in BaB, most of the time consumption is devoted to the problem solving process for each sub-problem. While our proposed approach introduces little overhead in the selection of the sub-problems to proceed with, the overhead is almost invisible.

RQ3: How does the hyperparameter λ and α in the order influences the performance of Oliva?

Fig. 8 highlights the impact of the λ parameter on the speedup of Oliva^{GR} over BaB-baseline for the verification task on $\texttt{OVAL21}_{\texttt{WIDE}}$ model. In this experiment, we randomly select 20% of the instances, including 13 conclusive and 12 unknown cases, based on the BaB-baseline performance. The chosen instances are at varying levels of time consumption. This analysis examines λ values ranging from 0.0 to 1.0, and summarizes the speedup ratios of Oliva^{GR} under different λ values. The findings reveal that, first, the performances of Oliva^{GR} under different λ are relatively stable, mostly outperforming BaB-baseline. This implies that both of the attributes, including the level of problem splitting and the verifier assessment, are effective in guiding the space exploration. Moreover, the performance slightly improves as λ increases from 0.0 to 0.5, peaking at a speedup of 4.69. Beyond $\lambda = 0.5$, speedup declines, indicating 0.5 as the optimal value. In terms of the average improvement, the speedup vary from 3.39 to 4.69 and is not negligible, which emphasizes the importance of careful tuning of λ to maximize the efficiency of Oliva^{GR} in verification tasks.

Then, we settle $\lambda = 0.5$ as the default value to evaluate the temperature reduction parameter α, which ranges from 0.95 to 0.999, as suggested by [24]. Fig. 9 compares the speedup of Oliva^{SA} over Oliva^{GR}. The hyperparameter α determines the speed of "temperature" decreases during the verification processes, directly affecting the balances between exploration and exploitation. First, by the medians, we find that Oliva^{SA} under all α outperform Oliva^{GR}. Specifically, with $\alpha = 0.999$, Oliva^{SA} achieves better performance than Oliva^{GR} in average,

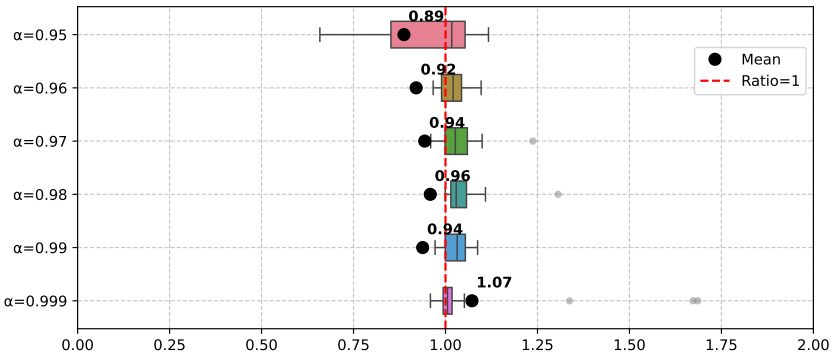


Figure 9 RQ3 – Speedup over Oliva$^{\text{GR}}$ under hyperparameter α values on verification tasks with $\texttt{OVAL21}_{\texttt{WIDE}}$.

which is a very slow cooling schedule (α close to 1) that allows the algorithm to thoroughly explore the search space early on before transitioning to exploitation. As α decreases, the temperature drops more rapidly, giving the algorithm less time to explore before converging on promising areas. The lower performance ratios seen with $\alpha = 0.95 - 0.98$ suggest that this faster cooling schedule may be too aggressive, causing Oliva$^{\text{SA}}$ to commit too quickly to certain branches of the verification tree without adequately exploring alternatives. By our inspection of experimental results, the relative low mean values are caused by some specific cases, for which Oliva$^{\text{SA}}$ does not perform well. Notably, when selecting $\alpha = 0.99$, while its mean ratio of 0.94 might seem underwhelming at a first glance, its median performance is notably strong compared to other α values, representing a balanced probability that allows sufficient exploration while maintaining steady progress toward exploitation. In practical verification tasks, as consistency can be as a valuable property, $\alpha = 0.99$ can thus be preferable despite its lower mean ratio.

6 Related Work

Neural Network Verification

Neural network verification has been extensively studied in the past few years, giving birth to many practical approaches [47, 23, 14, 22, 41, 44, 43, 29, 49, 39]. Approximated verifiers are perferable due to their efficiency, and many works aim to seek for tighter bounds for approximation refinement [2, 46, 40, 32, 31, 54, 27, 36, 15, 13, 9]. Some works aim to refine the approximation [55, 34, 20, 12, 60] by exploiting information from (spurious) counterexamples (known as CEGAR approach). In contrast, our approach does not aim to refine the approximation of the backend verifiers, but we estimate the probability of finding counterexamples in different sub-problems to efficiently explore the sub-problem space.

Problem Splitting Strategies in BaB

Problem splitting [38, 26, 16] is an important component in the `BaB` workflow, which is critical to the efficiency of verification [4]. Early research considers splitting input space by its dimensions [50, 1, 37]; however, that can lead to an exponential growth in the number of sub-problems w.r.t. the number of dimensions, which is intractable especially for applications that have high-dimensional data, such as image classification. More recently, many works start to perform problem splitting by decomposing ReLU functions, which can achieve

better performance [5] and scales better than input domain splitting. In these works, the ReLU selection strategy that decides the next neuron (i.e., the ReLU function) to be split is important and there emerge many effective strategies including DeepSplit [21] (which we adopted), BaBSR [5] and FSB [10]. Compared to those works, our approach is different and is orthogonal to them, because instead of selecting ReLU, we select the sub-problems to proceed with, which changes the naive breadth-first strategy of `BaB` in visiting the sub-problems. Notably, our work is in line with [57]; however, unlike [57] that targets incremental neural network verification, we target the problem of classic neural network verification, and we adopt a different stochastic optimization framework, i.e., simulated annealing.

Testing and Attacks

Testing is another effective quality assurance for neural networks, and it has also been extensively studied, such as [35, 33, 17, 45]. A similar line of works are adversarial attacks, that aim to generate adversarial examples [18, 3, 53, 58, 6] to fool the neural networks. While these approaches are efficient, they cannot provide rigorous guarantee on specification satisfaction, when they cannot find a counterexample, because they solve the problem by checking infinitely many single inputs in the input space. In comparison, our approach is sound, in the sense that if there does not exist a counterexample in the network, we can finally reach the certification of the network. This is because by our problem splitting, we deal with finitely many sub-problems.

7 Conclusion and Future Work

In this paper, we propose an approach Oliva with two variants Oliva^{GR} and Oliva^{SA} in order to achieve high efficiency of neural network verification. Oliva introduces a severity order over the sub-problems produced by `BaB`. This order, called counterexample potentiality, estimates the suspiciousness of each sub-problem in the sense how likely it contains a counterexample, based on both the level of problem splitting and the assessment from approximated verifiers that signify how far a sub-problem is from being violated. By prioritizing the sub-problems with higher counterexample potentiality, Oliva can efficiently reach verification conclusions – either finding a counterexample quickly or certifying the neural network without significant performance degradation. Specifically, our two variants of Oliva implement different strategies: *i*) Oliva^{GR} greedily exploits the most suspicious sub-problems; *ii*) Oliva^{SA}, inspired by simulated annealing, strikes a balance between exploitation of the promising sub-problems and exploration of the sub-problems that are less promising. Our experimental evaluation across 5 neural network models and 690 verification problems demonstrates that Oliva can achieve up to 80× speedup, compared to the state-of-the-art baseline approaches. Moreover, we also show the breakdown comparison for the certified problems and for the falsified problems, and we demonstrate that the performance advantages of the proposed approach indeed come from the strategy guided by counterexample potentiality.

To the best of our knowledge, our work is the first that exploits the power of stochastic optimization in neural network verification, by taking the sub-problems as the search space. In future, we plan to do more exploration in this direction, e.g., trying other stochastic optimization algorithms, such as genetic algorithms, and comparing the performances of different strategies, in order to deliver better verification approaches.

References

1 Greg Anderson, Shankara Pailoor, Isil Dillig, and Swarat Chaudhuri. Optimization and abstraction: a synergistic approach for analyzing neural network robustness. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*, pages 731–744, 2019. doi:10.1145/3314221.3314614.

2 R Anderson, J Huchette, C Tjandraatmadja, and JP Vielma. Strong convex relaxations and mixed-integer programming formulations for trained neural networks (2018), 1811.

3 Maksym Andriushchenko, Francesco Croce, Nicolas Flammarion, and Matthias Hein. Square attack: a query-efficient black-box adversarial attack via random search. In *European conference on computer vision*, pages 484–501. Springer, 2020. doi:10.1007/978-3-030-58592-1_29.

4 Rudy Bunel, Jingyue Lu, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and M. Pawan Kumar. Branch and bound for piecewise linear neural network verification. *CoRR*, abs/1909.06588, 2019. doi:10.48550/arXiv.1909.06588.

5 Rudy Bunel, P Mudigonda, Ilker Turkaslan, Philip Torr, Jingyue Lu, and Pushmeet Kohli. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research*, 21(2020), 2020. URL: https://jmlr.org/papers/v21/19-468.html.

6 Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, pages 39–57. Ieee, 2017. doi:10.1109/SP.2017.49.

7 Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. Maximum resilience of artificial neural networks. In Deepak D'Souza and K. Narayan Kumar, editors, *Automated Technology for Verification and Analysis*, pages 251–268. Springer Int. Publishing, 2017. doi:10.1007/978-3-319-68167-2_18.

8 Moumita Das, Rajarshi Ray, Swarup Kumar Mohalik, and Ansuman Banerjee. Fast falsification of neural networks using property directed testing. *arXiv preprint arXiv:2104.12418*, 2021. doi:10.48550/arXiv.2104.12418.

9 Sumanth Dathathri, Krishnamurthy Dvijotham, Alexey Kurakin, Aditi Raghunathan, Jonathan Uesato, Rudy R Bunel, Shreya Shankar, Jacob Steinhardt, Ian Goodfellow, Percy S Liang, et al. Enabling certification of verification-agnostic networks via memory-efficient semidefinite programming. *Advances in Neural Information Processing Systems*, 33:5318–5331, 2020.

10 Alessandro De Palma, Rudy Bunel, Alban Desmaison, Krishnamurthy Dvijotham, Pushmeet Kohli, Philip HS Torr, and M Pawan Kumar. Improved branch and bound for neural network verification via lagrangian decomposition. *arXiv preprint arXiv:2104.06718*, 2021.

11 Hai Duong, Linhan Li, ThanhVu Nguyen, and Matthew B. Dwyer. A DPLL(T) framework for verifying deep neural networks. *CoRR*, abs/2307.10266, 2023. doi:10.48550/arXiv.2307.10266.

12 Hai Duong, Dong Xu, ThanhVu Nguyen, and Matthew B Dwyer. Harnessing neuron stability to improve dnn verification. *Proceedings of the ACM on Software Engineering*, 1(FSE):859–881, 2024. doi:10.1145/3643765.

13 Krishnamurthy Dj Dvijotham, Robert Stanforth, Sven Gowal, Chongli Qin, Soham De, and Pushmeet Kohli. Efficient neural network verification with exactness characterization. In *Uncertainty in artificial intelligence*, pages 497–507. PMLR, 2020.

14 Ruediger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *Automated Technology for Verification and Analysis: 15th Int. Symp., ATVA 2017, Proceedings 15*, pages 269–286. Springer, October 2017.

15 Mahyar Fazlyab, Manfred Morari, and George J Pappas. Safety verification and robustness analysis of neural networks via quadratic constraints and semidefinite programming. *IEEE Transactions on Automatic Control*, 67(1):1–15, 2020.

16 Claudio Ferrari, Mark Niklas Muller, Nikola Jovanovic, and Martin Vechev. Complete verification via multi-neuron relaxation guided branch-and-bound. *arXiv preprint arXiv:2205.00263*, 2022. doi:10.48550/arXiv.2205.00263.

17 Xiang Gao, Ripon K Saha, Mukul R Prasad, and Abhik Roychoudhury. Fuzz testing based data augmentation to improve robustness of deep neural networks. In *Proceedings of the acm/ieee 42nd international conference on software engineering*, pages 1147–1158, 2020. doi:10.1145/3377811.3380415.

18 Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *3rd Int. Conf. on Learning Representations (ICLR'15)*, San Diego, CA, United States, 2015. Int. Conf. on Learning Representations, ICLR.

19 Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL: https://www.gurobi.com.

20 Karthik Hanumanthaiah and Samik Basu. Iterative counter-example guided robustness verification for neural networks. In *International Symposium on AI Verification*, pages 179–187. Springer, 2024. doi:10.1007/978-3-031-65112-0_9.

21 Patrick Henriksen and Alessio Lomuscio. Deepsplit: An efficient splitting method for neural network verification via indirect effect analysis. In *IJCAI*, pages 2549–2555, 2021. doi:10.24963/ijcai.2021/351.

22 Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *Computer Aided Verification: 29th Int. Conf., CAV 2017, Part I 30*, pages 3–29. Springer, July 2017. doi:10.1007/978-3-319-63387-9_1.

23 Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 97–117. Springer Int. Publishing, 2017. doi:10.1007/978-3-319-63387-9_5.

24 Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983. doi:10.1126/science.220.4598.671.

25 Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher Strong, Clark Barrett, Mykel J Kochenderfer, et al. Algorithms for verifying deep neural networks. *Foundations and Trends® in Optimization*, 4(3-4):244–404, 2021. doi:10.1561/2400000035.

26 Jingyue Lu and M Pawan Kumar. Neural network branching for neural network verification. *arXiv preprint arXiv:1912.01329*, 2019. arXiv:1912.01329.

27 Zhongkui Ma, Jiaying Li, and Guangdong Bai. Relu hull approximation. *Proceedings of the ACM on Programming Languages*, 8(POPL):2260–2287, 2024. doi:10.1145/3632917.

28 Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *6th Int. Conf. on Learning Representations (ICLR'18)*, Vancouver, Canada, 2018.

29 Christoph Müller, Gagandeep Singh, Markus Püschel, and Martin T Vechev. Neural network robustness verification on gpus. *CoRR, abs/2007.10868*, 2020.

30 Mark Niklas Müller, Christopher Brix, Stanley Bak, Changliu Liu, and Taylor T Johnson. 3rd international verification of neural networks competition (VNN-COMP 2022): Summary and results. *arXiv preprint arXiv:2212.10376*, 2022. doi:10.48550/arXiv.2212.10376.

31 Mark Niklas Müller, Gleb Makarchuk, Gagandeep Singh, Markus Püschel, and Martin Vechev. Precise multi-neuron abstractions for neural network certification. *arXiv preprint arXiv:2103.03638*, 2021. arXiv:2103.03638.

32 Mark Niklas Müller, Gleb Makarchuk, Gagandeep Singh, Markus Püschel, and Martin Vechev. Prima: general and precise neural network certification via scalable convex hull approximations. *ACM on Programming Languages*, 6(POPL):1–33, 2022. doi:10.1145/3498704.

33 Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *International Conference on Machine Learning*, pages 4901–4911. PMLR, 2019. URL: http://proceedings.mlr.press/v97/odena19a.html.

34 Matan Ostrovsky, Clark Barrett, and Guy Katz. An abstraction-refinement approach to verifying convolutional neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 391–396. Springer, 2022. doi:10.1007/978-3-031-19992-9_25.

35 Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *26th Symp. on Operating Systems Principles*, pages 1–18, 2017. doi:10.1145/3132747.3132785.

36 Aditi Raghunathan, Jacob Steinhardt, and Percy S Liang. Semidefinite relaxations for certifying robustness to adversarial examples. *Advances in neural information processing systems*, 31, 2018.

37 Vicenc Rubies-Royo, Roberto Calandra, Dusan M Stipanovic, and Claire Tomlin. Fast neural network verification via shadow prices. *arXiv preprint arXiv:1902.07247*, 2019. arXiv:1902.07247.

38 Zhouxing Shi, Qirui Jin, Zico Kolter, Suman Jana, Cho-Jui Hsieh, and Huan Zhang. Neural network verification with branch-and-bound for general nonlinearities. *arXiv preprint arXiv:2405.21063*, 2024. doi:10.48550/arXiv.2405.21063.

39 Zhouxing Shi, Yihan Wang, Huan Zhang, J Zico Kolter, and Cho-Jui Hsieh. Efficiently computing local lipschitz constants of neural networks via bound propagation. *Advances in Neural Information Processing Systems*, 35:2350–2364, 2022.

40 Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin Vechev. Beyond the single neuron convex barrier for neural network certification. *Advances in Neural Information Processing Systems*, 32, 2019.

41 Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. Fast and effective robustness certification. In *Advances in Neural Information Processing Systems*, volume 31, 2018.

42 Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. Fast and effective robustness certification. *Advances in neural information processing systems*, 31, 2018. URL: https://proceedings.neurips.cc/paper/2018/hash/f2f446980d8e971ef3da97af089481c3-Abstract.html.

43 Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. Boosting robustness certification of neural networks. In *Int. Conf. on learning representations*, 2018.

44 Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *ACM on Programming Languages*, 3(POPL):1–30, 2019. doi:10.1145/3290354.

45 Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *40th Int. Conf. on Software Engineering*, pages 303–314, 2018. doi:10.1145/3180155.3180220.

46 Christian Tjandraatmadja, Ross Anderson, Joey Huchette, Will Ma, Krunal Kishor Patel, and Juan Pablo Vielma. The convex relaxation barrier, revisited: Tightened single-neuron relaxations for neural network verification. *Advances in Neural Information Processing Systems*, 33:21675–21686, 2020.

47 Vincent Tjeng, Kai Y Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *Int. Conf. on Learning Representations*, 2018.

48 Shubham Ugare, Debangshu Banerjee, Sasa Misailovic, and Gagandeep Singh. Incremental verification of neural networks. *ACM on Programming Languages*, 7(PLDI):1920–1945, 2023. doi:10.1145/3591299.

49 Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. *Advances in neural information processing systems*, 31, 2018. URL: https://proceedings.neurips.cc/paper/2018/hash/2ecd2bd94734e5dd392d8678bc64cdab-Abstract.html.

50 Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal security analysis of neural networks using symbolic intervals. In *27th USENIX Security Symp. (USENIX Security 18)*, pages 1599–1614, 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/wang-shiqi.

51 Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. *Advances in Neural Information Processing Systems*, 34:29909–29921, 2021. URL: https://proceedings.neurips.cc/paper/2021/hash/fac7fead96dafceaf80c1daffeae82a4-Abstract.html.

52 Haoze Wu, Omri Isac, Aleksandar Zeljic, Teruhiro Tagomori, Matthew L. Daggitt, Wen Kokke, Idan Refaeli, Guy Amir, Kyle Julian, Shahaf Bassan, Pei Huang, Ori Lahav, Min Wu, Min Zhang, Ekaterina Komendantskaya, Guy Katz, and Clark W. Barrett. Marabou 2.0: A versatile formal analyzer of neural networks. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II*, volume 14682 of *Lecture Notes in Computer Science*, pages 249–264. Springer, Springer, 2024. doi:10.1007/978-3-031-65630-9_13.

53 Cihang Xie, Zhishuai Zhang, Yuyin Zhou, Song Bai, Jianyu Wang, Zhou Ren, and Alan L Yuille. Improving transferability of adversarial examples with input diversity. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2730–2739, 2019. doi:10.1109/CVPR.2019.00284.

54 Xiaoyong Xue, Xiyue Zhang, and Meng Sun. kprop: Multi-neuron relaxation method for neural network robustness verification. In *International Conference on Fundamentals of Software Engineering*, pages 142–156. Springer, 2023. doi:10.1007/978-3-031-42441-0_11.

55 Pengfei Yang, Renjue Li, Jianlin Li, Cheng-Chao Huang, Jingyi Wang, Jun Sun, Bai Xue, and Lijun Zhang. Improving neural network verification through spurious region guided refinement. In *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–408. Springer, 2021. doi:10.1007/978-3-030-72016-2_21.

56 Yang Yu, Hong Qian, and Yi-Qi Hu. Derivative-free optimization via classification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016. doi:10.1609/aaai.v30i1.10289.

57 Guanqin Zhang, Zhenya Zhang, H.M.N. Dilum Bandara, Shiping Chen, Jianjun Zhao, and Yulei Sui. Efficient incremental verification of neural networks guided by counterexample potentiality. *Proc. ACM Program. Lang.*, 9(OOPSLA1), April 2025. doi:10.1145/3720417.

58 Huan Zhang, Shiqi Wang, Kaidi Xu, Yihan Wang, Suman Jana, Cho-Jui Hsieh, and Zico Kolter. A branch and bound framework for stronger adversarial attacks of relu networks. In *International Conference on Machine Learning*, pages 26591–26604. PMLR, 2022. URL: https://proceedings.mlr.press/v162/zhang22ae.html.

59 Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neural network robustness certification with general activation functions. *Advances in Neural Information Processing Systems*, 31, 2018. URL: https://proceedings.neurips.cc/paper/2018/hash/d04863f100d59b3eb688a11f95b0ae60-Abstract.html.

60 Zhe Zhao, Yedi Zhang, Guangke Chen, Fu Song, Taolue Chen, and Jiaxiang Liu. Cleverest: accelerating cegar-based neural network verification via adversarial attacks. In *International Static Analysis Symposium*, pages 449–473. Springer, 2022. doi:10.1007/978-3-031-22308-2_20.

RacerF: Lightweight Static Data Race Detection for C Code

Tomáš Dacík
Faculty of Information Technology, Brno University of Technology, Czech Republic

Tomáš Vojnar
Faculty of Informatics, Masaryk University, Brno, Czech Republic
Faculty of Information Technology, Brno University of Technology, Czech Republic

Abstract

We present RacerF, a novel static analyser for thread-modular data race detection. The approach behind RacerF exploits static analysis of sequential program behaviour whose results are generalised for multi-threaded programs using a combination of lightweight under- and over-approximating methods. The tool is implemented as a plugin of the Frama-C platform and can leverage several analysis backends, most notably the Frama-C's abstract interpreter EVA. Although our methods are mostly heuristic without providing formal guarantees, our experimental evaluation shows that even for intricate programs, RacerF can provide very precise results competitive with more heavyweight approaches while being faster than them.

2012 ACM Subject Classification Software and its engineering → Automated static analysis

Keywords and phrases concurrency, data race detection, static analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.37

Category Experience Paper

Supplementary Material
Software (Source Code): `https://github.com/TDacik/Deadlock_and_Racer` [11]
archived at `swh:1:dir:fc33aed78b227ea0738a89894b3dfa91b4f46844`

Funding The research was supported by the Czech Science Foundation project 23-06506S and the FIT BUT internal project FIT-S-23-8151. Tomáš Dacík was also supported by the Brno PhD Talent Scholarship of the Brno City Municipality. The research involved discussions with Julien Signoles from the Frama-C team at CEA that were supported by the project VASSAL: "Verification and Analysis for Safety and Security of Applications in Life" funded by the European Union under Horizon Europe WIDERA Coordination and Support Action/Grant Agreement No. 101160022.

Acknowledgements We appreciate discussions on the subject with Julien Signoles from the Frama-C team at CEA, France and with Adam Rogalewicz from FIT BUT, Czechia.

1 Introduction

Data races in concurrent programs belong among the most nasty bugs in computer programs: they are easy to cause, hard to spot during code inspection, and hard to catch by testing as they can manifest only very rarely. Due to this, a lot of effort has gone into designing as safe as possible libraries and languages for writing concurrent programs, but, on one hand, many programs are written using languages without such safety features (with the C language being a prominent example), and on the other hand, even in the safety-oriented languages such as Rust, unsafe programming is usually allowed and sometimes used, e.g., for performance reasons.

A lot of work has also been invested in designing various static as well as dynamic methods of finding data races and/or proving their absence. However, the complex behaviour of concurrent programs reflects in that such methods do often have problems with scalability

© Tomáš Dacík and Tomáš Vojnar;
licensed under Creative Commons License CC-BY 4.0

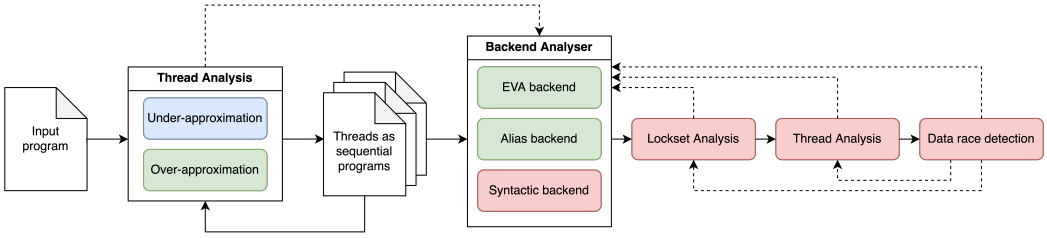


Figure 1 An overview of the workflow of RacerF. The phases with under- and over-approximation guarantees are marked by blue and green colour, respectively. The phases designed without guarantees are marked by red colour. The solid lines represent the workflow of individual analysis phases. The dashed lines represent queries on the results of other phases.

or precision. In this paper, we present a novel tool RacerF for thread-modular data race detection with the aim of providing a suitable balance between the scalability and precision of the analysis. We then also present our experience from running the tool and comparing it with several other data race detectors.

The approach implemented in RacerF relies on leveraging results of static analysis of the sequential behaviour of program threads for finding data races in multi-threaded programs. We use two strategies of combining information resulting from the analysis of the sequential behaviour of program threads: one under-approximates and one over-approximates the concurrent behaviour of the threads running in parallel. We combine these strategies to obtain a more precise analysis such that the under-approximation is used to find data races while the over-approximation is used to show their absence.

RacerF is implemented as a plugin of the Frama-C platform [10, 1] and is in particular designed for finding data races in C programs with POSIX threads (pthreads). RacerF can use several analysis backends implementing the underlying analysis of the sequential behaviour of program threads and differing in their precision and scalability. One of them is built on top of the Frama-C's value analysis plugin EVA based on abstract interpretation [6], one is purely syntactic, and the last one is based on Alias, Frama-C's plugin for may-alias and points-to analysis [5].

We have performed experiments with RacerF on a number of benchmarks coming both from the SV-COMP competition in software verification, which includes both synthetic benchmarks as well as benchmarks derived from real-world code, as well as on a number of student projects in an advanced course of operating systems. Although the approach behind RacerF is mostly heuristic without providing formal guarantees, our experimental evaluation shows that even for intricate programs, RacerF can provide very precise results competitive with more heavy-weight approaches while being faster than them.

Approach Overview

An overview of the workflow of RacerF is illustrated in Fig. 1. The core component is a *backend analyser* which answers several kinds of queries based on analysing individual threads as sequential programs (the general interface and several different backend analysers are described in Section 4). The *thread analysis* (Section 5) incrementally discovers new program threads and analyses them using the chosen backend analyser. Since the discovery of a new thread may add additional information, the process of thread discovery and their analysis using the backend analyser is repeated until a fixpoint is reached. Then, two helper

analyses are run to deal with checking whether two memory accesses may happen in parallel – a *lockset analysis* checks which accesses are guarded by which locks and an *active-threads analysis* checks when threads are created and joined (Section 6). Those analyses query results of the backend analysis, mainly for possible values of parameters of locking and threading functions. Finally, *data race detection* (Section 7) is performed. This phase checks memory accesses (using information provided by the backend analyser) and tracks whether they need to be protected and, if so, whether they indeed are. The detected issues are then classified into may- and must-data-races. Must-data-races are reported to the user, while an absence of may-data-races is used to claim program race-free.

Fig. 1 also depicts the approximation guarantees provided by the individual phases. Note that the combination of under- and over-approximating methods may seem problematic from the point of view of what can be guaranteed to hold for the produced results. However, we do not aim at providing guarantees in the sense of formal verification. Our aim is to achieve high efficiency and precision in practice even though some false alarms may be missed or some bugs not identified.

2 Related Work

RacerX [15] is a static analyser for C based primarily on a lockset analysis. The tool has been successfully used to find bugs in the kernels of several operating systems. Compared to our approach, RacerX uses a very simple pointer analysis for the representation of locks only. Further, RacerX uses many heuristics in individual analysis phases to deal with analysis mistakes while RacerF tries to achieve precision using careful ranking. RacerX is unfortunately not available for an experimental comparison. RacerD [4] is a compositional analyser implemented in the Infer framework. It provides scalability by analysing each function only once. Unlike us, RacerD targets Java programs with structural locking, assuming that all locking operations are balanced. Both RacerX and RacerD focus on under-approximation only. O2 [26] is another static checker building on the notion of origins unifying threads and events. The approach was implemented for C, C++, and Java/Android; and used to find many previously unknown bugs in real-world code. However, as we show in our experiments, it can still miss many data races in smaller but real-world programs.

On the other side, there are tools focused on soundness, rather than data race detection. Goblint [32] and Astrée [24, 27] rely on a thread-modular abstract interpretation to soundly over-approximate thread behaviour. The work [32] further proposes a way to handle common locking schemes in Linux driver devices. In our approach, we are not interested in the analysis of multithreaded programs in general and we can consequently afford more drastic approximations that seem to be sufficient for data race detection. We also focus on under-approximation to detect bugs.

As an alternative to static data race detection, various approaches based on *testing* and *dynamic analysis* can be used. A big challenge for these approaches is the fact that errors like data races can manifest in very rare behaviours only, which are difficult to catch, even through many repeated test runs. Coverage of such rare behaviours can be improved using approaches such as *systematic testing* [33] or *noise-based testing* [13, 17]. Another way is to use *extrapolating dynamic checkers*, such as [30, 14, 19], which can report warnings about possible errors even if those are not seen in the testing runs, based on spotting their symptoms. However, even though such checkers have proven quite useful in practice, they can, of course, still miss rarely occurring errors. Moreover, monitoring a run of large software

through such checkers may be quite expensive too.

Approaches based on *model checking*, i.e., exhaustive state-space exploration, can guarantee the discovery of all potentially present errors – either in general or at least up to some bound, often given in the number of context switches. However, so far, the scalability of these techniques is not sufficient to handle large industrial code, even when combined with methods such as *sequentialisation* [25, 28] or advanced techniques based on *automata* [12, 23] or *SMT encodings* [20, 21, 16]. Indeed, this shows up in our experiments too.

3 Preliminaries

We now introduce several basic notions that we use when describing the analysed programs and the analyses proposed.

Concurrency model. We primarily target C programs using threads and mutexes in the style of the POSIX threads (pthreads) execution model. We use the term *thread* to mean thread's entry point, i.e., an identifier of the function from which the given thread is started. In the examples, we shorten the calls of locking, unlocking, thread-creating, and thread-joining functions as `lock`(...), `unlock`(...), `create`(...), and `join`(...), respectively. We further implicitly assume that all locks are properly initialised and omit the arguments that are not relevant, e.g., we write just `create`(`t1`), where `t1` is an entry point, when the thread id, attributes, and argument are not relevant for the example.

The CValue domain and memory addresses. Throughout the paper, we frequently refer to the CValue abstract domain [6, 10] implemented in Frama-C and used by the abstract interpretation of EVA. We are primarily interested in the representation of memory addresses. A memory address is represented as a pair (b, o) where b is a *base address* (in the following referred to simply as a *base*) and o is an offset. Each variable (global, local, and also formal) has its associated base address. Dynamic allocations are modelled by creating so-called *dynamic bases* which come in two flavours – a *strong* dynamic base corresponds to the result of precisely one call to an allocation function; on the other hand, *weak* memory bases may represent multiple allocated addresses (typically, they are the consequence of allocations in loops). Offsets are represented by a dedicated integer domain, the details of which are not important for this paper. We use CValue.val for the domain representing possible values of a single variable and CValue.state for the domain representing program state, i.e., possible values for multiple variables. We use Addresses to denote the aforementioned set of memory addresses. In the rest of the paper, the operations *join* and *widening* as well as *top values* refer to those on the CValue domain.

Analysis contexts. We call an *analysis context* a triple $(thread, calls, stmt)$ consisting of a thread, a sequence of the latest n calls (each of them formed by a call site and the function called), and a program statement. Our data race analysis relies on various underlying analyses presented below to compute an abstract representation of thread states reachable by performing the given statement for each analysis context. Notice that a context holds no information about how the thread was created nor how its execution was interleaved with other threads. The empty call sequence is denoted by ε. The number n is a parameter of the analysis set to 1 by default to handle simple lock/unlock wrapper functions. We use Context to denote the set of all possible contexts.

$$
\begin{aligned}
\mathsf{analyse_thread} &: \mathsf{Function} \times \mathsf{CValue.state} \to \mathsf{Result} && \text{(sequential analysis of a thread)}\\
\mathsf{state} &: \mathsf{Result} \times \mathsf{Context} \to \mathsf{CValue.state} && \text{(state before a statement)}\\
\mathsf{value} &: \mathsf{Result} \times \mathsf{Context} \times \mathsf{Expr} \to \mathsf{CValue.val} && \text{(non-pointer expression values)}\\
\mathsf{value_ptr} &: \mathsf{Result} \times \mathsf{Context} \times \mathsf{Expr} \to 2^{\mathsf{Address}} && \text{(pointer expression values)}\\
\mathsf{functions} &: \mathsf{Result} \times \mathsf{Context} \times \mathsf{Expr} \to 2^{\mathsf{Function}} && \text{(function pointer resolution)}\\
\mathsf{accesses} &: \mathsf{Result} \times \mathsf{Context} \times \mathsf{Expr} \to (2^{\mathsf{Address}})^2 && \text{(memory reads and writes)}
\end{aligned}
$$

Figure 2 The interface of the backend analysis used by the proposed analysis of data races. The function analyse_thread performs the analysis of a thread as a sequential program given by its entry point, starting from the given initial state. The other functions answer queries based on the results computed by analyse_thread (specific for different backends). Note that the signatures do not include statements as those are already included in contexts.

4 Backend Analysis

Our data race analysis relies on a backend analysis of sequential program behaviour to get answers for several kinds of queries. Those include *function pointer resolution* (to determine which functions are to be analysed when and to determine thread entry points passed to thread create functions), getting *points-to information* (to determine targets of (un)lock operations, to perform escape analysis, and, most importantly, to determine on which addresses to look for data races), getting information on *memory accesses*, and characterizing *values of integer variables* (to determine whether offsets of two accesses overlap, and to check whether a trylock operation succeeds in the given branch of code).

To offer a selection of multiple backend analyses differing in their scalability, precision, and ease of use, we have implemented several such analyses. The interface a backend needs to implement is shown in Fig. 2. The function analyse_thread runs the analysis of a thread as a sequential program starting from the provided state of global variables and the thread's argument. Based on the results, the different queries are then answered.

Evolved Value Analysis (EVA) backend. This backend is based on EVA [6], a value analysis plugin of Frama-C based on abstract interpretation. EVA provides a combination of multiple abstract domains and an API to answer all the needed queries, and our backend is just a thin wrapper over EVA, which implements caching, simplification, and answering some trivial queries on its own. The interface in Fig. 2 is in fact inspired by the API of EVA, and the other backends mimic it. In particular, the EVA's API implements queries for call stacks which can be easily obtained from our contexts.

Syntactic backend. This backend provides a lightweight analysis based only on syntactical information in the style of RacerX [15] or RacerD [4]. The implementation of analyse_thread is simply an empty function and the functions state and value always return the top value. The function value_ptr searches the expression and extracts bases of the expected type. This is inspired by the insight that in some programs only global locks are accessed using `lock(&m)` and `unlock(&m)` (where `m` is a global variable), and it is thus sufficient to simply extract the base of `m`. The function memory_accesses is implemented similarly with a more detailed analysis of an expression to determine whether a base is being read or written to it. Finally,

```
1  void *t1(void *arg1) {
2    int *x = (int *)arg1;
3    (*x)++;
4  }
5
6  void *t2(void *arg2) {
7    int *y = (int *)arg2;
8    (*y)++;
9  }
10 int main() {
11   int data = 0;
12
13   create(t1, &data);
14   create(t2, &data);
15 }
```

Figure 3 An example of a program for which the Syntactic backend fails to find a data race.

function pointers are over-approximated by taking each function assigned in the program. In the case of thread entry points, this can be further refined by taking only functions matching the thread signature (i.e., `void *(*fn)(void *arg)` for pthreads).

Alias backend. The aim of the third backend is to provide a middle ground between the precision of the EVA and the scalability of the syntactic backend. It relies on another Frama-C plugin called Alias [5] which performs a may-alias analysis using a variant of the classical Steensgaard's algorithm [31]. For queries not related to pointer expressions (state and value), the top values are returned. For the other queries, since Alias does not have an API to answer our queries directly, we take the result of the syntactic backend and replace each base by a canonical representative of its alias equivalence class. In particular, for memory accesses, we ensure that a selected canonical address for an alias class is vulnerable to data races if such an address exists (e.g. because its base corresponds to a global variable or a local variable that escaped; see Section 7 for more details). Alias also does not allow starting with a user-supplied set of aliases (as expects the signature of our analyse_thread function). We therefore need to perform an additional saturation of the aliases set for all queries, as demonstrated by the example in the next paragraph. The Alias plugin is still relatively new (it lacks support, e.g., for the dynamic memory allocation) and thus our backend over it is also experimental.

▶ **Example 1.** The advantage of the Alias backend over the syntactic backend is demonstrated in Fig. 3. The program contains a data race on an address of the variable `data` that escapes its scope in the `main` function by passing it as an argument to the threads `t1` and `t2`. Both threads share the same code, which creates a local alias for this address. This is a very common pattern when thread arguments are used – since the type of the argument is generic `void*`, it needs to be casted to its actual type. The address is then written to by both threads using their local aliases. While the syntactic backend would not report any relevant access for data race detection, the Alias backend would compute aliases for `x` and `y` as $\{\texttt{x}, \texttt{arg1}\}$ and $\{\texttt{y}, \texttt{arg2}\}$, respectively. Our backend then performs a saturation of both sets by adding aliases $\{\texttt{arg1}, \&\texttt{data}\}$ and $\{\texttt{arg2}, \&\texttt{data}\}$, respectively. Finally, `&data` will be selected as a canonical base, because it is vulnerable to a data race.

5 Thread Analysis

In this section, we propose our approach on how to leverage an analyser for sequential programs to analyse programs with multiple threads. The design of our approach is based on the observation that answers to our queries described in Section 4 on objects relevant for our

Algorithm 1 Thread analysis.

Input : Program entry point main
Input : Initial values of global variables GlobalsVals ∈ Cvalue.state
Data : Initial states of threads $I : T \to$ Cvalue.state
Output : Thread-creation graph $G = (T, \to)$
Output : Results of analysis of sequential behaviour of the threads $R : T \to$ Result

1 $i = 0$
2 $G_0 = (\{\mathsf{main}\}, \emptyset)$ `// Initial graph containing main only`
3 $I_0 = \{\mathsf{main} \mapsto \mathsf{GlobalsVals}\}$ `// Initial state of the main thread`
4 $R_0 = \{\mathsf{main} \mapsto \mathsf{Backend.analyse_thread}(\mathsf{main}, \mathsf{GlobalsVals})\}$ `// Analyse main`
5 **do**
6 $\quad i = i + 1$
7 $\quad G_i = G_{i-1}$
$\quad$ `/* Update the graph with reachable threads. */`
8 $\quad$ **foreach** thread represented as a vertex p in G_{i-1} **do**
9 $\quad\quad$ **foreach** stmt s of the form $\mathsf{thread_create}(f)$ syntactically reachable by p **do**
10 $\quad\quad\quad children = \mathsf{Backend.functions}(R_{i-1}(p), (p, \varepsilon, s), f)$
11 $\quad\quad\quad G_i = G_i \cup \{p \xrightarrow{s} c \mid c \in children\}$
$\quad$ `/* Build equations wrt. the chosen approx. strategy. */`
12 $\quad \Phi_i = \mathsf{ConstructEquations}(G_i)$
$\quad$ `/* Solve the system of equations. */`
13 $\quad I_i = \mathsf{SolveEquations}(\Phi_i, I_{i-1})$
$\quad$ `/* Analyse each thread as a sequential program. */`
14 $\quad R_i = \{t \mapsto \mathsf{Backend.analyse_thread}(t, I_i(t)) \mid t \in \mathsf{dom}(I_i)\}$
15 **while** $G_{i-1} \neq G_i$
16 **return** G_i, R_i

analysis are often not too much affected by thread interleavings. We design two strategies for combining results of the analyses of the sequential behaviour of individual threads that under- and over-approximate the overall behaviour of the concurrent program consisting of the threads (not including behaviour possibly caused by errors in the code, i.e., for example undefined behaviour after a data race). Furthermore, these strategies can subsequently be combined to increase the precision of the resulting analysis by using under-approximation to find races while using over-approximation to show their absence. However, as we show in our experimental evaluation in Section 8, both of our strategies are often sufficiently precise on their own.

The core idea of our thread analysis is similar to that of the Frama-C's proprietary Mthread plugin described in [34]. Starting with the `main` thread, we iteratively build a thread-creation graph that captures the parent-child relationship among the so-far discovered threads. In each iteration, we add newly discovered threads and perform the analysis (potentially multiple times) of each thread as a sequential program (using the `analyse_thread` function of the selected backend). This is done by solving a system of equations over the initial states of the threads given by whose construction we describe below. However, before that, we first describe the graph construction algorithm.

The section is relevant mainly for our EVA backend and partly for our Alias backend. For the Syntactic backend, only the thread-creation graph construction is relevant (because no analysis of threads is performed).

5.1 Construction of the Thread-Creation Graph

The method is given in Algorithm 1. The output of the algorithm is (1) a thread-creation graph $G = (T, \to)$ where T is the set of all discovered threads and $\to$ is a labelled edge relation such that $p \xrightarrow{s} c$ means that the parent p creates a child c at the program statement s, and (2) a mapping $R : T \to \mathsf{Result}$ containing results of the analysis of the behaviour of each thread. The algorithm further operates with the mapping $I : T \to \mathsf{CValue.state}$ representing the initial states of discovered threads[1]. Once we derive a stable mapping I (i.e., a fixpoint of the thread's initial states), we have also reached a fixpoint of the results R, needed to answer all the queries issued by our analysis.

The algorithm starts with a graph that contains only the main thread whose initial state is given by the initial values of global variables. With this initial state, main is analysed on line 4.

In the main loop of the algorithm starting on line 5, the graph is updated by adding threads created from (syntactically) reachable statements. Notice that the evaluation of an expression f on line 10 is performed without using any calling context. This is because our analysis, in particular the system of equations in Sections 5.2 and 5.3, is not very sensitive to thread-create wrappers. Consequently, we can construct smaller graphs by distinguishing edges only by statements and not by calling contexts.

After a new iteration of the graph G_i is constructed, we build a system of equations, one for each discovered thread, given by G_i. Each of these equations describes how an initial state of a thread can be computed using initial states of other threads. Then, the system is solved using initial states from the previous iteration I_{i-1} to obtain new initial states I_i. The solution may need an application of widening operators – we provide more details on our implementation of the equation solving in Section 8. Based on the new initial states, we compute new results R_i on line 14.

This loop is repeated until the thread-creation graph stabilises. Overall, the algorithm runs two fixpoint computations – the outer updates the graph with new threads, while the inner (equation solving on line 13) performs analysis on the current graph, and can thus help to discover previously unknown threads in new iterations.

Below, we discuss how the aforementioned equations are constructed.

5.2 Under-Approximating Thread Analysis

We use the following under-approximated semantics of thread creation. A thread is started with the initial state corresponding to the join of initial states of its create statements, and its writes to global variables are not propagated to other threads. The creation of threads then roughly corresponds to calling its entry point in a "sandbox" that makes all of its actions invisible for other threads. Formally, given a thread-create graph G_i with edges $\to$, we compute the initial state of a thread t (other than `main` – the initial state of `main` is given by the initial states of global variables) as follows:

$$I(t) \triangleq \bigsqcup_{p \xrightarrow{s} t} \mathsf{let}\ R = \mathsf{Backend.analyse_thread}(p, I(p))\ \mathsf{in}\ \mathsf{Backend.state}(R, (p, \varepsilon, s)),$$

[1] In the implementation, the product $\mathsf{CValue.state} \times \mathsf{CValue.val}$ is used as the domain of the initial states to represent the values of global variables and the value of the thread's argument. However, for simplicity of the presentation, we may assume that the argument is simply included among the global variables.

```
1  int *g;
2  void *t1() {
3    g = malloc(...);
4  }
5  void *t2() {
6    (*g)++;
7  }
8  int main() {
9    create(t1); join(t1);
10   create(t2); create(t2);
11 }
```

(a)

```
1  int g = 1;
2  void *thread() {
3    lock(L);
4    if (g == 1) {
5      unlock(L); return NULL;}
6    g = 2;
7  }
8  int main() {
9    create(t1);
10   lock(L); g = 0; g = 1; unlock(L);
11 }
```

(b)

Figure 4 Examples of programs on which (a) the under- and (b) over-approximating thread analysis reports a false negative and a false positive, respectively.

i.e., we take all statements s creating t, and for each of them, we recompute the corresponding parent p and take its state at the statement s, returning the join of all those values. In this way, the creation of a thread does not affect its parent at all.

The motivation for this strategy is straightforward – for acyclic thread-creation graphs, it is enough to analyse each entry point only once (provided caching is used for the `analyse_thread` function). On the other hand, there are two main sources of possible imprecisions – thread interleavings are not taken into account, and parents do not see effects of their children. The latter can sometimes result in seemingly weird false negatives when performing data race detection, as demonstrated in Listing 4a. In the program, the `main` thread first creates a thread `t1` which allocates an integer value and then a thread `t2` (created twice) accesses this value without any synchronization. However, we cannot detect this data race using EVA[2] since in our under-approximated setting, `t2` will never see the allocation, and in such a situation, EVA will mark the memory access on line 6 as surely invalid and will not report any memory access for it.

5.3 Over-Approximating Thread Analysis

While for the under-approximating strategy, we assumed that no changes done by threads are propagated, in our over-approximating strategy, we assume that every thread can see all possible changes (even those that should be invisible due to some synchronization method). For this method to work correctly, we also need to perform a simple transformation of the analysed program, which we discuss, together with a motivation example, later in this section.

We compute the initial state of a thread t (here also including the `main` thread) as the join of the states our analysis has encountered so far. Formally, given a thread graph G_i with a set T of currently known threads, the initial values of global variables $V = \mathsf{GlobalsVals}$, and a set of all program statements Stmt, we proceed as follows:

$$I(t) \triangleq V \sqcup \bigsqcup_{s \in \mathsf{Stmt},\ t' \in T} \mathsf{let}\ R = \mathsf{Backend.analyse_thread}(t', I(t'))\ \mathsf{in}\ \mathsf{Backend.state}(R, (t', \varepsilon, s)),$$

i.e., we take the join of the states of all statements for each currently discovered thread.

[2] We could detect the memory access with the other backends that do not care about memory initialization.

```
1  atomic_int g;
2
3  void *t() {
4    g++;
5  }
6
7  int main() {
8    g = 0;
9    create(t);
10   if (g == 1)
11     ...
12 }
```

(a) Original program.

```
atomic_int g;

void *t() {
  g = (*) ? g : g+1;
}

int main() {
  g = (*) ? g : 0;
  create(t);
  if (g == 1)
    ...
}
```

(b) Transformed program.

Figure 5 An example of a program for which the over-approximating strategy fails to compute a sound over-approximation unless the program is transformed ($*$ denotes non-deterministic condition).

To solve such a system, more than one sequential analysis of each entry point and an application of widening is needed to converge[3] (cf. Section 8). Observe that the equations do not depend on edges of the thread-creation graph, just on its vertices.

As we also consider context switches that cannot happen due to synchronization (even those that cannot happen because a child was not yet created, e.g., when computing the initial state of the main thread and already using states of its children), we may later report false data races as demonstrated in Listing 4b. In this example, we will compute the initial state of `thread` to be $\{\mathsf{g} \mapsto \{0, 1, 2\}\}$. However, the value 0 is never visible to `thread` because the sequence of updates on line 10 (as well as the condition on line 4) is performed atomically, resulting in no change of g. Thus, the memory access on line 6, which would be flagged as a data race (together with one of the accesses on line 10), is in fact unreachable.

Although both examples in Fig. 4 are taken from publicly available benchmarks, we believe that they are quite rare in practice. Moreover, our strategies can be combined by allowing the under-approximation to only report errors and the over-approximation to only report that no error has been found. Otherwise, an unknown verdict will be reported, which would be also the case in both examples in Fig. 4.

5.3.1 Program transformation

The presented over-approximating strategy is not in itself sufficient to compute a sound over-approximation of multi-threaded behaviour as demonstrated on the program in Figure 5. In the example, we are particularly interested in the block inside the condition on line 10, which may contain some important event for our analyser. This code is unreachable when the main thread is analysed as a sequential program regardless of the initial state. Indeed, our over-approximation strategy would compute the initial state as $\{\mathsf{g} \mapsto [0, \mathsf{max_int}]\}$, but already on line 8, this state would be modified to $\{\mathsf{g} \mapsto [0, 0]\}$ and the over-approximation would be lost.

[3] We apply widening quite aggressively by performing it already after the second iteration to reach a fixpoint as soon as possible.

We solve the above problem using a simple source code transformation rather than modifying the internal state of the sequential analyser (so we do not need to make further requirements on the interface of our backends). The idea behind our solution is that it is sufficient to ensure that each program instruction is *extensive*, i.e., if S and S' are the sets of concrete states before and after the instruction, respectively, then $S \subseteq S'$. We modify all program instructions to be extensive using the following rules.

- An assignment $\mathtt{x} = \mathtt{y}$ is transformed into a non-deterministic choice between the old and the new value as $\mathtt{x} = (*)\ ?\ \mathtt{x} : \mathtt{y}$.
- A function call $\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ where at least one of the arguments x_i is passed by reference is transformed into a condition which non-deterministically skips the call as $\mathtt{if}(*)\ \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n)$.

In the rest of the paper, we implicitly assume that all backend queries for a thread t are evaluated over the results $R_i(t)$ where R_i is the value returned by Algorithm 1.

6 Lockset and Active-Threads Analyses

This section presents two helper analyses for data race detection. Both are used to check whether a pair of contexts can happen in parallel – the *lockset analysis* checks this based on locks being held in the given contexts, and the *active-thread analysis* deals with creating and joining of threads.

6.1 Lockset Analysis

Lockset analysis is traditionally used to compute must- or may-locksets being held for each statement. Our lockset analysis is context-sensitive, path-insensitive, and it computes both must- and may-locksets simultaneously. It is inspired by the approach of RacerX described in [15], which, however, does not employ any advanced alias analysis and is not context-sensitive.

