5th International Conference on Formal Structures for Computation and Deduction

FSCD 2020, June 29–July 6, 2020,
Paris, France (Virtual Conference)

Edited by

Zena M. Ariola

LIPIcs – Vol. 167 – FSCD 2020 www.dagstuhl.de/lipics

Editors

Zena M. Ariola
University of Oregon, Eugene, Oregon, USA
ariola@cs.uoregon.edu

ACM Classification 2012
Theory of computation → Models of computation; Theory of computation → Formal languages and automata theory; Theory of computation → Logic; Theory of computation → Semantics and reasoning; Software and its engineering → Language features; Software and its engineering → Formal language definitions; Software and its engineering → Formal methods

ISBN 978-3-95977-155-9

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/978-3-95977-155-9.

Publication date
June, 2020

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.

Digital Object Identifier: 10.4230/LIPIcs.FSCD.2020.0

ISBN 978-3-95977-155-9 **ISSN 1868-8969** **https://www.dagstuhl.de/lipics**

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.

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Invited Talks

Regular Papers

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

System Descriptions

Preface

This volume contains the proceedings of the 5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020). The conference was planned to be held June 29 through July 6, 2020, in Paris, France, co-located with the 10th International Joint Conference on Automated Reasoning (IJCAR 2020). Due to the COVID-19 pandemic, FSCD 2020 was instead held virtually. The conference (`http://fscd-conference.org/`) covers all aspects of formal structures for computation and deduction, from theoretical foundations to applications. Building on two communities, RTA (Rewriting Techniques and Applications) and TLCA (Typed Lambda Calculi and Applications), FSCD embraces their core topics and broadens their scope to include closely related areas in logics and proof theory, new emerging models of computation, semantics and verification in new challenging areas.

The FSCD program featured four invited talks given by René Thiemann (University of Innsbruck), Andrew Pitts (University of Cambridge), Simona Ronchi della Rocca (Universitá di Torino), and Brigitte Pientka (McGill University). FSCD 2020 received 81 submissions with contributing authors from 28 countries. The program committee consisted of 35 members from 16 countries. Each submitted paper has been reviewed by at least three PC members with the help of 98 external reviewers. The reviewing process, which included a rebuttal phase, took place over eight weeks. A total of 28 regular research papers and 5 system description papers were accepted for publication and are included in these proceedings. The Program Committee awarded two FSCD 2020 Best Paper Awards by Junior Researchers: Petar Vukmirović, Alexander Bentkamp and Visa Nummelin for the paper "Efficient Full Higher-Order Unification", and Andrej Dudenhefner for the paper "Undecidability of Semi-unification on a Napkin". This year we also introduced the Best System Description Award by Junior Researchers, and the winner was Ankush Das for the paper "Resource-Aware Session Types with Arithmetic Refinements" (co-authored with Frank Pfenning).

In addition to the main program, 9 FSCD-associated workshops were held, also virtually:

• IWC	International Workshop on Confluence
• IFIP WG 1.6	Rewriting IFIP Meeting - 23rd edition
• Linearity & TLLA	Joint workshop on Linearity and Trends in Linear Logic and Applications
• UNIF	International Workshop on Unification
• WPTE	International Workshop on Rewriting Techniques for Program Transformations and Evaluation
• WiL2020	Women in Logic
• HoTT/UF	Workshop on Homotopy Type Theory/Univalent Foundations
• GeoCat	Geometric and Categorical Structures for Computation and Deduction
• TERMGRAPH	International Workshop on Computing with Terms and Graphs

This volume of FSCD 2020 is published in the LIPIcs series under a Creative Commons license: online access is free to all papers and authors retain rights over their contributions. We thank the Leibniz Center for Informatics at Schloss Dagstuhl, in particular Michael Wagner and Michael Didas for their prompt replies to any questions regarding the production of these proceedings.

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

On behalf of the Program Committee, I thank the many authors of submitted papers for considering FSCD as a venue for their work and all of the speakers for adapting their presentations to a virtual environment. The Program Committee and the external reviewers deserve thanks for their careful and detailed reviews of the submitted papers (the members of the Program Committee and the list of external reviewers can be found on the following pages). The EasyChair conference management system has been a useful tool in all phases of the work of the Program Committee.

The associated workshops made a big contribution to the lively scientific atmosphere of this virtual meeting. I thank the workshop organizers and local Workshop Chair, Giulio Manzonetto, for their efforts and enthusiasm in making sure that workshops continued to be an important element of FSCD. Stefano Guerrini, the Conference Chair, deserves appreciation for rearranging the overall organization of the conference and for the smooth functioning of the virtual meeting. Sandra Alves, as Publicity Chair, made a significant contribution in advertising the conference. The steering committee, led by Delia Kesner, provided excellent guidance in setting up this meeting and in ensuring that FSCD will have a bright and enduring future.

FSCD 2020 was held in-cooperation with ACM SIGLOG and ACM SIGPLAN. It was supported by Université Sorbonne Paris Nord, LIPN (Laboratoire d'Informatique de Paris Nord), IRIF (Institut de Recherche en Informatique Fondamentale), CNRS (Centre National de la Recherche Scientifique), INRIA (Institut National de Recherche en Informatique et en Automatique), Tezos, and Amazon. Finally, I thank all of the participants of the virtual conference for contributing to the success of the event in spite of the unusual circumstances that we faced this year.

Zena M. Ariola
Program Chair of FSCD 2020

Steering Committee

Sandra Alves	University of Porto
Mauricio Ayala-Rincón	Brasilia University
Carsten Fuhs	Birkbeck, University of London
Herman Geuvers	Radboud University Nijmegen
Delia Kesner (Chair)	University Paris-Diderot
Hélène Kirchner	INRIA Paris
Cynthia Kop	Radboud University Nijmegen
Damiano Mazza	University Paris 13
Dale Miller	INRIA Saclay
Luke Ong	University of Oxford
Jakob Rehof	Technical University Dortmund
Sam Staton	University of Oxford
Jamie Vicary	University of Oxford

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Program Committee

M. Alpuente	Technical University of Valencia	Spain
S. Alves	University of Porto	Portugal
Z. M. Ariola (Program Chair)	University of Oregon	USA
A. Bauer	University of Ljubljana	Slovenia
M. P. Bonacina	Università degli studi di Verona	Italy
P.-L. Curien	CNRS - Univ. of Paris Diderot	France
P. Dybjer	Chalmers Univ. of Technology	Sweden
U. De'Liguoro	University of Torino	Italy
M. Fernandez	King's College London	UK
M. Gaboardi	Boston University	USA
D. Ghica	University of Birmingham	UK
S. Ghilezan	University of Novi Sad	Serbia
J. Giesl	RWTH Aachen University	Germany
S. Guerrini (Conference Chair)	University of Paris 13	France
R. Harper	Carnegie Mellon University	USA
M. Hasegawa	Kyoto University	Japan
N. Hirokawa	Japan Advanced Institute of Science and Technology	Japan
P. Johann	Appalachian State University	USA
O. Kammar	University of Edinburgh	UK
D. Kesner	University of Paris Diderot	France
C. Kop	Radboud University	Netherlands
O. Laurent	ENS Lyon	France
D. Licata	Wesleyan University	USA
A. Middeldorp	University of Innsbruck	Austria
J. Mitchell	Stanford University	USA
K. Nakata	SAP Postdam	Germany
M. Pagani	University of Paris Diderot	France
E. Pimentel	Federal University of Rio Grande do Norte	Brasil
F. van Raamsdonk	Vrije University Amsterdam	Netherlands
G. Rosu	University of Illinois	USA
A. Sabry	Indiana University	USA
A. Stump	University of Iowa	USA
P. Urzyczyn	University of Warsaw	Poland
T. Uustalu	Reykjavik University	Icelan
S. Zdancewic	University of Pennsylvania	USA

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

External Reviewers

Takahito Aoto
Kazuyuki Asada
Mauricio Ayala-Rincón
Davide Barbarossa
Pablo Barenbaum
Valentin Blot
Flavien Breuvart
Florian Bruse
Domenico Cantone
Xiaohong Chen
Jacek Chrząszcz
Lorenzo Clemente
Andrea Corradini
Karl Crary
Daniel de Carvalho
Larry Diehl
Daniel Dougherty
Jérémy Dubut
Thomas Ehrhard
Santiago Escobar
José Espírito Santo
Nathanaël Fijalkow
Mário Florido
Jonas Frey
Nicola Gambino
Jacques Garrigue
Thomas Genet
Giulio Guerrieri
Naohiko Hoshino
Jonathan Julian Huerta Y Munive
Pascual Julian-Iranzo
Benjamin Lucien Kaminski
Ambrus Kaposi
Tatsuji Kawai
Nicolai Kraus
Jan Křetínský
James Laird
Tomer Libal
Alexander Lochmann
Salvador Lucas
Hendrik Maarand
Giulio Manzonetto
Andrew M. Marshall
Ralph Matthes
Dylan McDermott
Julius Michaelis
Etienne Miquey
Andrew Miranti
Fabian Mitterwallner
Barbara Morawska
Koko Muroya
Naoki Nishida
Ulf Norell
Carlos Olarte
Luca Padovani
Miguel Pagano
Adrian Palacios
Luca Paolini
Paweł Parys
Luc Pellissier
Lucas Pena
Detlef Plump
Damien Pous
Tim Quatmann
Steven Ramsay
Yann Regis-Gianas
Laurent Regnier
Jakob Rehof
Christian Retoré
Mitchell Riley
Nishant Rodrigues
Jurriaan Rot
Clemente Rubio-Manzano
Julia Sapinã
Manasvi Saxena
Alceste Scalas
Kiraku Shintani
Ana Sokolova
Sam Staton
Jonathan Sterling
Thomas Streicher
Vasco T. Vasconcelos
René Thiemann
Riccardo Treglia
Jan Tušil
Taichi Uemura
Deivid Vale
Benno van den Berg
Vincent van Oostrom
Lionel Vaux Auclair
Niccolò Veltri
Daniel Ventura
German Vidal
Laurent Vigneron
Janis Voigtländer
Uwe Waldmann
Sarah Winkler
Yannick Zakowski
Hans Zantema

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

List of Authors

Mario Alvarez-Picallo (32)
University of Oxford, UK

Takahito Aoto (11)
Niigata University, Niigata, Japan

Kazuyuki Asada (21, 22)
Tohoku University, Sendai, Japan

Alexander Bentkamp (5)
Vrije Universiteit Amsterdam, The Netherlands

Dariusz Biernacki (7, 18)
University of Wrocław, Poland

Frédéric Blanqui (13, 35)
Université Paris-Saclay, ENS Paris-Saclay, CNRS, Inria, France; Laboratoire Spécification et Vérification, Cachan, France

Rose Bohrer (14)
Carnegie Mellon University, Pittsburgh, PA, USA

Guillaume Boisseau (17)
University of Oxford, UK

Paul Brunet (8)
University College London, UK; paul.brunet-zamansky.fr

David M. Cerna (26)
Johannes Kepler Univerisity Linz, Austria

James Cheney (28)
Laboratory for Foundations of Computer Science, University of Edinburgh, UK

Cyril Cohen (34)
Université Côte d'Azur, Inria, Sophia Antipolis, France

Ugo Dal Lago (1)
Dipartimento di Informatica, Università di Bologna, Italy

Ankush Das (33)
Carnegie Mellon University, Pittsburgh, PA, USA

Henry DeYoung (29)
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Tim Lukas Diezel (24)
FAU Erlangen-Nürnberg, Germany

Andrej Dudenhefner (9)
Saarland University, Saarbrücken, Germany

Besik Dundua (30)
FBT, International Black Sea University, Tbilisi, Georgia; VIAM, Ivane Javakhishvili Tbilisi State University, Georgia

Rick Erkens (20)
Eindhoven University of Technology, The Netherlands

Claudia Faggian (1)
Université de Paris, IRIF, CNRS, France

Zeinab Galal (16)
Université de Paris, IRIF, CNRS, Paris, France

Guillaume Genestier (31)
Université Paris-Saclay, ENS Paris-Saclay, Inria, CNRS, LSV, France; MINES ParisTech, PSL University, France

Sergey Goncharov (24)
FAU Erlangen-Nürnberg, Germany

André Hirschowitz (12)
Université Côte d'Azur, CNRS, Nice, France

Tom Hirschowitz (12)
Univ. Grenoble Alpes, Univ. Savoie Mont Blanc, CNRS, LAMA, 73000 Chambéry, France

Gabriel Hondet (35)
Université Paris-Saclay, ENS Paris-Saclay, CNRS, Inria, Laboratoire Spécification et Vérification, Gif-sur-Yvette, France

Mathias Hülsbusch (10)
Universität Duisburg-Essen, Germany

Andrej Ivašković (15)
Department of Computer Science and Technology, University of Cambridge, UK

Ambrus Kaposi (23)
Eötvös Loránd University, Budapest, Hungary

Naoki Kobayashi (19, 21, 22)
The University of Tokyo, Japan

Cynthia Kop (36)
Radboud University, The Netherlands

Temur Kutsia (26, 30)
Johannes Kepler Univerisity Linz, Austria

Barbara König (10)
Universität Duisburg-Essen, Germany

Sebastian Küpper (10)
FernUniversität in Hagen, Germany

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Ambroise Lafont (12)
University of New South Wales, Sydney, Australia

Maurice Laveaux (20)
Eindhoven University of Technology, The Netherlands

Sergueï Lenglet (7)
Université de Lorraine, Nancy, France

Yu-Yang Lin (27)
Queen Mary University of London, UK

Luigi Liquori (37)
Université Côte d'Azur, Nice, France; Inria Sophia Antipolis - Méditerranée, France

Mircea Marin (30)
West University of Timişoara, Romania

Paul-André Melliès (6)
CNRS, Institut de Recherche en Informatique Fondamentale (IRIF), Université de Paris, France

Yo Mitani (19)
The University of Tokyo, Japan

Alan Mycroft (15)
Department of Computer Science and Technology, University of Cambridge, UK

Yoshiki Nakamura (21)
Tokyo Institute of Technology, Japan

Visa Nummelin (5)
Vrije Universiteit Amsterdam, The Netherlands

C.-H. Luke Ong (32)
University of Oxford, UK

Dominic Orchard (15)
School of Computing, University of Kent, UK

Cleopatra Pau (30)
Johannes Kepler University, Research Institute for Symbolic Computation, Linz, Austria

Frank Pfenning (29, 33)
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Brigitte Pientka (2)
McGill University, Montreal, QC, Canada

Andrew M. Pitts (3)
Department of Computer Science & Technology, University of Cambridge, UK

André Platzer (14)
Carnegie Mellon University, Pittsburgh, PA, USA; Technische Universität München, Germany

Piotr Polesiuk (7)
University of Wrocław, Poland

Klaas Pruiksma (29)
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

David Pym (8)
University College London, UK; www.cantab.net/users/david.pym/

Mateusz Pyzik (18)
Institute of Computer Science, University Wrocław, Poland

Wilmer Ricciotti (28)
Laboratory for Foundations of Computer Science, University of Edinburgh, UK

Nicolas Rolland (6)
Institut de Recherche en Informatique Fondamentale (IRIF), Université de Paris, France

Simona Ronchi Della Rocca (1)
Dipartimento di Informatica, Università di Torino, Italy

Kazuhiko Sakaguchi (34)
University of Tsukuba, Japan

Jonas Schöpf (4)
University of Innsbruck, Austria

Filip Sieczkowski (18)
Institute of Computer Science, University Wrocław, Poland

Ryoma Sin'ya (21)
Akita University, Japan

Christian Sternagel (4)
DVT, Innsbruck, Austria

Lara Stoltenow (10)
Universität Duisburg-Essen, Germany

Claude Stolze (37)
IRIF, Université de Paris, France

Enrico Tassi (34)
Université Côte d'Azur, Inria, Sophia Antipolis, France

René Thiemann (4)
University of Innsbruck, Austria

Takeshi Tsukada (19, 21)
The University of Tokyo, Japan

Nikos Tzevelekos (27)
Queen Mary University of London, UK

Jakob von Raumer (23)
University of Nottingham, UK

Petar Vukmirović (5)
Vrije Universiteit Amsterdam, The Netherlands

Chuangjie Xu (25)
Ludwig-Maximilians-Universität München, Germany

Akihisa Yamada (4)
National Institute of Advanced Industrial Science and Technology, Tokyo, Japan

Masaomi Yamaguchi (11)
Tohoku University, Sendai, Japan

Solvability in a Probabilistic Setting

Simona Ronchi Della Rocca
Dipartimento di Informatica, Università di Torino, Italy

Ugo Dal Lago
Dipartimento di Informatica, Università di Bologna, Italy

Claudia Faggian
Université de Paris, IRIF, CNRS, France

Abstract

The notion of solvability, crucial in the λ-calculus, is conservatively extended to a probabilistic setting, and a complete characterization of it is given. The employed technical tool is a type assignment system, based on non-idempotent intersection types, whose typable terms turn out to be precisely the terms which are solvable with nonnull probability. We also supply an operational characterization of solvable terms, through the notion of head normal form, and a denotational model of $\Lambda_\oplus$, itself induced by the type system, which equates all the unsolvable terms.

2012 ACM Subject Classification Theory of computation → Lambda calculus; Theory of computation → Models of computation

Keywords and phrases Probabilistic Computation, Lambda Calculus, Solvability, Intersection Types

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.1

Category Invited Talk

1 Introduction

In *probabilistic computation*, the current state of the underlying program or machine can evolve in different ways depending on the outcome of probabilistic choices, this way turning an essentially deterministic process into a stochastic one. This computing paradigm has proved useful, in particular, in the area of cryptography [20] or in so-called randomized algorithmics [25]. From a theoretical point of view, all evolutions of a computation possibly contribute to the final result, according to the laws of probability. As a consequence, the result of a computation, if formalized using rewriting, is not *a* normal form with respect to some set of reduction rules, but a *probabilistic distribution* on all the possible outcomes. If the languages one has in mind are higher-order probabilistic languages, a natural model to consider is the λ-calculus, of course enriched with one or more probabilistic constructs.

The simplest approach, followed in [11, 8, 13] consists in endowing the λ-calculus with an operator $\oplus$ modeling fair coin flipping. This suffices to reach universality [8]: the mere presence of binary fair probabilistic choice allows to get all *computable* probability distributions on the natural numbers. The resulting calculus, called $\Lambda_\oplus$, is however well-known to be non-confluent, as recalled in Example 3.2 below. In the literature, such a problem has been handled by fixing deterministic reduction strategies. In [15], a foundational investigation of all this has been initiated following a principle stated by Plotkin [28], where a clear distinction is made between *calculi* and *programming languages*: the former consist of reduction rules (and are thus independent of any reduction strategy), enjoy confluence and standardization, while the latter are implementations of calculi, obtained by fixing a deterministic standard strategy. The aforementioned reference [28] is the first considering $\Lambda_\oplus$ as a calculus in the former sense, endowing it with the β-rule in its full generality, and

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 1; pp. 1:1–1:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

a rule dealing with the probabilistic operator $\oplus$. The main results were that, under mild conditions on the probabilistic rule, confluence and standardization hold. This is done both in presence of call-by-name and of call-by-value evaluation.

In this paper, we continue this foundational investigation, from a semantic perspective. We restrict ourselves to consider the call-by-name version of $\Lambda_\oplus$, and we study in it the notion of *solvability*. Solvability is a central notion in λ-calculus theory: *solvable* programs are those which are meaningful, i.e. those that can produce any desired result when applied to a suitable sequence of arguments. More formally, a closed term M of the usual λ-calculus is solvable if there is a sequence of terms $\vec{P}$ such that $M\vec{P}$ reduces to the identity. The importance of this notion is witnessed by the fact that it is sound to equate all unsolvable terms in any denotational semantics. We define solvability in $\Lambda_\oplus$, in a conservative way with respect to λ-calculus, as follows. A closed term M of $\Lambda_\oplus$ is said to be *p-solvable*, where $p \in \mathbb{R}_{[0,1]}$ if p is the least upper bound on the probabilities of observing the identity in the distributions obtained reducing $M\vec{P}$ for any sequence of terms $\vec{P}$. In order to study solvability, we use a type assignment system, based on non-idempotent intersection types, where types are multisets (so intersections) of simple types, weighted by probabilities. The result we obtain is a complete characterization of probabilistic solvability: an operational characterization, through the notion of head normal form, a logical one, through typing, and a denotational one. In fact, the type assignment system we define supplies a model for $\Lambda_\oplus$, giving not trivial denotation to all and only the p-solvable terms, for a strictly positive real.

Related Work. The idea of endowing the λ-calculus with a form of probabilistic choice is not at all new (see, e.g., [31, 17, 27, 29, 11, 13, 8]). Most of the introduced idioms, however, come with a fixed reduction strategy, i.e. they are indeed languages, not calculi, according to Plotkin's distinction. To the authors' knowledge, the only proposals of a probabilistic λ-calculus in which reduction is studied independently on a *specific strategy* are the call-by-name calculus introduced in [22], which stems from the line of work of differential [14] and algebraic [32] λ-calculi, and the already mentioned work by the first and third authors [15].

The study of semantical properties of probabilistic λ-calculi has itself a long tradition, starting from the pioneering contributions which introduced and studied the so-called probabilistic powerdomain [31, 17], down to some deep observations about the technical problems one inevitably encounters along this route [18], until, e.g., a very recent contribution about how probabilistic higher-order computation can be reconciled with domain-theoretic semantics based on continuous functions, through call-by-push-value [16].

An alternative way of giving a denotational semantics to probabilistic λ-calculi based on coherent spaces has also been investigated [9], and the obtained model has been proved fully abstract for a probabilistic variation on Plotkin's PCF [12]. A model (itself based on coherent spaces) for a calculus very similar in spirit to $\Lambda_\oplus$ has been given [13], and proved fully abstract [24]. A full abstraction result has also been proved [5] for a model which is based on the weighted relational semantics [21].

Observational equivalence for a probabilistic λ-calculus has been further studied by Leventis [23] who proved it to coincide with the equivalence induced by Nakajima trees, i.e. Böhm trees quotiented by infinitary extensionality. A probabilistic variation on Abramsky's applicative bisimilarity has been proved sound for contextual equivalence in an untyped λ-calculus with weak-head evaluation [7], and fully abstract in presence of sequencing [19] or head evaluation.

An intersection type assignment system for $\Lambda_{\oplus}$ has been designed in [4], but with an aim different than the one we have in the present paper: the authors show how *weak*-head termination can be characterized by typability in idempotent intersection types. Type disciplines of other kinds, sized types [6] and linear dependent types [1] in particular, have been shown to be sound for termination of probabilistic higher-order programs.

Outline. In Section 2, we give basic notions about probability theory. In particular, we define the central notions of distribution and multidistribution. In Section 3, we introduce the calculus $\Lambda_{\oplus}$, a λ-calculus endowed with a probabilistic choice operator, indicated as $\oplus$. The calculus' operational semantics is given in terms of multidistributions, following [2, 15]. Then, we define the semantic notion of p-solvability, and the notion of having head normal form with probability p. In Section 4, we give a type assignment system, based on non-idempotent intersection types, where types are multidistributions of simple types, and we prove that the system enjoys the good properties of subject reduction and expansion. In Section 5 we give, through the type assignment system, a threefold characterization of solvability: operational, logical and denotational. In fact, the type assignment system induces a model for $\Lambda_{\oplus}$ in which terms are interpreted by *sets* of typings. Section 6 contains some concluding remarks and hints for future work. Some technical proofs are in the Appendix.

2 Preliminaries

A *discrete probability space* is given by a pair (Ω,μ), where Ω is a *countable* set, and μ is a *discrete probability distribution* on Ω, *i.e.* is a function from Ω to $[0,1] \subset \mathbb{R}$ such that $\|\mu\| := \sum_{\omega\in\Omega}\mu(\omega) = 1$. In this case, a probability measure is assigned to any subset $\mathcal{A} \subseteq \Omega$ as $\mu(\mathcal{A}) = \sum_{\omega\in\mathcal{A}}\mu(\omega)$. Given a countable set Ω, a function $\mu : \Omega \to [0,1]$ is a probability *subdistribution* if $\|\mu\| \leq 1$. We write $\mathtt{DST}(\Omega)$ for the set of subdistributions on Ω. Subdistributions allow us to deal with partial results and non-successful computations. Slightly abusively, we often use the term distribution also when referring to subdistributions.

Let (Ω,μ) be as above. Any *function* $F : \Omega \to \Delta$, where Δ is another countable set, *induces a probability distribution* μ^F on Δ by composition: $\mu^F(d \in \Delta) := \mu(F^{-1}(d))$ *i.e.* $\mu\{\omega\in\Omega : F(\omega) = d\}$. The *support* of μ is $\mathit{Supp}(\mu) = \{\omega : \mu(\omega) > 0\}$. We represent a distribution by explicitly indicating the support, and (as superscript) the probability μ assigns to each element. We write $\mu = \{a_1^{p_1},\ldots,a_n^{p_n}\}$ (where the a_is are pairwise distinct) if $\mu(a_i) = p_i$ for every $1 \leq i \leq n$ and $\mu(b) = 0$ for every $b \in \{a_1,\ldots,a_n\}$.

3 The Calculus

Terms and Contexts. *Terms* of $\Lambda_{\oplus}$ are generated by the grammar

$$M,N,P,Q ::= x \mid \lambda x.M \mid MM \mid M \oplus M \qquad\qquad (\textbf{Terms})$$

where x ranges over a countable set of *variables* (indicated as $x,y,\ldots$). As usual, $\lambda xy.PQR$ abbreviates $\lambda x.(\lambda y.(PQ)R)$, $\vec{x}$ and $\vec{M}$ denote respectively a sequence of variables and a sequence of terms, and $|\vec{x}|$ and $|\vec{M}|$ denote their lenghts. Free variables are defined as usual. $M[N/x]$ denotes the term obtained by the capture-avoiding substitution of N for each free occurrence of x in M. Terms we use frequently in our examples are $\mathsf{I} = \lambda x.x$, $\Delta = \lambda x.xx$, $\mathsf{K} = \lambda xy.x$, $\mathsf{O} = \lambda xy.y$ and $\Omega = (\lambda x.xx)(\lambda x.xx)$.

Contexts and *surface contexts* are generated by the grammars:

$$\mathbf{C} ::= \square \mid M\mathbf{C} \mid \mathbf{C}M \mid \lambda x.\mathbf{C} \mid \mathbf{C}\oplus M \mid M\oplus\mathbf{C} \qquad (\textbf{Contexts})$$

$$\mathbf{S},\mathbf{W},\mathbf{T} ::= \square \mid \lambda x.\mathbf{S} \mid \mathbf{S}M \qquad (\textbf{Surface Contexts})$$

where $\square$ denotes the *hole* of the context. Given a context $\mathbf{C}$, we denote by $\mathbf{C}(M)$ the term obtained from $\mathbf{C}$ by filling the hole with M, allowing the capture of free variables. Similarly for surface contexts. Since the hole will be filled with a redex, surface contexts formalize the fact that the redex (the hole) is neither in argument position nor in the scope of a $\oplus$.

Multidistributions. To syntactically represent the global evolution of a probabilistic system, we rely on the notion of multidistribution [2].

A *multiset* is a (finite) list of elements, modulo reordering, ranged over by $\mathtt{m}$,$\mathtt{n}$. Let $\mathtt{m}$ be a multiset of pairs of the form pM, with $p\in]0,1]$, and $M\in\Lambda_\oplus$. We call $\mathtt{m}=[p_iM_i \mid i\in I]$ (where the index set I ranges over finite subsets of a countable set a *multidistribution on* $\Lambda_\oplus$ if $\sum_{i\in I} p_i \leq 1$ (think of list concatenation). We denote by $\mathtt{MDST}(\Lambda_\oplus)$ the set of all multidistributions. We write the multidistribution $[1M]$ simply as $[M]$. The sum of multidistributions is denoted by $+$. The product $q\cdot\mathtt{m}$ of a scalar q and a multidistribution $\mathtt{m}$ is defined pointwise: $q\cdot[p_1M_1,...,p_nM_n]=[(qp_1)M_1,...,(qp_n)M_n]$.

Intuitively, a multidistribution $\mathtt{m}\in\mathtt{MDST}(\Lambda_\oplus)$ is a syntactical representation of a discrete probability space where to each element of the space are associated a probability and a term of $\Lambda_\oplus$. To the multidistribution $\mathtt{m}=[p_iM_i \mid i\in I]$, we associate a probability distribution $\mu_{\mathtt{m}}\in\mathtt{DST}(\Lambda_\oplus)$ as follows:

$$\mu_{\mathtt{m}}(M)=\sum_{i\in I} q_i \qquad q_i=\begin{cases} p_i & \text{if } M_i=M \\ 0 & \text{otherwise}\end{cases}$$

(Observe that, $\mathtt{m}$ being a multiset, there are in general *more than one* elements p_iM_i where $M_i = M$, or even multiple copies of the same element). As usual (see Section 2), the distribution $\mu_{\mathtt{m}}$ assigns a probability measure to *every* subset of $\Lambda_\oplus$, namely the sum of the probabilities of its elements. That is, given a set of terms $\mathcal{T}\subseteq\Lambda_\oplus$,

$$\mu_{\mathtt{m}}(\mathcal{T})=\sum_{M\in\mathcal{T}}\mu_{\mathtt{m}}(M)$$

▶ **Example 3.1** (Distributions vs. Multidistributions)**.** If $\mathtt{m}=[\frac{1}{2}a,\frac{1}{2}a]$, then $\mu_{\mathtt{m}}=\{a^1\}$. Please observe the *different nature* of distributions and multidistributions: if $\mathtt{n}=[1a]$, then $\mathtt{m}\neq\mathtt{n}$, but $\mu_{\mathtt{m}}=\mu_{\mathtt{n}}$.

Reduction Rules. We first define reduction rules on terms (Fig. 1), and one-step reduction from terms to multidistributions (Fig. 2). We then lift the definition of reduction to a binary relation on $\mathtt{MDST}(\Lambda_\oplus)$.

Observe that in the λ-calculus, a reduction step is given by the closure under context of the reduction rules. However, a reduction from terms to terms is not informative enough in a probabilistic setting, because the likelihood of each reduction step needs to be taken into account. The meaning of $M\oplus N$ is that this term reduces to either M or N, *with equal probability* $\frac{1}{2}$. There are various ways to formalize this fact, and here we follow [2, 15] and use multidistributions.

The *reduction rules* on the terms of $\Lambda_\oplus$ are defined in Fig. 1. The *(one-step) reduction relations* $\to_\beta, \to_\oplus \subseteq \Lambda_\oplus \times \mathtt{MDST}(\Lambda_\oplus)$ are defined in Fig. 2. Observe that the probabilistic rules $\mapsto_{r\oplus,l\oplus}$ are closed only under surface contexts, while the reduction rule $\mapsto_\beta$ is closed under general contexts. We denote by $\to$ the union $\to_\beta \cup \to_\oplus$. We lift the reduction relation $\to \subseteq \Lambda_\oplus \times \mathtt{MDST}(\Lambda_\oplus)$ to a relation $\Rightarrow \subseteq \mathtt{MDST}(\Lambda_\oplus) \times \mathtt{MDST}(\Lambda_\oplus)$, as defined in Fig. 3. Observe that $\Rightarrow$ is a reflexive relation.

$$\begin{array}{cc} \textit{The } \beta\textit{-rule} & \textit{Probabilistic Rules} \\ (\lambda x.M)N \mapsto_\beta M[N/x] & M \oplus N \mapsto_{l\oplus} M \quad M \oplus N \mapsto_{r\oplus} N \end{array}$$

Figure 1 Reduction Rules.

$$\frac{(\lambda x.M)N \mapsto_\beta M[N/x]}{\mathbf{C}((\lambda x.M)N) \to_\beta [\mathbf{C}(M[N/x])]} \qquad \frac{M \oplus N \mapsto_{l\oplus} M \quad M \oplus N \mapsto_{r\oplus} N}{\mathbf{S}(M \oplus N) \to_\oplus [\frac{1}{2}\mathbf{S}(M), \frac{1}{2}\mathbf{S}(N)]}$$

Figure 2 Reduction Steps.

We define in the same way the lifting of any relation $\to_r \subseteq \Lambda_\oplus \times \mathtt{MDST}(\Lambda_\oplus)$ to a binary relation $\Rightarrow_r$ on $\mathtt{MDST}(\Lambda_\oplus)$. In particular, we lift $\to_\beta, \to_\oplus$ to $\Rightarrow_\beta, \Rightarrow_\oplus$. The definition of lifting allows us to apply a reduction step $\to$ to any number of M_i in the multidistribution $\mathtt{m} = [p_i M_i \mid i \in I]$. If no M_i is reduced, then $\mathtt{m} \Rightarrow \mathtt{m}$ (the relation $\Rightarrow$ is reflexive).

$$\frac{}{[M] \Rightarrow [M]} \qquad \frac{M \to \mathtt{m}}{[M] \Rightarrow \mathtt{m}} \qquad \frac{([M_i] \Rightarrow \mathtt{m}_i)_{i \in I}}{[p_i M_i \mid i \in I] \Rightarrow +_{i \in I} p_i \cdot \mathtt{m}_i}$$

Figure 3 Lifting → to ⇒.

Reduction Sequences. A $\Rightarrow$-sequence (or *reduction sequence*) from $\mathtt{m}$ is a sequence $\mathtt{m}_0, \mathtt{m}_1, \mathtt{m}_2, \ldots$ such that $\mathtt{m} = \mathtt{m}_0$ and $\mathtt{m}_n \Rightarrow \mathtt{m}_{n+1}$ for every $n \in \mathbb{N}$. We write $\mathtt{m} \Rightarrow^* \mathtt{n}$ to indicate that there is a *finite* sequence from $\mathtt{m}$ to $\mathtt{n}$, and $\langle \mathtt{m}_n \rangle_{n \in \mathbb{N}}$ for an *infinite sequence.*

Confluence. It has been proved [15] that the reduction $\Rightarrow$ enjoys the confluence property. The restriction of $\Rightarrow_\oplus$ to surface contexts is essential to obtain confluence, as the following example shows.

▶ **Example 3.2.** Let M be the term $\Delta(\mathsf{K} \oplus \mathsf{I})$. $[M] \Rightarrow_\beta [(\mathsf{K} \oplus \mathsf{I})(\mathsf{K} \oplus \mathsf{I})] \Rightarrow^* [\frac{1}{4}\mathsf{KK}, \frac{1}{4}\mathsf{KI}, \frac{1}{4}\mathsf{IK}, \frac{1}{4}\mathsf{II}] \Rightarrow^* [\frac{1}{4}\lambda x.\mathsf{K}, \frac{1}{4}\lambda x.\mathsf{I}, \frac{1}{4}\mathsf{K}, \frac{1}{4}\mathsf{I}]$, which is a multidistribution on normal forms. But, if we would allow $\Rightarrow_\oplus$ also in the argument position, the result would be: $[M] \Rightarrow_\oplus [\frac{1}{2}\Delta\mathsf{K}, \frac{1}{2}\Delta\mathsf{I}] \Rightarrow^* [\frac{1}{2}\mathsf{KK}, \frac{1}{2}\mathsf{II}] \Rightarrow_\oplus [\frac{1}{2}\lambda x.\mathsf{K}, \frac{1}{2}\mathsf{I}]$, which is a different multidistribution, again on normal forms!

Head Normal Forms. The notion of head normal form can be extended to $\Lambda_\oplus$. Head normal forms (shortly *hnfs*) are the normal forms of *surface reduction* $\overset{s}{\to}$, i.e., the closure of both β and $\oplus$ reduction rules under surface contexts $\mathbf{S}$. Let us write $\mathcal{H}$ for the set of *head normal forms*, which can be seen as being defined by the following grammar:

$$\mathcal{H} ::= \lambda x.\mathcal{H} \mid \mathcal{K}; \qquad \mathcal{K} ::= x \mid \mathcal{K}M.$$

It is easy to check that any term of $\Lambda_\oplus$ can be written the following form:

$$\lambda x_1 ... x_n.\zeta M_1 ... M_m,$$

where $m,n \geq 0$ and ζ (the *head*) is either a variable or a redex. So, as in the λ-calculus, the head normal forms are the terms having a variable in their head position.

In order to generalize the notion of *having a head normal form* to $\Lambda_\oplus$ we need to take into account probabilities. Recall that $\mathcal{H}$ is the set of head normal forms, and therefore $\mu_{\mathtt{m}}(\mathcal{H})$ is the probability assigned by $\mu_{\mathtt{m}}$ to the event "a term is in head normal form". Let then p be any strictly positive real number. Then:

- A term M *has head normal form with probability at least* p (notation $\geq$*p-hnf*) if there is $\mathtt{m}$ such that $[M] \Rightarrow^* \mathtt{m}$ and $\mu_{\mathtt{m}}(\mathcal{H}) \geq p$.
- A term M *has head normal form with probability* p (notation *p-hnf*) if $p = sup\{q \,|\, [M] \Rightarrow^* \mathtt{m}$ and $\mu_{\mathtt{m}}(\mathcal{H}) = q\}$.
- A term M *has not head normal form* if for every $\mathtt{m}$ it holds that $[M] \Rightarrow^* \mathtt{m}$ implies $\mu_{\mathtt{m}}(\mathcal{H}) = 0$.