We represent locks by their concrete addresses, i.e., $\mathsf{Lock} = \mathsf{Address}$, and we define $\mathsf{Lockset} = 2^{\mathsf{Lock}}$. Our lockset analysis is implemented as a forward data flow analysis with the domain of locksets, computed for each thread in separation, starting with the empty lockset. On the intraprocedural level, we start to analyse a function with an entry lockset L_{entry} coming from the caller, and gradually update it using the following transfer function that, for a context c with a statement s, transforms the lockset L_{old} reached before executing s into a set of locksets $\mathcal{L}_{new}$ resulting from the execution of s as follows:

$$\mathcal{L}_{new} = \begin{cases} \{L_{old} \cup \{\ell\} \mid \ell \in \mathsf{Backend.value_ptr}(c, \mathtt{p})\} & \text{if s is } \mathtt{lock(p)}, \\ \{L_{old} \setminus \{\ell\} \mid \ell \in \mathsf{Backend.value_ptr}(c, \mathtt{p})\} & \text{if s is } \mathtt{unlock(p)}, \\ \{L_{old}\} & \text{otherwise.} \end{cases}$$

After applying the transfer function, if $\mathcal{L}_{\text{new}}$ is not a singleton set, we fork the analysis and continue with each element of $\mathcal{L}_{\text{new}}$ separately. As a result, we obtain the summary of a function f analysed in a context c in the form $(c, L_{entry}) \mapsto \mathcal{L}_{exit}$ where $\mathcal{L}_{exit}$ is the set of locksets at the return statements of all forked analysis runs inside f. Such summaries are then used for the interprocedural analysis. We also construct *statement summaries* $\mathcal{S}$ of the same form as for functions, i.e., mappings from pairs of an input context and a lockset to output sets of locksets. These are used for caching at the statement level and to implement the must- and may-lockset queries for a context c as

$$\mathsf{may_ls}(c) = \bigcup_{(c,L)\in\mathsf{dom}(\mathcal{S})} L \qquad \mathsf{must_ls}(c) = \bigcap_{(c,L)\in\mathsf{dom}(\mathcal{S})} L$$

Dealing with path explosion. As we represent locks using memory addresses and fork the analysis whenever we cannot precisely evaluate the parameter of a (un)locking operation, the set of traversed paths can easily explode. To tackle this problem, our transfer function selects only a small number of locks when the set $\mathsf{Backend.value_ptr}(c, \mathtt{p})$ is too large (currently, the threshold is set to three locks). This works in many practical cases because it is usually enough to differentiate the may- and must-cases through a few paths only. In the future, we would like to provide support for symbolic treatment of locks combined with may- and must-equality reasoning.

Our analysis does not support reentrant mutexes/semaphores but can handle non-blocking lock functions and read-write locks as discussed below.

Non-blocking locking functions. For non-blocking lock functions such as `trylock`, which does not block when the mutex is already locked, or `timedlock`, which blocks just for a limited time, we update the lockset with relevant locks for which we also track the variable used to store the return value of the corresponding call. Evaluation of a query $\mathsf{may_ls}(c)$ or $\mathsf{must_ls}(c)$ will then ask the backend for the state s computed for the context c through the query $s = \mathsf{Backend.state}(c)$, and filter out the locks for which the state s does not guarantee that the locking was (possibly or surely, depending on the query) successful. This guarantee will only be provided if the success of the locking was explicitly checked in the context c. This way we obtain a limited path-sensitivity for branching over expressions containing variables storing return values of lock operations. For the function `pthread_mutex_lock`, we assume that it is blocking (i.e., that it never fails), but the user can easily change the configuration to treat it as non-blocking.

Read-write locks. To support read-write locks, a lock object keeps additional information about whether it was obtained in the read- or write-mode. Based on this information, we modify the intersection of locksets, presented later on as the main operation used for data race detection, so that it contains only those locks which are present in both operands and at least in one of them in the write-mode. In other words, if two accesses are guarded only by a lock obtained in both cases in the read-mode, we will treat them as unprotected.

6.2 Active-Threads Analysis

The goal of this analysis is to deal with situations when two memory accesses cannot happen in parallel either because one happens before the thread of the other one is created, or because one happens after the thread of the other one is surely joined. We again compute this information in a thread-modular way by computing a *set of sets of threads* that may or must be active in parallel for each context, respectively. Two contexts are then may-parallel (must-parallel) if the thread of one of them is in the union (intersection) of sets computed for the other, and vice versa.

The sets of active threads are computed in a very similar way to how the lockset analysis is performed – just working with sets of threads instead of sets of locks, with the transfer functions for thread creation and joining analogous to those for locking/unlocking, and with the same way of handling the may- and must-queries. The set of thread entry points is usually small, and we thus do not need any special treatment to tackle path explosion unlike in the lockset analysis. One difference between the analyses is in the initial states. The lockset analysis of each thread is started with the empty lockset, but the active-threads analysis of a thread t starts with the union of all threads that may be active together with t

in some context of the parent of t (the analysis is thus run in the order given by a topological order on thread-creation relation, and for the very rare case of cyclic thread-creation graphs, we simply skip the analysis, providing trivial answers for queries based on it).

Thread identifiers. Another significant difference is the way in which threads are joined. While our analysis abstracts threads using their entry points, in the concrete semantics, they are manipulated using their *identifiers*. In many programs, however, the manipulation of identifiers is straightforward, and one can easily map identifiers to abstract threads and vice versa. Based on this observation, we assume that if an identifier expression of a created and joined thread is the same, then the thread is also the same. We have never encountered a situation when this would lead to a wrong analysis verdict.

7 Data Race Detection

Our data race detection does a forward traversal of the program and collects memory accesses consisting of a context (used for querying the results of lockset and active-thread analyses), a base address, and a symbolic offset (a set of intervals in the Cvalue domain that may be some simple value for backends different from EVA). We cluster memory accesses according to their bases to later check for data races only inside each of those clusters. This is justified by the following hypothesis about the memory model: it is not possible to get from one base to another by adding an offset. This hypothesis might not hold for some low-level C programs, but it is important for efficiency. Moreover, analysis of programs breaking this assumption is not even supported by EVA [8].

The tracking of accesses is inspired by [30]. For each cluster represented by its base, we track a state $s \in \{\mathsf{ReadOnly}\ o, \mathsf{Exclusive}\ o, \mathsf{Shared}, \mathsf{SharedMod}\}$ representing that a memory base is either read-only (ReadOnly) or modified (Exclusive) by a single thread o, read by several threads with none of them writing to it (Shared), or modified by distinct threads such that at least one of them writes to it (SharedMod). With each access to a memory base b by a thread t, the state of b is updated according to the transition graph in Fig. 6. When the analysis is finished, we keep only the memory bases marked as SharedMod or Exclusive o (when o is a thread created multiple times), and for each of them, we check each pair of accesses for a data race. The information about which threads are unique and which are created multiple times is computed during the thread analysis based on whether its create statement was visited once or multiple times.

Escape analysis. An important part of data race detection is discovering local bases that may escape their scope. Our syntactic backend assumes that dynamically allocated bases can always escape and static bases escape when their address is taken. Using the Alias and EVA backends, we enhance this by testing (for both static and dynamic bases) not only whether their addresses were taken, but also whether they were assigned to global variables.

May- and must-data-races. We distinguish two types of data races based on the confidence we have in them. For *may-data-races*, the below conditions need to be satisfied:

1. At least one of the accesses is a write access.
2. The accessing threads are either distinct or the same non-unique thread.
3. The offsets of the addresses may overlap (checked by a query over the CValue domain of intervals).

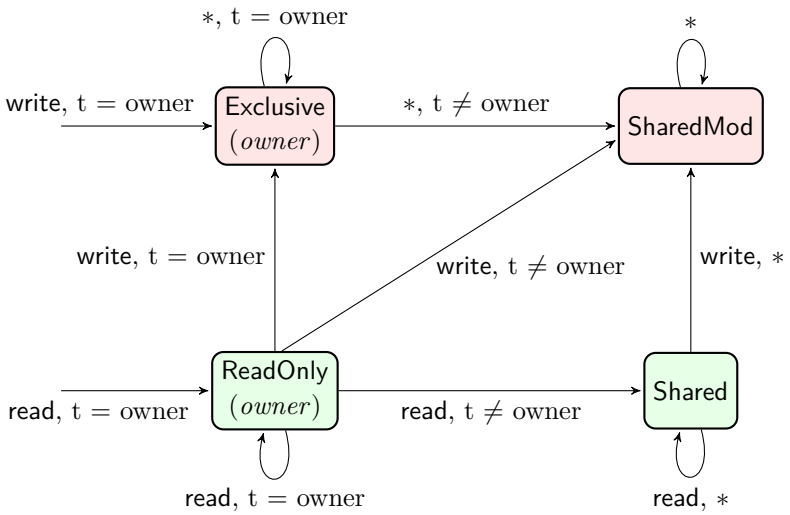


Figure 6 The transition graph for tracking states of memory bases. Each transition corresponds to a read or write memory access by a thread t. The symbol $*$ represents any access kind/thread. The states vulnerable to data races are marked in red.

4. The intersection of the may-locksets of the concerned contexts is empty (checked using the results of the lockset analysis).
5. The threads of both accesses may run in parallel (checked using the results of the active-threads analysis).

For *must-data-races*, we require the *must*-variants of Conditions 3–5 to hold. In addition, the following conditions are required to rule out races involving a memory access representing abstraction of multiple memory accesses:

6. The memory base is not *weak* (i.e., if it is dynamic, then it represents precisely one dynamic allocation) and it is not allocated in a non-unique thread.
7. The size of the access does not exceed the size of the type of the base.
8. The base is the only base being read/written to in its context.

Condition 6 ensures that we do not report must-races on bases obtained by abstracting and hence merging different calls to allocation functions. Condition 7 rules out races on possibly different indices to arrays (an access with an offset bigger than the size of the type of the base usually abstracts multiple accesses). Finally, Condition 8 handles situations when our context-sensitivity level is not sufficient to distinguish accesses to multiple bases.

Arrays and loop unfolding. To slightly relax the restrictiveness of Condition 7, we can perform a small number of unfoldings of each loop. This can help us report must data races on the accesses in unfolded cases. The loop unfolding is also useful for dealing with the creation of threads in loops – since our analyses are path-insensitive, they will also consider the infeasible path on which the cycle is never entered, and no thread is created. For memory accesses following the loop, the must variant of Condition 5 will thus not hold. Unfolding such loops at least once will fix this problem.

8 Implementation and Experimental Evaluation

The RacerF[4] analyser based on the above presented principles is implemented as a plugin of the Frama-C platform as a part of our bigger project for detection of concurrency bugs. The individual analyses are written as OCaml functors parametrised by a module implementing the backend analysis. We also provide a Python wrapper which runs a combination of under- and over-approximating analyses and also performs preprocessing (currently, the preprocessing is implemented outside of our analyser for technical reasons, thus the analyser is slowed a bit by the need of repeated parsing). To solve the systems of equations from Sections 5.2 and 5.3 we use the *chaotic iteration strategy with widenings* [7] implemented in the Ocamlgraph library [9]. The widening operators are those provided by the CValue domain.

Our implementation provides support for atomic and thread-local variables. The threading and locking functions (and their relevant properties) as well as atomic functions (such as gcc's `__atomic_store`) may be defined using configuration files.

We have conducted a series of experiments to support the following two hypotheses:

- Despite having no formal guarantees, RacerF can efficiently analyse intricate programs without sacrificing a precision. We validated this hypothesis on a collection of benchmarks from SV-COMP containing 1029 programs, many of them being intricate examples and corner cases contributed by participants to showcase capabilities of their tools.
- RacerF can detect data races in real-world programs for which other existing static analysers do not provide precise results (either because of too coarse approximation or running out of resources). We validated this hypothesis on a set of student programs from [18].

Experimental setup. Some data presented in experiments were taken from the results of SV-COMP 2025 [3]. All other experiments were run on a machine with 2.5 GHz Intel Core i5-7300HQ CPU and 16 GiB RAM, running Ubuntu 22.04 in a virtual machine limited to 10.7 GiB RAM. Our experiments were conducted using BenchExec [2], a framework for reliable benchmarking.

8.1 SV-COMP Benchmarks

The SV-COMP sub-category *NoDataRace-Main* is, to our knowledge, the largest publicly available data race benchmark. Many of its programs are small but intricate tests provided by the participants. Our main goal is to demonstrate the scalability of our approach and to also show that its heuristic nature does not compromise its precision even on those intricate examples. The benchmark also contains implementations of lock-free algorithms, which are out of the scope of our tool. We have implemented a single heuristic that detects active waiting (essentially a loop with an empty body) as a characteristic pattern for lock-free algorithms, and when it is present, returns an unknown result.

We compare with the five best-performing tools participating in SV-COMP 2025 [21, 22, 23, 29] (including $\text{RacerF}_{\text{sv-comp}}$, an older version of RacerF) and we also include $\text{RacerF}_{\text{under}}$ and $\text{RacerF}_{\text{over}}$ representing RacerF running with under- and over-approximating thread analysis only, respectively. We wanted to include also the O2 checker [26], but since its results were not good and all the other tools were configured specifically for this benchmark by their authors, we do not consider such a comparison fair.

[4] RacerF is available under the MIT license at `https://github.com/TDacik/Deadlock_and_Racer`.

Table 1 Results for the SV-COMP sub-category *NoDataRace-Main*. The columns represent true negatives, false positives, true positives, false negatives, out-of-resource (time or memory), score (as defined by SV-COMP rules, but excluding witness validation), and times (excluding timeouts) on local and competition machine, respectively.

	No race (794)		Races (235)					
Analyser	TN	FP	TP	FN	RO	Score	Time$_1$ [s]	Time$_2$ [s]
RacerF	674	4	105	0	0	1389	4620	–
RacerF$_{\text{over}}$	674	6	83	0	0	1335	3970	–
RacerF$_{\text{under}}$	680	4	105	6	0	1209	2710	–
RacerF$_{\text{sv-comp}}$	674	4	98	0	0	1382	3100	1300
Goblint	712	0	0	0	3	1424	–	2320
Deagle	685	0	203	1	43	1541	–	10k
UGemCutter	578	0	161	0	263	1317	–	152k
UAutomizer	609	0	121	0	274	1339	–	120k

The results are summarised in Table 1. Unlike the competition results [3], we present the results ignoring the witness validation. RacerF produced 4 false alarms, all of them on programs where a data race is ruled out by some intricate condition. Two of them are variants of a program where a data race does not manifest because an array being searched in parallel contains only zeros and a certain condition is thus never triggered which makes the potential racy access unreachable (which is the kind of behaviour that our tool cannot handle). The other two contain locking patterns that are beyond the scope of our lockset analysis.

We first focus on a comparison of the versions of RacerF. The difference between its newest version and the version competing in SV-COMP'25 lies mainly in improved escape analysis and preprocessing (loop unrolling in the under-approximating mode and assignment modification in the over-approximating mode). The changes show a positive impact on more reported data races but also a slowdown which is a consequence of preprocessing done in over-approximating mode. The slowdown mostly manifests for the active-thread analysis, which could be a performance bug in the implementation.

Interestingly, running RacerF in either the over- or under-approximating mode does not compromise its precision too much, leading to an additional 2 false positives (and 22 must-races classified just as may- and thus not reported) and 6 false negatives, respectively.

Compared to participants of SV-COMP'25, RacerF placed second in the competition ranking and third in our evaluation. Compared to Goblint, which never reports races, RacerF can produce correct results for more programs. Compared to model checker Deagle, which benefits from the fact that many programs are small and their loops can be fully unrolled, no version of RacerF ever runs out of time or memory.

Where RacerF shines is the ldv–linux–3.14–races benchmark subset containing 5 programs derived from the Linux kernel (all of them are race-free). None of the participants with a positive score can provide correct verdicts for those programs (except Goblint which handles one of them), but RacerF can correctly analyse each of them under two minutes.

8.2 Student Programs

To test our tool on more non-crafted programs, we performed an evaluation on a set of 116 student programs from [18]. The programs are rather short (around 100 – 300 LoC) and implement the same heavily concurrent algorithm parametric in the number of threads, but

Table 2 Results for student programs. The columns are true positives, false negatives, no race reported, race reported, out-of-resources, and time (excluding timeouts).

	Races (30)		Others (86)			
Analyser	TP	FN	NR	RR	OOR	Time [s]
RacerF	16	3	46	0	0	688
O2	7	23	86	0	0	54
Goblint$_{sv}$	0	3	46	0	0	41
Goblint	27	3	46	40	0	41
Deagle	3	1	6	0	78	1644

they differ in various intricate bugs (for their detailed discussion see [18]). We have identified 29 data races using dynamic analysers. However, there may be more that those tools missed. One additional race was discovered by all RacerF, O2, and Goblint. The issue could manifest only in a very rare situation when two threads simultaneously failed and called a clean-up function that assigns a global pointer without synchronization. The time limit was set to 60 seconds and the memory limit to 1GB.

We have compared with Goblint[5] and Deagle (version from SV-COMP'25) that performed better than us in SV-COMP, and with O2 (implemented in Coderrect[6]) that we selected as a recent representative of lightweight and scalable race detection tools.

The results are given in Table 2. As expected, since all the programs contain unbounded loops and also an unbounded number of threads is created, Deagle ran out of the resources on almost all programs and provided an answer in only 10 cases. Interestingly, Goblint and RacerF claimed *the same* 46 programs as race-free (but RacerF without providing any false positives). The three missed races by RacerF and Goblint are also the same and are caused by an undefined behaviour related to re-initialisation and destroying of mutexes that still may be used by some threads (neither of the tools supports detection of such problems). A detailed inspection suggests that both tools struggle with programs that use an array where each thread accesses only the element at its dedicated index.

Overall, RacerF achieves excellent precision on both sides. While O2 misses many races and Goblint reports many false alarms, RacerF is able to discover more than half of the races while not reporting any false alarms.

9 Conclusions and Future Work

We have presented a novel data race detector RacerF and performed a series of experiments indicating that RacerF achieves a nice compromise between precision and scalability both on crafted benchmarks and real-world programs. In the future, we would like to target other classes of concurrency bugs. We already have an implementation for deadlock detection based on top of our lockset analysis, which is not discussed in this paper. Another interesting direction is combination with dynamic approaches, e.g., guiding insertion of noise injection by results of static analysis.

[5] We use Goblint in the version from SV-COMP'25 but with the default configuration as the other available configurations did not improve the results and led to timeouts. Further, to provide more detailed results, we run Goblint via its SV-COMP wrapper that never reports detected races (Goblint$_{sv}$) as well as without it.

[6] The link `https://coderrect.com` from which we got version 1.1.3 is no longer available.

References

1 Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The Dogged Pursuit of Bug-Free C Programs: The Frama-C Software Analysis Platform. *Commun. ACM*, 64(8):56–68, 2021. doi:10.1145/3470569.

2 Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable Benchmarking: Requirements and Solutions. *International Journal on Software Tools for Technology Transfer*, 21(1):1–29, 2019. doi:10.1007/s10009-017-0469-y.

3 Dirk Beyer and Jan Strejček. Improvements in Software Verification and Witness Validation: SV-COMP 2025. In *Proc. of TACAS'25*, LNCS, 2025.

4 Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. RacerD: Compositional static race detection. *Proc. of ACM Program. Lang.*, 2(OOPSLA), 2018. doi:10.1145/3276514.

5 Allan Blanchard, Loïc Correnson, Tristan Le Gall, Jan Rochel, and Julien Signoles. Alias, Efficient May-alias and Points-to Analysis. https://www.frama-c.com/fc-plugins/alias.html.

6 Sandrine Blazy, David Bühler, and Boris Yakobowski. Structuring Abstract Interpreters Through State and Value Abstractions. In *Proc. of VMCAI*, pages 112–130, 2017. doi:10.1007/978-3-319-52234-0_7.

7 François Bourdoncle. Efficient Chaotic Iteration Strategies with Widenings. In *Formal Methods in Programming and Their Applications*, 1993.

8 David Bühler, Pascal Cuoq, and Boris Yakobowski. *The Eva Plug-in (For Frama-C 30.0)*. Available from https://frama-c.com/download/frama-c-eva-manual.pdf.

9 Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Designing a Generic Graph Library using ML Functors. In *Symposium on Trends in Functional Programming*, 2007.

10 Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A Software Analysis Perspective. In *Proc. of SEFM*, pages 233–247, 2012. doi:10.1007/978-3-642-33826-7_16.

11 Tomáš Dacík and Tomáš Vojnar. TDacik/Deadlock_and_Racer. Software, swhId: swh:1:dir:fc33aed78b227ea0738a89894b3dfa91b4f46844 (visited on 2025-06-18). URL: https://github.com/TDacik/Deadlock_and_Racer, doi:10.4230/artifacts.23614.

12 Daniel Dietsch, Matthias Heizmann, Dominik Klumpp, Mehdi Naouar, Andreas Podelski, and Claus Schätzle. Verification of Concurrent Programs Using Petri Net Unfoldings. In *Proc. of VMCAI*, pages 174–195, 2021. doi:10.1007/978-3-030-67067-2_9.

13 Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for Testing Multi-threaded Java Programs. *Concurrency and Computation: Practice and Experience*, 15(3-5), 2003. doi:10.1002/cpe.654.

14 Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proc. of PLDI'07*. ACM, 2007. doi:10.1145/1273442.1250762.

15 Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. *SIGOPS Oper. Syst. Rev.*, 37(5):237–252, 2003. doi:10.1145/1165389.945468.

16 Hongyu Fan, Zhihang Sun, and Fei He. Satisfiability Modulo Ordering Consistency Theory for SC, TSO, and PSO Memory Models. *ACM Trans. Program. Lang. Syst.*, 45(1), 2023. doi:10.1145/3579835.

17 J. Fiedor, V. Hrubá, B. Křena, Z. Letko, S. Ur, and T. Vojnar. Advances in Noise-Based Testing of Concurrent Software. *Software Testing Verification and Reliability*, 25(3):272–309, 2015. doi:10.1002/stvr.1546.

18 Jan Fiedor and Tomas Vojnar. Noise-based Testing and Analysis of Multi-Threaded C/C++ Programs on the Binary Level. *2012 10th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, PADTAD 2012 - Proceedings*, 2012. doi:10.1145/2338967.233681.

19 C. Flanagan and S. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proc. of PLDI'09*. ACM, 2009. doi:10.1145/1543135.1542490.

20 Natalia Gavrilenko, Hernán Ponce-de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings. In *Proc. of CAV*, pages 355–365, 2019. doi:10.1007/978-3-030-25540-4_19.

21 Fei He, Zhihang Sun, and Hongyu Fan. Deagle: An SMT-based Verifier for Multi-threaded Programs (Competition Contribution). In *Proc. of TACAS'22*, pages 424–428, 2022. doi:10.1007/978-3-030-99527-0_25.

22 M. Heizmann, M. Bentele, D. Dietsch, X. Jiang, D. Klumpp, F. Schüssele, and A. Podelski. Ultimate Automizer and the Abstraction of Bitwise Operations (Competition Contribution). In *Proc. of TACAS'24*, LNCS 14572, pages 418–423, 2024. doi:10.1007/978-3-031-57256-2_31.

23 Dominik Klumpp, Daniel Dietsch, Matthias Heizmann, Frank Schüssele, Marcel Ebbinghaus, Azadeh Farzan, and Andreas Podelski. Ultimate GemCutter and the Axes of Generalization. In *Proc. TACAS'22*, pages 479–483, 2022. doi:10.1007/978-3-030-99527-0_35.

24 Daniel Kästner, Antoine Miné, Laurent Mauborgne, X. Rival, Jérôme Feret, Patrick Cousot, Stephan Wilhelm, and Christian Ferdiand. Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. In *In ERTS'16*, 2016.

25 A. Lal and T. Reps. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. In *Proc. of CAV'08*, pages 37–51, 2008. doi:10.1007/978-3-540-70545-1_7.

26 B. Liu, P. Liu, Y. Li, C.-C. Tsai, D. Da Silva, and J. Huang. When Threads Meet Events: Efficient and Precise Static Race Detection with Origins. In *Proc. of PLDI'21*. ACM, 2021. doi:10.1145/3453483.345407.

27 Antoine Miné. Relational Thread-Modular Static Value Analysis by Abstract Interpretation. In *Proc. of VMCAI*, pages 39–58, 2014. doi:10.1007/978-3-642-54013-4_3.

28 T. Nguyen, B. Fischer, S. Torre, and G. Parlato. Lazy sequentialization for the safety verification of unbounded concurrent programs. In *Proc. of ATVA'16*, volume 9938 of *LNCS*, 2016. doi:10.1007/978-3-319-46520-3_12.

29 S. Saan, J. Erhard, M. Schwarz, S. Bozhilov, K. Holter, S. Tilscher, V. Vojdani, and H. Seidl. Goblint: Abstract Interpretation for Memory Safety and Termination (Competition Contribution). In *Proc. of TACAS'24*, LNCS 14572, pages 381–386, 2024. doi:10.1007/978-3-031-57256-2_25.

30 Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997. doi:10.1145/265924.265927.

31 Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *Proc. of POPL'96*, pages 32–41, 1996. doi:10.1145/237721.237727.

32 Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static Race Detection for Device Drivers: the Goblint Approach. In *Proc. of ASE*, pages 391–402, 2016. doi:10.1145/2970276.2970337.

33 J. Wu, Y. Tang, H. Hu, H. Cui, and J. Yang. Sound and Precise Analysis of Parallel Programs through Schedule Specialization. In *Proc. of PLDI'12*. ACM, 2012. doi:10.1145/2345156.2254090.

34 Boris Yakobowski and Richard Bonichon. *The Mthread plugin*. Available from `https://frama-c.com/download/frama-c-mthread-manual.pdf`.

Type-Safe and Portable Support for Packed Data

Arthur Jamet
University of Kent, Canterbury, UK

Michael Vollmer
University of Kent, Canterbury, UK

Abstract

When components of a system exchange data, they need to serialise the data so that it can be sent over the network. Then, the recipient has to deserialise the data in order to be able to process it. These steps take time and have an impact on the overall system's performance.

A solution to this is to use **packed data**, which has a unified representation between the memory and the network, removing the need for any marshalling steps. Additionally, using this data representation can improve the program's performance thanks to the data locality enabled by the compact representation of the data in memory. Unfortunately, no mainstream programming languages support packed data, whether it's out-of-the-box or through a compiler extension.

We present `packed-data`, a Haskell library that allows for type safe building and reading of packed data in a functional style. The library does not rely on compiler modifications, making it portable, and leverages meta-programming to allow programmers to pack their own data types effortlessly.

We evaluate the usability and performance of the library, and conclude that it allows traversing packed data up to 60% faster than *unpacked* data. We also reflect on how to enhance the performance of library-based support for packed data.

Our implementation approach is general and can easily be used with any programming languages that support higher-kinded types.

2012 ACM Subject Classification Information systems → Data layout

Keywords and phrases program optimisation, data structures, data layout, packed data

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.38

Category Experience Paper

Supplementary Material *Software (Source Code)*: `https://github.com/Arthi-chaud/packed-data` [8], archived at `swh:1:dir:027d47580c14b18a2969e17a715e86c2094246ad`

1 Introduction

Components within a system may need to exchange data. Consequently, data needs to be sent in a format that is processable by the recipient. For example, web application programming interfaces (web APIs) usually serialise data using the Java-Script Object Notation (JSON) before sending it over the network. On the other hand, the receiver of the data has to deserialise the response into a processable object; only then will it be able to use the data. For example, a web client has to parse the JSON received before being able to, say, compute the average length of the usernames in a list.

The most common serialisation formats used for these kinds of interactions are JSON, XML and HTML. They all have a simple, human-readable syntax, and it is very easy to serialise and deserialise data with them. However, because of their syntax, these formats make the data considerably larger, leading to a higher memory footprint and longer transfer times. Additionally, de/serialisation has a cost and can impact the performance of large exchange-heavy systems.

© Arthur Jamet and Michael Vollmer;
licensed under Creative Commons License CC-BY 4.0

Alternatively, such components could use a binary format to exchange data. For example, one could use Google's Protocol Buffers (Protobuf)[1], which generates code (for many languages) to serialise/deserialise data to/from a binary format. This leads to smaller payloads (compared to human-readable formats), and thus smaller transfer times. On the other hand, Cap'n Proto[2] allows using binary data as-is, without any marshalling steps, but its encoding leads to bigger buffers compared to Protobuf's.

Packed data is binary data that can be used as-is, safely and without any deserialisation step. Thus, it can be easily persisted to and read from the disk without any conversion cost. Additionally, the minimal layout information contained in the packed data leads to a smaller overhead compared to Cap'n Proto. Moreover, unlike objects in memory, packed data does not have pointers and nested fields are inlined within the parent structure. Consequently, the data locality enabled by the packed format can leverage better performance (e.g. using the L1 cache). This can help speed up programs that traverse large data trees, like compilers with abstract syntax trees (AST) or web browsers with document object models (DOM).

The current state-of-the art is Gibbon, a compiler that transforms programs that use pointer-based structures into programs that use this packed representation [14, 13, 9, 11]. Gibbon supports a small subset of the Haskell programming language, and compiles recursive functional programs operating on trees into C code operating on packed bytestrings. For example, Figure 1b shows the memory layout of a binary tree (defined in Figure 1a) for a C program. Gibbon's packed representation of that same tree (defined in Figure 1c) is given in Figure 1d: instead of containing pointers, the parent node contains its children directly, contiguously in memory.

Writing C code for traversing packed data by hand is not trivial, and requires raw and untyped pointer manipulations, leading to code that is hard to read and maintain. Figure 2 gives an example of such code, which computes the sum of the leaves in a packed tree. Using a specialised compiler like Gibbon can alleviate this concern by generating the tricky code automatically (and using a typed intermediate language that verifies packed data layout invariants [13]), but introducing a dependency on an experimental research compiler is not realistic for some software projects. Additionally, Gibbon supports only a small subset of Haskell, lacking many convenient features, and also requires whole-program compilation, making integration into larger programs awkward.

This data layout has another a drawback: it does not allow accessing members of a data structure easily. Since fields are inlined and can have variable sizes (e.g. for recursive data types), they do not have a predefined position (or offset) within the parent structure. For example, consider a tree similar to the one from Figure 1, but where the root node's children are swapped. We know that the leaf is *after* the node, but we do not know the node's size or where it ends. The only way to access the leaf would be to traverse the first child. Depending on the size of that tree, it can lead to catastrophic performance issues. One solution to this problem is to use *indirections*. In Gibbon, indirections are numbers inserted (when requested) in the packed buffer before each field of a data structure. These numbers represent the size in bytes of the field they precede [13]. Indirections make the packed data larger, but they help avoid useless traversals of (potentially big) preceding fields before getting to the target one.

While the benefits of packed data have been proven, Gibbon is one of the only compilers that use this data layout natively. But Gibbon is not the only tool to leverage a condensed data layout: we already mentioned high-level libraries like Cap'n Proto, which allows using packed

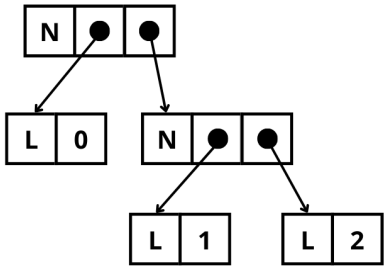
[1] `https://protobuf.dev`
[2] `https://capnproto.org`

```
enum Tag { LEAF, NODE };

struct Tree {
  enum Tag tag;
  union {
    // Content of a leaf
    struct {
      long value;
    };

    // Content of a node
    struct {
      struct Tree *left;
      struct Tree *right;
    };
  };
};
```

(a) Definition of the tree, in C.



(b) Pointer-based representation.

```
data Tree = Leaf Int
          | Node Tree Tree
```

(c) Definition of the tree in Gibbon/Haskell.

(d) Packed representation.

Figure 1 Pointer-based and packed representation of a data tree.

data as-is, and Protobuf, to pack/unpack data. On the compiler-level, the Glasgow Haskell Compiler (GHC) also supports packing data, through *compact normal forms* (CNF) [15], but we can't manipulate it (e.g. access a field) in this form. Packed data remains to be leveraged by mainstream programming languages. The performance improvements enabled by this data format remain limited to experimental research projects.

In this paper, we present a new effort of supporting packed data through a library. `packed-data` is a Haskell library for building and traversing packed data. Inspired by the example packed data interface from Bernardy et al.'s paper [4], the library contains a code generator that allows the programmer to easily use their own algebraic data types (ADT). We use Haskell in this paper, but our approach is not specific to the Haskell programming language. Thus, the library does not require any compiler modifications and can be implemented in any language with higher-kinded types. We present the design of this library along with a use-case for packed data (Section 2). Then, in Section 3, we evaluate the usability and performance of our library-based approach. Finally, we reflect on the limitations of our design in Section 4 and suggest improvements in Section 5.

The source code of `packed-data` is available on Hackage[3]. The package also includes the code of the benchmarks we present in Section 3.2.

Unless specified, all code snippets in this paper use the Haskell syntax.

[3] `https://hackage.haskell.org/package/packed-data`

```
long sum(char **t) {
  long res = 0;
  if (**t == LEAF) {
    *t = *t + 1;
    res = *((long *)*t);
    *t = *t + sizeof(long);
  } else if (**t == NODE) {
    *t = *t + 1;
    res = sum(t);
    res = res + sum(t);
  }
  return res;
}
```

Figure 2 Computing the sum of the values in a packed tree, in C.

2 The `packed-data` Library

In this section, we will go through and explain the features of the `packed-data` library. Then, we will give an example of a writing operation on packed data, and finally we will cover a use-case of packed data, implemented with the library.

2.1 Features

As its name suggests, `packed-data` allows creating and manipulating packed data. Since the library is implemented in Haskell, we leverage the language's type-system to guarantee that packed data manipulation is type-safe. Thus, any bad use of a packed buffer will be detected and rejected at compile-time. Additionally, we hide pointer arithmetic details behind a functional API. This allows programmers to use packed data like they would use data streams, instead of a raw and untyped memory buffer. The library also includes a code generator which generates all the necessary functions to pack, traverse and unserialize custom ADTs. Finally, `packed-data` allows programmers to choose whether indirections are inserted in the packed data.

2.1.1 Packing Data

Packing data can be done in two ways: by using an intermediate typed buffer, or by serializing an object that already exists.

Consider a `Tree` ADT isomorphic to the one defined in Figure 1c. The code sample in Figure 3a uses the `Needs` intermediary buffer and functions like `startNode` and `writeLeaf` (generated automatically for the `Tree` type, see Section 2.1.3). Alternatively, the code snippet in Figure 3b shows how to *pack* a tree using the `pack` function. The buffers produced by these two approaches are identical.

The `Needs` type represents intermediary buffers that *needs* to be written to, before one can use them as packed data. Its definition in Figure 4 is inspired by Bernardy et al.'s examples [4]. `Needs` has two types parameters: the first one is the list (denoted `'[...]`) of the types of the values that the buffer expects before being considered *ready*, and the second type argument is the types of the values packed in the buffer once reified. These parameters allow for type-safety when writing data and producing the final buffer. By using lists as type parameters, we can represent multiple values packed next to each other in a single buffer.

```
myPackedTree :: Packed '[Tree]
myPackedTree = finish needs
    where needs = withEmptyNeeds
        $
        do
            startNode
            writeLeaf 1
            startNode
            writeLeaf 2
            writeLeaf 3
```

(a) Using the "start" and "write" functions.

```
-- Defined in packed-data
pack :: Packable a =>
    a -> Packed '[a]
pack = ...

-- Generated automatically
instance Packable Tree where
    ...

myTree :: Tree
myTree = Node
        (Leaf 1)
        (Node (Leaf 2) (Leaf 3))

myPackedTree :: Packed '[Tree]
myPackedTree = pack myTree
```

(b) Using the "pack" function.

Figure 3 How to pack a value with `packed-data`.

For example, `Needs '[Int] '[Tree]` represents a buffer that *needs* a single integer, before it can be reified as a packed `Tree`. `Needs '[] '[Char, Int]` represents a buffer that is ready to be reified; it would contain a packed character, followed by a packed integer.

When a `Needs` is *ready* or *full* (i.e. its first type argument is an empty list `'[]`), we can produce the final packed buffer with `finish :: Needs '[] t -> Packed t`.

Finalised buffers take the form of a `Packed` value. This data type, defined in Figure 4 has a single type parameter which tells the type of values packed inside. For example, `Packed '[Tree, Tree]` is a buffer that contains two packed `Trees`.

```
newtype Needs p t = Needs ByteString

newtype Packed p = Packed ByteString
```

Figure 4 Definition of the "Needs" and "Packed" data type.

2.1.2 Traversing Buffers

We need a way to traverse packed data. To do so, we defined `PackedReader`, a computation modelling a cursor, *reading* from left to right. `PackedReader` is parameterised by the type of the `Packed` data to read, the type of the data after the cursor has been shifted, and the type of the value produced by the computation. For example, in Figure 5a, we define a function that computes the sum of two integers packed in the same buffer.

To illustrate the security that type-safety provides, consider the code snippet 5b, where we try to read a `Char` where the packed value is actually an integer. This code does not type-check, and an error will be raised at compile-time. This showcases how type-safety prevents programmers from reading values incorrectly and bad shifts of the `PackedReader`'s cursor.

`PackedReader` is actually an *indexed* monad, where the type parameters represents the state of the computation [2], i.e. the expected types of the data to traverse, and the types of the packed data after the cursor has been moved.

```
readInt :: PackedReader
    '[Int] r Int

sumInts :: PackedReader
    '[Int Int] '[] Int
sumInts = do
    i1 <- readInt
    i2 <- readInt
    return (i1 + i2)
```

(a) Reading Packed integers.

```
readChar :: PackedReader
    '[Char] r Char

example :: PackedReader
    '[Int, Int] '[] Int
example = do
    i1 <- readInt
    c <- readChar -- Type error
    return i1
```

(b) Invalid reading of packed values.

```
getSum :: Packed '[Int, Int] -> Int
getSum packed = runReader sumInts packed
```

(c) Execution of a PackedReader.

Figure 5 Reading packed values with `PackedReader`.

This allows us to leverage Haskell's type-system to ensure type-safe data manipulation, and the language's native syntax (e.g. `do`-notation) to give the code an imperative feel. To actually execute a `PackedReader` computation, we use the `runReader` function. An example is given in Figure 5c.

The code generator described in Section 2.1.3 provides a "case" function: a `PackedReader` which simulates pattern matching. Figure 6 is an example of a packed tree traversal, using the generated `caseTree` (defined in Figure 8b). This function computes the sum of the leaf values in a packed tree. The first lambda to pass (lines 4-7) is the function to execute if the packed tree is a `Leaf`, and the second one (lines 8-12) is for when the packed tree is a `Node`.

```
1  sumTree :: PackedReader '[Tree] '[] Int
2  sumTree =
3      caseTree
4          ( do
5              n <- reader
6              return n
7          )
8          ( do
9              left <- sumTree
10             right <- sumTree
11             return (left + right)
12         )
```

Figure 6 Computing the sum of the values in a Tree, using "caseTree".

2.1.3 Code Generator

Defining how to encode and decode packed data is a critical aspect of packed data manipulation. Since we deal with bytes, it is very easy to make mistakes, and tools like the type-system cannot be helpful here. `packed-data` takes away the programmer's responsibility to write encoding/decoding functions for their custom data types by providing a code generator.

Powered by Template Haskell [10], this code generator defines the following for a given ADT:

- An instance of the `Packable` typeclass, providing the `pack` function (seen in Figure 3b)
- An instance of the `Unpackable` typeclass, providing the `reader` function, to unpack a value from a packed stream.
- A "case" function, to simulate pattern matching on a packed object (see Section 2.1.2)
- A "transform" function, similar to "case"; it helps generate packed data out of `Packed` values (an example of its usage is given in Section 2.2)

The code generator's entry point also takes as parameter a list of flags, which change its behaviour. Currently, these flags only deal with indirections (described in Section 2.1.4).

To invoke the code generator, the programmer has to call the `mkPacked` function. An example is given in Figure 7.

```
$(mkPacked ''Tree [InsertFieldSize])
```

Figure 7 Invocation of the code generator.

2.1.4 Indirections

The code generator accepts two flags:

- `InsertFieldSize`: Inserts indirections before each field of the packed ADT.
- `SkipLastFieldSize`: Does not insert an indirection before the last field. We do not always need to jump over a final field, and not adding an indirection allows saving space in memory.

These flags will change the definition of the functions produced by the code generator. Figure 8 illustrates these changes. Notice in Figure 8a that each field is preceded by a `FieldSize` (a type alias to 32-bit integers we use as indirections), absent from the definition in Figure 8b, generated without passing indirection flags. This type transparency regarding indirections makes it easier to think about how to traverse a data structure: should we traverse everything to get to the nth field, or can we use indirections to jump there?

```
caseTree ::
    PackedReader '[FieldSize, Int] r a ->
    PackedReader '[FieldSize, Tree, FieldSize, Tree] r a ->
    PackedReader '[Tree] r a
```

(a) Definition of the "case" function for Trees, when the "InsertFieldSize" is on.

```
caseTree ::
    PackedReader '[Int] r a ->
    PackedReader '[Tree, Tree] r a ->
    PackedReader '[Tree] r a
```

(b) Definition of the "case" function for Trees, when no indirection flags are passed to the generator.

Figure 8 Changes on the API caused by indirections.

Consequently, code that manipulates packed data changes accordingly to the indirections flags given to the code generator. For example, consider Figure 9 where we traverse a binary tree to get its right-most value. In Figure 9a, we have to *read* each left sub-trees to get to the right one. This is not optimal, as it causes a complete traversal of that sub-tree. However, in Figure 9b, we *skip* over a field using the `FieldSize` that precedes it. One drawback is that we have to explicitly skip `FieldSize`s we do not care about (before an integer value for example).

```
getRightMost :: PackedReader '[Tree] r Int
getRightMost =
    caseTree
        readInt
        ( do
            _ <- readTree
            getRightMost
        )
```

(a) Traversing a tree, without indirections.

```
getRightMost :: PackedReader '[Tree] r Int
getRightMost =
    caseTree
        ( do
            skipFieldSize -- Skipping leading indirection
            readInt
        )
        ( do
            skipWithFieldSize -- Skipping the left subtree
            skipFieldSize -- Skipping the right subtree's indirection
            getRightMost
        )
```

(b) Traversing a tree, with indirections.

Figure 9 Tree traversals in `packed-data`.

2.2 Example: Incrementing values of a packed tree

The packed data produced by the library is immutable. Since we are working with pointers and raw memory access under the hood, giving writing abilities (even through a monad) would break the purity and referential transparency of the library's API. Writing inside a buffer used by multiple `Packed` values would impact each of these variables, possibly leading to corruption. Just like in functional languages, it is not possible to do in-place modifications on packed data; instead we have to duplicate it.

To do so, we can build a `Needs` value while traversing a packed value. This is what the generated "transform" function is for: it behaves similarly to the "case" function (in the sense that it helps with pattern matching on packed values, and it can be used recursively), but it forces its arguments to be `PackedReader`s that produce `Needs` values. It also helps the programmer by writing leading tags and inserting indirections when necessary.

Figure 10 gives an example of a function that takes a packed tree and produces another where all the leaf values are incremented by one. We use the `transformTree` function (line 3) instead of `caseTree`, as the former invokes the `startLeaf`/`startNode` functions for us and thus

helps keeping the code simple. The first `PackedReader` (lines 4-7) passed to `transformTree` reads the integer of a packed `Leaf` and calls the `write` function from the `Packable` typeclass. The second `PackedReader` (lines 8-12) calls the `incrementPacked` function recursively for each branch and builds back the packed `Node` using the `applyNeeds` function.

```
1  incrementPacked :: PackedReader '[Tree Int] r (Needs '[] '[Tree Int])
2  incrementPacked =
3      transformTree
4          ( do
5              n <- readInt
6              return (write (n + 1))
7          )
8          ( do
9              left <- incrementPacked
10             right <- incrementPacked
11             return (applyNeeds left >> applyNeeds right)
12         )
```

Figure 10 Incrementing values of a packed tree.

2.3 Use case: Web Server

To showcase the perks of using packed data, let's take the example of a very simple web API that has two endpoints "`/sum`" and "`/right-most`". These endpoints respectively compute the sum of the leaves' values and get the right-most value in a packed tree passed to the request body.

Using the Scotty web framework[4], we would implement the endpoints as in Figure 11a. The endpoints are very simple: we get the packed tree, call the corresponding function `sumTree` and `getRightMost` (from Figures 6 and 9a) and send back the result.

The most interesting part of these endpoints is how we get the packed tree (on lines 4 and 9): Scotty's `body` function returns the raw request's body as a *lazy* `ByteString`. However, a `Packed` value relies on a *strict* `ByteString`. Thus, we need to use the `toStrict` function to do the conversion. This operation is $\mathcal{O}(n)$, but considering that a JSON-based API would traverse the entire `ByteString` to parse JSON as well, this is acceptable. `unsafeToPacked` is a $\mathcal{O}(1)$ operation to cast a `ByteString` into a typed `Packed` value. Unlike a JSON-based API, we do not marshal the request body, saving both memory and time.

On the client side (Figure 11b), we enjoy a similar kind of optimisation, as the packed tree can be sent as-is (if the client already uses packed data). Additionally, both `fromPacked` and `fromStrict` (line 5) are $\mathcal{O}(1)$ operations, making the preparation of a request lightweight.

When running this server in parallel with one that uses *unpacked* trees and JSON, we notice that the former runs noticeably faster than the later (Proper benchmarks on packed vs. unpacked data are done in Section 3.2).

[4] `https://hackage.haskell.org/package/scotty`

```haskell
endpoints :: ScottyM ()
endpoints = do
    get "/sum" $ do
        packedTree <- unsafeToPacked . toStrict <$> body
        sum <- liftIO $ runReader sumTree packedTree
        response sum

    get "/right-most" $ do
        packedTree <- unsafeToPacked . toStrict <$> body
        rightMost <- liftIO $ runReader getRightMost packedTree
        response rightMost
```

(a) Handling packed data in a web REST API Server.

```haskell
request methodGet "/sum" [] reqBody
  where
    packedTree :: Packed '[Tree]
    packedTree = ...
    reqBody = fromStrict (fromPacked packedTree)
```

(b) Sending packed data to a web REST API.

Figure 11 Packed data in a web REST API.

3 Evaluation

To understand the impact of using the library to handle packed data, we evaluate it in two ways. First, we take a look at its usability and portability, by comparing how to manipulate packed data and unpacked data. Then, we look at the performance through a benchmark on simple tree traversals.

3.1 Practical considerations

Since computations on packed data are lifted, it imposes limitations on how code can be written. `packed-data` tries to fill the gap between Haskell's features and what one might want to do with packed information, without modifying the compiler. However, this approach has some shortcomings. Let's take a look at five aspects of our library-based approach to evaluate the usability of `packed-data`.

3.1.1 Type-safety and maintainability

`packed-data` provides a high-level and type-safe API for low-level data manipulation. It keeps the user from doing raw and unsafe pointer arithmetic by abstracting these operations behind a functional API.

As an example, consider Figures 6, 12 and 2 where we compute the sum of the leaves of a packed tree using `packed-data` and pointer arithmetic in Haskell and C. In the later cases, dereferencing and pointer arithmetic bloats the code with multiple operations on numeric values that are not very expressive. In both languages, the type-checker will not be able to detect any pointer misuse. On the other hand, when using `packed-data` (Figure 6), pointer arithmetic is hidden away behind functions like `reader`, `caseTree` and the `do` expressions,

which guarantee correct pointer manipulation. Therefore, our library makes code that manipulates packed data more readable, maintainable and memory-safe, compared to code that does raw pointer arithmetic.

```
sumPackedNonMonadic :: Ptr Word8  -> IO (Int, Word8)
sumPackedNonMonadic ptr = do
    tag <- peek ptr :: IO Word8
    let !nextPtr = ptr `plusPtr` 1
    case tag of
        0 -> do
            !n <- peek nextPtr
            let !nextPtr1 = plusPtr ptr 9
            return (n, nextPtr1)
        1 -> do
            (!left, !r) <- go (castPtr nextPtr)
            (!right, !r1) <- go r
            let !res = left + right
            return (res, r1)
        _ -> undefined
```

Figure 12 Summing the values in a tree using raw pointer manipulation in Haskell.

3.1.2 Portability

`packed-data` does not require modifying the compiler or the runtime system. This means that the library's code does not deal with the compiler's implementation details. Thus, `packed-data` can virtually be used with any modern version of GHC (as of today, the library has been tested with GHC versions 9.2 to 9.12). The library can be simply used as a third-party package, with no further constraints.

Not relying on the compiler's internal workings also means that `packed-data` could be reimplemented in other languages with similar type systems like Scala, Rust or TypeScript.

3.1.3 Code Generator's Applicability

Thanks to its code generator (powered by meta-programming), `packed-data` can be used with any user-defined algebraic data type, with the exception of unboxed types (denoted with a trailing `#`, like `ByteArray#`) and types that use them.

It also requires very little boilerplate code (see Figure 7 from Section 2.1.3). Using a Template Haskell entry point like `mkPacked` is a common approach used by other popular Haskell libraries like Aeson[5] for JSON encoding/decoding.

3.1.4 Generated Code and Flags

We saw in Section 2.1.4 that flags could change the definition of the functions created by the code generator, while the names of the generated functions do not change. But what if we wanted to call the code generator twice for the same ADTs, passing different flags? This would lead to the definition of two functions with the same name (e.g. `caseTree`) in the same

[5] `https://hackage.haskell.org/package/aeson`

module, which is not allowed in Haskell. A workaround would be to call the code generator separately in dedicated modules, so that the different definitions are split, and we can import these modules according to our needs.

Another thing to consider is that the `Packed` data type does not us tell whether indirections are present in the packed data. Therefore, if a user wants to build trees with and without indirections, it is up to them to use the correct "case" and "reader" functions generated with the correct flags.

We should also note that the same code cannot be reused with instances of the `Packable` typeclass generated with different indirection flags. Figures 8 and 9 illustrate this problem: because of the strong typing of the packed data, `PackedReader`s need to explicitly skip over indirections (with `skipFieldSize`) when they are present.

3.1.5 Pattern Matching

One of Haskell's strengths is its powerful pattern matching support. Unfortunately, its syntax is not overload-able, meaning that we can not use it to pattern match on packed data. As a solution, the code generator defines a "case" function which takes lambdas as parameter, one for each constructor of the ADT (an example is given in Figure 6).

However, an issue arises from not being able to use the standard syntax: we can't do nested pattern matching. In Haskell, we can pattern-match on a constructor's fields, while with our "case" functions, pattern matching only considers constructors.

`packed-data` is a portable solution that provides type-safety for packed data manipulation. It is flexible (with flags for the code generator) and can be easily integrated to most modern Haskell projects. Even though the API sticks to Haskell's functional and pure paradigm, it does not support features like nested pattern matching.