Note that even if a term has head normal form with probability 1, that degree of certitude is not necessarily reached in any finite number of steps, as the following Example (point 2) shows:

▶ **Example 3.3.**

1. $M = \lambda yz.(y\mathsf{I} \oplus y)\Omega$ has $\geq\frac{1}{2}$-*hnf* and $\geq$1-*hnf* so it has 1-*hnf*. In fact $[M] \Rightarrow \mathtt{m} = [\frac{1}{2}\lambda yz.y\mathsf{I}\Omega, \frac{1}{2}\lambda yz.y\Omega]$, and both components of $\mathtt{m}$ are in *hnf*.
2. Let $N = \lambda x.xx \oplus \mathsf{I}$, and let $M = NN$. It is easy to check that $[NN] \Rightarrow^* [\frac{1}{2}NN, \frac{1}{2}\mathsf{I}]$, which, for every n, reduces to $\mathtt{m}$ such that $\mu_{\mathtt{m}}(\mathcal{H}) = \frac{1}{2} + \frac{1}{4} + .. + \frac{1}{2^n}$. So M has $\geq \sum_1^n \frac{1}{2^n}$-*hnf*, for all $n \geq 1$, and it thus has 1-*hnf*.
3. $\mathsf{I} \oplus \Omega$ has $\frac{1}{2}$-*hnf*.

3.1 Solvability

The notion of solvability is a central semantic notion, capturing the property of a term being meaningful (or a program being meaningful, if we consider closed terms). In λ-calculus, the semantic notion of solvability has its operational counterpart in that of head normal form, moreover it can be characterized by suitable intersection type assignment systems. We will show that similar properties hold for $\Lambda_\oplus$.

Let us first recall this notion for λ-calculus [3]. A λ-term M is solvable if there is a surface context such that $\mathbf{S}(M)$ β-reduces to the identity I (obviously considering surface contexts restricted to λ-calculus[1]). Closed solvable terms represent meaningful programs: if M is closed and solvable, then M can produce any desired result when applied to a suitable sequence of arguments. The importance of this notion is certified by the fact that it is sound to equate all unsolvable terms in any denotational semantics.

Extending this notion to $\Lambda_\oplus$ is indeed possible by way of the following definition, where p is any strictly positive real:

- A term M is *solvable with probability at least* p (notation $\geq$*p*-solvable) if there is a surface context $\mathbf{S}$ such that $[\mathbf{S}(M)] \Rightarrow^* \mathtt{m}$, and $\mu_{\mathtt{m}}(\mathsf{I}) \geq p$.

[1] To be precise, the definition of solvability for λ-calculus uses the notion of head context, which is a restriction of that of surface context. But since the two induced β-reductions have the same normal forms, using one or the other in the definition is equivalent. Here, the use of surface contexts is motivated by the fact that the surface reduction is standard, while head reduction is not (see [15]).

- A term M is *p-solvable* if $p = sup\{q \mid M \text{ is } \geq q\text{-solvable}\}$.
- A term is unsolvable if for every $\mathbf{S}$ it holds that $[\mathbf{S}(M)] \Rightarrow^* \mathtt{m}$ implies $\mu_{\mathtt{m}}(\mathsf{I}) = 0$.

This definition, when restricted to the syntax of λ-calculus, coincides with the standard one. Note that, while proving a term $\geq p$-solvable requires to exhibit just *one* context, proving that a term is p-solvable may require to exhibit *an infinite* number of contexts. Point 3 of the following example is an instance of this fact.

▶ **Example 3.4.**

1. $M = \lambda yz.(y\mathsf{I} \oplus y)\Omega$ is 1-solvable and $\geq\frac{1}{2}$-solvable and $\geq$1-solvable. A context playing the job is $\square(\lambda xt.\mathsf{I})(\lambda xt.\mathsf{I})\mathsf{I}$. In fact, $[M(\lambda xt.\mathsf{I})(\lambda xt.\mathsf{I})\mathsf{I}] \Rightarrow^*_\beta [((\lambda xt.\mathsf{I})\mathsf{I} \oplus (\lambda xt.\mathsf{I}))\Omega\mathsf{I}] \Rightarrow_\oplus [\frac{1}{2}(\lambda xt.\mathsf{I})\mathsf{I}\Omega\mathsf{I}, \frac{1}{2}(\lambda xt.\mathsf{I})\Omega\mathsf{I}] \Rightarrow^*_\beta [\frac{1}{2}\mathsf{II}, \frac{1}{2}(\lambda t.\mathsf{I})\mathsf{I}] \Rightarrow_\beta [\frac{1}{2}\mathsf{I}, \frac{1}{2}\mathsf{I}]$.
2. Consider the term NN, defined in Example 3.3.2. Clearly NN is $\geq\sum_1^n \frac{1}{2^n}$-solvable, for every $n > 0$. To prove that M is 1-solvable, the context $\square$ suffices.
3. Let Y be a fixed-point operator, whose behavior is $[YM] \Rightarrow^* [M(YM)]$. Then $[Y(\mathsf{K} \oplus \mathsf{O})] \Rightarrow^* [\frac{1}{2}\lambda x.Y(\mathsf{K} \oplus \mathsf{O}), \frac{1}{2}\mathsf{I}] \Rightarrow^* [\frac{1}{2}\mathsf{I}, \frac{1}{4}\lambda x.\mathsf{I}, \frac{1}{8}\lambda x_1 x_2.\mathsf{I}, \ldots, \frac{1}{2^n}\lambda x_1..x_{n+1}.\mathsf{I}]$, $(n \geq 0)$. Then the context $\mathbf{S}_n = \square\underbrace{\mathsf{I}..\mathsf{I}}_{n+1}$ is a witness that M is $\geq\sum_1^n \frac{1}{2^n}$-solvable, for every $n \geq 0$. Taking the supremum, this term is 1-solvable.

Three Characterizations of Solvability. In the λ-calculus, solvability can be characterized [3, 30] in three different ways:

- *operationally*, through the notion of head normal form;
- *logically*, through suitable type assignment systems, based on intersection types;
- *denotationally*, though some denotational models.

To be more precise, the operational characterization says that a term is solvable if and only if it has head normal form, the logical characterization says that there are type assignment systems assigning types to all and only the solvable terms, and the denotational one says that there are λ-models which are sensible, i.e., which assign a significant denotation to all and only the solvable terms. The aim of this paper is to show that similar characterizations hold also for $\Lambda_\oplus$, taking into account the differences between the two calculi. For proving all three characterizations, one tool is sufficient, namely an intersection type assignment system.

4 A Type Assignment System for $\Lambda_\oplus$

In this section we will present a type assignment system, based on non idempotent intersection types, which is the technical tool we will use to characterize the solvability property of $\Lambda_\oplus$.

Types. Types are defined by the following grammars:

$$\mathtt{A}, \mathtt{B} ::= \alpha \mid \mathcal{A} \to \mathtt{A} \qquad \textbf{(Simple Types)}$$

$$\mathsf{a}, \mathsf{b}, \mathsf{c} ::= \langle p_1 \mathtt{A}_1, \ldots, p_n \mathtt{A}_n \rangle \qquad \textbf{(Types)}$$

$$\mathcal{A}, \mathcal{B} ::= [\mathsf{a}_1, \ldots, \mathsf{a}_n] \qquad \textbf{(Context Types)}$$

where $n \geq 0$, α ranges over a countable set of constants, types are multidistributions on simple types, and context types are multisets of types. Note that the definition of types as multidistributions means that, syntactically, $\mathsf{a} = \langle p_1 \mathtt{A}_1, \ldots, p_n \mathtt{A}_n \rangle$ implies $p_i > 0$ for every $1 \leq i \leq n$ and $\Sigma_{1 \leq i \leq n} p_i \leq 1$. If $\mathsf{a} = \langle p_i A_i \mid i \in I \rangle$ then its *norm* is $\|\mathsf{a}\| = \sum_i p_i$. As usual, in simple types, the type constructor → associates to the right.

Type Contexts. Type contexts, ranged over by Γ,Δ,Φ,Ψ are partial functions from variables to context types, with finite domain. $\Gamma \uplus \Delta$ denotes the function such that $(\Gamma \uplus \Delta)(x) = \Gamma(x) + \Delta(x)$. We denote by $\leq$ the set-theoretical order between partial functions.

The Type Assignment System $\mathcal{S}$. The type assignment $\mathcal{S}$ is given in Fig. 4; it proves judgements of the shape $\Gamma \vdash M : \mathsf{a}$, or $\Gamma \vdash M : \mathcal{A}$, where Γ is a type context, M is a term, a is a type and $\mathcal{A}$ is a context type.

▶ **Notation 4.1.** *$p\mathsf{a}$ denotes $\langle pq\mathtt{A} \,|\, q\mathtt{A} \in \mathsf{a}\rangle$; recall that $+$ denotes the multiset union, so $\mathsf{a}+\mathsf{b}$ denotes the concatenation of a and b. The type $\langle 1\mathtt{A}\rangle$ is abbreviated by $\langle \mathtt{A}\rangle$. $\Gamma \vdash M : \mathsf{a}$ denotes the existence of a derivation proving this judgment, while $\Pi \triangleright \Gamma \vdash M : \mathsf{a}$ denotes a particular type derivation proving the judgment $\Gamma \vdash M : \mathsf{a}$. If $\Pi \triangleright \Gamma \vdash M : \mathsf{a}$, then M and a are respectively the subject and the object of Π. $\vdash M : \mathsf{a}$ abbreviates $\emptyset \vdash M : \mathsf{a}$, where $\emptyset$ is the empty function. If $\Pi \triangleright \Gamma \vdash M : \mathsf{a}$ we say that M is typed in Π with probability $\|\mathsf{a}\|$.*

$$\frac{\mathsf{a} \in \mathcal{A}}{\Gamma, x : \mathcal{A} \vdash x : \mathsf{a}}\ \mathit{var}$$

$$\frac{\Gamma, x : \mathcal{A} \vdash M : \langle p_i \mathtt{A}_i \,|\, i \in I\rangle}{\Gamma \vdash \lambda x.M : \langle p_i(\mathcal{A} \to \mathtt{A}_i) \,|\, i \in I\rangle}\ {\to} I \qquad \frac{\Gamma \vdash M : \langle p_i(\mathcal{A}_i \to \mathtt{B}_i) \,|\, i \in I\rangle \quad (\Delta_i \vdash N : \mathcal{A}_i)_{i \in I}}{\Gamma \uplus_{i \in I} \Delta_i \vdash MN : \langle p_i \mathtt{B}_i \,|\, i \in I\rangle}\ {\to} E$$

$$\frac{(\Gamma_i \vdash M : \mathsf{a}_i)_{i \in I}}{\uplus_i \Gamma_i \vdash M : [\mathsf{a}_i \,|\, i \in I]}\ !$$

$$\frac{\Gamma \vdash M : \mathsf{a} \quad \Gamma \vdash N : \mathsf{b}}{\Gamma \vdash M \oplus N : \frac{1}{2}\mathsf{a} + \frac{1}{2}\mathsf{b}}\ \oplus \qquad \frac{\Gamma \vdash M : \mathsf{a}}{\Gamma \vdash M \oplus N : \frac{1}{2}\mathsf{a}}\ \oplus_l \qquad \frac{\Gamma \vdash N : \mathsf{a}}{\Gamma \vdash M \oplus N : \frac{1}{2}\mathsf{a}}\ \oplus_r$$

Figure 4 The Type Assignment System $\mathcal{S}$.

The *size* of a derivation Π, denoted by $|\Pi|$, is defined as follows. Note that the size of Π is not the number of its rule applications (the dimension of the derivation tree) because of the cases of rules (!) and ($\oplus$).

- If Π is an application of the rule (var), then $|\Pi| = 1$;
- If Π ends with an application of rule ($\to I$), with premise Φ, then $|\Pi| = |\Phi| + 1$;
- If Π ends with an application of rule ($\to E$), with premise Φ and $(\Psi_i)_{i \in I}$, then $|\Pi| = |\Phi| + \sum_{i \in I} |\Psi_i| + 1$;
- If Π ends with an application of rule (!), with premises $(\Phi_i)_{i \in I}$, then $|\Pi| = \sum_{i \in I} |\Phi_i|$;
- If Π ends with an application of rule ($\oplus$), with premises Φ,Ψ, then $|\Pi| = \mathit{max}\{|\Phi|,|\Psi|\} + 1$;
- If Π ends with an application of rule ($\oplus_r$) (resp. ($\oplus_l$)), with premise Φ, then $|\Pi| = |\Phi| + 1$;

The size of a derivation is a key notion here, since one of the characterizations, namely the proof of (1⇒2) in Theorem 5.1 is by induction on it. A benefit of using non idempotent intersection consists in the fact that it is possible to define a measure of derivations that decreases while reducing the subject. In case of λ-calculus, the size corresponds to the dimension of the derivation tree, i.e., the number of rule applications in it. Here the additive behavior of the ($\oplus$) rule obliges us to a different choice.

Some comments about the rules of $\mathcal{S}$ are in order. The rule (var) uses implicitly a weakening property. Rules ($\to I$) and ($\to E$) are similar to the usual rules for λ-calculus. Note that to the subject of the major premise is assigned a type, while to the subject of the minor premise is assigned a context type; this is possible through rule (!). The three rules for the constructor $\oplus$ are as expected. Notice that these last rules treat type environments addively, while the rule $(\to E)$ treats them multiplicatively. The use of an additive presentation for $\oplus$ rules is justified by the fact that that, in order to have the subject reduction property with respect to $\Rightarrow_{\oplus}$, we need weakening, as the following example shows.

▶ **Example 4.2.** Let $M=\lambda x.(x\oplus\mathsf{I})$. The following (incomplete) derivation can be built:

$$\dfrac{\dfrac{\dfrac{}{x:[\langle\mathtt{A}\rangle]\vdash x:\langle\mathtt{A}\rangle}\,var \qquad \dfrac{}{\vdash\mathsf{I}:\langle[\langle\mathtt{B}\rangle]\to\mathtt{B}\rangle}\,\to I}{x:[\langle\mathtt{A}\rangle]\vdash x\oplus\mathsf{I}:\langle\frac{1}{2}\mathtt{A},\frac{1}{2}([\langle\mathtt{B}\rangle]\to\mathtt{B})\rangle}\,\oplus}{\vdash\lambda x.(x\oplus\mathsf{I}):\langle\frac{1}{2}([\langle\mathtt{A}\rangle]\to\mathtt{A},\frac{1}{2}([\langle\mathtt{A}\rangle]\to[\langle\mathtt{B}\rangle]\to\mathtt{B}\rangle}\,\to I$$

Note that $[M]\Rightarrow_{\oplus}[\frac{1}{2}\mathsf{I},\frac{1}{2}\lambda x.\mathsf{I}]$; while $\vdash\mathsf{I}:\langle[\langle\mathtt{A}\rangle]\to\mathtt{A}\rangle$, it is necessary to have weakening in order to built a derivation proving $\vdash\lambda x.\mathsf{I}:\langle[\langle\mathtt{A}\rangle]\to([\langle\mathtt{B}\rangle]\to\mathtt{B})\rangle$.

In fact the weakening rule is derivable, as the following property formalizes.

▶ **Property 4.3.** $\Pi\triangleright\Gamma\vdash M:\mathtt{a}$ *implies there is* Φ *such that* $\Phi\triangleright\Gamma\uplus\Delta\vdash M:\mathtt{a}$ *and* $|\Phi|=|\Pi|$.

On the other hand, the multiplicative presentation for the rule $(\to E)$ comes naturally from the use of non idempotent intersection, which avoid the use of difficult tools to prove termination, like computability or reducibility candidates.

Rule (!) allows to assign context types to terms, and it can assign the type context $[\,]$ to any term, in case I is the empty set. It is not strictly necessary, a system without it could be easily designed, but it allows for an easy presentation of the $(\to E)$ rule. Note that the rule cannot be iterated.

The system can assign type also to terms with untyped subterms, through rules $(\oplus_l)$,$(\oplus r)$ and $(\to E)$, in case $I=\emptyset$. Consider the following examples:

▶ **Example 4.4.**

$$\dfrac{\dfrac{\dfrac{}{x:[\langle\mathtt{A}\rangle]\vdash x:\langle\mathtt{A}\rangle}\,var}{\vdash\mathsf{I}:\langle[\langle\mathtt{A}\rangle]\to\mathtt{A}\rangle}\,\to I}{\vdash\mathsf{I}\oplus\Omega:\langle\frac{1}{2}([\langle\mathtt{A}\rangle]\to\mathtt{A})\rangle}\,\oplus_l \qquad \dfrac{\dfrac{\dfrac{}{x:[\langle[\,]\to\mathtt{B}\rangle]\vdash x:\langle[\,]\to\mathtt{B}\rangle}\,var}{x:[\langle[\,]\to\mathtt{B}\rangle]\vdash x\Omega:\langle\mathtt{B}\rangle}\,\to E}{\vdash\lambda x.x\Omega:\langle[\langle[\,]\to\mathtt{B}\rangle]\to\mathtt{B}\rangle}\,\to I$$

Properties of the Type Assignment System $\mathcal{S}$. The system $\mathcal{S}$ enjoys the good properties we expect, namely subject reduction and expansion. Before going into that, we need to prove an important property of surface contexts, namely that terms filling their hole positions inherit from them both the typability and the norm, as expressed by the following lemma.

▶ **Lemma 4.5.** *If* $\Pi\triangleright\Gamma\vdash\mathbf{S}(M):\mathtt{a}$*, then there are* Δ *and* $\mathtt{b}$ *such that* $\Delta\vdash M:\mathtt{b}$*, where* $\|\mathtt{a}\|=\|\mathtt{b}\|$.

Proof. By induction on $\mathbf{S}$. If $\mathbf{S}=\square$ the proof is obvious. If $\mathbf{S}=\mathbf{T}N$, then Π is of the shape:

$$\dfrac{\Pi'\triangleright\Gamma'\vdash\mathbf{T}(M):\langle p_i(\mathcal{A}_i\to\mathtt{A}_i)\,|\,i\in I\rangle \qquad (\Delta_i\vdash N:\mathcal{A}_i)_{i\in I}}{\Gamma'\uplus_{i\in I}\Delta_i\vdash\mathbf{T}(M)N:\langle p_i\mathtt{A}_i\,|\,i\in I\rangle}\,\to E$$

We conclude by induction. If $\mathbf{S}=\lambda x.\mathbf{T}$, then the claim follows by induction, too. ◀

Typing is preserved by both reduction and expansion, but these properties, which are standard in intersection type assignment systems, must be adapted to the probabilistic setting.

▶ **Lemma 4.6** (One-Step Subject Reduction)**.** *Let* $\Pi\triangleright\Gamma\vdash M:\mathtt{a}$.

1. *If* $M \to_\beta [N]$ *then there is* $\Psi \triangleright \Gamma \vdash N : \mathtt{a}$.
2. *If* $M \to_\oplus [\frac{1}{2}N_1, \frac{1}{2}N_2]$, *then one of the two following cases happens:*
   - $\mathtt{a} = \frac{1}{2}\mathtt{a}_1 + \frac{1}{2}\mathtt{a}_2$ *and* $\Psi_1 \triangleright \Gamma \vdash N_1 : \mathtt{a}_1$, $\Psi_2 \triangleright \Gamma \vdash N_2 : \mathtt{a}_2$*;*
   - $\mathtt{a} = \frac{1}{2}\mathtt{b}$ *and* $\Psi \triangleright \Gamma \vdash N_i : \mathtt{b}$, *for some* $1 \leq i \leq 2$.

Moreover, if the redex is typed in Π, *then* $|\Psi| < |\Pi|$ *(resp.* $|\Psi_i| < |\Pi|$*).*

Proof. The proof is in the Appendix. ◀

▶ **Lemma 4.7** (Subject Reduction). $\Pi \triangleright \Gamma \vdash M : \mathtt{a}$ *and* $[M] \Rightarrow^* \mathtt{m} = [p_i N_i \mid i \in I]$ *imply there is* $J \subseteq I$, $\mathtt{a} = \Sigma_{j \in J} p_j \mathtt{a}_j$ *and* $\Pi_j \triangleright \Gamma \vdash N_j : \mathtt{a}_j$.

Proof. By induction on the lenght of the reduction, using Lemma 4.6. ◀

▶ **Lemma 4.8** (Subject Expansion). $[M] \Rightarrow^* [p_i N_i \mid i \in I]$ *and* $\Gamma \vdash N_j : \mathtt{a}_j$ *for some* $j \in J \subseteq I$ *imply* $\Delta \vdash M : \Sigma_{j \in J} p_j \mathtt{a}_j$, *for some* Δ, $\Gamma \leq \Delta$.

Proof. By induction on the length of the reduction, see the Appendix. ◀

The system can assign type to every *hnf*.

▶ **Property 4.9.** *Let* $M \in \mathcal{H}$. *Then for every* $p \in]0,1]$ *there are* Γ,$\mathtt{a}$ *such that* $\Gamma \vdash M : \mathtt{a}$ *and* $\|\mathtt{a}\| = p$.

Proof. Let $M \in \mathcal{H}$. The proof is by induction on the grammar defining $\mathcal{H}$. Let $M \in \mathcal{K}$: we will prove that, for every $\mathtt{a}$ there is Γ such that $\Gamma \vdash M : \mathtt{a}$. Let $\mathtt{a} = \langle p_i \mathtt{A}_i \mid i \in I \rangle$. If $M = x$, then choose $\Gamma = x : [\mathtt{a}]$, if $M = NP$, where $N \in \mathcal{K}$, then by induction there is Γ such that $\Gamma \vdash N : \langle p_i([\,] \to \mathtt{A}_i) \mid i \in I \rangle$ and the proof follows by rule $(\to E)$. If $M = \lambda x.N$, with $N \in \mathcal{H}$ the proof comes by induction and rule $(\to I)$. ◀

Note that the previous property says, in particular, that if a term is in *hnf*, then it is always possible to assign it a type with norm 1.

5 Characterizing Solvability

The next theorem shows the key result of this paper.

▶ **Theorem 5.1** (Finitary Characterization). *The three following statements are equivalent:*

1. $\Gamma \vdash M : \mathtt{a}$, *with* $\|\mathtt{a}\| = p$.
2. M *has* $\geq p$-hnf.
3. M *is* $\geq p$*-solvable.*

Proof. 1⇒2 Let $\Pi \triangleright \Gamma \vdash M : \mathtt{a}$; we prove that $[M] \Rightarrow^* \mathtt{m}$ with $\mu_\mathtt{m}(\mathcal{H}) \geq \|\mathtt{a}\|$, by induction on $|\Pi|$. Note that if M is in head normal form, the claim holds. If $|\Pi| = 1$, then $M = x$ is in *hnf*. Let $|\Pi| > 1$. If M is in *hnf*, the claim holds; if M not in *hnf*, then, according to Lemma 4.6, three cases can happen:

a $M \overset{\mathsf{s}}{\to}_\beta [N]$, and $\Psi \triangleright \Gamma \vdash N : \mathtt{a}$;

b $M \overset{\mathsf{s}}{\to}_\oplus [\frac{1}{2}N_1, \frac{1}{2}N_2]$, $\mathtt{a} = \frac{1}{2}\mathtt{a}_i$, and $\Psi \triangleright \Gamma \vdash N_i : \mathtt{a}_i$, for some $i \in \{1,2\}$;

c $M \overset{\mathsf{s}}{\to}_\oplus [\frac{1}{2}N_1, \frac{1}{2}N_2]$, $\mathtt{a} = \frac{1}{2}\mathtt{a}_1 + \frac{1}{2}\mathtt{a}_2$, and $\Psi_i \triangleright \Gamma \vdash N_i : \mathtt{a}_i$, for all $i \in \{1,2\}$;

and in all cases $|\Psi| < |\Pi|$, $|\Psi_i| < |\Pi|$ by Lemma 4.5. In case **a**, the result follows by induction on the structure of Ψ. In the case **b**, by induction it holds that $[N_i] \Rightarrow^* \mathtt{n}_i$ with $\mu_{\mathtt{n}_i}(\mathcal{H}) \geq \|\mathtt{a}_i\|$, for some $1 \leq i \leq 2$. Assume $i = 1$. Then $[M] \Rightarrow^* \frac{1}{2}\mathtt{n}_1 + \frac{1}{2}[N_2] = \mathtt{m}$, so $\mu_\mathtt{m}(\mathcal{H}) \geq \frac{1}{2}\mu_{\mathtt{n}_1}(\mathcal{H}) \geq \|\mathtt{a}\|$. Let us consider case **c**. By i.h., it holds that $[N_i] \Rightarrow^* \mathtt{n}_i$ with $\mu_{\mathtt{n}_i}(\mathcal{H}) \geq \|\mathtt{a}_i\|$ for every $i \in \{1,2\}$. Hence we have that $M \overset{\mathsf{s}}{\to} [\frac{1}{2}N_1, \frac{1}{2}N_2] \Rightarrow^* (\frac{1}{2}\mathtt{n}_1 + \frac{1}{2}\mathtt{n}_2) = \mathtt{m}$, where $\mu_\mathtt{m}(\mathcal{H}) = \frac{1}{2}\mu_{n_1}(\mathcal{H}) + \frac{1}{2}\mu_{n_2}(\mathcal{H}) \geq_{i.h.} \frac{1}{2}\|\mathtt{a}_1\| + \frac{1}{2}\|\mathtt{a}_2\| = \|\mathtt{a}\|$.

2⇒3 Assume M in $\mathcal{H}$; we prove that M is 1-solvable, by showing how to build a particular head context $\mathbf{S}$ such that $[\mathbf{S}(M)] \Rightarrow^* [\mathsf{I}]$. Let $M = \lambda x_1...x_n.zM_1...M_m$. If $z = x_i$, for some i, then the context $\square\underbrace{(\lambda z_1...z_m.\mathsf{I})}_{n}$ does the job. If z is free, then use the context $\lambda z.\square(\lambda z_1...z_m.\mathsf{I})$. For the general case, let $[M] \Rightarrow^* \mathtt{m}$, and $\mu_{\mathtt{m}}(\mathcal{H}) = q \geq p$. Let $\mathtt{m} = [q_1N_1,...,q_nN_n] + \mathtt{n}$, where $\mu_{\mathtt{n}}(\mathcal{H}) = 0$ and $\Sigma_{1\leq i\leq n}q_i \geq p$. W.l.o.g., we assume M closed. Every N_i is of the shape $\lambda\vec{x_i}.z^i\vec{P_i}$, where $z^i \in \vec{x_i}$; let $r = max_{1\leq i\leq n}|\vec{x_i}|$ and $s{=}max_{1\leq i\leq n}|\vec{P_i}|$. Choose $w_1,..,w_r$ fresh variables. Then the desired context is:

$$\mathbf{H}{=}(\lambda w_1...w_r.\square w_1...w_r)\underbrace{(\lambda t_1...t_{r+s}.\mathsf{I})...(\lambda t_1...t_{r+s}.\mathsf{I})}_{r}\underbrace{\mathsf{I}...\mathsf{I}}_{r+s}$$

$[\mathbf{S}(M)]{\Rightarrow^*}[q_1\mathbf{S}(N_1),...,q_n\mathbf{S}(N_n)]{+}\mathtt{p}$. It is sufficient to prove that $[\mathbf{S}(N_i)]{\Rightarrow^*}[\mathsf{I}]$. Now,

$$[\mathbf{S}(N_i)]{\Rightarrow^*_\beta}[(\lambda w_1...w_r.w^i\vec{P'_i})\underbrace{(\lambda t_1...t_{r+s}.\mathsf{I})...(\lambda t_1...t_{r+s}.\mathsf{I})}_{r}\underbrace{\mathsf{I}...\mathsf{I}}_{r+s}],$$

where $\vec{P'_i}{=}\vec{Q}_i w_{|P_i|+1}...w_r$, $w^i{\in}\{w_1,...,w_r\}$, $Q_i{=}P_i[\vec{w_i}/\vec{x_i}]$ and $|\vec{P'_i}|{\leq}|\vec{P_i}|{+}r{-}1{\leq}s{+}r{-}1$. Then, after r reduction steps, we obtain: $[(\lambda t_1...t_{r+s}.\mathsf{I})\vec{P''_i}\underbrace{\mathsf{I}...\mathsf{I}}_{r+s}] \Rightarrow^*_\beta [(\lambda\vec{t}.\mathsf{I})\underbrace{\mathsf{I}...\mathsf{I}}_{l}]$, where $l{\leq}r{+}s$. Since s $|\vec{t}|{\leq}r{+}s$, $[(\lambda\vec{t}.\mathsf{I})\underbrace{\mathsf{I}...\mathsf{I}}_{l}]{\Rightarrow^*_\beta}[\underbrace{\mathsf{I}...\mathsf{I}}_{o}]{\Rightarrow^*_\beta}[I]$ $(o{\leq}r{+}s{-}1)$.

3⇒1 Let M be $\geq p$-solvable. Then there is $\mathbf{S}$ such that $[\mathbf{S}(M)]{\Rightarrow^*}[p_i\mathsf{I} \mid i \in I]{+}\mathtt{m}$, where $\Sigma_{i\in I}p_i \geq p$. By Property 4.9, there is $\mathtt{a}$, with $\|\mathtt{a}\| = 1$, such that $\vdash \mathsf{I} : \mathtt{a}$, so, by Subject Expansion, $\Gamma{\vdash}\mathbf{S}(M){:}\Sigma_{i\in I}p_i\mathtt{a}$. By Lemma 4.5, $\Gamma{\vdash}M{:}\mathtt{b}$, where $\|\mathtt{b}\|{=}\Sigma_{i\in I}p_i\|\mathtt{a}\|$.

◀

The results of Theorem 5.1 can be extended to the supremum, this way enabling a complete characterization:

▶ **Theorem 5.2** (Characterization). *The three following statements are equivalent*

1. $p{=}sup\{q \mid \Gamma{\vdash}M{:}\mathtt{a}$, *for some* Γ,$\mathtt{a}$, *and* $q{=}\|\mathtt{a}\|\}$.
2. M *has* p-hnf.
3. M *is* p-*solvable.*

Theorem 5.2 implicitly supplies three different characterizations of solvability, similarly to what happens in the λ-calculus. Namely, the equivalence 2⇔3 corresponds to an *operational characterization* of solvability, and 1⇔3 corresponds to a *logical characterization.* Moreover 1⇔2 gives a logical characterization of *hnf*s.

A Model for $\Lambda_\oplus$. $\mathcal{S}$ is an extension of the basic type assignment system defined in [26], which gives rise to a relational model of λ-calculus. It is possible to reason in a similar way here, and to extract from $\mathcal{S}$ a model of $\Lambda_\oplus$, in the sense specified by Property 5.3. As has been proved in [26], following a seminal observation of [10], the interpretation of a term in a model extracted from a type assignment system with non-idempotent intersections, depends not only on the types derivable for it, but also on the related type contexts. In fact, the context is necessary to preserve the quantitative aspect of types. Let us define the basic ingredients of our model. An *abstract typing* is a pair $(\Gamma;\mathtt{a})$, where Γ is a type context and $\mathtt{a}$ is a type, not necessarily related to each-other. Let $\mathcal{T}$ be the set of abstract typings: the space $\mathcal{D}$ of *denotations* of our model is the power set of $\mathcal{T}$. $\mathcal{D}$ is equipped by two operations

$\circ,\oplus:\mathcal{D}\longrightarrow\mathcal{D}$ which allow to interpret terms of $\Lambda_\oplus$. Their definitions reflect, respectively, the behavior of the typing rules $(\to E)$ and $(\oplus)$ of $\mathcal{S}$:

$$\begin{aligned}
t_1\circ t_2 &= \{(\Gamma;\langle p_i\mathtt{A}_i \mid i\in I\rangle) \mid (\Gamma;\langle p_i([]\to\mathtt{A}_i)\mid i\in I\rangle\in t_1\}\cup\\
&\qquad \left\{(\Gamma\uplus_{i\in I}\Delta_i;\langle p_i\mathtt{A}_i\mid i\in I\rangle \left| \begin{array}{l}(\Gamma;\langle p_i(\mathcal{A}_i\to\mathtt{A}_i)\mid i\in I\rangle\in t_1,\mathcal{A}_i=[\mathtt{a}^i_j\mid j\in J_i],\\ (\Delta^i_j;\mathtt{a}^i_j)\in t_2,\Delta_i=\uplus_{j\in J_i}\Delta^i_j\end{array}\right.\right\};\\
t_1\oplus t_2 &= \left\{\left(\Gamma;\frac{1}{2}\mathtt{a}\right)\mid(\Gamma;\mathtt{a})\in t_1\right\}\cup\left\{\left(\Gamma;\frac{1}{2}\mathtt{a}\right)\mid(\Gamma;\mathtt{a})\in t_2\right\}\cup\\
&\qquad \left\{\left(\Gamma;\frac{1}{2}\mathtt{a}_1+\frac{1}{2}\mathtt{a}_2\right)\mid(\Gamma;\mathtt{a}_i)\in t_i,i=1,2\right\}.
\end{aligned}$$

Moreover, if $t\in\mathcal{D}$ and p is a probability, $p\bullet t$ denotes the element of $\mathcal{D}$ such that, if $(\Gamma;\mathtt{a})\in t$ then $(\Gamma;p\mathtt{a})\in p\bullet t$. Let ρ be a denotational environment, assigning an element of $\mathcal{D}$ to every variable: the interpretation of a term under the environment ρ is defined by induction as follows:

$$\begin{aligned}
[\![x]\!]_\rho &= \rho(x)\\
[\![MN]\!]_\rho &= [\![M]\!]_\rho\circ[\![N]\!]_\rho\\
[\![M\oplus N]\!]_\rho &= [\![M]\!]_\rho\oplus[\![N]\!]_\rho\\
[\![\lambda x.M]\!]_\rho &= \{(\Gamma;\langle p_i(\mathcal{A}\to\mathtt{A}_i)\mid i\in I\rangle \mid \mathcal{A}=[\mathtt{a}_j\mid j\in J],(\Gamma\uplus_{j\in J}\Delta_j;\langle p_i\mathtt{A}_i\mid i\in I\rangle)\in[\![M]\!]_{\rho[t/x]},\\
&\qquad t=\{(\Delta_j;\mathtt{a}_j)\mid j\in J\}\}
\end{aligned}$$

It is easy to check that the interpretation of a term is related to its concrete typings in the following way:

$$\begin{aligned}
[\![M]\!]_\rho = \{(\Gamma;\mathtt{a})\mid \Delta\vdash M:\mathtt{a},\Gamma=\uplus_{i\in I}\Delta_i \text{ such that for every } x,\\
\Delta(x)=[\mathtt{a}_i\mid i\in I] \text{ implies } (\Delta_i;\mathtt{a}_i)\in\rho(x)\}
\end{aligned}$$

If M is a closed term, then its interpretation is even simpler, namely:

$$[\![M]\!]=\{(\Gamma;\mathtt{a})\mid\exists\Gamma,\mathtt{a}.\Gamma\vdash M:\mathtt{a}\}$$

In particular, since M closed and $\Gamma\vdash M:\mathtt{a}$ together imply that $\vdash M:\mathtt{a}$, the interpretation of a closed term depends only on the types derivable for it. In the following we will restrict ourselves to consider only closed terms: clearly all the properties we prove hold also for the open terms, but are expressed in a more cumbersome way.

The model is correct with respect to the operational behavior of $\Lambda_\oplus$, i.e., the following property holds.

▶ **Property 5.3** (Adequacy). *Let M be closed. $M\Rightarrow^*[p_iM_i\mid i\in I]$ implies $[\![M]\!]=\cup_{i\in I}p_i\bullet[\![M_i]\!]$.*

Finally, the model characterizes solvability, in the following sense:

▶ **Property 5.4.** *Let M be closed. M is p-solvable if and only if $p=sup\{q\mid(\Gamma;\mathtt{a})\in[\![M]\!]\}$ and $\|\mathtt{a}\|=q$.*

Note that M is unsolvable if and only if $[\![M]\!]_\rho=\emptyset$ for every ρ, so, using the terminology of λ-calculus, this model is sensible, since it equates all unsolvable terms.

6 Conclusions and Future Work

We investigated the notion of solvability in the context of the calculus $\Lambda_\oplus$ as introduced in [15], and focusing on the call-by-name parameter passing regime. Solvability being a semantic property, and call-by-name and call-by-value behaving quite differently semantically [30], we leave the task of extending our work to call-by-value to some future work. The definition of solvability we give is a conservative extension of the one from the pure λ-calculus, and explicitly takes probability into account: a term is dubbed p-solvable if, put in a suitable context, it reduces to the identity with probabilities *at most*, but *arbitrary close to*, p. We characterize solvability through a type assignment system based on non-idempotent intersection types. Such a system supplies a logical characterization of solvability, but also induces an operational one in which being p-solvable corresponds to having head normal form with probability p. Finally, the type system induces a model for $\Lambda_\oplus$, in which all unsolvable terms (i.e., terms which are 0-solvable) are equated.

It would be interesting to study the theory induced by our model from a finer point of view, in particular with respect to the equivalence it induces on terms. Certainly, this equivalence cannot coincide with the operational one, characterized in [23], since our model is not extensional. The type assignment system could however be enriched with an equivalence between types, in such a way as to induce an extensional model, thus catching the operational semantics of $\Lambda_\oplus$, in the sense of Plotkin.

Moreover, we intend to give a domain-theoretic account of our model: we believe that it gives a logical description of the category of weighted relational models [21], as conjectured by an anonymous referee.

References

1 Martin Avanzini, Ugo Dal Lago, and Alexis Ghyselen. Type-based complexity analysis of probabilistic functional programs. In *Proc. of LICS 2019*, pages 1–13, 2019.

2 Martin Avanzini, Ugo Dal Lago, and Akihisa Yamada. On probabilistic term rewriting. In *Proc. of FLOPS 2018*, pages 132–148, 2018.

3 H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103. North Holland, 1984.

4 Flavien Breuvart and Ugo Dal Lago. On intersection types and probabilistic lambda calculi. In David Sabel and Peter Thiemann, editors, *Proc. of PPDP 2018*, pages 8:1–8:13, 2018.

5 Pierre Clairambault and Hugo Paquet. Fully abstract models of the probabilistic lambda-calculus. In *Proc. of CSL 2018*, pages 16:1–16:17, 2018.

6 Ugo Dal Lago and Charles Grellois. Probabilistic termination by monadic affine sized typing. *ACM Trans. Program. Lang. Syst.*, 41(2):10:1–10:65, 2019.

7 Ugo Dal Lago, Davide Sangiorgi, and Michele Alberti. On coinductive equivalences for higher-order probabilistic functional programs. In *Proc. of POPL 2014*, pages 297–308, 2014.

8 Ugo Dal Lago and Margherita Zorzi. Probabilistic operational semantics for the lambda calculus. *RAIRO - Theor. Inf. and Applic.*, 46(3):413–450, 2012.

9 Vincent Danos and Thomas Ehrhard. Probabilistic coherence spaces as a model of higher-order probabilistic computation. *Inf. Comput.*, 209(6):966–991, 2011.

10 Daniel de Carvalho. Execution time of λ-terms via denotational semantics and intersection types. *Math. Struct. Comput. Sci.*, 28(7):1169–1203, 2018.

11 Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Probabilistic lambda-calculus and quantitative program analysis. *J. Log. Comput.*, 15(2):159–179, 2005.

12 T. Ehrhard, C. Tasson, and M. Pagani. Probabilistic coherence spaces are fully abstract for probabilistic PCF. In *Proc. of POPL 2014*, pages 309–320, 2014.

13 Thomas Ehrhard, Michele Pagani, and Christine Tasson. The computational meaning of probabilistic coherence spaces. In *Proc. of LICS 2011*, pages 87–96, 2011.

14 Thomas Ehrhard and Laurent Regnier. The differential lambda-calculus. *Theor. Comput. Sci.*, 309(1-3):1–41, 2003.

15 Claudia Faggian and Simona Ronchi Della Rocca. Lambda calculus and probabilistic computation. In *Proc. of LICS 2019*, pages 1–13, 2019.

16 Jean Goubault-Larrecq. A probabilistic and non-deterministic call-by-push-value language. In *Proc. of LICS 2019*, pages 1–13, 2019.

17 Claire Jones. *Probabilistic non-determinism.* PhD thesis, University of Edinburgh, UK, 1990.

18 Achim Jung and Regina Tix. The troublesome probabilistic powerdomain. *Electron. Notes Theor. Comput. Sci.*, 13:70–91, 1998.

19 Simona Kasterovic and Michele Pagani. The discriminating power of the let-in operator in the lazy call-by-name probabilistic lambda-calculus. In *Proc. of FSCD 2019*, pages 26:1–26:20, 2019.

20 Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography.* Chapman & Hall Cryptography and Network Security Series. Chapman & Hall, 2007.

21 J. Laird, G. Manzonetto, G. McCusker, and M. Pagani. Weighted relational models of typed lambda-calculi. In *proc. of LICS 2013*, pages 301–310, 2013.

22 Thomas Leventis. *Probabilistic lambda-theories.* Phd Thesis, Aix-Marseille Université, 2016. Available at `https://tel.archives-ouvertes.fr/tel-01427279v2/document`.

23 Thomas Leventis. Probabilistic böhm trees and probabilistic separation. In *Proc. of LICS 2018*, pages 649–658, 2018.

24 Thomas Leventis and Michele Pagani. Strong adequacy and untyped full-abstraction for probabilistic coherence spaces. In *Proc. of FoSSaCS 2019*, pages 365–381, 2019.

25 Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

26 Luca Paolini, Mauro Piccolo, and Simona Ronchi Della Rocca. Essential and relational models. *Math. Struct. Comput. Sci.*, 27(5):626–650, 2017.

27 Sungwoo Park. A calculus for probabilistic languages. In *Proc. of TLDI 2003*, pages 38–49, 2003.

28 Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

29 Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Proc. of POPL 2002*, pages 154–165, 2002.

30 Simona Ronchi Della Rocca and Luca Paolini. *The Parametric Lambda Calculus - A Metamodel for Computation.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

31 N. Saheb-Djahromi. Probabilistic LCF. In *Proc. of MFCS 1978*, pages 442–451, 1978.

32 Lionel Vaux. The algebraic lambda calculus. *Mathematical Structures in Computer Science*, 19(5):1029–1059, 2009.

A Some Technical Proofs

Subject Reduction. As usual, the subject reduction property relies on a substitution property.

▶ **Lemma A.1** (Substitution). $\Pi \triangleright \Gamma, x : \mathcal{A} \vdash M : \mathtt{a}$ *(resp.* $\Pi \triangleright \Gamma, x : \mathcal{A} \vdash M : \mathcal{B}$*) and* $\Theta \triangleright \Delta \vdash N : \mathcal{A}$ *imply* $\Pi[\Theta/x] \triangleright \Gamma \uplus \Delta \vdash M[N/x] : \mathtt{a}$ *(resp.* $\Pi[\Theta/x] \triangleright \Gamma \vdash M[N/x] : \mathcal{B}$*). Moreover,* $|\Pi[\Theta/x]| < |\Pi| + |\Theta|$.

Proof. By induction on Π:

- Let Π be:

$$\frac{a \in \mathcal{A}}{\Gamma, y : \mathcal{A} \vdash y : \mathtt{a}}\ var$$

On the other side, the derivation Θ is of the shape:

$$\frac{(\Pi_i \triangleright \Gamma_i \vdash N : \mathtt{a}_i)_{i \in I}}{\uplus_{i \in I} \Gamma_i \vdash N : [\mathtt{a}_i \mid i \in I]}\ !$$

so $\mathtt{a} = \mathtt{a}_i$, for some i. Let $y = x$. By Property 4.3, there is a derivation $\Xi \triangleright \Gamma \uplus_{i \in I} \Gamma_i \vdash N : \mathtt{a}_i$, such that $|\Xi| = |\Pi_i|$. So $\Pi[\Theta/x] = \Xi$. If $y \neq x$, then $\Pi[\Theta/x] = \Pi$. In both cases the condition on the size of $\Pi[\Theta/x]$ is obvious.

- Let Π be:

$$\frac{\Xi \triangleright \Gamma, x : \mathcal{A}' \vdash P : \langle p_i(\mathcal{B}_i \to \mathtt{B}_i) \mid i \in I \rangle \quad (\Pi_i \triangleright \uplus_{i \in I} \Sigma_i, x : \mathcal{A}_i \vdash Q : \mathcal{B}_i)_{i \in I}}{\Gamma \uplus_{i \in I} \Sigma_i \uplus x : (\mathcal{A}' +_{i \in I} \mathcal{A}_i) \vdash PQ : \langle p_i \mathtt{B}_i \mid i \in I \rangle}\ \to E$$

where $\mathcal{A} = \mathcal{A}' +_{i \in I} \mathcal{A}_i$. Then the derivation Θ is:

$$\frac{(\Phi_{\mathtt{b}} \triangleright \Delta_{\mathtt{b}} \vdash N : \mathtt{b})_{\mathtt{b} \in \mathcal{A}'} \quad ((\Phi^i_{\mathtt{c}} \triangleright \Delta^i_{\mathtt{c}} \vdash N : \mathtt{c})_{\mathtt{c} \in \mathcal{A}_i})_{i \in I}}{+_{\mathtt{b} \in \mathcal{A}'} \Delta_{\mathtt{b}} +_{\mathtt{c} \in \mathcal{A}', i \in I} \Delta^i_{\mathtt{c}} \vdash N : \mathcal{A}}\ !$$

Then we can build derivations Ψ and Ψ_i, with subject N, by rule (!) with premises respectively $(\Phi_{\mathtt{b}})_{\mathtt{b} \in \mathcal{A}'}$ and $(\Phi^i_{\mathtt{c}})_{\mathtt{c} \in \mathcal{A}_i}$; by induction there are $\Xi[\Psi/x] \triangleright P[N/x] : \langle p_i(\mathcal{B}_i \to \mathtt{b}_i) \mid i \in I \rangle$ and $(\Pi_i[\Psi_i/x] \triangleright \vdash Q[N/x] : \mathcal{B}_i)$. Since $PQ[N/x] = P[N/x]Q[N/x]$ the result follows by rule $(\to E)$. Moreover by induction $|\Xi[\Psi/x]| < |\Xi| + |\Psi|$, $|\Pi_i[\Psi_i/x]| < |\Pi_i| + |\Psi_i|$, so $|\Pi[\Theta/x]| = |\Xi[\Psi/x]| +_{i \in I} |\Pi_i[\Psi_i/x]| + 1 < |\Xi| + |\Psi| +_{i \in I} (|\Pi_i| + |\Psi_i|) + 1 = |\Pi| + |\Phi'|$

- Let Π be:

$$\frac{\Pi_1 \triangleright \Gamma, x : \mathcal{A} \vdash P : \mathtt{a} \quad \Pi_2 \triangleright \Gamma, x : \mathcal{A} \vdash N : \mathtt{b}}{\Gamma, x : \mathcal{A} \vdash P \oplus Q : \frac{1}{2}\mathtt{a} + \frac{1}{2}\mathtt{b}}\ \oplus$$

By induction there are $\Pi_1[\Theta/x] \triangleright \Gamma \uplus \Delta \vdash P[N/x] : \mathtt{a}$ and $\Pi_2[\Theta/x] \triangleright \Gamma \uplus \Delta \vdash Q[N/x] : \mathtt{b}$, so $\Pi[\Theta/x]$ can be built by rule $(\oplus)$. Moreover $|\Pi[\Theta/x]| = max\{|\Pi_1[\Theta/x]|, |\Pi_2[\Theta/x]|\} + 1 <_{i.h.} max\{(|\Pi_1| + |\Theta|, |\Pi_2| + |\Theta|) + 1 = max\{|\Pi_1|, |\Pi_2|\} + |\Theta| + 1 = |\Pi| + |\Theta|$.

- If the last used rule is $(\to I)$, $(\oplus_r)$, $\oplus_l)$ the proof follows by induction.

◀

Let us identify an occurrence of a term N in a term M by the context $\mathbf{C}$ such that $M = \mathbf{C}(N)$. Then, given a typing derivation $\Pi \triangleright \Gamma \vdash M : \mathtt{a}$, an occurrence of a subterm of M is a typed occurrence of Π if and only if it is the subject of a subderivation of Π.

▶ **Lemma** (4.6, One-Step Subject Reduction). *Let $\Pi \triangleright \Gamma \vdash M : \mathtt{a}$.*

1. *If $M \to_\beta [M']$ then there is $\Pi' \triangleright \Gamma \vdash M' : \mathtt{a}$.*
2. *If $M \to_\oplus [\frac{1}{2}M_1, \frac{1}{2}M_2]$, then one of the two following cases happens:*
   - $\mathtt{a} = \frac{1}{2}\mathtt{a}_1 + \frac{1}{2}\mathtt{a}_2$ *and* $\Pi'_1 \triangleright \Gamma \vdash M_1 : \mathtt{a}_1$, $\Pi'_2 \triangleright \Gamma \vdash M_2 : \mathtt{a}_2$*;*
   - $\mathtt{a} = \frac{1}{2}\mathtt{b}$ *and* $\Pi' \triangleright \Gamma \vdash M_i : \mathtt{b}$*, for some* i $(i \in \{1,2\})$*.*

   Moreover, if the redex is typed in Π, then $|\Pi'| < |\Pi|$ (resp. $|\Pi'_i| < |\Pi|$).

Proof.

1. By induction on the context $\mathbf{C}$ such that $M = \mathbf{C}((\lambda x.P)Q)$ and $M' = \mathbf{C}(P[Q/x])$. The base case follows by Lemma A.1, the induction case is easy.

2. By induction on the context $\mathbf{S}$ such that $M=\mathbf{S}(P\oplus Q)$, and then by induction on Π.
   a. In case $\mathbf{S}=\Box$, then the last rule of Π is $\oplus$, and the proof is obvious. Let $\mathbf{S}=\lambda x.\mathbf{S}'$. So $M=\lambda x.N\to_{\oplus}[\frac{1}{2}\lambda x.N_1,\frac{1}{2}\lambda x.N_2]$, so $N\to_{\oplus}[\frac{1}{2}N_1,\frac{1}{2}N_2]$. Π is of the shape:

$$\frac{\Pi'\rhd\Gamma,x:\mathcal{A}\vdash N:\langle p_i\mathtt{A}_i\,|\,i\in I\rangle}{\Gamma\vdash\lambda x.N:\langle p_i(\mathcal{A}\to\mathtt{A}_i)\,|\,i\in I\rangle}\to I$$

   by induction $\langle p_i\mathtt{A}_i\,|\,i\in I\rangle=\langle\frac{q_i}{2}\mathtt{A}_i\,|\,i\in I_1\rangle+\langle\frac{q_i}{2}\mathtt{A}_i\,|\,i\in I_2\rangle$, and $\Gamma\vdash N_j:\langle q_i\,|\,i\in I_j\rangle$, where $I=I_1\cup I_2$, $p_i=\frac{q_i}{2}$ $(j=1,2)$. Then by rule $(\to I)$, $\Pi_j\rhd\Gamma\vdash\lambda x.N_j:\langle q_i(\mathcal{A}\to\mathtt{A}_i)\,|\,i\in I_j\rangle$ $(j=1,2)$, and then, by rule $(\oplus)$, $\Phi\rhd\Gamma\vdash\lambda x.N_1\oplus\lambda x.N_2:\frac{1}{2}\langle q_i(\mathcal{A}\to\mathtt{A}_i)\,|\,i\in I_1\rangle+\frac{1}{2}\langle q_i(\mathcal{A}\to\mathtt{A}_i)\,|\,i\in I_2\rangle=\langle p_i(\mathcal{A}\to\mathtt{A}_i)\,|\,i\in I\rangle$. Moreover, if the redex is typed in Π, then it is typed in Π', so by induction $|\Pi'_i|<|\Pi'|$. Then $|\Phi|=\frac{1}{2}|\Pi_1|+\frac{1}{2}|\Pi_2|=\frac{1}{2}(|\Pi'_1|+|\Pi'_2|)+1<|\Pi'|+1=|\Pi|$. Let $\mathbf{S}=\mathbf{S}'P$. Then $M=NP\to_{\oplus}[\frac{1}{2}N_1P,\frac{1}{2}N_2P]$ and Π is of the shape:

$$\frac{\Pi'\rhd\Sigma\vdash N:\langle p_i(\mathcal{A}_i\to\mathtt{A}_i)\,|\,i\in I\rangle\quad(\Psi_i\rhd\Delta_i\vdash P:\mathcal{A}_i)_{i\in I}}{\Gamma=\Sigma\uplus_{i\in I}\Delta_i\vdash NP:\langle p_i\mathtt{A}_i\,|\,i\in I\rangle}\to E$$

   By induction on Π', $I=I_1\cup I_2$ and $\langle p_i(\mathcal{A}_i\to\mathtt{A}_i)\mid i\in I\rangle=\langle\frac{1}{2}p_i(\mathcal{A}_i\to\mathtt{A}_i)\mid i\in I_1\rangle+\langle\frac{1}{2}p_i(\mathcal{A}_i\to\mathtt{A}_i)\mid i\in I_2\rangle$ such that $\Pi_1\rhd\Sigma\vdash N_1:\langle p_i(\mathcal{A}_i\to\mathtt{A}_i)\mid i\in I_1\rangle$ and $\Pi_2\rhd\Sigma\vdash N_2:\langle p_i(\mathcal{A}_i\to\mathtt{A}_i)\,|\,i\in I_2\rangle$, so, by rule $(\to E)$, $\Phi_j\rhd\Sigma\uplus_{i\in I_j}\Delta_j\vdash N_jP:\langle p_i\mathtt{A}_i\,|\,i\in I_j\rangle$ $(j\in\{1,2\})$. By Property 4.3, $\Sigma\uplus_{i\in I}\Delta_i\vdash N_jP:\langle p_i\mathtt{A}_i\,|\,i\in I_j\rangle$. So, by rule $(\oplus)$, we obtain $\Theta\rhd\Gamma\vdash N_1P\oplus N_2P:\frac{1}{2}\langle p_i\mathtt{A}_i\,|\,i\in I_1\rangle+\frac{1}{2}\langle p_i\mathtt{A}_i\,|\,i\in I_2\rangle$. Moreover, if the redex is typed in Π, then it is typed in Π', so by induction $|\Pi_i|<|\Pi'|$. Since $|\Phi_j|=|\Pi_j|+_{i\in I_j}|\Psi_i|+1$, we have: $|\Theta|=\frac{1}{2}|\Phi_1|+\frac{1}{2}|\Phi_2|=\frac{1}{2}(|\Pi_1|+_{i\in I_1}|\Psi_i|+1)+\frac{1}{2}(|\Pi_2|+_{i\in I_2}|\Psi_i|+1)=\frac{1}{2}(|\Pi_1|+|\Pi_2|)+_{i\in I}|\Psi_i|+1<|\Pi'|+_{i\in I}|\Psi_i|+1=|\Pi|$.
   b. Similar to the previous case, but easier.

If the redex occurs in an untyped occurence of Π, then, by Lemma 4.5, it is a β-redex. So, if $\Pi\rhd\Gamma\vdash\mathbf{C}(M):\mathtt{a}$, and $M\to_{\beta}M'$, then $\Pi'\rhd\Gamma\vdash\mathbf{C}(M'):\mathtt{a}$ can be obtained from Π just replacing the occurrence of M by M'. ◀

Subject Expansion.

▶ **Lemma A.2** (Inverse Substitution). *$\Pi\rhd\Gamma\vdash M[N/x]:\mathtt{a}$ implies there is $\mathcal{A}$ such that $\Sigma,x:\mathcal{A}\vdash M:\mathtt{a}$, $\Delta\vdash N:\mathcal{A}$ and $\Gamma\subseteq\Sigma\uplus\Delta$.*

Proof. By induction on M. All the cases follow easily by induction. Note that, in case all the occurrences of N in M are untyped in Π, then $\mathcal{A}=[\,]$ and $\Sigma=\Gamma$. ◀

▶ **Lemma A.3** (One-Step Subject Expansion).
1. *$\Pi\rhd\Gamma\vdash M:\mathtt{a}$ and $N\to_{\beta}[M]$ imply $\Gamma\vdash N:\mathtt{a}$.*
2. *$\Pi\rhd\Gamma\vdash M:\mathtt{a}$ and $N\to_{\oplus}[\frac{1}{2}M,\frac{1}{2}P]$ imply $\Gamma\to N:\frac{1}{2}\mathtt{a}$.*
3. *$\Pi\rhd\Gamma\vdash M_i:\mathtt{a}_i$ for every $1\leq i\leq 2$ and $N\to_{\oplus}[\frac{1}{2}M_1,\frac{1}{2}M_2]$ imply $\Gamma\vdash N:\frac{1}{2}\mathtt{a}_1+\frac{1}{2}\mathtt{a}_2$.*

Proof.
1. By Lemma A.2.
2. By induction on the context $\mathbf{S}$ such that $N=\mathbf{S}(R_1\oplus R_2)\to_{\oplus}[\frac{1}{2}\mathbf{S}(R_1),\frac{1}{2}\mathbf{S}(R_2)]$ and either $M=\mathbf{S}(R_1)$ or $P=\mathbf{S}(R_2)$, and then by induction on Π. In case $\mathbf{S}=\Box$, the proof is obvious. Let $\mathbf{S}=\lambda x.\mathbf{S}'$, so $N=\lambda x.Q\to_{\oplus}[\frac{1}{2}\lambda x.N_1,\frac{1}{2}\lambda x.N_2]$, where $Q\to_{\oplus}[\frac{1}{2}N_1,\frac{1}{2}N_2]$. Let $\Pi\rhd\Gamma\vdash\lambda x.N_1:\mathtt{a}$. Then Π is of the shape:

$$\frac{\Gamma,x:\mathcal{A}\vdash N_1:\langle p_i\mathtt{B}_i\,|\,i\in I\rangle}{\Gamma\vdash\lambda x.N_1:\langle p_i(\mathcal{A}\to\mathtt{B}_i\,|\,i\in I\rangle}\to I$$

By induction $\Gamma, x\!:\!\mathcal{A} \vdash Q : \langle \frac{1}{2} p_i \mathtt{B}_i \mid i \in I \rangle$, and then, by rule $(\rightarrow I)$, $\Gamma \vdash \lambda x.Q : \langle \frac{1}{2} p_i (\mathcal{A} \rightarrow \mathtt{B}_i) \mid i \in I \rangle$. Let $\mathbf{S} = \mathbf{S}' R$, so $N = QR \rightarrow_{\oplus} [\frac{1}{2} N_1 R, \frac{1}{2} N_2 R]$. Then Π is of the shape:

$$\frac{\Gamma \vdash N_1 : \langle p_i (\mathcal{A}_i \rightarrow \mathtt{A}_i) \mid i \in I \rangle \quad (\Delta_i \vdash R : \mathcal{A}_i)_{i \in I}}{\Gamma \uplus_{i \in I} \Delta_i \vdash N_1 R : \langle p_i \mathtt{A}_i \mid i \in I \rangle} \rightarrow E$$

By induction, $\Gamma \vdash Q : \langle \frac{1}{2} p_i (\mathcal{A}_i \rightarrow \mathtt{A}_i) \mid i \in I \rangle$, so the proof follows by rule $(\rightarrow E)$. ◀

A Modal Analysis of Metaprogramming, Revisited

Brigitte Pientka
McGill University, Montreal, QC, Canada
`http://www.cs.mcgill.ca/~bpientka`
bpientka@cs.mcgill.ca

Abstract

Metaprogramming is the art of writing programs that produce or manipulate other programs. This opens the possibility to eliminate boilerplate code and exploit domain-specific knowledge to build high-performance programs. Unfortunately, designing language extensions to support type-safe multi-staged metaprogramming remains very challenging.

In this talk, we outline a modal type-theoretic foundation for multi-staged metaprogramming which supports the generation and the analysis of polymorphic code. It has two main ingredients: first, we exploit contextual modal types to describe open code together with the context in which it is meaningful; second, we model code as a higher-order abstract syntax (HOAS) tree within a context. These two ideas provide the appropriate abstractions for both generating and pattern matching on open code without committing to a concrete representation of variable binding and contexts.

Our work is a first step towards building a general type-theoretic foundation for multi-staged metaprogramming which on the one hand enforces strong type guarantees and on the other hand makes it easy to generate and manipulate code. This will allow us to exploit the full potential of metaprogramming without sacrificing reliability of and trust in the code we are producing and running.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Type structures; Theory of computation → Lambda calculus; Theory of computation → Modal and temporal logics

Keywords and phrases Type systems, Metaprogramming, Modal Type System

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.2

Category Invited Talk

Funding *Brigitte Pientka*: This work has been funded by the Natural Sciences and Engineering Research Council of Canada (NSERC).

Acknowledgements I would like to thank my collaborators: Wilson Cheang, Samuel Gélineau, Junyoung Jang, and Stefan Monnier.

1 Summary

Metaprogramming provides programmers with the ability to write programs that generate specialized and optimized code. This makes it possible to design and implement domain-specific program optimizations that complement general compiler optimizations yielding substantial performance gains. Unfortunately, designing language extensions to support writing type-safe meta-programs remains very challenging.

One widely used approach to metaprogramming going back to Lisp/Scheme is using quasiquotation which allows programmers to generate and compose code fragments. For example, the quasiquotation ⌈`2 + 2`⌉ is representing an abstract syntax tree (AST) of the expression `2 + 2`. We can embed and compose code fragments using unquote, written as ⌊ ⌋. Assuming that the function `square 2` generates code ⌈`2 * 2`⌉, the expression ⌈`2 +` ⌊`square 2`⌋⌉

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 2; pp. 2:1–2:3
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

evaluates to the code ⌈2 + 2 * 2⌉ where we splice in the generated code. There are two immediate questions that arise:

1. Can we express and reason statically about open code, i.e. code that may contain and refer to variables?
2. Can we analyze and further manipulate code via pattern matching?

A milestone in developing a logical foundation for characterizing code and reasoning about code generation is the work by Davies and Pfenning [2]. Davies and Pfenning distinguish between code and programs using the necessity modality. For example, the code ⌈2 + 2⌉ has the modal type ⌈int⌉, while the program square has type int → ⌈int⌉. This allows us to statically reason about different stages of computation. However, Pfenning and Davies' work has two limitations: first, it only allows us to generate and reason about closed code and second, it does not support analysis of code via pattern matching. Subsequent work by Nanevski, Pfenning and Pientka [3] suggests to characterize open code 2 + x together with the context x:int, ascribing the code ⌈x. 2 + x⌉ the contextual type ⌈x:int ⊢ int⌉ thereby removing the first restriction. Yet, a type-safe multi-staged metaprogramming foundation that supports both the generation of and pattern matching on open code remains elusive.

In this talk, we outline a modal type-theoretic foundation for polymorphic multi-staged metaprogramming that brings together the generation and the analysis of open code within the same framework. In particular, we draw on the theory and practice of contextual types and first-class contexts in the Beluga proof and programming environment [4, 7, 8, 1, 6, 5] and adapt two main ideas to the metaprogramming setting: first, we exploit contextual modal types to describe open polymorphic code together with the context in which it is meaningful; second, we model code as a higher-order abstract syntax (HOAS) tree within a context. These two ideas provide the appropriate abstractions for both generating and pattern matching on open code without committing to a concrete representation of variable binding and contexts which is left open to the implementor of the language.

Our work is a first step towards building a general type-theoretic foundation for multi-staged metaprogramming which enforces strong type guarantees and whose provided abstractions make it easy to generate and manipulate code. This will allow us to exploit the full potential of metaprogramming without sacrificing reliability of and trust in the code we are producing and running.

References

1 Andrew Cave and Brigitte Pientka. Programming with binders and indexed data-types. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 413–424. ACM Press, 2012.

2 Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001. doi:10.1145/382780.382785.

3 Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):1–49, 2008.

4 Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 371–382. ACM Press, 2008.

5 Brigitte Pientka, Andreas Abel, Francisco Ferreira, David Thibodeau, and Rebecca Zucchini. A type theory for defining logics and proofs. In *34th IEEE/ ACM Symposium on Logic in Computer Science (LICS'19)*, pages 1–13. IEEE Computer Society, 2019.

6 Brigitte Pientka and Andrew Cave. Inductive Beluga:Programming Proofs (System Description). In *25th International Conference on Automated Deduction (CADE-25)*, Lecture Notes in Computer Science (LNCS 9195), pages 272–281. Springer, 2015.

7 Brigitte Pientka and Joshua Dunfield. Programming with proofs and explicit contexts. In *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 163–173. ACM Press, 2008.

8 Brigitte Pientka and Joshua Dunfield. Beluga: a framework for programming and reasoning with deductive systems (System Description). In Jürgen Giesl and Reiner Haehnle, editors, *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI 6173), pages 15–21. Springer, 2010.

Quotients in Dependent Type Theory

Andrew M. Pitts
Department of Computer Science & Technology, University of Cambridge, UK

Abstract

Constructs that involve taking a quotient are commonplace in mathematics. Here I will consider how they are treated within dependent type theory. The notion of *quotient type* has its origins in the Nuprl theorem-proving system [4] for extensional type theory. Later Hofmann formulated a version for intensional type theory in his thesis [7]. This depends on having a pre-existing notion of intensional identity type. Hofmann used Martin-Löf's notion, the indexed family inductively generated from proofs of reflexivity [9, chapter 8]. The recent homotopical view of identity in terms of path types [10] gives a more liberal perspective and has brought with it the notion of *higher inductive type* (HIT) [8], subsuming both inductive and quotient types.

Inductively defined indexed families of types (in all their various forms) are perhaps the most useful concept that dependent type theory has contributed to the practice of computer assistance for formalizing mathematical proofs. However, it is often the case that a particular application of such types needs not only to inductively generate a collection of objects, but also to make identifications between the objects. In classical mathematics one can first generate and then identify, using the Axiom of Choice to lift infinitary constructions to the quotient. HITs can allow one to avoid such non-constructive uses of choice by inter-twining generation and identification. Perhaps more important than the constructive/non-constructive issue is that simultaneously declaring how to generate and how to identify can be a very natural way of defining some construct from the user's point of view. This is why HITs promise to be so useful, once we have robust and convenient mechanisms in theorem-proving systems for defining HITs and defining functions out of HITs. Although some HITs have been axiomatized in various systems, at the moment the only system I know of with built in support for defining quite general forms of HIT and using them is the implementation of cubical type theory [3] within recent versions of the Agda system [11].

The higher dimensional aspect of identity in cubical type theory is fascinating; nevertheless, the simpler one-dimensional version of identity, in which one has uniqueness of identity proofs (UIP), is adequate for many applications. Although some regard UIP as a bug of early versions of Agda with it's original form of dependent pattern matching [5], it is by choice a feature of the Lean prover [2]. Altenkirch and Kaposi [1] have termed the one-dimensional version of HITs *quotient inductive types* (QITs) and they promise to be very useful even in the setting of type theory with UIP.

In this talk I survey some of these developments, including a recent reduction of QITs to quotients [6], and the prospects for better support in theorem-provers for quotient constructions.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Constructive mathematics

Keywords and phrases dependent type theory, quotient, quotient inductive type, theorem-proving systems

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.3

Category Invited Talk

References

1 T. Altenkirch and A. Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 18–29, New York, NY, USA, 2016. ACM.

2 J. Avigad, L. de Moura, and S. Kong. *Theorem Proving in Lean*, 3.4.0 edition, February 2020.

3 C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In T. Uustalu, editor, *21st International Conference on*

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 3; pp. 3:1–3:2
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Types for Proofs and Programs (TYPES 2015), volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

4 R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Inc., USA, 1986.

5 T. Coquand. Pattern matching with dependent types. In B. Nordström, K. Petersson, and G. D. Plotkin, editors, *Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden*, pages 66–79, June 1992.

6 M. P. Fiore, A. M. Pitts, and S. C. Steenkamp. Constructing infinitary quotient-inductive types. In J. Goubault-Larrecq and B. König, editors, *23rd International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2020)*, volume 12077 of *Lecture Notes in Computer Science*, pages 257–276. Springer, 2020.

7 M. Hofmann. *Extensional Concepts in Intensional Type Theory.* PhD thesis, University of Edinburgh, 1995.

8 P. L. Lumsdaine and M. Shulman. Semantics of higher inductive types. *Mathematical Proceedings of the Cambridge Philosophical Society*, page 1–50, 2019.

9 B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory.* Oxford University Press, 1990.

10 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations for Mathematics.* `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

11 A. Vezzosi, A. Mörtberg, and A. Abel. Cubical agda: A dependently typed programming language with univalence and higher inductive types. *Proc. ACM Program. Lang.*, 3(ICFP):87:1–87:29, July 2019.

Certifying the Weighted Path Order

René Thiemann
University of Innsbruck, Austria

Jonas Schöpf
University of Innsbruck, Austria

Christian Sternagel
DVT, Innsbruck, Austria

Akihisa Yamada
National Institute of Advanced Industrial Science and Technology, Tokyo, Japan

Abstract

The weighted path order (WPO) unifies and extends several termination proving techniques that are known in term rewriting. Consequently, the first tool implementing WPO could prove termination of rewrite systems for which all previous tools failed. However, we should not blindly trust such results, since there might be problems with the implementation or the paper proof of WPO.

In this work, we increase the reliability of these automatically generated proofs. To this end, we first formally prove the properties of WPO in Isabelle/HOL, and then develop a verified algorithm to certify termination proofs that are generated by tools using WPO. We also include support for max-polynomial interpretations, an important ingredient in WPO. Here we establish a connection to an existing verified SMT solver. Moreover, we extend the termination tools NaTT and $\mathsf{T_TT_2}$, so that they can now generate certifiable WPO proofs.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Equational logic and rewriting

Keywords and phrases certification, Isabelle/HOL, reduction order, termination analysis

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.4

Category Invited Talk

Supplementary Material Details on the experiments are provided at `http://cl-informatik.uibk.ac.at/isafor/experiments/wpo/`. This website also provides links to the formalization.

Funding René Thiemann was supported by the Austrian Science Fund (FWF) project Y757. Jonas Schöpf and Christian Sternagel were supported by FWF project P27502. A part of Akihisa Yamada's contribution was made while he was working in University of Innsbruck under support by FWF projects Y757 and P27502.

Acknowledgements We thank Sarah Winkler for her assistance and discussions on IsaFoR, $\mathsf{T_TT_2}$ and WPO.

1 Introduction

Automatically proving termination of *term rewrite systems (TRSs)* has been an active field of research for half a century. A number of *simplification orders* [13] are classic methods for proving termination, while more general pairs of orders called *reduction pairs* play a central role in the more modern *dependency pair framework* [19].

The *weighted path order (WPO)* was first [51] introduced as a simplification order that unifies and extends classical ones, and then generalized to a reduction pair to further subsume more recent techniques [53]. The ***Na****goya* ***T****ermination* ***T****ool* (NaTT) [52] was originally developed solely to demonstrate the power of WPO. It participated in the full run of the

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 4; pp. 4:1–4:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

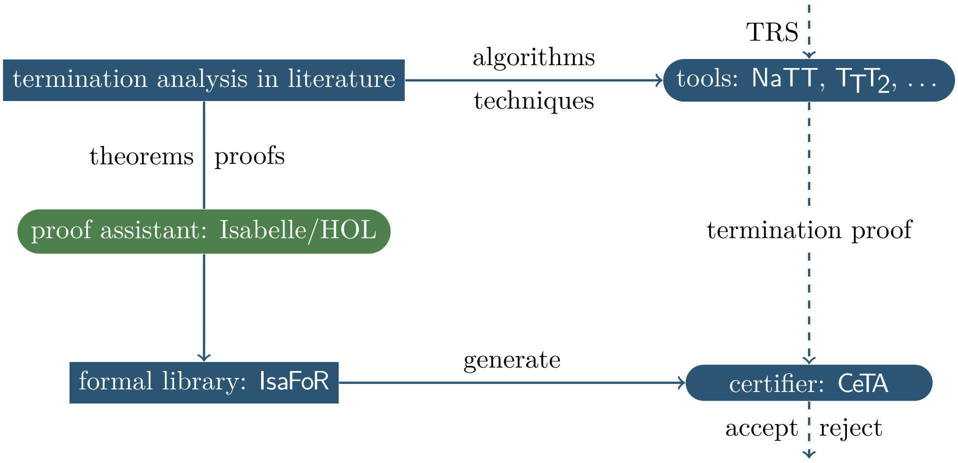


Figure 1 Procedure for Certification of Termination Proofs via IsaFoR/CeTA.

2013 edition of the Termination Competition [18] and won the second place, closing 34 of 159 then-open problems in the TRS Standard category. In 28 of them WPO was essential (the others are due to the efficiency of NaTT) [53].

Despite the significance of the result, two natural questions arise:

(1) "Is the theory of WPO correct?," and if yes

(2) "Is NaTT's implementation of the theory correct?".

So far, nobody investigated the 34 proofs found by NaTT; these benchmarks are obtained via automatic transformations from other systems, and hence hard to analyze by hand (they have up to a few hundred of rules). In this work, we answer the two questions.

To this end, we extend IsaFoR and CeTA [47]. The former, **Isa**belle **Fo**rmalization of **R**ewriting, is an Isabelle/HOL [35]-formalized library of correctness proofs of analysis techniques for term rewriting and transition systems, and the latter, **Ce**rtified **T**ool **A**ssertions, is a verified Haskell code generated from IsaFoR that takes machine-readable output from untrusted verifiers and checks whether techniques are applied correctly. This workflow is illustrated in Figure 1.

In this paper we describe two main extensions of IsaFoR and CeTA. After preliminaries we develop formal proofs of the properties of WPO being a reduction pair in Section 3. Here, we illustrate that one refinement of WPO provided in [53] breaks transitivity in a corner case, but we also show how to repair it by adding a mild precondition. Second, in Section 4 we formalize the max-polynomial interpretations that are used in [53] in a general manner. There we utilize our recently developed verified SMT solver for integer arithmetic [7, 8]. In Section 5 we give a short overview of our new certificate parser implementation in Isabelle/HOL and the format for certificates for WPO and max-polynomial interpretations. In Section 6, we experimentally evaluate our extensions of CeTA. To this end, we extend NaTT to be able to output certificates introduced in the preceding section, and we also integrate WPO in the **T**yrolean **T**ermination **T**ool **2** (TTT2) [27]. Details on the experiments are provided at:

`http://cl-informatik.uibk.ac.at/isafor/experiments/wpo/`

This website also provides links to the formalization.

Related Work

There is plenty of work on orders for proving termination of rewriting. The earliest such work we are aware of is Knuth and Bendix' order (KBO), introduced in their celebrated paper in 1970 along with the Knuth–Bendix completion [26]. In the same year, Manna and Ness [33] proposed a semantic approach, which nowadays is called *interpretation methods*. One instantiation of this approach is Lankford's *polynomial interpretation method* [30], which he also combined with KBO [31]. Dershowitz [14] initiated a purely syntactic approach called *recursive path orders (RPO)*, where he also discovered the notion of simplification orders.