3.2 Performance

We evaluate the performance of `packed-data` by comparing the execution time of simple tree traversals with pointer-based Haskell, Gibbon, C and Golang. We focus on four kinds of full tree traversals: computing the sum of values in a tree, evaluating an AST for a simple arithmetic language, getting the right-most node and incrementing the leaves' values. Table 1 reports the execution times of these tree traversals.

We should note that we will not compare the size of the packed data with buffers created by Cap'n Proto. This is because `packed-data`'s serialisation format is very similar to Gibbon's, and Vollmer et al. have already demonstrated that the compiler's packing format is more space-efficient than Cap'n Proto [13]. For the same reason, we will not benchmark Cap'n Proto's traversals, as the same paper showcases that Gibbon's traversals are generally faster than Cap'n Proto. Thus, we can use Gibbon's performance to indirectly compare `packed-data` with Cap'n Proto.

For each benchmark, each tree is built symmetrically, meaning that a tree of "size" 1 will have $2^1 = 2$ leaves, a tree of "size" 5 will have $2^5 = 32$ leaves, etc. The C and Haskell code were benchmarked using the Criterion library[6](with Haskell's Foreign Function Interface (FFI) to invoke the C code with negligible overhead). The Gibbon code was compiled with the `-p` and `--no-gc` options to use a packed layout and disable garbage collection. The Golang

[6] `https://hackage.haskell.org/package/criterion`

code was benchmarked using the `testing` library[7]. Code was compiled using GCC 11.4, GHC 9.10, Golang 1.18 and the latest Gibbon version as of March 2025. All benchmarks were run on an Intel machine with two Xeon Gold 6244 CPUs at 3.60 GHz, with 32Gb of RAM, running Ubuntu 22.04 LTS.

In this section, we will ignore the benchmark cases prefixed with *Non-monadic*, we will get back to them in Section 4.

For summing the values in a tree, using `packed-data` can provide up to a 20% speed-up compared to native Haskell. It also beats C by running 15% faster. We can note that, for this specific scenario, having indirections slightly slows down the traversal (around 2%): we do not need them to optimise the traversal, so we skip over them, leading to additional smaller jumps.

Evaluating an AST is similar to summing the values of a tree, as it requires a full traversal of a tree, without skipping any fields. The AST used for this benchmark is one for a small arithmetic language with integer values, addition, subtraction, multiplication, and division. This means that we will not have to deal with two constructors/tags like for the previous benchmark (for leaves and nodes), but five (one for the value, and one for each arithmetic operation). Similarly to the previous benchmark, using `packed-data` provides a 60% speed-up compared to regular Haskell. The *Unpacking, then traverse* line shows the runtime for functions that unpacks the packed AST, and then evaluates it. This case shows that doing so leads to an 8.5x slowdown compared to native Haskell and illustrates the cost of unpacking a value before processing it, i.e. that `time(unpacking)+ time(eval_unpacked)> time (eval_packed)`.

The third benchmark focuses on getting the right-most value in a tree. Unlike the previous cases, here a lot of jumps will be done, as we do not need to go through each branch of the tree. However, using `packed-data`, the operation is 10000 times slower than regular Haskell, as we need to traverse the entire packed tree to reach the last value. But this benchmarks illustrates the benefits of indirections: when they are present, we can skip over subtrees, avoiding a lot of useless traversals. Indirections allow this computation to be only four times slower than unpacked Haskell.

In the final benchmark, we evaluate the performance of incrementing the leaves' values in a tree. Note that the C and Golang benchmarks increment the trees *in-place*, while the others create a new tree. Here, Haskell is much slower than C, since Haskell's functional paradigm forces such operations to produce an entirely new tree. In `packed-data`, this can be done by using the `traverse` function and calling the `finish` function. The performance of `packed-data` are surprisingly comparable with regular Haskell. This can be justified by the fact that `packed-data` only does one big memory allocation for the entire buffer, instead of one per tree node. In the *Unpacking, increment and repack* case, we deserialise the packed tree, increment the leave's values (using the same function as for the native Haskell case), and then repack the tree. The *Deserialise and increment, and repack* case produces an unpacked tree already incremented (i.e. the leave's values are incremented in the "case" function's lambdas) and then re-serialiases the tree. The latter case has better performance than the former, as it avoids an additional traversal and the creation of a new unpacked tree. Again, these last two cases show that it is faster to use unpacked data as-is, instead of deserialising it first, whether it is for read-only or writing operations.

As a final note, we observed via some preliminary benchmarking that the packed data approach appears to perform worse on Apple ARM devices, where the non-packed (pointer-chasing) code is generally faster. We speculate that this may be due to ARM either handling

[7] `https://pkg.go.dev/testing`

Table 1 Execution times of tree traversals.

Language / Tree Size		1	5	10	15	20
C		7.56 ns	57.33 ns	1.76 us	84.04 us	6.06 ms
Haskell		16.21 ns	202.68 ns	6.47 us	207.17 us	6.56 ms
Gibbon		23.00 ns	359.00 ns	13.97 us	234.59 us	6.28 ms
packed-data	①	20.99 ns	150.31 ns	4.86 us	157.94 us	5.14 ms
	②	20.68 ns	152.82 ns	4.84 us	156.19 us	5.22 ms
	③	9.11 ns	133.68 ns	4.47 us	143.39 us	4.68 ms
	④	8.95 ns	134.16 ns	4.54 us	145.97 us	4.91 ms
Golang		5.30 ns	111.30 ns	3.86 us	165.05 us	16.49 ms

①: Without indirections. ②: With indirections.
③: Non-monadic, without indirections. ④: Non-monadic, with indirections.

(a) Execution times for summing the value in a tree.

Language / Tree Size		1	5	10	15	20
C		10.74 ns	121.19 ns	4.65 us	181.40 us	9.24 ms
Haskell		16.22 ns	278.23 ns	8.67 us	280.97 us	13.20 ms
Gibbon		73.00 ns	723.00 ns	10.60 us	181.50 us	5.34 ms
packed-data	①	24.08 ns	168.80 ns	5.25 us	168.18 us	5.41 ms
	②	27.84 ns	471.86 ns	15.69 us	751.96 us	111.96 ms
	③	10.89 ns	151.21 ns	5.11 us	164.25 us	5.27 ms
Golang		5.29 ns	108.90 ns	4.42 us	180.51 us	15.41 ms

①: Without indirections. ②: Unpacking, then traverse. ③: Non-monadic, without indirections.

(b) Execution times for evaluating a packed AST for arithmetic expressions.

Language / Tree Size		1	5	10	15	20
C		7.35 ns	9.20 ns	10.75 ns	28.23 ns	86.15 ns
Haskell		11.63 ns	15.52 ns	19.45 ns	23.33 ns	27.30 ns
Gibbon		29.00 ns	39.00 ns	62.00 ns	93.00 ns	74.00 ns
packed-data	①	21.68 ns	93.63 ns	2.80 us	95.27 us	3.24 ms
	②	21.67 ns	25.85 ns	32.36 ns	43.06 ns	112.77 ns
	③	20.94 ns	215.35 ns	6.60 us	349.04 us	67.94 ms
	④	9.29 ns	77.98 ns	2.82 us	95.95 us	3.32 ms
	⑤	8.12 ns	10.54 ns	16.52 ns	26.86 ns	94.32 ns
Golang		3.45 ns	8.58 ns	18.25 ns	26.61 ns	35.05 ns

①: Without indirections. ②: With indirections. ③: Unpacking, then traverse.
④: Non-monadic, without indirections. ⑤: Non-monadic, with indirections.

(c) Execution times for getting the right-most value in a tree.

Language / Tree Size		1	5	10	15	20
C		7.33 ns	35.45 ns	1.79 us	82.26 us	7.68 ms
Haskell		19.50 ns	294.18 ns	10.55 us	501.31 us	75.83 ms
Gibbon		71.00 ns	635.00 ns	30.40 us	486.78 us	14.42 ms
packed-data	①	41.99 ns	469.68 ns	16.80 us	648.15 us	74.50 ms
	②	60.29 ns	821.67 ns	33.45 us	1.99 ms	245.64 ms
	③	42.57 ns	580.03 ns	21.97 us	1.03 ms	165.04 ms
Golang		6.61 ns	129.60 ns	4.29 us	179.44 us	16.97 ms

①: Using NeedsBuilder. ②: Unpacking, increment and repack.
③: Deserialise and increment, and repack.

(d) Execution times for incrementing the values in a tree.

pointer-chasing within a tree better, or punishing unaligned access more severely. Full investigation of packed data performance on ARM architecture is future work. Regardless, even on ARM, there was a significant benefit to operating directly on packed data rather than the "unpack, process and repack" round-trip.

We conclude that `packed-data` can provide better performance than native Haskell, but not in every case. In the best scenarios, the speed-ups can reach 60%, compared to unpacked data, but in the worst scenarios, it is considerably slower (from 4 to 10000 times). In the next section, we speculate on why sometimes the library performs slower than native Haskell.

4 Reflection

We concluded in the previous section that the performance of `packed-data` were satisfactory. This can be explained by the fact that we make the most of the L1 cache and do not jump around using pointers. In this section we consider an alternative way of writing code in Haskell for packed data that can provide better performance than `packed-data`. This approach, however, is far less maintainable and safe.

In Haskell, monads are not defined at the compiler level, they are regular data structures defined in Haskell's `base` standard library. Therefore, even though they are a big part of the functional paradigm, they do not enjoy any dedicated compiler optimisation.

To manipulate packed data, `packed-data` uses two monads: `IO`, required to do any kind of low-level operation (like dereferencing a pointer using `peek`), and `PackedReader`, which provides the high-level API for packed data traversals.

Since monadic operations can be as optimised as any other kind of computation (e.g. through inlining), inevitably, using two monads on top of each other like `packed-data` must lead to some computing overhead.

We argued in Section 3.1.1 that `packed-data`'s API allowed for readable and type-safe code. Figure 13b is an example of what traversing packed data without `packed-data` would look like. It is safe to say that this code is way less pleasant to read and harder to maintain than with the library (Figure 13a). However, it does get rid of one level of computation induced by the `PackedReader` monad.

Benchmarks using this non-monadic approach are included in Table 1 and labelled as *Non-monadic*. In the summing and AST evaluation benchmarks, this approach yields slightly better performance (between 2 and 6% compared to the monadic approach). However, in the third benchmark, with indirections, it is 19% faster than its `packed-data` counterpart. But it is still 3.5 times slower than regular Haskell, possibly because this computation relies heavily on following indirections and jumping around in the serialized tree. This leads to an extensive use of the `IO` monad, slowing down the traversal.

This confirms that our monadic approach to packed data manipulation causes some kind of computing overhead. However, this is specific to Haskell. Implementations for other languages might not suffer from this, but they should be able to leverage similar performance enhancements. As future work, we could use staging and compile-time evaluation to eliminate that computing overhead.

5 Future Work

While `packed-data` is a working library, it can be improved in many ways; or it can serve as a stepping stone for further research on the topic. In this section, we enumerate possible future work that can be done on the library or in this research domain.

```
evalPacked :: PackedReader '[AST] r Int32
evalPacked =
    caseAST
        reader
        (opLambda (+))
        (opLambda (-))
        (opLambda (*))
        (opLambda div)
  where
    {-# INLINE opLambda #-}
    opLambda ::
        (Int32 -> Int32 -> Int32) ->
        PackedReader '[AST, AST] r Int32
    opLambda f = R.do
        left <- evalPacked
        right <- evalPacked
        R.return (f left right)
```

(a) Using `packed-data`.

```
evalPackedNonMonadic :: Packed (AST ': r) -> IO Int32
evalPackedNonMonadic packed = fst <$> go (unsafeForeignPtrToPtr fptr)
  where
    (BS fptr _) = fromPacked packed
    go :: Ptr Word8 -> IO (Int32, Ptr Word8)
    go ptr = do
        tag <- peek ptr :: IO Word8
        let !nextPtr = ptr `plusPtr` 1
        case tag of
            0 -> do
                !n <- peek nextPtr :: IO Int32
                return (fromIntegral n, plusPtr nextPtr (sizeOf n))
            1 -> opLambda (+) nextPtr
            2 -> opLambda (-) nextPtr
            3 -> opLambda (*) nextPtr
            4 -> opLambda div nextPtr
            _ -> undefined
    {-# INLINE opLambda #-}
    opLambda :: (Int32 -> Int32 -> Int32) -> Ptr Int32 -> IO (Int32,
       Ptr Word8)
    opLambda f ptr = do
        (!left, !r) <- go $ castPtr ptr
        (!right, !r1) <- go r
        let !res = left `f` right
        return (res, r1)
```

(b) Using pointer arithmetic.

Figure 13 Evaluating a packed AST.

Building packed data (through a `Needs` buffer) is slow (four times slower than building native objects). Internally, it relies on the `bytestring-strict-builder` library[8] which, in itself provides good performance[9]. However, as for packed data traversals, building packed data is implemented through a monad, `NeedsBuilder`. As discussed in Section 4, even though it provides a functional and type-safe API, such approach can cause some computing overhead. A future version of `packed-data` could focus on making the packing process faster.

`packed-data` only supports its own packing layout. However, some systems could take advantage of data packed with a custom or pre-defined layout. Take JSON for example: the client of an API that serves JSON data would not have to deserialise the server's response, and could use it as is. This is also applicable for web browsers and HTML, especially since web pages can be sizeable. A next step for `packed-data` would be to handle data packed with a more complex layout like JSON, HTML or CBOR[10].

We have seen that accessing a field in a packed structure is not a straightforward task: we have to either use indirections or traverse the preceding fields. We could draw inspiration from Chilimbi, Davison & Larus' work [5] and reorder fields according to their access pattern, moving the most accessed ones first in a packed structure [11]. This would help programs access "popular" fields more easily, with fewer jumps in the packed buffer.

In `packed-data`, indirections take the form of a 32-bit integer. We could definitely save some space in the packed buffer by choosing the size of this indirection accordingly to the following field. For example, we do not need a 32-bit indirection for a four-characters string, one byte would be enough. To go even further, some packed data do not need indirections at all, if their size can be guesses statically (e.g. a packed character will always be one byte, a 16-bit number two bytes, etc.). In that case, indirections are useless. We can avoid inserting them (reducing the size of the packed buffer) and deduce the size of that packed data by relying on the type-checker and a couple of typeclasses.

As mentioned in Section 3.1, we could also allow the same code on packed data to be reused whether the data has been packed with or without indirections.

Another area of improvement would be not to insert indirections in the packed buffer itself, but in a separate heap object. This would allow for packed data to be smaller and enable parallelism in tree traversals.

Finally, because of the data layout used by `packed-data`, memory access can be unaligned. This noticeably affects performance on ARM CPUs. In the future, `packed-data` could include a flag that forces the insertion of padding in the packed buffer, to align tags, indirections and fields.

The portability of `packed-data` across other languages remains theoretical. While its implementation does not rely on modifying the compiler or the runtime system, it does heavily rely on Haskell's type-system. It would be interesting to implement `packed-data` in an other typed language like Scala, TypeScript or Rust.

We concluded in Section 4 that our library-based approach in Haskell suffered from a computing overhead caused by the `IO` monad, blocking any possible future enhancement of the library's performance. Deeply-embedded domain-specific languages (EDSLs) could be a solution to this problem. EDSLs expose functions that do not actually execute an operation,

[8] `https://hackage.haskell.org/package/bytestring-strict-builder`
[9] `https://github.com/haskell-perf/strict-bytestring-builders`
[10] `https://cbor.io`

but instead produce an AST [12]. These ASTs can then be optimised [1] and compiled into a full-blown program in the host language or produce a standalone executable [3]. A few EDSLs have performance as their main objective [1], and to reach it, it's not uncommon to rely on a backend written in a lower-level language, like C [6].

To provide better performance, `packed-data` could be reimplemented as a deeply embedded DSL: at compile time, using Template Haskell, we could produce an AST and generate C code. Using Template Haskell's quotation feature, we could inject the call to that C code into the Haskell program, using the language's C FFI. This way, we would still have a type-safe and functional API, while enjoying C's fast performance [7]. Furthermore, this C code could be reused as a common backend for implementations of `packed-data` in other languages.

6 Conclusion

Exchanging data using a packed format can be faster than when using a human-readable format like JSON, as the recipients can use the serialised data as-is, avoiding any deserialisation step. Manipulating packed data can also help save computation time, as it avoids chasing pointers. However, this format is not easy to adopt, as most implementations require either compiler modifications or a dedicated compiler.

In this paper, we investigate library-based support for packed data through `packed-data`. This Haskell library provides a type-safe and functional API that allows packing data, processing it, and unpacking it. It relies on Template Haskell and Haskell's type-system, meaning that this library could virtually be implemented in any language that supports meta-compilation and higher-kinded types.

Our benchmarks show that the library can leverage better performance than native Haskell (up to 60%), but in some scenarios like getting the right-most value in a tree, `packed-data` (with indirections) is four times slower. This is due to a computation overhead caused by the monadic approach used in the implementation, as benchmarks show that "pure" pointer arithmetic-based implementations can provide more competitive performance.

While a library-based approach for packed data support is feasible and portable, there is a performance and usability tradeoff to consider. We think that `packed-data` would benefit from staging and compile-time optimisations, for example, through an EDSL, which would generate and call C code. This would help get rid of computing overhead caused by the host language, and could be reused to build a common backend used by implementations of similar libraries in other languages.

References

1 Johan Ankner and Josef Svenningsson. An EDSL approach to high performance haskell programming. In Chung-chieh Shan, editor, *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, pages 1–12. ACM, 2013. doi:10.1145/2503778.2503789.

2 Robert Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3-4):335–376, 2009. doi:10.1017/S095679680900728X.

3 Emil Axelsson and Mary Sheeran. Feldspar: Application and implementation. In Viktória Zsók, Zoltán Horváth, and Rinus Plasmeijer, editors, *Central European Functional Programming School - 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*, volume 7241 of *Lecture Notes in Computer Science*, pages 402–439. Springer, 2011. doi:10.1007/978-3-642-32096-5_8.

4 Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL):5:1–5:29, 2018. doi:10.1145/3158093.

5 Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In Barbara G. Ryder and Benjamin G. Zorn, editors, *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*, pages 13–24. ACM, 1999. doi:10.1145/301618.301635.

6 Frank Dedden. Compiling an haskell edsl to c, 2018. URL: https://studenttheses.uu.nl/handle/20.500.12932/29176.

7 Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *J. Funct. Program.*, 13(3):455–481, 2003. doi:10.1017/S0956796802004574.

8 Arthur Jamet and Michael Vollmer. packed-data. Software, version 0.1.0.3. (visited on 2025-05-02). URL: https://hackage.haskell.org/package/packed-data.

9 Chaitanya Koparkar, Mike Rainey, Michael Vollmer, Milind Kulkarni, and Ryan R. Newton. Efficient tree-traversals: reconciling parallelism and dense data representations. *Proc. ACM Program. Lang.*, 5(ICFP):1–29, 2021. doi:10.1145/3473596.

10 Tim Sheard and Simon L. Peyton Jones. Template meta-programming for haskell. *ACM SIGPLAN Notices*, 37(12):60–75, 2002. doi:10.1145/636517.636528.

11 Vidush Singhal, Chaitanya Koparkar, Joseph Zullo, Artem Pelenitsyn, Michael Vollmer, Mike Rainey, Ryan Newton, and Milind Kulkarni. Optimizing Layout of Recursive Datatypes with Marmoset: Or, Algorithms + Data Layouts = Efficient Programs. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, volume 313 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:28, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2024.38.

12 Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding for EDSL. In Hans-Wolfgang Loidl and Ricardo Peña, editors, *Trends in Functional Programming - 13th International Symposium , TFP 2012, St. Andrews, UK, June 12-14, 2012, Revised Selected Papers*, volume 7829 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2012. doi:10.1007/978-3-642-40447-4_2.

13 Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. Local: a language for programs operating on serialized data. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 48–62. ACM, 2019. doi:10.1145/3314221.3314631.

14 Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, Chaitanya Koparkar, Milind Kulkarni, Sam Tobin- Hochstadt, and Ryan Newton. Compiling tree transforms to operate on packed representations. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 26:1–26:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ECOOP.2017.26.

15 Edward Z. Yang, Giovanni Campagna, Ömer S. Agacan, Ahmed El-Hassany, Abhishek Kulkarni, and Ryan R. Newton. Efficient communication and collection with compact normal forms. In Kathleen Fisher and John H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC , Canada, September 1-3, 2015*, pages 362–374. ACM, 2015. doi:10.1145/2784731.2784735.

WebGlitch: A Randomised Testing Tool for the WebGPU API

Matthew K. L. Wong
Department of Computing, Imperial College London, UK

Alastair F. Donaldson
Department of Computing, Imperial College London, UK

Abstract

We report on our experience designing a new technique and tool for fuzzing implementations of WebGPU, a W3C standard JavaScript API for in-browser GPU computing. We also report on our experience using our WebGlitch tool to test industrial-strength implementations of WebGPU, leading to the discovery of numerous bugs. WebGPU enables programmatic access to a device's graphics processing unit (GPU) for in-browser GPU computing, and is being implemented by Google, Mozilla and Apple for inclusion in all of the major web browsers. Guaranteeing the security and reliability of WebGPU is crucial to avoid wide-reaching browser security vulnerabilities and to facilitate portability by ensuring uniform behaviour across different platforms. To that end – inspired by randomised compiler testing techniques – our approach to fuzzing creates random, valid-by-construction programs by continuously selecting a WebGPU API function, then recursively generating all requirements necessary for that API call to be valid based on careful modelling of the API specification. This is implemented as a new open source tool, WebGlitch, which we designed in consultation with engineers at Google who work on the Chrome WebGPU implementation. WebGlitch identifies bugs through sanitiser-boosted crash oracles, differential testing, and by identifying cases where valid-by-construction API calls lead to runtime errors. We present an evaluation showing that WebGlitch can find bugs missed by an existing WebGPU fuzzer, wg-fuzz, and across the broader WebGPU ecosystem: to date, WebGlitch has found 24 previously-unknown bugs (15 fixed so far in response to our reports). Among these, 17 bugs affected WebGPU implementations from Google, Mozilla, and the Deno project. WebGlitch found an additional 4 bugs in the shader compilers used by the graphics APIs that WebGPU interfaces with. The remaining 3 bugs affect the widely-used JavaScript runtimes Node.js and Deno. Fuzzing with WebGlitch also led us to identify an ambiguity in the specification of the WebGPU shading language, for which we proposed an amendment that was accepted by W3C and which has been adopted in the latest version of the specification. Analysing the line coverage of a WebGPU implementation by WebGlitch-generated programs revealed that WebGlitch covers code missed by wg-fuzz and the official conformance test suite. Our hope is that this report on the design of WebGlitch and its deployment in practice will be useful for practitioners and researchers interested in using API fuzzing to improve the reliability of industrial codebases.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging; Software and its engineering → Object oriented languages

Keywords and phrases Fuzzing, WebGPU, WGSL, API, shaders

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.39

Category Experience Paper

Supplementary Material *Software (ECOOP 2025 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.11.2.14`

Funding This work was supported by EPSRC grant EP/R006865/1 and gift funding from Google.

Acknowledgements We are grateful to Corentin Wallez, Alan Baker and David Neto at Google for their input during the design and deployment of WebGlitch, and to Daniel Simols for assistance in using the wg-fuzz tool.

© Matthew K. L. Wong and Alastair F. Donaldson;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

In this experience report we investigate the use of randomised testing techniques for improving the reliability of implementations of the WebGPU API.

WebGPU [58] is a new W3C standardised web application programming interface (API) that allows building websites that make programmatic use of a device's graphics processing unit (GPU) for real-time rendering and general-purpose computation. WebGPU abstracts away the intricacies of platform-specific graphics APIs, providing a unified, object-oriented JavaScript interface to execute commands on the GPU and configure render and compute passes, including the running of *shader programs*, which execute across the massively parallel hardware that GPUs offer. In practice, implementations of WebGPU conform to this interface by delegating WebGPU API calls to appropriate combinations of calls in lower level, platform-specific graphics APIs such as Direct3D [37], Metal [2], and Vulkan [25].

Three major WebGPU implementations are currently under development, by Google (for Chromium-based browsers, including Chrome and Edge), Apple (for Safari), and an implementation primarily by Mozilla (for Firefox) [51].

As a web standard, WebGPU will be enabled on billions of browser instances, so defects in implementations of the API that affect WebGPU-enabled content, or worse, exploitable vulnerabilities, have the potential to be wide-reaching. Since WebGPU requires interfacing with kernel-level graphics drivers, exploits can potentially bypass browser sandboxing features [49], and these bugs have the potential to be triggered by merely visiting a web page. Indeed, many high severity CVEs have been identified for an existing web graphics API, WebGL [40, 42, 41]. WebGPU's increased complexity over WebGL presents an even wider attack surface and makes it more difficult to test. While there is an official conformance test suite (CTS) [50], it is unrealistic for handwritten tests to cover all edge cases, especially as problems may arise due to both bugs in the WebGPU implementation itself and bugs in platform-specific shader compilers and graphics APIs.

API fuzzing can help cover these corner cases by randomly generating sequences of API calls [5, 44, 22, 21]. In this work, we investigate the use of API fuzzing for finding bugs in implementations of WebGPU. In consultation with engineers at Google who work on the WebGPU implementation in Chromium, we developed a Java-based open source command line tool, WebGlitch: a black box fuzzer for the WebGPU API. We report on our experience designing and implementing WebGlitch, and deploying it in practice to find bugs in production WebGPU implementations.

WebGlitch generates JavaScript programs that make intricate use of the WebGPU API. Generated programs can then be fed to a WebGPU implementation under test, also testing the downstream graphics API and graphics drivers on which the implementation depends.

We have designed WebGlitch to ensure that the WebGPU programs it generates by default are valid-by-construction, according to the WebGPU specification. This is important because programs generated in a naive fashion would be unlikely to yield programs featuring API calls that satisfy WebGPU's complex validation rules. Invalid API calls would only exercise a limited subset of the implementation under test. Another benefit of generating valid-by-construction programs is that it allows for the automatic detection of validation bugs, where valid calls to API functions are wrongly rejected.

To ensure rigorous testing of parts of WebGPU related to compiling and executing shaders, we have integrated WebGlitch with WGSLsmith, a state-of-the-art random generator of WebGPU shading language (WGSL) programs [13, 28].

WebGlitch constructs programs incrementally, iteratively choosing random API functions to generate until a pre-defined limit is reached. Many API functions require parameters that refer to objects that must have been created previously via other API calls. At each generation step, rather than being restricted to only using previously-generated objects as the receiver and parameters for an API call, WebGlitch uses an *on-demand* approach, as follows. An *arbitrary* API function is chosen at random. A suitable receiving object and parameters for the chosen function are then obtained via a combination of (a) using in-scope objects from previous calls (if available), and (b) recursively generating randomised calls to other API functions to yield a receiving object and suitable parameter arguments when needed. On-demand generation increases the probability of rich API functions that require complex arguments being invoked: taking the simpler approach of only allowing calls to API functions for which suitable arguments have already been generated would bias generation towards favouring simple functions requiring a small number of simple arguments.

To maximise the chances of finding bugs, WebGlitch uses multiple test oracles in conjunction: sanitiser-boosted crash oracles (using sanitisers such as AddressSanitizer [47], LeakSanitizer [20], and UndefinedBehaviorSanitizer [20]), differential testing [32], and confirming that no errors are thrown for a valid WebGPU program (hereafter termed the "validity" oracle). The validity oracle hinges on the fact that WebGlitch generates programs that are valid by construction.

Over a period of 5 months, at various stages during its development, we have used WebGlitch to test the Dawn WebGPU implementation from Google [19] and the wgpu implementation from (primarily) Mozilla [45], used in the Chromium and Firefox browsers, respectively. This has led to WebGlitch identifying 24 previously-unknown bugs, of which 17 affected WebGPU implementations, 4 affected shader compilers used in downstream graphics APIs, and 3 affected the JavaScript runtimes Node.js and Deno. In addition, WebGlitch independently identified 5 bugs that turned out to have already been reported.

As well as finding bugs in implementations of WebGPU, our testing efforts identified an ambiguity in the specification of WGSL related to overflow checking for numeric types. We clarified the intended semantics with engineers at Google who are leading the design of WGSL, and based on this discussion we proposed a clarification to its specification to resolve the ambiguity, which as been accepted and incorporated into its latest version [52].

We present a set of controlled experiments on Google's WebGPU implementation, comparing the lines of code covered by running the WebGPU CTS vs. by conducting a short fuzzing campaign using WebGlitch, and a comparable fuzzing campaign using an existing WebGPU fuzzer, wg-fuzz [48]. Our results show that both WebGlitch and wg-fuzz generated programs that covered lines missed by the CTS. WebGlitch was able to cover 1,530 lines not reached by wg-fuzz, despite supporting the generation of fewer WebGPU API functions outlined in the specification than wg-fuzz. These two fuzzers complement each other, with wg-fuzz being able to cover 5,180 lines missed by WebGlitch.

We also present a summary of the insights that we gained on API fuzzing from our fuzz testing campaign of WebGPU, which may be valuable for researchers and practitioners considering deploying API fuzzing in the future. A key point here is that although our approach allowed the detection of numerous previously-unknown bugs, our fuzzing campaign required frequent manual intervention because of both the evolving and layered nature of the WebGPU API.

In summary, the contributions of this paper are:

- The development of a technique for fuzzing WebGPU implementations, implemented as the WebGlitch open source tool.

- A report on the use of WebGlitch in practice to find bugs in industrial-strength implementations of WebGPU from Google and Mozilla, to find bugs in associated infrastructure including the Node.js and Deno JavaScript runtimes, and to resolve an ambiguity in the WebGPU shading language specification.
- The findings of a controlled experiment assessing the code coverage achieved by WebGlitch in comparison to the WebGPU CTS and another WebGPU fuzzer, wg-fuzz. Our results show that WebGlitch covers lines of code that are missed by wg-fuzz, and that there are potential gaps in the CTS.

Paper Structure

We first give background on the WebGPU ecosystem, including the way in which it interfaces with downstream graphics APIs and the shader compilation process (Section 2). We then discuss the design and implementation details of WebGlitch (Section 3), before presenting an overview of the bugs found by WebGlitch (Section 4). We also conduct a controlled study to evaluate the abilities of different tools – the WebGPU CTS, wg-fuzz, and WebGlitch – to exercise and test a WebGPU implementation (Section 5). We finish by discussing key insights and lessons learned from our work (Section 6), related work on API fuzzing and compiler testing (Section 7), and directions for future work (Section 8).

Availability

Our WebGlitch tool is available as open source [53], and an artifact accompanying the paper is available [56]. WebGlitch is exposed as a set of command line tools: a *generator* component that generates JavaScript program that uses WebGPU; a *runner* component that runs sets of generated programs on a target platform, and a separate *fault-finder* program that mines the logs produced by the runner component to check for bugs, which are then categorised by the test oracle that found them.

2 WebGPU Background

WebGPU is a powerful new API that allows web applications to leverage the GPU for demanding tasks such as data visualisation, producing high-quality renders for 3D design tools, and on-device machine learning [57]. The WebGPU API allows a JavaScript program to encode GPU commands in buffers, which are then sent to the GPU for execution [58]. For example, there are commands for setting up render and compute passes, wherein shader program execution can be configured. The entire API is object-oriented, and all API functions are invoked on some receiving WebGPU object.

There are three major WebGPU implementations currently in development: Dawn, developed by Google and used in the Chromium project, on which the Google Chrome and Microsoft Edge browsers are based [19]; wgpu [45], primarily developed by Mozilla and used in Firefox, and an implementation from Apple [8] as part of the WebKit browser engine on which Safari is based. The wgpu implementation is noteworthy in that it is written in and made for Rust, and it is not fully compliant with the W3C WebGPU standard to make its usage more idiomatic to the Rust language. The wgpu project is used as the core implementation of WebGPU functionality in Firefox. However, an additional layer above wgpu is required to map JavaScript WebGPU API calls to wgpu ones and to fully adhere to the specification [45]. Google and Apple's WebGPU implementations are both written in C++. Although WebGPU is designed to be used in-browser, bindings are also available to

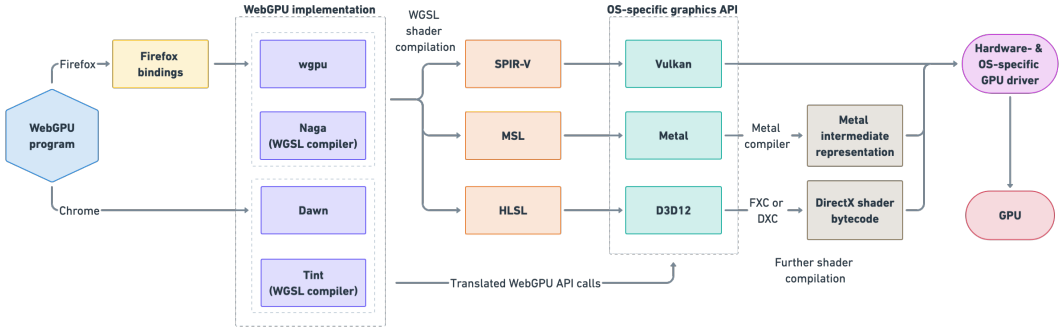


Figure 1 Simplified overview of the WebGPU architecture and WGSL compilation chain, created using information from wgpu and Chromium documentation [45, 49]. Tint is technically part of Dawn, but has been shown separately here for clarity. HLSL can be compiled by FXC [34] (deprecated) or DXC [35]. Apple's implementation in Safari, which only targets the Metal backend, has been excluded for clarity.

enable its use from JavaScript runtimes such as Node.js and Deno. As with Firefox, Deno [11] also uses wgpu as the core of its WebGPU implementation, building on top of it with its own independent implementation.

When a WebGPU API call is made in JavaScript, it is validated according to the correct usage defined in the WebGPU specification. If valid, it is then mapped on to one or more platform-specific graphics API calls (see Figure 1). At the time of writing, the APIs that are used in practice are: Direct3D 12 (D3D12) (for Windows) [37], Metal (for macOS, iOS, iPadOS) [2], and Vulkan (for Linux, Android) [25].

The downstream API calls made by the WebGPU implementation should be valid-by-construction. But bugs related to making these calls could mean that the lower-level API is used incorrectly, which can lead to undefined behaviour and crashes that could be exploited [36]. Even if there are no errors related to their usage, bugs within the implementation of these lower level APIs could be exploited through WebGPU programs. The major WebGPU implementations are also written in languages that allow for memory-unsafe code to be written (C++ and Rust[1]). As such, heap-use-after-frees and buffer overflows could be exploited for arbitrary remote code execution.

A key part of graphics programming is the issuing of shader programs for execution on GPU hardware. Because WebGPU provides an abstraction that is independent of any particular graphics API, it has its own associated shading language: WebGPU Shading Language (WGSL) [59]. Because the underlying graphics APIs expect shaders to be written in specific shading languages, such as HLSL (D3D12) [38], MSL (Metal) [3], and SPIR-V (Vulkan) [26], a WebGPU implementation must first compile a WGSL shader into the appropriate target language (Figure 1). This is accomplished using the WGSL compilers Tint (for Dawn) [19], Naga (for wgpu) [45], and wgslc (for Safari/WebKit). The resulting shader will then be further compiled by the shader compiler that ships with the driver for the user's GPU device to enable execution on the GPU hardware. Worryingly, bugs within the WGSL compilers themselves, as well as downstream compilers, have the potential to be exploited.

[1] Rust code that uses the "unsafe" feature of the language is not guaranteed to be memory-safe, and unsafe Rust is used in the wgpu project.

All these potential attack vectors combine to create a highly complex attack surface, similar to the one present in WebGPU's predecessor, WebGL [27]. This attack surface includes the WebGPU implementation and WGSL compiler used by a browser, alongside the lower-level platform-specific graphics API used by a device. However, the implementations of these APIs and their associated shader compilers are typically managed by third parties, resulting in a multi-layered attack surface that is difficult to test comprehensively.

Potential vulnerabilities have already been identified. These include an issue within the WebGPU implementation Dawn [49], an overflow bug in a downstream shader compiler [49], and several high-severity CVEs related to Tint [6].

3 Design and Implementation of WebGlitch

After an overview of how programs are generated by WebGlitch (Section 3.1), we explain how WebGPU API functions are represented (Section 3.2) and discuss the algorithm WebGlitch uses to generate valid-by-construction programs (Section 3.3). A worked example of this algorithm is then presented in Section 3.4 We then discuss how shaders are incorporated in generated programs (Section 3.5), and the test oracles that WebGlitch supports (Section 3.6). Finally, we present experimental results that give a ballpark estimate of the throughput of fuzzing using WebGlitch (Section 3.7).

3.1 WebGPU Program Generation Overview

To generate random WebGPU programs from scratch, WebGlitch incrementally adds new API calls to a skeleton JavaScript program. This skeleton contains components such as the main function and a helper function to load shaders. We refer to the program comprising all the calls that have been generated so far as a *partially-generated* program. At all times, WebGlitch keeps track of the state of the partially-generated program in order to generate valid API calls.

The following steps are repeated N times, for some fixed N that is provided as a parameter to the tool.

1. Choose a random API function, f, to call.
2. Select a receiver for f: an object on which f should be called. This may be an existing object resulting from a previous call, or may involve WebGlitch recursively generating additional API calls to bring a receiver of the right type into scope.
3. Issue any necessary function calls on the chosen receiver to bring it into a suitable state for a call to f to be valid.
4. Analyse the arguments required for the call to f. For primitive types and enumerated values, randomly choose a value from a set or range of valid ones. For other WebGPU objects, either find an in-scope object that satisfies the requirements, or generate one that does.

To illustrate step 3: suppose f needs to be called on a `GPUCommandEncoder` object with no active child passes. A `GPUCommandEncoder` is a WebGPU object that stores commands intended for execution on the GPU, such as render passes and compute passes. These "child" passes of a `GPUCommandEncoder` are considered "active" if the `end` method has not yet been invoked on it. If WebGlitch has selected a receiver representing a `GPUCommandEncoder` object with an active child pass, WebGlitch would need to invoke `end` on the child pass to bring the receiver into the necessary state.

Two programs generated using N repetitions of the above steps will have *roughly* the same size, because they will each feature N API function calls chosen via step 1. However, the sequence of functions that are called will likely be entirely different. Randomised recursive generation in steps 2–4 to fulfil the requirements of these calls will also lead to diversity.

The generation of a given call is guaranteed to terminate as there are a finite number of arguments per API function, and there are no cyclic dependencies between the requirements of functions in the API. In addition, WebGlitch uses a bound on the number of calls that will be generated, so generation is always guaranteed to terminate.

We now delve into the technical details of how this approach works, and our choice of this strategy for program generation.

3.2 Defining API Call Signatures and Requirements

Generation of valid WebGPU programs, which we describe in full in Section 3.3 below, hinges on a rigorous understanding of the signatures and requirements of API functions as defined in the prose specification.

Inspired by RESTler [5], an API-agnostic fuzzer which uses an interface definition language (IDL) to represent the structure of API functions, we designed a custom representation to specify the WebGPU API. This allowed us to add support for generating more basic API calls quickly. Each API function is specified via a JSON object, using a format that expresses characteristics shared by all API functions (e.g., number and types of arguments, and certain categories of requirements) to be expressed externally to the WebGlitch codebase (Figure 2). On execution, WebGlitch parses these definitions and uses the requirements they specify to guide its generation process. This decoupling of core API requirements from the source code of WebGlitch proved convenient during the development of the tool: it allowed additional functions to be made available to the tool, or early errors in our understanding of the API specification to be fixed without modifying or recompiling the WebGlitch source code.

```
"name": "writeBuffer",
"returnType": "none",
"properties": [{
  "buffer": {
    "type": "GPUBuffer",
    "conditions": {
      "same": ["GPUDevice"],
      "withAttributes": {
        "GPUBuffer.usage": ["GPUBufferUsage.COPY_DST"],
        "GPUBuffer.mapped": ["false"]
  }}}}
]
```

(a) Function to write data to a buffer on the GPU.

```
"name": "unmap",
"returnType": "none",
"conditions": {
  "withAttributes": {
    "GPUBuffer.mapped": ["true"]
  }
},
"setAttributes": {
  "this.mapped": ["false"]
}
```

(b) Function to return access to a buffer back to the GPU.

Figure 2 Examples of the JSON format for specifying WebGPU API functions and their requirements. Not all arguments have been listed for clarity. Some arguments require WebGPU objects, and their required state can be specified through the *withAttributes* field under the *conditions* field. These values can also be dynamic. The *setAttributes* field indicates that the function modifies the state of the object on which the function is called.

We populated these function requirements through careful analysis of the WebGPU specification [58], and updated them in response to occasional changes to the specification. Gathering such requirements was manual and could not be automated, as many API functions had unique constraints that required iterative updates to the JSON representation we used.

In addition to the general classes of requirements expressed in our JSON format, some API functions have more intricate requirements. Since extending our JSON representation to express such requirements was not straightforward nor elegant, information about them is "baked in" to WebGlitch – expressed in the logic of the WebGlitch source code. For example, creating a pipeline for rendering textures (through `createRenderPipeline`) can involve specifying various information about the vertex shader that will be used, which is difficult to express in a JSON representation.

Validation of WebGPU API function signatures was principally through practical testing against two implementations of WebGPU: Dawn and wgpu. As valid-by-construction programs are generated by default, when a WebGPU implementation under test rejects such a program this indicates that there is a bug in that implementation or a bug in the way the API function has been modelled in WebGlitch, which would be fixed iteratively. In cases where this was not clear from the specification, we consulted engineers at Google who work on Dawn.

3.3 Creating Valid Programs

WebGlitch constructs programs using an internal representation akin to an abstract syntax tree, but simpler because generated programs consist of a linear sequence of API calls.[2] Once generation is complete, the internal representation is pretty-printed as a JavaScript program.

Recall from the introduction and Section 3.1 that we have designed WebGlitch generate programs that are valid-by-construction: they always call WebGPU API functions on receiving objects that are in an appropriate state, passing the right number and types of arguments, with arguments satisfying any requirements described in the API specification.

The motivation for ensuring that generated programs are valid is as follows. When an *invalid* API call is made, minimal logic of the WebGPU implementation is executed: only the logic that checks for argument validity. Importantly, no functionality of the underlying graphics API will be invoked, and the WebGPU API call will not produce a meaningful result that can be used as input for subsequent functions.

We considered two approaches to generating valid-by-construction programs: a *conservative* approach, where an API function is selected to be invoked only if the partially-generated program has already created suitable WebGPU objects to fulfil the requirements of the selected function, and an *on demand* approach, where *any* API function can be selected to be invoked, with recursive generation being used to fulfil any missing requirements.

As already illustrated by the overview in Section 3.1, we opted for on-demand generation. Although the conservative approach is simpler, it suffers from the problem that many WebGPU functions have extensive requirements that must be satisfied in order for calls to be valid. It is unlikely that all such requirements will happen to be satisfied as a result of prior API calls in the partially-generated program. As a result, the most interesting and complex API functions would rarely be invoked in generated programs.

Once an API function has been selected at random, generation of a call to this function by WebGlitch can be split into two stages:

1. choose or create a random receiver object and ensure that it is in a suitable state for the function call;

[2] From the perspective of testing WebGPU implementations, there would be no value in including loops or conditional statements in generated JavaScript programs: such constructs would exercise the JavaScript engine, but would not exercise the WebGPU API in any additional depth, since e.g. a long linear sequence of calls can be used to call particular API functions multiple times.

2. for each function argument, choose a suitable value at random (for enumerated and numeric types), or choose or create a suitable WebGPU object at random and ensure that it is in a suitable state for the function call (for object arguments).

The state of a WebGPU object o is determined by the arguments passed into the function that created o, the state of the WebGPU object on which the constructing function was called, and any API calls that have been made on o since it was constructed.

In step 1 above, if a suitable receiver object with the required state is not available, WebGPU issues calls to construct one from scratch and bring it into a suitable state, or to bring an existing object into a suitable state. This process is recursive, because the creation of a required WebGPU object may require the creation of other WebGPU objects. We illustrate this via a worked example below.

Even if suitable receiver objects are already available for a call, WebGlitch randomly decides (via a configurable probability) whether to use one of these or generate a new receiver. The motivation for this is to reduce the probability that all API calls are made on the same or a small number of objects, which would limit program diversity.

Once a suitable receiver has been found or created, WebGlitch iteratively generates a valid value for each expected argument. For arguments with enumerated types, invalid values are determined by analysing the state of the receiver and object on which the constructor was called, and are filtered out. A value is chosen at random from the remaining subset. For arguments with numeric type, a random value is chosen within a valid minimum and maximum value. Certain numeric arguments may also need to be divisible by a constant value to be considered valid, in which case the randomly chosen value is rounded to the nearest divisible number within the range.

The most complex arguments to generate are other WebGPU objects, because their validity for a given call does not only depend on their type, but also on their state. The process of selecting or creating suitable arguments for a call is analogous to that for selecting or creating the receiving object for the call. However, in some cases the chosen receiving object can constrain the choice of valid arguments for the call. We illustrate this with a worked example in Section 3.4.

To support generation of valid programs, WebGlitch uses a symbol table (akin to that of a compiler) to track the types and states of objects in the partially-generated program throughout its generation. This includes relationships between parent and child objects (because new objects are created by invoking API functions on existing objects), and history of API calls that have been invoked on each object.

Allowing Invalid Function Calls

While there are several advantages to generating WebGPU programs that are guaranteed to be valid (as argued above), we deemed it also useful to equip WebGlitch with a mode where API calls within a program have some probability of being invalid, creating programs that are *nearly valid*. This enables the detection of instances where a WebGPU implementation's validity checks deviate from the specification or differ from those of another implementation.

To cater for this, WebGlitch can be configured with a percentage chance to bypass its internal validity analyses. WebGlitch can skip verifying both the receiver's state and the validity of the arguments passed into the API call, such that invalid WebGPU objects can also be passed as arguments in API calls further down.

3.4 Worked Example: Generating a Valid WebGPU API Call

To illustrate the above ideas, we show a call to the `writeBuffer` API function (Figure 3), which writes data into GPU memory, could be generated.

```
writeBuffer(GPUBuffer buffer, GPUSize64 bufferOffset, AllowSharedBufferSource data)
```

Figure 3 Signature of the `writeBuffer` method of the `GPUDevice` class. Optional arguments not shown for simplicity. GPUBuffer: a WebGPU object of type GPUBuffer. GPUSize64: an unsigned long long. AllowSharedBufferSource: an instance or view of a JavaScript ArrayBuffer or SharedArrayBuffer object. Adapted from [58].

Figure 4 shows what the partially-generated program might look like after all the ingredients required for the call to `writeBuffer`, and the call itself, have been generated. Being able to make this call depends on various WebGPU-specific constructs, including `GPUAdapter` (from which a WebGPU object representing the GPU is requested), `GPUDevice` (programmatic representation of a GPU), `GPUQueue` (handles executing GPU commands), and `GPUBuffer` (representing GPU memory) [58]. The specific details of these constructs do not matter much for the example.

```
const adapter = await navigator.gpu.requestAdapter();
const device = await adapter.requestDevice();
const queue = device.queue;
const buffer = device.createBuffer({
  mappedAtCreation: true
  size: 12
  usage: GPUBufferUsage.COPY_DST
});
const bufferOffset = 8;
const data = new Uint8Array([1, 2, 3, 4]);
queue.writeBuffer(buffer, bufferOffset, data);
```

Figure 4 WebGPU program to write data to a buffer.

Suppose WebGlitch is at the very start of creating a WebGPU program, so that the partially-generated program is empty, and suppose that the `writeBuffer` function is randomly selected. The receiver object for a call to this function must be a `GPUQueue`, but no such object exists in the (empty) partially-generated program. Because a `GPUQueue` is accessed via a property of a `GPUDevice`, and no `GPUDevice` is available in the empty program, WebGlitch issues a call to `requestDevice` to create a `GPUDevice`. However, `requestDevice` requires a `GPUAdapter` as its receiver, thus WebGlitch issues a call to `requestAdapter` to create a `GPUAdapter`. No additional generation work is required to call `requestAdapter` because it is invoked on the `navigator.gpu` property. The `navigator` object is part of the Document Object Model and provides access to the WebGPU API.

Armed with a `GPUQueue` object, `queue`, WebGlitch can now proceed to generate a call to `writeBuffer` on `queue`. In some cases, certain state-changing functions must be invoked on an object before a particular desired function can be called, but this is not the case for `writeBuffer`: it can be safely called on a freshly-created `GPUQueue` object. WebGlitch can therefore proceed to generate suitable parameters for the call to `writeBuffer`. Instructed by the JSON files in which requirements of WebGPU API functions are encoded, WebGlitch knows that `writeBuffer` requires a `GPUBuffer` created from the same `GPUDevice` from

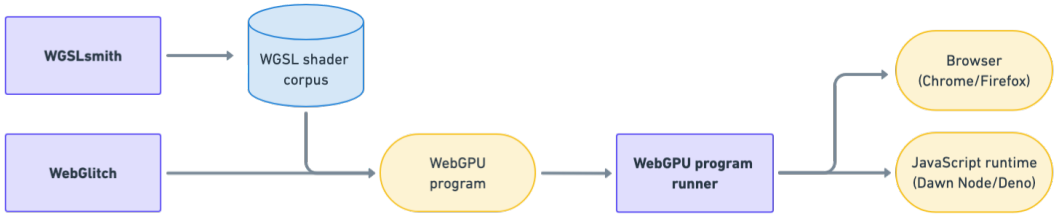


Figure 5 Overview of the WebGlitch workflow. The program runner configures the environment such that the same program generated by WebGlitch can be run within a browser and a JavaScript runtime.

which the `GPUQueue` was created, and with a size that is a multiple of 4. The buffer must also be allowed to be writeable, so it issues a call to `createBuffer` on `device` with the necessary arguments. The remaining arguments to `writeBuffer` also have restrictions (e.g., the `bufferOffset` parameter must be a multiple of 4 and the data to be written must fit within the offset and buffer size). Correspondingly, WebGlitch creates arguments that meet these requirements, employing randomisation where the requirements allow for a range of acceptable values.

Without taking care to generate valid programs by construction, it is very unlikely that even this relatively simple function would be called with valid arguments by chance.