The *dependency pair method* of Arts and Giesl [1] boosted the power of termination proving techniques, and around the same time many automated termination provers emerged: AProVE [17], TTT [21], CiME3 [10], Matchbox [49], muterm [32], TORPA [57], and so on, which have been evaluated annually in *the Termination Competition* [18] since 2004. Results of the competition regularly reveal that we cannot blindly trust such automated tools, when one tool claims a TRS terminating, while another claims the same TRS nonterminating.

Hence *certification* came into play. Besides our IsaFoR/CeTA, we are aware of at least two other systems for certifying termination proofs of TRSs: Coccinelle/CiME3 [11] and CoLoR/Rainbow [6]. Here, Coccinelle and CoLoR are similar to IsaFoR: they are all formal libraries on rewriting, though the former two are in Coq [5] instead of Isabelle. The choice of proof assistant aside, a significant difference to IsaFoR/CeTA is in the workflow when performing certification: CiME3 and Rainbow transform termination proofs into Coq files that reference their corresponding formal libraries, and then Coq does the final check, whereas in our case we just run the generated Haskell code CeTA outside of Isabelle.

Within IsaFoR, most closely related to the current work is the previous formalization [46] of RPO, since RPO and WPO are similar in its structure. We refer to Section 3 for more details on how we exploit this similarity.

We would also like to mention some related work outside of pure term rewriting. Recently a verified ordered resolution prover [36] has been developed as part of the IsaFoL project, the **Isa**belle **Fo**rmalization of **L**ogic. Currently the verified prover is based on KBO, which could be replaced by the stronger and more general WPO. In fact, WPO is already utilized in the E theorem prover [24].

In recent work [8] IsaFoR became capable of certifying termination proofs for *integer transition systems*. This work eventually led to a verified SMT solver for linear integer arithmetic [7], which we heavily reuse in our current work.

2 Preliminaries

2.1 Term Rewriting

We assume familiarity with term rewriting [2], but briefly recall notions that are used in the following. A *term* built from *signature* $\mathcal{F}$ and set $\mathcal{V}$ of variables is either $x \in \mathcal{V}$ or of form $f(t_1,\dots,t_n)$, where $f \in \mathcal{F}$ is n-ary and $t_1,\dots,t_n$ are terms. A *context* C is a term with one hole, and $C[t]$ is the term where the hole is replaced by t. The *subterm relation* $\trianglerighteq$ is defined by $C[t] \trianglerighteq t$. A *substitution* is a function σ from variables to terms, and we write $t\sigma$ for the *instance* of term t in which every variable x is replaced by $\sigma(x)$. A *term rewrite system (TRS)* is a set $\mathcal{R}$ of *rewrite rules*, which are pairs of terms ℓ and r indicating that an instance of ℓ in a term can be rewritten to the corresponding instance of r. $\mathcal{R}$ is *terminating* if no term can be rewritten infinitely often.

A *reduction pair* is a pair $(\succ, \succsim)$ of two relations on terms that satisfies the following requirements: $\succ$ is well-founded, $\succsim$ and $\succ$ are compatible (i.e., $\succsim \circ \succ \circ \succsim \subseteq \succ$), both are closed under substitutions, and $\succsim$ is closed under contexts. If $\succ$ is also closed under contexts, then we call $(\succ, \succsim)$ a *monotone* reduction pair. If the first component $\succ$ of a monotone reduction pair is transitive, it is called a *reduction order.* While reduction orders are used to directly prove termination by $\mathcal{R} \subseteq \succ$, reduction pairs are usually employed for termination proofs with dependency pairs. We write $\succ^{\mathsf{lex}}$ and $\succ^{\mathsf{mul}}$ for the lexicographic and multiset extension induced by $(\succ, \succsim)$, respectively.

A *weakly monotone ($\mathcal{F}$-)algebra* $\mathcal{A}$ is a well-founded ordered set $(A, >)$ equipped with an interpretation $f_{\mathcal{A}} : A^n \to A$ for every n-ary $f \in \mathcal{F}$, such that $f_{\mathcal{A}}(\dots, a, \dots) \geq f_{\mathcal{A}}(\dots, b, \dots)$ whenever $a \geq b$. Any weakly monotone algebra $\mathcal{A}$ induces a reduction pair $(>_{\mathcal{A}}, \geq_{\mathcal{A}})$ defined by $s \;{}_{(}{\geq}_{)\mathcal{A}}\; t$ iff $[\![s]\!]^{\alpha}_{\mathcal{A}} \;{}_{(}{\geq}_{)}\; [\![t]\!]^{\alpha}_{\mathcal{A}}$ for all assignments α. Here, $[\![t]\!]^{\alpha}_{\mathcal{A}}$ denotes term evaluation in the algebra with respect to an assignment $\alpha : \mathcal{V} \to A$.

A *(partial) status* is a mapping π which assigns to each n-ary symbol f a list $\pi(f) = [i_1, \dots, i_m]$ of indices in $\{1, \dots, n\}$. Abusing notation, we also use $\pi(f)$ as the set $\{i_1, \dots, i_m\}$, and as an operation on n-ary lists defined by $\pi(f)[t_1, \dots, t_n] = [t_{i_1}, \dots, t_{i_m}]$.

A binary relation $\succ$ over terms is *simple with respect to* status π, if $f(t_1, \dots, t_n) \succ t_i$ for all $i \in \pi(f)$. It is *simple*, if it is simple independent of the status. In particular, a simple reduction order is called a *simplification order.*

A *precedence* is a preorder $\succsim$ on $\mathcal{F}$, such that $\succ := \succsim \setminus \precsim$ is well-founded.

▶ **Definition 1** (WPO [53, Def. 10, incl. Refinements (2c) and (2d) of Sect. 4.2]). *Let $\mathcal{A}$ be a weakly monotone algebra, $\succsim$ a precedence, and π be a status. Let $\geq_{\mathcal{A}}$ be simple with respect to π. The WPO reduction pair $(\succ_{\mathsf{WPO}}, \succsim_{\mathsf{WPO}})$ is defined as follows: $s \succ_{\mathsf{WPO}} t$ iff*

1. *$s >_{\mathcal{A}} t$, or*
2. *$s \geq_{\mathcal{A}} t$ and*
   - **a.** *$s = f(s_1, \dots, s_n)$ and $\exists i \in \pi(f).\ s_i \succsim_{\mathsf{WPO}} t$, or*
   - **b.** *$s = f(s_1, \dots, s_n)$, $t = g(t_1, \dots, t_m)$, $\forall j \in \pi(g).\ s \succ_{\mathsf{WPO}} t_j$ and*
     - **i.** *$f \succ g$ or*
     - **ii.** *$f \succsim g$ and $\pi(f)[s_1, \dots, s_n] \succ^{\mathsf{lex}}_{\mathsf{WPO}} \pi(g)[t_1, \dots, t_m]$.*

The relation $s \succsim_{\mathsf{WPO}} t$ is defined in the same way, where $\succ^{\mathsf{lex}}_{\mathsf{WPO}}$ in the last line is replaced by $\succsim^{\mathsf{lex}}_{\mathsf{WPO}}$, and there are the following additional subcases in case **2***:*

- **c.** *$s \in \mathcal{V}$ and either $s = t$ or $t = g(t_1, \dots, t_m)$, $\pi(g) = \emptyset$ and g is least in precedence,*
- **d.** *$s = f(s_1, \dots, s_n)$, $t \in \mathcal{V}$, $>_{\mathcal{A}}$ is simple w.r.t. π, and $\forall g.\ f \succ g \vee (f \succsim g \wedge \pi(g) = \emptyset)$.*

▶ **Theorem 2** ([53]). *WPO forms a reduction pair.* ◀

For the certification purpose it suffices to formalize Theorem 2 and to provide a verified implementation to check WPO constraints of the form $s \;{}_{(}{\succsim}_{)}\; t$ for a concrete instance of WPO. In [53] it is further shown that a number of existing methods are obtained as instances of WPO, namely: the Knuth–Bendix order (KBO) [26], interpretation methods [15, 30], polynomial KBO [31], lexicographic path orders (LPO) [25], and non-collapsing argument filters [1, 29]. This means that, by having a WPO certifier, one can also certify these existing methods.

2.2 Isabelle/HOL and IsaFoR

We do not assume familiarity with Isabelle/HOL, since most of the illustrated formal statements are close to mathematical text. We give some brief explanations by illustrating certain term rewriting concepts via their counterparts in IsaFoR. For instance, IsaFoR contains a datatype for terms, ('f,'v)term, where 'f and 'v are type-variables representing the signature $\mathcal{F}$ and the set of variables $\mathcal{V}$, respectively. A typing judgment is of the form *term* :: *type*. As an example, R :: ('f,'v)term rel states that R has type ('f,'v)term rel, i.e., R is a binary relation over terms.

An Isabelle *locale* [3] is a named context where certain elements can be fixed and properties can be assumed. Locales are frequently used in IsaFoR. For instance, reduction pairs in IsaFoR are formulated as a locale redpair.[1] Here, O is relation composition, and SN is a predicate for well-foundedness (strong normalization).

```
locale redpair =
  fixes S NS :: "('f,'v)term rel"
  assumes "SN S"
    and "ctxt.closed NS"
    and "subst.closed S" and "subst.closed NS"
    and "NS O S ⊆ S" and "S O NS ⊆ S"
```

Locales are also useful to model hierarchical structures. For instance, whereas redpair does not require that the relations are orders, this is required in the upcoming locale redpair_order which is an extension of redpair.

```
locale redpair_order = redpair S NS +
  assumes "trans S" and "trans NS" and "refl NS"
```

Beside the abstract definitions for reduction pairs, IsaFoR also provides several instances of them, e.g., one for RPO, one for KBO [40], etc. These instances can then be used in termination techniques like the reduction pair processor to validate concrete termination proofs. However, often the requirements of a reduction pair are not yet enough. As an example, the usable rules refinement [20, 48] requires $\mathcal{C}_e$-compatible reduction pairs and argument filters. To this end IsaFoR contains the locale ce_af_redpair_order. It extends redpair_order by a new parameter π for the argument filter, and demands the additional requirements.

```
locale ce_af_redpair_order = redpair_order S NS +
  fixes π :: "'f af"
  assumes "af_compatible π NS"
    and "ce_compatible NS"
```

There are further locales for monotone reduction pairs, for reduction pairs which can be used in complexity proofs, etc.

3 Formalization of WPO

In this section we present our formalization of WPO. It starts by formalizing the properties of WPO in Section 3.1, so that we can add WPO as a new instance of a reduction pair to IsaFoR. Afterwards we illustrate our verified implementation for checking WPO constraints in Section 3.2.

[1] In IsaFoR, there is a more general locale for reduction *triples* (redtriple), which we simplify to reduction *pairs* in the presentation of this paper.

3.1 Properties of WPO

As we have seen in Section 2.2, IsaFoR already contains several formalized results about reduction pairs, including general results, instances, and termination techniques based on reduction pairs. In contrast, at the start of this formalization of WPO, IsaFoR did not contain a single locale about generic weakly monotone algebras. In particular, the formalization of matrix interpretations and polynomial interpretations [42] directly refers to redpair and its variants. So, the question arises, how the generic version of WPO in Definition 1 can be formalized, which is based on arbitrary weakly monotone algebras.

The obvious approach would have been to just add the missing pieces. To be more precise, we could have formalized weakly monotone algebras in IsaFoR and then on top have formally verified the properties of WPO. However, this approach has the disadvantage that we would have had to adjust also existing instances of weakly monotone algebras (like polynomial interpretations, arctic interpretations, and matrix interpretations) to the new interface.

Therefore, we choose a different approach, namely to reformulate the definition of WPO such that it does no longer depend on the notion of weakly monotone algebra, but instead directly refers to reduction pairs (cf. Definition 3).

▶ **Definition 3** (WPO based on Reduction Pairs). *Let $(>_{\mathcal{A}}, \geq_{\mathcal{A}})$ be a reduction pair, $\succsim$ a precedence, ... and continue as in Definition 1 to define the relations $\succ_{\mathsf{WPO}}$ and $\succsim_{\mathsf{WPO}}$.*

In this way, all instances of reduction pairs in IsaFoR immediately become available as parameters to WPO. On the one hand, we can parameterize WPO with (max-)polynomial interpretations and matrix interpretations as is already done in the literature. On the other hand, it is also possible to use KBO or RPO as parameter to WPO, or even to nest WPOs recursively.

Of course the question is, how easy it is to formally prove properties of this WPO based on reduction pairs. At this point we profit from the fact that the structure of WPO is quite close to other path orders like RPO, and that the latter has already been fully formalized in IsaFoR.

▶ **Definition 4** (RPO as formalized in IsaFoR). *Let $\succsim$ be a precedence and σ be a function of type $\mathcal{F} \to \{\mathsf{lex}, \mathsf{mul}\}$. We define the RPO reduction pair $(\succ_{\mathsf{RPO}}, \succsim_{\mathsf{RPO}})$ as follows: $s \succ_{\mathsf{RPO}} t$ iff*

- **a.** *$s = f(s_1, \ldots, s_n)$ and $\exists i \in \{1, \ldots, n\}.\ s_i \succsim_{\mathsf{RPO}} t$, or*
- **b.** *$s = f(s_1, \ldots, s_n)$, $t = g(t_1, \ldots, t_m)$, $\forall j \in \{1, \ldots, m\}.\ s \succ_{\mathsf{RPO}} t_j$ and*
  - **i.** *$f \succ g$ or*
  - **ii.** *$f \succsim g$ and $\sigma(f) = \sigma(g)$ and $[s_1, \ldots, s_n] \succ_{\mathsf{RPO}}^{\sigma(f)} [t_1, \ldots, t_m]$.*
  - **iii.** *$f \succsim g$ and $\sigma(f) \neq \sigma(g)$ and $n > 0$ and $m = 0$.*

The relation $s \succsim_{\mathsf{RPO}} t$ is defined in the same way, where $\succ_{\mathsf{RPO}}^{\sigma(f)}$ in case bii is replaced by $\succsim_{\mathsf{RPO}}^{\sigma(f)}$, the condition $n > 0$ in case biii is dropped, and there is one additional subcase:

- **c.** *$s \in \mathcal{V}$ and either $s = t$ or $t = c$ where c is a constant in $\mathcal{F}$ that is least in precedence.*

So, we start our formalization of WPO by copy-and-pasting the definitions and proofs about RPO, and renaming every occurrence of "RPO" to "WPO." At this point we have a fully verifiable Isabelle theory which defines WPO as a copy of RPO.

Next, we modify a couple of definitions, such that eventually, we arrive at a formalized variant of the WPO in Definition 3. For each modification, we immediately adjust the formal proofs. Such adjustments are mostly straight-forward, not least due to the valuable support by the proof assistant: we are immediately pointed to those parts of proofs which are broken by a modification, without having to manually recheck the remaining proofs that were not affected.

To be more precise, we perform the following sequence of modifications.

- We delete σ from RPO and replace it by lex, as the choice between multiset and lexicographic comparison via σ is not present in WPO. As a result, case biii is dropped, case bii always uses lexicographic comparison, and the formal proofs become shorter.
- We add the two tests $s \geq_{\mathcal{A}} t$ and $s >_{\mathcal{A}} t$ that are present in WPO, but not in RPO. Moreover, we add the requirement of WPO, that $\geq_{\mathcal{A}}$ must be simple, in order to adjust all the proofs of the defined relations.
- We include the status π, which is present in the WPO definition, but not in RPO. In this step we also weaken the requirement of $\geq_{\mathcal{A}}$ being simple to the requirement that $\geq_{\mathcal{A}}$ is simple with respect to π.
- We generalize case c of RPO in such a way that not only for constants c we permit $x \succsim_{\mathsf{WPO}} c$, but also $x \succsim_{\mathsf{WPO}} g(t_1, \ldots, t_n)$ is possible if $\pi(g) = \emptyset$.
- We finally add refinement 2d under the premise that $>_{\mathcal{A}}$ is simple with respect to π. At this point we have precisely a formalized version of WPO as defined in Definition 3.

Interestingly, after the final refinement we were no longer able to show all properties of $(\succ_{\mathsf{WPO}}, \succsim_{\mathsf{WPO}})$. For example, the transitivity proof of $\succsim_{\mathsf{WPO}}$ was broken and we were not able to repair it. Indeed, it turns out that $\succsim_{\mathsf{WPO}}$ is no longer transitive with the refinement as illustrated by Example 5. This example was constructed with the help of Isabelle, since it directly pointed us to the case where the transitivity proof got broken.

▶ **Example 5.** Consider $\mathcal{F} = \{\mathsf{a}\}$, $\pi(\mathsf{a}) = []$, and a reduction pair (or algebra) where $\geq_{\mathcal{A}}$ relates all terms and $>_{\mathcal{A}}$ is empty. Then $x \succsim_{\mathsf{WPO}} \mathsf{a} \succsim_{\mathsf{WPO}} y$, but $x \succsim_{\mathsf{WPO}} y$ does not hold.

The reduction pair (or algebra) in Example 5 is obviously a degenerate case. In fact, by excluding this degenerate case, we can formally prove that WPO including refinement 2d is a reduction pair.

To this end, we gather all parameters of WPO in a locale and assume relevant properties of these parameters, either via other locales or as explicit assumptions. The precedence $\succsim$ is specified by way of three functions prc, pr_least, and pr_large: prc takes two symbols f and g and returns a pair of Booleans $(f \succ g, f \succsim g)$; pr_least is a predicate telling whether a symbol is least in $\succsim$ or not; and pr_large states whether a symbol is largest in $\succsim$ with respect to π or not, as required in rule 2d of Definition 1. Whereas most of the properties of the precedence are encoded via an existing locale precedence, for a symbol being of *largest* precedence we add two new assumptions explicitly. In the locale we further use a Boolean ssimple to indicate whether $>_{\mathcal{A}}$ is simple with respect to π, i.e., whether it is allowed to apply rule 2d or not. Only then, the properties of pr_large must be satisfied and the degenerate case must be excluded. Being simple with respect to π is enforced via the predicate simple_arg_pos: for any relation R the property simple_arg_pos R f i ensures that $f(t_1, \ldots, t_n)$ R t_i holds for all $t_1, \ldots, t_n$.

```
locale wpo_params = redpair_order S NS + precedence prc pr_least
  for S NS :: “('f, 'v) term rel”                          (* underlying reduction pair *)
    and prc :: “'f ⇒ 'f ⇒ bool × bool” and pr_least pr_large :: “'f ⇒ bool”(* precedence *)
    and ssimple :: bool                         (* flag whether rule (2d) is permitted *)
    and π :: “'f status” +                                                   (* status *)
  assumes “S ⊆ NS”
    and “i ∈ π f ⟹ simple_arg_pos NS f i”                       (* NS is simple w.r.t. π *)
    and “ssimple ⟹ i ∈ π f ⟹ simple_arg_pos S f i”               (* S is simple w.r.t. π *)
    and “ssimple ⟹ NS ≠ UNIV”                                  (* exclude degenerate case *)
    and “ssimple ⟹ pr_large f ⟹ fst (prc f g) ∨ snd (prc f g) ∧ π g = []”
    and “ssimple ⟹ pr_large f ⟹ snd (prc g f) ⟹ pr_large g”
```

Within the locale we define the relations WPO_S and WPO_NS ($\succ_{\mathsf{WPO}}$ and $\succsim_{\mathsf{WPO}}$ of Definition 3) with the help of a recursive function, and prove the main theorem:

```
theorem "redpair_order WPO_S WPO_NS"
```

Moreover, we prove that whenever the non-strict relation is compatible with an argument filter μ then also the WPO is compatible with $\pi \cup \mu$, defined as $(\pi \cup \mu)(f) = \pi(f) \cup \mu(f)$.

```
lemma assumes "af_compatible μ NS"
  shows "af_compatible (π ∪ μ) WPO_NS"
```

We further prove that WPO is also $\mathcal{C}_e$-compatible under mild preconditions, namely whenever $\pi(f)$ includes the first two positions of some symbol f. Finally, we formalize that WPO can be used in combination with usable rules, since it is an instance of the corresponding locale:

```
lemma assumes "∃f. {0,1} ⊆ π f"           (* positions in IsaFoR start from 0 *)
  and "af_compatible μ NS"
  shows "ce_af_redpair_order WPO_S WPO_NS (π ∪ μ)"
```

At the moment, our formalization does not cover any comparison to other term orders. There is, for example, no formal statement that each polynomial KBO can be formulated as an instance of WPO. The simple reason is that such a formalization will not increase the power of the certifier, and the support for polynomial KBO can much easier be added by just translating an instance of polynomial KBO into a corresponding WPO within a certificate, e.g., when generating certificates in a termination tool or when parsing certificates in CeTA.

3.2 Checking WPO Constraints

Recall that our formalization of WPO in Section 3.1 has largely been developed by adjusting the existing formal proofs for RPO. When implementing an executable function to check constraints of a particular WPO instance, where precedence, status, etc. are provided, there is however one fundamental difference to RPO: in WPO we need several tests $s >_{\mathcal{A}} t$ and $s \geq_{\mathcal{A}} t$ of the underlying reduction pair. And in general, these tests are just *approximations*, e.g., since testing positiveness of non-linear polynomials is undecidable.

In order to cover approximations, the implementations of reduction pairs in IsaFoR adhere to the following interface, which is a record named redpair that contains five components:

- One component is for checking validity of the input. For instance, for polynomial interpretations here one would check that each interpretation of an n-ary function symbol is a polynomial which only uses variables $x_1, \ldots, x_n$.
- There are two functions check_S and check_NS of type ('f,'v)term ⇒ ('f,'v)term ⇒ bool for approximating whether two terms are strictly and weakly oriented, respectively.
- There is a flag mono which indicates whether the reduction pair is monotone. An enabled mono-flag is required for checking termination proofs without dependency pairs.
- The implicit argument filter of the reduction pair can be queried, a feature that is essential for usable rules.

The generic interface is instantiated by all reduction pair (approximations) in IsaFoR, and they satisfy the common soundness property, that for a given approximation of a reduction pair rp and for given finite sets of strict- and non-strict-constraints, represented as two lists S_list and NS_list, there exists a corresponding reduction pair that orients all constraints in

S_list strictly and in NS_list weakly. In order to simplify the presentation, in the upcoming formal sources we uniformly use set comprehensions instead of list comprehensions, and we omit all conversions between lists and sets.

```
assumes "redpair.valid rp"                          (* generic_reduction_pair *)
  and "∀ (s,t) ∈ S_list. redpair.check_S rp s t"
  and "∀ (s,t) ∈ NS_list. redpair.check_NS rp s t"
shows "∃ S NS.
     ce_af_redpair_order S NS (redpair.af rp) ∧
     S_list ⊆ S ∧ NS_list ⊆ NS ∧
     (redpair.mono rp ⟶ ctxt.closed S)"
```

We next explain how to instantiate this interface by WPO. To be more precise, we are given a status π, a precedence, and an approximated reduction pair rp and have to implement the interface for WPO such that generic_reduction_pair is satisfied.

For checking validity of WPO, we assert redpair.valid rp and in addition perform checks that the status π is well-defined, i.e., $\pi(f) \subseteq \{1, \ldots, n\}$ must hold for each n-ary symbol f. Moreover, we globally compute symbols of largest and least precedence, i.e., the functions pr_least and pr_large of the wpo_params-locale. We further set the argument filter of WPO to $\pi \cup$ redpair.af af.

For determining the ssimple parameter of the wpo_params-locale, there is the problem, that we do not know whether the generated strict relation S will be simple with respect to π. Moreover, to instantiate the locale, we always must ensure that NS is simple with respect to π. Unfortunately, the formal statement of generic_reduction_pair does not include any such information.

We solve this problem by enlarging the record redpair by two new entries for strict and weak simplicity, and require in generic_reduction_pair that if these flags are enabled, then the relations S and NS must be simple with respect to π, respectively. Whereas now all required information for WPO is accessible via the interface, the change of the interface requires to adapt all existing reduction pairs in IsaFoR, like polynomial interpretations, to provide the new information. To be more precise, we formalize two sufficient criteria for each reduction pair in IsaFoR, that ensure simplicity of the weak and strict relation, respectively.

At this point all parameters of WPO are fixed, except for S and NS. We now define the approximation of WPO as the WPO where S and NS are replaced by redpair.check_S rp and redpair.check_NS rp, respectively.

Next, we are given two lists of constraints wpo_S_list and wpo_NS_list that are oriented by the approximation of WPO. Out of these we extract the lists S_list and NS_list that contain all invocations of the underlying approximated reduction pair rp within the recursive definition of WPO, for instance:

$$\mathsf{S\_list} = \{(s_i, t_i) \mid (s, t) \in \mathsf{wpo\_S\_list} \cup \mathsf{wpo\_NS\_list}, s \trianglerighteq s_i, t \trianglerighteq t_i, \mathsf{redpair.check\_S\ rp}\ s_i\ t_i\}$$

After these lists have been defined, we apply generic_reduction_pair to get access to the (non-approximated) reduction pair in the form of relations S and NS. With these we are able to instantiate the wpo_params-locale and get access to the reduction pair WPO_S and WPO_NS. We further know that the approximations in S_list and NS_list are correct, e.g., whenever $(s, t) \in$ wpo_S_list $\cup$ wpo_NS_list, $s \trianglerighteq s_i$, $t \trianglerighteq t_i$ and redpair.check_S rp s_i t_i then $(s_i, t_i) \in$ S. With this auxiliary statement we finally prove that the approximated WPO corresponds to the actual WPO for all constraints in wpo_S_list $\cup$ wpo_NS_list. So, we have a reduction pair WPO_S and WPO_NS and an approximation statement, as required by generic_reduction_pair.

In total, we get an interpretation of the generic interface for WPO, and thus can use WPO in every termination technique of IsaFoR which is based on reduction pairs.

4 Integration of Max-Polynomial Interpretation

As already mentioned in the previous section, various kinds of interpretation methods have been formalized in IsaFoR and supported by CeTA. However, max-polynomial interpretations [16] were not yet supported. Hence we extend IsaFoR and CeTA to incorporate them, in particular those over natural numbers as required by WPO instances introduced in [53].

In order for CeTA to certify proofs using max-polynomial interpretations, we must formally prove that the pair of relations $(>_{\mathcal{A}}, \geq_{\mathcal{A}})$ forms a reduction pair, and implement a verifier to check $s >_{\mathcal{A}} t$ and $s \geq_{\mathcal{A}} t$. The former is easy, it is clearly weakly monotone and well-founded. For a verified comparison of max-polynomials, instead of implementing a dedicated checker from scratch, we chose to reduce the comparison of max-polynomials to the validity of an integer arithmetic formula without max, for which we already have a formalized validity checker [7, 8]. This checker is essentially an SMT-solver for linear integer arithmetic that we utilize to ensure unsatisfiability of negated formulas.

We formalize max-polynomials in IsaFoR as terms over the following signature.

```
datatype sig = ConstF nat | SumF | ProdF | MaxF
```

The interpretation of these symbols is as expected:

```
primrec I where
  "I (ConstF n) = (λx. n)"
| "I SumF = sum_list"
| "I ProdF = prod_list"
| "I MaxF = max_list"
```

In order to compare max-polynomials, we first normalize them according to the following four distribution rules:

$$\max(x,y) + z \to \max(x+z, y+z) \qquad x + \max(y,z) \to \max(x+y, x+z)$$
$$\max(x,y) \cdot z \to \max(x \cdot z, y \cdot z) \qquad x \cdot \max(y,z) \to \max(x \cdot y, x \cdot z)$$

Note that the distribution of multiplication over max is admissible because we are only considering natural numbers. This way, the max-polynomials s and t are normalized to $\max_{i=1}^{n} s_i$ and $\max_{i=1}^{m} t_i$, where $s_1, \ldots, s_n$ and $t_1, \ldots, t_m$ are polynomials (without max). In IsaFoR we define the mapping from s to $s_1, \ldots, s_n$ as to_IA. Then the comparison of two such normal forms is easily translated to an arithmetic formula without max [4]:

$$s \;_{(}\!\leq_{)}\; t \iff \max_{i=1}^{n} s_i \;_{(}\!\leq_{)}\; \max_{j=1}^{m} t_j \iff \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m} s_i \;_{(}\!\leq_{)}\; t_j$$

This reduction is formalized in Isabelle as follows. Here, operators with subscript "$_{\mathsf{f}}$" build syntactic formulas, and those with prefix "IA." or subscript "$_{\mathsf{IA}}$" come from the formalization of integer arithmetic; e.g., "$\bigwedge_{\mathsf{f}}$ x ← xs. IA.const 0 $\leq_{\mathsf{IA}}$ IA.var x" denotes an integer arithmetic formula representing "$0 \leq x_1 \wedge \cdots \wedge 0 \leq x_n$", where xs $= [x_1, \ldots, x_n]$. Since we are originally concerned about natural numbers, in the following definitions we insert such assumptions for the list of variables occurring in s and t. Initially we did not impose these assumptions and consequently, several valid termination proofs could not be certified. We thank Sarah Winkler for spotting and fixing this omission.

definition le_via_IA **where** "le_via_IA s t ≡
($\bigwedge_f$ x ← vars_term_list s @ vars_term_list t. IA.const 0 $\le_{IA}$ IA.var x) $\longrightarrow_f$
($\bigwedge_f$ s_i ← to_IA s. $\bigvee_f$ t_j ← to_IA t. $s_i \le_{IA} t_j$)"

definition less_via_IA **where** "less_via_IA s t ≡
($\bigwedge_f$ x ← vars_term_list s @ vars_term_list t. IA.const 0 $\le_{IA}$ IA.var x) $\longrightarrow_f$
($\bigwedge_f$ s_i ← to_IA s. $\bigvee_f$ t_j ← to_IA t. $s_i <_{IA} t_j$)"

The soundness of the reduction is formally proved as follows.

lemma le_via_IA:
assumes "$\models_{IA}$ le_via_IA s t" **shows** "s $\le_{\mathcal{A}}$ t"

lemma less_via_IA:
assumes "$\models_{IA}$ less_via_IA s t" **shows** "s $<_{\mathcal{A}}$ t"

Because of lemmas le_via_IA and less_via_IA it is now possible to invoke the validity checker for integer arithmetic on the formulas le_via_IA t s and less_via_IA t s in order to soundly validate the comparisons s $\ge_{\mathcal{A}}$ t and s $>_{\mathcal{A}}$ t, respectively.

Finally all results are put together to form an instance of an generic_reduction_pair of Section 3.2, namely a verified implementation for max-polynomial interpretations.

5 Certificate Format and Parser

The *Certification Problem Format (CPF)* [41] is a machine-readable XML format, which was codeveloped by several research groups of the term rewriting community to serve as the standard communication language between automated provers and certifiers.

Here we present the additions to CPF that are part of the current work, i.e., the certificate format for WPO and max-polynomial interpretations. Moreover, we comment on our complete overhaul of CeTA's certificate parser. Before this overhaul, the certificate parser was built on top of an Isabelle/HOL formalized XML transformer library [43] that has several limitations. In this context an XML transformer is a parser that consumes an XML element and produces results represented by arbitrary (custom) data types. (In the remainder, we will use "parser" and "transformer" synonymously.) In the current work we develop a more concise and flexible XML transformer library, which allows for syntax similar to Haskell's do-notation.

In our formalization, the type of XML transformers is 'a xmlt2, which is a function that takes the internal representation of an XML element and returns, in form of direct sums, either the result of a successful parse (type 'a) or an error state.

The notation "XMLdo s $\{\ldots\}$" yields a transformer for an XML element whose root tag is s. Within an XMLdo block, we can parse child elements by the binding "x ← $inner$;" or one of its variants such as "xs $\leftarrow$^$\{\ell..u\}$ $inner$;" which binds xs to the list of values resulting from transforming at least ℓ and at most u inner elements using the XML transformer *inner*. Here u is of type enat (extended naturals), so that it can be ∞. The frequent instance $\leftarrow$^$\{0..\infty\}$ is also written $\leftarrow *$. Typically a parser block should end with "xml_return r", where r is a return value that may rely on previously bound variables. This invocation also checks that after parsing, there are no child elements left in the current XML element. This ensures that the transformer defines a grammar.

Given two parsers p_1 and p_2, we allow a choice between them by "p_1 XMLor p_2". This works as follows: if parser p_1 returns a *recoverable* error state, then we continue with p_2. Otherwise, the result of "p_1 XMLor p_2" is the result of p_1. Here recoverable means that the tag of the root element is not consumed by p_1. If p_1 consumed the root node but failed for some child element, then it yields an unrecoverable error state containing an appropriate error message.

In the following we illustrate our approach by some parsers from our formalization. The new notation should make it fairly easy to translate between such parsers and corresponding specifications for the CPF format. Until a certain moment in the development we stated all parsers using Isabelle's **function** command, which specifies a recursive function along with a proof that the function is totally defined. For humans, proving XML transformers well-defined is rather easy, only requiring a dedicated measure for the internal XML representations. For Isabelle, however, accepting the proofs turned out to be excessively slow. Especially for the big parser that covers the entire CPF. Therefore, we now define our transformers via the **partial_function** [28] command, which does not perform such proofs and therefore is much faster.

A first concrete example is a parser for expressions occurring in max-polynomial interpretations. Here notions defined in Section 4 are accessed via prefix "max_poly.", and (STR "...") is the notation for target-language strings in Isabelle/HOL.

```
partial_function (sum_bot) exp_parser :: "(max_poly.sig, nat) term xmlt2" where
  [code]: "exp_parser xml = (
  XMLdo (STR "product") {
    exps ←∗ exp_parser; xml_return (Fun max_poly.ProdF exps)
  } XMLor XMLdo (STR "sum") {
    exps ←∗ exp_parser; xml_return (Fun max_poly.SumF exps)
  } XMLor XMLdo (STR "max") {
    exps ←^{1..∞} exp_parser; xml_return (Fun max_poly.MaxF exps)
  } XMLor XMLdo (STR "constant") {
    n ←nat; xml_return (max_poly.const n)
  } XMLor XMLdo (STR "variable") {
    n ←nat; xml_return (Var (n − 1))
  }) xml"
```

The parser recursively defines the grammar of max-polynomial expressions (as a *complex type* in XML schema terminology). It is a choice among the elements `<product>`, `<sum>`, `<max>`, `<constant>` and `<variable>`. Elements `<product>` and `<sum>` recursively contain an arbitrary number of subexpressions and construct corresponding terms over signature max_poly.sig. Element `<max>` is similar, except that it demands at least one subexpression. Element `<constant>` contains just a natural number, which is parsed as a constant. Element `<variable>` also contains a natural number, which indicates the i-th variable (using zero-based indexing).

The extended format for reduction pairs (triples) is as follows:

```
partial_function (sum_bot) redtriple :: "'a redtriple_impl xmlt2" where
  [code]: "redtriple xml = ( ...                    (∗ existing reduction pairs ∗)
    XMLor XMLdo (STR "maxPoly") {             (∗ max−polynomial interpretations ∗)
      inters ←∗ XMLdo (STR "interpret") {
        f ← xml2name;
        a ← XMLdo (STR "arity") { a ←nat; xml_return a };
        e ← exp_parser;
        xml_return ((f, a), e)
      };
      xml_return (Max_poly inters)
    } XMLor XMLdo (STR ''weightedPathOrder'') {        (* new alternative for WPO *)
      a ← wpo_params;
      b ← redtriple;
      xml_return (WPO a b)
    }
    XMLor XMLdo (STR ''filteredRedPair'') {...}       (* collapsing argument filter *)
  ) xml''
```

It is extended from the previous reduction pairs with three new alternatives. Element `<maxPoly>` is the reduction pair induced by max-polynomial interpretations, which is a list of elements `<interpret>`, each assigning a function symbol f of arity a its interpretation as expression e. The `<weightedPathOrder>` element characterizes a concrete WPO reduction pair. It consists of WPO specific parameters wpo_params that fixes status and precedences, and another reduction pair in a recursive manner, which specifies the "algebra" $\mathcal{A}$ in terms of $(>_{\mathcal{A}}, \geq_{\mathcal{A}})$. The `<filteredRedPair>` element is newly added specially for *collapsing* argument filters. Since partial status subsumes non-collapsing argument filters [50], only dedicated collapsing ones have to be specially supported.

6 Implementations and Experiments

In order to evaluate the relevance of our extension of CeTA by WPO and max-polynomial interpretations, we implement certificate output for WPO in two termination analyzers: NaTT and $\mathsf{T_TT_2}$.

NaTT. originates as an experimental implementation of WPO [51]. From its early design NaTT followed the trend [54, 55, 37, 9] of reducing termination problems into SMT problems and employ an external SMT solver, by default, Z3 [12]. Further, NaTT utilizes incremental SMT solving, and implements some tricks for efficiency [52]. In the current work, its output is adjusted to conform to the newly defined XML certificate format for WPO, max-polynomials, and collapsing argument filters. These are essentially the central techniques implemented in NaTT, but a few techniques implemented later on in NaTT had to be deactivated to be able to be certified by CeTA; some of them, such as nontermination proofs, are actually supported but NaTT is not yet adjusted to produce certificates for them.

$\mathsf{T_TT_2}$. succeeded the automated termination analyzer $\mathsf{T_TT}$ in 2007. It implements numerous (non-)termination techniques. For searching reduction pairs it uses a SAT/SMT-based approach and the SMT solver MiniSMT [56]. We extend $\mathsf{T_TT_2}$ by an implementation of WPO, following mostly the presented encodings in [53]. A notable difference in the search space for max-polynomials: while NaTT heuristically chooses between max and sum, $\mathsf{T_TT_2}$ embeds this choice into the SMT encoding.

Besides the integration of the full WPO search engine, we would also like to mention an additional feature of $\mathsf{T_TT_2}$ regarding WPO. Usual termination tools just try to find *any* proof. Even if users want a specific shape of proofs, they cannot impose constraints on proofs that termination tools find. $\mathsf{T_TT_2}$ provides *termination templates* [38] where users can fix parts of proofs via parameters when invoking $\mathsf{T_TT_2}$. We also added support for termination templates for WPO, i.e., if one wants to find a specific proof with WPO then (some) values can be fixed with $\mathsf{T_TT_2}$ and afterwards CeTA can validate if the proof is correct.

▶ **Example 6.** Consider the following TRS (`Zantema_05/z10.xml` of TPDB):

$$\begin{array}{rlrl}
\mathsf{a}(\mathsf{lambda}(x), y) \to \mathsf{lambda}(\mathsf{a}(x, \mathsf{p}(\mathsf{1}, \mathsf{a}(y, \mathsf{t})))) & & \mathsf{a}(\mathsf{a}(x, y), z) \to \mathsf{a}(x, \mathsf{a}(y, z)) & \\
\mathsf{a}(\mathsf{p}(x, y), z) \to \mathsf{p}(\mathsf{a}(x, z), \mathsf{a}(y, z)) & & \mathsf{a}(\mathsf{id}, x) \to x & \\
\mathsf{a}(\mathsf{1}, \mathsf{id}) \to \mathsf{1} & & \mathsf{a}(\mathsf{t}, \mathsf{id}) \to \mathsf{t} & \\
\mathsf{a}(\mathsf{1}, \mathsf{p}(x, y)) \to x & & \mathsf{a}(\mathsf{t}, \mathsf{p}(x, y)) \to y &
\end{array}$$

If we just call TTT2 with WPO (☑[2]) on this TRS then we get a termination proof consisting of arbitrary values. However, e.g., we might want a specific WPO proof with the precedence $\mathsf{id} > \mathsf{a} > \mathsf{lambda} > \mathsf{t} > \mathsf{1} > \mathsf{p}$ and a status reversing the arguments of p for the lexicographic comparison. For this we can use the following call (☑):

```
./ttt2 -s "wpo -msum -st \"p = [1;0]\" -prec \"id > a > lambda > t > 1 >
                          p\"" Zantema_05/z10.xml
```

The flag `-msum` activates $\mathcal{MS}um$ (from [53]) as interpretation for WPO, the flag `-st` fixes statuses and the flag `-prec` fixes a (part of a) precedence. Also all other WPO parameters, for the standard instances of [53], can be fixed via flags. In order to be sure that the proof is correct we can call CeTA on the certificate.

As a result we obtain a proof with the stated preconditions and in a broader sense TTT2 can be used to find specific WPO proofs. For some applications, it even makes sense to fix all parameters of WPO, so that there is no search at all. This option is useful for validating WPO-based termination proofs in papers, since writing XML-files in CPF by hand is tedious, but it is easy to invoke TTT2 on an ASCII representation of both the TRS and the WPO parameters. Then one automatically gets the corresponding proof in XML so that validation by CeTA is possible afterwards.

Evaluation. We now evaluate CeTA over the certifiable proofs generated by NaTT and TTT2. Experiments are run on StarExec [45], a computation resource service for evaluating logic solvers and program analyzers. The environment offers an Intel® Xeon® CPU E5-2609 running at 2.40GHz and 128GB main memory for each pair of a solver and problem. We set 300s timeout for each pair, as in the Termination Competition 2019.

We compare six configurations: NaTT, TTT2 without WPO and with WPO, and their variants that restrict to certifiable techniques. The results are summarized in Table 1. We remark that all the proofs generated by certifiable configurations are successfully certified by CeTA. Most notably, the termination proofs for the 34 examples mentioned in the introduction that reportedly only NaTT could prove terminating are verified.

The impact of WPO in TTT2, unfortunately, appears marginal. It only brings two additional termination proofs in the certifiable setting; and in the other setting the small difference would vanish if one slightly modifies the timeout. It is most likely that the proof search heuristic of TTT2 is not optimal, and more engineering effort is necessary in order to maximize the effect of WPO for TTT2.

There are still significant gaps between full and certifiable versions of each tool, since the certifiable versions must disable techniques that are not (fully) supported by CeTA. Among them, both NaTT and TTT2 had to disable or restrict:

- max-polynomial interpretations with negative constants [22, 16];

[2] The link in this icon directs to the web interface of TTT2, preloaded with this example.

Table 1 Certification Experiments.

Tool	Yes	No	Time (tool)	Time (CeTA)
NaTT certifiable	751	7	02:32:39	00:13:46
TTT2 w/ WPO certifiable	754	194	1d 10:31:29	00:08:18
TTT2 w/o WPO certifiable	752	194	1d 06:26:32	00:03:55
NaTT	864	169	02:42:48	–
TTT2 w/ WPO	827	205	13:48:29	–
TTT2 w/o WPO	826	205	13:46:57	–

- reachability analysis techniques: for NaTT satisfiability-oriented ones [44], and for TTT2 ones based on tree automata [34];
- uncurrying [23]: although the technique itself is fully supported [39], both NaTT and TTT2 have their own variants which exceed the capabilities of CeTA.

These observations lead to promising directions of future work. For instance, negative constants seem essentially within reach in light of certified SMT solving.

7 Summary

We have presented an extension of the IsaFoR library and the certifier CeTA with a formalization of WPO. First, we discussed how we obtained WPO as a new reduction pair in IsaFoR while relying on the already existing formalization of RPO and adapting its proofs for the requirements of WPO. Second, we described how max-polynomial interpretations were added to IsaFoR as these are often used in combination with WPO. Afterwards we gave a brief overview of the CPF format and the corresponding parser in CeTA. For this parser we define and employ a notation similar to the do-notation of Haskell, which makes the parser implementation more concise and easier to understand. Finally, we tested the new version of CeTA with the termination provers NaTT and TTT2, which both have been extended to generate CPF certificates for WPO. All generated proofs have been validated, including those for the 34 TRSs that reportedly only NaTT could prove terminating.

The main formal developments in this paper consist of only 3669 lines of Isabelle source code, since several concepts were already available in IsaFoR, e.g., lexicographic comparisons and precedences for WPO and the integer arithmetic solver for max-polynomial interpretations.

References

1 Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000. doi:10.1016/S0304-3975(99)00207-8.

2 Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998. doi:10.1017/CBO9781139172752.

3 Clemens Ballarin. Locales: A module system for mathematical theories. *J. Autom. Reasoning*, 52(2):123–153, 2014. doi:10.1007/s10817-013-9284-7.

4 Amir M. Ben-Amram and Michael Codish. A SAT-based approach to size change termination with global ranking functions. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume

4963 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2008. doi:10.1007/978-3-540-78800-3_16.

5 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.

6 Frédéric Blanqui and Adam Koprowski. CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Math. Struct. Comput. Sci.*, 21(4):827–859, 2011. doi:10.1017/S0960129511000120.

7 Ralph Bottesch, Max W. Haslbeck, Alban Reynaud, and René Thiemann. Verifying a solver for linear mixed integer arithmetic in Isabelle/HOL. In *NASA Formal Methods Symposium, 12th International Conference, Proceedings*, 2020. To appear.

8 Marc Brockschmidt, Sebastiaan J. C. Joosten, René Thiemann, and Akihisa Yamada. Certifying safety and termination proofs for integer transition systems. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 454–471. Springer, 2017. doi:10.1007/978-3-319-63046-5_28.

9 Michael Codish, Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. SAT solving for termination proofs with recursive path orders and dependency pairs. *J. Autom. Reasoning*, 49(1):53–93, 2012. doi:10.1007/s10817-010-9211-0.

10 Evelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Certification of automated termination proofs. In Boris Konev and Frank Wolter, editors, *Frontiers of Combining Systems, 6th International Symposium, FroCoS 2007, Liverpool, UK, September 10-12, 2007, Proceedings*, volume 4720 of *Lecture Notes in Computer Science*, pages 148–162. Springer, 2007. doi:10.1007/978-3-540-74621-8_10.

11 Evelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Automated certified proofs with CiME3. In Manfred Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia*, volume 10 of *LIPIcs*, pages 21–30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011. doi:10.4230/LIPIcs.RTA.2011.21.

12 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

13 Nachum Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982. doi:10.1016/0304-3975(82)90026-3.

14 Nachum Dershowitz. Termination of rewriting. *J. Symb. Comput.*, 3(1/2):69–116, 1987. doi:10.1016/S0747-7171(87)80022-6.

15 Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. *J. Autom. Reasoning*, 40(2-3):195–220, 2008. doi:10.1007/s10817-007-9087-9.

16 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. Maximal termination. In Andrei Voronkov, editor, *Rewriting Techniques and Applications, 19th International Conference, RTA 2008, Hagenberg, Austria, July 15-17, 2008, Proceedings*, volume 5117 of *Lecture Notes in Computer Science*, pages 110–125. Springer, 2008. doi:10.1007/978-3-540-70590-1_8.

17 Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Analyzing program termination and complexity automatically with aprove. *J. Autom. Reasoning*, 58(1):3–31, 2017. doi:10.1007/s10817-016-9388-y.

18 Jürgen Giesl, Albert Rubio, Christian Sternagel, Johannes Waldmann, and Akihisa Yamada. The Termination and Complexity Competition. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, volume 11429 of *Lecture Notes in Computer Science*, pages 156–166. Springer, 2019. doi:10.1007/978-3-030-17502-3_10.

19 Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In Franz Baader and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004, Montevideo, Uruguay, March 14-18, 2005, Proceedings*, volume 3452 of *Lecture Notes in Computer Science*, pages 301–331. Springer, 2004. doi:10.1007/978-3-540-32275-7_21.

20 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Mechanizing and improving dependency pairs. *J. Autom. Reasoning*, 37(3):155–203, 2006. doi:10.1007/s10817-006-9057-7.

21 Nao Hirokawa and Aart Middeldorp. Tsukuba Termination Tool. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706 of *Lecture Notes in Computer Science*, pages 311–320. Springer, 2003. doi:10.1007/3-540-44881-0_22.

22 Nao Hirokawa and Aart Middeldorp. Polynomial interpretations with negative coefficients. In Bruno Buchberger and John A. Campbell, editors, *Artificial Intelligence and Symbolic Computation, 7th International Conference, AISC 2004, Linz, Austria, September 22-24, 2004, Proceedings*, volume 3249 of *Lecture Notes in Computer Science*, pages 185–198. Springer, 2004. doi:10.1007/978-3-540-30210-0_16.

23 Nao Hirokawa, Aart Middeldorp, and Harald Zankl. Uncurrying for termination and complexity. *J. Autom. Reasoning*, 50(3):279–315, 2013. doi:10.1007/s10817-012-9248-3.

24 Jan Jakubuv and Cezary Kaliszyk. Towards a unified ordering for superposition-based automated reasoning. In James H. Davenport, Manuel Kauers, George Labahn, and Josef Urban, editors, *Mathematical Software - ICMS 2018 - 6th International Conference, South Bend, IN, USA, July 24-27, 2018, Proceedings*, volume 10931 of *Lecture Notes in Computer Science*, pages 245–254. Springer, 2018. doi:10.1007/978-3-319-96418-8_29.

25 Sam Kamin and Jean-Jacques Lévy. Two generalizations of the recursive path ordering, 1980. Unpublished note.

26 Donald E. Knuth and Peter Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, New York, 1970. doi:10.1016/B978-0-08-012975-4.50028-X.

27 Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean Termination Tool 2. In Ralf Treinen, editor, *Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil, June 29 - July 1, 2009, Proceedings*, volume 5595 of *Lecture Notes in Computer Science*, pages 295–304. Springer, 2009. doi:10.1007/978-3-642-02348-4_21.

28 Alexander Krauss. Recursive definitions of monadic functions. In Ekaterina Komendantskaya, Ana Bove, and Milad Niqui, editors, *Partiality and Recursion in Interactive Theorem Provers, PAR@ITP 2010, Edinburgh, UK, July 15, 2010*, volume 5 of *EPiC Series*, pages 1–13. EasyChair, 2010. URL: http://www.easychair.org/publications/paper/52847.

29 Keiichirou Kusakari, Masaki Nakamura, and Yoshihito Toyama. Argument filtering transformation. In Gopalan Nadathur, editor, *Principles and Practice of Declarative Programming, International Conference PPDP'99, Paris, France, September 29 - October 1, 1999, Proceedings*, volume 1702 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 1999. doi:10.1007/10704567_3.

30 Dallas Lankford. Canonical algebraic simplification in computational logic. Technical Report ATP-25, University of Texas, 1975.

31 Dallas Lankford. On proving term rewrite systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA, 1979.

32 Salvador Lucas. MU-TERM: A tool for proving termination of context-sensitive rewriting. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 200–209. Springer, 2004. doi:10.1007/978-3-540-25979-4_14.

33 Zohar Manna and Stephen Ness. On the termination of Markov algorithms. In *3rd Hawaii International Conference on System Science*, pages 789–792, 1970.

34 Aart Middeldorp. Approximating dependency graphs using tree automata techniques. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, Proceedings*, volume 2083 of *Lecture Notes in Computer Science*, pages 593–610. Springer, 2001. doi:10.1007/3-540-45744-5_49.

35 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.

36 Anders Schlichtkrull, Jasmin Christian Blanchette, and Dmitriy Traytel. A verified prover based on ordered resolution. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 152–165. ACM, 2019. doi:10.1145/3293880.3294100.

37 Peter Schneider-Kamp, René Thiemann, Elena Annov, Michael Codish, and Jürgen Giesl. Proving termination using recursive path orders and SAT solving. In Boris Konev and Frank Wolter, editors, *Frontiers of Combining Systems, 6th International Symposium, FroCoS 2007, Liverpool, UK, September 10-12, 2007, Proceedings*, volume 4720 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 2007. doi:10.1007/978-3-540-74621-8_18.

38 Jonas Schöpf and Christian Sternagel. $\mathsf{T_TT_2}$ with termination templates for teaching. In *Proc. of the 16th International Workshop on Termination*, 2018. URL: http://arxiv.org/abs/1806.05040.

39 Christian Sternagel and René Thiemann. Generalized and formalized uncurrying. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings*, volume 6989 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2011. doi:10.1007/978-3-642-24364-6_17.

40 Christian Sternagel and René Thiemann. Formalizing Knuth-Bendix orders and Knuth-Bendix completion. In Femke van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands*, volume 21 of *LIPIcs*, pages 287–302. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013. doi:10.4230/LIPIcs.RTA.2013.287.

41 Christian Sternagel and René Thiemann. The certification problem format. In Christoph Benzmüller and Bruno Woltzenlogel Paleo, editors, *Proceedings Eleventh Workshop on User Interfaces for Theorem Provers, UITP 2014, Vienna, Austria, 17th July 2014*, volume 167 of *EPTCS*, pages 61–72, 2014. doi:10.4204/EPTCS.167.8.

42 Christian Sternagel and René Thiemann. Formalizing monotone algebras for certification of termination and complexity proofs. In Gilles Dowek, editor, *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8560 of *Lecture Notes in Computer Science*, pages 441–455. Springer, 2014. doi:10.1007/978-3-319-08918-8_30.

43 Christian Sternagel and René Thiemann. XML. *Archive of Formal Proofs*, October 2014. , Formal proof development. URL: http://isa-afp.org/entries/XML.html.

44 Christian Sternagel and Akihisa Yamada. Reachability analysis for termination and confluence of rewriting. In Tomás Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 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, Part I*, volume 11427 of *Lecture Notes in Computer Science*, pages 262–278. Springer, 2019. doi:10.1007/978-3-030-17462-0_15.

45 Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A cross-community infrastructure for logic solving. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, volume 8562 of *Lecture Notes in Computer Science*, pages 367–373. Springer, 2014. doi:10.1007/978-3-319-08587-6_28.

46 René Thiemann, Guillaume Allais, and Julian Nagele. On the Formalization of Termination Techniques based on Multiset Orderings. In Ashish Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, volume 15 of *LIPIcs*, pages 339–354. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPIcs.RTA.2012.339.

47 René Thiemann and Christian Sternagel. Certification of termination proofs using CeTA. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468. Springer, 2009. doi:10.1007/978-3-642-03359-9_31.

48 Xavier Urbain. Modular and incremental automated termination proofs. *J. Autom. Reasoning*, 32(4):315–355, 2004. doi:10.1007/BF03177743.

49 Johannes Waldmann. Matchbox: A tool for match-bounded string rewriting. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 85–94. Springer, 2004. doi:10.1007/978-3-540-25979-4_6.

50 Akihisa Yamada, Keiichirou Kusakari, and Toshiki Sakabe. Partial status for KBO. In Johannes Waldmann, editor, *13th International Workshop on Termination (WST 2013)*, pages 74–78, 2013.

51 Akihisa Yamada, Keiichirou Kusakari, and Toshiki Sakabe. Unifying the Knuth-Bendix, recursive path and polynomial orders. In Ricardo Peña and Tom Schrijvers, editors, *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, pages 181–192. ACM, 2013. doi:10.1145/2505879.2505885.

52 Akihisa Yamada, Keiichirou Kusakari, and Toshiki Sakabe. Nagoya Termination Tool. In Gilles Dowek, editor, *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8560 of *Lecture Notes in Computer Science*, pages 466–475. Springer, 2014. doi:10.1007/978-3-319-08918-8_32.

53 Akihisa Yamada, Keiichirou Kusakari, and Toshiki Sakabe. A unified ordering for termination proving. *Sci. Comput. Program.*, 111:110–134, 2015. doi:10.1016/j.scico.2014.07.009.

54 Harald Zankl, Nao Hirokawa, and Aart Middeldorp. Constraints for argument filterings. In Jan van Leeuwen, Giuseppe F. Italiano, Wiebe van der Hoek, Christoph Meinel, Harald Sack, and Frantisek Plasil, editors, *SOFSEM 2007: Theory and Practice of Computer Science, 33rd Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 20-26, 2007, Proceedings*, volume 4362 of *Lecture Notes in Computer Science*, pages 579–590. Springer, 2007. doi:10.1007/978-3-540-69507-3_50.

55 Harald Zankl, Nao Hirokawa, and Aart Middeldorp. KBO orientability. *J. Autom. Reasoning*, 43(2):173–201, 2009. doi:10.1007/s10817-009-9131-z.

56 Harald Zankl and Aart Middeldorp. Satisfiability of non-linear (ir)rational arithmetic. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 481–500. Springer, 2010. `doi:10.1007/978-3-642-17511-4_27`.

57 Hans Zantema. Termination of string rewriting proved automatically. *J. Autom. Reasoning*, 34(2):105–139, 2005. `doi:10.1007/s10817-005-6545-0`.

Efficient Full Higher-Order Unification

Petar Vukmirović
Vrije Universiteit Amsterdam, The Netherlands
p.vukmirovic@vu.nl

Alexander Bentkamp
Vrije Universiteit Amsterdam, The Netherlands
a.bentkamp@vu.nl

Visa Nummelin
Vrije Universiteit Amsterdam, The Netherlands
visa.nummelin@vu.nl

Abstract

We developed a procedure to enumerate complete sets of higher-order unifiers based on work by Jensen and Pietrzykowski. Our procedure removes many redundant unifiers by carefully restricting the search space and tightly integrating decision procedures for fragments that admit a finite complete set of unifiers. We identify a new such fragment and describe a procedure for computing its unifiers. Our unification procedure is implemented in the Zipperposition theorem prover. Experimental evaluation shows a clear advantage over Jensen and Pietrzykowski's procedure.

2012 ACM Subject Classification Computing methodologies → Theorem proving algorithms

Keywords and phrases unification, higher-order logic, theorem proving, term rewriting, indexing data structures

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.5

Related Version A technical report contains further details and proofs. It is available at `http://matryoshka.gforge.inria.fr/pubs/hounif_report.pdf`.

Supplementary Material An archive with raw results, all used problems, and scripts for running each configuration is available at `http://matryoshka.gforge.inria.fr/pubs/hounif_data.zip`.

Funding Vukmirović and Bentkamp's research has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). Nummelin has received funding from the Netherlands Organization for Scientific Research (NWO) under the Vidi program (project No. 016.Vidi.189.037, Lean Forward).

Acknowledgements We are grateful to the maintainers of StarExec for letting us use their service. We thank Ahmed Bhayat, Jasmin Blanchette, Daniel El Ouraoui, Mathias Fleury, Pascal Fontaine, Predrag Janičić, Robert Lewis, Femke van Raamsdonk, Hans-Jörg Schurr, Sophie Tourret, Dmitriy Traytel, and the anonymous reviewers for suggesting many improvements to this text.

1 Introduction

Unification is concerned with finding a substitution that makes two terms equal, for some notion of syntactic equality. Since the invention of Robinson's first-order unification algorithm [19], it has become an indispensable tool in theorem proving, logic programming, natural language processing, programming language compilation and other areas of computer science.

Many of these applications are based on higher-order formalisms and require higher-order unification. Due to its undecidability and explosiveness, the higher-order unification problem is considered one of the main obstacles on the road to efficient higher-order tools.

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 5; pp. 5:1–5:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

One of the reasons for higher-order unification's explosiveness lies in *flex-flex pairs*, which consist of two applied variables, e.g., $F\,X \overset{?}{=} G\,\mathsf{a}$, where F, G, and X are variables and a is a constant. Even this seemingly simple problem has infinitely many incomparable unifiers. One of the first methods designed to combat this explosion is Huet's preunification [10]. Huet noticed that some logical calculi would remain complete if flex-flex pairs are not eagerly solved but postponed as constraints. If only flex-flex constraints remain, we know that a unifier must exist and we do not need to solve them. Huet's preunification has been used in many reasoning tools including Isabelle [17], Leo-III [22], and Satallax [3]. However, recent developments in higher-order theorem proving [1,2] require full unification – i.e., enumeration of unifiers even for flex-flex pairs, which is the focus of this paper.

Jensen and Pietrzykowski's (JP) procedure [11] is the best known procedure for this purpose (Section 2). Given two terms to unify, it first identifies a position where the terms disagree. Then, in parallel branches of the search tree, it applies suitable substitutions, involving a variable either at the position of disagreement or above, and repeats this process on the resulting terms until they are equal or trivially nonunifiable.

Building on the JP procedure, we designed an improved procedure with the same completeness guarantees (Section 3). It addresses many of the issues that are detrimental to the performance of the JP procedure.

First, the JP procedure does not terminate in many cases of obvious nonunifiability, e.g., for $X \overset{?}{=} \mathsf{f}\,X$, where X is a non-functional variable and f is a function constant. This example also shows that the JP procedure does not generalize Robinson's first-order procedure gracefully. To address this issue, our procedure detects whether a unification problem belongs to a fragment for which unification is decidable and finite complete sets of unifiers (CSUs) exist. We call algorithms that enumerate elements of the CSU for such fragments *oracles*. Noteworthy fragments with oracles are first-order terms, patterns [16], functions-as-constructors [13], and a new fragment we present in Section 4. The unification procedures of Isabelle and Leo-III check whether the unification problem belongs to a decidable fragment, but we take this idea a step further by checking this more efficiently and for every subproblem arising during unification.

Second, the JP procedure computes many redundant unifiers. Consider the example $F\,(G\,\mathsf{a}) \overset{?}{=} F\,\mathsf{b}$, where JP produces, in addition to the desired unifiers $\{F \mapsto \lambda x.\,H\}$ and $\{G \mapsto \lambda x.\,\mathsf{b}\}$, the redundant unifier $\{F \mapsto \lambda x.\,H,\ G \mapsto \lambda x.\,x\}$. The design of our procedure avoids computing many redundant unifiers, including this one. Additionally, as oracles usually return a small CSU, their integration reduces the number of redundant unifiers.

Third, the JP procedure applies infinitely branching rule to flex-rigid pairs, whereas Huet's preunification procedure solves flex-rigid pairs using simpler, finitely branching rules. To gracefully generalize Huet's procedure, we show that his rules for flex-rigid pairs suffice to enumerate CSUs if combined with appropriate rules for flex-flex pairs.

Fourth, the JP procedure repeatedly traverses the parts of the unification problem that have already been unified. Consider the problem $\mathsf{f}^{100}\,(G\,\mathsf{a}) \overset{?}{=} \mathsf{f}^{100}\,(H\,\mathsf{b})$, where the exponents denote repeated application. It is easy to see that this problem can be reduced to $G\,\mathsf{a} \overset{?}{=} H\,\mathsf{b}$. However, the JP procedure will wastefully retraverse the common context $\mathsf{f}^{100}[\,]$ after applying each new substitution. Since the JP procedure must apply substitutions to the variables occurring in the common context above the disagreement pair, it cannot be easily adapted to eagerly decompose unification pairs. By contrast, our procedure is designed to decompose the pairs eagerly, never traversing a common context twice.

Last, optimizations such as lazy application of substitutions and lazy β-normalization cannot easily be integrated into the JP procedure. The rules of simpler procedures (e.g., first-order [9] and pattern unification [16]) depend only on the heads of the unification pair. Thus, to determine the next step, implementations of these procedures need to substitute and β-reduce only until the heads of the current unification pair are not mapped by the substitution and are not λ-abstractions. Since the JP procedure is not based on the decomposition of unification pairs, it is unfit for optimizations of this kind. We designed our procedure to allow for this optimization.

To filter out some of the terms that are not unifiable with a given query term from a set of terms, we developed a higher-order extension of fingerprint indexing [20] (Section 5). We implemented our procedure, several oracles, and the fingerprint index in the Zipperposition prover (Section 6). Since a straightforward implementation of the JP procedure already existed in Zipperposition, we used it as a baseline to evaluate the performance of our procedure (Section 7). The results show substantial performance improvements.

This paper lays out the main ideas behind our unification procedure. A separate technical report contains details and proofs of all statements [27].

2 Background

Our setting is the simply typed λ-calculus. Types α, β, γ are either base types or functional types $\alpha \to \beta$. By convention, when we write $\alpha_1 \to \cdots \to \alpha_n \to \beta$, we assume β to be a base type. Basic terms are free variables (denoted $F, G, H, \ldots$), bound variables (x, y, z), and constants $(\mathsf{f}, \mathsf{g}, \mathsf{h})$. Complex terms are applications of one term to another $(s\,t)$ or λ-abstractions $(\lambda x.\, s)$. Following Nipkow [16], we use these syntactic conventions to distinguish free from bound variables. Bound variables with no enclosing binder, such as x in $\lambda y.\, x$, are called *loose bound variables*. We say that a term without loose bound variables is *closed* and a term without free variables is *ground*. Iterated λ-abstraction $\lambda x_1 \ldots \lambda x_n.\, s$ is abbreviated as $\lambda \overline{x}_n.\, s$ and iterated application $(s\,t_1)\ \ldots\ t_n$ as $s\,\overline{t}_n$, where $n \geq 0$. Similarly, we denote a sequence of terms $t_1, \ldots, t_n$ by $\overline{t}_n$, omitting its length $n \geq 0$ where it can be inferred or is irrelevant.

We assume the standard notions of α-, β-, η-conversions. A term is in *head normal form* (*hnf*) if it is of the form $\lambda \overline{x}.\, a\,\overline{t}$, where a is a free variable, bound variable, or a constant. In this case, a is called the *head* of the term. By convention, a and b denote heads. If a is a free variable, we call it a *flex* head; otherwise, we call it a *rigid* head. A term is called flex or rigid if its head is flex or rigid, respectively. By $s_{\downarrow\mathsf{h}}$ we denote the term obtained from a term s by repeated β-reduction of the leftmost outermost redex until it is in hnf. Unless stated otherwise, we view terms syntactically, as opposed to $\alpha\beta\eta$-equivalence classes. We write $s \leftrightarrow^*_{\alpha\beta\eta} t$ if s and t are $\alpha\beta\eta$-equivalent. Substitutions $(\sigma, \varrho, \theta)$ are functions from free and bound variables to terms; σt denotes application of σ to t, which α-renames t to avoid variable capture. The composition $\varrho\sigma$ of substitutions is defined by $(\varrho\sigma)t = \varrho(\sigma t)$. A variable F is mapped by σ if $\sigma F \not\leftrightarrow^*_{\alpha\beta\eta} F$. We write $\varrho \subseteq \sigma$ if for all variables F mapped by ϱ, $\varrho F \leftrightarrow^*_{\alpha\beta\eta} \sigma F$.

Deviating from the standard notion of higher-order subterm, we define subterms on β-reduced terms as follows: a term t is a subterm of t at position ε. If s is a subterm of u_i at position p, then s is a subterm of $a\,\overline{u}_n$ at position $i.p$. If s is a subterm of t at position p, then s is a subterm of $\lambda x.\, t$ at position $1.p$. Our definition of subterm gracefully generalizes the corresponding first-order notion: a is a subterm of $\mathsf{f\,a\,b}$, but f and $\mathsf{f\,a}$ are not subterms of $\mathsf{f\,a\,b}$. A context is a term with zero or more subterms replaced by a hole $\square$. We write $C[\overline{u}_n]$ for the term resulting from filling in the holes of a context C with the terms $\overline{u}_n$ from left to right. The common context $\mathcal{C}(s, t)$ of two η-long β-reduced terms s and t of the same type is defined inductively as follows, assuming that $a \neq b$: $\mathcal{C}(\lambda x.\, s, \lambda y.\, t) = \lambda x.\, \mathcal{C}(s, \{y \mapsto x\}t)$; $\mathcal{C}(a\,\overline{s}_m, b\,\overline{t}_n) = \square$; $\mathcal{C}(a\,\overline{s}_m, a\,\overline{t}_m) = a\,\mathcal{C}(s_1, t_1)\ \ldots\ \mathcal{C}(s_m, t_m)$.

A *unifier* for terms s and t is a substitution σ, such that $\sigma s \leftrightarrow^*_{\alpha\beta\eta} \sigma t$. Following JP [11], a *complete set of unifiers* (*CSU*) of terms s and t is defined as a set U of unifiers for s and t such that for every unifier ϱ of s and t, there exists $\sigma \in U$ and substitution θ such that $\varrho \subseteq \theta\sigma$. A *most general unifier* (*MGU*) is a one-element CSU. We use $\subseteq$ instead of $=$ because a CSU element σ may introduce auxiliary variables not mapped by ϱ.

3 The Unification Procedure

To unify two terms s and t, our procedure builds a tree as follows. The nodes of the tree have the form (E, σ), where E is a multiset of unification constraints $\{(s_1 \overset{?}{=} t_1), \ldots, (s_n \overset{?}{=} t_n)\}$ and σ is the substitution constructed up to that point. A unification constraint $s \overset{?}{=} t$ is an unordered pair of two terms of the same type. The root node of the tree is $(\{s \overset{?}{=} t\}, \mathrm{id})$, where id is the identity substitution. The tree is then constructed applying the transitions listed below. The leaves of the tree are either failure nodes $\bot$ or substitutions σ. Ignoring failure nodes, the set of all substitutions in the leaves forms a complete set of unifiers for s and t.

The transitions are parametrized by a mapping $\mathcal{P}$ that assigns a set of substitutions to a unification pair; this mapping abstracts the concept of unification rules present in other unification procedures. Moreover, the transitions are parametrized by a selection function S mapping a multiset E of unification constraints to one of those constraints $S(E) \in E$, the *selected* constraint in E. The transitions, defined as follows, are only applied if the grayed constraint is selected.

Succeed $(\varnothing, \sigma) \longrightarrow \sigma$

Normalize$_{\alpha\eta}$ $(\{\lambda\overline{x}_m.\, s \overset{?}{=} \lambda\overline{y}_n.\, t\} \uplus E, \sigma) \longrightarrow (\{\lambda\overline{x}_m.\, s \overset{?}{=} \lambda\overline{x}_m.\, t'\, x_{n+1} \ldots x_m\} \uplus E, \sigma)$
where $m \geq n$, $\overline{x}_m \neq \overline{y}_n$, and $t' = \{y_1 \mapsto x_1, \ldots, y_n \mapsto x_n\}t$

Normalize$_\beta$ $(\{\lambda\overline{x}.\, s \overset{?}{=} \lambda\overline{x}.\, t\} \uplus E, \sigma) \longrightarrow (\{\lambda\overline{x}.\, s_{\downarrow\mathsf{h}} \overset{?}{=} \lambda\overline{x}.\, t_{\downarrow\mathsf{h}}\} \uplus E, \sigma)$
where s or t is not in hnf

Dereference $(\{\lambda\overline{x}.\, F\, \overline{s} \overset{?}{=} \lambda\overline{x}.\, t\} \uplus E, \sigma) \longrightarrow (\{\lambda\overline{x}.\, (\sigma F)\, \overline{s} \overset{?}{=} \lambda\overline{x}.\, t\} \uplus E, \sigma)$
where none of the previous transitions apply and F is mapped by σ

Fail $(\{\lambda\overline{x}.\, a\, \overline{s}_m \overset{?}{=} \lambda\overline{x}.\, b\, \overline{t}_n\} \uplus E, \sigma) \longrightarrow \bot$
where none of the previous transitions apply, and a and b are different rigid heads

Delete $(\{s \overset{?}{=} s\} \uplus E, \sigma) \longrightarrow (E, \sigma)$
where none of the previous transitions apply

OracleSucc $(\{s \overset{?}{=} t\} \uplus E, \sigma) \longrightarrow (E, \varrho\sigma)$
where none of the previous transitions apply, some oracle found a finite CSU U for $\sigma(s) \overset{?}{=} \sigma(t)$, and $\varrho \in U$; if multiple oracles found a CSU, only one of them is considered

OracleFail $(\{s \overset{?}{=} t\} \uplus E, \sigma) \longrightarrow \bot$
where none of the previous transitions apply, and some oracle determined $\sigma(s) \overset{?}{=} \sigma(t)$ has no solutions

Decompose $(\{\lambda\overline{x}.\, a\, \overline{s}_m \overset{?}{=} \lambda\overline{x}.\, a\, \overline{t}_m\} \uplus E, \sigma) \longrightarrow (\{s_1 \overset{?}{=} t_1, \ldots, s_m \overset{?}{=} t_m\} \uplus E, \sigma)$
where none of the transitions Succeed to OracleFail apply

Bind $(\{s \overset{?}{=} t\} \uplus E, \sigma) \longrightarrow (\{s \overset{?}{=} t\} \uplus E, \varrho\sigma)$
where none of the transitions Succeed to OracleFail apply, and $\varrho \in \mathcal{P}(s \overset{?}{=} t)$.

The transitions are designed so that only OracleSucc, Decompose, and Bind can introduce parallel branches in the constructed tree. OracleSucc can introduce branches using different unifiers of the CSU, Bind can introduce branches using different substitutions in $\mathcal{P}$, and Decompose can be applied in parallel with Bind.

Our approach is to apply substitutions and $\alpha\beta\eta$-normalize terms lazily. In particular, the transitions that modify the constructed substitution, OracleSucc and Bind, do not apply that substitution to the unification pairs directly. Instead, the transitions $\mathsf{Normalize}_{\alpha\eta}$, $\mathsf{Normalize}_{\beta}$, and Dereference partially normalize and partially apply the constructed substitution just enough to ensure that the heads are the ones we would get if the substitution was fully applied and the term was fully normalized. To support lazy dereferencing, OracleSucc and Bind must maintain the invariant that all substitutions are idempotent.

The OracleSucc and OracleFail transitions invoke oracles, such as pattern unification, to compute a CSU faster, produce fewer redundant unifiers, and discover nonunifiability earlier. In some cases, addition of oracles lets the procedure terminate more often.

In the literature, oracles are usually stated under the assumption that their input belongs to the appropriate fragment. To check whether a unification constraint is inside the fragment, we need to fully apply the substitution and β-normalize the constraint. To avoid these expensive operations and enable efficient oracle integration, oracles must be redesigned to lazily discover whether the terms belong to their fragment. Most oracles contain a decomposition operation which requires only a partial application of the substitution and only partial β-normalization. If one of the constraints resulting from decomposition is not in the fragment, the original problem is not in the fragment. This allows us to detect that the problem is not in the fragment without fully applying the substitution and β-normalizing.

The core of the procedure lies in the Bind step, parameterized by the mapping $\mathcal{P}$ that determines which substitutions (called *bindings*) to create. The bindings are defined as follows:

Iteration for F Let F be a free variable of the type $\alpha_1 \to \cdots \to \alpha_n \to \beta_1$ and let some α_i be the type $\gamma_1 \to \cdots \to \gamma_m \to \beta_2$, where $n > 0$ and $m \geq 0$. Iteration for F at i is

$$F \mapsto \lambda \overline{x}_n.\, H\, \overline{x}_n\, (\lambda \overline{y}.\, x_i\, (G_1\, \overline{x}_n\, \overline{y}) \ldots (G_m\, \overline{x}_n\, \overline{y}))$$

The free variables H and $G_1, \ldots, G_m$ are fresh, and $\overline{y}$ is an arbitrary-length sequence of bound variables of arbitrary types. All new variables are of appropriate type. Due to indeterminacy of $\overline{y}$, this step is infinitely branching.

JP-style projection for F Let F be a free variable of type $\alpha_1 \to \cdots \to \alpha_n \to \beta$, where some α_i is equal to β and $n > 0$. Then the JP-style projection binding is

$$F \mapsto \lambda \overline{x}_n.\, x_i$$

Huet-style projection for F Let F be a free variable of type $\alpha_1 \to \cdots \to \alpha_n \to \beta$, where some $\alpha_i = \gamma_1 \to \cdots \to \gamma_m \to \beta$, $n > 0$ and $m \geq 0$. Huet-style projection is

$$F \mapsto \lambda \overline{x}_n.\, x_i\, (F_1\, \overline{x}_n)\ \ldots\ (F_m\, \overline{x}_n)$$

where the fresh free variables $\overline{F}_m$ and bound variables $\overline{x}_n$ are of appropriate types.