3.5 Integrating Shader Programs

To exercise the portion of WebGPU related to shader compilation and execution, WebGlitch integrates with an existing WGSL shader generator, WGSLsmith [13, 28]. This combination leverages WGSLsmith's ability to construct a diverse range of shaders, which WebGlitch uses within the WebGPU API calls it generates (Figure 5). This results in WebGPU programs featuring a variety of GPU pipelines composed of randomised shader programs and configurations, which are ultimately executed in either a browser or JavaScript runtime environment. However, it is challenging to integrate shader programs and, by extension, WGSLsmith, into WebGlitch. API calls involving shaders often require specific arguments, such as for buffer sizes and bind group layouts, in order for them to be valid. It is difficult to automate this process without tooling to parse and understand WGSL shaders, which is challenging to get right. Rather than having WebGlitch invoke WGSLsmith during the program generation process, we instead used WGSLsmith to generate a corpus of shaders ahead of time, and manually determined the necessary bindings for these, encoding information about bindings in a JSON format that WebGlitch processes. The corpus of shaders used has varied over time: as discussed in Section 4.2 after finding that some shaders triggered interesting shader compiler bugs (which we reported), we removed them to avoid such bugs persistently occurring. We also removed some shaders due to changes in the WGSL specification which meant that certain features of WGSLsmith-generated shader programs were no longer valid.

3.6 Test Oracles

WebGlitch uses three test oracles to detect bugs: sanitiser-boosted crash oracles, the validity oracle, and differential testing. The validity oracle is enabled by default as long as the option to generate invalid programs is turned off. Straightforward crashes are also detected by default. The sanitiser-boosted crash oracle amounts to compiling the WebGPU

implementation under test with AddressSanitizer [47], which automatically enables LeakSanitizer [20], or UndefinedBehaviorSanitizer [20]. AddressSanitizer helps detect memory errors like out-of-bounds accesses and use-after-frees by making them more likely to cause a crash with a clear error message when they may otherwise go unnoticed. In a similar manner, UndefinedBehaviorSanitizer detects undefined behaviour like integer overflows and dereferencing a null pointer.

The differential testing oracle requires the user of WebGlitch to first execute the same set of generated WebGPU programs on different platforms and save the console output. We then use a script to parse and compare the results, flagging any discrepancies and notable error messages. WebGlitch then highlights programs which have unexpected differing outputs. The sanitiser-boosted crash oracle and validity oracle are straightforward to detect bugs with. Although differential testing is conceptually straightforward (run the same program on different implementations and check for differences), there are various nuances that must be considered when applying differential testing to WebGPU, which we now discuss.

Comparing Shader Outputs

One challenge is with accessing compute shader outputs, which are stored in GPU memory rather than main memory. Copying back and printing the results of every shader executed requires many additional API calls. But forcing every WebGPU program to always include these API calls would limit program diversity, and potentially prevent some bugs from being discovered. For these reasons, when invoking a compute shader WebGlitch makes a random choice as to whether each of the buffers associated with the compute shader should be copied back and printed.

Another issue applies to both graphics and compute shaders. While WebGPU contains very few implementation-dependent behaviours, the result of executing shaders can be different across different GPUs and drivers due to floating-point rounding [28]. Round-off error can lead to legitimate differences between results computed by shaders on different platforms, which could confound differential testing. As such, comparing outputs would require distance metrics and threshold values to identify differences [14], which is non-trivial. However, for compute shaders, this issue can be addressed using WGSLsmith [13, 28]. WGSLsmith only generates compute shaders and uses reconditioning [28] to avoid floating-point rounding. During reconditioning, all floating point operations are substituted with calls to functions that always return numeric values that can be represented exactly using the IEEE-754 standard. Implementation-dependent behaviours can also arise from "dynamic errors" in WGSL, which can be caused by out-of-bounds array accesses and unbounded loop iterations. WGSLsmith also avoid these behaviours are also by inserting checks on array bounds and loop iterations [28]. If WebGlitch were to be integrated with other shader generators, this would need special attention.

Comparing Error Messages

Recall that WebGlitch has a mode wherein validity checking has a configurable chance of being skipped for any receiver chosen and argument generated. In this case, we *do* expect to receive errors from the implementations under test. The type of error thrown for an invalid WebGPU API call is standardised in the WebGPU specification, so we can use differential testing to check for equivalent error output for a WebGPU program executed using different WebGPU implementations. A challenge here, however, is WebGPU does not standardise error message contents [58], so we parse and convert to an implementation-

independent representation for comparison. Different WebGPU implementations are also at varying degrees of completion [51], requiring manual analysis to determine the subset of API functions and arguments supported by all implementations for differential testing.

Differential Testing at WebGPU Program Runtime

A practical challenge is that it is difficult to directly differentially test WebGPU programs across multiple platforms as this would require platform-specific graphics APIs and drivers to be available simultaneously. To work around this, we run generated WebGPU programs separately on different platforms and log their outputs. The outputs are then parsed and compared post-execution by a separate program *FaultFinder*, which looks for discrepancies in outputs, error messages in valid programs, and crashes.

3.7 Throughput of WebGlitch

To provide an estimate of the throughput of fuzzing using WebGlitch, we used commit 29a4e87 to generate 1000 programs. We then executed each program against the Dawn implementation of WebGPU, targeting the Vulkan downstream graphics API via Node.js. Throughput was not measured on wgpu/Deno due to a bug that resulted in a Rust panic when executing almost every WebGPU program generated (Table 1, row 26). Experiments were performed on a machine with a 5.4 GHz Intel i7-13700K CPU, 64 GB DDR5 RAM, and an NVIDIA RTX 4080. They were run on Ubuntu 24.04. With default settings, WebGlitch took an average of 0.675 seconds to generate a WebGPU program. The majority of the time was spent on execution, which averaged 2.932 seconds.

4 Practical Impact on the WebGPU Ecosystem

To date, our use of WebGlitch has identified 29 bugs across the WebGPU ecosystem, consisting of 24 previously-unknown bugs and the independent discovery of 5 bugs that turned out to have already been reported.

Of the previously-unknown bugs, all have been reported, with 15 fixed following our submitted reports and 5 others confirmed.

We focused our testing efforts on two of the three major WebGPU implementations, Dawn (from Google and used by Chrome) and wgpu (primarily from Mozilla, and used in Firefox), on the major graphics APIs supported by Windows (D3D12), macOS (Metal) and Linux/Android (Vulkan).

We began by testing Dawn and wgpu through their JavaScript runtime bindings for Node.js and Deno, respectively. We then moved on to direct testing of the Chrome and Firefox browsers for completeness, because WebGPU is ultimately exposed to end users via browsers. Testing was conducted in-browser and within JavaScript runtimes on macOS,[3] Windows,[4] and Linux.[3]

Table 1 provides a summary of the 29 bugs identified by WebGlitch. The “Issue” column provides the ID of the bug in the issue tracker for the relevant project, with a hyperlink. The table is separated into specification issues, issues that might affect WebGPU support in Chromium and Firefox, shader compilation issues, issues impacting WebGPU bindings in Node.js and Deno, and JavaScript runtime bugs. WebGlitch was able to find bugs across

[3] MacBook Pro, Apple M2 Pro, 16 GB RAM
[4] Intel i7-13700K, 32 GB DDR5 RAM, NVIDIA RTX 4080

Table 1 Summary of all issues found by WebGlitch and the test oracles that found them. All "Duplicate" bugs marked with a * were first found by wg-fuzz. All other bugs classed as "Duplicate" were user-reported. FXC [34] is an HLSL shader compiler used in D3D12. The LLVMpipe driver from the Mesa 3D graphics library [33] allows for commands intended for execution on the GPU to be run on the CPU instead. Graphics libraries have been specified in parentheses for cases where bugs were specific to them. Otherwise, bugs affected all graphics libraries tested for that operating system (Linux: Vulkan, Windows: D3D11 with FXC & D3D12 with DXC, macOS: Metal).

	Issue	Type	Backend	Oracle	Description	Status
Issue affecting WGSL specification						
1	#4795	Ambiguity	N/A	Differential	Description of language behaviour when inferring types of numeric literals unclear	Fixed
Issues affecting Chromium (via Dawn)						
2	#350775867	Logic	All	Crash	Error message thrown for valid WebGPU program (exhibited only via Chrome)	Duplicate
3	#355605693	Logic	Windows	Validity	Validation error thrown for correct program	Confirmed
Issues affecting Firefox (via wgpu, or its Firefox integration)						
4	#1923237	Logic	All	Validity	Incorrectly throws error for valid program (exhibited only via Firefox)	Confirmed
5	#24798	Crash/hang	All	Crash	Crashes/hangs when specific buffer is created, followed by the destruction of a GPUDevice	Fixed
6	#24813	Logic	All	Differential	Fails to check that all WebGPU objects within a pass are from the same GPUDevice	Fixed
7	#24832	Logic	All	Differential	Missing check for GPUTextureView validity within a render pass	Fixed
8	#6040	Implementation lag	All	Validity	wgpu unknowingly lagging behind current WebGPU spec	Confirmed
9	#6041	Logic	All	Differential	Incorrect error message thrown	Fixed
Issues affecting shader compilation						
10	#6023	Logic	All	Differential	Naga: Implementation of constant evaluator deviates from WGSL specification	Fixed
11	#352496179	BSOD	Windows (FXC)	Crash	FXC: Poor performance of FXC resulted in 100% CPU and memory usage	Fixed
12	#42250583	Crash	Windows (FXC)	Crash	FXC: Valid HLSL code causes FXC to crash when paths with no side effects are encountered	Duplicate
13	#355603413	Crash	macOS	Crash	Metal shader compiler: Valid MSL involving division by 0 crashes Metal on Apple GPUs only	Confirmed
14	#372512081	ASan	Linux (LLVMpipe)	Sanitiser	LLVMpipe: Floating point exception	Unconfirmed
Issues affecting Node.js bindings for Dawn						
15	#346264226	Crash	All	Crash	Unreachable statement reached	Duplicate*
16	#346356636	Crash	All	Crash	Unreachable statement reached	Duplicate*
17	#346197748	Crash	All	Crash	Fatal error when making illegal API call	Duplicate*
18	#364871809	Crash	All	Crash	Fatal error when retrieving error message	Unconfirmed
19	#351644575	ASan	All	Sanitiser	Heap-use-after-free (object accessed after destruction)	Fixed
20	#355006329	LeakSan	All	Sanitiser	Not all resources freed on program exit	Fixed
Issues affecting wgpu integration in Deno						
21	#24806	Crash	All	Crash	Missing argument in error message constructor	Fixed
22	#24672	Logic	All	Differential	Fails to throw error message for invalid render/compute pass	Fixed
23	#24821	Logic	All	Differential	Arguments used in an unrelated render pass affects compute pass output	Confirmed
24	#25874	Logic	All	Validity	GPUAdapter incorrectly invalidated after specific API call	Fixed
25	#25870	Crash	All	Crash	Panics when destroyed texture used in render pass	Fixed
26	#28966	Crash	All	Crash	Panics when command encoder is finished	Unconfirmed
Issues affecting JavaScript runtimes (unrelated to WebGPU)						
27	#24575	LeakSan	All	Sanitiser	Deno leaks memory when JavaScript VM is exited	Fixed
28	#849	LeakSan	All	Sanitiser	Deno leaks memory when JavaScript VM is exited	Fixed
29	#54178	LeakSan	Linux	Sanitiser	Node.js leaks memory when "requiring" a non-existent module	Unconfirmed

the WebGPU ecosystem, including WebGPU implementations, WGSL compilers, the WGSL specification itself, downstream shader compilers, and JavaScript runtimes. Four new bugs (Table 1, rows 6, 7, 24, 25) were found by executing WebGPU programs containing invalid API calls, highlighting the importance of designing WebGlitch with a configuration that allows for invalid API calls in addition to its default mode where generated programs are valid-by-construction (see Section 3.3). The remaining bugs were all found through programs that were valid-by-construction. The range of sources within which bugs were detected demonstrates the importance of fuzz testing WebGPU in as many environments as possible.

We now elaborate on some of the bugs found by WebGlitch and their impact on the WebGPU ecosystem.

4.1 Specification Issue

WGSL is statically-typed and supports integer types of various bit-widths [59]. It also allows the bit-widths of certain numeric expressions to be left unspecified and inferred based on usage. This concept is referred to as *abstract numerics* [59]. The WGSL specification requires that expressions involving abstract numerics are checked at compile-time for overflow, so that a program is statically rejected if the value of an abstract numeric does not fall within the allowed range of values supported by a type of a specific bit-width.

To illustrate this, consider the first declaration in Figure 6. The abstract numeric expression `-1 * -2147483648` evaluates to `2147483648`, which is too large to be represented by the `i32` type (a 32-bit signed integer). The declaration should thus be rejected.

```
// Correctly rejected by Tint & Naga
const num1 : i32 = -1 * -2147483648

// Incorrectly rejected by Naga only
const num2 : i32 = -1 * i32(-2147483648)
```

Figure 6 We fixed a problem in the WGSL language specification where rules for implicit type casting were ambiguous. This was brought to light by the Naga compiler incorrectly rejecting the second declaration in this example.

In contrast, the second declaration should *not* be rejected. This is because the expression `i32(-2147483648)` specifies that `-2147483648` should be stored as a 32-bit signed integer, which is allowed because the value is in the representable range of the `i32` type. Overload resolution means that the abstract numeric argument `-1` is implicitly converted to an `i32` value (as an abstract numeric value is used with an `i32` value). The WGSL specification mandates two's complement semantics for `i32` operations, so the overall expression evaluates to `-2147483648`.

Differential testing between Dawn and wgpu revealed that the Tint shader compiler (used by Dawn) was behaving correctly, while the Naga shader compiler (used by wgpu) was incorrectly rejecting the second declaration [39].

However, when we investigated this discrepancy, it was not clear to us from the wording of rules about abstract numerics in the WGSL specification which of Tint or Naga were behaving correctly. Our understanding of the type-checking rules described above only came about after discussions with engineers at Google leading the development of the WGSL specification: they were not clear from the specification itself. We submitted a proposal to clarify this behaviour in the specification, which has since been accepted and incorporated [52]. After investigation, the bug in Naga stemmed from its constant evaluator rather than the way in which it handles type casting. A fix to then align the behaviour of Naga with the WGSL specification was submitted and accepted [55].

4.2 Bugs Affecting In-browser Execution

We found 13 previously-unknown bugs that affected either the core logic of a WebGPU implementation or downstream shader compiler behaviour. Although we found most of these bugs by fuzzing using JavaScript runtimes, rather than through in-browser fuzzing, they would affect any WebGPU workload that exercised these parts of the relevant WebGPU implementations, or that used shaders that would trigger the shader compiler bugs. Discussed below, we also found some bugs via in-browser testing that do not trigger via out-of-browser JavaScript runtimes, demonstrating the value of also conducting in-browser testing.

Bugs in the Implementation of API Functions

We found 8 bugs arising due to the incorrect implementation of API functions (7 of them previously-unknown); these are under the "Issues affecting Chromium (via Dawn)" and "Issues affecting Firefox (via wgpu, or its Firefox integration)" headings in Table 1. We now discuss a selection of these.

Two bugs were due to wgpu failing to reject invalid API calls related to configuring render passes (Table 1, rows 6, 7). One WebGlitch program generated by WebGlitch (shown in reduced form in Figure 7) caused wgpu to crash on macOS, and to hang on Linux and Windows.

```
const adapter = await navigator.gpu.requestAdapter();
const device = await adapter.requestDevice();
const buffer = device.createBuffer({
  mappedAtCreation: true,
  size: 12,
  usage: GPUBufferUsage.COPY_SRC
});
device.destroy();
```

Figure 7 Minimal test case causing wgpu to hang on Linux and Windows but crash on macOS.

Identifying the component responsible for this bug was difficult due to the layered architecture in use: it was initially detected while testing Deno, which builds on top of the Rust-native wgpu. As such, we had to manually translate the WebGPU program to a corresponding sequence of direct Rust API calls to confirm the bug was in wgpu.

Two bugs could *only* be triggered by in-browser fuzz testing with Firefox and Chrome – i.e. they could not be found by fuzzing against the Node.js and Deno runtimes.

For Firefox, this is expected: like Deno, Firefox uses wgpu, but builds on top of wgpu in a separate and non-trivial way to enable its use through the Firefox internal JavaScript engine. As such, bugs in this layer are only reachable by fuzz testing in Firefox; this is the case for the bug that we found (Table 1, row 4).

In the case of the bug found in Chrome, which led to an error being thrown for a valid WebGPU program (Table 1, row 2), it is unclear why it was not reachable through Dawn's Node.js bindings, and the cause of the bug is still under investigation.

Shader Compiler Bugs

In addition to the Naga bug discussed in Section 4.1 and Figure 6, we found 4 bugs affecting downstream shader compilers of specific graphics APIs, demonstrating the need to thoroughly test on a range of platforms.

One such bug (Table 1, row 13) affected the Metal compiler for Apple's Metal shading language. Recall from Section 2 that the WGSL compiler in a WebGPU implementation emits code for a downstream shading language. On macOS, we found that Tint, the WGSL compiler that ships with Dawn, generated Metal shading language code that caused the Metal compiler to fail. Determining that the problem was in the Metal compiler was non-trivial due to the layered nature of WebGPU: it is often unclear whether some invalid program was compiled by the WGSL compiler, or whether the source of the bug is further downstream. Moreover, this bug only affected shader compilation when targeting Apple GPUs – we found the crash did not happen when targeting Intel and AMD GPUs on Apple platforms, showing the importance of testing on different hardware configurations. To avoid repeatedly triggering this bug (which would impede the discovery of further bugs), we had to remove the offending shader from the corpus of shaders used by WebGlitch, something that we had to do for various shader programs that would reliably trigger compiler bugs.

4.3 Bugs Related to Runtime Bindings

A number of the previously-unknown bugs we found – 6 in total – turned out to relate to problems with the bindings between the Node.js and Deno JavaScript engines and the Dawn and wgpu implementations of WebGPU, and could not be triggered via in-browser execution (which does not use these bindings).

```
const adapter = await navigator.gpu.requestAdapter();
const device = adapter.requestDevice();
console.log(adapter.info);
```

Figure 8 Minimal test case where valid WebGPU program rejected by Deno.

While less serious in terms of impact than browser-related bugs, our reports of these problems improved the stability of these runtime bindings, and early fixes for some of these issues unblocked our work, allowing us to perform more intensive testing of Dawn and wgpu via these JavaScript runtimes without the binding-related problems masking the presence of other bugs.

Figure 8 shows a reduced test case that triggered a bug in Deno (Table 1, row 24). Our report led to this being raised in the wgpu project's weekly WebGPU meeting [54]. Within this test case, the `adapter.info` call was incorrectly rejected by Deno, throwing an error message that the `GPUAdapter` had been invalidated because a `GPUDevice` had been created from it.

In the Dawn bindings for Node.js we discovered a use-after-free (Table 1, row 19) and memory leak (Table 1, row 20). These were problematic for our fuzz testing campaign: even a trivial program Figure 9, which consists of just the entry point into the WebGPU API, triggered ASan and LeakSan errors Figure 9. Since these errors were thrown upon using the WebGPU API, we had to disable AddressSanitizer and LeakSanitizer while testing Dawn until engineers at Google fixed the problems. Because these bugs were limited to Dawn's Node.js bindings, which, understandably, are a lower priority compared than issues reproducible in Chrome, they took longer to address. As such, it temporarily blocked the discovery of a few other bugs in Dawn.

```
const adapter = await navigator.gpu.requestAdapter();
const device = await adapter.requestDevice();
```

(a) Minimal reproducible WebGPU program that raised repeated Sanitiser errors.

```
==87490==ERROR: AddressSanitizer: heap-use-after-free on address 0x00010d317d38 at pc 0x000127ec6a08 bp 0x00016b21e190 sp 0x00016b21e188
  WRITE of size 1 at 0x00010d317d38 thread T0
SUMMARY: AddressSanitizer: heap-use-after-free AsyncRunner.cpp:63 in wgpu::binding::AsyncRunner::QueueTick(Napi::Env)
  ::$_0::operator()(Napi::CallbackInfo const&) const

==17903==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 108 byte(s) in 1 object(s) allocated from:
  #0 0x7f660d2c1357 in operator new[](unsigned long) ../../../../src/libsanitizer/asan/asan_new_delete.cpp:102
SUMMARY: AddressSanitizer: 60285 byte(s) leaked in 55 allocation(s).
```

(b) Erroneous output detected by WebGlitch. The full error message has been excluded for brevity.

Figure 9 Minimal reproducible test case and console output for a bug raising ASan and LeakSan errors on Dawn node.

4.4 Bugs Affecting JavaScript Runtimes

Our testing led to the discovery of three previously-unknown bugs – all memory leaks – that affected the Node.js and Deno JavaScript runtimes and which turned out to be unrelated to WebGPU: they could be triggered by JavaScript programs that do not issue any WebGPU

API calls. We found such errors by running generated WebGPU programs against WebGPU implementations within JavaScript runtimes. When we reduced these programs to their minimal reproducible example, in each case we found that we could remove all WebGPU API calls while preserving the memory leak. This led to very simple JavaScript programs, e.g. a program that simply imports a module that cannot be found. The lack of any use of WebGPU indicated that such bugs lay within the underlying runtimes. One of the memory leak bug reports we submitted to Deno (Table 1, row 27) helped identify the root cause of an earlier, elusive memory leak that had been challenging to track down. Although this was not a WebGPU-related issue, and we did not set out to discover such types of issues, the impact of this bug was arguably more substantial than that of WebGPU-specific bugs as it affects all developers using Deno, not just those working with WebGPU.

5 Code Coverage Achieved by WebGlitch

To evaluate how thoroughly fuzzing with WebGlitch exercises a WebGPU implementation, we report on statement coverage on the Dawn codebase achieved via a short fuzzing campaign. We compare this to coverage achieved through fuzzing with wg-fuzz. This is the only other open source fuzzer we are aware of for the WebGPU API [48], and the coverage achieved by running the WebGPU conformance test suite (CTS) [50]. The WebGPU CTS is a fixed test suite to check if a WebGPU implementation conforms to the specification, ensuring coverage results gathered from it are comprehensive. We decided to focus on Dawn because at time of writing it is more mature than other open source WebGPU implementations.

Dawn already ships with tooling to measure coverage achieved by the CTS using llvm-cov [30]. To maintain a consistent environment for collecting coverage data, we generated 2000 WebGPU programs using each of WebGlitch and wg-fuzz, but in a format expected by the CTS, so that the Dawn coverage infrastructure could work with these programs as if they were CTS tests. We attempted using more WebGPU programs but found this led to an out-of-memory error as it exceeded the available RAM (32 GB) in our test system. However, analysis of coverage from one of these aborted experiments suggested that coverage had already saturated after 2000 programs.

Among the 2000 programs generated by each fuzzer, 1000 programs were valid by construction, and the remaining programs had a 10% chance for any given call to be invalid.

We then collected coverage data on the Dawn codebase separately by executing (a) the 2000 programs generated by WebGlitch, (b) the 2000 programs generated by wg-fuzz, and (c) the WebGPU CTS. Due to the randomised nature of WebGlitch and wg-fuzz, we repeated coverage collection for these tools three times, using three different sets of 2000 programs, and averaged the absolute percentage coverage values that we obtained. We then analysed the results to identify coverage gaps in the CTS: lines of code in Dawn covered by the WebGPU fuzzers but not by the CTS.

WebGlitch, wg-fuzz, and the CTS were able to cover 13.52%, 15.71%, and 28.00% of Dawn, respectively [56]. These values are all low because many parts of a WebGPU implementation are graphics API-specific, and it is not possible to execute the entirety of any WebGPU implementation on one machine; e.g. the macOS platform that we used for this experiment would only exercise the Metal back-end of Dawn. Even though the coverage of Dawn achieved by both WebGlitch and wg-fuzz is lower than that of the CTS, both fuzz testing tools successfully uncovered new bugs. Absolute coverage alone does not capture the distinction between deep, exploratory execution paths and broad functional coverage. As such, these fuzzing tools serve as a valuable complement to the CTS.

WebGlitch and wg-fuzz both achieve similar coverage, but the coverage achieved by wg-fuzz is slightly higher. This is likely because wg-fuzz covers all API functions, unlike WebGlitch which currently supports 75% of all WebGPU API functions. However, WebGlitch can generate many optional arguments that are not considered by wg-fuzz. Supporting these optional arguments required additional engineering to ensure that generated programs are still valid by construction. The use of these optional arguments will lead to different code paths being taken in the underlying WebGPU implementation under test. Despite WebGlitch supporting fewer WebGPU API functions, it managed to cover 1,530 lines that wg-fuzz missed and uncovered many bugs that wg-fuzz could not. Conversely, wg-fuzz covered 5,180 lines of Dawn that WebGlitch missed. Together, these tools complement each other.

Both WebGlitch and wg-fuzz were also able to cover lines of Dawn missed by the CTS, although most were related to formatting outputs. But among these lines of code was a Dawn function entirely missed by the CTS, and this function was linked to a correctness bug within Dawn that was identified by WebGlitch (Table 1, row 3). This underlines the value that fuzzing can play in identifying and filling coverage gaps in a CTS, as prior work on shader compiler fuzzing has demonstrated [15].

6 Insights and Lessons Learned

We now summarise a number of key lessons learned from our experience designing and deploying WebGlitch, which we hope will be insightful to other researchers and practitioners interested in API fuzzing.

On-demand Generation of API Calls Is Good for Bug Finding

Recall from Section 3 that WebGlitch takes an *on-demand* approach to API call sequence generation, whereby the conditions for a particular API function to be called are satisfied by issuing previous API function calls to create necessary ingredients. In contrast, the wg-fuzz fuzzer for WebGPU takes a more cautious approach where an API function is only selected for invocation if suitable prerequisite objects are already in scope.

WebGlitch uncovered many bugs (24) that wg-fuzz could *theoretically* find, but missed in practice due to its conservative approach: the probability of specific calls being generated is low. For example, the minimal reproducible test case for one identified bug (Table 1, row 23) consisted of a valid WebGPU program with 24 WebGPU API calls and 15 different WebGPU objects. It is already unlikely that a generated program will contain these 24 API calls in the required sequence, and the probability of generating such a program would be even lower when using a conservative approach that only invokes API functions using objects already available in the partially generated program. WebGlitch is capable of exercising API functions that require complex dependencies by creating these on-demand, and the success of WebGlitch at finding bugs in practice provides evidence that the tool is effective at exercising complex WebGPU implementations in depth.

Based on our experience, we recommend considering an on-demand approach when designing a generator of programs that make calls into an API whose functions have non-trivial prerequisites.

Fuzz Blockers Limit the Automated Nature of Fuzzing

Although much of the fuzz testing process can be automated (test case generation, execution, and analysis), our fuzz testing campaign required frequent manual intervention to make fuzzing productive. While using WebGlitch to find bugs is fully automatic, since WebGlitch

uses a black box approach, determining the root causes of bugs is still manual. This was mainly due to *fuzz blockers*, where "a fuzzer encounters the same bug very often, which blocks further fuzzing of downstream code" [12]. In our setting this corresponds to bugs in a WebGPU implementation (or bugs affecting a JavaScript runtime) that are extremely easy to trigger, manifesting so often that they make it hard to find deeper bugs. For example, we had to identify and remove certain shaders that had a high probability of triggering shader compiler bugs, or disable sanitisers until certain bugs were addressed by maintainers.

The issue of fuzz blockers is well known to fuzzing experts; e.g. Google's ClusterFuzz framework assigns a "Fuzz-blocker" label to a bug that is deemed to trigger so frequently that it hurts fuzzer performance [4], and indeed fuzz blockers were a real impediment to our work deploying WebGlitch in practice.

Finding the Root Cause of a Bug Within a Layered API Is Unclear

Generally, the standardization of WebGPU as JavaScript API makes the task of testing WebGPU implementations easier than it would otherwise be: a single generated WebGPU program can be executed across a range of operating systems and graphics drivers, potentially triggering bugs in any of the components shown in the layered diagram of Figure 1.

However, standardization also presents a challenge: when a test fails it can be hard to identify the source of the bug due to the layered nature of the API. Root-causing bugs requires a degree of detective work, e.g. ascertaining that a bug is likely to be Vulkan-specific because it triggers only when a Vulkan back-end is used, regardless of operating system or graphics driver, or ascertaining that a bug lies in an NVIDIA graphics driver by differentially testing across drivers from multiple vendors. This insight is likely to be relevant to fuzzing efforts for other APIs with diverse, layered implementations.

Testing Implementations of an Evolving Standard: A Challenge and an Opportunity

We also faced challenges due to the WebGPU API standard being incomplete during our fuzzing efforts. A benefit of fuzzing at this early stage was that we were able to find some issues with the specification of the API itself, prompting discussion and refinement. However, it meant that we had to actively work on keeping the fuzzer up-to-date with the evolving specification. We also had to be mindful of whether implementations under test were up-to-speed with the latest specification changes, otherwise updating the fuzzer to reflect the latest specification changes would waste developer time by identifying known issues where developers have not yet caught up with such changes. Similar observations were made in recent work on the deployment of compiler fuzzing techniques for WGSL, where early fuzzing of WGSL compilers uncovered an edge case unaccounted for by the specification, showcasing a benefit of applying fuzzing at an early stage, but where the evolving WGSL syntax introduced additional engineering overhead and posed similar challenges with differential testing [13].

This is a trade-off that should be borne in mind when deciding how early to commence a fuzzing effort for implementations of a new API.

Diversity of Testing Environments Is Important

Our work has benefited from the application of WebGlitch to WebGPU implementations in the context of a variety of browsers and operating systems.

However, it is likely that some bugs that WebGlitch would be capable of finding have gone undetected because the range of environments in which we were able to test is nevertheless limited. For example, some bugs affecting WebGPU are known to be specific to operating

system versions [1], or hardware-specific [18]. Any configuration that is left untested is a potential source of undiscovered bugs. This insight is particularly applicable to performance-critical APIs where it is likely that implementations of the API will leverage low level features of operating systems and hardware.

Supporting a new environment in the testing process is not always straightforward. For example, enabling sanitisers on macOS requires a particular environment variable to be set, but this variable is cleared prior to command execution for security purposes. Although there are workarounds, some can be fairly involved [46].

We are working with the WebGPU team at Google to deploy WebGlitch as part of ClusterFuzz, Google's continuous fuzzing infrastructure, which will ensure that fuzzing of WebGPU support in Chromium is conducted on a wide range of target platforms.

7 Related Work

WebGlitch generates valid-by-construction WebGPU programs from scratch, inspired the random C program generator Csmith [60]. Both these fuzzers are specific to a system under test. Many API fuzzers are more general [21, 23, 31, 5]. Among these, RESTler [5] is notable for generating syntactically correct calls by parsing an IDL and attempts to learn the sequences of calls required for validity by gathering and analysing the API call responses. FuzzGen [22], a fuzzer for C++ libraries, attempts to determine which sequences of API calls are valid by constructing a dependency graph of API calls based on analysis of a corpus of programs that use a given API. This is useful when a library's API has not been defined using an IDL. Although applicable to many different APIs, these fuzzers are unlikely to capture all nuances of any specific API, especially related to call validity.

Like Fuzzgen, the Randoop fuzzer for Java classes [43] also determines valid sequences of function calls at runtime. The design of WebGlitch is also similar to Randoop [43]: they both randomly choose a method to call and populate it with arguments created from previously generated calls. But rather than having prior knowledge of each API function's requirements, dependencies, and exactly how to construct each object like WebGlitch, Randoop executes the partially generated program as soon as it is created [43]. It then checks the result of executing the program against a set of pre-defined rules (e.g., a function should not throw an exception under certain conditions). If executing the partially-generated program did not throw any errors or violate any rules, then it is added to a corpus of partially-generated programs, otherwise it is discarded, unless the program threw an error while not violating any conditions, suggesting a potential bug. Randoop can then take objects created within these partially-generated programs and use them in another method call by extending the partially-generated program [43].

One drawback of this approach is that the search space for programs is large as Randoop does not have exact knowledge of which arguments are valid. Moreover, program generation can be inefficient as each time a method call is added, it must be executed to check if it is still valid. Most importantly, such an approach would not work well with the WebGPU API: the API contains many functions to configure the state of objects such as render/compute pipelines, buffers, and textures. However, errors are not always thrown immediately: it is possible to pass invalid configurations to an object, but if that object is never passed to the GPU for manipulation, no error is thrown. Given how unlikely it is that a partially-generated program within the corpus could be extended to include a valid call that passes objects to the GPU, most programs would fail to meaningfully exercise the underlying graphics

APIs. While this could potentially be addressed by adding extensive custom validation when executing programs, it would likely be more efficient to adopt a fuzzing approach that is tailored towards WebGPU instead.

One such example is wg-fuzz [48], which generates valid WebGPU programs from the ground up like WebGlitch, but takes a different approach to ensuring validity. It iteratively chooses random API calls to construct based on the state of the partially-generated program rather than first constructing all the API calls needed for a chosen call to be valid like WebGlitch. WebGlitch nevertheless found 25 bugs that wg-fuzz could not and rediscovered 3 that wg-fuzz had already reported (Section 4). That said, the two tools remain complementary, as wg-fuzz can cover some parts of Dawn that WebGlitch does not and vice versa (Section 5).

Fuzzing the WebGL and WebGPU ecosystems has not been limited to program generation from scratch. GLeeFuzz [44], which targets WebGL, uses an error message-guided mutation approach. GLeeFuzz is not aware of the validity of the programs it mutates, but uses the error messages thrown by mutant programs to guide its mutation. It uses error message diversity as a proxy for coverage, intensively exercising the call validation aspect of WebGL implementations. A GLeeFuzz-like technique would be interesting to investigate for WebGPU, but because its mutation-based approach primarily generates invalid programs, it would likely serve as a complement to WebGlitch, which focuses on generating valid programs.

GLFuzz [14] is a testing technique targeting compilers for the OpenGL shading language, a language similar to WGSL and used by the OpenGL, OpenGL ES and WebGL APIs [24]. GLFuzz uses metamorphic testing [7], wherein semantics-preserving mutations are applied to a seed program and the outputs of the original and mutant are compared for differences. Unlike WebGlitch, GLFuzz focuses solely on shader compiler testing and not on API testing.

There has also been work on fuzzing WebGPU's embedded shading language, WGSL, rather than the WebGPU API itself [6, 13, 28]. The WGSLsmith tool [13, 28] exclusively tests WebGPU shader compilers by generating random WGSL shader programs and detecting errors during compilation or execution of the shaders post-compilation. However, WGSLsmith exercises the WebGPU API in a limited way, as WGSL shaders are executed using a fixed WebGPU test harness. By contrast, WebGlitch makes extensive use of the broader WebGPU API in general and incorporates a fixed corpus of shaders into the API calls it generates, complementing WGSLsmith's shader-focussed approach. A recent paper [13] reports on the use of coverage-guided mutation-based fuzzing (via libFuzzer [29]) and black box fuzzing (via WGSLsmith and spirv-fuzz [17]) to directly test the Tint shader compiler in Google's Dawn WebGPU implementation, and to test tooling associated with SPIR-V, the Vulkan shading language. DarthShader [6] combines both program generation and mutation, fuzzing WGSL compilers in a coverage-guided manner. It has successfully found numerous bugs including several high severity CVEs affecting Chrome [6]. Part of its success can be attributed to its ability to mutate both the abstract syntax tree representation of a shader and its intermediate representation [6], enabling it to more effectively find bugs across different compilation stages. However, this approach is not directly applicable to the WebGPU API layer.

As discussed in Section 6, the WebGlitch and WGSLsmith projects both required manual effort to keeping the respective tools up-to-date with changes to the WebGPU specification. Furthermore, both tools tailored specifically to the WebGPU setting. Recent work has investigated Large Language Model (LLM)-based approaches that could potentially address these limitations [9, 10]. Both TitanFuzz [9] and FuzzGPT [10] use program generation techniques that can be applied to any API or programming language. TitanFuzz first gives a LLM step-by-step instructions to generate seeds, which are then mutated using an evolutionary algorithm to mask portions of the seed. This is then fed back to an LLM for

infilling, and mutants that execute without errors can then act as new seeds [9]. FuzzGPT extends TitanFuzz by either prompting or fine-tuning the LLM with code snippets from historical bugs to generate more unusual programs [10]. Both have been evaluated on the deep learning libraries PyTorch and TensorFlow, finding 21 and 27 new bugs, respectively via crash oracles and differential testing.

We speculate that the effectiveness of these LLM-based approaches on fuzzing for WebGPU may be more limited, due to the much smaller amount of WebGPU data available for training and/or fine-tuning compared with that for PyTorch and TensorFlow, and the fact that some existing training data may already be obsolete due to recent changes to the WebGPU standard [58]. These issues would also apply to generating WGSL shaders for use in such WebGPU programs. Given that WebGPU also requires complex setups of object states and buffers in specific sequences for calls to be valid, it would be more challenging for LLMs to produce valid programs that meaningfully exercise downstream graphics APIs (roughly a third of the programs generated by TitanFuzz were valid [9]).

As discussed in Sections 4.1 and 6, the deployment of fuzzing can identify problems not only in software but also in language specifications. WebGlitch was able to detect an area of ambiguity in the WGSL specification, and similar issues have been found before by WGSLsmith [13] and other fuzzers for the programming language Dafny [16].

8 Conclusions and Future Work

We have introduced a new method for fuzzing the WebGPU API that generates sequences of API calls that are valid-by-construction, based on careful modelling of the requirements of API functions, and an on-demand generation approach. We have implemented this in a new open source tool, WebGlitch, which has helped to find bugs across the WebGPU ecosystem – in the WGSL specification, the Dawn and wgpu implementations of WebGPU, the Node.js and Deno JavaScript runtimes, and various downstream shader compilers – in a manner that would be difficult to achieve with manually written tests alone.

Avenues for future work include testing of Apple's WebGPU implementation in WebKit/Safari once it is in a more complete state, as well testing on a broader range of hardware configurations (the importance of this is evidenced by the fact that one of our bugs was specific to Apple GPUs). Currently, WebGlitch supports around 75% of all functions in the WebGPU API; adding support for the full API would improve the thoroughness of testing and might increase bug-finding ability. In the meantime, WebGlitch is ready for browser development teams in industry to integrate into their testing processes to help identify bugs as early as possible.

References

1 Anon. Automated tests fail on macOS 10.13, 2024. Accessed 23rd April 2025. URL: `https://github.com/gfx-rs/wgpu/issues/4632`.

2 Apple. Metal, 2024. Accessed 23rd April 2025. URL: `https://developer.apple.com/documentation/metal`.

3 Apple. Metal shading language specification, 2024. Accessed 23rd April 2025. URL: `https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf`.

4 Abhishek Arya and Oliver Chang. ClusterFuzz: Fuzzing at Google scale. In *Black Hat Europe 2019*, 2019. Accessed 23rd April 2025. URL: `https://i.blackhat.com/eu-19/Wednesday/eu-19-Arya-ClusterFuzz-Fuzzing-At-Google-Scale.pdf`.

5 Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. RESTler: stateful REST API fuzzing. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 748–758. IEEE / ACM, 2019. `doi:10.1109/ICSE.2019.00083`.

6 Lukas Bernhard, Nico Schiller, Moritz Schloegel, Nils Bars, and Thorsten Holz. DarthShader: Fuzzing WebGPU shader translators & compilers. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, pages 690–704. ACM, 2024. doi:10.1145/3658644.3690209.

7 T.Y. Chen, S.C. Cheung, and S.M. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, The Hong Kong University of Science and Technology, 1998.

8 Jon Davis. Release notes for Safari Technology Preview 185, 2024. Accessed 23rd April 2025. URL: https://www.webkit.org/blog/14885/release-notes-for-safari-technology-preview-185/.

9 Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 423–435. ACM, 2023. doi:10.1145/3597926.3598067.

10 Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 70:1–70:13. ACM, 2024. doi:10.1145/3597503.3623343.

11 Deno. Deno, 2024. Accessed 23rd April 2025. URL: https://github.com/denoland/deno.

12 Zhen Yu Ding and Claire Le Goues. An empirical study of OSS-fuzz bugs. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*, pages 131–142. IEEE, 2021. doi:10.1109/MSR52588.2021.00026.

13 Alastair F. Donaldson, Ben Clayton, Ryan Harrison, Hasan Mohsin, David Neto, Vasyl Teliman, and Hana Watson. Industrial deployment of compiler fuzzing techniques for two GPU shading languages. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2023, Dublin, Ireland, April 16-20, 2023*, pages 374–385. IEEE, 2023. doi:10.1109/ICST57152.2023.00042.

14 Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. *PACMPL*, 1(OOPSLA):93:1–93:29, 2017. doi:10.1145/3133917.

15 Alastair F. Donaldson, Hugues Evrard, and Paul Thomson. Putting randomized compiler testing into production (experience report). In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPIcs*, pages 22:1–22:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.ECOOP.2020.22.

16 Alastair F. Donaldson, Dilan Sheth, Jean-Baptiste Tristan, and Alex Usher. Randomised testing of the compiler for a verification-aware programming language. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2024, Toronto, ON, Canada, May 27-31, 2024*, pages 407–418. IEEE, 2024. doi:10.1109/ICST60714.2024.00044.

17 Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpinski. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, pages 1017–1032. ACM, 2021. doi:10.1145/3453483.3454092.

18 Benjamin Fyre. Test failures on Dx12/AMD, 2024. Accessed 23rd April 2025. URL: https://github.com/gfx-rs/wgpu/issues/6830.

19 Google. Dawn, 2024. Accessed 23rd April 2025. URL: https://github.com/google/dawn.

20 Google. Sanitizers, 2024. Accessed 23rd April 2025. URL: https://github.com/google/sanitizers.

21 Harrison Green and Thanassis Avgerinos. Graphfuzz: Library API fuzzing with lifetime-aware dataflow graphs. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1070–1081. ACM, 2022. doi:10.1145/3510003.3510228.

22 Kyriakos K. Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. FuzzGen: Automatic fuzzer generation. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2271–2287. USENIX Association, 2020. Accessed 23rd April 2025. URL: https://www.usenix.org/system/files/sec20-ispoglou.pdf.

23 Jianfeng Jiang, Hui Xu, and Yangfan Zhou. RULF: Rust library fuzzing via API dependency graph traversal. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 581–592. IEEE, 2021. doi:10.1109/ASE51524.2021.9678813.

24 Khronos Group. The OpenGL shading language, version 4.60.8, 2023. Accessed 23rd April 2025. URL: https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.60.pdf.

25 Khronos Group. Khronos Vulkan Registry, 2024. Accessed 23rd April 2025. URL: https://registry.khronos.org/vulkan/.

26 Khronos Group. SPIR-V specification, 2024. Accessed 23rd April 2025. URL: https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html.

27 Khronos Group. WebGL specification, 2024. Accessed 23rd April 2025. URL: https://registry.khronos.org/webgl/specs/latest/1.0/.

28 Bastien Lecoeur, Hasan Mohsin, and Alastair F. Donaldson. Program reconditioning: Avoiding undefined behaviour when finding and reducing compiler bugs. *Proc. ACM Program. Lang.*, 7(PLDI):1801–1825, 2023. doi:10.1145/3591294.

29 LLVM Compiler Infrastructure. libFuzzer – a library for coverage-guided fuzz testing, 2024. Accessed 23rd April 2025. URL: http://llvm.org/docs/LibFuzzer.html.

30 LLVM Project. llvm-cov - emit coverage information, 2024. Accessed 23rd April 2025. URL: https://llvm.org/docs/CommandGuide/llvm-cov.html.

31 Chenyang Lyu, Jiacheng Xu, Shouling Ji, Xuhong Zhang, Qinying Wang, Binbin Zhao, Gaoning Pan, Wei Cao, Peng Cheng, and Raheem Beyah. MINER: A hybrid data-driven approach for REST API fuzzing. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 4517–4534. USENIX Association, 2023. Accessed: 23rd April 2025. URL: https://www.usenix.org/system/files/usenixsecurity23-lyu.pdf.

32 William M. McKeeman. Differential testing for software. *Digit. Tech. J.*, 10(1):100–107, 1998. Accessed: 23rd April 2025. URL: http://www.dtjcd.vmsresource.org.uk/pdfs/dtj_v10-01_1998.pdf.

33 Mesa 3D project. LLVMpipe, 2024. Accessed 23rd April 2025. URL: https://docs.mesa3d.org/drivers/llvmpipe.html.

34 Microsoft. Effect-Compiler Tool, 2020. Accessed 23rd April 2025. URL: https://learn.microsoft.com/en-us/windows/win32/direct3dtools/fxc.

35 Microsoft. DirectX Shader Compiler, 2024. Accessed 23rd April 2025. URL: https://github.com/microsoft/DirectXShaderCompiler.

36 Microsoft Corporation. GPU-based validation and the Direct3D 12 debug layer, 2021. Accessed 23rd April 2025. URL: https://learn.microsoft.com/en-us/windows/win32/direct3d12/using-d3d12-debug-layer-gpu-based-validation.

37 Microsoft Corporation. DirectX graphics and gaming, 2022. Accessed 23rd April 2025. URL: https://learn.microsoft.com/en-us/windows/win32/directx.

38 Microsoft Corporation. HLSL specifications, 2024. Accessed 23rd April 2025. URL: https://github.com/microsoft/hlsl-specs.

39 David Neto and Alan Baker. Personal correspondence, July 2024.

40 NIST National Vulnerability Database. CVE-2016-2824, 2018. Accessed 23rd April 2025. URL: https://nvd.nist.gov/vuln/detail/CVE-2016-2824.

41 NIST National Vulnerability Database. CVE-2012-3968, 2020. Accessed 23rd April 2025. URL: https://nvd.nist.gov/vuln/detail/CVE-2012-3968.

42 NIST National Vulnerability Database. CVE-2017-5112, 2023. Accessed 23rd April 2025. URL: https://nvd.nist.gov/vuln/detail/CVE-2017-5112.

43 Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 75–84. IEEE Computer Society, 2007. doi:10.1109/ICSE.2007.37.

44 Hui Peng, Zhihao Yao, Ardalan Amiri Sani, Dave Tian, and Mathias Payer. GLeeFuzz: Fuzzing WebGL through error message guided mutation. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 1883–1899. USENIX Association, 2023. Accessed 23rd April 2025. URL: https://www.usenix.org/system/files/usenixsecurity23-peng.pdf.

45 Rust Graphics Mages. wgpu, 2024. Accessed 23rd April 2025. URL: https://github.com/gfx-rs/wgpu?tab=readme-ov-file.

46 Hynek Schlawack. Surprising consequences of macOS's environment variable sanitization, 2023. Accessed 23rd April 2025. URL: https://hynek.me/articles/macos-dyld-env/.

47 Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Address-Sanitizer: A fast address sanity checker. In Gernot Heiser and Wilson C. Hsieh, editors, *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012, Boston, MA, USA, June 13-15, 2012*, pages 309–318. USENIX Association, 2012. Accessed 23rd April 2025. URL: https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf.

48 Daniel Simols. wg-fuzz, 2024. Accessed 23rd April 2025. URL: https://github.com/wg-fuzz/wg-fuzz.

49 Brendon Tiszka, Chris Bookholt, and Matthew Riley. WebGPU technical report. Technical report, Google, 2023. Accessed 23rd April 2025. URL: https://chromium.googlesource.com/chromium/src/+/main/docs/security/research/graphics/webgpu_technical_report.md?pli=1#.

50 W3C group for GPU web standards. Conformance Test Suite, 2024. Accessed 23rd April 2025. URL: https://github.com/gpuweb/cts.

51 W3C group for GPU web standards. Implementation status, 2024. Accessed 23rd April 2025. URL: https://github.com/gpuweb/gpuweb/wiki/Implementation-Status.

52 Matthew Wong. Add extra example for operations involving abstract numerics, 2024. Accessed 23rd April 2025. URL: https://github.com/gpuweb/gpuweb/pull/4795.

53 Matthew Wong. WebGlitch, 2024. Accessed 23rd April 2025. URL: https://github.com/matthew-wong1/WebGlitch.

54 Matthew Wong. WebGPU adapter.info call fails after device creation, 2024. Accessed 23rd April 2025. URL: https://github.com/denoland/deno/issues/25874.

55 Matthew Wong. naga change i32 arithmetic operations to use wrapping instead of checked, 2025. Accessed 23rd April 2025. URL: https://github.com/gfx-rs/wgpu/pull/6835.

56 Matthew Wong. WebGlitch artifact, 2025. Accessed 25th April 2025. doi:10.5281/zenodo.15281221.

57 World Wide Web Consortium. WebGPU explainer draft community group report, 2024. Accessed 23rd April 2025. URL: https://gpuweb.github.io/gpuweb/explainer/.

58 World Wide Web Consortium. WebGPU W3C working draft, 2024. Accessed 23rd April 2025. URL: https://www.w3.org/TR/webgpu/.

59 World Wide Web Consortium. WebGPU shading language W3C working draft, 4 September 2024, 2024. Accessed 23rd April 2025. URL: https://www.w3.org/TR/2024/WD-WGSL-20240904/.

60 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011. doi:10.1145/1993498.1993532.

Scaling Up: Revisiting Mining Android Sandboxes at Scale for Malware Classification

Francisco Handrick Tomaz da Costa
Computer Science Department, University of Brasília, Brazil
The Environmental Sanitation Company of the Federal District, Brasília, Brazil

Ismael Medeiros
Computer Science Department, University of Brasília, Brazil

Leandro Oliveira
Computer Science Department, University of Brasília, Brazil

João Calássio
Computer Science Department, University of Brasília, Brazil

Rodrigo Bonifácio
Computer Science Department, University of Brasília, Brazil

Krishna Narasimhan
F1RE, Amsterdam, The Netherlands

Mira Mezini
TU Darmstadt, Germany
hessian.AI, Darmstadt, Germany
National Research Center for Applied Cybersecurity ATHENE, Darmstadt, Germany

Márcio Ribeiro
Federal University of Alagoas, Brazil

Abstract

The widespread use of smartphones in daily life has raised concerns about privacy and security among researchers and practitioners. Privacy issues are generally highly prevalent in mobile applications, particularly targeting the Android platform – the most popular mobile operating system. For this reason, several techniques have been proposed to identify malicious behavior in Android applications, including the Mining Android Sandbox approach (MAS approach), which aims to identify malicious behavior in repackaged Android applications (apps). However, previous empirical studies evaluated the MAS approach using a small dataset consisting of only 102 pairs of original and repackaged apps. This limitation raises questions about the external validity of their findings and whether the MAS approach can be generalized to larger datasets. To address these concerns, this paper presents the results of a replication study focused on evaluating the performance of the MAS approach regarding its capabilities of correctly classifying malware from different families. Unlike previous studies, our research employs a dataset that is an order of magnitude larger, comprising 4,076 pairs of apps covering a more diverse range of Android malware families. Surprisingly, our findings indicate a poor performance of the MAS approach for identifying malware, with the F1-score decreasing from 0.90 for the small dataset used in the previous studies to 0.54 in our more extensive dataset. Upon closer examination, we discovered that certain malware families partially account for the low accuracy of the MAS approach, which fails to classify a repackaged version of an app as malware correctly. Our findings highlight the limitations of the MAS approach, particularly when scaled, and underscore the importance of complementing it with other techniques to detect a broader range of malware effectively. This opens avenues for further discussion on addressing the blind spots that affect the accuracy of the MAS approach.

2012 ACM Subject Classification Security and privacy → Malware and its mitigation

Keywords and phrases Android Malware Detection, Dynamic Analysis, Mining Android Sandboxes

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.40

© Francisco Handrick Tomaz da Costa, Ismael Medeiros, Leandro Oliveira, João Calássio, Rodrigo Bonifácio, Krishna Narasimhan, Mira Mezini, and Márcio Ribeiro;
licensed under Creative Commons License CC-BY 4.0

Category Replication Paper

Supplementary Material *Software*: `https://github.com/droidxp/paper-ecoop-results`
archived at `swh:1:dir:b12f77f05adfb2fcef252b97252590a872cf7eb4`

Funding This work was partially supported by CNPq (grant 314834/2021-4) and INES (Brazilian National Institute of Science and Technology for Software Engineering).

Acknowledgements We thank all anonymous reviewers for their insightful comments.

1 Introduction

Mobile technologies, such as smartphones and tablets, have become fundamental to how we function as a society. Almost two-thirds of the world population uses mobile technologies [10, 33], with the Android Platform dominating this market and accounting for more than 70% of the *mobile market share*, with almost 2.5 million Android applications [1] (apps) available on the Google Play Store, in June 2023 [44]. As popularity rises, so does the risk of potential attacks, prompting collaborative efforts from both academia and industry to design and develop new techniques for identifying malicious behavior or vulnerable code in Android apps [45]. One popular class of Android malware is based on repackaging [6, 25], where a benign version of an app is infected with malicious code, e.g., to broadcast sensitive information to a private server [28], and subsequently shared with users using even official app stores.

The Mining Android Sandbox approach (MAS approach) was initially designed to construct sandboxes based on exploratory calls to sensitive APIs [20]. The MAS approach operates in two distinct phases. In the first phase (exploratory phase), automated test case generation tools are utilized to abstract the behavior of an app, focusing on recording calls to sensitive APIs (Application Programming Interfaces). Subsequently, during normal app execution (exploitation phase), the generated sandbox blocks any calls to sensitive APIs that were not observed during the exploratory phase. Prior studies [6, 13] have investigated the effectiveness of the MAS approach in detecting potential malicious behavior in repackaged apps. These studies have also conducted comparisons of the approach's performance by employing different test case generation tools during the exploratory phase, including Monkey [18], DroidBot [30], and Droidmate [22] – bringing evidence that DroidBot outperforms other test generation tools, uncovering many potential malicious behaviors.

Nonetheless, these previous studies have two main limitations. First, they use a small dataset of malware comprising only 102 pairs of original/repackaged versions of an app – which might compromise external validity. Second, their assessments do not investigate the impact of relevant features of the repackaged apps on the accuracy of the MAS approach for malware classification, including (a) whether or not the repackaged version is a malware, (b) the similarity between the original and the repackaged versions of an app, and (c) the malware family [2] when the repackaged version of an app is a malware. These limitations compromise a broader understanding of the MAS approach performance. We present more details about the MAS approach and related work in Section 2.

[1] In this paper, we will use the terms Android Applications, Android Apps, and Apps interchangeably, to refer to Android software applications

[2] Malware families (such as *gappusin*, *kuguo*, *dowgin*, etc.) are often used to classify malware in groups that share similar codebases, attack methods, and objectives [4].

To better understand the impact of these issues on previously published results, this paper presents a replication of the study conducted by Bao et al. [6]. We aim to verify the original study's findings by executing the test case generation tool DroidBot [30] in the same settings as the original research. Unlike the original study, here we use a curated dataset of app pairs (original/repackaged versions) significantly larger than the dataset used in Bao et al.'s study. Our new dataset contains 4,076 pairs of original and repackaged apps. We present more details about the datasets, data collection, and analysis procedures in Section 3.

Negative result. Our study reveals a significantly lower accuracy (F1-score of 0.54) of the MAS approach in comparison to what the MAS approach approach performs in the small dataset (F1-score of 0.90). Since an accuracy of 0.54 is unsatisfactory for a trustworthy malware classification technique, we conduct a set of experiments to understand the reasons for the lower accuracy in our dataset. Our further assessments reveal that the MAS approach fails to correctly classify most samples from a specific set of malware families, particularly those from the *gappusin* family (a particular adware class frequently appearing in repackaged apps). Out of the total of 1,337 samples within this family in our large dataset, the MAS approach failed to classify 1,170 samples as malware correctly. Accordingly, these families are responsible for substantially reducing the recall of the MAS approach. We detail the results of our experiments in Section 4. We also discuss the implications and possible threats to the validity of our study in Section 5 and present some final remarks in Section 6. The main artifacts we produced during this research are available in the paper repository.

`https://github.com/droidxp/paper-ecoop-results`

2 Background and Related Work

There are many tools available that help developers reverse engineer Android bytecode [47]. For this reason, software developers can easily decompile trustworthy apps, modify their contents by inserting malicious code, repackage them with malicious payloads, and re-publish them in app stores, including official ones like the Google Play Store. It is well-known that repackaged Android apps can leverage the popularity of real apps to increase their propagation and spread malware [28]. As an example, in 2016 a repackaged version of the famous Pokémon Go app was discovered less than 72 hours after the game was officially released in the United States, Australia, and New Zealand [28]. The repackaged version, originated from an unofficial app store, gained full control over the victim's phone, obtaining access to main functions such as the phonebook, audio recorder, and camera.

Repackaging has been raised as a noteworthy security concern in Android ecosystem by stakeholders in the app development industry and researchers [23]. Indeed, there are reports claiming that about 25% of Google Play Store app content correspond to repackaged apps [46]. Nevertheless, all the workload to detect and remove malware from markets by the stores (official and non-official ones), have not been accurate enough to address the problem. As a result, repackaged Android apps threaten security and privacy of unsuspicious Android app users, beyond compromising the copyright of the original developers [24]. Aiming at mitigating the threat of malicious code injection in repackaged apps, several techniques based on both static and dynamic analysis of Android apps have been proposed, including the MAS approach for malware classification [20, 6].

2.1 Mining Android Sandboxes

A *sandbox* is a well-known mechanism to secure a system and forbid a software component from accessing resources without appropriate permissions. Sandboxes have also been used to build an isolated environment within which applications cannot affect other programs, the network, or other device data [32]. The idea of using sandboxes emerged from the need to test unsafe software, possible malware, without worrying about the integrity of the device under test [7], shielding the operating system from security issues. To this end, a sandbox environment should have the minimum requirements to run the program, and make sure it will never assign the program more privileges than it should have, respecting the *least privilege* principle. Within the Android ecosystem, sandbox approaches ensure the principle of the *least privilege* by preventing apps from having direct access to resources like device hardware (e.g., GPS, Camera), or sensitive data from other apps. Access to sensitives data like contact list or resources are granted through specific APIs, known as sensitive APIs, which are managed by coarse-grained Android permissions system [12].

The MAS approach [20] aims at automatically building a sandbox through dynamic analysis (i.e., using automatic test generation tools). The main idea is to grant permissions to an app based on its calls to sensitive APIs. Thus, sandboxes build upon these calls to create safety rules and then block future calls to other sensitive resources, which diverge from those found in the first exploratory phase. Using the Droidmate test generation tool [21], Jamrozik et al. proposed a full-fledged implementation of the MAS approach, named Boxmate [20]. Boxmate records the occurrences of calls to sensitive APIs and, optionally, the UI events (e.g., a button click) that trigger these calls. Therefore, it is possible to configure Boxmate to record events associated with each sensitive call as tuples (event, API), instead of recording just the set of calls to sensitive APIs. Jamrozik et al. argue that, in this way, Boxmate generates finer grain results, which might improve the accuracy of the MAS approach – even with the presence of reflection, a feature commonly used in malicious apps [29].

In fact, the MAS approach can be implemented using a mix of static and dynamic analysis. In the first phase, one can instrument an Android app to log any call to the Android sensitive methods. After that, one can execute a test case generation tool (such as DroidMate, DroidBot, or Monkey) to explore the app behavior at runtime, while the calls to sensitive APIs are recorded. This set of calls to sensitive APIs is then used to configure the sandbox. The general MAS approach suggests that the more efficient the test generation tool (for instance, code coverage), the more accurate the resulting sandbox would be.

2.2 Mining Android Sandbox for Malware Classification

Besides being used to generate Android sandboxes, the MAS approach can also be used to detect if a repackaged version of an Android app contains an unexpected (perhaps malicious) behavior [6]. In this scenario, the *effectiveness* of the approach is estimated in terms of the accuracy in which malicious behavior is correctly identified in the repackaged version of the apps.

The MAS approach for malware classification typically works as follows. In a first step (**instrumentation phase**), a tool instruments the code of the apps (both original and repackaged versions) to collect relevant information during the apps execution in later stages. Then, in a second step (**exploration phase**), the MAS approach collects a set S_1 with all calls to sensitive APIs the original version of an app executes while running a test case generator tool (like DroidBot). In the third step (**exploitation phase**), the MAS approach (a) collects a set S_2 with all calls to sensitive APIs the repackaged version of an app executes

while running a test case generator tool, and then (b) computes the set $S = S_2 \setminus S_1$ and checks whether S is empty or not. The MAS approach classifies the repackaged version as a malware whenever $|S| > 0$.

Previous research works reported the results of empirical studies that aim to investigate the effectiveness of the MAS approach for malware classification [6, 11]. For instance, Bao et al. found that, in general, the sandboxes constructed using test generation tools classify at least 66% of repackaged apps as malware in a dataset comprising 102 pairs of apps (original/repackaged versions) [6]. Actually, the mentioned work performed two studies: one pilot study involving a dataset of 10 pairs of apps (`SmallE`), in which the authors executed each test case generation tools for one hour; and a larger experiment (`LargeE`) involving 102 pairs of apps in which the authors executed the test case generation tools for one minute [6].

The authors also presented that, among five test generation tools used, DroidBot [30] leads to the most effective sandbox. Le et al. extend the MAS approach for malware classification with additional verification, such as the values of the actual parameters used in the calls to sensitive APIs [25], while Costa et al.[13] investigated the impact of static analysis to complement the accuracy of the MAS approach for malware classification. Their study reports that DroidFax [8], the static analysis infrastructure used in [6], classifies as malware almost half of the repackaged apps.

2.3 Android Malware Classification

The field of malware detection for the Android platform is fertile, with a significant number of secondary studies already published [41, 26, 48, 49]. In general, malware detection techniques are divided into static detection, dynamic detection, and hybrid detection [9]. Several studies have also conducted surveys on malware detection techniques and presented a review of them [31, 45, 35]. For instance, M. Odusami et al. [35] discuss various static analyses approaches that have been used in the literature to identify malicious behavior in Android apps. The authors present some works with permission and signature-based malware detection systems. They highlight that both approaches have a low false positive rate; however, they are very ineffective in detecting new malware. Although they could reveal possible malicious behaviors, the authors discuss several limitations of these approaches, as they are limited regarding code obfuscation and dynamic code loading.

The literature also presents surveys based on dynamic analysis, where malicious behavior is analyzed at runtime, exposing risks that are not detected by static analysis. As a malicious app "is alive", dynamic analysis adds another degree of analysis since it observes how Android apps interacts with the environment. However, if applied inappropriately, it may provide limited code coverage, which repeated executions can improve. Therefore, dynamic analysis's time cost and computation resources are higher when compared with static analysis. K. Tam et al. [45] presented several dynamic analysis studies based on Android architectural layers. The survey also exposed that dynamic analysis can be performed in emulator environments, real devices, or both. The authors discuss that the choice of environments is an important issue for analysis, as there are malware families that can detect emulated environments and do not exhibit malicious behaviors [42]. Finally, K. Tam et al. also exposed some works based on hybrid malware detectors and claim that these methods can increase code coverage and robustness, taking advantage of the best of each technique to find malicious behaviors.

Several studies have also explored Android malware detection approaches based on machine learning (ML) techniques, employing both static and dynamic analyses to extract features and train ML models [31]. Most of these approaches have demonstrated high accuracy (above 90%), effectively detecting previously unseen malware families with low

false positives rate [1]. However, some studies have identified limitations of machine learning approaches for Android malware classification. In their work, K. Liu et al. [31] highlighted challenges related to machine learning techniques, identifying several factors that could lead to biased results, such as the quality of the sample set. The authors argue that samples of poor quality, with a non-representative size or outdated samples, may yield promising results in experimental settings but might not perform similarly in a real environment [31]. Another critical aspect is the quality of the extracted feature dataset. The efficacy of machine learning approaches heavily relies on the selection of correct features and their extraction methods, particularly dynamic features. Moreover, in addition to the computational costs involved, other studies [14, 17] have indicated that machine learning approaches exhibit weaknesses when dealing with malicious apps that alter their behavior to mislead learning algorithms, which may restricts their applicability in real-world scenarios.

3 Experimental Setup

This research aims to develop a deeper understanding of the performance of the MAS approach for detecting malware. To this end, in this paper, we **replicate** the study by Bao et al. [6], which advocates for using the MAS approach for malware classification. However, in contrast to the original study [6], we use a dataset of repackaged apps that is an order of magnitude larger. Accordingly, we investigate the following research questions:

(RQ1) What is the impact of considering a larger and diverse dataset on the accuracy of the MAS approach for malware classification? Answering this question may help shed light on potential generalization issues in previous studies that empirically assess the MAS approach approach to malware classification.

(RQ2) What is the influence of the similarity between the original and repackaged versions of the apps on the performance of the MAS approach for malware classification? Answering this research question helps clarify whether the similarity between an original app and its repackaged version affects the MAS approach approach's performance in malware classification.

(RQ3) What is the influence of the malware family (e.g., *gappusin*, *kuguo*, *dowgin*) on the performance of the MAS approach for malware classification? Answering this research question may help identify potential blind spots in the MAS approach approach to malware classification, revealing possible extensions that could improve the detection of specific malware families.

In this section, we describe our study settings. First, we present our procedures to create our datasets (Section 3.1). Then, we describe the data collection and data analysis procedures (Sections 3.2 and 3.3).

3.1 Malware Dataset

To address our research questions, we contribute a dataset designed to meet two primary requirements. First, it should provide a comprehensive and up-to-date selection of Android repackaged apps. By "comprehensive", we mean at least an order of magnitude larger than the dataset used in the original study [6]. Given its comprehensiveness, we expect it to include a diverse range of malware families to ensure representativeness. Second, our dataset should be properly labeled, ensuring each sample includes key attributes such as similarity and malware family. This is particularly necessary to answer research questions RQ2 and RQ3.

3.1.1 Procedures for Building the Dataset

We curate our dataset in three main phases. In the first phase, we use two repositories of repackaged Android apps (RePack [28] and AndroMalPack [37]) to build the dataset we use in our research. RePack was curated using automatic procedures that extract repackaged apps from the Androzoo repository [2]. It comprises 18,073 apps, from which 2,776 are original versions of an app and the remaining ones are repackaged. RePack contains 15,297 pairs of original and repackaged Android apps, many repackaged versions of the same original app may coexist within the RePack dataset – note that all repackaged variants of a given app are derived from the same original version, as confirmed by their matching hash identifier. RePack is the leading dataset used in Android repackaged research [23], even though it only contains packages built until 2018. For this reason, we decided to include samples from the AndroMalPack dataset collected after 2018 in our research. Unlike RePack, AndroMalPack lacks information about the original apps, leading us to follow an existing heuristic [28] to identify the original versions of its repackaged apps, leading to a sample from the AndroMalPack dataset that contains 1,190 pairs (original/repackaged) of apps, all pairs satisfying our constraint of being collected after 2018. Altogether, our initial dataset contains a total of 16,487 pairs of apps.

In the second phase, we discarded some samples from our initial dataset because, during the execution of our experiments, we encountered recurrent issues related to the instrumentation of the apps using DroidFax [8]. Other problems occurred after the execution of the apps in the Android emulator, while analyzing the apps or their execution logs. More precisely, we encountered failures while instrumenting 919 original apps from our initial dataset, including both RePack and AndroMalPack. After removing these original apps from our dataset, we were left with 5,875 pairs (original/repackaged) of apps. Among these pairs, 430 repackaged apps could not be instrumented. Failures also occurred while analyzing either the original or repackaged version of 586 apps, resulting in a dataset containing 4,742 pairs. Failures at this phase were expected, as some malware samples employ evasion tactics, such as deliberately crashing test apps in simulated environments, to avoid detection [3]. Finally, we could not install five apps in the version of the Android emulator (API level 28) we used in our research. Compared to our experience building our dataset, a more significant percentage of failures has been reported in previous research [6]. Note that we did not apply any filters to increase the representation of certain malware families in our dataset.

Third, we queried the `VirusTotal` repository to identify original versions of apps labeled as malware. Samples with such labels were excluded from our dataset, as the MAS approach assumes that the original version of an app is not malware (otherwise, the repackaged versions might also exhibit malicious behavior). `VirusTotal` is a widely recognized tool that scans software assets, including Android apps, using over 60 antivirus engines [23]. Thus, we excluded 661 samples from our dataset that do not satisfy this constraint.

In the end, we are left with our final dataset (hereafter `LargeDS`) of 4,076 apps which we use in our study. To bring evidence that we were able to reproduce the results of previous research, we also consider in our research a small dataset (`SmallDS`) used in the original study [6]. This is the same dataset referenced in Section 2.2 as (`LargeE`).

3.1.2 Features of the Datasets

We queried the `VirusTotal` repository to find out which repackaged apps in our dataset have indeed been labeled as a malware. According to `VirusTotal`, in the `SmallDS` (102 pairs), 69 of the repackaged apps (67.64%) were identified as malware by at least two security engines.

Here, we consider a repackaged version of an app to be malware only if `VirusTotal` reports that at least two security engines identify malicious behavior within the asset. Although this decision aligns with previous research [50, 23], we assess its potential impact on our findings in Section 5. Considering the `LargeDS`, at least two security engines identified 2,895 out of the 4,076 repackaged apps as malware (71.02%). Again, in Section 5, we show that our results remain consistent across three additional scenarios: classifying a repackaged version of an app as malware if at least one, five, or ten `VirusTotal` engines flag it as malicious.

Classifying malware into different categories is a common practice. For instance, Android malware can be classified into categories like riskware, trojan, adware, etc. Each category might be further specialized in several malware families, depending on its characteristics and attack strategy – e.g., steal network info (IP, DNS, WiFi), collect phone info, collect user contacts, send/receive SMS, and so on [38]. According to the `avclass2` tool [40], the malware samples in the `SmallDS` come from 17 different families – most of them from the Kuguo (49.27%) and Dowgin (17.39%) families. Our `LargeDS`, besides comprising a large sample of repackaged apps (4,076 in total), contains 116 malware families – most of them from the Gappusin (46.18%) family. Despite being flagged as malicious by at least two security engines, unfortunately `avclass2` tool cannot correctly identify the family of 253 samples in our `LargeDS`.

We also characterize our dataset according to the similarity between the original and repackaged versions of the apps, using the SimiDroid tool [27]. SimiDroid quantifies the similarity based on (a) the methods that are either identical or similar in both versions of the apps (original and repackaged versions), (b) methods that only appear in the repackaged version of the apps (new methods), and (c) methods that only appear in the original version of the apps (deleted methods). Our `LargeDS` has an average similarity score of 90.39%, with the following distribution: 87 app pairs have a similarity score below 25%, 49 pairs fall between 25% and 50%, 353 apps between 50% and 75%, and 3,587 apps exceed 75%. The `SmallDS` has an average similarity score of 89.41%.

After executing our experiments, we identified the most frequently abused sensitive APIs called by the repackaged version of our samples. We observed that upon execution of all samples from our dataset (`SmallDS` and `LargeDS`), malicious app versions injected 133 distinct methods from sensitive APIs (according to the AppGuard [5] security framework). Malicious code often exploits these APIs to compromise system security and access sensitive data. Table 1 lists the 10 most frequently called methods from sensitive APIs that appear only in the repackaged versions of the apps.

We must highlight that the `LargeDS` samples come from different Android app stores. Most of our repackaged apps come from a non-official Android app store, Anzhi [34]. However, some repackaged apps also come from the official Android app store, Google Play.

3.2 Data Collection Procedures

We take advantage of the DroidXP infrastructure [11] for data collection. DroidXP allows researchers to compare test case generation tools for malicious app behavior identification, using the MAS approach. Although the comparison of test case generation tools is not the goal of this paper, DroidXP was still useful for automating the following steps of our study.

(Step1) **Instrumentation**: In the first step, we configure DroidXP to instrument all pairs of apps in our datasets (`SmallDS` and `LargeDS`). Here, we instrument both versions of the apps (as APK files) to collect relevant information during their execution. Under the hood, DroidXP leverages DroidFax to instrument the apps and collect static information about them. To improve the performance across multiple executions, this phase executes only once for each version of the apps in our dataset.

Table 1 Sensitive APIs that frequently appear in the repackaged versions of the apps. The *Occurrences* column gives the number of distinct repackaged apps that introduce a call to a sensitive method.

Method of Sensitive API	Occurrences
android.telephony.TelephonyManager: int getPhoneType()	311
android.telephony.TelephonyManager: java.lang.String getNetworkOperatorName()	297
android.location.LocationManager: java.lang.String getBestProvider(android.location.Criteria,boolean)	292
android.telephony.TelephonyManager: int getSimState()	284
java.lang.reflect.Field: java.lang.Object get(java.lang.Object)	277
android.net.NetworkInfo: java.lang.String getTypeName()	271
android.database.sqlite.SQLiteDatabase: android.database.Cursor query(java.lang.String,java.lang.String[],...,...,...,...,...)	270
java.lang.reflect.Field: int getInt(java.lang.Object)	250
android.net.wifi.WifiInfo: java.lang.String getMacAddress()	238
android.telephony.TelephonyManager: java.lang.String getNetworkOperator()	237

(Step2) **Execution**: In this step, DroidXP first installs the (instrumented) version of the APK files in the Android emulator we use in our experiment (API 28) and then starts a test case generation tool for executing both app versions (original and repackaged). We execute the apps via DroidBot [30], mainly because the original research we replicate here reports that DroidBot leads to the best accuracy of the MAS approach for malware identification. Since previous studies suggest that DroidBot's coverage nearly reaches its maximum within one minute [6], we run each app for three minutes. To mitigate the randomness inherent in test case generation tools, we repeat this process three times. To also ensure that each execution gets the benefit of running on a fresh Android instance without biases that could stem out of history, DroidXP wipes out all data stored on the emulator that has been collected from previous executions.

(Step3) **Data Collection**: After the execution of the instrumented apps, once again, DroidXP leverages DroidFax, this time to collect all relevant information (such as calls to sensitive APIs, test coverage metrics, and so on). We use this information to analyze the performance of the MAS approach for detecting malicious behavior.

3.3 Data Analysis Procedures

We consider that the MAS approach builds a sandbox that labels a repackaged version of an app as malware if there is at least one call to a sensitive API that (a) was observed while executing the repackaged version of the app and that (b) was not observed while executing the original version of the same app. If the set of sensitive methods that only the repackaged version of an app calls is empty, we conclude that the sandbox does not label the repackaged version of an app as malware. The set of sensitive APIs we use was defined in the AppGuard framework [5], which was based on the mapping from sensitive APIs to permissions proposed by Song et al. [16]. We triangulate the results of the MAS approach classification with the outputs of `VirusTotal`, which might lead to one of the following situations:

- **True Positive (TP)**. The MAS approach labels a repackaged version as malware and, according to `VirusTotal`, at least two security engines label the asset as a malware.
- **True Negative (TN)**. The MAS approach does not label a repackaged version as malware and, according to `VirusTotal`, at most one security engine labels the asset as a malware.
- **False Positive (FP)**. The MAS approach labels a repackaged version as malware and, according to `VirusTotal`, at most one security engine labels the asset as a malware.
- **False Negative (FN)**. The MAS approach does not label a repackaged version as malware, and according to `VirusTotal`, at least two security engines label the asset as a malware.

In Section 4 we compute *Precision*, *Recall*, and *F-measure* (F_1) from the number of true-positives, false-positives, and false-negatives (using standard formulae). We use basic statistics (average, median, standard deviation) to identify the accuracy of the MAS approach for malware classification, using both datasets – i.e., the `SmallDS` with 102 pairs of apps and `LargeDS` with 4,076 pairs. We use the Spearman Correlation [43] method and Logistic Regression [19] to understand the strengths of the associations between the similarity index between the original and the repackaged versions of a malware with the MAS approach accuracy – that is, if the approach was able to classify an asset as malware correctly. We also use existing tools to reverse engineer a sample of repackaged apps in order to better understand the (lack of) accuracy of the MAS approach.

Table 2 highlights the differences between the original study [6] and our replication study. In the best-case scenario, where no re-executions are required, our experiment would take at least 611 hours. In contrast, under the best conditions, the original experiment's execution would last 60 hours. This difference is one of the reasons we focus our research on DroidBot, the test case generation tool that demonstrated the best performance in the original study.

Table 2 Characterization of this replication study.

Study Feature	Original Study	Replication Study
Dataset	102 pairs of samples	4,076 pairs of samples
Execution time	One minute	Three minutes
Number of executions	Single execution	Three executions
Metrics	Malware prevalence	Precision, Recall, and F1-score
Test case generation tool	Six different tools (DroidBot with best performance)	DroidBot only

4 Results

In this section, we detail the findings of our study. We remind the reader that this replication study's primary goal is to better understand the strengths and limitations of the MAS approach for malware detection by replicating the work of Bao et al., using DroidBot as the test case generation tool. We explore the results of our research using two datasets: the `SmallDS` (102 app pairs), and `LargeDS` (4,076 pairs).

4.1 Exploratory Data Analysis of Accuracy

`SmallDS`. Considering the `SmallDS` (102 apps), the MAS approach for malware detection classifies a total of 69 repackaged versions as malware (67.64%). This result is close to what Bao et al. reported [6]. That is, in their original paper, the MAS approach using DroidBot classifies 66.66% of the repackaged version of the apps as malware [6]. This result confirms that we could reproduce the findings of the original study using our implementation settings of the MAS approach.

Finding 1. We were able to reproduce the results of existing research using our implementation of the MAS approach, achieving a malware classification in the `SmallDS` close to what has been reported in previous studies.

In the original study [6], the authors assume that all repackaged versions are malware and contain a malicious code. For this reason, the authors do not explore accuracy metrics (such as Precision, Recall, and F-measure (F_1)) – all repackaged apps labeled as malware are considered true positives in the original study. As we mentioned, in this paper we take advantage of `VirusTotal` to label our dataset and build a ground truth: In our datasets, we classify a repackaged version of an app as malware if, according to our `VirusTotal` query results, at least two security engines identify malicious behavior in the asset. This decision aligns with existing recommendations [50, 23]). The first row of Table 3 shows that the MAS approach achieves an accuracy of 0.90 when considering the `SmallDS`. Nonetheless, the MAS approach fails to classify seven assets as malware on the `SmallDS` correctly (FN column, first row of Table 3), and wrongly labeled the repackaged version of six apps as malware (FP column).

Table 3 Accuracy of the MAS approach in both datasets.

Dataset	TP	FP	FN	Precision	Recall	F_1
`SmallDS` (102)	63	6	7	0.91	0.90	0.90
`LargeDS` (4,076)	1,175	220	1,720	0.84	0.40	0.54

`LargeDS`. Surprisingly, considering our complete dataset (4,076 apps), the MAS approach labels a total of 1,395 repackaged apps as malware (34.22% of the total number of repackaged apps) – for which the repackaged version calls at least one additional sensitive API. Our analysis also reveals a **negative result** related to the accuracy of the approach: here, the accuracy is much lower in comparison to what we reported for the `SmallDS` (see the second row of Table 3): F_1 dropping from 0.90 to 0.54. This result indicates that, when considering a large dataset, the accuracy of the MAS approach using DroidBot drops significantly.

Finding 2. The MAS approach for malware detection leads to a substantially lower performance on the `LargeDS` (4,076 pairs of apps), dropping F1-score from 0.90 to 0.54 in comparison to what we observed in the `SmallDS`.

Therefore, the resulting sandbox we generate using DroidBot suffers from a significantly low accuracy rate when considering a large dataset. This is shown in the second row of Table 3. The negative performance of the MAS approach in the `LargeDS` encouraged us to endorse efforts to identify potential reasons for this phenomenon and motivated us to explore the research questions RQ2 and RQ3.

4.2 Assessment Based on Similarity Score

Figure 1 shows the Similarity Score distribution over the `LargeDS` we use in our research. Recall that the Similarity Score measures how similar an app's original and repackaged versions are. The complete dataset averages a Similarity Score of 0.90 (with a median of 0.98 and standard deviation of 0.18).

In this section we investigate how the Similarity Score influences the accuracy of the MAS approach – which might help us understand if it might explain the low small performance of the MAS approach in the `LargeDS`. To this end, we leverage Logistic Regression to quantify the relationship between Similarity Score and F1-score. This analysis excludes instances of true negatives (i.e., cases where the repackaged version is benign according to `VirusTotal` and the MAS approach correctly labels it as benign). As such, we test the following null hypothesis:

H_0 Similarity Score does not influence the accuracy of the MAS approach for malware detection.

The logistic regression results suggest that we should reject our null hypothesis (p-value $= 2.22 \cdot 10^{-16}$). This finding indicates that the accuracy of the MAS approach on `LargeDS` is influenced by the similarity between the original and repackaged versions of an app.

Finding 3. There is an association between the Similarity Score and the MAS approach performance, which means that the similarity between the original and repackaged versions of an app can explain the performance of the MAS approach for malware classification.

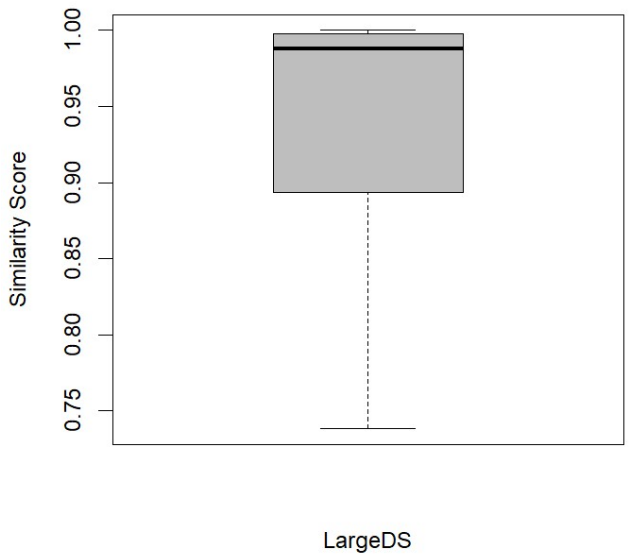


Figure 1 Similarity Score of the malware samples in the `LargeDS`. The boxplots in the figure do not show outliers.

To clarify the association between Similarity Score and accuracy, we use the *K-Means* algorithm to split the `LargeDS` into ten clusters – according to the Similarity Score. We then estimate the percentage of correct classifications for each cluster, as shown in Table 4. Note that the MAS approach achieves the highest percentage of correct classification (77.35%) for the second cluster (cId = 2), which presents an average Similarity Score of (0.56). Nonetheless, the cluster cId = 10, with a larger number of samples (1,302) and Similarity Score (0.99), presents a percentage of correct classifications of 26.5%. We can observe that as the average similarity rate decreases, there is a tendency toward greater accuracy in the MAS approach. Hence, the average similarity score could explain the poor performance of the MAS approach on the `LargeDS`, especially considering that most samples exhibit a high average similarity of 0.99.

4.3 Assessment Based on Malware Family

As we discussed in the previous section, the similarity assessment partially explains the low performance of the MAS approach on the `LargeDS`. Since the `LargeDS` covers a wide range of malware families, we investigate the hypothesis that the diversity of malware families in the `LargeDS` also contributes to the poor performance of the MAS approach on the `LargeDS`. Indeed, in the `LargeDS`, we identified a total of 116 malware families, though the most frequent ones are *gappusin* (1,337 samples), *revmob* (207 samples), *dowgin* (183 samples) and *airpush* (120 samples). Together, they account for 63.79% of the repackaged apps in our `LargeDS` labeled as malware according to `VirusTotal`.

This family distribution in the `LargeDS` is different from the family distribution in the `SmallDS` (used in the original study) – where the families *kuguo* (34 samples), *dowgin* (12 samples), and *youmi* (5 samples) account for 73.91% of the families considering the 69

Table 4 Characteristics of the clusters. Note there is a specific pattern associating the percentage of correct answers with the Similarity Score. For this analysis, we removed the true negatives in our dataset.

cId	Similarity Score	Samples	Correct Answers	%
1	0.42	42	30	71.42
2	0.56	181	140	77.35
3	0.68	131	98	74.81
4	0.80	170	104	61.18
5	0.88	263	83	31.56
6	0.91	236	129	54.66
7	0.95	167	51	30.54
8	0.97	150	67	44.67
9	0.98	421	112	26.60
10	0.99	1302	345	26.50

repackaged apps in the `SmallDS` for which `VirusTotal` labels as malware. Most important, in the `SmallDS`, there is just one sample from the *gappusin* family and no sample from *revmob* family, two of the most frequent families in our `LargeDS`. This observation leads us to the question: how does the MAS approach perform when considering only samples from the *gappusin* and *revmob* families?

The confusion matrix of Table 5 summarizes the accuracy assessment of the MAS approach considering only the *gappusin* and *revmob* samples in the `LargeDS`. To make clear, `VirusTotal` classifies as malware all repackaged versions in the *gappusin* and *revmob* family. It is worth noting that the MAS approach failed to classify correctly 1,170 (87.5%) samples of *gappusin* as malware. Similarly, 92 samples (44.44%) from *revmob* were not classified as malware. Furthermore, if we exclude the *gappusin* and *revmob* samples from the `LargeDS`, the recall of the MAS approach increases to 0.72, which, although improved, remains relatively low compared to the original studies.

Table 5 Confusion matrix of the MAS approach when considering only the samples from the *gappusin* and *revmob* family in the `LargeDS`.

Actual Condition	Predicted Condition	
	Benign	Malware
Benign (0)	TN (0)	FP (0)
Gappusin (1,337)	FN (1,170)	TP (167)
Revmob (207)	FN (92)	TP (115)

Finding 4. The MAS approach fails to correctly identify 87.50% of the samples from the *gappusin* family and 44.44% of the samples from the *revmob* as malware. Just like the Similarity Score, the presence of some malware families with a high false negative rate also influences the low recall of the MAS approach in the `LargeDS`

We further analyze the samples from the *gappusin* and *revmob* malware families in our dataset, given their relevance to the negative results presented in our paper. First, we examined the Similarity Score of the samples. Figure 2 shows a histogram of the Similarity Score for both families. Most repackaged versions are similar to the original ones, with an

average Similarity Score of 0.94, a median of 0.99, and a standard deviation (SD) of 0.16 for the *gappusin* family. For the *revmob* family, the average Similarity Score is 0.81, the median is 0.91, and the SD is 0.26.

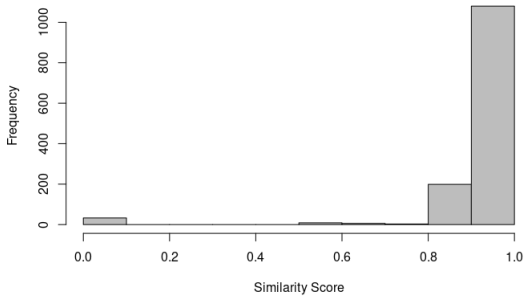
We also reverse-engineered samples from both families. Due to the significant effort required for reverse engineering, we limited our analysis to a sample of 30 *gappusin* and 30 *revmob* malware samples, using the `SimiDroid`[3], `apktool` [4], and `smali2java` [5] tools. Considering this sample, the median Similarity Score is 0.99 and 0.90 for the *gappusin* and *revmob* families, respectively. Table 6 and Table 7 summarize the outputs of `SimiDroid` for these samples.

Regarding the *gappusin* malware, the similarity assessment of this sample of 30 apps reveals a few modification patterns when comparing the original and the repackaged versions. First, no instance in this *gappusin* sample dataset modifies the Android Manifest file to require additional permissions. In most cases, the repackaged version just changes the Manifest file to modify either the package name or the main activity name. Moreover, 29 out of the 30 samples in this dataset **modifies** the method `void onReceive(Context, Intent)` of the class `com.games.AdReciver`. Although the results of the decompilation process are difficult to understand in full (due to code obfuscation), the goal of this modification is to change the behavior of the benign version, so that it can download a different version of the `data.apk` asset. Figure 3 shows the code pattern of the `onReceive` method present in the samples. This modification typically uses a new method (`public void a(Context)`) in the repackaged versions, often introduced into the same class (`AdReceiver`). Since there are no additional calls to sensitive APIs, the MAS approach fails to correctly label the *gappusin* samples. This limitation holds regardless of our experimental choices, such as using the DroidBot tool (instead of more recent test case generation tools) or running the samples for three minutes.

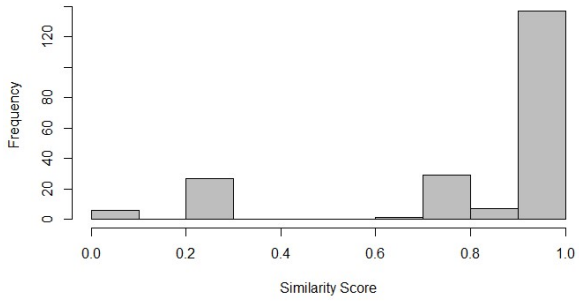
Our assessment also reveals recurrent modification patterns that **delete** methods in the repackaged version of the apps. For instance, 20 repackaged apps in our *gappusin* sample of 30 malware remove the method `void b(Context)` from the class `com.game.a`. This class extensively uses the Android reflection API. Although it is not clear the real purpose of removing these methods, that decision simplifies the procedure of downloading a `data.apk` asset that is different from the asset available in the original version of the apps. Removing those methods might also be a strategy for antivirus evasion. For instance, although some usages of the class `DexClassLoader` might be legitimate, it allows specific types of attack based on dynamic code injection [15]. As such, antivirus might flag specific patterns using the Android reflection API suspect. Unfortunately, the MAS approach also fails to identify a malicious behavior with this type of change (i.e., changes that remove methods), again, regardless of the decisions we follow in our experiment. Listing 4 shows an example of code pattern frequently removed from the repackaged versions from the *gappusin* family.

In summary, our reverse engineering effort brings evidence that malware samples from the *gappusin* family neither modify the Android Manifest files nor call additional sensitive APIs. It acts as a downloader for further malicious app [4] – which reduces the ability of the MAS approach to classify a sample as a malware correctly. Both versions (original/repackaged) from the *gappusin* family have the same behavior for showing advertisements to the user, however, the repackaged version has additional call sites to the advertisement API and the advertisement sources are different.

[3] `https://github.com/lilicoding/SimiDroid`
[4] `https://ibotpeaches.github.io/Apktool/`
[5] `https://github.com/AlexeySoshin/smali2java`



(a) Similarity Score for the samples in the *gappusin* family.



(b) Similarity Score for the samples in the *revmob* family.

Figure 2 Histogram of the Similarity Score for the samples in the *gappusin* and *revmob* families.

Similar to the approach used for *gappusin* samples, we also reverse-engineered a random selection of 30 samples from the *revmob* family that were not detected by the MAS approach. As with the *gappusin* family samples, no instance from the *revmob* family modifies the Manifest file or inserts extra calls to sensitive APIs, making it harder for the MAS approach to label the samples as malware correctly. However, our reverse engineering reveals that all apps store a file with the extension ".so" (Shared Object files) in their lib directory. These files are dynamic libraries containing native code written in C or C++, and are often used by apps for performance reasons, when resource-intensive tasks need to be performed [39].

Unfortunately, creating malicious repackaged apps using ".so" files is also possible, as they can be replaced by a version containing harmful code [36]. The Shared Object files also allow for attacks based on dynamic code injection [15], considering that Android apps can use methods like `System.loadLibrary()` or `System.load()` to download malicious ".so" files from a remote server. Once on the device, malicious apps can use these files to interface with Java code in Android apps, via the Java Native Interface (JNI), performing malicious operations on low-level code and bypassing security mechanisms, like the MAS approach.

Our assessment confirms that all *revmob* samples contain Java code that loads a native library. In particular, to load these libraries, the samples use the `loadLibrary` method of the `System` class, which is called in the static constructor of the `mainActivity` class. The

```
public void onReceive(Context c, Intent intent) {
  SharedPreferences sp = c.getSharedPreferences(String.valueOf("com.")+"game."+"param",0);
  int i = sp.getInt("sn", 0) + 1;
  System.out.println("sn: " + i);
  if (i < 2) {
    mo4a(context);
    SharedPreferences.Editor edit = sp.edit();
    edit.putInt("sn", i);
    edit.commit();
  } else if (!new C0004b(context).f7h.equals("")) {
    String str1 = context.getApplicationInfo().dataDir;
    String str2 = String.valueOf(str1) + "/fi" + "les/d" + "ata.a" + "pk";
    String str3 = String.valueOf(str1) + "/files";
    String str4 = String.valueOf("com.") + "ccx." + "xm." + "SDKS" + "tart";
    String str5 = String.valueOf("InitS") + "tart";
    String str6 = "ff048a5de4cc5eabec4a209293513b6e";
    C0003a.m3a(context, str2, str3, str4, str5, str6);
    SharedPreferences.Editor edit2 = sp.edit();
    edit2.putInt("sn", 0);
    edit2.commit();
  }
}
```

Figure 3 Method introduced in 29 out of 30 *gappusin* malware we randomly selected from the LargeDS.

```
public static void m7a(Activity activity, String str, String str2, ... , String str5) {
  try {
    Class loadClass = new DexClassLoader(..., activity.getClassLoader()).loadClass(str3);
    Object newInstance = loadClass.getConstructor(new Class[0]).newInstance(new Object[0]);
    Method method = loadClass.getMethod(str4, new Class[]{Activity.class, String.class});
    method.setAccessible(true);
    method.invoke(newInstance, new Object[]{activity, str5});
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Figure 4 Example of method that is typically removed from the repackaged apps of the *gappusin* family.