Imitation of g for F Let F be a free variable of type $\alpha_1 \to \cdots \to \alpha_n \to \beta$ and let g be a constant of type $\gamma_1 \to \cdots \to \gamma_m \to \beta$ where $n, m \geq 0$. The imitation binding is

$$F \mapsto \lambda \overline{x}_n.\, \mathsf{g}\, (F_1\, \overline{x}_n) \ldots (F_m\, \overline{x}_n)$$

where the fresh free variables $\overline{F}_m$ and bound variables $\overline{x}_n$ are of appropriate types.

Identification for F and G Let F and G be different free variables. Furthermore, let the type of F be $\alpha_1 \to \cdots \to \alpha_n \to \beta$ and the type of G be $\gamma_1 \to \cdots \to \gamma_m \to \beta$, where $n, m \geq 0$. Then, identification binding binds F and G with

$$F \mapsto \lambda \overline{x}_n.\, H\, \overline{x}_n\, (F_1\, \overline{x}_n) \ldots (F_m\, \overline{x}_n) \qquad G \mapsto \lambda \overline{y}_m.\, H\, (G_1\, \overline{y}_m) \ldots (G_n\, \overline{y}_m)\, \overline{y}_m$$

where the fresh free variables $H, \overline{F}_m, \overline{G}_n$ and bound variables $\overline{x}_n$,$\overline{y}_m$ are of appropriate types. We call fresh variables emerging from this binding in the role of H *identification variables.*

Elimination for F Let F be a free variable of type $\alpha_1 \to \cdots \to \alpha_n \to \beta$, where $n > 0$. In addition, let $1 \leq j_1 < \cdots < j_i \leq n$ and $i < n$. Elimination for the sequence $(j_k)_{k=1}^{i}$ is

$$F \mapsto \lambda \overline{x}_n.\, G\, x_{j_1}\, \ldots\, x_{j_i}$$

where the fresh free variable G as well as all x_{j_k} are of appropriate type. We call fresh variables emerging from this binding in the role of G *elimination variables.*

Given a unification constraint $\lambda\overline{x}.\, s \overset{?}{=} \lambda\overline{x}.\, t$, $\mathcal{P}$ is defined as follows:

- If the constraint is rigid-rigid, $\mathcal{P}(\lambda\overline{x}.\, s \overset{?}{=} \lambda\overline{x}.\, t) = \varnothing$.
- If the constraint is flex-rigid, let $\mathcal{P}(\lambda\overline{x}.\, F\,\overline{s} \overset{?}{=} \lambda\overline{x}.\, a\,\overline{t})$ be
  - an imitation of a for F, if a is some constant g, and
  - all Huet-style projections for F, if F is not an identification variable.
- If the constraint is flex-flex and the heads are different, let $\mathcal{P}(\lambda\overline{x}.\, F\,\overline{s} \overset{?}{=} \lambda\overline{x}.\, G\,\overline{t})$ be
  - all identifications and iterations for both F and G, and
  - all JP-style projections for non-identification variables among F and G.
- If the constraint is flex-flex and the heads are identical, we distinguish two cases:
  - if the head is an elimination variable, $\mathcal{P}(\lambda\overline{x}.\, s \overset{?}{=} \lambda\overline{x}.\, t) = \varnothing$;
  - otherwise, let $\mathcal{P}(\lambda\overline{x}.\, F\,\overline{s} \overset{?}{=} \lambda\overline{x}.\, F\,\overline{t})$ be all iterations for F at arguments of functional type and all eliminations for F.

Comparison with the JP Procedure. In contrast to our procedure, the JP procedure constructs a tree with only one unification constraint per node and does not have a Decompose rule. Instead, at each node $(s \overset{?}{=} t, \sigma)$, the JP procedure computes the common context C of s and t, yielding term pairs $(s_1, t_1), \ldots, (s_n, t_n)$, called *disagreement pairs*, such that $s = C[s_1, \ldots, s_n]$ and $t = C[t_1, \ldots, t_n]$. The procedure heuristically chooses one of the disagreement pairs (s_i, t_i) and applies a binding to the heads of s_i and t_i or to a free variable occurring above the disagreement pair in the common context C. Due to this application of bindings above the disagreement pair, lazy normalization and dereferencing cannot easily be integrated into the JP procedure.

The mapping $\mathcal{P}$ uses many binding rules of the JP procedure, but our procedure explores the search space differently. In particular, the JP procedure allows iteration or elimination to be applied at a free variable in the common context of the unification constraint, even if bindings were already applied below that free variable. In contrast, our procedure forces the eliminations and iterations to be applied as soon as it observes a flex-flex pair with identical heads. After applying the Decompose transition, this flex-flex pair will be reduced to pairs representing the arguments of the identical heads. Therefore, unlike the JP procedure, it will not apply bindings to the flex-flex pair after bindings have been applied to its arguments.

The JP procedure can be modified to solve the preunification problem by making it choose only flex-rigid disagreement pairs and terminate with a preunifier when no flex-rigid pair remains. However, such a procedure would be less efficient than Huet's procedure because it would use the iteration binding instead of Huet-style projection to solve flex-rigid pairs. Our procedure applies Huet-style projections on flex-rigid pairs, which results in two important improvements over the JP procedure. First, our procedure terminates more often than the JP procedure because Huet-style projections cause only a finite branching, whereas iteration causes an infinite branching. Second, when our procedure is modified to solve the preunification problem by never selecting flex-flex pairs and stopping when only flex-flex pairs are left, it becomes an optimized variant of Huet's procedure that supports oracles as well as lazy substitution and β-reduction.

The bindings of our procedure contain further optimizations that are absent in the JP procedure: The JP procedure applies eliminations for only one parameter at a time, yielding multiple paths to the same unifier. It applies imitations to flex-flex pairs, which we found to be unnecessary. Moreover, it does not keep track of which rules introduced which variables: iterations and eliminations are applied on elimination variables, and projections are applied on identification variables.

Examples. We present some illustrative derivations. The displayed branches of the constructed trees are not necessarily exhaustive. We abbreviate JP-style projection as JP Proj, imitation as Imit, identification as Id, Decompose as Dc, Dereference as Dr, Normalize$_\beta$ as N$_\beta$, and Bind of a binding x as B(x). Transitions of the JP procedure are denoted by $\Longrightarrow$. For the JP transitions we implicitly apply the generated bindings and fully normalize terms, which significantly shortens JP derivations.

▶ **Example 1.** The JP procedure does not terminate on the problem $G \overset{?}{=} \mathsf{f}\, G$:

$$(G \overset{?}{=} \mathsf{f}\, G, \mathrm{id}) \overset{\mathsf{Imit}}{\Longrightarrow} (\mathsf{f}\, G' \overset{?}{=} \mathsf{f}^2\, G', \sigma_1) \overset{\mathsf{Imit}}{\Longrightarrow} (\mathsf{f}^2\, G'' \overset{?}{=} \mathsf{f}^3\, G'', \sigma_2) \overset{\mathsf{Imit}}{\Longrightarrow} \cdots$$

where $\sigma_1 = \{G \mapsto \lambda x.\, \mathsf{f}\, G'\}$ and $\sigma_2 = \{G' \mapsto \lambda x.\, \mathsf{f}\, G''\}\sigma_1$. By including any oracle that supports first-order occurs check, such as the pattern oracle or the fixpoint oracle described in Section 6, our procedure gracefully generalizes first-order unification:

$$(\{G \overset{?}{=} \mathsf{f}\, G\}, \mathrm{id}) \xrightarrow{\mathsf{OracleFail}} \bot$$

▶ **Example 2.** The following derivation illustrates the advantage of the Decompose rule.

$$\begin{aligned}
&(\{\mathsf{h}^{100}\,(F\,\mathsf{a}) \overset{?}{=} \mathsf{h}^{100}\,(G\,\mathsf{b})\}, \mathrm{id}) \xrightarrow{\mathsf{Dc}^{100}} (\{F\,\mathsf{a} \overset{?}{=} G\,\mathsf{b}\}, \mathrm{id}) \xrightarrow{\mathsf{B(Id)}} (\{F\,\mathsf{a} \overset{?}{=} G\,\mathsf{b}\}, \sigma_1)\\
&\xrightarrow{\mathsf{Dr{+}N}_\beta} (\{H\,\mathsf{a}\,(F'\,\mathsf{a}) \overset{?}{=} H\,(G'\,\mathsf{b})\,\mathsf{b}\}, \sigma_1) \xrightarrow{\mathsf{Dc}} (\{\mathsf{a} \overset{?}{=} G'\,\mathsf{b}, F'\,\mathsf{a} \overset{?}{=} \mathsf{b}\}, \sigma_1)\\
&\xrightarrow{\mathsf{B(Imit)}} (\{\mathsf{a} \overset{?}{=} G'\,\mathsf{b}, F'\,\mathsf{a} \overset{?}{=} \mathsf{b}\}, \sigma_2) \xrightarrow{\mathsf{Dr{+}N}_\beta} (\{\mathsf{a} \overset{?}{=} \mathsf{a}, F'\,\mathsf{a} \overset{?}{=} \mathsf{b}\}, \sigma_2) \xrightarrow{\mathsf{Delete}} (\{F'\,\mathsf{a} \overset{?}{=} \mathsf{b}\}, \sigma_2)\\
&\xrightarrow{\mathsf{B(Imit)}} (\{F'\,\mathsf{a} \overset{?}{=} \mathsf{b}\}, \sigma_3) \xrightarrow{\mathsf{Dr{+}N}_\beta} (\{\mathsf{b} \overset{?}{=} \mathsf{b}\}, \sigma_3) \xrightarrow{\mathsf{Delete}} (\varnothing, \sigma_3) \xrightarrow{\mathsf{Succeed}} \sigma_3
\end{aligned}$$

where $\sigma_1 = \{F \mapsto \lambda x.\, H\, x\,(F'\,x), G \mapsto \lambda y.\, H\,(G'\,y)\,y\}$; $\sigma_2 = \{G' \mapsto \lambda x.\,\mathsf{a}\}\sigma_1$; and $\sigma_3 = \{F' \mapsto \lambda x.\,\mathsf{b}\}\sigma_2$. The JP procedure produces the same intermediate substitutions σ_1 to σ_3, but since it does not decompose the terms, it retraverses the common context $\mathsf{h}^{100}\,[\,]$ at every step to identify the contained disagreement pair:

$$\begin{aligned}
&(\mathsf{h}^{100}\,(F\,\mathsf{a}) \overset{?}{=} \mathsf{h}^{100}\,(G\,\mathsf{b}), \mathrm{id}) \overset{\mathsf{Id}}{\Longrightarrow} (\mathsf{h}^{100}\,(H\,\mathsf{a}\,(F'\,\mathsf{a})) \overset{?}{=} \mathsf{h}^{100}\,(H\,(G'\,\mathsf{b})\,\mathsf{b}), \sigma_1)\\
&\overset{\mathsf{Imit}}{\Longrightarrow} (\mathsf{h}^{100}\,(H\,\mathsf{a}\,(F'\,\mathsf{a})) \overset{?}{=} \mathsf{h}^{100}\,(H\,\mathsf{a}\,\mathsf{b}), \sigma_2) \overset{\mathsf{Imit}}{\Longrightarrow} (\mathsf{h}^{100}\,(H\,\mathsf{a}\,\mathsf{b}) \overset{?}{=} \mathsf{h}^{100}\,(H\,\mathsf{a}\,\mathsf{b}), \sigma_3) \overset{\mathsf{Succeed}}{\Longrightarrow} \sigma_3
\end{aligned}$$

▶ **Example 3.** The search space restrictions also allow us to prune some redundant unifiers. Consider the problem $F\,(G\,\mathsf{a}) \overset{?}{=} F\,\mathsf{b}$, where a and b are of base type. Our procedure produces only one failing branch and the following two successful branches:

$$\begin{aligned}
&(\{F\,(G\,\mathsf{a}) \overset{?}{=} F\,\mathsf{b}\}, \mathrm{id}) \xrightarrow{\mathsf{Dc}} (\{G\,\mathsf{a} \overset{?}{=} \mathsf{b}\}, \mathrm{id}) \xrightarrow{\mathsf{B(Imit)}} (\{G\,\mathsf{a} \overset{?}{=} \mathsf{b}\}, \{G \mapsto \lambda x.\,\mathsf{b}\})\\
&\xrightarrow{\mathsf{Dr{+}N}_\beta} (\{\mathsf{b} \overset{?}{=} \mathsf{b}\}, \{G \mapsto \lambda x.\,\mathsf{b}\}) \xrightarrow{\mathsf{Delete}} (\varnothing, \{G \mapsto \lambda x.\,\mathsf{b}\}) \xrightarrow{\mathsf{Succeed}} \{G \mapsto \lambda x.\,\mathsf{b}\}
\end{aligned}$$

$$\begin{aligned}
&(\{F\,(G\,\mathsf{a}) \overset{?}{=} F\,\mathsf{b}\}, \mathrm{id}) \xrightarrow{\mathsf{B(Elim)}} (\{F\,(G\,\mathsf{a}) \overset{?}{=} F\,\mathsf{b}\}, \{F \mapsto \lambda x.\,F'\})\\
&\xrightarrow{\mathsf{Dr{+}N}_\beta} (\{F' \overset{?}{=} F'\}, \{F \mapsto \lambda x.\,F'\}) \xrightarrow{\mathsf{Delete}} (\varnothing, \{F \mapsto \lambda x.\,F'\}) \xrightarrow{\mathsf{Succeed}} \{F \mapsto \lambda x.\,F'\}
\end{aligned}$$

The JP procedure additionally produces the following redundant unifier:

$$(F\,(G\,\mathsf{a}) \overset{?}{=} F\,\mathsf{b}, \mathrm{id}) \overset{\textsf{JP Proj}}{\Longrightarrow} (F\,\mathsf{a} = F\,\mathsf{b}, \{G \mapsto \lambda x.\, x\})$$
$$\overset{\textsf{Elim}}{\Longrightarrow} (F' = F', \{G \mapsto \lambda x.\, x, F \mapsto \lambda x.\, F'\}) \overset{\textsf{Succeed}}{\Longrightarrow} \{G \mapsto \lambda x.\, x, F \mapsto \lambda x.\, F'\}$$

Moreover, the JP procedure does not terminate because an infinite number of iterations is applicable at the root. Our procedure terminates in this case since we only apply iteration bindings for non base-type arguments, which F does not have.

Proof of Completeness. Like the JP procedure, our procedure misses no unifiers:

▶ **Theorem 4.** *The procedure described above is complete, meaning that the substitutions on the leaves of the constructed tree form a CSU. In other words, for any unifier ϱ of a multiset of constraints E there exists a derivation $(E, \mathrm{id}) \longrightarrow^* \sigma$ and a substitution θ such that $\varrho \subseteq \theta\sigma$.*

The proof of Theorem 4 is an adaptation of the proof given by JP [11]. Definitions and lemmas are reused, but they are combined together differently. The full proof is given in our technical report [27]. The backbone of the proof is as follows. We incrementally define states (E_j, σ_j) and *remainder substitutions* ϱ_j starting with $(E_0, \sigma_0) = (E, \mathrm{id})$ and $\varrho_0 = \varrho$. These will satisfy the invariants that ϱ_j unifies E_j and $\varrho_0 \subseteq \varrho_j\sigma_j$. Intuitively, ϱ_j is what remains to be added to σ_j to reach a unifier subsuming ϱ_0. In each step, ϱ_j guides the choice of the next transition $(E_j, \sigma_j) \longrightarrow (E_{j+1}, \sigma_{j+1})$.

To show that we eventually reach a state with an empty E_j, we employ a well-founded measure of (E_j, ϱ_j) that strictly decreases with each step. It is the lexicographic product of the syntactic size of $\varrho_j E_j$ and a measure on ϱ_j, which is taken from the JP proof.

Contrary to our procedure, the proof assumes that all terms are in η-long β-reduced form and that all substitutions are fully applied. These assumptions are justified because all bindings depend only on the head of terms and hence replacing the lazy transitions Normalize$_{\alpha\eta}$, Normalize$_\beta$, and Dereference by eager counterparts only affects the efficiency but not the overall behavior of our procedure.

Fix a state (E_j, σ_j). If E_j is empty, then a unifier σ_j of E is found by Succeed and we are done because $\varrho_0 \subseteq \varrho_j\sigma_j$ by the induction hypothesis. Otherwise, let $E_j = \{u \overset{?}{=} v\} \uplus E'_j$ where $u \overset{?}{=} v$ is selected. We must find a transition that reduces the measure and preserves the invariants. Fail and OracleFail cannot be applicable, because $\varrho_j u = \varrho_j v$ by the induction hypothesis. If applicable, Delete reduces the size of $\varrho_j E_j$ by removing a constraint.

OracleSucc has similar effect as Delete, but the remainder changes. Since ϱ_j is a unifier of $u \overset{?}{=} v$ and oracles compute CSUs, the oracle will find a unifier δ such that there exists a ϱ_{j+1} satisfying $\varrho_j \subseteq \varrho_{j+1}\,\delta$. Then $(E_{j+1}, \sigma_{j+1}) = \big(\delta E'_j, \delta\,\sigma_j\big)$ is a result of an OracleSucc transition. Observe that $\varrho_{j+1}E_{j+1} = \varrho_{j+1}\delta E'_j$ is a proper subset of $\varrho_j E_j$. Hence, the measure decreases and ϱ_{j+1} unifies E_{j+1}. The other invariant holds, because $\varrho_0 \subseteq \varrho_j\,\sigma_j \subseteq \varrho_{j+1}\,\delta\,\sigma_j = \varrho_{j+1}\,\sigma_{j+1}$.

If none of the previous transitions are applicable, we must find the right Decompose or Bind transition to apply. The choice is determined by the head a of u, the head b of v, and their values under ϱ_j. If $u \overset{?}{=} v$ is flex-rigid, then either $\varrho_j a$ has b as head symbol, enabling imitation, or $\varrho_j a$ has a bound variable as head symbol, enabling Huet-style projection. In the flex-flex case, if $a \neq b$, we apply either iteration, identification, or JP-style projection based on the form of $\varrho_j a$ and $\varrho_j b$. Similarly, if $a = b$, we apply either iteration, elimination, or Decompose guided by the form of $\varrho_j a$. To show preservation of the induction invariants for Bind, we determine a binding δ that can be factored out of ϱ_j as $\varrho_j \subseteq \varrho_{j+1}\,\delta$ similarly to the OracleSucc case. Here we have $\varrho_{j+1}E_{j+1} = \varrho_j E_j$; so we must ensure that the measure of ϱ_{j+1} is strictly smaller than that of ϱ_j. For Decompose, we set $\varrho_{j+1} = \varrho_j$ and show that $\varrho_{j+1}E_{j+1}$ is smaller than $\varrho_j E_j$.

Pragmatic Variant. We structured our procedure so that most of the unification machinery is contained in the Bind step. Modifying $\mathcal{P}$, we can sacrifice completeness and obtain a pragmatic variant of the procedure that often performs better in practice. Our preliminary experiments showed that $\mathcal{P}$ defined as follows is a reasonable compromise between completeness and performance:

- If the constraint is rigid-rigid, $\mathcal{P}(\lambda\overline{x}.\, s \overset{?}{=} \lambda\overline{x}.\, t) = \varnothing$.
- If the constraint is flex-rigid, let $\mathcal{P}(\lambda\overline{x}.\, F\,\overline{s} \overset{?}{=} \lambda\overline{x}.\, a\,\overline{t})$ be
  - an imitation of a for F, if a is some constant g, and
  - all Huet-style projections for F if F is not an identification variable.
- If the constraint is flex-flex and the heads are different, let $\mathcal{P}(\lambda\overline{x}.\, F\,\overline{s} \overset{?}{=} \lambda\overline{x}.\, G\,\overline{t})$ be
  - an identification binding for F and G, and
  - all Huet-style projections for F if F is not an identification variable
- If the constraint is flex-flex and the heads are identical, we distinguish two cases:
  - if the head is an elimination variable, $\mathcal{P}(\lambda\overline{x}.\, F\,\overline{s} \overset{?}{=} \lambda\overline{x}.\, F\,\overline{t}) = \varnothing$;
  - otherwise, $\mathcal{P}(\lambda\overline{x}.\, F\,\overline{s} \overset{?}{=} \lambda\overline{x}.\, F\,\overline{t})$ is the set of all eliminations bindings for F.

The pragmatic variant of our procedure removes all iteration bindings to enforce finite branching. Moreover, it imposes limits on the number of bindings applied, counting the applications of bindings locally, per constraint. It is useful to distinguish the Huet-style projection cases where α_i is a base type (called *simple projection*), which always reduces the problem size, and the cases where α_i is a functional type (called *functional projection*). We limit applications of the following bindings: functional projections, eliminations, imitations and identifications. In addition, a limit on the total number of applied bindings can be set. An elimination binding that removes k arguments counts as k elimination steps. Due to limits on application of bindings, the pragmatic variant terminates.

To fail as soon as any of the limits is reached, the pragmatic variant employs an additional oracle. If this oracle determines that the limits are reached and the constraint is of the form $\lambda\overline{x}.\, F\,\overline{s}_m \overset{?}{=} \lambda\overline{x}.\, G\,\overline{t}_n$, it returns a *trivial unifier* – a substitution $\{F \mapsto \lambda\overline{x}_m.\, H, G \mapsto \lambda\overline{x}_n.\, H\}$, where H is a fresh variable; if the limits are reached and the constraint is flex-rigid, the oracle fails; if the limits are not reached, it reports that terms are outside its fragment. The trivial unifier prevents the procedure from failing on easily unifiable flex-flex pairs.

Careful tuning of each limit optimizes the procedure for a specific class of problems. For problems originating from proof assistants, shallow unification depth usually suffices. However, hard hand-crafted problems often need deeper unification.

4 A New Decidable Fragment

We discovered a new fragment that admits a finite CSU and a simple oracle. The oracle is based on work by Prehofer and the PT procedure [18], a modification of Huet's procedure. PT transforms an initial multiset of constraints E_0 by applying bindings ϱ. If there is a sequence $E_0 \Longrightarrow^{\varrho_1} \cdots \Longrightarrow^{\varrho_n} E_n$ such that E_n has only flex-flex constraints, we say that PT produces a preunifier $\sigma = \varrho_n \ldots \varrho_1$ with constraints E_n. A sequence fails if $E_n = \bot$. Unlike previously, in this section we consider all terms to be $\alpha\beta\eta$-equivalence classes with the η-long β-reduced form as their canonical representative and we view unification constraints $s \overset{?}{=} t$ as ordered pairs.

The following rules, however, are stated modulo orientation. The PT transition rules, adapted for our presentation style, are as follows:

Deletion $\{s \overset{?}{=} s\} \uplus E \Longrightarrow^{\mathrm{id}} E$

Decomposition $\{\lambda\overline{x}.\, a\,\overline{s}_m \overset{?}{=} \lambda\overline{x}.\, a\,\overline{t}_m\} \uplus E \Longrightarrow^{\mathrm{id}} \{s_1 \overset{?}{=} t_1, \ldots, s_m \overset{?}{=} t_m\} \uplus E$
where a is rigid

Failure $\{\lambda\overline{x}.\, a\,\overline{s} \overset{?}{=} \lambda\overline{x}.\, b\,\overline{t}\} \uplus E \Longrightarrow^{\mathrm{id}} \bot$
where a and b are different rigid heads

Solution $\{\lambda\overline{x}.\, F\,\overline{x} \overset{?}{=} \lambda\overline{x}.\, t\} \uplus E \Longrightarrow^{\varrho} \varrho(E)$
where F does not occur in t, t does not have a flex head, and $\varrho = \{F \mapsto \lambda\overline{x}.\, t\}$

Imitation $\{\lambda\overline{x}.\, F\,\overline{s}_m \overset{?}{=} \lambda\overline{x}.\, \mathsf{f}\,\overline{t}_n\} \uplus E \Longrightarrow^{\varrho} \varrho(\{G_1\,\overline{s}_m \overset{?}{=} t_1, \ldots, G_n\,\overline{s}_m \overset{?}{=} t_n\} \uplus E)$
where $\varrho = \{F \mapsto \lambda\overline{x}_m.\, \mathsf{f}\,(G_1\,\overline{x}_m) \ldots (G_n\,\overline{x}_m)\}$, $\overline{G}_n$ are fresh variables of appropriate types

Projection $\{\lambda\overline{x}.\, F\,\overline{s}_m \overset{?}{=} \lambda\overline{x}.\, a\,\overline{t}\} \uplus E \Longrightarrow^{\varrho} \varrho(\{s_i\,(G_1\,\overline{s}_m) \ldots (G_j\,\overline{s}_m) \overset{?}{=} a\,\overline{t}\} \uplus E)$
where $\varrho = \{F \mapsto \lambda\overline{x}_m.\, x_i\,(G_1\,\overline{x}_m) \ldots (G_j\,\overline{x}_m)\}$, $\overline{G}_j$ are fresh variables of appropriate types

The grayed constraints are required to be selected by a given selection function S. We call S *admissible* if it prioritizes selection of constraints applicable for Failure and Decomposition, and of descendant constraints of Projection transitions with $j = 0$ (i.e., for x_i of base type), in that order of priority. In the remainder of this section we consider only admissible selection functions, an assumption that Prehofer also makes implicitly in his thesis.

Prehofer showed that PT terminates for some classes of constraints. We call a term *linear* if no free variable has repeated occurrences in it. We call a term *solid* if its free variables are applied either to bound variables or ground base-type terms. We call it *strictly solid* if its free variables are applied either to bound variables or second-order ground base-type terms. For example, if G, a, and x are of base type, and F, H, g, and y are binary, the terms $F\,G\,\mathsf{a}$, and $H\,(\lambda x.\, x)\,\mathsf{a}$ are not solid; $\lambda x.\, F\,x\,x$ is strictly solid; $F\,\mathsf{a}\,(\mathsf{g}\,(\lambda y.\, y\,\mathsf{a}\,\mathsf{a})\,\mathsf{a})$ is solid, but not strictly. Prehofer's thesis states that PT terminates on $\{s \overset{?}{=} t\}$ if s is linear, s shares no free variables with t, s is strictly solid, and t is second-order.

We extend this result in Theorem 8 along two axes: we create an oracle for the full unification problem, and we lift some order constraints by requiring s and t to be solid. Lemma 5 lifts Prehofer's preunification result to solid terms:

▶ **Lemma 5.** *If s and t are solid, s is linear and shares no free variables with t, PT terminates for the preunification problem $\{s \overset{?}{=} t\}$, and all remaining flex-flex constraints are solid.*

Enumerating a CSU for a solid flex-flex pair may seem as hard as for any other flex-flex pair; however, the following two lemmas show that solid pairs admit an MGU:

▶ **Lemma 6.** *The unification problem $\{\lambda\overline{x}.\, F\,\overline{s}_m \overset{?}{=} \lambda\overline{x}.\, F\,\overline{t}_m\}$, where both terms are solid, has an MGU of the form $\sigma = \{F \mapsto \lambda\overline{x}_m.\, G\,x_{j_1} \ldots x_{j_r}\}$ where G is a fresh variable, and $1 \leq j_1 < \cdots < j_r \leq m$ are exactly those indices j_i for which $s_{j_i} = t_{j_i}$.*

▶ **Lemma 7.** *Let $\{\lambda\overline{x}.\, F\,\overline{s}_m \overset{?}{=} \lambda\overline{x}.\, G\,\overline{t}_n\}$ be a solid unification problem where $F \neq G$. Then there is a finite CSU $\{\sigma_i^1, \ldots, \sigma_i^{k_i}\}$ of the problem $\{s_i \overset{?}{=} H_i\,\overline{t}_n\}$, where H_i is a fresh free variable. Let $\lambda\overline{y}_n.\, s_i^j = \lambda\overline{y}_n.\, \sigma_i^j(H_i)\,\overline{y}_n$. Similarly, there is a finite CSU $\{\tilde{\sigma}_i^1, \ldots, \tilde{\sigma}_i^{l_i}\}$ of the problem $\{t_i \overset{?}{=} \tilde{H}_i\,\overline{s}_m\}$, where $\tilde{H}_i$ is a fresh free variable. Let $\lambda\overline{x}_m.\, t_i^j = \lambda\overline{x}_m.\, \tilde{\sigma}_i^j(\tilde{H}_i)\,\overline{x}_m$. Let Z be a fresh free variable. An MGU σ for the given problem is*

$$F \mapsto \lambda\overline{x}_m.\, Z\, \underbrace{x_1 \ldots x_1}_{k_1 \textit{ times}} \ldots \underbrace{x_m \ldots x_m}_{k_m \textit{ times}}\, t_1^1 \ldots t_1^{l_1} \ldots t_n^1 \ldots t_n^{l_n}$$

$$G \mapsto \lambda\overline{y}_n.\, Z\, s_1^1 \ldots s_1^{k_1} \ldots s_m^1 \ldots s_m^{k_m}\, \underbrace{y_1 \ldots y_1}_{l_1 \textit{ times}} \ldots \underbrace{y_n \ldots y_n}_{l_n \textit{ times}}$$

Our proof that finite CSUs exist relies on Prehofer's proof that PT terminates without producing flex-flex pairs for the matching problem $\{\lambda\overline{x}_n.\, F\,\overline{s}_k \overset{?}{=} \lambda\overline{x}_n.\, t\}$ where $F\,\overline{s}_k$ is strictly solid and t is ground and second-order. His proof is easily generalized to the case where t is arbitrary order and $F\,\overline{s}_k$ is solid. Since PT is complete, we conclude that such problems have finite CSUs.

▶ **Theorem 8.** *Let s and t be solid terms that share no free variables, and let s be linear. Then the unification problem $\{s \overset{?}{=} t\}$ has a finite CSU.*

This CSU is straightforward to compute. By Lemma 5, PT terminates on $\{s \overset{?}{=} t\}$ with a finite set of preunifiers σ, each associated with a multiset E of solid flex-flex pairs. An MGU δ_E of E can be found as follows. Choose a constraint $(u \overset{?}{=} v) \in E$ and determine an MGU ϱ for it using Lemma 6 or 7. Then the set $\varrho(E \setminus \{u \overset{?}{=} v\})$ also contains only solid flex-flex constraints, and we iterate this process by choosing a constraint from $\varrho(E \setminus \{u \overset{?}{=} v\})$ next until there are no constraints left, eventually yielding an MGU ϱ' of $\varrho(E \setminus \{u \overset{?}{=} v\})$. Finally, let $\delta_E = \varrho'\varrho$. Then $\{\delta_E\sigma \mid \text{PT produces preunifier } \sigma \text{ with constraints } E\}$ is a finite CSU.

▶ **Example 9.** For example, let $\{F\,(\mathsf{f}\,\mathsf{a}) \overset{?}{=} \mathsf{g}\,\mathsf{a}\,(G\,\mathsf{a})\}$ be the unification problem to solve. Projecting F onto the first argument will lead to a nonunifiable problem, so we perform imitation of g building a binding $\sigma_1 = \{F \mapsto \lambda x.\, \mathsf{g}\,(F_1\,x)\,(F_2\,x)\}$. This yields the problem $\{F_1\,(\mathsf{f}\,\mathsf{a}) \overset{?}{=} \mathsf{a}, F_2\,(\mathsf{f}\,\mathsf{a}) \overset{?}{=} G\,\mathsf{a}\}$. Again, we can only imitate a for F_1 – building a new binding $\sigma_2 = \{F_1 \mapsto \lambda x.\, \mathsf{a}\}$. Finally, this yields the problem $\{F_2\,(\mathsf{f}\,\mathsf{a}) \overset{?}{=} G\,\mathsf{a}\}$. According to Lemma 7, we find CSUs for the problems $J_1\,\mathsf{a} = \mathsf{f}\,\mathsf{a}$ and $I_1\,(\mathsf{f}\,\mathsf{a}) \overset{?}{=} \mathsf{a}$ using PT. The latter problem has a singleton CSU $\{I_1 \mapsto \lambda x.\, \mathsf{a}\}$, whereas the former has a CSU containing $\{J_1 \mapsto \lambda x.\, \mathsf{f}\,x\}$ and $\{J_1 \mapsto \lambda x.\, \mathsf{f}\,\mathsf{a}\}$. Combining these solutions, we obtain an MGU $\sigma_3 = \{F_2 \mapsto \lambda x.\, H\,x\,x\,\mathsf{a},\, G \mapsto \lambda x.\, H\,(\mathsf{f}\,\mathsf{a})\,(\mathsf{f}\,x)\,x\}$ for $F_2\,(\mathsf{f}\,\mathsf{a}) \overset{?}{=} G\,\mathsf{a}$. Finally, we get the MGU $\sigma = \sigma_3\sigma_2\sigma_1 = \{F \mapsto \lambda x.\, \mathsf{g}\,\mathsf{a}\,(H\,x\,x\,\mathsf{a}),\, G \mapsto \lambda x.\, H\,(\mathsf{f}\,\mathsf{a})\,(\mathsf{f}\,x)\,x\}$ of the original problem.

Small examples that violate conditions of Theorem 8 and admit only infinite CSUs can be found easily. The problem $\{\lambda x.\, F\,(\mathsf{f}\,x) \overset{?}{=} \lambda x.\, \mathsf{f}\,(F\,x)\}$ violates variable distinctness and is a well-known example of a problem with only infinite CSUs. Similarly, $\lambda x.\, \mathsf{g}\,(F\,(\mathsf{f}\,x))\,F \overset{?}{=} \lambda x.\, \mathsf{g}\,(\mathsf{f}\,(G\,x))\,G$, which violates linearity, reduces to the previous problem. Only ground arguments to free variables are allowed because $\{F\,X \overset{?}{=} G\,\mathsf{a}\}$ has only infinite CSUs. Finally, it is crucial that functional arguments to free variables are only bound variables: the problem $\{\lambda y.\, X(\lambda x.\, x)\,y \overset{?}{=} \lambda y.\, y\}$ has only infinite CSUs.

5 An Extension of Fingerprint Indexing

A fundamental building block for almost all automated reasoning tools is the operation of retrieving term pairs that satisfy certain conditions, e.g., unifiable terms, instances or generalizations. Indexing data structures are used to implement this operation efficiently. If the data structure retrieves precisely the terms that satisfy the condition it is called *perfect*.

Higher-order indexing has received little attention, compared to its first-order counterpart. However, recent research in higher-order theorem proving increased the interest in higher-order indexing [2,14]. A *fingerprint index* [20,28] is an imperfect index based on the idea that the skeleton of the term consisting of all non-variable positions is not affected by substitutions. Therefore, we can easily determine that the terms are not unifiable (or matchable) if they disagree on a fixed set of sample positions.

More formally, when we sample an untyped first-order term t on a sample position p, the *generic fingerprinting function* gfpf distinguishes four possibilities:

$$\mathrm{gfpf}(t, p) = \begin{cases} \mathsf{f} & \text{if } t|_p \text{ has a symbol head } \mathsf{f} \\ \mathsf{A} & \text{if } t|_p \text{ is a variable} \\ \mathsf{B} & \text{if } t|_q \text{ is a variable for some proper prefix } q \text{ of } p \\ \mathsf{N} & \text{otherwise} \end{cases}$$

We define the *fingerprinting function* $\mathrm{fp}(t) = (\mathrm{gfpf}(t, p_1), \ldots, \mathrm{gfpf}(t, p_n))$, based on a fixed tuple of positions $\overline{p}_n$. Determining whether two terms are compatible for a given retrieval operation reduces to checking their fingerprints' componentwise compatibility. The following matrices determine the compatibility for retrieval operations:

	f_1	f_2	A	B	N
f_1		✗			✗
A					✗
B					
N	✗	✗	✗		

	f_1	f_2	A	B	N
f_1		✗	✗	✗	✗
A				✗	✗
B					
N	✗	✗	✗	✗	

The left matrix determines unification compatibility, while the right matrix determines compatibility for matching term s (rows) onto term t (columns). Symbols f_1 and f_2 stand for arbitrary distinct constants. Incompatible features are marked with ✗. For example, given a tuple of term positions $(1, 1.1.1, 2)$, and terms $\mathsf{f}(\mathsf{g}(X), \mathsf{b})$ and $\mathsf{f}(\mathsf{f}(\mathsf{a}, \mathsf{a}), \mathsf{b})$, their fingerprints are $(\mathsf{g}, \mathsf{B}, \mathsf{b})$ and $(\mathsf{f}, \mathsf{N}, \mathsf{b})$, respectively. Since the first fingerprint component is incompatible, terms are not unifiable.

Fingerprints for the terms in the index are stored in a trie data structure. This allows us to efficiently filter out terms that are not compatible with a given retrieval condition. For the remaining terms, a unification or matching procedure must be invoked to determine whether they satisfy the condition or not.