```
public class PZPlayer extends Cocos2dxActivity {
    // ...
    System.loadLibrary("game");
    // ...
}
```

Figure 5 Thie code links this java file into libgame shared library.

`loadLibrary` method takes "game" as an argument, and the code automatically searches the default lib directory for the `.so` file named (`lib+argument`). The "lib" directory contains the `libgame.so` file in all samples. Figure 5 presents the code pattern of the `mainActivity` class found in the samples from *revmob* family.

The `libgame.so` file contains compiled code written in C or C++, which is loaded into memory and linked to the apps at runtime. Although machine code is difficult to analyze, all the files include the `JNI_OnLoad` function, which the JNI implementation automatically uses to link Java methods and native functions. When we analyzed the ".so" file, we found that they all differ in size and content between the original and repackaged apps. It is possible that changes of interest occurred in the native `libgame.so` file and have gone unnoticed

Table 6 Summary of the outputs of the `SimiDroid` tool for the sample of 30 *gappusin* malware. (IM) Identical Methods, (SM) Similar Methods, (NM) New Methods, and (DM) Deleted Methods.

Hash	Similarity Score	IM	SM	NM	DM
33896E	0.9994	3205	2	0	0
0C962D	0.9994	3413	1	1	10
BCDF91	0.9992	2645	2	0	0
01ECE4	0.9991	5697	4	1	10
A306DA	0.9989	1886	1	1	6
4010CA	0.9987	3721	1	4	6
5B5F2D	0.9983	1164	2	3	0
010C07	0.9982	2248	4	3	0
F9FC04	0.9982	1121	1	1	6
E29F53	0.9976	842	1	1	6
FE76EB	0.9976	839	1	1	6
842BD5	0.9973	2249	3	3	3
295B66	0.9972	1081	2	1	10
92209D	0.9971	698	2	3	0
0977B0	0.9969	1613	4	1	10
347FCF	0.9967	613	1	1	6
00405B	0.9965	864	2	1	10
67310E	0.9957	1164	2	3	3
CCD29E	0.9954	436	2	0	0
610113	0.9941	836	4	1	10
A871E0	0.9941	836	4	1	10
ECEA10	0.9913	229	1	1	6
E53FAA	0.9889	267	2	1	10
723C23	0.9870	228	2	1	10
D95B6E	0.9870	833	10	1	10
17722D	0.9743	265	6	1	10
537492	0.9504	134	6	1	10
078E0A	0.9504	134	6	1	10
D83F1C	0.9494	150	2	6	6
E5D716	0.8840	2035	68	199	199

by the MAS approach. Again, since the MAS approach only considers differences in calls to sensitive APIs, it is unlikely to correctly classify these samples using other test case generation tools or by extending the execution time during its exploratory phase.

5 Discussion

In this section, we answer our research questions, summarize the implications of our results, and discuss possible limitations of our study that might threaten the validity of the results presented so far.

5.1 Answers to the Research Questions

The results we presented in the previous sections allow us to answer our three research questions, as we summarize in the following.

Table 7 Summary of the outputs of the `SimiDroid` tool for the sample of 30 *revmob* malware. (IM) Identical Methods, (SM) Similar Methods, (NM) New Methods, and (DM) Deleted Methods.

Hash	Similarity Score	IM	SM	NM	DM
14BBE2	0.9940	3348	6	532	14
BFEF74	0.9940	3348	6	532	14
A3FACA	0.7918	2667	80	112	1 621
10F22D	0.9940	3348	6	532	14
50193A	0.9940	3348	6	532	14
5A7536	0.9940	3348	6	532	14
BCC0DB	0.7918	2667	80	112	1 621
E866CB	0.9940	3348	6	532	14
CDD316	0.9940	3348	6	532	14
DF39F6	0.7918	2667	80	112	1 621
3FFAFF	0.9121	3072	184	628	112
C8C63D	0.9940	3348	6	532	14
48C562	0.9121	3072	184	628	112
D27F26	0.7918	2667	80	112	1 621
F4BBEC	0.9121	3072	184	628	112
BCF14C	0.9127	3074	182	628	112
7FBF11	0.7918	2667	80	112	1 621
9D35D4	0.7918	2667	80	112	1 621
D1B27E	0.9940	3348	6	532	14
94DD4B	0.9940	3348	6	532	14
2D217E	0.7918	2667	80	1121	621
66F167	0.7918	2667	80	1121	621
155D4A	0.9940	3348	6	532	14
8CB780	0.9127	3074	82	628	112
C251FA	0.9940	3348	6	532	14
40487B	0.7918	2667	80	1121	621
F29692	0.9940	3348	6	532	14
0E3679	0.9127	3074	182	628	112
7A4F31	0.9121	3072	184	628	112
BB3EDE	0.7105	2393	256	1217	719

- **Performance of the MAS approach (RQ1).** Our study indicates that the accuracy of the MAS approach reported in previous studies [6, 13] does not generalize to a larger dataset. That is, while in our reproduction study (using the `SmallDS` of previous research) the MAS approach leads to an accuracy of 0.90, we observed a drop of precision and recall that leads to an accuracy of 0.54 in the presence of our `LargeDS` (4,076 pairs of original and repackaged versions of Android apps).
- **Similarity Analysis (RQ2).** Our results bring evidence about the association between the similarity of the original and repackaged versions of an app and the ability of the MAS approach to correctly classify a repackaged version of an app as a malware. Therefore, the similarity assessment is relevant for explaining the performance of the MAS approach to classify certain repackaged versions of an app as malware.
- **Malware Family Analysis (RQ3).** The results indicate that some families are responsible for the largest number of false negatives in the complete dataset. We specifically further investigate the *gappusin* and *revmob* families – a particular type of

Adware, designed to display advertisements while an app is running automatically. After reverse engineering a sample of 60 malware apps from *gappusin* and *revmob* family, we confirmed that the MAS approach cannot identify the patterns of changes introduced in the repackaged versions of the apps. The prevalence of the *gappusin* and *revmob* families in the Android malware landscape accounts for the poor performance of the MAS approach in malware classification on the large dataset.

5.2 Implications

Contrasting to previous research works [6, 25, 13], our results lead to a more systematic understanding of the strengths and limitations of using the MAS approach for malware classification. In particular, this is the first study that empirically evaluates the MAS approach considering as ground truth the outcomes of `VirusTotal` – a common practice in the malware identification research. This decision allowed us to explore the MAS approach performance using well-known accuracy metrics (i.e., precision, recall, and F_1 score). Contrasting with previous studies that assume that all repackaged versions of the apps were malware. Our triangulation with `VirusTotal` reveals this is not true. Although the MAS approach presents a good accuracy for the `SmallDS` ($F_1 = 0.90$), in the presence of a large dataset the MAS approach accuracy drops significantly ($F_1 = 0.54$).

We also reveal that some families in the `LargeDS` are responsible for a large number of false negatives, compromising the accuracy of the MAS approach. Altogether, the takeaways of this research are twofold:

- Negative result: the MAS approach for malware detection exhibits a much higher false negative rate than previous research reported.
- Future directions: Researchers should advance the MAS approach for malware detection by exploring more sophisticated techniques to differentiate between benign and malicious apps. In particular, since our reverse engineering results suggest that *gappusin* and *revmob* – two recurrent malware families – use the network to download new assets, new approaches might benefit from monitoring not only calls to sensitive APIs but also network traffic, as well as mining sensitive calls to native APIs embedded in `so` files. The versatility of the Java Native Interface (JNI) has introduced challenges. Malware authors increasingly use the native layer to hide malicious code, making both static and dynamic analysis more difficult. The current state of the art in sandbox mining overlooks native calls.

5.3 Threats to Validity

There are some threats to the validity of our results. Regarding **external validity**, one concern relates to the representativeness of our malware datasets and how generic our findings are. Indeed, mitigating this threat was one of the motivations for our research, since, in the existing literature on the MAS approach for malware classification, researchers had explored just one dataset of 102 pairs of original/repackaged apps. Curiously, for this small dataset, the performance of the MAS approach is substantially superior to its performance on our `LargeDS` (4,076 pairs of apps).

We contacted the authors of the Bao et al. original research paper [6], asking them if they had used any additional criteria for selecting the pairs of apps in their dataset. Their answers suggest the contrary: they have not used any particular app selection process that could explain the superior performance of the MAS approach for the `SmallDS`. We believe our results in the `LargeDS` generalize better than previous research work, since we have a more comprehensive collection of malware with different families and degrees of similarity.

Nonetheless, our research focuses only on Android repackaged malware. Thus, we cannot generalize our findings to malware that targets other platforms or uses different approaches to instantiate a malicious asset. Besides that, repackaging is a recurrent approach for implementing Android malware.

Regarding **conclusion validity**, during the exploratory phase of the MAS approach, we collected the set of calls to sensitive APIs the original version of an app executes, while running a test case generation tool (DroidBot). In the exploratory phase, the MAS approach assumes the existence of a benign original version of a given app. We also query `VirusTotal` to confirm this assumption, and found that the original version of seven (out 102) apps in the `SmallDS` contains malicious code. We believe the authors of previous studies carefully check that assumption, and this difference had occurred because the outputs of `VirusTotal` change over time [50], and a dataset that is consistent on a given date may not remain consistent in the future. Therefore, while reproducing this research, it is necessary to query `VirusTotal` to get the most up-to-date classification of the assets, which might lead to results that might slightly diverge from what we have reported here. Besides that, in the `LargeDS` we only consider pairs of original/repackaged apps for which `VirusTotal` classifies the original version as benign.

Regarding **construct validity**, we address the main threats to our study by using simple and well-defined metrics that are in use for this type of research: number of malware samples the MAS approach correctly/wrongly classify in a dataset (true positives/false negatives). We computed the accuracy results using precision and recall based on these metrics. In a preliminary study, we investigated whether or not the MAS approach would classify an original version of an app as malware, computing the results of the test case generation tools in multiple runs. After combining three executions in an original version to build a sandbox, we did not find any other execution that could wrongly label an original app as malware. Also, we label a repackaged version of an app as malware only if `VirusTotal` reports that at least two engines detect suspicious behavior in that asset. This decision might be viewed as either a weak or strong constraint and could raise concerns about construct validity. However, when we relax this constraint and label an asset as malware whenever at least one engine detects suspicious behavior, precision improves to 0.85, but recall drops to 0.39. Overall, the accuracy of the MAS approach remains almost unchanged ($F_1 = 0.53$) – still significantly lower than the precision of the MAS approach for `SmallDS`. We also evaluated accuracy by considering an asset as malware when at least five or ten `VirusTotal` security engines flagged it. As shown in Table 8, the results did not diverge significantly from what we have reported in this paper.

Table 8 Accuracy of the MAS approach at `LargeDS` (4,076 pairs) based on engines.

Engine(s)	TP	FP	FN	Precision	Recall	F_1
At least 01	1,222	220	1,900	0.85	0.39	0.53
At least 02	1,175	220	1,720	0.84	0.40	0.54
At least 05	1,087	220	1,578	0.83	0.40	0.54
At least 10	1,002	220	1,469	0.81	0.40	0.54

6 Conclusions

To better understand the strengths and limitations of the MAS approach for repackaged malware detection, this paper reported the results of an empirical study that replicates previous research works [6, 13]. The study utilizes a more diverse dataset compared to those used in previous research, with the aim of providing a more comprehensive evaluation

of the approach. To our surprise, compared to published results, the performance of the MAS approach drops significantly for our comprehensive dataset (F_1 score reduces from 0.90 in previous papers to 0.54 here). This result is partially explained by the high prevalence of specific malware families (named *gappusin* and *revmob*), whose samples are incorrectly classified by the MAS approach. We also report the results of a reverse engineering effort, whose goal was to understand the characteristics of the *gappusin* and *revmob* family that reduce the performance of the MAS approach for malware classification. Our reverse engineering effort revealed common changing patterns in the *gappusin* repackaged versions of original apps, which mostly use reflection to download an external apk asset for handling advertisements without introducing additional calls to sensitive APIs. Similarly, the *revmob* family does not include any additional calls to sensitive resources; however, it often uses JNI to interact with native code, which can be used to perform malicious operations at a low level, compromising the effectiveness of the MAS approach for malware identification. These negative results highlight the current limitations of the MAS approach for malware classification and suggest the need for further research to integrate the MAS approach with other techniques for more effective malware identification.

References

1 Milad Taleby Ahvanooey, Qianmu Li, Mahdi Rabbani, and Ahmed Raza Rajput. A survey on smartphones security: Software vulnerabilities, malware, and attacks. *CoRR*, abs/2001.09406, 2020. arXiv:2001.09406.

2 Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: collecting millions of android apps for the research community. In Miryung Kim, Romain Robbes, and Christian Bird, editors, *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, pages 468–471. ACM, 2016. doi:10.1145/2901739.2903508.

3 Omar Alrawi, Miuyin Yong Wong, Athanasios Avgetidis, Kevin Valakuzhy, Boladji Vinny Adjibi, Konstantinos Karakatsanis, Mustaque Ahamad, Douglas M. Blough, Fabian Monrose, and Manos Antonakakis. Sok: An essential guide for using malware sandboxes in security applications: Challenges, pitfalls, and lessons learned. *CoRR*, abs/2403.16304, 2024. doi:10.48550/arXiv.2403.16304.

4 Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. DREBIN: effective and explainable detection of android malware in your pocket. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014. URL: https://www.ndss-symposium.org/ndss2014/drebin-effective-and-explainable-detection-android-malware-your-pocket.

5 Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. Appguard - fine-grained policy enforcement for untrusted android applications. In Joaquín García-Alfaro, Georgios V. Lioudakis, Nora Cuppens-Boulahia, Simon N. Foley, and William M. Fitzgerald, editors, *Data Privacy Management and Autonomous Spontaneous Security - 8th International Workshop, DPM 2013, and 6th International Workshop, SETOP 2013, Egham, UK, September 12-13, 2013, Revised Selected Papers*, volume 8247 of *Lecture Notes in Computer Science*, pages 213–231. Springer, 2013. doi:10.1007/978-3-642-54568-9_14.

6 Lingfeng Bao, Tien-Duy B. Le, and David Lo. Mining sandboxes: Are we there yet? In Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd, editors, *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 445–455. IEEE Computer Society, 2018. doi:10.1109/SANER.2018.8330231.

7 Lorenzo Bordoni, Mauro Conti, and Riccardo Spolaor. Mirage: Toward a stealthier and modular malware analysis sandbox for android. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*, volume 10492 of *Lecture Notes in Computer Science*, pages 278–296. Springer, 2017. doi:10.1007/978-3-319-66402-6_17.

8 Haipeng Cai and Barbara G. Ryder. Droidfax: A toolkit for systematic characterization of android applications. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 643–647. IEEE Computer Society, 2017. doi:10.1109/ICSME.2017.35.

9 Mahima Choudhary and Brij Kishore. Haamd: Hybrid analysis for android malware detection. In *2018 International Conference on Computer Communication and Informatics (ICCCI)*, pages 1–4. IEEE, 2018.

10 Inc. Comscore. Comscore. URL: https://www.comscore.com/Insights/Presentations-and-Whitepapers/2018/Global-Digital-Future-in-Focus-2018.

11 Francisco Handrick da Costa, Ismael Medeiros, Pedro Costa, Thales Menezes, Marcos Vinícius, Rodrigo Bonifácio, and Edna Dias Canedo. Droidxp: A benchmark for supporting the research on mining android sandboxes. In *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 28 - October 2, 2020*, pages 143–148. IEEE, 2020. doi:10.1109/SCAM51674.2020.00021.

12 Francisco Handrick da Costa, Ismael Medeiros, Thales Menezes, João Victor da Silva, Ingrid Lorraine da Silva, Rodrigo Bonifácio, Krishna Narasimhan, and Márcio Ribeiro. Exploring the use of static and dynamic analysis to improve the performance of the mining sandbox approach for android malware identification. *CoRR*, abs/2109.06613, 2021. arXiv:2109.06613.

13 Francisco Handrick da Costa, Ismael Medeiros, Thales Menezes, João Victor da Silva, Ingrid Lorraine da Silva, Rodrigo Bonifácio, Krishna Narasimhan, and Márcio Ribeiro. Exploring the use of static and dynamic analysis to improve the performance of the mining sandbox approach for android malware identification. *J. Syst. Softw.*, 183:111092, 2022. doi:10.1016/j.jss.2021.111092.

14 Ambra Demontis, Marco Melis, Battista Biggio, Davide Maiorca, Daniel Arp, Konrad Rieck, Igino Corona, Giorgio Giacinto, and Fabio Roli. Yes, machine learning can be more secure! A case study on android malware detection. *IEEE Trans. Dependable Secur. Comput.*, 16(4):711–724, 2019. doi:10.1109/TDSC.2017.2700270.

15 Luca Falsina, Yanick Fratantonio, Stefano Zanero, Christopher Kruegel, Giovanni Vigna, and Federico Maggi. Grab 'n run: Secure and practical dynamic code loading for android applications. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, pages 201–210. ACM, 2015. doi:10.1145/2818000.2818042.

16 Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David A. Wagner. Android permissions demystified. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 627–638. ACM, 2011. doi:10.1145/2046707.2046779.

17 Joseph Gardiner and Shishir Nagaraja. On the security of machine learning in malware c&c detection: A survey. *ACM Comput. Surv.*, 49(3):59:1–59:39, 2016. doi:10.1145/3003816.

18 google. Monkey. https://developer.android.com/studio/test/monkey. Accessed: 2020-02-10.

19 Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014.

20 Konrad Jamrozik, Philipp von Styp-Rekowsky, and Andreas Zeller. Mining sandboxes. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 37–48. ACM, 2016. doi:10.1145/2884781.2884782.

21 Konrad Jamrozik and Andreas Zeller. Droidmate: a robust and extensible test generator for android. In *Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILESoft '16, Austin, Texas, USA, May 14-22, 2016*, pages 293–294. ACM, 2016. doi:10.1145/2897073.2897716.

22 Nataniel P. Borges Jr., Jenny Hotzkow, and Andreas Zeller. Droidmate-2: a platform for android test generation. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 916–919. ACM, 2018. doi:10.1145/3238147.3240479.

23 Kobra Khanmohammadi, Neda Ebrahimi, Abdelwahab Hamou-Lhadj, and Raphaël Khoury. Empirical study of android repackaged applications. *Empir. Softw. Eng.*, 24(6):3587–3629, 2019. doi:10.1007/s10664-019-09760-3.

24 Byoung-chir Kim, Kyeonghwan Lim, Seong-je Cho, and Minkyu Park. Romadroid: A robust and efficient technique for detecting android app clones using a tree structure and components of each app's manifest file. *IEEE Access*, 7:72182–72196, 2019. doi:10.1109/ACCESS.2019.2920314.

25 Tien-Duy B. Le, Lingfeng Bao, David Lo, Debin Gao, and Li Li. Towards mining comprehensive android sandboxes. In *23rd International Conference on Engineering of Complex Computer Systems, ICECCS 2018, Melbourne, Australia, December 12-14, 2018*, pages 51–60. IEEE Computer Society, 2018. doi:10.1109/ICECCS2018.2018.00014.

26 Ahmed Lekssays, Bouchaib Falah, and Sameer Abufardeh. A novel approach for android malware detection and classification using convolutional neural networks. In Marten van Sinderen, Hans-Georg Fill, and Leszek A. Maciaszek, editors, *Proceedings of the 15th International Conference on Software Technologies, ICSOFT 2020, Lieusaint, Paris, France, July 7-9, 2020*, pages 606–614. ScitePress, 2020. doi:10.5220/0009822906060614.

27 Li Li, Tegawendé F. Bissyandé, and Jacques Klein. Simidroid: Identifying and explaining similarities in android apps. In *2017 IEEE Trustcom/BigDataSE/ICESS, Sydney, Australia, August 1-4, 2017*, pages 136–143. IEEE Computer Society, 2017. doi:10.1109/Trustcom/BigDataSE/ICESS.2017.230.

28 Li Li, Tegawendé F. Bissyandé, and Jacques Klein. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *IEEE Trans. Software Eng.*, 47(4):676–693, 2021. doi:10.1109/TSE.2019.2901679.

29 Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. Droidra: taming reflection to support whole-program analysis of android apps. In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 318–329. ACM, 2016. doi:10.1145/2931037.2931044.

30 Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 23–26. IEEE Computer Society, 2017. doi:10.1109/ICSE-C.2017.8.

31 Kaijun Liu, Shengwei Xu, Guoai Xu, Miao Zhang, Dawei Sun, and Haifeng Liu. A review of android malware detection approaches based on machine learning. *IEEE Access*, 8:124579–124607, 2020. doi:10.1109/ACCESS.2020.3006143.

32 Michael Maass, Adam Sales, Benjamin Chung, and Joshua Sunshine. A systematic analysis of the science of sandboxing. *PeerJ Comput. Sci.*, 2:e43, 2016. doi:10.7717/peerj-cs.43.

33 William J. Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. *IEEE Trans. Software Eng.*, 43(9):817–847, 2017. doi:10.1109/TSE.2016.2630689.

34 none. anzhi. http://www.anzhi.com/. Accessed: 2021-02-15.

35 Modupe Odusami, Olusola Abayomi-Alli, Sanjay Misra, Olamilekan Shobayo, Robertas Damasevicius, and Rytis Maskeliunas. Android malware detection: A survey. In Hector Florez, Cesar Diaz, and Jaime Chavarriaga, editors, *Applied Informatics - First International Conference, ICAI 2018, Bogotá, Colombia, November 1-3, 2018, Proceedings*, volume 942 of *Communications in Computer and Information Science*, pages 255–266. Springer, 2018. doi:10.1007/978-3-030-01535-0_19.

36 Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin T. S. Chan. On tracking information flows through JNI in android applications. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 180–191. IEEE Computer Society, 2014. doi:10.1109/DSN.2014.30.

37 Husnain Rafiq, Nauman Aslam, Muhammad Aleem, Biju Issac, and Rizwan Hamid Randhawa. Andromalpack: enhancing the ml-based malware classification by detection and removal of repacked apps for android systems. *Scientific Reports*, 12(1):19534, 2022.

38 Abir Rahali, Arash Habibi Lashkari, Gurdip Kaur, Laya Taheri, François Gagnon, and Frédéric Massicotte. Didroid: Android malware classification and characterization using deep image learning. In *ICCNS 2020: The 10th International Conference on Communication and Network Security, Tokyo, Japan, November 27-29, 2020*, pages 70–82. ACM, 2020. doi:10.1145/3442520.3442522.

39 Antonio Ruggia, Andrea Possemato, Savino Dambra, Alessio Merlo, Simone Aonzo, and Davide Balzarotti. The dark side of native code on android. *Authorea Preprints*, 2023.

40 Silvia Sebastián and Juan Caballero. Avclass2: Massive malware tag extraction from AV labels. In *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*, pages 42–53. ACM, 2020. doi:10.1145/3427228.3427261.

41 Saeed Seraj, Michalis Pavlidis, and Nikolaos Polatidis. Trojandroid: Android malware detection for trojan discovery using convolutional neural networks. In Lazaros Iliadis, Chrisina Jayne, Anastasios Tefas, and Elias Pimenidis, editors, *Engineering Applications of Neural Networks - 23rd International Conference, EAAAI/EANN 2022, Chersonissos, Crete, Greece, June 17-20, 2022, Proceedings*, volume 1600 of *Communications in Computer and Information Science*, pages 203–212. Springer, 2022. doi:10.1007/978-3-031-08223-8_17.

42 Vikas Sihag, Manu Vardhan, and Pradeep Singh. A survey of android application and malware hardening. *Comput. Sci. Rev.*, 39:100365, 2021. doi:10.1016/J.COSREV.2021.100365.

43 C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904. URL: http://www.jstor.org/stable/1412159.

44 statista. Statista. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. Accessed: 2023-08-25.

45 Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. The evolution of android malware and android analysis techniques. *ACM Comput. Surv.*, 49(4):76:1–76:41, 2017. doi:10.1145/3017427.

46 Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In Sujay Sanghavi, Sanjay Shakkottai, Marc Lelarge, and Bianca Schroeder, editors, *ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2014, Austin, TX, USA, June 16-20, 2014*, pages 221–233. ACM, 2014. doi:10.1145/2591971.2592003.

47 Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. Wukong: a scalable and accurate two-phase approach to android app clone detection. In Michal Young and Tao Xie, editors, *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA*

2015, Baltimore, MD, USA, July 12-17, 2015, pages 71–82. ACM, 2015. doi:10.1145/2771783.2771795.

48 Linfeng Wei, Weiqi Luo, Jian Weng, Yanjun Zhong, Xiaoqian Zhang, and Zheng Yan. Machine learning-based malicious application detection of android. *IEEE Access*, 5:25591–25601, 2017. doi:10.1109/ACCESS.2017.2771470.

49 Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In Elisa Bertino and Ravi S. Sandhu, editors, *Second ACM Conference on Data and Application Security and Privacy, CODASPY 2012, San Antonio, TX, USA, February 7-9, 2012*, pages 317–326. ACM, 2012. doi:10.1145/2133601.2133640.

50 Shuofei Zhu, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang. Measuring and modeling the label dynamics of online anti-malware engines. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2361–2378. USENIX Association, 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/zhu.

Shouting at Memory: Where Did My Write Go?

Vasileios Klimis
Queen Mary University of London, UK

Abstract

Non-Volatile Memory (NVM) promises persistent data, but verifying that promise on real hardware is challenging due to opaque caching and internal buffers like Intel's WPQ, which obscure the true state of writes. Traditional validation methods often fall short. This paper introduces a novel perspective: leveraging the subtle timing variations of memory accesses as a direct probe into write persistence.

We present a software technique, inspired by echolocation, that uses high-resolution timers to measure memory load latencies. These timings act as distinct signatures ("echoes") revealing whether a write's data resides in volatile caches or has reached the NVM persistence domain. This offers a non-invasive method to track write progression towards durability.

To reliably interpret these potentially noisy timing signatures and systematically explore complex persistence behaviours, we integrate this echolocation probe into an active model learning framework. This synergy enables the automated inference of a system's actual persistency semantics directly from black-box hardware observations. The approach is hardware-agnostic, adaptive, and scalable. Preliminary experiments on Intel x86 – a platform where persistence validation is notably challenged by the opaque Write Pending Queue (WPQ) – demonstrate the feasibility of our technique. We observed distinct latency clusters differentiating volatile cache accesses from those reaching the persistence domain. This combined approach offers a promising path towards robust and automated validation of NVM persistency across diverse architectures.

2012 ACM Subject Classification Software and its engineering → Empirical software validation

Keywords and phrases Persistency Memory Semantics, Fuzz Testing, Model Learning

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.41

Category Pearl/Brave New Idea

Acknowledgements Special thanks to Andy Rudoff, Non-Volatile Memory Software Architect at Intel Corporation, for his insightful input on the development of the read-after-write timestamps mechanism. Gratitude is also extended to Glynn Winskel for his thoughtful feedback on the application of Event Structures in the intricate domain of memory persistency.

1 Introduction

Non-volatile memory (NVM) combines the speed of RAM with persistent data retention, providing reliable storage that survives power outages. The write-handling mechanisms of NVM are captured by formal *persistency* models, such as Px86 for Intel x86 architectures [37] and PArm for ARM architectures [38]. These models aim to align with architectural specifications, guiding developers in programming for NVM. However, they often lack extensive empirical validation to confirm their real-world applicability.

Empirical validation serves two key purposes. First, it ensures hardware vendors adhere to specified standards – an area with historical inconsistencies [2, 4]. Second, it provides a benchmark for testing whether theoretical persistency models align with actual hardware behaviour. This process ensures that models reflect observable behaviours while avoiding those that cannot occur on real machines, thus maintaining both soundness and completeness.

Persistency correctness is not merely academic but critical for system reliability. In practical applications – financial services, healthcare, industrial automation – the failure to respect persistency guarantees can undermine system integrity. For example, in financial

© Vasileios Klimis;
licensed under Creative Commons License CC-BY 4.0

systems that rely on persistent storage, improper write ordering could violate transaction atomicity guarantees, potentially contributing to data inconsistencies if other safeguards fail. Healthcare systems with patient records might experience data loss or inconsistent state retrieval after power events. In safety-critical domains, such as automotive control systems, inconsistent recovery of calibration parameters or system state could lead to unpredictable behaviour during operation. While modern systems implement multiple layers of protection (transaction logs, checksums, reconciliation processes), the fundamental persistency guarantees of NVM serve as a critical foundation for these higher-level reliability mechanisms. Ensuring that persistency behaves as specified is therefore essential for building trustworthy systems across these domains.

Challenges in Validating Persistency Semantics. Validating persistent memory semantics involves significant challenges. A naive approach might involve inducing program crashes by abruptly powering off the system and retrieving data from Non-Volatile Memory (NVM) upon restart. However, this method only reveals the most recent write, leaving the order of prior writes uncertain. Moreover, frequent power cycling is impractical for comprehensive testing, and scheduling precise crashes proves nearly impossible.

Memory bus interception methods also face limitations, particularly with Intel x86 architectures [25]. This arises from the unique data pathway in Intel x86, where data destined for memory first pass through a battery-backed buffer called the *write-pending queue* (WPQ) in the memory controller [40] (see Figure 1). Since bus intercept observations occur after data have left the processor, they fail to accurately reflect the true order of data persistence.

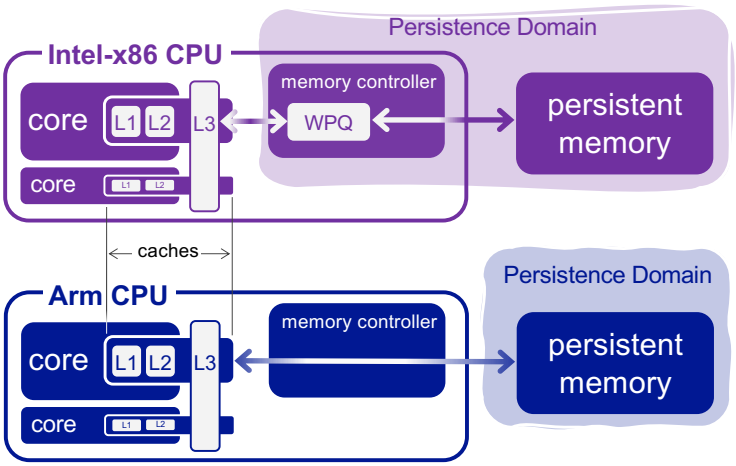


Figure 1 Persistency pathway in Intel x86 and ARM Systems.

In the realm of memory *consistency* models, empirical validation is a well-established practice. Tools like `litmus` [5] execute small multi-threaded tests, known as *litmus tests*, on target machines to examine the visibility and sequencing of writes across threads. This helps clarify the visibility and sequencing of writes among threads. However, unlike memory consistency, a `litmus`-like approach to validating persistency is impractical, as programs cannot directly detect data persistence. Stores may reside in various locations, such as caches, the Write-Pending Queue (WPQ), host memory, or in *Compute Express Link* (CXL) devices.

Representing Behaviour for Validation. Effectively validating NVM persistence – whether checking hardware compliance against specifications like Px86/PArm or diagnosing deviations – requires methods to precisely represent and compare expected versus observed behaviours. This involves capturing the complex interplay of concurrency, weak memory effects, and durability guarantees. While various formalisms exist for modelling concurrent systems (e.g., state machines, Petri nets [34], algebraic approaches [18]), representing the specific partial orders, fence semantics, and persistence states relevant to NVM validation poses unique challenges. Finding a formalism capable of clearly expressing both the specified model's predictions and the potentially divergent behaviours observed on real hardware is crucial for building robust validation techniques. We discuss our choice of formalism, *Event Structures* [44], in Section 5.

Contributions. Our work makes two key contributions:

1) **Introduction of Memory Echolocation Technique:** We introduce memory echolocation, a conceptually architecture-agnostic *timing analysis technique* that uses instruction latencies to non-invasively infer store locations. By analysing retrieval times, this approach identifies distinct timing patterns corresponding to memory hierarchy levels (e.g., caches, WPQ, NVM), revealing write progression towards persistence. Its effectiveness is demonstrated on Intel x86, a challenging platform for validation.
2) **Model-Learning Framework Leveraging Echolocation:** We propose a model-learning based validation framework that utilises memory echolocation (Contribution 1) for black-box observation. This framework enables the iterative refinement of persistency models based on observed timing data, facilitating scalable, adaptive, and more effective validation. It is grounded in *Event Structures* and incorporates persistency properties to accurately capture system behaviour.

2 Background

Non-Volatile Memory (NVM) technologies have introduced new challenges in the reliable management of persistent data. As NVMs retain data across power failures, system designers must ensure that updates to persistent state are crash-consistent – i.e., that data remains consistent even in the event of unexpected interruptions. Achieving this requires explicit reasoning about when writes leave volatile caches and become durable.

Modern architectures such as Intel-x86 and Arm support persistency through specialised instructions that transfer modified data from volatile cache layers to the persistent domain. These mechanisms enable fine-grained control over persist ordering, which is essential because contemporary processors employ multi-level caching for performance, decoupling program execution order from persistence visibility.

Illustrative Litmus Tests. To enforce correct persistency semantics, both Intel-x86 and Arm provide architectural primitives that explicitly govern data flushes to the persistent domain. The examples in Litmus Tests 1 and 2 illustrate minimal persistency patterns: if a crash occurs after y is assigned the value 1, then the programmer expects $x = 1$ to have been persisted, thereby ruling out states where $y = 1$ but $x = 0$.

Litmus Test 1: INTEL.	**Litmus Test 2:** ARM.
1: $x \leftarrow 1$	1: $x \leftarrow 1$
2: $clflush(x)$	2: $dc_cvap(x)$
3: $y \leftarrow 1$	3: $dsb(sy)$
	4: $y \leftarrow 1$

`upon recovery:` $y = 1 \Rightarrow x = 1$

Intel-x86 Persistency Semantics. The Intel-x86 persistency model is inherently buffered, allowing the *persist order* (the sequence in which writes are committed to persistent memory) to diverge from the *coherence order* (the sequence in which writes become visible to threads) [37]. For instance, in the sequence $x \leftarrow 1; y \leftarrow 1$, recovery states after a crash may include $x = 0, y = 0$, $x = 1, y = 0$, or $x = 1, y = 1$, but not $x = 0, y = 1$, which contradicts the program order. This discrepancy arises due to reordering of persist operations.

Intel-x86 offers instructions such as $clflush(x)$ to enforce persistence. This instruction flushes the cache line of x to the persistence domain. Variants such as *clflushopt* and *clwb* provide similar guarantees with differing performance characteristics: *clwb* retains the cache line, while *clflushopt* evicts it. These are non-blocking operations; they trigger persistence asynchronously.

Arm Persistency Semantics. The Arm architecture offers a comparable, though architecturally distinct, model [15, 38]. The instruction $dc_cvap(x)$ transfers data to the persistent domain, but requires a subsequent $dsb(sy)$ barrier to ensure visibility. These synchronisation mechanisms preserve persist order and enforce crash-consistency guarantees akin to those on Intel-x86, as demonstrated in Litmus Test 2.

Persistence Domains and Observability. The scope of the persistence domain – that is, the part of the memory hierarchy where data is guaranteed to persist across power failures – is determined by hardware architecture. On Intel-x86 systems, writes are considered persistent once they reach the memory controller (MC), often equipped with battery-backed buffers. Enhanced ADR (eADR) extends this domain to include L2 and L3 caches [40], further decoupling program execution from persistence observability.

In contrast, Arm architectures maintain a more confined persistence domain, which does not extend to the memory controller, allowing for more direct and observable control over write persistence.

Black-Box Model Learning. Validating persistency models requires a precise understanding of how write and flush operations interact under relaxed behaviours. However, target architectures often function as black boxes, concealing internal details such as persistence ordering and flush timing. These behaviours are instead inferred from observable outcomes of carefully orchestrated operation sequences.

Black-box model learning addresses this challenge by abstracting internal structures and focusing on input-output behaviours. This approach systematically issues crafted sequences of operations (e.g., writes, flushes) and observes the resulting persistence behaviour. Learning algorithms [6] iteratively construct state-transition models that approximate the system's persistency semantics.

For persistency validation, these learned models reveal how writes propagate to the persistence domain under varying conditions, enabling the detection of inconsistencies or undocumented behaviours. By systematically testing even opaque architectures, black-box learning provides a rigorous framework for evaluating complex, poorly understood persistency mechanisms [32].

Event Structures for Modelling Concurrent Behaviour. Persistent behaviours in concurrent programs are naturally expressed in terms of partial orders. *Event Structures* [44] offer a formal framework for modelling these behaviours. An event structure comprises a set of atomic events, a causality relation that expresses dependencies, and a conflict relation that defines mutual exclusions.

This representation is well-suited to reasoning about NVM persistency, which involves partially ordered interactions among writes, flushes, and fences. In this work, we adapt event structures to model the hypotheses explored during model learning. This enables precise specification and comparison of observed system behaviours with theoretical expectations. We present our specific adaptation of Event Structures for NVM persistency in Section 5.

3 Memory Echolocation

This section outlines a straightforward method for inferring store locations within the memory hierarchy, drawing metaphorical inspiration from the concept of *echolocation*. By leveraging timing analysis within the memory hierarchy, we can "map" where data is stored across different memory layers. By doing so, it avoids the need for intrusive approaches like crash-and-recovery, which disrupt system execution and involve directly inspecting memory contents.

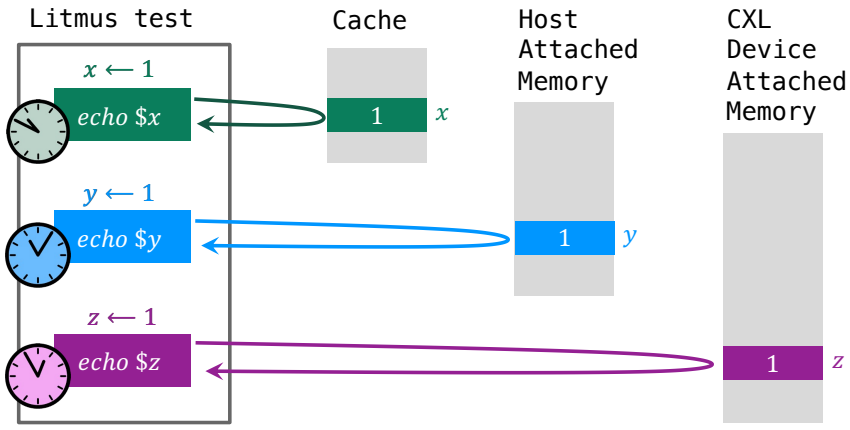


Figure 2 Memory Echolocation.

3.1 Echolocation-Inspired Insight

Echolocation, as observed in nature, involves emitting a signal and interpreting its returning echoes to infer the spatial arrangement of the environment. Our methodology mirrors this approach in the context of memory systems. By performing a store operation, we create a disruption in the memory hierarchy – analogous to emitting a signal. Subsequently, retrieving this data produces measurable latencies, or "echoes", that reveal the location and behaviour of the memory hierarchy components involved.

This analogy underscores the power of latency measurement as a diagnostic tool, revealing patterns of memory residency and transitions across architectural layers. The emitted "signal" in our case is the store operation, and the echoes are derived from the variability in memory access times observed during data retrieval.

3.2 Timing as a Mapping Tool

At the core of our approach is the precise measurement of memory access times. Once the initial store operation is performed, we measure the time required to fetch the data (Figure 2). These timing measurements serve as unique fingerprints of the specific memory layer where the data resides, offering a non-invasive mechanism to explore the memory hierarchy. For example:

- Extremely low latencies indicate L1 cache residency.
- Intermediate latencies may correspond to L2 or L3 caches.
- Longer latencies often point to the Write Pending Queue (WPQ) or main memory (RAM).
- In extended setups, latencies may also reveal accesses to emerging technologies like CXL-enabled memory.

Interpreting Latency Patterns. By correlating these retrieval times with the hierarchical levels of the memory system – such as L1/L2/L3 caches, Write Pending Queue (WPQ), main memory (RAM) and CXL – we can deduce where the stored value resides within the memory architecture. Shorter access times indicate cache residency, while longer delays suggest data retrieval from deeper layers like WPQ or RAM. These timing-based observations serve as a guide to "echo-map" the memory hierarchy, pinpointing the location of stored data with high precision.

4 Learning Persistency Models

Validating persistency semantics in modern systems is a complex and dynamic task, largely due to the intricate interactions between memory management, concurrency, and system failures. As systems grow in complexity, the number of potential states they can reach also increases. Traditional validation methods often fail to capture the full spectrum of behaviours that can emerge under different conditions. This is particularly true when dealing with non-deterministic behaviours, such as those triggered by crashes, race conditions, and the failure-recovery processes that involve complex memory state transitions.

To address these challenges and ensure more comprehensive validation process for persistency semantics, we propose integrating *model learning* techniques. This approach leverages the dynamic interplay between a knowledgeable *Persistency Learner* and an informative *oracle*, enabling automated and optimised validation tasks in a black-box setting. The learner formulates hypotheses about the system's persistency behaviour and tests them against the system through *Membership Queries* (MQ) and *Equivalence Queries* (EQ), receiving feedback from the oracle to validate or refine these hypotheses. This interactive, feedback-driven process enables the model to evolve continuously in real time, uncovering previously undetected edge cases, inconsistencies, and subtle aspects of persistency semantics that may have otherwise gone unnoticed.

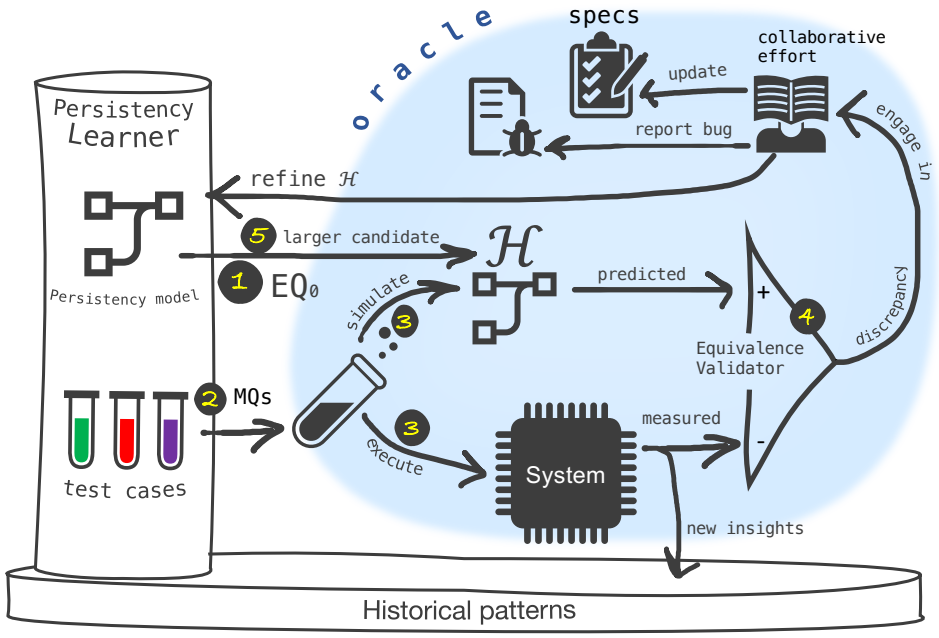


Figure 3 Cycle of the Model Learning Process for Persistency Semantics.

4.1 Persistency Model Learning in Action

The interaction between the learner and the oracle follows a systematic cycle, illustrated in Figure 3. To explore this process, consider a program P with two consecutive writes: $x \leftarrow 1$ followed by $y \leftarrow 1$. Following a crash and recovery, the values of x and y may vary, and the state $x = 0$ and $y = 1$ could appear, even though this state is never observable during normal execution. This inconsistency presents an opportunity for the model learner to refine its hypothesis about write persistency.

To reason about the persistency of writes in this example, we use a toy model built around the concept of a *persist buffer*. This buffer serves as an abstraction that captures the intermediate state of writes that are not yet persisted but are still available within the volatile memory hierarchy. The model incorporates the following elements:

- *Writes:* $w \in W$, where W is the set of all writes executed by the program.
- *Program Order:* The relation $\preceq_{po}$ captures the sequential order of writes in P, where $w_i \preceq_{po} w_j$ means w_i appears before w_j in the program.
- *Persist Buffer:* $\mathcal{B} = (w_1, w_2, \ldots, w_n)$, the ordered sequence of writes currently in the persist buffer, where $w_i \in W$ and $w_i \prec_{pb} w_j$ if write w_i occurs before w_j in the order within the buffer.
- *Persistence States:* A write w can be in one of two states:

$$\mathcal{P}(w) = \begin{cases} w^{\mathcal{P}} & \text{if } w \text{ has been persisted (written to persistent memory)} \\ w^{\mathcal{B}} & \text{if } w \text{ is still in the persist buffer (non-persistent)} \end{cases}$$

  Alternatively, we may use $w^{\neg\mathcal{P}}$ to denote $w^{\mathcal{B}}$.
- *Buffer Transitions:* A transition $\mathcal{B}^{(i)} \xrightarrow{w} \mathcal{B}^{(i+1)}$ represents a change in the contents of the persist buffer $\mathcal{B}$ when write $w \in W$ occurs, resulting in a new buffer state $\mathcal{B}^{(i+1)}$. The exact transformation is defined by specific persistence rules.

- *Crash and Echo Recovery:* X_{echo} represents a "soft crash" simulated using our non-invasive echolocation technique. Instead of a physical crash, this step involves probing the system using timed memory loads to infer the likely location (and thus persistence state) of relevant variables. It approximates the outcome of a crash-recovery cycle without disrupting the system. The specific mechanism for interpreting these probes is detailed in Section 6.

Initially, the model assumes that all writes are persisted immediately upon execution. This is captured by the following invariance rule for the persist buffer:

Immediate Persistence

$$\forall w \in W,\ \mathcal{B}^{(i)} \overset{w}{\hookrightarrow} \mathcal{B}^{(i)}$$

This simplistic assumption sets the stage for the learner to iterate and refine its understanding of the persistency model. The learning process unfolds as follows, with the numbering matching that in Figure 3:

1. **Initial Hypothesis:** The learner begins with the simplest hypothesis, assuming immediate persistence. This predicts $x = 1$ and $y = 1$ post-crash. The hypothesis is submitted as an *Equivalence Query (EQ)* to the oracle, which tests it against the actual System Under Learning (SUL).
2. **Targeted Test Case Generation:** To refine its understanding, the learner generates a test case (a Membership Query, MQ) to probe the system's crash recovery behaviour. The test involves executing a sequence of operations followed by the echolocation probe:

   $$\mathrm{TQ}_i : (x \leftarrow 1;\ y \leftarrow 1;\ \mathsf{X}_{echo})$$

   The effectiveness of echolocation hinges on using carefully constructed, simple test cases like TQ_i. For validating persistency semantics, we employ litmus tests which are kept intentionally small, typically involving only a few memory accesses and persistence-related instructions (flushes, fences). The echolocation probe (X_{echo}) is strategically placed *after* the sequence of operations whose outcome needs assessment, representing the timed load measurements for relevant variables (here, x and y). Complexity in these test cases must be minimised to avoid introducing extraneous micro-architectural effects that could obscure the primary latency signal related to data location. Echolocation is therefore suited for targeted microbenchmarks designed to probe specific persistency interactions, not for direct timing analysis within complex application code.
   The test case TQ_i is submitted as an MQ to observe the actual system behaviour as interpreted through the echolocation probe.
3. **Oracle Feedback:** The oracle executes the MQ on the SUL. Using the X_{echo} probe mechanism (detailed in Section 6), it determines the persistence state of the variables. Suppose the oracle concludes from the probes that x was non-persistent at the time of the simulated crash, while y was persistent. This translates to an observed 'recovered' state of $x = 0, y = 1$, which contradicts the learner's initial hypothesis. This counterexample (the test trace TQ_i and its observed outcome) is returned to the learner.
4. **Discrepancy Analysis and Model Refinement:** The learner revises its hypothesis to account for asynchronous persistence, acknowledging that writes pass through the persist buffer before being persisted, with reordering possible. The updated rule is expressed as follows:

Relaxed Persistence

$\forall w \in W,\ \mathcal{B}^{(i)} \overset{w}{\hookrightarrow} \mathcal{B}^{(i+1)}$ where $\mathcal{B}^{(i+1)} = \mathcal{B}^{(i)} \cup \{w\}$ and $\forall w' \in \mathcal{B}^{(i)}, w' \prec_{pb} w$

This rule accounts for the transient nature of writes in the persist buffer, allowing them to remain temporarily before being committed to persistent storage, with the persistence order potentially non-deterministic.

To accommodate controlled operations that enforce the immediate persistence of specific writes, the model is further refined with a dedicated rule for explicit flushes:

Explicit Flush Handling

$\forall w \in W$ where w modifies x and $w \preceq_{po}$ `CLFLUSH` x, `CLFLUSH` $x \implies w^{\mathcal{P}}$

5. **Iterative Hypothesis Testing:** The learner continues generating test cases (MQs or as part of EQs) to challenge the updated model. For instance:

   $\text{TQ}_{i+1} : (x \leftarrow 1;$ `CLFLUSH` $x;\ y \leftarrow 1;\ \mathsf{X}_{echo})$

   This test case examines whether the system flushes x to persistent memory before the write to y occurs. If the X_{echo} probe (following $y \leftarrow 1$) leads the oracle to conclude that x is persistent but y is not, the inferred recovered state would be $x = 1, y = 0$.

4.2 Algorithmic Implementation of Persistency Learning

Algorithm 3 outlines the iterative learning process at a high level, highlighting the roles of Membership Queries (MQs), Equivalence Queries (EQs), and the echolocation probe (X_{echo}). The algorithm addresses several practical considerations in model learning. It establishes explicit termination conditions through *maxIterations* and *convergenceThreshold* parameters. This ensures the learning process eventually terminates, even when a perfect model cannot be achieved due to system complexity or non-determinism. The *errorRate* tracking provides a quantitative measure of model quality, calculated as the proportion of test cases where the model's predictions differ from observed outcomes. Central to our approach is the *observationTable* T, which serves as a structured knowledge repository mapping test sequences to their observed outcomes. This table plays a crucial role in both generating test sequences for equivalence queries and constructing refined hypothesis models. The learning process has two distinct phases:

1. *Equivalence Query Phase*: The algorithm generates test sequences based on the current hypothesis and observation history. Each test is augmented with the echolocation probe and executed on the System Under Learning. Discrepancies between predicted and observed outcomes are collected as counterexamples. If no counterexamples are found, the current hypothesis is accepted as equivalent to the target model.
2. *Hypothesis Refinement Phase*: When counterexamples are found, the algorithm selects the most informative one to guide the generation of targeted membership queries. These queries explore specific aspects of the system's behaviour revealed by the counterexample. Results from these queries update the observation table, which is then used to construct a new hypothesis model.

The *HypothesisModel* H maintains rich state, tracking the abstract persist buffer content and the status of writes ($w^{\mathcal{P}}$ or $w^{\mathcal{B}}$). The *observed_outcome* and *predicted_outcome* represent the inferred set of persisted variable states (e.g., $x = 0, y = 1$) after a test sequence. The refinement process involves updating H's internal representation of buffering and persistence rules based on the observed outcomes stored in the observation table. This algorithm strikes a balance between theoretical soundness and practical applicability, accommodating the inherent challenges in learning models of complex persistency semantics in real-world systems.

Algorithm 3 Persistency Model Learning with Echolocation.

Require: SUL (System Under Learning, provides Oracle access)
Require: *maxIterations* (maximum learning iterations)
Require: *convergenceThreshold* (acceptable error rate)
Ensure: *HypothesisModel* H approximating SUL's persistency behaviour

Initialise H (e.g., based on "Immediate Persistence" rule)
Initialise *observationTable* $T \leftarrow \emptyset$ *// Explicitly define knowledge structure*
$iteration \leftarrow 0$
$errorRate \leftarrow 1.0$
while $iteration < maxIterations$ **and** $errorRate > convergenceThreshold$ **do**
// – Equivalence Query Phase –
  Generate test sequences T_{EQ} based on current H and *observationTable* T
  $counterexamples \leftarrow \emptyset$
  $totalTests \leftarrow 0$
  for each test sequence $t \in T_{EQ}$ **do**
    $t_{probe} \leftarrow t\,;\, \mathsf{X}_{echo}$ *// Append echolocation probe*
    $observed_outcome \leftarrow \text{Oracle}(SUL, t_{probe})$ *// Execute on SUL, interpret* X_{echo} *results (Sec 6.3)*
    $predicted_outcome \leftarrow H.\text{predict}(t)$
    $totalTests \leftarrow totalTests + 1$
    if $observed_outcome \neq predicted_outcome$ **then**
      $counterexamples \leftarrow counterexamples \cup \{(t, observed_outcome)\}$
  $errorRate \leftarrow |counterexamples|/totalTests$
  if $counterexamples = \emptyset$ **then**
    return H *// No discrepancies found*
// – Hypothesis Refinement Phase –
  Select most informative counterexample $ce = (t_{ce}, obs_{ce})$ from *counterexamples*
  Generate targeted Membership Queries Q_{MQ} based on ce and *observationTable* T
  for each query sequence $q \in Q_{MQ}$ **do**
    $q_{probe} \leftarrow q\,;\, \mathsf{X}_{echo}$
    $mq_outcome \leftarrow \text{Oracle}(SUL, q_{probe})$
    $T \leftarrow T \cup \{(q, mq_outcome)\}$ *// Update observation table*
  $H_{new} \leftarrow \text{constructHypothesis}(T)$ *// Build model from observation table*
  $H \leftarrow H_{new}$
  $iteration \leftarrow iteration + 1$
return H with annotation of achieved *errorRate*

5 Modelling Formalism

To formally model the complex interactions between concurrency, memory consistency, and NVM persistence guarantees, we adopt *Event Structures* [44]. This formalism provides a natural and expressive framework well-suited to the challenges of reasoning about persistency semantics, as detailed below.

5.1 Rationale for Choosing Event Structures

Event Structures [44] provide a principled foundation for modelling persistency for several key reasons:

True Concurrency and Causal Dependencies. Unlike interleaving models that artificially serialise concurrent operations, Event Structures directly represent concurrency through partial orders (*causality*) and mutual exclusion (*conflict*). This approach naturally captures non-total orderings inherent in weak memory models, particularly the complex interactions between memory operations, flushes, and fences in persistent memory systems.

Explicit Persistency Constraints. The core relations in event structures – causality, conflict, and concurrency – provide a natural framework for encoding the fine-grained behaviours specific to non-volatile memory systems. Causal dependencies represent ordering enforced by barriers, while conflict relations precisely capture requirements such as write persistence ordering (e.g., the "sfence" or dependent flush[1]), preventing executions that violate persistence rules.

Compositionality. Persistency guarantees operate at multiple levels of abstraction. Event Structures support modular reasoning, allowing models of individual instructions or components to be composed without needing a monolithic global state representation. This compositionality is essential for scaling verification to complex systems.

Leveraging these strengths, the next section provides our customised definition of *Persistent Event Structures*, tailored to the specifics of NVM operations and drawing parallels to concrete instruction sets like Intel x86 where relevant.

5.2 Persistent Event Structures

▶ **Definition 1** (Persistent Event Structure). *A* persistent event structure *for a program P over a set of persistent memory locations X and values V is defined as a tuple: $\mathcal{E}_P = (E, T, \theta, \preceq_{po}, \preceq_{hb}, \#, \succ_{rf}, \lambda)$, where:*

- *E is a finite set of events representing atomic memory operations;*
- *T is a set of threads, each executing a sequence of events;*
- *$\theta : E \to T$ maps each event to its issuing thread;*
- *$\preceq_{po} \subseteq E^2$ is the program order, a disjoint union of strict total orders on events within each thread. Specifically, for any $a, b \in E$, if $a \preceq_{po} b$, then $\theta(a) = \theta(b)$, meaning both events occur within the same thread. This relation enforces a sequential ordering of events within each thread.*
- *$\preceq_{hb} \subseteq E^2$ is the happens-before relation, a partial order (transitive, irreflexive, and antisymmetric) that extends program order ($\preceq_{po} \subseteq \preceq_{hb}$) and captures synchronisation effects across threads;*
- *$\# \subseteq E^2$ is the conflict relation, which captures mutual exclusivity between events that cannot occur together in the same execution due to resource contention (such as conflicting reads or writes to the same variable). The relation is irreflexive, symmetric and upward-closed with respect to happens-before order: $\forall e, e' \in E, e \# e' \wedge e \preceq_{hb} e_1 \wedge e' \preceq_{hb} e'_1 \Rightarrow e_1 \# e'_1$.*

[1] "Dependent flush" refers to the common pattern where the persistence guarantee of an asynchronous flush like `CLFLUSHOPT` or `CLWB` is made effective relative to subsequent code via an ordering instruction like `SFENCE`.

- $\succ_{rf} \subseteq E^2$ *is the resulting-from relation, where* $r \succ_{rf} w$ *means that the value read by event* r *was written by event* w*. This relation links each read operation to the specific write that provided its value.*
- $\lambda : E \to \mathcal{A}$ *labels each event with a memory operation from alphabet* $\mathcal{A}$*.*

The alphabet $\mathcal{A}$ of memory operations is structured hierarchically:

1. **Basic Memory Operations:**
   - *Initialisation Operation* $\mathbb{0}$, representing the initialisation of all locations in persistent memory appeared in the program, typically setting them to 0.
   - *Read Operations* $R = \{\mathtt{R}_x^v \mid x \in X, v \in V\}$, where $\mathtt{R}_x^v$ denotes reading the value v from x.
   - *Write Operations* $W = \{\mathtt{W}_x^v \mid x \in X, v \in V\} \cup \{\mathbb{0}\}$, where $\mathtt{W}_x^v$ denotes writing v to x, and $\mathbb{0} = \{\mathtt{W}_x^0 \mid x \in X\}$ represents the initialisation of all locations to zero. All writes are durable, meaning their effects *may* be observed after system recovery.
   - *Read-Modify-Write Operations* $RMW = \{\mathtt{RMW}_x^f \mid x \in X, f \in F\}$ denotes atomic updates to x using the function f. These operations atomically read the value of x, modify it using f, and write the result back to x, ensuring no interference from other threads during execution. Common atomic operations in Intel architectures include:
     - `lock xadd`: Atomic fetch-and-add. Adds a value to x and returns its previous value.
     - `lock cmpxchg`: Atomic compare-and-swap. Compares x with a specified operand and updates x if they match.
     - `lock or/and`: Atomic bitwise OR/AND. Performs a bitwise OR/AND operation on x.
     - `lock add`: Atomic addition. Adds a value to x and stores the result atomically.
2. **Synchronisation Operations:**
   - *Memory Fence* (FE_{mem}) ensures all preceding memory operations (loads and stores) complete before any subsequent ones. In Intel architectures, this is enforced via the `MFENCE` instruction.
   - *Store Fence* (FE_{store}) guarantees that all previous store operations complete before any subsequent stores. This is implemented in Intel architectures via the `SFENCE` instruction.
3. **Persistency Operations:**
   - *Regular Flush* $FL = \{\texttt{flush}\ x \mid x \in X\}$ writes back the cache line containing x to memory, ensuring that all pending writes to that cache line are asynchronously persisted. In Intel architectures, this is implemented via the `CLFLUSH` instruction.
   - *Optimised Flush* $FL_{opt} = \{\texttt{flushopt}\ x \mid x \in X\}$ also flushes the cache line containing x. However, it imposes different ordering constraints: a `flushopt` x operation is ordered only with respect to earlier writes to the same cache line; in contrast, a `flush` x operation is ordered with respect to all writes, regardless of their cache line. In Intel architectures, this functionality is provided by the `CLFLUSHOPT` instruction.
   - *Write Back* $WB = \{\texttt{wb}\ x \mid x \in X\}$ behaves like `flushopt` x but provides better performance; both persist the cache line of x, but `flushopt` x invalidates it, whereas `wb` x may retain it for reuse. Implemented in Intel architectures via the 'CLWB' instruction.

We define $W^{\mathcal{P}} \subseteq W$ as the subset of writes that have reached persistent memory. The initialisation is always persistent: $\mathbb{0} \in W^{\mathcal{P}}$.

The complete event alphabet is therefore:

$$\mathcal{A} = \underbrace{\{\mathbb{0}\} \cup R \cup W \cup RMW}_{basic\ memory\ operations} \cup \underbrace{\{FE_{mem}, FE_{store}\}}_{fences} \cup \underbrace{\{FL, FL_{opt}, WB\}}_{persist\ operations}$$

Refining the previously defined *resulting-from* relation over the event alphabet, we have:

$$R \times W \ \supseteq\ \succ_{rf} = \{(\mathtt{R}^v_x, \sigma) \mid x \in X, v \in V, f \in F, \sigma \in \{\mathbb{0}, \mathtt{W}^v_x, \mathtt{RMW}^f_x\}\}$$

meaning:

- $\mathbb{0}$ justifies reading *zero* from any location x;
- a write of v to x justifies reading v from x.

A refinement of the *program order* is the *adjacent program order* relation, $\preceq_{adj} \subseteq \preceq_{po}$, which captures the adjacency of events within the same thread. Formally, it is defined as:

$$a \preceq_{adj} b \iff a \preceq_{po} b \wedge \forall c \in E, (a \preceq_{po} c \wedge c \preceq_{po} b) \Rightarrow (c = a \vee c = b)$$

In other words, two events a and b are adjacent in the program order if there is no event c such that $a \preceq_{po} c \preceq_{po} b$ except for a and b themselves.

Events that are neither ordered by *happens-before* nor in *conflict* are considered concurrent. To formally capture the independence of events across different threads, we define the *concurrency* relation, denoted $\|$, as follows:

$$e_1 \parallel e_2 \iff \theta(e_1) \neq \theta(e_2) \wedge \neg(e_1 \preceq_{hb} e_2) \wedge \neg(e_2 \preceq_{hb} e_1) \wedge \neg(e_1 \# e_2).$$

The following example illustrates the application of event structures in this context.

Illustrative Example. Consider two concurrent processes, P_1 and P_2 (Figure 4), running on an Intel x86 architecture and interacting with the shared variable `ingredients`.

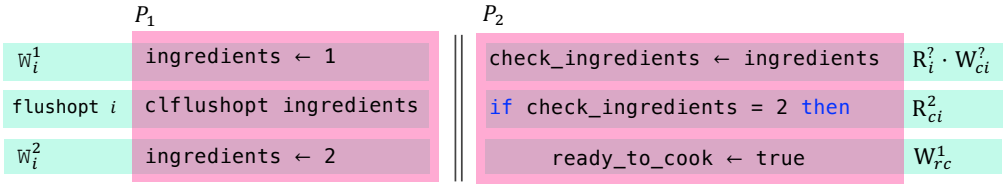


Figure 4 A 2-thread toy persistent program.

A natural expectation is that if `ready_to_cook` is `true`, then `ingredients` should be 2. However, due to the buffered persistence model of Px86, a crash and recovery may result in a state where `ready_to_cook` ← `true` persists while `ingredients` ← `2` does not.

The instruction `clflushopt ingredients` invalidates the cache line containing `ingredients` across the coherence domain, ensuring all cores either discard or update the cached copy. However, it does not enforce immediate persistence and may be reordered with later stores. Because of this, it is possible for the following sequence of events to occur:

1. P_1 writes `ingredients` ← `1` followed by `ingredients` ← `2`, but these updates remain in the cache and have not yet persisted.
2. P_2 reads `ingredients = 2`, sets the local variable `check_ingredients` to 2, and subsequently `ready_to_cook` ← `true`, with the latter being successfully persisted to memory.

3. Before P_1 executes `clflushopt ingredients`, the system crashes. As a result, the *persistent state* upon recovery can have:[2]

$$\texttt{ready_to_cook = true} \quad \wedge \quad \texttt{ingredients} = 0 \tag{1}$$

contradicting the program's expected logic: `ready_to_cook` $\implies$ `ingredients = 2`. This counterintuitive behaviour can, however, be avoided by using the stricter `clflush` instruction instead of `clflushopt`, as `clflush` prevents reordering with respect to any writes.

Representing persistent program behaviour with Event Structures. The semantics of a persistent event structure can be visualised using a directed acyclic graph. In the program shown in Figure 4, each instruction is mapped to a modelled *action*, highlighted next to it. These actions represent reads, writes and flushes in the program. Figure 5 (left) depicts the semantics of the concurrent execution of P_1 and P_2 as an event structure graph, where solid edges show *happens-before* order, dashed edges indicate *resulted-from*, and saw-like edges denote conflicts. When the right thread P_2 reads the value 2 from `ingredients`, which was written by the left thread P_1, P_1 passes a message to P_2. Under the Intel-x86 architecture, message passing ensures that the instruction writing the message ($\mathtt{W}_i^2$) is executed and ordered before the corresponding read instruction ($\mathtt{R}_i^2$) in P_2. This ordering is implicitly captured by the *resulted-from* relation, which links reads to their corresponding writes.

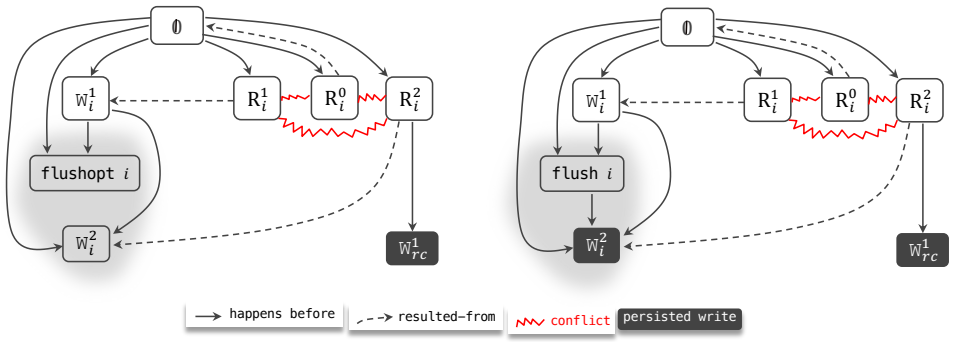


Figure 5 Event Structure semantics for the program in Figure 4, illustrating the *happens-before* relation with `flushopt` (left) and `flush` (right). The affected region of the graph is shaded to indicate where the changes occur.

Replacing `flushopt` i with `flush` i enforces a stricter *happens-before* ordering by introducing an additional edge from `flush` i to $\mathtt{W}_i^2$. As a result, this guarantees that whenever $\mathtt{R}_i^2$ is executed, i has already been persisted.

5.3 Alloy Encoding

We encode the semantics of persistent event structures using the Alloy modelling language [20], a declarative framework that expresses systems as relational structures. This encoding facilitates the analysis of program behaviour under diverse memory access patterns. The model is structured into a set of entities, each representing a distinct aspect of memory

[2] Upon recovery, the possible values are: `ready_to_cook` $\implies$ `ingredients` $\in \{0, 1, 2\}$, which accounts for all potential states.

semantics. Figure 6 illustrates the hierarchical relationships between these entities and their interconnections within the event structure. The corresponding Alloy code defining these entities is provided in Listing 1.

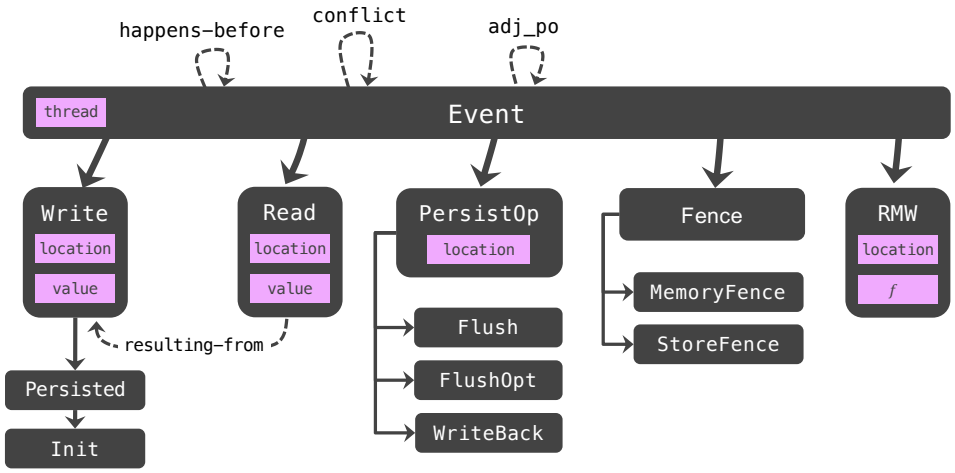


Figure 6 Hierarchical structure of entities (signatures) encoded in Alloy: black boxes represent memory operation events, with solid arrowed lines indicating inheritance. Coloured boxes represent relations that map the events within the outer black boxes to the corresponding entity indicated by the coloured box. Dashed arrow-headed lines denote event-to-event relationships.

Listing 1 Alloy specification for memory model.

```
abstract sig Thread {}
abstract sig Event {
    thread   : some Thread,
    adj_po   : set  Event,
    hb       : set  Event,
    conflict: set  Event,
    rf       : lone Event
}
sig Location {}
sig Write extends Event {
    location: set Location,
    value   : one Int
}
sig Read extends Event {
    location: one Location,
    value   : one Int
}
sig RMW, Fence extends Event {}
sig MemoryFence, StoreFence extends Fence {}
sig PersistOp extends Event {
    location: one Location
}
sig Flush, FlushOpt, WriteBack extends PersistOp {}
sig Persisted in Write {}
one sig Init in Persisted {}
```

In Alloy, memory events are defined as signatures, introducing new types in the model. The `Event` signature serves as the supertype, with specialised event types extending it via the `extends` keyword. Events are structured through relations that link their attributes to other entities (e.g., `Read` $\xrightarrow{rf}$ `Write`) or within the same entity (e.g., `Event` $\xrightarrow{hb}$ `Event`). These relationships are further constrained using facts – axiomatic formulae that Alloy assumes to be always true, shaping the space of possible examples and counterexamples generated by the Alloy analyser. For instance, to enforce that the relation *adjacent program order* (`adj_po`) applies only within the same thread, the following fact is defined:

Listing 2 An Alloy `fact`.

```
fact ProgramOrderScoped {
    all e1, e2: Event | e1 -> e2 in adj_po implies e1.thread = e2.thread
}
```

To illustrate how our Alloy model is executed to generate concrete instances, consider the following `pred`icate, `volatile`. It expresses conditions under which a write `w` could be considered volatile when event `e` occurs. The Alloy analyser systematically explores the model to find instances satisfying this predicate, revealing concrete memory event orderings that meet these constraints.

Listing 3 Checking volatility conditions in memory events.

```
pred volatile[w: Write, e: Event] {
    /* No Flush operation on the same location precedes e in the happens-before
       relation */
    no f: Flush | f.location = w.location and f -> e in hb
    /* No FlushOpt or WriteBack operation on the same location precedes w in hb
     * with a Fence event occurring between them */
    no f: FlushOpt + WriteBack, fence: Fence |
        f.location = w.location and f -> fence in hb and fence -> e in hb
}
```

By executing Alloy's `run` command with this predicate, we generate concrete executions that reveal scenarios where a write remains volatile. The analyser constructs instances that adhere to the specified constraints. One such instance, discussed in Figure 5, satisfies the `volatile` predicate and can be obtained by executing the following command, which explores possible executions within a bounded model of 7 events, 2 threads, and 3 memory locations:

```
run { some w: Write, e: Event | volatile[w, e] } for 7 Event, 2 Thread, 3 Location
```

6 Implementation and Feasibility Demonstration

As a foundational step, we developed a time-based measurement framework designed to explore memory access patterns, inspired by echolocation principles. This initial framework leverages high-resolution timestamps to analyse memory fetch latencies, enabling the identification of subtle timing variations. Central to this is the CPU's Time Stamp Counter (TSC), which provides the high-resolution timings necessary to observe micro-level memory access variations.

6.1 Measurement Technique

Our preliminary implementation, used to generate the latency data shown in Figure 7 and Table 1, employs a microbenchmark written in C (available at DOI 10.5281/zenodo.15045528). The core measurement technique involves timing individual memory load operations using the TSC. Specifically, we utilise the `rdtscp`[3] instruction, accessed via inline assembly, to obtain cycle counts immediately before and after the target memory load operation (e.g., `sum += array[i]` in our benchmark code).

[3] *Read Time-Stamp Counter and Processor ID* is an assembly instruction available on x86 processors that retrieves the current value of the CPU's time-stamp counter (TSC) along with the processor ID. This instruction offers a high-resolution timing mechanism often used for performance monitoring and benchmarking, and is partially serializing.

To prevent the compiler from optimising away the timed memory load, the target memory array (`array`) and the variable accumulating the result (`sum`) are declared `volatile`. Memory for the array is allocated using `posix_memalign` to ensure 64-byte alignment, minimising potential variations due to unaligned accesses.

The elapsed cycle count (end cycles - start cycles) for each load is then converted into nanoseconds. This conversion uses the CPU's clock frequency, which is determined dynamically at the start of the benchmark by parsing the 'cpu MHz' field from `/proc/cpuinfo` on the Linux test system. We acknowledge this frequency detection method is specific to the Linux environment.

In the specific experiment designed to populate the latency distribution (Figure 7), each timed load access to an array element `array[i]` was preceded by a write to that element and an explicit `clflush` instruction (also invoked via inline assembly) targeting its cache line. This was done to ensure that the subsequent timed load would likely fetch data from deeper levels of the memory hierarchy (e.g., L3, WPQ, DRAM) rather than potentially hitting in L1/L2 caches, allowing us to observe the full spectrum of access latencies. The benchmark performs numerous such timed accesses across a large memory region and repeats the entire measurement process multiple times. Individual nanosecond latency measurements are recorded (written to a CSV file in our implementation) to allow for detailed statistical analysis and visualization.

6.2 Feasibility Results

Our tests, carried out on an Intel® Xeon® E-2286G CPU (4.00 GHz, 14 cores) running Linux kernel version 4.18.0 on Red Hat Enterprise Linux 8, demonstrated that we could differentiate memory access latencies across various hierarchical levels based on these measurements. In Figure 7, we present a distribution plot based on approximately 25 million memory fetch operations. The observed latencies ranged from as low as 15 ns for the fastest L1 cache to up to 10,000 ns for the slowest swapped memory on disk. The vertical axis represents the order in which these memory accesses occurred, serving only to visualise the distribution of memory accesses across the entire dataset.

The fetch latencies are clearly segmented into six distinct zones, as highlighted by the shaded regions in the plot and detailed with range values in Table 1. These zones correspond to the hierarchical memory locations of the test system: L1 cache, L2 cache, L3 cache, Write Pending Queue (WPQ), DRAM, and swapped memory. This clear segmentation suggests the potential of our approach in profiling memory access patterns and indicates the reliability of timestamp-based latency measurements for inferring data location.

When measuring memory fetch latencies, the elapsed time includes not only the pure data fetch duration but also several overheads. These include the time required to issue the memory access command, the execution time of the memory access instruction itself, and the small but consistent cost of retrieving timestamps using the `rdtscp` instruction. Additionally, micro-architectural factors, such as cache coherence traffic, prefetcher activity, pipeline effects, resource contention, or cache invalidation caused by instructions like `clflush`, can contribute noise to the measured time. While these overheads and noise sources are present in the absolute measurements, they often remain relatively consistent or manifest as outliers across different fetches under controlled conditions.

For our purpose, which is to differentiate memory access locations within the hierarchy (e.g., cache, write pending queue, DRAM), the absolute latencies are less significant than the relative *differences* and distinct *clusters* observed between fetch times originating from different levels (Figure 7). These large differences tend to cancel out fixed overheads, allowing

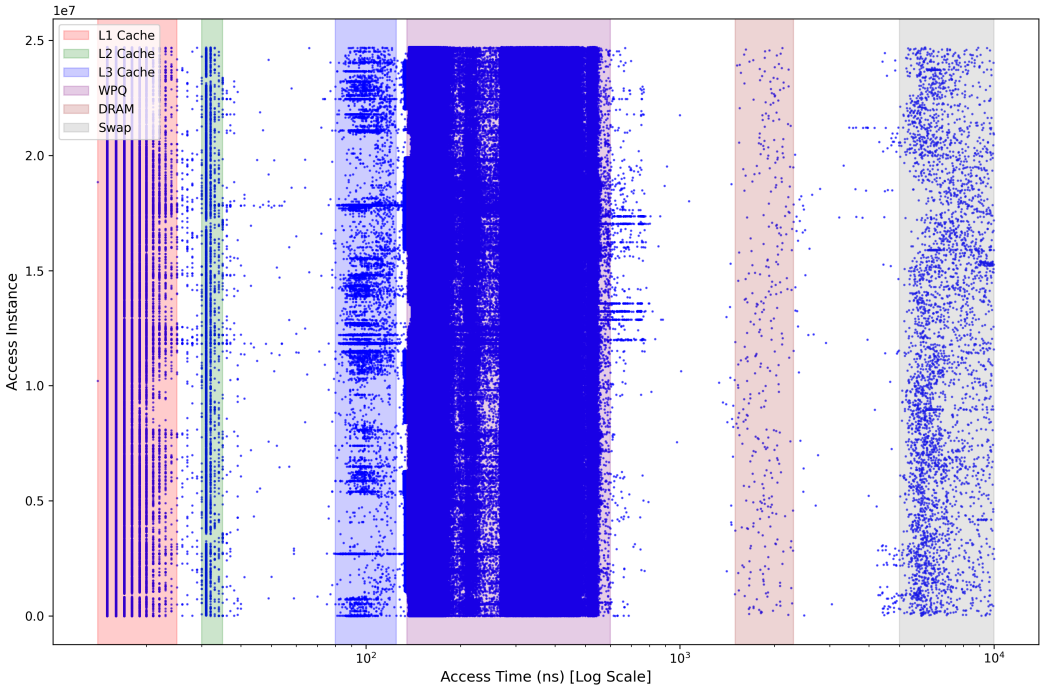


Figure 7 Memory Access Latencies Across Different Levels of the Hierarchy.

Table 1 Latency Ranges and Corresponding Memory Locations. The highlighted rows denote the persistence domain.

Latency Range [ns]	Memory Location	Description
10–25	**L1 Cache**	The closest and fastest memory tier to the CPU (64 KB per core), providing ultra-low latency for frequent data access.
30–35	**L2 Cache**	A slightly larger (256 KB) but slower memory cache near the CPU core, designed for rapid access to recently used data.
75–125	**L3 Cache**	A shared cache among multiple CPU cores, larger in size (12 MB) but slower due to increased complexity in access and coordination.
150–500	**Write Pending Queue (WPQ)**	A battery-backed buffer where data becomes persistent before it leaves the processor.
1500–2300	**Main Memory (DRAM)**	Represents direct access to DRAM, where latency is higher due to physical distance and slower access times compared to cache memory.
5000–10000	**Cached/Prefetched Swap Pages** and **Fully Swapped Memory on HDD**	Encompasses memory pages swapped to disk. Cached or prefetched pages (lower latency) are in RAM, while fully swapped pages (higher latency) require physical disk access.

us to reliably identify the memory hierarchy level from which the data is being fetched based on its characteristic latency range. This approach ensures that variations in fetch latencies used for inference are primarily attributable to the memory location itself rather than extraneous factors, although careful statistical analysis and threshold selection (see Section 8) are needed to handle ambiguity.

6.3 Application to Persistence Validation

The core idea of using echolocation for validation is to map the measured memory load latencies, discussed in the previous subsection, to an inferred persistence state for a given write. This allows us to approximate the outcome of a crash-recovery cycle non-invasively.

▶ **Definition 2** (Inferred Persistence State from Latency). *Let $L(w)$ be the latency measured by the echolocation probe for write w. Let $\mathcal{T}_{PD}$ be the experimentally determined minimum latency threshold indicating entry into the persistence domain, derived from the observed latency ranges (Table 1). On our test machine, $\mathcal{T}_{PD} \approx 150\,ns$, corresponding to the lower bound of WPQ access times. We define the inferred persistence state function $\hat{\mathcal{P}}$ as:*

$$\hat{\mathcal{P}}(w) = \begin{cases} w^{\mathcal{P}} & \text{if } L(w) \geq \mathcal{T}_{PD} \quad \textit{(Inferred Persistent)} \\ w^{\mathcal{B}} & \text{if } L(w) < \mathcal{T}_{PD} \quad \textit{(Inferred Non-Persistent)} \end{cases}$$

Crucially, to mitigate potential misinterpretation due to measurement noise near the threshold (as discussed in Section 8), we adopt a conservative approach when selecting $\mathcal{T}_{PD}$: it is aligned with the upper bound of non-persistent latency clusters plus a safety margin. This prioritises soundness (avoiding false positives for persistence). The oracle function within the model learning algorithm (Algorithm 3) uses this mapping $\hat{\mathcal{P}}$, based on the conservatively chosen $\mathcal{T}_{PD}$, to determine the definitive $w^{\mathcal{P}}$ or $w^{\mathcal{B}}$ state contributing to the observed_outcome from the raw latencies measured during the X_{echo} step.

With this mapping (Definition 2) established, we can validate correctness conditions required after a potential crash. For example, consider the litmus test from Figure 4 and its instrumented version in Figure 8. The expected post-crash condition is:

$$\texttt{ready_to_cook} \implies \texttt{ingredients} \in \{0, 1, 2\} \tag{2}$$

Validation of this condition relies on determining the state of `ready_to_cook` after the program sequence. Our approach uses the X_{echo} probe (executed after the write `ready_to_cook == true` in Figure 8) to measure $L(\texttt{ready_to_cook})$. The Oracle then applies Definition 2:

$$\text{if } \; L(\texttt{ready_to_cook}) \geq \mathcal{T}_{PD}, \; \text{ then } \; \hat{\mathcal{P}}(\texttt{ready_to_cook}) = \texttt{ready_to_cook}^{\mathcal{P}}$$

and the condition (Equation (2)) must hold for the validation to pass. Figure 8 illustrates how this check, driven by the echolocation-derived status, is conceptually embedded in the validation logic.

P_1

```
ingredients ← 1
clflushopt ingredients
ingredients ← 2
```

P_2'

```
check_ingredients ← ingredients
if check_ingredients = 2 then
    # Instrumentation check for persistence
    latency(ready_to_cook ← true) >= PERSISTENCE_THRESHOLD
          implies  ingredients ∈ {0, 1, 2}
```

Figure 8 Instrumented 2-Thread Program for Persistence Validation.

This implementation demonstrating the echolocation probe and its application concept represents a foundational step. As discussed in Section 10, future work involves implementing the full model learning framework and extending validation to more complex scenarios and diverse hardware.

7 Scope and Applicability

The previous sections introduced memory echolocation as a technique for non-invasively probing data location within the memory hierarchy (Section 6) and framed its use within a model learning cycle for validating persistency semantics (Section 4). This combined approach aims to provide a more robust, adaptable, and hardware-agnostic methodology compared to existing techniques. This section discusses the broader scope and applicability of this approach, particularly concerning different hardware architectures and potential extensions.

Hardware Agnosticism and Generality. A key advantage of our proposed methodology lies in its software-based nature. Unlike hardware bus interception methods, which require physical probes, platform-specific knowledge, and may be hindered by component integration (e.g., on SoCs) or internal buffering (like Intel's WPQ), echolocation relies solely on timing standard memory access instructions. This makes the fundamental probing technique portable across any architecture that provides a sufficiently high-resolution timer accessible from software (such as the TSC on x86, or equivalent counters on ARM, RISC-V, etc.). When coupled with the black-box model learning framework, which adapts based on observed outcomes, the overall approach offers a path towards a unified validation framework applicable across diverse hardware platforms, reducing the need for architecture-specific tooling.

Applicability to ARM Systems. While the challenge of the WPQ makes software-based validation particularly appealing for Intel x86, the echolocation technique offers significant advantages for ARM architectures as well:

- **Non-Invasiveness and Accessibility:** Echolocation avoids the cost, complexity, and potential physical impossibility of hardware bus probing, which is especially relevant for highly integrated ARM-based Systems-on-Chip (SoCs) commonly found in mobile, embedded, or IoT devices where physical access might be limited or impractical.
- **Unified Methodology:** It provides the same validation methodology used for other architectures, simplifying cross-platform testing, comparison, and tool development, rather than requiring different approaches (e.g., software for x86, hardware probes for ARM).
- **Future-Proofing:** As ARM architectures evolve, particularly in server or high-performance domains, they may incorporate more complex cache hierarchies, memory controllers with internal buffering, or technologies like CXL, potentially obscuring the exact persistence boundary from external probes, similar to the WPQ issue. Echolocation, being software-based, remains applicable regardless of these internal complexities.
- **Complementary Insights:** Even on ARM systems where bus snooping might be feasible, echolocation provides timing data reflecting the state *before* data necessarily reaches an externally snoopable point. This can offer complementary insights into buffering or ordering effects internal to the CPU core, cache hierarchy, or SoC interconnect.

Therefore, the motivation for using echolocation on ARM extends beyond merely bypassing a WPQ-like issue; its non-invasiveness, generality, and accessibility make it a valuable tool for validating persistency on these platforms.

Synergy of Echolocation and Model Learning. The proposed approach leverages the synergy between the two techniques. Echolocation provides the fine-grained, albeit potentially noisy, observational data about memory state transitions without system disruption. Model learning provides the structured framework (inspired by L^*, as discussed in Section 4) to systematically generate test cases (MQs, EQs), interpret the noisy observations from echolocation within the context of a hypothesis, identify discrepancies, and iteratively refine a model of the system's actual persistency behaviour. This allows for uncovering subtle or undocumented behaviours that might be missed by static analysis or purely hardware-based methods.

Potential for Broader Application. While this paper focuses on validating NVM persistency semantics, the underlying principles might be adaptable. The ability to infer data location in the memory hierarchy via timing could potentially be applied to study cache behaviour, coherence protocol actions, or even aspects of memory consistency, although these applications would require careful calibration, different instrumentation strategies, and potentially more sophisticated analysis techniques to deconvolve timing effects. These remain interesting avenues for future exploration.

Despite its promise, this methodology is not without challenges. The accuracy of echolocation can be affected by micro-architectural noise, and the completeness of model learning depends on the quality of test generation and the ability to explore relevant system states. The next section delves into these potential threats to validation and outlines mitigation strategies.

8 Threats to Validation

The proposed validation approach applies echolocation and model learning to systematically analyse persistency semantics. While promising, its robustness, accuracy, and generality must be scrutinised to strengthen its credibility.

8.1 Accuracy of Echolocation as a Proxy for Crash Behaviours

Echolocation is used to approximate memory access patterns occurring during recovery by analysing write echo times. As demonstrated in Figure 7, the access time zones are generally well-segmented for different memory regions. However, very few access times fall ambiguously between these zones, making it difficult to assign them definitively to a specific storage hierarchy. Such ambiguities may lead to misinterpretations of persistency behaviours.

Mitigation. To resolve this, we propose a two-pronged approach. First, *outlier detection* methods, such as the Local Outlier Factor (LOF) [11], can identify and potentially filter anomalous access times likely caused by transient system noise, improving the clarity of latency clusters. Second, to directly address the ambiguity of times falling near zone boundaries, we adopt the conservative threshold selection strategy detailed in Definition 2. By setting the persistence threshold $\mathcal{T}_{PD}$ based on the upper bound of non-persistent levels plus a margin, we explicitly trade off potentially missing some borderline persisted writes (completeness) for increased confidence that inferred persistence is correct (soundness), thus minimising misinterpretations arising from ambiguous timings.

8.2 Coverage of System States and Edge Cases

While the proposed validation approach uses model learning to penetrate the actual system and explore edge cases, some rare or complex execution paths, particularly those that are non-deterministic or timing-dependent, may remain untested, leading to incomplete coverage and potentially missing subtle persistency behaviours.

Mitigation. Our chosen modelling language, Alloy [20], helps mitigate coverage gaps through its exhaustive analysis within bounded scopes. By encoding the persistency semantics (including timing-related constraints where possible via ordering relations) derived from our model learning, Alloy can explore potential interleavings and states, including those leading to non-deterministic outcomes. Its relational logic is adept at exploring configurations that might expose subtle bugs related to concurrency or complex interactions, complementing the execution-based approach of model learning. Prior work [16, 24, 37, 38] confirms its utility in finding subtle memory model issues. Furthermore, the iterative nature of both Alloy model refinement (reacting to discrepancies found during validation) and the overarching model learning cycle (Section 4) helps systematically expand coverage as inconsistencies are uncovered and the model of system behaviour improves.

8.3 Assumptions in the Initial Model Hypothesis

The model learning process, as discussed in Section 4.1, begins with a simplistic initial hypothesis. While this serves as a foundation for systematic refinement, it may inadvertently introduce constraints or assumptions that bias subsequent learning steps. Such biases could narrow the exploration of the state space, leading to an incomplete or inaccurate understanding of the system's persistency semantics.

Mitigation. To minimise initial hypothesis bias, we adopt several strategies. First, the learning process can be initiated with different starting assumptions beyond simple immediate persistence (e.g., hypotheses incorporating relaxed ordering, different buffering behaviours). Second, we explicitly leverage existing formal persistency models from the literature (e.g., Px86 [37], PArm [38]) as initial hypotheses or baselines for comparison during the learning process on relevant platforms. This provides both a sanity check and ensures our learned models are contextualised against established specifications, adding rigor to the validation of our assumptions and conclusions.

Addressing these threats is crucial for realising the full potential of this validation approach. By systematically mitigating these risks, the methodology can be refined to deliver robust insights into persistency semantics across diverse systems.

9 Related Work

Memory Consistency Model Validation. Significant research has focused on empirically validating memory consistency models across architectures such as x86 [5, 41], IBM Power [39], and Arm [3], as well as GPUs [2] and hybrid CPU/FPGA systems [19]. Recent work extends these models to support features like virtual memory [42] and non-temporal accesses [36]. Traditional multi-threaded validation methods, while effective for consistency, struggle to validate *persistency* models, as they cannot distinguish between data retrieved from volatile caches and persistent memory.

Testing Persistent Programs. Tools such as PMTest [29], Pmemcheck [23], and PMAT [22] test persistent programs by embedding assertions to validate memory persistency guarantees dynamically. XFDetector [28] addresses pre- and post-crash instruction interactions, while PMFuzz [27] generates test cases to maximise path coverage, focusing on persistency-related operations. These tools generally rely on vendor-provided architectural specifications to evaluate whether a program violates persistency guarantees.

Event Structures for Modelling Relaxed Memory. The use of Event Structures (ES) to model systems with relaxed memory ordering guarantees has gained significant attention in recent research [13, 14, 21, 35]. These studies highlight the adaptability of ES in reasoning about concurrency, demonstrating their ability to effectively capture the complexities of weak-memory models across a wide range of architectures and scenarios.

Active Model Learning. Several algorithms have been proposed for learning models across different types of automata, as explored in various studies [1,7,10,12,17,31]. These algorithms can extend their capabilities to learning models of complex, distributed systems, including assemblies of communicating automata [9]. The versatility of these approaches makes them well-suited for testing intricate systems and ensuring the accuracy of persistency models.

Timing-Based Inference. Several studies have leveraged hardware performance counters to track memory access locality, primarily in DRAM [8, 26, 30, 33]. Our memory echolocation technique shares closer methodological similarities with CacheQuery [43], which employs timing-based measurements to detect cache side-channel vulnerabilities. However, our approach differs fundamentally in both objective and execution. While CacheQuery focuses on cache hierarchies to uncover security weaknesses, we extend timing-based inference to persistent memory systems, specifically analysing persistency properties rather than information leakage. Moreover, our method accounts for system-specific variations in persistence behaviour across different architectures, addressing challenges such as write-back buffers, ADR domains, and persistence ordering guarantees – factors beyond CacheQuery's security-oriented scope.

10 Conclusion and Future Work

This paper introduced a software-centric methodology aimed at overcoming persistent challenges in validating NVM semantics on real hardware. Our proposed approach uniquely combines fine-grained memory access timing analysis (*echolocation*) with the structured exploration capabilities of active model learning. We presented the core echolocation technique, demonstrating experimentally on an Intel x86 platform its feasibility for non-invasively distinguishing memory hierarchy levels, including the NVM persistence domain, based on load latency signatures. Furthermore, we outlined how these timing probes serve as the essential black-box observation mechanism within a proposed model learning framework, enabling the automated inference of persistency behaviours. While the full integration of the learning component remains future work, this synergistic approach, conceptually underpinned by *Event Structures*, represents a promising direction towards scalable, adaptive, and hardware-agnostic NVM validation.

10.1 Future Work

Adaptive Echolocation: Enhancing Robustness and Accuracy. As highlighted in our discussion of threats (Section 8), precise analysis using echolocation is complicated by inherent noise in modern systems arising from OS jitter, concurrent processes, architectural variability, micro-architectural interference (e.g., prefetchers, coherence traffic), and even thermal effects. To address this, future work will explore adaptive algorithms capable of dynamically adjusting to system state and filtering noise during real-time latency interpretation. Techniques potentially include online statistical filtering, change point detection, or lightweight machine learning models trained to recognise characteristic latency patterns amidst noise. The goal is to significantly improve the accuracy and reliability of inferred persistence states across diverse architectures, including complex emerging systems incorporating technologies like CXL-attached memory. Successfully learning distinctive latency patterns under varying conditions will enhance data residency detection, identify performance anomalies, and allow more confident prediction of memory state transitions.

Learning Event Structure Models of Persistency. Building on our proposed formalisation using Event Structures (Section 5), a key next step is developing algorithms to automatically infer these structures from echolocation observations. The input would consist of execution traces (sequences of operations generated by the learner) paired with their outcomes (inferred persistence states of relevant variables) obtained via the echolocation probe. The core challenge is to learn the specific causal ($\preceq_{hb}$) and conflict (#) relations that accurately model the target system's possibly undocumented or non-deterministic NVM persistency behaviour.

In conclusion, while the proposed approach represents a promising step forward in persistency validation, it is important to acknowledge that it is not a catch-all solution. The complexity and variability of memory subsystems pose ongoing challenges. Continued research will be essential to refining coverage of dynamic memory conditions beyond our current scope.

11 Data Availability Statement

The Alloy model discussed in Section 5.3, along with instructions for running it, is available on Zenodo (DOI 10.5281/zenodo.15045528). Additionally, we provide the fetch latencies plotted in Figure 7, along with the C script used to obtain them.

References

1 Fides Aarts and Frits Vaandrager. Learning I/O Automata. In *CONCUR 2010 - Concurrency Theory*, 2010.

2 Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In *ASPLOS*, 2015.

3 Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. Armed cats: Formal concurrency modelling at arm. *ACM Trans. Program. Lang. Syst.*, 2021. doi:10.1145/3458926.

4 Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *Computer Aided Verification*, 2010.

5 Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2011.

6 Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.*, 1987. doi:10.1016/0890-5401(87)90052-6.

7 Dana Angluin and Miklós Csundefinedrös. Learning Markov Chains with Variable Memory Length from Noisy Output. In *COLT*, 1997.

8 Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. Anvil: Software-based protection against next-generation rowhammer attacks. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

9 B. Bollig, M. Leucker, J. Katoen, and C. Kern. Learning Communicating Automata from MSCs. *IEEE Transactions on Software Engineering*, 2010.

10 Benedikt Bollig, Peter Habermehl, Martin Leucker, and Benjamin Monmege. A Fresh Approach to Learning Register Automata. In *DLT*, 2013.

11 Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. Lof: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 2000.

12 Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Active Learning for Extended Finite State Machines. *Form. Asp. Comput.*, 2016.

13 Simon Castellan. Weak memory models using event structures. In *Vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA 2016)*, 2016.

14 Soham Chakraborty and Viktor Vafeiadis. Grounding thin-air reads with event structures. *Proc. ACM Program. Lang.*, 3(POPL), 2019. doi:10.1145/3290383.

15 Kyeongmin Cho, Sung-Hwan Lee, Azalea Raad, and Jeehoon Kang. Revamping hardware persistency models: View-based and axiomatic persistency models for intel-x86 and armv8. In *PLDI*, 2021.

16 Nathan Chong, Tyler Sorensen, and John Wickerson. The semantics of transactions and weak memory in x86, Power, ARM, and C++. In *PLDI*, 2018.

17 Samuel Drews and Loris D'Antoni. Learning Symbolic Automata. In *TACAS*, 2017.

18 C. A. R. Tony Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. Concurrent Kleene Algebra. In *CONCUR*, 2009.

19 Dan Iorga, Alastair F. Donaldson, Tyler Sorensen, and John Wickerson. The semantics of shared memory in intel CPU/FPGA systems. *Proc. ACM Program. Lang.*, 5(OOPSLA), 2021. doi:10.1145/3485497.

20 Daniel Jackson. Alloy: a language and tool for exploring software designs. *Commun. ACM*, 2019. DOI: 10.1145/3338843. doi:10.1145/3338843.

21 Alan Jeffrey and James Riely. On thin air reads towards an event structures model of relaxed memory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2016.

22 Louis Jenkins and Michael L. Scott. Persistent memory analysis tool (PMAT). In *11th Annual Non-Volatile Memories Workshop*, 2020. URL: https://louisjenkinscs.github.io/publications/PMAT_EA.pdf.

23 Tomasz Kapela. An introduction to pmemcheck, 2015. URL: https://pmem.io/blog/2015/07/an-introduction-to-pmemcheck-part-1-basics/.

24 Vasileios Klimis, Jack Clark, Alan Baker, David Neto, John Wickerson, and Alastair F. Donaldson. Taking back control in an intermediate representation for gpu computing. *Proc. ACM Program. Lang.*, 7(POPL), 2023. doi:10.1145/3571253.

25 Vasileios Klimis, Alastair F. Donaldson, Viktor Vafeiadis, John Wickerson, and Azalea Raad. Challenges in Empirically Testing Memory Persistency Models. In *ICSE-NIER*, 2024.

26 Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: cache attacks on mobile devices. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC)*, 2016.

27 Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Manabi Khan. Pmfuzz: test case generation for persistent memory programs. In *ASPLOS*, 2021.

28 Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas F. Wenisch, Aasheesh Kolli, and Samira Manabi Khan. Cross-failure bug detection in persistent memory programs. In *ASPLOS*, 2020.

29 Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Manabi Khan. Pmtest: A fast and flexible testing framework for persistent memory programs. In *ASPLOS*, 2019.

30 Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

31 Joshua Moerman, Matteo Sammartino, Alexandra Silva, Bartek Klin, and Michał Szynwelski. Learning Nominal Automata. In *POPL*, 2017.

32 Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. *J. Autom. Lang. Comb.*, 2001.

33 Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for Cross-CPU attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 565–581, Austin, TX, August 2016. USENIX Association. URL: `https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl`.

34 Carl Petri and Wolfgang Reisig. Petri net, 2008. DOI: 10.4249/scholarpedia.6477. `doi:10.4249/SCHOLARPEDIA.6477`.

35 Jean Pichon-Pharabod and Peter Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. *SIGPLAN Not.*, 2016.

36 Azalea Raad, Luc Maranget, and Viktor Vafeiadis. Extending intel-x86 consistency and persistency: formalising the semantics of intel-x86 memory types and non-temporal stores. *Proc. ACM Program. Lang.*, 6(POPL), 2022. `doi:10.1145/3498683`.

37 Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the intel-x86 architecture. *Proc. ACM Program. Lang.*, 4(POPL), 2020. `doi:10.1145/3371079`.

38 Azalea Raad, John Wickerson, and Viktor Vafeiadis. Weak persistency semantics from the ground up: formalising the persistency semantics of armv8 and transactional models. *Proc. ACM Program. Lang.*, 3(OOPSLA), 2019. `doi:10.1145/3360561`.

39 Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *PLDI*, 2011.

40 Steve Scargall. Persistent memory architecture. In *Programming Persistent Memory: A Comprehensive Guide for Developers*, pages 11–30. Apress, 2020.

41 Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7), 2010. `doi:10.1145/1785414.1785443`.

42 Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. Relaxed virtual memory in armv8-a. In *ESOP*, 2022.

43 Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. Cachequery: learning replacement policies from hardware caches. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.

44 Glynn Winskel. Event structures. In *Proceedings of an Advanced Course on Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986-Part II*, 1986.

Contract Systems Need Domain-Specific Notations

Cameron Moy
PLT @ Northeastern University, Boston, MA, USA

Ryan Jung[1]
Northeastern University, Boston, MA, USA

Matthias Felleisen
PLT @ Northeastern University, Boston, MA, USA

Abstract

Contract systems enable programmers to state specifications and have them enforced at run time. First-order contracts are expressed using ordinary code, while higher-order contracts are expressed using the notation familiar from type systems. Most interface descriptions, though, come with properties that involve not just assertions about single method calls, but entire call chains. Typical contract systems cannot express these specifications concisely. Such specifications demand domain-specific notations. In response, this paper proposes that contract systems abstract over the notation used for stating specifications. It presents an architecture for such a system, some illustrative examples, and an evaluation in terms of common notations from the literature.

2012 ACM Subject Classification Software and its engineering → Domain specific languages

Keywords and phrases software contracts, domain-specific languages

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.42

Category Pearl/Brave New Idea

Related Version *Full Version*: https://doi.org/10.5281/zenodo.15285750

Supplementary Material *Software*: https://doi.org/10.5281/zenodo.15285740
Software (ECOOP 2025 Artifact Evaluation approved artifact):
https://doi.org/10.4230/DARTS.11.2.4

Funding This work was supported by NSF grant SHF 2116372.

1 From General-Purpose to Domain-Specific Notations

Eiffel's contract system assumes that programmers can formulate Boolean expressions to articulate simple pre-conditions and post-conditions for methods [33, 34]. Such expressions suffice to state constraints that quotidian type systems cannot express. Here is an example, from Eiffel's documentation, of a method that sets the `seconds` field of a clock object:

```
set_seconds (s: INTEGER)
  require
     valid_argument_for_seconds: 0 <= s and s <= 59
  do
     seconds := s
  end
```

By monitoring the pre-condition, Eiffel's contract system can signal an error when control enters the method, not after `seconds` has been set and retrieved elsewhere. Eiffel's syntax has `require` and `ensure` keywords that cleanly separate specifications from implementations.

[1] Currently working in industry.

© Cameron Moy, Ryan Jung, and Matthias Felleisen;
licensed under Creative Commons License CC-BY 4.0

Findler and Felleisen [16] generalize Eiffel's design to modern, higher-order languages with first-class functions and objects. They realize that such a generalization necessitates creating wrappers around behavioral values. Here is the code for checking that a `derivative` function takes real-valued functions to real-valued functions:

```
(define (derivative unsafe-f)
  (define (check-real->real g)
    (lambda (x)
      (check-real (g (check-real x)))))
  (define f (check-real->real unsafe-f))
  (check-real->real (lambda (x) ... compute derivative ...)))
```

Programmers should never read, let alone write, such complex code. Contract systems, such as the one found in Racket [14, 19], therefore supply the familiar notation from type systems to eliminate these awkward and repetitive patterns:

```
(define/contract (derivative f)
  (-> (-> real? real?) (-> real? real?))
  (lambda (x) ... compute derivative ...))
```

In other words, developers may use the `->` notation as a shorthand for the tedious wrappers in the first version. In Eiffel, contract-specific notations are mere conveniences, but here the notation is essential – writing complex higher-order contracts without them is infeasible.

Moy and Felleisen [37] present an extension of higher-order contracts, dubbed *trace contracts*, that can express and monitor properties across multiple function and method calls. The idea looks straightforward: collect a trace of method-call events and check the trace with a functional predicate whenever it is updated.

Consider the HasNext property that Java programmers must respect when they use an iterator: each call to `next` must follow a call to `hasNext` [39]. A typical implementation would equip the class with an extra field to determine whether a violation has occurred:

```
class Iterator {
  private boolean canCallNext = false;

  public boolean hasNext() {
    this.canCallNext = true;
    ... method body ...
  }

  public Element next() {
    if (!canCallNext)
      throw new Exception("HasNext property violated");

    this.canCallNext = false;
    ... method body ...
  }
}
```

Comingling run-time checks with proper functionality in this way is a technique known as *defensive programming*. As Meyer [33, 34] argued long ago, comingling is bad for the person who maintains the implementation, bad for the person who programs to the interface, and bad for the person who wants to implement the interface differently.

By contrast, a contract system brings relief to all three groups. It enables the developer of an *interface* to state contracts for cross-cutting specifications and attach them to concrete implementations. A contract, independent of any particular implementation, confers two critical advantages:

1. Developers can program to interfaces that express constraints explicitly as code.
2. Contracts are compiled to run-time checks that monitor implementations of interfaces, so developers can trust that specification violations are reported in a timely fashion.

```
(provide
  (contract-out [an-iterator-object has-next/c]))

(define has-next/c
  (object-trace/c
   #:satisfies (re (star (seq 'has-next (opt 'next))))
   [has-next (->m boolean?) 'has-next]
   [next     (->m any/c)    'next]))

—— definition of the iterator ——
```

Figure 1 Checking the HasNext Property.

With trace contracts, the creator of an iterator class can express the HasNext property as a contract and gain all the benefits thereof. Figure 1 shows a module that exports an iterator object and, in doing so, attaches a contract that concisely states the HasNext property with the regular expression $(\texttt{hasNext}, \texttt{next}?)^*$ over the alphabet of method names. Other than the usual fully parenthesized syntax of Racket, the regular expression is rendered directly. The use of a problem-specific notation, here regular expressions, clarifies the developer's intent much better than comingled code, and permits the automated construction of dynamic checks. Each method, after the signature given by `->m`, is associated with an *event*. An event is a value to be added to the trace when that method is called. Here, the events are just atomic symbols: `'has-next` and `'next`.

Which brings us to a brave new idea: *contract systems should be abstracted over domain-specific notations for specifications*. In other words, it should be easy to plug in any number of domain-specific specification languages into a contract system. This paper illustrates the idea with *trace contracts* as the enforcement mechanism and a variety of *domain-specific specification languages*, which together yield domain-specific contracts (DSCs). The remainder of the paper describes DSCs and the underlying architecture in detail, shows how it works with a simple example (Section 2), presents a more sophisticated example from a real-world API (Section 3), describes a general plan for implementing DSCs in different host languages (Section 4), and evaluates a Racket implementation of DSCs (Section 5).

2 Domain-Specific Contracts in Action

Many interface specifications employ temporal terminology to constrain sequences of method calls. Consider the MapIterator property from Java's API [40]. This property states that an `Iterator`, created from a `Collection`, which is itself derived from a `Map`, cannot be used *after* the `Map` has been mutated. The word "after" suggests a temporal relationship – a need to look back at whether a mutating method of `Map` has previously been called.

An interface, therefore, should use a matching formal language, such as temporal logic, to state the property. Given that such statements tend to look at past events, Past-Time Linear Temporal Logic (PLTL) [31] is one suitable choice for stating the MapIterator contract. Indeed, the definition of satisfaction for PLTL formulas employs finite traces of past events – just like trace contracts.

PLTL comes with the primitive temporal operators $\bullet\,\phi$ (pronounced "previous") and $\phi\,\mathsf{S}\,\psi$ (pronounced "since"). Otherwise, PLTL looks like first-order logic. All other temporal operators, for example $\blacklozenge\,\phi$ (pronounced "once"), are derived notions. With these operators, an *invalid* sequence of method-call events is easy to express: `next` $\wedge\,\blacklozenge$ `mutate`. In prose, calling the `next` method of an iterator is invalid if the map has been mutated.

One contribution of this paper is a number of libraries, including one for PLTL, that developers can use in contracts. So, suppose a developer wishes to release a collections library that states and enforces the MapIterator property. Realizing this desire calls for three steps.

First, the developer imports the trace contract library and the PLTL library:

```
(require trace-contract)
(require logic/pltl)
```

Importing `trace-contract` brings in, among other forms, `object-trace/c`. As the introduction mentions, `object-trace/c` constructs an object contract that records a trace of method-call events. This trace is checked by a predicate every time a new element is added.

Second, the developer formulates and names the PLTL formula associated with the MapIterator property:

```
(define-pltl map-iterator-violation
  (∧ 'next (♦ 'mutate)))
```

Using the PLTL library, a developer can formulate the predicate as the above temporal formula, matching the informal prose found in Java's documentation. To use a named PLTL formula, a programmer must use the `pltl` form, which translates it to a (state) `machine` representation. A `machine` acts as a predicate over traces, i.e., it can check whether a trace satisfies or refutes a formula:

```
> (define s1 (list 'next 'next 'mutate 'next))
> (machine-accepts? (pltl map-iterator-violation) s1)
#true
```

Third, the developer combines a PLTL expression with `object-trace/c` to create an enforceable contract. Figure 2 shows a first attempt at such a contract for MapIterator. The three displayed definitions express a specification involving three methods across two classes, with an intervening class that bridges the gap. Two methods from the `Map` class – `remove` and `set` – mutate the `Map`. The intent with these contracts is that any iterator derived from a collection of keys becomes unusable when the underlying map is mutated. The `#:refutes` clause requests that the PLTL formula named `map-iterator-violation` is translated into a predicate over the trace of method-call events. If the contract ever discovers that the formula is satisfied, then the system signals a violation.

As is, these contracts are not quite right. An `object-trace/c` contract knows only about method calls for the current object. Therefore, `map-iterator/c` sees only calls to the `next` method and is unaware of any mutations to the underlying map.

Figure 3 shows how to adapt these contracts to inform the `map-iterator/c` contract of the other relevant method calls. To express this relationship, the contracts for collection and iterator objects become *functions* that accept the `map/c` contract itself. The `object-trace/c` form uses the keyword `this-contract` which, similar to `this` for objects, refers to the

```
(define map-iterator/c
  (object-trace/c
   #:refutes (pltl map-iterator-violation)
   [next (->m any/c) 'next]))

(define map-collection/c
  (object-trace/c
   [iterator (->m map-iterator/c)]))

(define map/c
  (object-trace/c
   [keys   (->m map-collection/c)]
   [remove (->m any/c void?)       'mutate]
   [set    (->m any/c any/c void?) 'mutate]))
```

Figure 2 First Attempt: Checking the MapIterator Property.