The fundamental idea of first-order fingerprint indexing carries over to higher-order terms – application of a substitution does not change the rigid skeleton of a term. However, to extend fingerprint indexing to higher-order terms, we must address the issues of $\alpha\beta\eta$-normalization and figure how to cope with λ-abstractions and bound variables. To that end, we define a function $\lfloor t \rfloor$, defined on β-reduced terms in De Bruijn [4] notation:

$$\lfloor F\,\overline{s} \rfloor = F \qquad \lfloor x_i\,\overline{s}_n \rfloor = \mathsf{db}_i^{\alpha}(\lfloor s_1 \rfloor, \ldots, \lfloor s_n \rfloor) \qquad \lfloor \mathsf{f}\,\overline{s}_n \rfloor = \mathsf{f}(\lfloor s_1 \rfloor, \ldots, \lfloor s_n \rfloor) \qquad \lfloor \lambda\overline{x}.\, s \rfloor = \lfloor s \rfloor$$

We let x_i be a bound variable of type α with De Bruijn index i, and db_i^{α} be a fresh constant corresponding to this variable. All constants db_i^{α} must be fresh. Effectively, $\lfloor\ \rfloor$ transforms a η-long β-reduced higher-order term to an untyped first-order term. Let $t_{\downarrow\beta\eta}$ be the η-long β-reduced form of t; the higher-order generic fingerprinting function $\mathrm{gfpf}_{\mathsf{ho}}$, which relies on conversion $\langle t \rangle_{\mathsf{db}}$ from named to De Bruijn representation, is defined as

$$\mathrm{gfpf}_{\mathsf{ho}}(t, p) = \mathrm{gfpf}(\lfloor \langle t_{\downarrow\beta\eta} \rangle_{\mathsf{db}} \rfloor, p)$$

If we define $\mathrm{fp}_{\mathsf{ho}}(t) = \mathrm{fp}(\lfloor \langle t_{\downarrow\beta\eta} \rangle_{\mathsf{db}} \rfloor)$, we can support fingerprint indexing for higher-order terms with no changes to the compatibility matrices. For example, consider the terms $s = (\lambda xy.\, x\, y)\, \mathsf{g}$ and $t = \mathsf{f}$, where g has the type $\alpha \to \beta$ and f has the type $\alpha \to \alpha \to \beta$. For the tuple of positions $(1, 1.1.1, 2)$ we get

$$\mathrm{fp}_{\mathsf{ho}}(s) = \mathrm{fp}(\lfloor \langle s_{\downarrow\beta\eta} \rangle_{\mathsf{db}} \rfloor) = \mathrm{fp}(\mathsf{g}(\mathsf{db}_0^{\alpha})) = (\mathsf{db}_0^{\alpha}, \mathsf{N}, \mathsf{N})$$
$$\mathrm{fp}_{\mathsf{ho}}(t) = \mathrm{fp}(\lfloor \langle t_{\downarrow\beta\eta} \rangle_{\mathsf{db}} \rfloor) = \mathrm{fp}(\mathsf{f}(\mathsf{db}_1^{\alpha}, \mathsf{db}_0^{\alpha})) = (\mathsf{db}_1^{\alpha}, \mathsf{N}, \mathsf{db}_0^{\alpha})$$

Since the first and third fingerprint component are incompatible, the terms are not unifiable.

Other first-order indexing techniques such as feature vector indexing and substitution trees can probably be extended to higher-order terms using the method described here as well.

6 Implementation

Zipperposition [5,6] is an open-source[1] theorem prover written in OCaml. It is a versatile testbed for prototyping extensions to superposition-based theorem provers. It was initially designed as a prover for polymorphic first-order logic and then extended to higher-order logic. The most recent addition is a complete mode for Boolean-free higher-order logic [1], which depends on a unification procedure that can enumerate a CSU. We implemented our procedure in Zipperposition.

We used OCaml's functors to create a modular implementation. The core of our procedure is implemented in a module which is parametrized by another module providing oracles and implementing the Bind step. In this way we can obtain the full or pragmatic procedure and seamlessly integrate oracles while reusing as much common code as possible.

To enumerate all elements of a possibly infinite CSU, we rely on lazy lists whose elements are subsingletons of unifiers (either one-element sets containing a unifier or empty sets). The search space must be explored in a *fair* manner, meaning that no branch of the constructed tree is indefinitely postponed.

Each Bind step will give rise to new unification problems $E_1, E_2, \ldots$ to be solved. Solutions to each of those problems are lazy lists $p_1, p_2, \ldots$ containing subsingletons of unifiers. To avoid postponing some unifier indefinitely, we use the dovetailing technique: we first take one subsingleton from p_1, then one from each of p_1 and p_2. We continue with one subsingleton from each of p_1, p_2 and p_3, and so on. Empty lazy lists are ignored in the traversal. To ensure we do not remain stuck waiting for a unifier from a particular lazy list, the procedure will periodically return an empty set, indicating that the next lazy list should be probed.

The implemented selection function for our procedure prioritizes selection of rigid-rigid over flex-rigid pairs, and flex-rigid over flex-flex pairs. However, since the constructed substitution σ is not applied eagerly, heads can appear to be flex, even if they become rigid after dereferencing and normalization. To mitigate this issue to some degree, we dereference the heads with σ, but do not normalize, and use the resulting heads for prioritization.

We implemented oracles for the pattern, solid, and fixpoint fragment. Fixpoint unification [10] is concerned with problems of the form $\{F \overset{?}{=} t\}$. If F does not occur in t, $\{F \mapsto t\}$ is an MGU for the problem. If there is a position p in t such that $t|_p = F\,\overline{u}_m$ and for each prefix $q \neq p$ of p, $t|_q$ has a rigid head and either $m = 0$ or t is not a λ-abstraction, then we can conclude that $F \overset{?}{=} t$ has no solutions. Otherwise, the fixpoint oracle is not applicable.

7 Evaluation

We evaluated the implementation of our unification procedure in Zipperposition, assessing a complete variant and a pragmatic variant, the latter with several different combinations of limits for number of bindings. As part of the implementation of the complete mode for Boolean-free higher-order logic in Zipperposition [1], Bentkamp implemented a straightforward version of JP procedure. This version is faithful to the original description, with a check as

[1] `https://github.com/sneeuwballen/zipperposition`

	jp	cv	pv^{12}_{6666}	pv^{6}_{3333}	pv^{4}_{2222}	pv^{2}_{1222}	pv^{2}_{1121}	pv^{2}_{1020}
TPTP	1551	1717	1722	**1732**	**1732**	1715	1712	1719
SH	242	**260**	253	255	255	254	259	257

Figure 1 Proved problems, per configuration.

	n	f	p	s	fp	fs	ps	fps
TPTP	1658	1717	1717	1720	1719	**1724**	1720	1723
SH	245	255	**260**	259	255	254	258	254

Figure 2 Proved problems, per used oracle.

to whether a (sub)problem can be solved using a first-order oracle as the only optimization. Our evaluations were performed on StarExec Miami [23] servers with Intel Xeon E5-2620 v4 CPUs clocked at 2.10 GHz with 60 s CPU limit.

Contrary to first-order unification, there is no widely available corpus of benchmarks dedicated solely to evaluating performance of higher-order unification algorithms. Thus, we used all 2606 monomorphic higher-order theorems from the TPTP library [25] and 832 monomorphic higher-order Sledgehammer (SH) generated problems [24] as our benchmarks[2]. Many TPTP problems require synthesis of complicated unifiers, whereas Sledgehammer problems are only mildly higher-order – many of them are solved with first-order unifiers.

We used the naive implementation of the JP procedure (**jp**) as a baseline to evaluate the performance of our procedure. We compare it with the complete variant of our procedure (**cv**) and pragmatic variants (**pv**) with several different configurations of limits for applied bindings. All other Zipperposition parameters have been fixed to the values of a variant of a well-performing configuration we used for the CASC-27 theorem proving competition [26]. The cv configuration and all of the pv configurations use only pattern unification as an underlying oracle. To test the effect of oracle choice, we evaluated the complete variant in 8 combinations: with no oracles (**n**), with only fixpoint (**f**), pattern (**p**), or solid (**s**) oracle, and with their combinations: **fp**, **fs**, **ps**, **fps**.

Figure 1 compares different variants of the procedure with the naive JP implementation. Each pv configuration is denoted by pv^{a}_{bcde} where a is the limit on the total number of applied bindings, and b, c, d, and e are the limits of functional projections, eliminations, imitations, and identifications, respectively. Figure 2 summarizes the effects of using different oracles.

The configuration of our procedure with no oracles outperforms the JP procedure with the first-order oracle. This suggests that the design of the procedure, in particular lazy normalization and lazy application of the substitution, already reduces the effects of the JP procedure's main bottlenecks. Raw evaluation data shows that on TPTP benchmarks, complete and pragmatic configurations differ in the set of problems they solve – cv solves 19 problems not solved by pv^{4}_{2222}, whereas pv^{4}_{2222} solves 34 problems cv does not solve. Similarly, comparing the pragmatic configurations with each other, pv^{6}_{3333} and pv^{4}_{2222} each solve 13 problems that the other one does not. The overall higher success rate of pv^{2}_{1020} compared to pv^{2}_{1222} suggests that solving flex-flex pairs by trivial unifiers often suffices for superposition-based theorem proving.

[2] An archive with raw results, all used problems, and scripts for running each configuration is available at `http://matryoshka.gforge.inria.fr/pubs/hounif_data.zip`.

Counterintuitively, in some cases, using oracles can hurt the performance of Zipperposition. Using oracles typically results in generating smaller CSUs, whose elements are more general substitutions than the ones we obtain without oracles. These more general substitutions usually contain more applied variables, which Zipperposition's heuristics avoid due to their explosive nature. This can make Zipperposition postpone necessary inferences for too long. Configuration n benefits from this effect and therefore solves 18 TPTP problems that no other configuration in Figure 2 solves. The same effect also gives configurations with only one oracle an advantage over configurations with multiple oracles on some problems.

The evaluation sheds some light on how often solid unification problems appear in practice. The raw data show that configuration s solves 5 TPTP problems that neither f nor p solve. Configuration f solves 8 TPTP problems that neither s nor p solve, while p solves 9 TPTP problems that two other configurations do not. This suggests that the solid oracle is slightly less beneficial than the fixpoint or pattern oracles, but still presents a useful addition the set of available oracles.

A subset of 11 TPTP benchmarks, concerning operations on Church numerals, is designed to test the efficiency of higher-order unification. Our procedure performs exceptionally well on these problems – it solves all of them, usually faster than other competitive higher-order provers. In particular, on two of these problems, neither Leo-III 1.4 nor Satallax 3.4 produce a proof within a 60 seconds CPU limit, while the cv configuration proves each of them in less than 4.5 s. A full list of these problems is in our technical report [27].

8 Discussion and Related Work

The problem addressed in this paper is that of finding a complete and efficient higher-order unification procedure. Three main lines of research dominated the research field of higher-order unification over the last forty years.

The first line of research went in the direction of finding procedures that enumerate CSUs. The most prominent procedure designed for this purpose is the JP procedure [11]. Snyder and Gallier [21] also provide such a procedure, but instead of solving flex-flex pairs systematically, their procedure blindly guesses the head of the necessary binding by considering all constants in the signature and fresh variables of all possible types. Another approach, based on higher-order combinators, is given by Dougherty [7]. This approach blindly creates (partially applied) S-, K-, and I-combinator bindings for applied variables, which results in returning many redundant unifiers, as well as in nonterminating behavior even for simple problems such as $X\ \mathsf{a} = \mathsf{a}$.

The second line of research is concerned with enumerating preunifiers. The most prominent procedure in this line of research is Huet's [10]. The Snyder–Gallier procedure restricted not to solve flex-flex pairs is a version of the PT procedure presented in Section 4. It improves Huet's procedure by featuring a Solution rule.

The third line of research gives up the expressiveness of the full λ-calculus and focuses on decidable fragments. Patterns [16] are arguably the most important such fragment in practice, with implementations in Isabelle [17], Leo-III [22], Satallax [3], λProlog [15], and other systems. Functions-as-constructors [13] unification subsumes pattern unification but is significantly more complex to implement. Prehofer [18] lists many other decidable fragments, not only for unification but also preunification and unifier existence problems. Most of these algorithms are given for second-order terms with various constraints on their variables. Finally, one of the first decidability results is Farmer's discovery [8] that higher-order unification of terms with unary function symbols is decidable.

Our procedure draws inspiration from and contributes to all three lines of research. Accordingly, its advantages over previously known procedures can be laid out along those three lines. First, our procedure mitigates many issues of the JP procedure. Second, it can be modified not to solve flex-flex pairs, and become a version of Huet's procedure with important built-in optimizations. Third, it can integrate any oracle for problems with finite CSUs – including the one we discovered.

9 Conclusion

We presented a procedure for enumerating a complete set of higher-order unifiers that is designed for efficiency. Due to design that restricts search space and tight integration of oracles it reduces the number of redundant unifiers returned and gives up early in cases of nonunifiability. In addition, we presented a new fragment of higher-order terms that admits finite CSUs. Our evaluation shows a clear improvement over previously known procedure.

In future work, we will focus on designing intelligent heuristics that automatically adjust unification parameters according to the type of the problem. For example, we should usually choose shallow unification for mostly first-order problems and deeper unification for hard higher-order problems. We plan to investigate other heuristic choices, such as the order of bindings and the way in which search space is traversed (breadth- or depth-first). We are also interested in further improving the termination behavior of the procedure, without sacrificing completeness. Finally, following the work of Libal [12] and Zaionc [29], we would like to consider the use of regular grammars to finitely present infinite CSUs. For example, the grammar $G ::= \lambda x.\, x \mid \lambda x.\, \mathsf{f}\,(G\, x)$ represents all elements of the CSU for the problem $\lambda x.\, G\,(\mathsf{f}\, x) \overset{?}{=} \lambda x.\, \mathsf{f}\,(G\, x)$.

References

1 Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, Petar Vukmirović, and Uwe Waldmann. Superposition with lambdas. In Pascal Fontaine, editor, *CADE-27*, volume 11716 of *LNCS*, pages 55–73. Springer, 2019.

2 Ahmed Bhayat and Giles Reger. Restricted combinatory unification. In Pascal Fontaine, editor, *CADE-27*, volume 11716 of *LNCS*, pages 74–93. Springer, 2019.

3 Chad E. Brown. Satallax: An automatic higher-order prover. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR 2012*, volume 7364 of *LNCS*, pages 111–117. Springer, 2012.

4 Nicolaas G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *J. Symb. Log.*, 40(3):470–470, 1975.

5 Simon Cruanes. *Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond.* PhD thesis, École polytechnique, 2015.

6 Simon Cruanes. Superposition with structural induction. In Clare Dixon and Marcelo Finger, editors, *FroCoS 2017*, volume 10483 of *LNCS*, pages 172–188. Springer, 2017.

7 Daniel J. Dougherty. Higher-order unification via combinators. *Theor. Comput. Sci.*, 114(2):273–298, 1993.

8 William M. Farmer. A unification algorithm for second-order monadic terms. *Ann. Pure Appl. Logic*, 39(2):131–174, 1988.

9 Krystof Hoder and Andrei Voronkov. Comparing unification algorithms in first-order theorem proving. In Bärbel Mertsching, Marcus Hund, and Muhammad Zaheer Aziz, editors, *KI 2009: Advances in Artificial Intelligence, 32nd Annual German Conference on AI, Paderborn, Germany, September 15-18, 2009. Proceedings*, volume 5803 of *Lecture Notes in Computer Science*, pages 435–443. Springer, 2009.

10 Gérard P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975.

11 Don C. Jensen and Tomasz Pietrzykowski. Mechanizing omega-order type theory through unification. *Theor. Comput. Sci.*, 3(2):123–171, 1976.

12 Tomer Libal. Regular patterns in second-order unification. In Amy P. Felty and Aart Middeldorp, editors, *CADE-25*, volume 9195 of *LNCS*, pages 557–571. Springer, 2015.

13 Tomer Libal and Dale Miller. Functions-as-constructors higher-order unification. In Delia Kesner and Brigitte Pientka, editors, *FSCD 2016*, volume 52 of *LIPIcs*, pages 26:1–26:17. Schloss Dagstuhl, 2016.

14 Tomer Libal and Alexander Steen. Towards a substitution tree based index for higher-order resolution theorem provers. In Pascal Fontaine, Stephan Schulz, and Josef Urban, editors, *IJCAR 2016*, volume 1635 of *CEUR-WS*, pages 82–94. CEUR-WS, 2016.

15 Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012.

16 Tobias Nipkow. Functional unification of higher–order patterns. In E. Best, editor, *LICS '93*, pages 64–74. IEEE Computer Society, 1993.

17 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

18 Christian Prehofer. *Solving higher order equations: from logic to programming*. PhD thesis, Technical University Munich, Germany, 1995.

19 John Alan Robinson. A machine–oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

20 Stephan Schulz. Fingerprint indexing for paramodulation and rewriting. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR 2012*, volume 7364 of *LNCS*, pages 477–483. Springer, 2012.

21 Wayne Snyder and Jean H. Gallier. Higher-order unification revisited: Complete sets of transformations. *J. Symb. Comput.*, 8(1/2):101–140, 1989.

22 Alexander Steen and Christoph Benzmüller. The higher–order prover Leo-III. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *IJCAR 2018*, volume 10900 of *LNCS*, pages 108–116. Springer, 2018.

23 Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. Starexec: A cross-community infrastructure for logic solving. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *IJCAR 2014*, volume 8562 of *LNCS*, pages 367–373. Springer, 2014.

24 Nik Sultana, Jasmin Christian Blanchette, and Lawrence C. Paulson. LEO-II and Satallax on the Sledgehammer test bench. *J. Applied Logic*, 11(1):91–102, 2013.

25 Geoff Sutcliffe. The TPTP problem library and associated infrastructure - from CNF to TH0, TPTP v6.4.0. *J. Autom. Reasoning*, 59(4):483–502, 2017.

26 Geoff Sutcliffe. The CADE-27 automated theorem proving system competition - CASC-27. *AI Commun.*, 32(5-6):373–389, 2019.

27 Petar Vukmirović, Alexander Bentkamp, and Visa Nummelin. Efficient full higher-order unification (technical report), 2020. URL: http://matryoshka.gforge.inria.fr/pubs/hounif_report.pdf.

28 Petar Vukmirović, Jasmin Christian Blanchette, Simon Cruanes, and Stephan Schulz. Extending a brainiac prover to lambda-free higher-order logic. In Tomás Vojnar and Lijun Zhang, editors, *TACAS 2019*, volume 11427 of *LNCS*, pages 192–210. Springer, 2019.

29 Marek Zaionc. The set of unifiers in typed lambda-calculus as regular expression. In Jean-Pierre Jouannaud, editor, *RTA-85*, volume 202 of *LNCS*, pages 430–440. Springer, 1985.

Comprehension and Quotient Structures in the Language of 2-Categories

Paul-André Melliès

CNRS, Institut de Recherche en Informatique Fondamentale (IRIF), Université de Paris, France
`http://www.irif.fr/~mellies`
mellies@irif.fr

Nicolas Rolland

Institut de Recherche en Informatique Fondamentale (IRIF), Université de Paris, France
nicolas.rolland@gmail.com

Abstract

Lawvere observed in his celebrated work on hyperdoctrines that the set-theoretic schema of comprehension can be elegantly expressed in the functorial language of categorical logic, as a comprehension structure on the functor $p : \mathcal{E} \to \mathcal{B}$ defining the hyperdoctrine. In this paper, we formulate and study a strictly ordered hierarchy of three notions of comprehension structure on a given functor $p : \mathcal{E} \to \mathcal{B}$, which we call (i) comprehension structure, (ii) comprehension structure with section, and (iii) comprehension structure with image. Our approach is 2-categorical and we thus formulate the three levels of comprehension structure on a general morphism $p : \mathbf{E} \to \mathbf{B}$ in a 2-category $\mathcal{K}$. This conceptual point of view on comprehension structures enables us to revisit the work by Fumex, Ghani and Johann on the duality between comprehension structures and quotient structures on a given functor $p : \mathcal{E} \to \mathcal{B}$. In particular, we show how to lift the comprehension and quotient structures on a functor $p : \mathcal{E} \to \mathcal{B}$ to the categories of algebras or coalgebras associated to functors $F_{\mathcal{E}} : \mathcal{E} \to \mathcal{E}$ and $F_{\mathcal{B}} : \mathcal{B} \to \mathcal{B}$ of interest, in order to interpret reasoning by induction and coinduction in the traditional language of categorical logic, formulated in an appropriate 2-categorical way.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Proof theory; Theory of computation → Logic and verification; Theory of computation → Linear logic

Keywords and phrases Comprehension structures, quotient structures, comprehension structures with section, comprehension structures with image, 2-categories, formal adjunctions, path objects, categorical logic, inductive reasoning on algebras, coinductive reasoning on coalgebras

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.6

Funding The research underlying this article was partially supported by the ERC Advanced Grant DuaLL, number 670624.
Paul-André Melliès: ERC Advanced Project DuaLL, Grant agreement ID: 670624.

1 Introduction

A fundamental duality between comprehension and quotient structures

One fundamental discovery by Lawvere [14] is that the *comprehension schema* of Zermelo set theory [15] can be elegantly expressed in the functorial language of categorical logic, in the following way. Consider the category **Set** of sets and functions, and the category **Pred** of predicates, defined in the following way: its objects are the pairs (A, R) consisting of a set A and of a function $R : A \to \Omega$ to the set $\Omega = \{\text{false}, \text{true}\}$ of booleans, describing a specific predicate R of A ; its morphisms $f : (A, R) \to (B, S)$ are the functions $f : A \to B$ such that $\forall a \in A, Ra \Rightarrow S(fa)$. The functor $p : \mathbf{Pred} \to \mathbf{Set}$ is the forgetful functor which transports every predicate (A, R) to its underlying set A. The comprehension schema enables one to

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 6; pp. 6:1–6:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

turn every predicate (A,R) into a set $[A,R]$ defined as follows

$$[A,R] \ := \ \{ \ a \in A \mid Ra = \text{true} \ \}$$

equipped moreover with a function

$$\iota_{A,R} \quad : \quad [A,R] \longrightarrow A$$

which transports every element a of the set $[A,R]$ to itself, seen as element in A. The construction is natural in (A,R) in the sense that it defines a functor

$$[-] \quad : \quad \mathbf{Pred} \longrightarrow \mathbf{Set}$$

together with a natural transformation

$$\iota \ : \ [-] \Longrightarrow p \ : \ \mathbf{Pred} \longrightarrow \mathbf{Set}.$$

Here, naturality means that every morphism $f : (A,R) \to (B,S)$ between predicates induces the commutative diagram below, in the category **Set**.

$$\begin{tikzcd}
{[A,R]} \arrow[r, "\iota_{A,R}"] \arrow[d, "{[A,f]}"'] & A \arrow[d, "f"] \\
{[B,S]} \arrow[r, "\iota_{B,S}"] & B
\end{tikzcd} \tag{1}$$

More generally, considering this example of the functor $p : \mathbf{Pred} \to \mathbf{Set}$ as typical, it makes sense to formulate the following “minimalist” notion of comprehension structure:

▶ **Definition 1.** *A comprehension structure on a functor* $p : \mathcal{E} \to \mathcal{B}$ *is a pair* $([-],\iota)$ *consisting of a functor* $[-] : \mathcal{E} \to \mathcal{B}$ *and of a natural transformation* $\iota : [-] \Rightarrow p$.

Interestingly, Jacobs provides in [6] a useful and detailed survey of a hierarchy of axiomatic requirements on a functor $p : \mathcal{E} \to \mathcal{B}$ appearing in the literature, from which such a comprehension structure can be derived. In a decreasing order of generality, one finds:

- Jacob’s comprehension categories, defined in [6], Def. 4.1, page 181.
- Ehrhard’s D-categories [4] called comprehension categories with unit in [6], Def. 4.12.
- Lawvere categories [14] as Jacobs defined them in [6], first paragraph of p. 190.

Our definition just given of a comprehension structure (Def. 1) does not appear as such in the literature, at least in the elementary 2-categorical way we express it here. The reason is that the comprehension pair $([-],\iota)$ can be equivalently formulated as a functor $\mathcal{P} : \mathcal{E} \to \mathcal{B}^{\to}$ to the category of arrows of $\mathcal{B}$, making the diagram below commute:

$$\begin{tikzcd}
\mathcal{E} \arrow[rr, "\mathcal{P}"] \arrow[dr, "p"'] & & \mathcal{B}^{\to} \arrow[dl, "\mathbf{cod}"] \\
& \mathcal{B} &
\end{tikzcd} \tag{2}$$

where $\mathbf{cod} : \mathcal{B}^{\to} \to \mathcal{B}$ denotes the codomain functor. The definitions of comprehension category [6] and of Lawvere category [14] are based on this formulation, while the definition of D-categories [4] works in an entirely different way, which we analyze later in this introduction, as well as in §4. One purpose of the present paper is to revisit these three levels definitions from a purely 2-categorical point of view. This search for a clean 2-categorical account of comprehension in categorical logic is motivated by our desire to understand at this level of abstraction a recent observation by Fumex, Ghani and Johann [2, 3], who establish a very nice duality between (a) the *operation of comprehension* which underlies reasoning by induction using initial algebras, and (b) the *operation of quotienting* which underlies reasoning by coinduction using terminal coalgebras. In particular, Fumex introduces in his PhD thesis a notion of tC-opfibration $p : \mathcal{E} \to \mathcal{B}$ (where tC refers to the section functor t and the comprehension functor C of the structure) adapted for induction reasoning, which he then dualizes into a notion of QCE-category $p^{op} : \mathcal{E}^{op} \to \mathcal{B}^{op}$ (where QCE stands for quotient category with equality) adapted for coinduction reasoning, and simply obtained by reversing the orientation of every morphism in $\mathcal{E}$ and $\mathcal{B}$.

In order to understand and to illustrate this idea of quotient structures, consider the category **Rel** whose objects are pairs (A, R) consisting of a set A and of a binary relation $R \subseteq A \times A$, and whose morphisms $f : (A, R) \to (B, S)$ are functions $f : A \to B$ such that $R(a, a') \Rightarrow S(fa, fa')$. As in the previous case, the functor $p : \mathbf{Rel} \to \mathbf{Set}$ is the forgetful function which transports every binary relation (A, R) to its underlying set A. Every binary relation (A, R) induces a set A/R defined as the quotient

$$A/R := A/{\sim_R}$$

of the underlying set A by the equivalence relation $\sim_R$ generated by the binary relation R. The set A/R comes together with a function

$$\pi_{A,R} \quad : \quad A \longrightarrow A/R$$

which transports every element of A to its equivalence class modulo $\sim_R$. The construction is natural in (A, R) in the sense that it defines a functor

$$[\![-]\!] \quad : \quad \mathbf{Rel} \longrightarrow \mathbf{Set}$$

with $[\![A, R]\!] = A/R$, together with a natural transformation

$$\pi \;:\; p \Longrightarrow [\![-]\!] \;:\; \mathbf{Rel} \longrightarrow \mathbf{Set}.$$

Here, naturality means that every predicate morphism $f : (A, R) \to (B, S)$ induces the commutative diagram below in the category **Set**.

$$\begin{tikzcd} A \arrow[r, "\pi_{A,R}"] \arrow[d, "f"'] & {[\![A, R]\!]} \arrow[d, "{[\![f]\!]}"] \\ B \arrow[r, "\pi_{B,S}"] & {[\![B, S]\!]} \end{tikzcd} \tag{3}$$

In the same way as previously, this example leads us to the following definition, obtained by dualizing Def. 1.

▶ **Definition 2.** *A quotient structure on a functor $p : \mathcal{E} \to \mathcal{B}$ is a pair $([\![-]\!], \pi)$ consisting of a functor $[\![-]\!] : \mathcal{E} \to \mathcal{B}$ and of a natural transformation $\pi : p \Rightarrow [\![-]\!]$.*

In the same way as previously, and by duality, a quotient structure is the same thing as a functor $\mathcal{Q} : \mathcal{E} \to \mathcal{B}^{\to}$ to the category of arrows of $\mathcal{B}$, making the diagram below commute:

$$\begin{tikzcd} \mathcal{E} \arrow[rr, "\mathcal{Q}"] \arrow[dr, "p"'] & & \mathcal{B}^{\to} \arrow[dl, "\mathbf{dom}"] \\ & \mathcal{B} & \end{tikzcd} \tag{4}$$

where $\mathbf{dom} : \mathcal{B}^{\to} \to \mathcal{B}$ denotes the domain functor.

A 2-categorical classification of comprehension structures

(i) Comprehension structures. In order to understand the duality between comprehension and quotient structures, we find enlightening to take seriously the 2-categorical nature of Def. 1 and 2, and to reformulate them in the following way. Suppose given a 2-category $\mathcal{K}$ such as $\mathcal{K} = \mathbf{Cat}$, the 2-category of categories. We consider the 2-category $\mathcal{K}//\mathcal{K}$ whose objects are the triples $(\mathbf{E}, \mathbf{B}, p : \mathbf{E} \to \mathbf{B})$ consisting of a pair of 0-cells $\mathbf{E}$ and $\mathbf{B}$ and a 1-cell $p : \mathbf{E} \to \mathbf{B}$ of the 2-category $\mathcal{K}$, and whose morphisms

$$(f_{\mathbf{E}}, f_{\mathbf{B}}, \varphi) \quad : \quad (\mathbf{E}_1, \mathbf{B}_1, p_1 : \mathbf{E}_1 \to \mathbf{B}_1) \longrightarrow (\mathbf{E}_2, \mathbf{B}_2, p_2 : \mathbf{E}_2 \to \mathbf{B}_2) \tag{5}$$

are triples consisting of a pair of 1-cells $f_{\mathbf{B}} : \mathbf{B}_1 \to \mathbf{B}_2$ and $f_{\mathbf{E}} : \mathbf{E}_1 \to \mathbf{E}_2$ and a 2-cell

$$\begin{tikzcd} \mathbf{E}_1 \arrow[rr, "f_{\mathbf{E}}"] \arrow[d, "p_1"'] & & \mathbf{E}_2 \arrow[d, "p_2"] \\ \mathbf{B}_1 \arrow[rr, "f_{\mathbf{B}}"'] & & \mathbf{B}_2 \arrow[from=1-3, to=2-1, phantom, "\xLeftarrow{\ \varphi\ }"] \end{tikzcd} \qquad \varphi : p_2 \circ f_{\mathbf{E}} \Longrightarrow f_{\mathbf{B}} \circ p_1$$

A morphism (5) is called *strict* when the 2-cell φ is the identity. We write in that case $(f_{\mathbf{E}}, f_{\mathbf{B}})$ instead of $(f_{\mathbf{E}}, f_{\mathbf{B}}, \mathrm{id})$. We also write $\mathcal{K}/\mathcal{K}$ for the sub-2-category of $\mathcal{K}//\mathcal{K}$ of strict morphisms, with the same notion of 2-cell. It is essentially immediate that

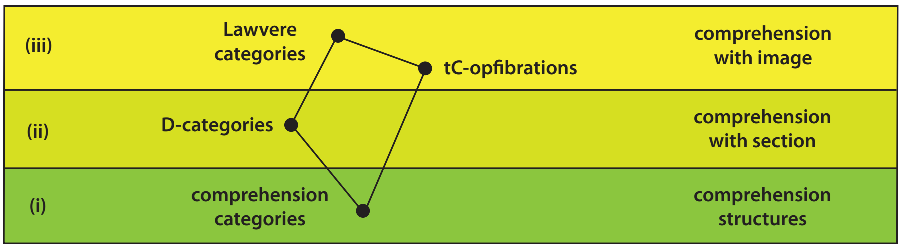
▶ **Proposition 3.** *A comprehension structure* $([-], \iota)$ *(in the sense of Def. 1) is the same thing as a morphism in* $\mathbf{Cat}//\mathbf{Cat}$ *of the form*

$$(f_{\mathcal{E}}, \mathrm{id}_{\mathcal{B}}, \varphi) \quad : \quad (\mathcal{E}, \mathcal{B}, p : \mathcal{E} \to \mathcal{B}) \longrightarrow (\mathcal{B}, \mathcal{B}, \mathrm{id}_{\mathcal{B}} : \mathcal{B} \to \mathcal{B}) \tag{6}$$

One main contribution of the paper is to revisit in this 2-categorical style the hierarchy of comprehension categories described by Jacobs [6]. To that purpose, we introduce three corresponding levels of comprehension structures, each of them coming with an elementary and concise 2-categorical formulation, as depicted in the figure below:

- comprehension structures, Def. 1 as reformulated in Prop. 3,
- comprehension structures with section, Def. 4 as reformulated in Prop. 6,
- comprehension structures with image, Def. 7 as formulated in Def. 25.

One basic observation is that our minimalist notion of comprehension structure (Def. 1) generalizes Jacobs' notion of comprehension category, by relaxing the assumption that the associated functor $\mathcal{P} : \mathcal{E} \to \mathcal{B}^{\to}$ in (2) transports every p-cartesian map of $\mathcal{E}$ to a $\mathbf{cod}$-cartesian map of $\mathcal{B}^{\to}$, that is, to a pullback diagram of the form (1) in the category $\mathcal{B}$. This observation underlies the first layer (in dark green) of our classification below.



(ii) Comprehension structures with section. We move to the next layer and consider Ehrhard's notion of D-category [4, 6] which is based on a convenient but somewhat mysterious recipe to equip a functor $p : \mathcal{E} \to \mathcal{B}$ with a comprehension structure $([-], \iota)$. The recipe [4, 6] works in two stages: (1) first, one equips the functor p with a section $\star : \mathcal{B} \to \mathcal{E}$, (2) then one requires that the section $\star$ has a right adjoint $[-] : \mathcal{E} \to \mathcal{B}$. Recall that a section $\star : \mathcal{B} \to \mathcal{E}$ is a functor such that $p \circ \star = \mathrm{id}_{\mathcal{B}}$. This leads us to the following definition:

▶ **Definition 4.** *A comprehension structure with section on a functor $p : \mathcal{E} \to \mathcal{B}$ is a section $\star : \mathcal{B} \to \mathcal{E}$ together with a right adjoint functor $[-] : \mathcal{E} \to \mathcal{B}$.*

One astonishing aspect of the definition is that the natural transformation $\iota : [-] \Rightarrow p$ of the associated comprehension structure $([-], \iota)$ is not given explicitly, but derived as the image by the functor $p : \mathcal{E} \to \mathcal{B}$ of the counit $\star \circ [-] \Rightarrow \mathrm{id}_{\mathcal{E}}$ of the adjunction $\star \dashv [-]$. From this it follows that the relationship between the natural transformation ι and the two functors $\star, [-]$ is not entirely obvious from a conceptual point of view. We clarify this point by observing here that the original adjunction $\star \dashv [-]$ of Def. 4 living in $\mathcal{K} = \mathbf{Cat}$ is the "emerged part" of a more fundamental adjunction $\star \dashv ([-], \iota)$ living in the 2-category $\mathbf{Cat}//\mathbf{Cat}$, and where the natural transformation ι is thus integrated. A preliminary observation is that

▶ **Proposition 5.** *A section of the functor $p : \mathcal{E} \to \mathcal{B}$ is the same thing as a strict morphism in* $\mathbf{Cat}/\mathbf{Cat}$ *of the form*

$$(s_{\mathcal{E}}, \mathrm{id}_{\mathcal{B}}) \quad : \quad (\mathcal{B}, \mathcal{B}, \mathrm{id}_{\mathcal{B}} : \mathcal{B} \to \mathcal{B}) \longrightarrow (\mathcal{E}, \mathcal{B}, p : \mathcal{E} \to \mathcal{B}) \tag{7}$$

We will prove in the course of the paper (see §4, Prop. 17) that the adjunction $\star \dashv [-]$ in Def. 4 may be equivalently formulated as an adjunction in $\mathbf{Cat}//\mathbf{Cat}$ between the section $\star$ seen as a strict morphism (7) and the comprehension structure $([-], \iota)$ seen as a morphism (6). This property establishes the secretly 2-categorical nature of the notion (Def. 4) of comprehension structure with section:

▶ **Proposition 6.** *A comprehension structure with section is a comprehension structure (6) right adjoint to a section (7) in the 2-category* $\mathbf{Cat}//\mathbf{Cat}$.

The resulting 2-categorical notion of *comprehension structure with section* (Prop. 6) captures the essence of the notion of D-category, and generalizes it in an interesting and useful way to the categories of algebras and coalgebras, see §4 and §8 for a discussion.

(iii) Comprehension structures with image. We move finally to the next layer of our hierarchy, and observe that the functor $p : \mathcal{E} \to \mathcal{B}$ is required to be an opfibration in both notions of Lawvere category and of tC-opfibration [6, 2]. From this follows that the functor $p : \mathcal{E} \to \mathcal{B}$ has an image structure, in the sense elaborated in §6 of this paper. This additional image structure on the functor p enables one to construct a functor

$$\mathbf{image} \quad : \quad \mathcal{B}^{\to} \longrightarrow \mathcal{E} \tag{8}$$

from the arrow category $\mathcal{B}^{\to}$ of the basis category $\mathcal{B}$ to the category $\mathcal{E}$. The functor **image** transports every morphism $f : A \to B$ of the basis category $\mathcal{B}$ to an object $\mathbf{image}(f)$ in the fiber category $\mathcal{E}_B$ of the object B, called the image of $f : A \to B$, and satisfying the expected universality property, see §6 for details. By construction, the image functor (8) makes the diagram below commute:

$$\begin{tikzcd}
\mathcal{E} \arrow[dr, "p"'] & & \mathcal{B}^{\to} \arrow[ll, "\mathbf{image}"'] \arrow[dl, "\mathbf{cod}"] \\
& \mathcal{B} &
\end{tikzcd} \tag{9}$$

In order to recover a comprehension structure (2), the definition of a Lawvere category requires that the functor **image** has a right adjoint $\mathcal{P} : \mathcal{E} \to \mathcal{B}^{\to}$ in the fibered sense above the category $\mathcal{B}$. This leads us to the following definition.