```
(define (map-iterator/c map/c)
  (object-trace/c
   #:refutes (pltl map-iterator-violation)
   #:include map/c
   [next (->m any/c) 'next]))

(define (map-collection/c map/c)
  (object-trace/c
   [iterator (->m (map-iterator map/c))]))

(define map/c
  (object-trace/c
   [keys   (->m (map-collection/c this-contract))]
   [remove (->m any/c void?)       'mutate]
   [set    (->m any/c any/c void?) 'mutate]))
```

Figure 3 Checking the MapIterator Property.

`object-trace/c` contract itself. While the `map-collection/c` function just passes this contract through to the `map-iterator/c` function, the latter actually makes use of it. The `#:include` clause informs the contract system that events from the trace for `map/c` are to be injected into the trace for `map-iterator/c`. Consequently, `map-iterator-violation`, which refers to the `'mutate` event, now works correctly in the `#:refutes` clause.

These contracts look simple, but the underlying property is sophisticated. Consider the scenario where an iterator is created, the underlying map is mutated, and then another iterator is created. It should be the case that the first iterator is invalidated, while the second one is usable. In other words, these two iterators require independent pieces of state. Some events, such as mutations to the underlying map, are shared with both. Other events, such as calls to `next`, are particular to a specific iterator and are not shared. All this careful state management is handled automatically.

Conversely, manually constructing defensive checks to correctly keep track of this information is difficult. Appendix A shows an implementation of defensive checks for MapIterator, similar to the defensive code shown in Section 1. The code is complex and takes several pages to explain. By contrast, Figure 3 is nearly self-explanatory.

3 Real Code Needs a Mixture of Domain-Specific Notations

Interfaces should be expressed as directly as possible, using the notation most appropriate for the property at hand. Sometimes a single interface benefits from mixing several domain-specific notations. The DSCs presented in this paper are flexible enough to allow a heterogeneous mixture of notations because they all compile to the same uniform mechanism for capturing and checking traces of events.

This section presents a series of contracts that ensure TCP sockets are used correctly. Consider an idealized TCP interface, similar to the one found in Java, which consists of four classes: a TCP manager, a TCP listener, an input stream, and an output stream. A TCP manager, when given a port number, is responsible for creating a TCP listener that deals with the given port. The purpose of a TCP listener is to accept client connections. For each connected client, a TCP listener produces an input and output stream for receiving and sending data, respectively.

Many languages, including Java, guarantee the following properties of TCP sockets:

- StreamClose. Input and output streams can be used until either the stream itself is closed or the underlying TCP listener is closed.
- PortExclusivity. At any point in the program, there can be at most one active TCP listener on each port.

The Java standard library uses defensive programming to enforce these properties.

Instead, DSCs can nicely express both properties, but they warrant two different notations. The StreamClose property can be expressed as a regular expression:

```
(define-re port-re
  (seq (star 'use) (opt (union 'close-stream `(close-listener
      ,_)))))
```

This regular expression accepts traces of method-call events that use a stream (input or output) until it has been closed, or the underlying listener is closed. There is one unusual feature here, which is the close listener event. In all prior examples, events were symbolic names of methods carrying no additional data. The close listener event also carries the port number of the corresponding listener, which is simply ignored in `port-re` using a wildcard pattern. Why the close listener event carries a port number will be clarified momentarily.

The PortExclusivity property is not easily expressed as a regular expression. Another formalism, known as quantified event automata (QEA [4]), is a good match:

```
(define manager-qea
  (qea
   (∀ port)
   (start ready)
   [-> ready `(listen ,port) listening]
   [-> listening `(close-listener ,port) ready]))
```

A QEA is similar to an ordinary finite-state automaton, but it is enriched with quantifiers. By quantifying over a variable, a QEA describes a *family* of automata: one per instantiation of all quantified variables. Intuitively, `manager-qea` constructs a two-state automaton for every possible port number. These automata are created on demand, though, to avoid unnecessary overhead. Each automaton starts in the `ready` state and transitions to the `listening` state when a new TCP listener is created on that port. When the listener is closed, the automaton moves back to the `ready` state. Each listener event must carry a port number so the event can be dispatched to the appropriate automaton.

```
(define (input-stream/c listener/c)
  (object-trace/c
   #:satisfies port-re
   #:include listener/c
   [read-byte (->m byte?) 'use]
   [peek-byte (->m byte?) 'use]
   [close     (->m void?) 'close-stream]))

(define (output-stream/c listener/c)
  (object-trace/c
   #:satisfies port-re
   #:include listener/c
   [write-byte (->m byte? void?) 'use]
   [close      (->m void?)       'close-stream]))

(define (tcp-listener/c manager/c port)
  (object-trace/c
   #:extend manager/c
   [close  (->m void?) `(close-listener ,port)]
   [accept (->m (values (input-stream/c this-contract)
                        (output-stream/c this-contract)))]))

(define tcp-manager/c
  (object-trace/c
   #:satisfies manager-qea
   [listen (->dm ([port (integer-in 0 65535)])
                 [res (tcp-listener/c this-contract port)])
           `(listen ,port)]))
```

Figure 4 The TCP Interface.

Using `object-trace/c`, these logical specifications can be connected to the TCP interface itself; see Figure 4. Two differences from the example in Figure 3 are worth pointing out.

First, `tcp-manager/c` uses the dependent method contract form `->dm` [48] to produce an event that is dependent on the argument given to `listen`. This form permits naming the arguments and result, binding those names for use in the event constructor. Here, the listen event contains the `port` argument.

Second, `tcp-listener/c` uses `#:extend` to contribute events to the parent `manager/c` contract. As shown in Figure 3, `#:include` allows events to flow from "ancestor" to "descendant" object. By contrast, the `#:extend` keyword allows events to flow in the other direction, from "descendant" to "ancestor" object. Specifically, the TCP manager (i.e., the parent) must be aware if a TCP listener (i.e., the child) is closed on a particular port, because then that port becomes available for listening once again according to PortExclusivity. Using `#:extend manager/c` tells the contract to communicate close listener events from the listener to the manager. It turns out that StreamClose requires information to flow in the other direction because input and output streams (i.e., the children) must be closed if the underlying listener (i.e., the parent) is closed. This direction is covered using `#:include listener/c`. In sum, this example demonstrates that `object-trace/c` can accommodate information flow in both directions, even within the same interface, and can mix distinct notations to check all desired properties.

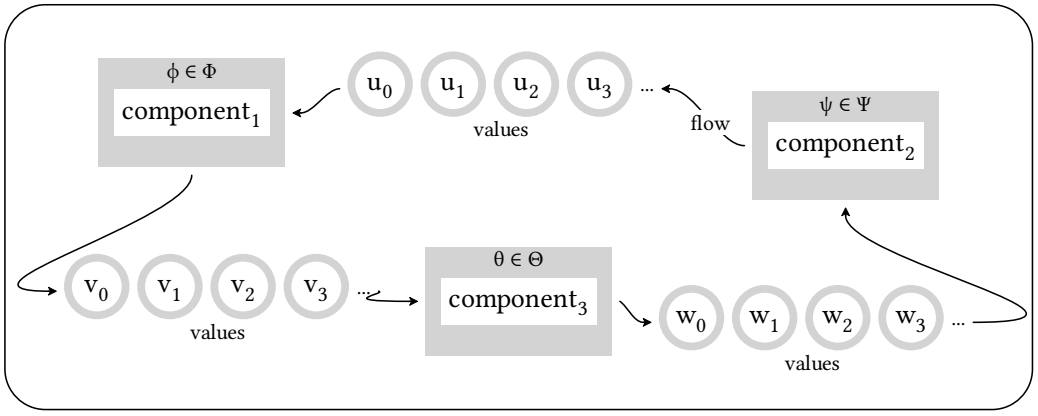


Figure 5 A System with Domain-Specific Contracts.

4 Implementing Domain-Specific Contracts

Implementing DSCs demands two ingredients: a low-level contract system and a mechanism for embedding domain-specific syntax. If a language lacks such a mechanism, a developer can use design patterns to construct embedded languages [41].

Figure 5 graphically summarizes how DSCs work. Each component is combined with a logical formulas (the gray frame). The component's developers choose the most appropriate logic to use, based on the property to be enforced. These logic languages come with translators that turn these specifications into predicates, which then become pieces of trace contracts. In the diagram, the enforcement mechanism comes into play with the sequences of values that components exchange. The contract code checks the sequence of values that a component receives and sends out. Gray borders on values in Figure 5 indicate that values may acquire a *proxy* layer that contains specification-checking code from the component's interface [16, 50]. A proxy interposes on interactions with the underlying object, e.g., a method call, and performs dynamic checks.

This section starts with a brief tutorial of Racket contracts, focusing on trace contracts (Section 4.1). Next, the section introduces essential features of Racket's domain-specific language (DSL) implementation system (Section 4.2) and what features readers should watch out for when implementing DSCs (Section 4.3). Lastly, the section gives some hints concerning the adaptation of this blueprint to languages with different DSL-programming facilities and contract systems (Section 4.4).

4.1 Low-Level Contract Mechanisms

For enforcement, DSCs rely on low-level contract mechanisms for stating and checking various kinds of properties: plain pre-conditions and post-conditions for functions; higher-order contracts for imposing invariants on foreign and behavioral values; and trace contracts for checking properties about sequences of calls.

In an untyped programming language, simple software contracts provide the programmer with the means to check type-like properties; in typed languages, they supplement method signatures not expressible in the type system. In Racket, contracts are also first-class values: they can be named, composed, and attached to values.

Higher-order contracts [16] go beyond plain assertions and describe invariants of behavioral values – such as objects – generating blame information[2] when dynamic[3] checks detect a violation. Here is a vanilla object contract for iterators:

```
(define iterator/c
  (object/c
   [has-next (->m boolean?)]
   [next     (->m any/c)]))
```

This contract specifies the method names along with a contract for the inputs and outputs of each method. When `iterator/c` is attached to an object, it ensures that the object comes with the expected methods and wraps it in a proxy. This wrapper performs checks on the inputs and outputs of the methods according to the stated expectations in method contracts (`->m`). Here, neither method consumes any arguments; both return one value.

An object contract merely enforces the pre-conditions and post-conditions on method arguments and results. It neither states nor checks the HasNext property. To do so requires the trace-contract extension to Racket's higher-order contract system.

A trace contract [37] collects values that flow through certain contract positions and enforces properties of the collected sequence. Hence, trace contracts can check temporal properties such as the HasNext property. Figure 6 shows such a contract.

```
(define iterator/c
  (trace/c ([evts symbol?])
    (object/c
     [has-next (and/c (->m boolean?) (apply/c [evts 'has-next]))]
     [next     (and/c (->m any/c)    (apply/c [evts 'next]))])
    (full (evts) has-next?)))

(define (has-next? s [okay? #false])
  (match s
    [(stream)                    #true]
    [(stream-cons 'has-next s*) (has-next? s* #true)]
    [(stream-cons 'next s*)      (and okay? (has-next? s*
       #false))]))))
```

Figure 6 Checking the HasNext Property.

The `iterator/c` contract is constructed with `trace/c`, which consists of three parts:

1. a declaration of a trace variable (`evts`) and its element type;
2. a contract (here an `object/c` contract); and
3. a predicate clause (`full`).

The declared trace variable names the trace of collected events. Annotations within the object contract push symbols into this trace whenever the corresponding method is called. Concretely, `apply/c` pushes the symbols `'has-next` and `'next` into `evts` when the corresponding method is called. Whenever a symbol is added to `evts`, the trace contract runs the predicate from the third part on the entire trace of events. For the iterator contract, the predicate ensures that the trace does not have two consecutive `'next` symbols.

[2] In a higher-order context, values enter contracts as they flow from one component to another. A contract violation may take place in the receiving component or in some other component, unrelated to the two involved in the contract. Hence, tracking the provenance of values is critical so that a contract violation can be related to a contract attachment [16].

[3] On occasion they can be statically verified [38, 53], but this direction is beyond the scope of this paper.

Importantly, each time a trace contract is attached to a value, it initializes a new trace just for that value. That means, different iterators have distinct traces that are separate from one another. State management is completely left to the trace-contract system and not the programmer. Hence, predicates over traces are plain (testable) functions.

Clearly, the direct use of this low-level contract mechanism "drowns" the essence of the iterator contract. By contrast, with a bit of training, every programmer can comprehend the contract in Figure 1 much more quickly.[4] First, it uses `object-trace/c` which provides a thin layer of syntactic sugar over the repetitive `apply/c` contracts. Second, it uses a one-line regular expression over method names instead of a recursive predicate.

Other features of `object-trace/c` are more complicated to desugar. Trace contracts, as originally implemented, were not expressive enough to support the `#:include` clause from Figure 3. The original implementation restricted contracts such that a predicate clause could only depend on traces from the same `trace/c` form. To support `#:include`, the underlying trace-contract system had to be modified to remove this restriction, allowing predicate clauses to depend on an arbitrary trace.

4.2 Domain-Specific Languages via Macros

Nearly all modern Lisp-like languages [9, 19, 23, 46, 47], and many languages outside the Lisp family [6, 13, 42], support syntactic abstraction via macros. A Lisp-style macro is a programmer-defined transformation from one syntax tree to another. In its most basic form, a macro is just a simple rewrite rule. But some languages, and Racket in particular, take the idea much further. Racket enables language-oriented programming [14] through macros, so programmers can construct rich DSLs and comingle expressions in these DSLs, without ever leaving the host language.

Racket provides one particularly useful tool for constructing DSC implementations: the `syntax-spec` library [2, 3]. Two features of `syntax-spec` make it suitable for creating new DSCs. First, developers can declaratively specify the language grammar, making it straightforward to develop and maintain the Racket-like syntax of DSCs. Second, `syntax-spec` is *binding aware*, meaning it understands the scope of variable declarations. It can thus reliably provide services such as rename refactoring or determine the free variables of a formula.

Figure 7 shows the Backus–Naur form (BNF) representation of the PLTL grammar and the `syntax-spec` definition for the same grammar in Racket. These grammars directly correspond to each other and it is easy to go back and forth. In order to make the `syntax-spec` grammar binding aware, a developer adds *binding specifications* such as the one for the `exists` form. Here, the binding specification states that in $\exists x.\phi$, the variable x is bound in ϕ.

Implementing the static semantics of a DSC is almost as straightforward as declaring its grammar. Take the monitorability criteria of PLTL [5], which says that a well-formed PLTL formula can be turned into a predicate over a trace if it satisfies the judgment on the left side of Figure 8. The equivalent Racket code is on the right side of the figure. Once again, the differences between these two are superficial. The Racket implementation pattern matches on the syntax using the `syntax-parse` system [10]. Each rule in the monitorability judgment corresponds to a branch in the match. In the PLTL pattern case, the function uses the free variables (`fvs`) feature mentioned earlier.[5]

[4] Most programmers are familiar with regular expressions, but other specification languages are less well-understood. For example, LTL is known to pose challenges for developers [21, 22]. Employing such specification languages may necessitate more training on the part of programmers.

[5] A straightforward feature to add to this static semantics is a monitorability search procedure. Some tools, such as MonPoly [5], attempt to rewrite nonmonitorable formulas into monitorable ones if possible. The same can easily be done with DSCs, immediately after the monitorability check fails.

$\phi \in \mathsf{Formula} = p \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \exists x.\phi \mid \bullet\,\phi \mid \phi\,\mathsf{S}\,\phi$

```
(syntax-spec
 (nonterminal pltl-formula
  #:allow-extension pltl-macro
  #:binding-space pltl-space
  (neg f:pltl-formula)
  (disj2 f1:pltl-formula f2:pltl-formula)
  (conj2 f1:pltl-formula f2:pltl-formula)
  (exists x:pltl-var f:pltl-formula)
  #:binding (scope (bind x) f)
  (previous f:pltl-formula)
  (since f1:pltl-formula f2:pltl-formula)
  p:pltl-pat))
```

Figure 7 Grammar of PLTL (Simplified).

Finally, if a formula satisfies the static semantics, it can be translated into a predicate. Figure 9 shows a simple translation for PLTL. It translates each formula into a tree of structs, which is then run using the PLTL interpreter provided by the library.

4.3 Features To Observe

While the use of `syntax-spec` greatly facilitates the declaration and translation of DSCs, this very simplicity obscures a number of aspects that come "for free" but may have to be realized separately in alternative implementations. When implemented with `syntax-spec`, domain-specific contracts signal syntax errors at the appropriate level, have direct support from the IDE, and are syntactically extensible.

Static Errors

A `syntax-spec` DSC implementation can automatically synthesize reasonable syntax errors for the new notation. For example, ill-formed PLTL formulas are identified like this:

```
> (define-pltl φ
    (S 'next))
expected more terms in:  (S 'next)
```

In ϕ, the S operator expects another operand, so the error points out an arity mismatch.

Similarly, a `syntax-spec` implementation can issue an informative message when static checking fails. As mentioned in the preceding subsection, a syntactically well-formed PLTL formula may be *invalid* because it cannot be translated into a predicate (as is):

```
> (define-pltl ψ
    (∃ (t) (¬ (• t))))
not monitorable in:  (¬ (• t))
```

Even if a PLTL formula is well-formed and valid, it can still be misused:

```
> (+ ψ 13)
cannot use PLTL formula here:  ψ
```

$$\frac{\rho \vdash \phi \qquad \rho \vdash \psi}{\rho \vdash \phi \vee \psi}$$

$$\frac{\rho_\phi \vdash \phi \qquad \rho_\psi \vdash \psi \qquad \rho_\phi \subseteq \rho_\psi}{\rho_\phi \cup \rho_\psi \vdash \neg\phi \wedge \psi}$$

$$\frac{\rho_\phi \vdash \phi \qquad \rho_\psi \vdash \psi \qquad \rho_\phi \subseteq \rho_\psi}{\rho_\phi \cup \rho_\psi \vdash \neg\phi \,\mathsf{S}\, \psi}$$

$$\frac{\rho_\phi \vdash \phi \qquad \rho_\psi \vdash \psi \qquad \rho_\phi \subseteq \rho_\psi}{\rho_\phi \cup \rho_\psi \vdash \phi \,\mathsf{S}\, \psi}$$

$$\frac{\rho_\phi \vdash \phi \qquad \rho_\psi \vdash \psi}{\rho_\phi \cup \rho_\psi \vdash \phi \wedge \psi} \qquad \frac{\rho \cup \{x\} \vdash \phi}{\rho \vdash \exists x.\phi}$$

$$\frac{\rho \vdash \phi}{\rho \vdash \bullet\phi} \qquad \frac{\rho = \mathrm{FV}(p)}{\rho \vdash p}$$

```
(define (monitorable? stx)
  (syntax-parse stx
    [(disj2 f g)
     (and (equal? (fvs #'f) (fvs #'g))
          (monitorable? #'f)
          (monitorable? #'g))]
    [(conj2 (neg f) g)
     (and (subset? (fvs #'f) (fvs #'g))
          (monitorable? #'f)
          (monitorable? #'g))]
    ── similar cases ──
    [(conj2 f g)
     (and (monitorable? #'f)
          (monitorable? #'g))]
    [(exists (x:id ...) f)
     (monitorable? #'f)]
    ── similar cases ──
    [(neg _) #false]
    [p #true]))
```

Figure 8 Static Semantics for PLTL (Simplified).

```
(define-syntax (translate stx)
  (syntax-parse stx
    [(_ (neg f))             #'(neg-s (translate f))]
    [(_ (disj2 f g))         #'(disj2-s (translate f) (translate g))]
    ── similar cases ──
    [(_ (exists (x ...) f)) #'(exists-s (set 'x ...) (translate f))]
    [(_ x:id)                #'x]
    [(_ p)                   #'(translate-pattern p)]))
```

Figure 9 Translator for PLTL (Simplified).

The `define-pltl` form creates definitions in a separate *binding space*, as required by the one-line `#:binding-space` option in Figure 7. Languages, including domain-specific notations, are often composed of several syntactic categories that are mixed at particular interface points. For example, Java has a syntactic category of expressions and types that are interleaved in a certain way; definitions created in one do not exist in the other. The `syntax-spec` implementation of PLTL creates its own binding space, distinct from the one for expressions, meaning that a reference to a PLTL name from an expression context signals a static error.

User Extensibility

PLTL comes with just two primitive temporal operators. Although derived operators add no expressive power to the logic, they are necessary for writing concise specifications. While the PLTL library could provide direct support for such derived operators, doing so has the serious downside of complicating the static semantics, the translator, and the runtime system, which would have to deal with each kind of operator separately.

Instead, a `syntax-spec` implementation of a DSC can be made syntactically extensible with a single annotation: `#:allow-extension`. This `#:allow-extension` option opens up the `pltl-formula` nonterminal for simple macro extensibility. Using this capability, a user can define additional operators in a straightforward manner:

```
(define-pltl-rule ⊤
  (? (λ _ #true)))

(define-pltl-rule (⧫ ϕ)
  (S ⊤ ϕ))
```

The `syntax-spec`-generated `define-pltl-rule` form defines a syntactic transformation that rewrites all instances of $\top$ and $\blacklozenge\phi$ in a PLTL formula: $\top$ is rewritten to a predicate that always returns `#true`, meaning a machine that succeeds on all traces of collected events; and $\blacklozenge$ is rewritten to a formula that requires nothing to be true until ϕ holds.

These syntactic transformations are seamlessly integrated into PLTL. For example, a user can write a utility library with these abbreviations and export them for use in other modules. In short, user-extensible transformations make the PLTL library flexible to use and simplify its implementation tremendously.

IDE Integration

The `syntax-spec` library facilitates the integration of DSC implementations with Racket IDEs. One such IDE is DrRacket [15]; another one is Racket Mode for Emacs.[6]

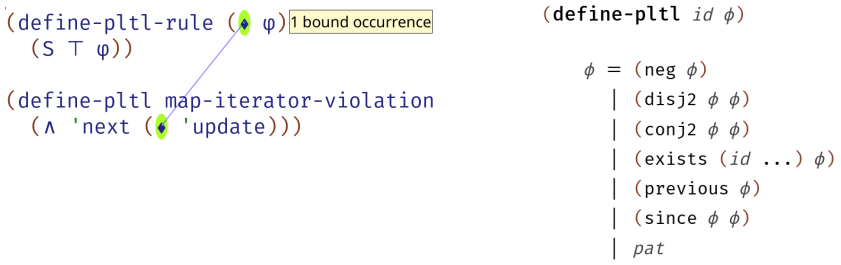


Figure 10 IDE Services for PLTL.

Figure 10 shows two such services. First, when a user hovers over an identifier, arrows are drawn from the binding occurrence of an identifier to every reference (Figure 10, left). Here, the definition of $\blacklozenge$ points to its use in the formula. The IDE has a rich understanding of the binding structure of the DSC, which can be used to support reliable program transformations including rename refactorings. Second, if a DSC comes equipped with documentation, as PLTL does, this documentation can be displayed directly in the IDE. When the cursor is placed inside an identifier with recognized documentation, a box containing a summary of the documentation appears in the upper-right corner of the IDE. For `define-pltl`, that summary is the grammar of primitive PLTL forms (Figure 10, right). This service is available because Racket's documentation language Scribble [18] is binding aware.

[6] This major mode is maintained by Greg Hendershott; see `https://racket-mode.com`.

Compositionality

It is also possible for users to compose contracts that may have been written independently into a single interface contract. For example, the iterator contract from Figure 3 can easily be augmented to additionally check the HasNext property, even though these specifications may have been defined separately:

```
(define (map-iterator*/c map/c)
  (and/c has-next/c (map-iterator/c map/c)))
```

The `and/c` form comes from Racket's contract library and enforces that values satisfy both conjuncts (in order). It may compose *any* kind of contract that compiles to the already-mentioned proxy mechanism in Racket.

Blame Assignments

In the case of Racket and the DSCs in this paper, contracts have a blame-assignment mechanism [16]. That is, once a dynamic contract check discovers a violation, it raises an exception that assigns blame to a particular component. Here is an excerpt from the error message that explains a violation of the HasNext property:

```
> (define m (new map% ——))
> (define c (send m keys))
> (define t (send c iterator))
> (send t next)
next: contract violation
blaming: top level
```

An iterator created from a collection has its `next` method called without a corresponding call to `has-next`. The contract system issues a blame assignment that points to the method that failed and the component that violated the specification – here, the top-level code. This information might be useful for tracking down the source of bugs in large systems [27, 28, 29].

4.4 Ingredients for Adding DSCs to a Programming Language

The DSC architecture is generic and should apply to many programming languages, with varying degrees of ease. It requires two essential ingredients: (1) facilities for creating embedded DSLs for the desired logics, such as macros; and (2) a mechanism for dynamically checking logical formulas, such as contracts. The quality of these ingredients in a language determines how easy it is to instantiate this architecture and how many of the features from the preceding subsection can be realized. For example, a programming language with powerful metaprogramming facilities reduces the notational overhead on programmers; without such facilities, a DSC implementer may have to resort to shallow or deep embedding techniques [20], which may impose an extra burden on developers.

Metaprogramming

A metaprogramming system enables programmers to extend their language with new linguistic constructs and collect them in libraries or frameworks. Ideally, it permits blurring the distinction between built-in constructs and user-defined ones. Importantly, metaprogramming facilities exist *inside* the programming language. Depending on the chosen language, metaprogramming facilities can be used in the small and large: from quick syntactic abbreviations all the way up to full feature-rich special-purpose DSLs. For DSCs, the metaprogramming facilities must be powerful enough to create the *embedded* DSLs necessary for expressing logical specifications.

Different languages have vastly different metaprogramming mechanisms. Some languages provide lightweight syntactic metadata: decorators in Python and annotations in Java. In other languages, such as Ruby, the syntax is sufficiently flexible and the semantics sufficiently dynamic that a DSC notation can be built without any explicit language support. Another major class of metaprogramming facilities are those based on (syntax-tree) *macros*: compile-time rewriting rules. Before code generation, the compiler applies these rules to obtain the program in a core syntax – a process called *macro expansion*. Languages in the Lisp family, such as Clojure [23], are especially well-known for macros, but others including Rust and Scala have adopted them too. Parenthetical syntax is not necessary [17].

Racket is well-suited for defining DSCs as it provides a great deal of support for language-oriented programming [14]. While its advanced macro system essentially makes up a nearly complete compiler API, the implementation of a DSC needs few of these capabilities.

Dynamic Checking

A contract system consists of a notation and a run-time mechanism for dynamically checking a program's behavior. The notation has to provide the means to identify relevant events, e.g., "the first argument to the method call `o.m`," and the predicate that should hold of such events, e.g., "is a positive integer." These statements are usually understandable as ordinary program code – Boolean-valued assertions – and are easily translated into run-time checks. The runtime system issues a descriptive error when any of these checks fail.

Modern contract systems permit programmers to monitor specifications over any kind of value, including higher-order values such as objects, functions, and even the kind of first-class classes found in Python, Racket, and Ruby. The corresponding run-time checks rely on proxies that wrap values flowing through contracts and contain the Boolean-valued run-time checks. Direct support for proxy values is available in a number of languages, e.g., JavaScript [51]. Even if proxies are unavailable, implementing proxies in some dynamic languages does not require explicit support. Ruby, for example, supports objects that can interpose on all `send` method invocations. Such a facility has been used to implemented higher-order contracts [49]. So, while contracts are implemented as a core feature in some languages, such as Eiffel [33, 34], they can often be implemented as a library, especially if proxies are available.

Racket comes with a pioneering higher-order contract system [16]. The contract notation is realized as an embedding, via a rich set of contract combinators; the base contracts are predicates as in any other system. With trace contracts [37], programmers can articulate predicates over traces of values that flow through particular points in a contract. Trace contracts make it particularly straightforward to bridge the gap between the DSC specifications presented here and the low-level contract system itself.

5 Evaluating the Domain-Specific Contracts Approach

An assessment of the DSC idea and its instantiation with trace contracts must address several aspects: (1) whether DSCs can accommodate existing logic notations; (2) what developer-facing features these implementations provide; and (3) how precisely DSCs can report specification violations. The first two points call for identifying and implementing a corpus of logic notations for specifications in the literature (Section 5.1) and for a qualitative assessment of them along several dimensions (Section 5.2). The last point is a matter of implementation quality; an extensive test suite can validate that the blame assignment points to the problematic party for some common example properties from the literature (Section 5.3). In addition, this section relates the DSC idea to prior work (Section 5.4).

5.1 Corpus of Logics

The literature proposes a wide variety of logic notations for specifications that are suitable to check with trace contracts. Here are six common ones:

Past-Time Linear Temporal Logic (PLTL) is a past-time variant of the more commonly used future-time linear temporal logic. See Section 2 for a complete description of PLTL and an example.

Regular Expressions (RE) are a simple formalism equivalent in expressive power to finite state machines [26]; they are a well-known way for specifying parts of the behavior of a software system [45].

Finite State Machines (NFA, DFA) have a finite number of states and transitions between them. A deterministic finite-state machine (DFA) denotes the sequences of events that, when run on the state machine, follow a unique path that ends in an accepting state [43]. A non-deterministic finite-state machine (NFA) denotes the sequences of events that, when run on the state machine, end in an accepting state along *some* path.

Quantified Event Automata (QEA) resemble DFAs but are augmented with extra features [4]. As the name suggests, the machine supports quantification. Conceptually, a QEA represents a family of automata – one for every possible instantiation of quantified variables. Practical implementations of QEA more efficiently compute transitions via a compact representation of this family of automata.

Stream Logic (SL) describes properties by defining stream equations [11]. A stream is a sequence of events that is accessible with a finite lookback. The dynamic checks for SL specifications compute the value for each stream at the current index, making use of the static dependencies of equations to solve them in the correct order. Irrelevant values, i.e., ones that are beyond the finite lookback, are automatically discarded.

All these formalisms have been implemented for use in Racket DSCs.

5.2 Qualitative Assessment

The implementations of these six logic notations have distinct qualities from the perspective of language creators and users. This subsection reports those qualities in terms of four dimensions: binding positions, definitions, static semantics, and macro extensibility. Table 1 summarizes the discussion in a table.

Table 1 Summary Qualitative Assessment of Logic DSLs.

	PLTL	RE	DFA	NFA	QEA	SL
Binding Rules	✓	×	×	×	✓	✓
Definitions	✓	✓	×	×	×	×
Static Semantics	✓	×	×	×	×	✓
Macro Extensible	✓	✓	×	×	×	×

Binding Positions

Some DSLs are variable-free; others come with variable declarations that introduce binding positions and determine the scope of a declaration. Since the `syntax-spec` system comes with a binding-rules mechanism, it allows DSC creators to specify which pieces of a formula are variable references, which are binding positions, and the scope of bindings. By declaring the binding structure, DSC implementations can automatically support IDE services, static-semantics passes, and translations into predicates.

Logic notations without quantification, i.e., RE, DFA, and NFA, do not require binding rules. Specifically, the state-machine notations do not use an up-front declaration of states; if they did, binding rules would be required. By contrast, both PLTL and QEA come with variable quantification, and quantified variables are binders. In SL, equations must reference other equations or primitive event streams. The binding rules for SL guarantee that there are no free variables referencing an undefined stream.

Separate Definitions

For some logic notations, formulas can be defined piecemeal via the binding space mechanism. Developers can build up libraries of reusable formulas that others can then compose into specifications of modules, classes, and other components. For such compositional notations, definition forms are a highly useful mechanism. While the state-machine notations from the literature do not seem compositional, both RE and PLTL greatly benefit from the introduction of definition forms.

Static Semantics

Some specification notations impose criteria that determine whether well-formed formulas are valid. If an implementation can enforce such static-semantics constraints, it is superior to plain frameworks or functional libraries because it informs developers at compile time about basic mistakes, instead of delaying the discovery of specification mistakes until run time. Programmers can thus avoid unexpected run-time errors and dynamic debugging due to invalid specifications.

While Racket's `syntax-spec` library automatically guarantees that logical specifications are well-formed and closed with respect to any binding rules, the static semantics of DSCs must be written as an additional check. Of the investigated notations, PLTL and SL must satisfy such criteria. Section 4 covers the monitorability criteria for PLTL. For SL, the DSL implementation statically checks that the dependency graph of the stream equations has no closed walk with a total weight of zero, where the weights are determined by indices into streams. This check statically generates the dependency graph from the stream equations via local macro expansion, even when references to streams are embedded inside Racket sub-expressions.

Macro Extensibility

When logical formulas are compositional, programmers benefit from a macro-extensible language. The RE and PLTL notations make heavy use of this feature because many useful operators can be derived from a small set of primitive operators. By expanding the surface syntax to a small core language, the implementation can be organized like an ordinary compiler (as in functional languages such as Haskell, ML, or Scheme). Equally important, macro extensibility enables users to create syntactic abstractions when plain functional abstractions do not suffice to hide repeated patterns in formulas.

In summary, the implementation of these specification notations preserve all desirable aspects (e.g., the static checks) and, indeed, go beyond usual implementations, offering users additional affordances. Furthermore, DSCs are embedded into the host, meaning the host's IDEs (DrRacket, Emacs) support DSCs essentially as well as plain Racket programs. When implemented correctly, users do not notice any friction.

5.3 Blame Assignment

One claimed benefit of contracts is that error messages come with blame information identifying the source of the specification violation. Blame assignment in higher-order settings, and even more so for temporal properties, is subtle. In general, blame assignment points to the components that contribute to a violation of a specification.

To understand whether blame assignment from the underlying contract system works well for DSCs, it is necessary to adapt examples from the literature and to test violations on sample code snippets. That is, unit tests should validate that the Racket implementation actually produces blame assignments in accordance with the source of the bug.

Table 2 Sample Properties.

Property	Logic / SLOC	Description
MapIterator	PLTL (7), RE (1)	An iterator constructed from a collection, which is itself constructed from a map, is invalidated once the underlying map is mutated.
HasNext	QEA (6), RE (1)	A call to the `has-next` method must precede every call to the `next` method.
HashCode	QEA (5), SL (7)	An object's `hash-code` must not change so long as it is being used as a key in a map.
ClosedReader	SL (8)	A call to the `read` method of a reader object must not take place after the underlying input stream has been closed.
SetFromMap	SL (6)	Creating a set from a map renders the latter unusable.
RemoveOnce	RE (1)	An iterator's `remove` method can be called only once per invocation of the `next` method.
StreamClose	RE (1)	A stream may be used until it is closed or the underlying TCP listener is closed.
PortExclusivity	QEA (5)	Listeners can be created for a port, so long as there is no active listener already associated with that port.

Table 2 list some sample properties from the literature, which logics they were implemented in, and how many lines it took to write each formula. All explain properties of the Java API that are stated informally but may or may not be checked via the assertion system. Other systems can enforce these properties [30], and so can the Racket realization of DSCs.

The test suite for the Racket implementation contains adaptations for all these properties. The corresponding code come with components that violate the properties. Unit tests ensure that checking these DSC specifications catches all violations and that the relevant blame assignment points to the buggy component.

5.4 Prior Work

Dimoulas et al. [12] present the first and only analysis in the contract-systems literature that explicitly mentions the need for a contract-specific notation. Their analysis explains a contract system as resting on three pillars: (1) a notation for expressing contracts; (2) a set of interposition points in the run-time system that allows inspecting values; and (3) a syntactic

mechanism for attaching contracts to values. Generally speaking, research on contracts has tightly coupled the three, associating one contract notation with specific interposition points and attachment mechanisms. Instead, the DSC architecture proposes that any language with contracts benefits from a *loose coupling* between contract notations on the one hand, and attachment mechanisms and interposition points on the other.

JavaMOP [7], a run-time verification system, is the only existing system with a similarly loose coupling. From a high-level perspective, JavaMOP rests on an aspect-like interposition system, dubbed monitor-oriented programming (MOP) [32]. Since JavaMOP is a run-time verifier, programmers do not formulate method-level or even class-level contracts, but whole-program properties. The JavaMOP framework compiles these properties into monitoring code snippets that are woven into the program at appropriate join points and that send values to a parallel property-checking process.

Unlike other run-time verification systems, JavaMOP permits programmers to choose the property notation best suited to the task and, indeed, allows them to create and plug in their own notation. From this angle, JavaMOP resembles the Racket implementation of the DSC architecture.

However, three differences stand out. First, the DSC proposal is intended to be realizable in any programming language. As detailed in Section 4, an implementation can be realized with minimal effort in languages with macro systems such as Racket, Clojure, Julia, Rust, or Scala. In languages without macro systems, an implementation may require a bit more labor, depending on existing features, but using conventional techniques for embedding domain-specific languages make DSCs feasible, even without significant engineering resources. Second, the end-product of DSCs is more convenient for programmers to use than a toolchain-based solution such as JavaMOP. A library can be installed directly from a language's package manager and used immediately. Finally, DSCs provide additional benefits beyond what JavaMOP delivers, e.g., IDE integration and blame assignment (from the underlying contract system, if available). The remainder of this section provides a detailed comparison.

Logic Plugins

JavaMOP users can develop their own logic plugins to add new, possibly domain-specific, specification languages. These plugins are essentially ordinary compilers from the surface syntax to Java code. Constructing such a compiler requires the use of traditional compiler generator tools, such as a parser generator, and a solid understanding of the semantics and behavior of the target language. As a result, developing a new logic or extending an existing one is challenging and not a lightweight exercise.

In the context of the proposed architecture, DSCs rely on the host language's existing metaprogramming facilities or established DSL implementation patterns. In a language with a macro system, such as Racket, new DSCs with sophisticated features can be created easily with a combination of declarative grammar specifications and recursive functions. And, if the host allows for declarative DSL specifications, these DSCs are maintainable in a straightforward manner.

Linguistic Integration

A byproduct of JavaMOP's extra-linguistic approach is that logical specifications are syntactically isolated from the rest of the program. JavaMOP specifications are typically added as comments to the code or as entirely separate text files. As a consequence, these artifacts end up as second-class citizens within a software project.

By contrast, DSCs make logical specifications an integrated part of the underlying project; they exist *within* the program. As such, all host-language tools continue to work with logical specifications. Notably, DrRacket's existing IDE services seamlessly extend to DSC code.

Trace Management

JavaMOP takes a global perspective on monitoring. System events are captured and stored in a global sequence. This monitoring strategy is central to the run-time verification approach. Since checking globally generated traces requires a slicing technique, JavaMOP's implementation makes slicing independent of the underlying logic to reduce the burden on the developer of a new logic notation [7, 24]. In the end, however, the program-global property approach remains an obstacle to reasoning about code in a compositional manner. Slicing also requires a notion of equality on objects which, if one is not defined already, relies on pointer equality. Depending on pointer equality for objects violates a key principle of object-oriented programming [8].

Contracts, by contrast, often eliminate the need for slicing because the state for a given value is freshly initialized when the value passes through an attachment boundary. Monitoring is local. Thus, the MapIterator property from Section 2 can be attached to a specific object and remains quantifier free; the equivalent formula in JavaMOP requires quantifiers and hence a slicing implementation in the monitoring process. Localized monitoring also unlocks unique advantages such as blame assignment (if the underlying contract system provides it). Slicing is still occasionally useful in contracts. For example, Section 3 shows slicing based on port numbers. A notion of equality is defined for port numbers, so the principles of object-oriented design are maintained.

Run-time Enforcement

As originally published, JavaMOP supported only one run-time enforcement mechanism. The AspectJ [25] weaver would statically inject monitoring code into a system based on the given specification. This requires AspectJ as a compile-time dependency and alters the ordinary build process for a Java project. Now, JavaMOP supports a dynamic enforcement mechanism known as the JavaMOP agent. With this mode, an instrumented JVM supplies the means to record system events during execution. The JavaMOP agent offers some advantages over the static approach, but requires a specialized runtime. Both of these mechanisms are somewhat unique to Java and are not readily available in other languages.

Contracts use a completely different implementation approach. While first-order contracts essentially get away with Boolean-valued assertions, higher-order and trace-contract systems rely on proxy objects to monitor a running program. If the language does not support proxy objects directly, implementing a reasonable approximation is feasible. The key is that proxies enable a simple, local-enforcement mechanism, which is crucial for blame assignment in higher-order settings.

6 A Future for Domain-Specific Contracts

Racket makes it straightforward to instantiate the DSC idea. It comes with an expressive library of contract combinators and a powerful facility for creating domain-specific notations. Leveraging the latter allows the creators of DSCs to implement notations with little effort and translate them to contract expressions. End-users can employ a library that provides DSCs just like any other dependency – critically, while continuing to use, and benefiting from, existing IDEs.

While the Racket implementation is particularly straightforward, several programming languages offer similar combinations of metaprogramming and contract features. For example, Clojure has ported most of Racket's contract system and comes with its own macro system. Scala too has a rich metaprogramming system for embedding DSLs [44] and could be equipped with a contract system based on Java's assertion system.

This paper has discussed the DSC idea only in the context of trace contracts with temporal specification languages. However, the idea of combining contracts with domain-specific specification languages goes far beyond that particular instantiation. For example, effect-handler contracts [36] combined with an authorization logic [1] yields authorization contracts [35] that can enforce security properties. Or, network-aware contracts [52] combined with a DSL for describing the legal moves of a game [12] yields the rule checker for an implementation of an online game. Indeed, there are plenty of other combinations not yet studied and worthy of exploration from the DSC perspective. Contracts and DSLs are useful alone. But together, they are better.

References

1 Owen Arden, Jed Liu, and Andrew C. Myers. Flow-Limited Authorization. In *Computer Security Foundations (CSF)*, 2015. doi:10.1109/csf.2015.42.

2 Michael Ballantyne and Matthias Felleisen. Injecting Language Workbench Technology into Mainstream Languages. In *Eelco Visser Commemorative Symposium (EVCS)*, 2023. doi:10.4230/OASIcs.EVCS.2023.3.

3 Michael Ballantyne, Mitch Gamburg, and Jason Hemann. Compiled, Extensible, Multi-Language DSLs (Functional Pearl). In *International Conference on Functional Programming (ICFP)*, 2024. doi:10.1145/3674627.

4 H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. Rydeheard. Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In *Formal Methods (FM)*, 2012. doi:10.1007/978-3-642-32759-9_9.

5 David Basin, Felix Klaedtke, and Eugen Zălinescu. The MonPoly Monitoring Tool. In *Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES)*, 2017. doi:10.29007/89hs.

6 Eugene Burmako. Scala Macros: Let Our Powers Combine! In *Workshop on Scala*, 2013. doi:10.1145/2489837.2489840.

7 Feng Chen and Grigore Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2007. doi:10.1145/1297027.1297069.

8 William R. Cook. On Understanding Data Abstraction, Revisited. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2009. doi:10.1145/1640089.1640133.

9 Ryan Culpepper. Fortifying Macros. *Journal of Functional Programming (JFP)*, 2012. doi:10.1017/s0956796812000275.

10 Ryan Culpepper and Matthias Felleisen. Fortifying Macros. In *International Conference on Functional Programming (ICFP)*, 2010. doi:10.1145/1863543.1863577.

11 B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H.B. Sipma, S. Mehrotra, and Z. Manna. LOLA: Runtime Monitoring of Synchronous Systems. In *Temporal Representation and Reasoning (TIME)*, 2005. doi:10.1109/TIME.2005.26.

12 Christos Dimoulas, Max S. New, Robert Bruce Findler, and Matthias Felleisen. Oh Lord, Please Don't Let Contracts Be Misunderstood (Functional Pearl). In *International Conference on Functional Programming (ICFP)*, 2016. doi:10.1145/2951913.2951930.

13 Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: Library-Based Syntactic Language Extensibility. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2011. doi:10.1145/2048066.2048099.

14 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A Programmable Programming Language. *Communications of the ACM (CACM)*, 2018. doi:10.1145/3127323.

15 Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A Programming Environment for Scheme. *Journal of Functional Programming (JFP)*, 2002. doi:10.1017/S0956796801004208.

16 Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *International Conference on Functional Programming (ICFP)*, 2002. doi:10.1145/581478.581484.

17 Matthew Flatt, Taylor Allred, Nia Angle, Stephen De Gabrielle, Robert Bruce Findler, Jack Firth, Kiran Gopinathan, Ben Greenman, Siddhartha Kasivajhula, Alex Knauth, Jay McCarthy, Sam Phillips, Sorawee Porncharoenwase, Jens Axel Søgaard, and Sam Tobin-Hochstadt. Rhombus: A New Spin on Macros without All the Parentheses. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2023. doi:10.1145/3622818.

18 Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the Book on Ad Hoc Documentation Tools. In *International Conference on Functional Programming (ICFP)*, 2009. doi:10.1145/1596550.1596569.

19 Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. URL: https://racket-lang.org/tr1/.

20 Jeremy Gibbons and Nicolas Wu. Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl). In *International Conference on Functional Programming (ICFP)*, 2014. doi:10.1145/2628136.2628138.

21 Ben Greenman, Siddhartha Prasad, Antonio Di Stasio, Shufang Zhu, Giuseppe De Giacomo, Shriram Krishnamurthi, Marco Montali, Tim Nelson, and Milda Zizyte. Misconceptions in Finite-Trace and Infinite-Trace Linear Temporal Logic. In *Formal Methods*, 2024. doi:10.1007/978-3-031-71162-6_30.

22 Ben Greenman, Sam Saarinen, Tim Nelson, and Shriram Krishnamurthi. Little Tricky Logic: Misconceptions in the Understanding of LTL. *The Art, Science, and Engineering of Programming*, 2023. doi:10.22152/programming-journal.org/2023/7/7.

23 Rich Hickey. A History of Clojure. In *History of Programming Languages (HOPL)*, 2020. doi:10.1145/3386321.

24 Dongyun Jin, Patrick O'Neil Meredith, Dennis Griffith, and Grigore Rosu. Garbage Collection for Monitoring Parametric Properties. In *Programming Language Design and Implementation (PLDI)*, 2011. doi:10.1145/1993498.1993547.

25 Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *European Conference on Object-Oriented Programming (ECOOP)*, 2001. doi:10.1007/3-540-45337-7_18.

26 S.C. Kleene. Representation of Events in Nerve Nets and Finite Automata. In *Automata Studies*. Princeton University Press, Princeton, N.J., 1956. doi:10.1515/9781400882618-002.

27 Lukas Lazarek, Ben Greenman, Matthias Felleisen, and Christos Dimoulas. How To Evaluate Blame For Gradual Types. In *International Conference on Functional Programming (ICFP)*, 2021. doi:10.1145/3473573.

28 Lukas Lazarek, Ben Greenman, Matthias Felleisen, and Christos Dimoulas. How To Evaluate Blame For Gradual Types, Part 2. In *International Conference on Functional Programming (ICFP)*, 2023. doi:10.1145/3607836.

29 Lukas Lazarek, Alexis King, Samanvitha Sundar, Robert Bruce Findler, and Christos Dimoulas. Does Blame Shifting Work? In *Principles of Programming Languages (POPL)*, 2020. doi:10.1145/3371133.

30 Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. How Good are the Specs? A Study of the Bug-Finding Effectiveness of Existing Java API Specifications. In *Automated Software Engineering (ASE)*, 2016. doi:10.1145/2970276.2970356.

31 Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. The Glory Of The Past. In *Logics of Programs*, 1985. doi:10.1007/3-540-15648-8_16.

32 Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An Overview of the MOP Runtime Verification Framework. *International Journal on Software Tools for Technology Transfer*, 2011. doi:10.1007/s10009-011-0198-6.

33 Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

34 Bertrand Meyer. Applying "Design by Contract". *Computer*, 1992. doi:10.1109/2.161279.

35 Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong. Extensible Access Control with Authorization Contracts. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2016. doi:10.1145/2983990.2984021.

36 Cameron Moy, Christos Dimoulas, and Matthias Felleisen. Effectful Software Contracts. In *Principles of Programming Languages (POPL)*, 2024. doi:10.1145/3632930.

37 Cameron Moy and Matthias Felleisen. Trace Contracts. *Journal of Functional Programming (JFP)*, 2023. doi:10.1017/S0956796823000096.

38 Phúc C. Nguyễn, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. Soft Contract Verification for Higher-Order Stateful Programs. In *Principles of Programming Languages (POPL)*, 2018. doi:10.1145/3158139.

39 Oracle. Iterator(Java SE 23 & JDK 23). https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/Iterator.html, 2024.

40 Oracle. Map (Java SE 23 & JDK 23). https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/Map.html, 2024.

41 Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 2009.

42 The Rust Project. The Rust Reference: Procedural Macros. https://doc.rust-lang.org/reference/procedural-macros.html, 2024.

43 M. O. Rabin and D. Scott. Finite Automata and their Decision Problems. *IBM Journal of Research and Development*, 1959. doi:10.1147/rd.32.0114.

44 Tiark Rompf and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Generative Programming and Component Engineering (GPCE)*, 2010. doi:10.1145/1868294.1868314.

45 Usa Sammapun and Oleg Sokolsky. Regular Expressions for Run-Time Verification. In *Automated Technology for Verification and Analysis (ATVA)*, 2003.

46 Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. Revised^6 Report on the Algorithmic Language Scheme. *Journal of Functional Programming*, 2009. doi:10.1017/s0956796809990074.

47 Guy L. Steele. *Common Lisp the Language*. Digital Press, 1990.

48 T. Stephen Strickland, Christos Dimoulas, Asumu Takikawa, and Matthias Felleisen. Contracts For First-Class Classes. *Transactions on Programming Languages and Systems (TOPLAS)*, 2013. doi:10.1145/2518189.

49 T. Stephen Strickland, Brianna M. Ren, and Jeffrey S. Foster. Contracts For Domain-Specific Languages In Ruby. In *Dynamic Languages Symposium (DLS)*, 2014. doi:10.1145/2661088.2661092.

50 T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and Impersonators: Run-Time Support for Reasonable Interposition. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2012. doi:10.1145/2384616.2384685.

51 Tom Van Cutsem and Mark S. Miller. Proxies: Design Principles for Robust Object-oriented Intercession APIs. In *Dynamic Languages Symposium (DLS)*, 2010. doi:10.1145/1869631.1869638.

52 Lucas Waye, Stephen Chong, and Christos Dimoulas. Whip: Higher-Order Contracts for Modern Services. In *International Conference on Functional Programming (ICFP)*, 2017. doi:10.1145/3110280.

53 Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static Contract Checking for Haskell. In *Principles of Programming Languages (POPL)*, 2009. doi:10.1145/1480881.1480889.