▶ **Definition 7.** *A comprehension structure with image on a functor* $p : \mathcal{E} \to \mathcal{B}$ *is a functor* $\mathbf{image} : \mathcal{B}^{\to} \to \mathcal{E}$ *together with a right adjoint* $\mathcal{P} : \mathcal{E} \to \mathcal{B}^{\to}$ *in the fibered sense above* $\mathcal{B}$*.*

On the other hand, and somewhat surprisingly, the definition of tC-opfibration is apparently weaker, since it only requires that the functor $p : \mathcal{E} \to \mathcal{B}$ has a comprehension structure with section, in the sense of Def. 4. In order to clarify the situation, and to get a clean and harmonious picture, we establish that every tC-opfibration has comprehension with image (in the sense of Def 7) using the following statement, which applies in particular to the case of an opfibration $p : \mathcal{E} \to \mathcal{B}$:

▶ **Proposition 8.** *Suppose that the functor* $p : \mathcal{E} \to \mathcal{B}$ *has an image structure. In that case, every comprehension structure with section (in the sense of Def. 4) defines a comprehension structure with image (in the sense of Def. 7).*

Illustration: inductive reasoning on algebras, coinductive reasoning on coalgebras

Suppose given a functor $p : \mathcal{E} \to \mathcal{B}$ equipped with a comprehension structure with section $\star : \mathcal{B} \to \mathcal{E}$, where the categories $\mathcal{E}$ and $\mathcal{B}$ are moreover equipped with endofunctors $F : \mathcal{B} \to \mathcal{B}$ and $G : \mathcal{E} \to \mathcal{E}$ related by a distributivity law

$$\delta \quad : \quad F \circ p \Longrightarrow p \circ G \quad : \quad \mathcal{E} \longrightarrow \mathcal{B}. \tag{10}$$

One guiding ambition of our 2-categorical account of comprehension structures is to explain by conceptual means the recent characterization by Fumex, Ghani and Johann [2, 3] of the initial G-algebra of $\mathcal{E}$ as the section $\star_A$ of the initial F-algebra μF of the basis category $\mathcal{B}$. To that purpose, we describe in §8 the necessary and sufficient conditions which characterize when the distributivity law (10) on a comprehension structure with section $p : \mathcal{E} \to \mathcal{B}$ induces a comprehension structure with section $\mathbf{Alg}(p) : \mathbf{Alg}_G(\mathcal{E}) \to \mathbf{Alg}_F(\mathcal{B})$ on the associated categories of algebras. In this situation, we obtain a simple conceptual explanation for the forementioned result ([3], Thm 2.10) by Fumex, Ghani and Johann:

▶ **Corollary 9.** *The comprehension structure with section* $\star : \mathcal{B} \to \mathcal{E}$ *lifts to a comprehension structure with section* $\star : \mathbf{Alg}_F(\mathcal{B}) \to \mathbf{Alg}_G(\mathcal{E})$ *which is left adjoint to comprehension* $[-]$ *and thus transports the initial* F*-algebra* μF *to the initial* G*-algebra* $\mu G = \star_{\mu F}$*.*

We proceed dually in the case of quotient structures and obtain necessary and sufficient conditions to ensure that

▶ **Corollary 10.** *The quotient structure with section* $\star : \mathcal{B} \to \mathcal{E}$ *lifts to a quotient structure with section* $\star : \mathbf{CoAlg}_F(\mathcal{B}) \to \mathbf{CoAlg}_G(\mathcal{E})$ *which is right adjoint to quotient* $[\![-]\!]$ *and thus transports the terminal* F*-coalgebra* νF *to the terminal* G*-coalgebra* $\nu G = \star_{\nu F}$*.*

Plan of the paper

After this long and detailed introduction, we recall in §2 the notion of arrow 2-category $\mathcal{K}//\mathcal{K}$ and establish in §3 a simple and useful description of the formal adjunctions in this 2-category. This leads us to formulate in §4 our 2-categorical notion of *comprehension with section.* We then formulate in §5 and §6 the notion of path object $(\mathbf{B}^{\to}, \beta)$ of an object $\mathbf{B}$ in any 2-category $\mathcal{K}$, and the related notion of morphism $p : \mathbf{E} \to \mathbf{B}$ with an image structure. This leads us to establish in §7 that a comprehension with image $p : \mathbf{E} \to \mathbf{B}$ is the same thing as a comprehension structure with section $\star : \mathbf{B} \to \mathbf{E}$, whose underlying section comes with an image structure, and thus a morphism $\mathbf{image} : \mathbf{B}^{\to} \to \mathbf{E}$. We then illustrate in §8 the benefits of our 2-categorical approach with the example of inductive and coinductive reasoning on algebra and coalgebra structures, and finally conclude in §9.

2 Definition of the arrow 2-categories $\mathcal{K}//\mathcal{K}$ and $\mathcal{K}/\mathcal{K}$

We explained in the introduction, see (5), how to define the objects and the morphisms of the 2-category $\mathcal{K}//\mathcal{K}$ associated to a 2-category $\mathcal{K}$. For the sake of completeness, we recall now that a 2-cell

$$(\theta_{\mathbf{B}}, \theta_{\mathbf{E}}) \; : \; (f_{\mathbf{E}}, f_{\mathbf{B}}, \varphi) \Longrightarrow (g_{\mathbf{E}}, g_{\mathbf{B}}, \psi) \; : \; (\mathbf{E}_1, \mathbf{B}_1, p_1) \longrightarrow (\mathbf{E}_2, \mathbf{B}_2, p_2)$$

of the 2-category $\mathcal{K}//\mathcal{K}$ is defined as a pair of 2-cells $\theta_B : f_{\mathbf{B}} \Rightarrow g_{\mathbf{B}}$ and $\theta_{\mathbf{E}} : f_{\mathbf{E}} \Rightarrow g_{\mathbf{E}}$ of the original 2-category $\mathcal{K}$, making the two pasting diagrams equal:

$$\begin{tikzcd}
\mathbf{E}_1 \arrow[r, bend left, "f_{\mathbf{E}}"] \arrow[d, "p_1"'] & \mathbf{E}_2 \arrow[d, "p_2"] \arrow[dl, Rightarrow, bend right, "\varphi"'] \\
\mathbf{B}_1 \arrow[r, bend left, "f_{\mathbf{B}}", ""{name=U, below}] \arrow[r, bend right, "g_{\mathbf{B}}"', ""{name=D, above}] & \mathbf{B}_2
\arrow[from=U, to=D, Rightarrow, "\theta_B"]
\end{tikzcd}
=
\begin{tikzcd}
\mathbf{E}_1 \arrow[r, bend left, "f_{\mathbf{E}}", ""{name=A, below}] \arrow[r, bend right, "g_{\mathbf{E}}"', ""{name=C, above}] \arrow[d, "p_1"'] & \mathbf{E}_2 \arrow[d, "p_2"] \arrow[dl, Rightarrow, bend left, "\psi"] \\
\mathbf{B}_1 \arrow[r, bend right, "g_{\mathbf{B}}"'] & \mathbf{B}_2
\arrow[from=A, to=C, Rightarrow, "\theta_{\mathbf{E}}"]
\end{tikzcd}$$

It is worth mentioning that, thanks to this carefully chosen definition of 2-cells, there exists a pair of 2-functors

$$\begin{tikzcd}
\mathcal{K}//\mathcal{K} \arrow[r, shift left, "\mathbf{source}"] \arrow[r, shift right, "\mathbf{target}"'] & \mathcal{K}
\end{tikzcd} \tag{11}$$

defined as the expected first and second projections, which transport every object $(\mathbf{E}, \mathbf{B}, p : \mathbf{E} \to \mathbf{B})$ of the 2-category $\mathcal{K}//\mathcal{K}$ to the object

$$\mathbf{source}(\mathbf{E}, \mathbf{B}, p : \mathbf{E} \to \mathbf{B}) = \mathbf{E} \qquad\qquad \mathbf{target}(\mathbf{E}, \mathbf{B}, p : \mathbf{E} \to \mathbf{B}) = \mathbf{B}$$

of the underlying 2-category $\mathcal{K}$. Finally, let us also mention that the 2-category $\mathcal{K}/\mathcal{K}$ of strict morphisms in $\mathcal{K}//\mathcal{K}$ comes exactly with the same notion of 2-cell. In other words, the inclusion 2-functor $\mathcal{K}/\mathcal{K} \to \mathcal{K}//\mathcal{K}$ is locally fully faithful.

3 Formal adjunctions in the 2-category $\mathcal{K}//\mathcal{K}$

As explained in the introduction in the case $\mathcal{K} = \mathbf{Cat}$, one main observation of the paper is that the notion of *comprehension structure with section* (Def. 4) can be elegantly expressed as a specific form of adjunction living in the 2-category $\mathcal{K}//\mathcal{K}$ (Prop. 6). As a warm up exercise, we study the notion of formal adjunction in $\mathcal{K}//\mathcal{K}$ in the sense of Street [13] and relate it in full generality to the notion of formal adjunction in the original 2-category $\mathcal{K}$. Suppose given a pair of morphisms

$$\begin{aligned}
L &= (L_{\mathbf{E}}, L_{\mathbf{B}}, \varphi) \quad : \quad (\mathbf{E}_1, \mathbf{B}_1, p_1 : \mathbf{E}_1 \to \mathbf{B}_1) \longrightarrow (\mathbf{E}_2, \mathbf{B}_2, p_2 : \mathbf{E}_2 \to \mathbf{B}_2) \\
R &= (R_{\mathbf{E}}, R_{\mathbf{B}}, \psi) \quad : \quad (\mathbf{E}_2, \mathbf{B}_2, p_2 : \mathbf{E}_2 \to \mathbf{B}_2) \longrightarrow (\mathbf{E}_1, \mathbf{B}_1, p_1 : \mathbf{E}_1 \to \mathbf{B}_1)
\end{aligned} \tag{12}$$

living in the 2-category $\mathcal{K}//\mathcal{K}$, and thus depicted as below in the underlying 2-category $\mathcal{K}$:

$$\begin{tikzcd}
\mathbf{E}_1 \arrow[r, "L_{\mathbf{E}}"] \arrow[d, "p_1"'] & \mathbf{E}_2 \arrow[d, "p_2"] \\
\mathbf{B}_1 \arrow[r, "L_{\mathbf{B}}"'] & \mathbf{B}_2
\arrow[from=1-1, to=2-2, phantom, "\xLeftarrow{\ \varphi\ }" description]
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathbf{E}_1 \arrow[d, "p_1"'] & \mathbf{E}_2 \arrow[l, "R_{\mathbf{E}}"'] \arrow[d, "p_2"] \\
\mathbf{B}_1 & \mathbf{B}_2 \arrow[l, "R_{\mathbf{B}}"]
\arrow[from=1-1, to=2-2, phantom, "\xRightarrow{\ \psi\ }" description]
\end{tikzcd}$$

By definition, a *formal adjunction* between L and R in the 2-category $\mathcal{K}//\mathcal{K}$ is defined as a pair of 2-cells

$$(\eta_{\mathbf{B}}, \eta_{\mathbf{E}}) : (\mathrm{id}, \mathrm{id}, \mathrm{id}) \Rightarrow (R_{\mathbf{E}} \circ L_{\mathbf{E}}, R_{\mathbf{B}} \circ L_{\mathbf{B}}, (R_{\mathbf{B}} \circ \varphi)(\psi \circ L_{\mathbf{E}})) : (\mathbf{E}_1, \mathbf{B}_1, p_1) \to (\mathbf{E}_1, \mathbf{B}_1, p_1)$$
$$(\varepsilon_{\mathbf{B}}, \varepsilon_{\mathbf{E}}) : (L_{\mathbf{E}} \circ R_{\mathbf{E}}, L_{\mathbf{B}} \circ R_{\mathbf{B}}, (L_{\mathbf{B}} \circ \psi)(\varphi \circ R_{\mathbf{E}})) \Rightarrow (\mathrm{id}, \mathrm{id}, \mathrm{id}) : (\mathbf{E}_2, \mathbf{B}_2, p_2) \to (\mathbf{E}_2, \mathbf{B}_2, p_2)$$

in the 2-category $\mathcal{K}//\mathcal{K}$, satisfying the triangular equations, see [13, 10] for details. One nice consequence of this definition by generators and relations is that the resulting notion of formal adjunction is preserved by 2-functors. Every formal adjunction $L \dashv R$ in $\mathcal{K}//\mathcal{K}$ is thus transported by the 2-functors (11) into a pair of formal adjunctions $L_{\mathbf{B}} \dashv R_{\mathbf{B}}$ and $L_{\mathbf{E}} \dashv R_{\mathbf{E}}$ in the underlying 2-category $\mathcal{K}$. From this follows that the 2-cell $\psi : p_1 \circ R_{\mathbf{E}} \Rightarrow R_{\mathbf{B}} \circ p_2$ in $\mathcal{K}$ induces a 2-cell $\widetilde{\psi} : L_{\mathbf{B}} \circ p_1 \Rightarrow p_2 \circ L_{\mathbf{E}}$ called the *mate* of ψ, of the form below:

$$\begin{tikzcd}
\mathbf{E}_1 \arrow[r, "L_{\mathbf{E}}"] \arrow[d, "p_1"'] & \mathbf{E}_2 \arrow[d, "p_2"] \\
\mathbf{B}_1 \arrow[r, "L_{\mathbf{B}}"'] & \mathbf{B}_2
\arrow[from=1-1, to=2-2, phantom, "\xRightarrow{\ \widetilde{\psi}\ }" description]
\end{tikzcd}$$

Suppose given two morphisms L and R in the 2-category $\mathcal{K}//\mathcal{K}$ as in (12). In that case,

▶ **Proposition 11.** *A formal adjunction $L \dashv R$ in the 2-category $\mathcal{K}//\mathcal{K}$ is the same thing as a pair of formal adjunctions $L_{\mathbf{B}} \dashv R_{\mathbf{B}}$ and $L_{\mathbf{E}} \dashv R_{\mathbf{E}}$ in the 2-category $\mathcal{K}$, such that the induced mate $\widetilde{\psi}$ of the 2-cell ψ is the inverse of the 2-cell φ in the 2-category $\mathcal{K}$.*

From this follows easily that

▶ **Proposition 12.** *A pair of formal adjunctions $L_{\mathbf{B}} \dashv R_{\mathbf{B}}$ and $L_{\mathbf{E}} \dashv R_{\mathbf{E}}$ in the 2-category $\mathcal{K}$ lifts to a formal adjunction $(L_{\mathbf{B}}, L_{\mathbf{E}}, \varphi) \dashv (R_{\mathbf{B}}, R_{\mathbf{E}}, \psi)$ in the 2-category $\mathcal{K}//\mathcal{K}$ precisely when the 2-cell φ is invertible in $\mathcal{K}$ and the 2-cell ψ coincides with the mate of φ^{-1}.*

It should be mentioned that a similar observation is made by Kelly [9] (Prop. 1.3) on the 2-category of D-algebras derived from a 2-monad D on the 2-category $\mathcal{K}$, see also [8], Section 3.5. It should be also noted that the characterization of formal adjunctions in $\mathcal{K}//\mathcal{K}$ is also very similar to the description of formal adjunctions in the 2-category of monoidal categories and lax monoidal functors, see for instance [10].

4 Comprehension structures with section

In this section, we suppose given a morphism $p : \mathbf{E} \to \mathbf{B}$ in the 2-category $\mathcal{K}$ and establish (Prop. 17) that a *comprehension structure with section* on a morphism $p : \mathbf{E} \to \mathbf{B}$ originally defined (Def. 16) as an adjunction $\star \dashv [-]$ in the 2-category $\mathcal{K}$, can be in fact lifted (and thus equivalently defined) as a specific form of adjunction $\star \dashv ([-], \iota)$ in the 2-category $\mathcal{K}//\mathcal{K}$. As expected, the proof of Prop. 17 relies on the characterization of formal adjunctions in $\mathcal{K}//\mathcal{K}$ just established in the previous section, see Prop. 12. The result provides a general 2-categorical formulation of the proposition (Prop. 6) stated in the introduction for the particular case $\mathcal{K} = \mathbf{Cat}$. In order to perform our construction at this general 2-categorical level of abstraction, we start by defining a *comprehension structure* on a morphism $p : \mathbf{E} \to \mathbf{B}$ in the 2-category $\mathcal{K}$, in a way which generalizes what we did in the introduction (see Def. 1) in the specific case $\mathcal{K} = \mathbf{Cat}$.

▶ **Definition 13.** *A comprehension structure on the morphism $p : \mathbf{E} \to \mathbf{B}$ is a pair $([-], \iota)$ consisting of a morphism $[-] : \mathbf{E} \to \mathbf{B}$ and of a 2-cell $\iota : [-] \Rightarrow p$ in the 2-category $\mathcal{K}$.*

We carry on as we did in the introduction (see Prop. 3) and observe that

▶ **Proposition 14.** *A comprehension structure $([-], \iota)$ on $p : \mathbf{E} \to \mathbf{B}$ is the same thing as a morphism in the 2-category $\mathcal{K}//\mathcal{K}$ of the form*

$$(f_{\mathbf{E}}, \mathrm{id}_{\mathbf{B}}, \varphi) \quad : \quad (\mathbf{E}, \mathbf{B}, p : \mathbf{E} \to \mathbf{B}) \longrightarrow (\mathbf{B}, \mathbf{B}, \mathrm{id}_{\mathbf{B}} : \mathbf{B} \to \mathbf{B}) \tag{13}$$

We proceed in just the same way as we did in the introduction with Prop. 5, and characterize a section $\star : \mathbf{B} \to \mathbf{E}$ of the morphism $p : \mathbf{E} \to \mathbf{B}$ as a specific form of strict morphism:

▶ **Proposition 15.** *A section $\star : \mathbf{B} \to \mathbf{E}$ of the morphism $p : \mathbf{E} \to \mathbf{B}$ in $\mathcal{K}$ is the same thing as a strict morphism in $\mathcal{K}//\mathcal{K}$ of the form*

$$(s_{\mathbf{E}}, \mathrm{id}_{\mathbf{B}}) \quad : \quad (\mathbf{B}, \mathbf{B}, \mathrm{id}_{\mathbf{B}} : \mathbf{B} \to \mathbf{B}) \longrightarrow (\mathbf{E}, \mathbf{B}, p : \mathbf{E} \to \mathbf{B}) \tag{14}$$

We are now ready to give our general 2-categorical definition of *comprehension structure with section* on the morphism $p : \mathbf{E} \to \mathbf{B}$ in the 2-category $\mathcal{K}$.

▶ **Definition 16.** *A comprehension structure with section on $p : \mathbf{E} \to \mathbf{B}$ is a section $\star : \mathbf{B} \to \mathbf{E}$ together with a right adjoint $[-] : \mathbf{E} \to \mathbf{B}$ in the 2-category $\mathcal{K}$.*

We then take advantage of Prop. 12 in order to establish that:

▶ **Proposition 17.** *A comprehension structure with section is a comprehension structure (13) right adjoint to a section (14) in the 2-category $\mathcal{K}//\mathcal{K}$.*

Proof. Suppose given a section $\star : \mathbf{B} \to \mathbf{E}$ of the morphism $p : \mathbf{E} \to \mathbf{B}$ in the 2-category $\mathcal{K}$, described (Prop. 15) as a strict morphism

$$L \;=\; (\star, \mathrm{id}_{\mathbf{B}}, \mathrm{id}) \quad : \quad (\mathbf{B}, \mathbf{B}, \mathrm{id}_{\mathbf{B}} : \mathbf{B} \to \mathbf{B}) \longrightarrow (\mathbf{E}, \mathbf{B}, p : \mathbf{E} \to \mathbf{B})$$

in the 2-category $\mathcal{K}/\mathcal{K}$. Consider moreover a morphism of the form

$$R \;=\; ([-], \mathrm{id}_{\mathbf{B}}, \iota) \quad : \quad (\mathbf{E}, \mathbf{B}, p : \mathbf{E} \to \mathbf{B}) \longrightarrow (\mathbf{B}, \mathbf{B}, \mathrm{id}_{\mathbf{B}} : \mathbf{B} \to \mathbf{B})$$

in the 2-category $\mathcal{K}//\mathcal{K}$. The morphisms L and R of the 2-category $\mathcal{K}//\mathcal{K}$ can be depicted as follows in the underlying 2-category $\mathcal{K}$:

$$\begin{tikzcd}
\mathbf{B} \arrow[r, "\star"] \arrow[d, "\mathrm{id}_{\mathbf{B}}"'] & \mathbf{E} \arrow[d, "p"] \\
\mathbf{B} \arrow[r, "\mathrm{id}_{\mathbf{B}}"'] & \mathbf{B}
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
\mathbf{B} \arrow[d, "\mathrm{id}_{\mathbf{B}}"'] \arrow[dr, phantom, "\overset{\iota}{\Longrightarrow}"] & \mathbf{E} \arrow[l, "{[-]}"'] \arrow[d, "p"] \\
\mathbf{B} & \mathbf{B} \arrow[l, "\mathrm{id}_{\mathbf{B}}"]
\end{tikzcd}$$

Now, suppose that $\star$ and $[-]$ define a comprehension structure with section, in the sense of Def 16. By definition, this means that there is an adjunction $\star \dashv [-]$. By Prop. 12, the adjunction $\star \dashv [-]$ lifts to an adjunction $L \dashv R$ between the morphisms L and R in the 2-category $\mathcal{K}//\mathcal{K}$ precisely when the 2-cell $\iota : [-] \Rightarrow p$ in the 2-category $\mathcal{K}$ is the mate defined as $\iota = p \circ \varepsilon$, of the identity 2-cell $\mathrm{id} : \mathrm{id}_{\mathbf{B}} \Rightarrow p \circ \star$. Here, the 2-cell $\varepsilon : [-] \circ \star \Rightarrow \mathrm{id}$ denotes the counit of the adjunction $\star \dashv [-]$ in the 2-category $\mathcal{K}$. This establishes one direction of the proof, while the other direction is immediate. ◀

Note that the definition of the 2-cell $\iota : [-] \Rightarrow p$ as the mate $\iota : [-] \Rightarrow p$ of the identity 2-cell $\mathrm{id} : \mathrm{id}_{\mathbf{B}} \Rightarrow p \circ \star$ provides a conceptual explanation for the definition of $\iota : [-] \Rightarrow p$ as the image by p of the counit ε of the counit of the adjunction $\star \dashv [-]$.

5 Path objects in a 2-category

We introduce the notion of *path object* on an object $\mathbf{B}$ in a 2-category $\mathcal{K}$. This construction, which generalizes the usual construction of the arrow category $\mathcal{B}^{\rightarrow}$ of a category $\mathcal{B}$ when $\mathcal{K} = \mathbf{Cat}$, will play an important role in §8 when we apply our constructions to inductive and coinductive reasoning on initial algebras and terminal algebras. Every object $\mathbf{B}$ in a 2-category $\mathcal{K}$ induces a 2-functor

$$\mathcal{K}(-,\mathbf{B})^{\rightarrow} \;:\; \mathcal{K} \longrightarrow \mathbf{Cat} \tag{15}$$

which transports every object $X \in \mathcal{K}$ to the arrow category $\mathcal{K}(\mathbf{X},\mathbf{B})^{\rightarrow}$ of the hom-category $\mathcal{K}(\mathbf{X},\mathbf{B})$ between $\mathbf{X}$ and $\mathbf{B}$.

▶ **Definition 18.** *A path object for the object* $\mathbf{B}$ *in the 2-category* $\mathcal{K}$ *is a pair* $(\mathbf{B}^{\rightarrow},\beta)$ *consisting of an object* $\mathbf{B}^{\rightarrow}$ *and of a family of isomorphisms*

$$\beta_{\mathbf{X}} \;:\; \mathcal{K}(\mathbf{X},\mathbf{B}^{\rightarrow}) \cong \mathcal{K}(\mathbf{X},\mathbf{B})^{\rightarrow}$$

2-natural in the object X*. Terminology: one says in that case that the pair* $(\mathbf{B}^{\rightarrow},\beta)$ *defines a representation of the 2-functor (15).*

Note that every path object $(\mathbf{B}^{\rightarrow},\beta)$ comes equipped with three morphisms and a 2-cell

$$\mathbf{cod},\mathbf{dom} : \mathbf{B}^{\rightarrow} \to \mathbf{B} \qquad \mathbf{id} : \mathbf{B} \to \mathbf{B}^{\rightarrow} \qquad \mathbf{hom} : \mathbf{dom} \Rightarrow \mathbf{cod} : \mathbf{B}^{\rightarrow} \to \mathbf{B}$$

satisfying $\mathbf{dom} \circ \mathbf{id} = \mathrm{id}_{\mathbf{B}} = \mathbf{cod} \circ \mathbf{id}$ and that the 2-cell $\mathbf{hom} \circ \mathbf{id}$ coincides with the identity 2-cell on $\mathrm{id}_{\mathbf{B}}$. A remarkable property is that

▶ **Proposition 19.** *The three morphisms* $\mathbf{cod}$*,* $\mathbf{id}$ *and* $\mathbf{dom}$ *are related by a pair of formal adjunctions* $\mathbf{cod} \dashv \mathbf{id}$ *and* $\mathbf{id} \dashv \mathbf{dom}$ *in the 2-category* $\mathcal{K}$*.*

Going back to the definition (Def. 16) of a comprehension structure with section, this establishes that

▶ **Proposition 20.** *Every path object* $(\mathbf{B}^{\rightarrow},\beta)$ *defines a comprehension structure with section* $\mathbf{id} : \mathbf{B} \to \mathbf{B}^{\rightarrow}$ *on the morphism* $\mathbf{cod} : \mathbf{B}^{\rightarrow} \to \mathbf{B}$ *in the 2-category* $\mathcal{K}$*.*

Note that the comprehension structure $([-],\iota)$ constructed in Prop. 17 using a 2-categorical mate in $\mathcal{K}$ is provided in that case by the pair $(\mathbf{dom},\mathbf{hom})$ with 2-cell $\mathbf{hom} : \mathbf{dom} \Rightarrow \mathbf{cod}$.

6 Functors with image structure

In this section, we introduce the notion of *functor* $p : \mathcal{E} \to \mathcal{B}$ *with image structure* which weakens (and thus generalizes) the usual notion of Grothendieck opfibration $p : \mathcal{E} \to \mathcal{B}$.

▶ **Definition 21** (Image structure)**.** *An image structure on a functor* $p : \mathcal{E} \to \mathcal{B}$ *is a pair* $(\star,\lambda)$ *consisting of a section* $\star$ *defined as a functor* $\star : \mathcal{B} \to \mathcal{E}$ *satisfying the equation* $p \circ \star = \mathrm{id}_{\mathcal{B}}$*, together with a family* λ *of opcartesian morphisms*

$$u \;:\; A \longrightarrow B \qquad \vDash \qquad \lambda_u \;:\; \star_A \longrightarrow \exists_u[\star_A] \tag{16}$$

indexed by the morphisms $u : A \to B$ of the basis category $\mathcal{B}$. We will also suppose for convenience that the opcartesian morphism

$$\mathrm{id}_A \;:\; A \longrightarrow A \qquad \vDash \qquad \lambda_u \;:\; \star_A \longrightarrow \exists_{\mathrm{id}_A}[\star_A]$$

coincides with the identity morphism on $\star_A$.

Here, we follow the fibered philosophy of refinement type systems [11], and write

$$u \;:\; A \longrightarrow B \qquad \vDash \qquad f \;:\; R \longrightarrow S$$

when a morphism $f : R \to S$ in the category $\mathcal{E}$ has image $p(f) = u : A \to B$ in the category $\mathcal{B}$. The intuition is that the morphism $f : R \to S$ is "above" the morphism $u : A \to B$, and "dependent" of it. Accordingly, we write $\mathcal{E}_{u:A\to B}(R,S)$ for the set of such morphisms $f : R \to S$ such that $p(f) = u$. Let us recall what universal property is required of the morphism (16) in order to make it opcartesian. By precomposition in the category $\mathcal{E}$, every morphism

$$v \;:\; B \longrightarrow B' \qquad \vDash \qquad h \;:\; \exists_u[\star_A] \longrightarrow S'$$

induces a morphism

$$v \circ u \;:\; A \longrightarrow B' \qquad \vDash \qquad h \circ \lambda_u \;:\; \star_A \longrightarrow S'$$

The fact that the morphism (16) is opcartesian simply means that the operation is reversible, and thus induces a bijection

$$\mathcal{E}_{v\circ u:A\to C}(\star_A, S') \quad \cong \quad \mathcal{E}_{v:B\to C}(\exists_u[\star_A], S')$$

for every morphism $v : B \to B'$ and every object S' in the fiber of B'.

▶ **Proposition 22.** *For every morphism $u : A \to B$ of the category $\mathcal{B}$, every functor $p : \mathcal{E} \to \mathcal{B}$ with an image structure $(\star, \lambda)$ comes equipped with a family of morphisms*

$$v \;:\; B \longrightarrow B' \qquad \vDash \qquad v\triangleright \;:\; \exists_u[\star_A] \longrightarrow \exists_{v\circ u}[\star_A] \tag{17}$$

indexed by the morphisms $v : B \to B'$ of the category $\mathcal{B}$, and a family of morphisms

$$\mathrm{id}_B \;:\; B \longrightarrow B \qquad \vDash \qquad \triangleleft w \;:\; \exists_{u\circ w}[\star_{A'}] \longrightarrow \exists_u[\star_A] \tag{18}$$

indexed by the morphisms $w : A' \to A$ of the category $\mathcal{B}$. These morphisms make a series of diagrams commute. First of all, the three coherence diagrams below commute

$$\begin{tikzcd}
 & \exists_u[\star_A] \arrow[dd, "v\triangleright"] \\
\star_A \arrow[ur, bend left, "\lambda_u"] \arrow[dr, bend right, "\lambda_{vu}"'] & (a) \\
 & \exists_{v\circ u}[\star_A]
\end{tikzcd}
\qquad
\begin{tikzcd}
\star_{A'} \arrow[r, "\lambda_{u\circ w}"] \arrow[d, "\star_w"'] \arrow[dr, phantom, "(b)"] & \exists_{u\circ w}[\star_{A'}] \arrow[d, "\triangleleft w"] \\
\star_A \arrow[r, "\lambda_u"'] & \exists_u[\star_A]
\end{tikzcd}
\qquad
\begin{tikzcd}
\exists_{u\circ w}[\star_{A'}] \arrow[r, "v\triangleright"] \arrow[d, "\triangleleft w"'] \arrow[dr, phantom, "(c)"] & \exists_{v\circ u\circ w}[\star_{A'}] \arrow[d, "\triangleleft w"] \\
\exists_u[\star_A] \arrow[r, "v\triangleright"'] & \exists_{v\circ u}[\star_A]
\end{tikzcd}
\tag{19}$$

for every path $A' \overset{w}{\to} A \overset{u}{\to} B \overset{v}{\to} B'$ in the category $\mathcal{B}$. Then, the functorial nature of (17) and (18) is ensured by the fact that the diagrams below commute

$$\begin{tikzcd}
 & \exists_{v\circ u}[\star_A] \arrow[dr, "v'\triangleright"] & \\
\exists_u[\star_A] \arrow[ur, "v\triangleright"] \arrow[rr, "(v'\circ v)\triangleright"] & & \exists_{v'\circ v\circ u}[\star_A]
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \exists_{u\circ w}[\star_{A'}] \arrow[dr, "\triangleleft w"] & \\
\exists_{u\circ w\circ w'}[\star_{A''}] \arrow[ur, "\triangleleft w'"] \arrow[rr, "\triangleleft(w\circ w')"] & & \exists_u[\star_A]
\end{tikzcd}
\tag{20}$$

for every pair of paths $A \xrightarrow{u} B \xrightarrow{v} B' \xrightarrow{v'} B''$ *and* $A'' \xrightarrow{w'} A' \xrightarrow{w} A \xrightarrow{u} B$ *of the category* $\mathcal{B}$*, and moreover, that the morphisms*

$$\begin{array}{lcl} \mathrm{id}_B : B \longrightarrow B & \vDash & (id_B)\triangleright : \exists_u[\star_A] \longrightarrow \exists_u[\star_A] \\ \mathrm{id}_B : B \longrightarrow B & \vDash & \triangleleft(id_A) : \exists_u[\star_A] \longrightarrow \exists_u[\star_A] \end{array} \tag{21}$$

coincide with the identity, for every morphism $u : A \to B$ *of the basis category* $\mathcal{B}$*.*

Proof. The two morphisms (17) and (18) are defined by the universal property of the family of opcartesian morphisms λ defining the image structure, as the unique morphisms $v\triangleright$ and $\triangleleft w$ making the two diagrams commute in (19-ab). The three coherence properties (19-c) (20) and (21) follow easily from the definition of the two morphisms (17) and (18). ◀

We deduce from the statement (Prop. 22) just established that

▶ **Corollary 23.** *Every functor* $p : \mathcal{E} \to \mathcal{B}$ *with an image structure comes with a functor*

$$\mathbf{image} \ : \ \mathcal{B}^{\to} \longrightarrow \mathcal{E}$$

called the ***image functor*** *associated to the image structure.*

Proof. The image functor transports every object $u : A \to B$ of the arrow category $\mathcal{B}^{\to}$ to the object $\exists_u[\star_A]$ defined by the image structure, and every morphism

$$(v, w) \quad : \quad (A, B, u : A \to B) \longrightarrow (A', B', u' : A' \to B')$$

to the composite morphism below in the category $\mathcal{E}$

$$v \ : \ B \longrightarrow B' \quad \vDash \quad \exists_u[\star_A] \xrightarrow{v\triangleright} \exists_{v\circ u}[\star_A] \xrightarrow{\mathrm{id}} \exists_{u'\circ w}[\star_A] \xrightarrow{\triangleleft w} \exists_{u'}[\star_{A'}]$$

The functoriality of **image** follows from the coherence properties (19-c) (20) and (21) established in Prop. 22. ◀

The resulting image functor $\mathbf{image} : \mathcal{B}^{\to} \to \mathcal{E}$ extends the section $\star : \mathcal{B} \to \mathcal{E}$, in the expected sense that the diagram below commutes:

$$\begin{tikzcd}
 & \mathcal{B} & \\
 & \mathcal{E} \arrow[u, "p"] & \\
\mathcal{B} \arrow[ruu, bend left, "\mathrm{id}_{\mathcal{B}}"] \arrow[ru, "\star"] \arrow[rr, "\mathbf{id}"'] & & \mathcal{B}^{\to} \arrow[luu, bend right, "\mathbf{cod}"'] \arrow[lu, "\mathbf{image}"']
\end{tikzcd} \tag{22}$$

In particular, as explained in the introduction, the diagram (9) commutes by definition of the image functor. Note that every Grothendieck opfibration $p : \mathcal{E} \to \mathcal{B}$ with a section $\star : \mathcal{B} \to \mathcal{E}$ comes equipped with an image structure, which is canonical when the opfibration is cloven.

7 Comprehension structures with image

In order to work in full generality, and to include the case of the 2-categories of algebras and coalgebras treated in §8, we find convenient to generalize our definition Def. 21 of image structure for a functor $p : \mathcal{E} \to \mathcal{B}$ in the specific case $\mathcal{K} = \mathbf{Cat}$ to any morphism $p : \mathbf{E} \to \mathbf{B}$ in a 2-category $\mathcal{K}$.

▶ **Definition 24** (Image structure). *An image structure on a morphism $p : \mathbf{E} \to \mathbf{B}$ in a 2-category $\mathcal{K}$ is a section $\star : \mathbf{B} \to \mathbf{E}$ equipped with a family λ of 2-cells*

$$\lambda_u \quad : \quad \star_a \Longrightarrow \exists_u[\star_a] \quad : \quad \mathbf{X} \longrightarrow \mathbf{E} \tag{23}$$

indexed by the objects $\mathbf{X}$, the morphisms $a, b : \mathbf{X} \to \mathbf{B}$ and the 2-cells $u : a \Rightarrow b$ of the 2-category $\mathcal{K}$, where the morphism $\star_a$ is defined as the composite $\star \circ a : \mathbf{X} \to \mathbf{E}$. One requires moreover that each 2-cell λ_u defines an opcartesian morphism

$$u \ : \ a \Longrightarrow b \quad \vDash \quad \lambda_u \ : \ \star_a \Longrightarrow \exists_u[\star_a] \tag{24}$$

with respect to the postcomposition functor

$$\mathcal{K}(\mathbf{X}, p) \quad : \quad \mathcal{K}(\mathbf{X}, \mathbf{E}) \longrightarrow \mathcal{K}(\mathbf{X}, \mathbf{B})$$

above the morphism $u : a \Rightarrow b$ in the category $\mathcal{K}(\mathbf{X}, \mathbf{B})$. We also ask for convenience that $\lambda_u : \star_a \Rightarrow \exists_u[\star_a]$ coincides with the identity 2-cell when $u : a \Rightarrow a$ is the identity 2-cell.

Note that every morphism $p : \mathbf{E} \to \mathbf{B}$ with an image structure $(\star, \lambda)$ to an object $\mathbf{B}$ equipped with a path-object $(\mathbf{B}^{\to}, \beta)$ in the 2-category $\mathcal{K}$ comes equipped with a morphism $\mathbf{image} : \mathbf{B}^{\to} \to \mathbf{E}$ defined as $\mathbf{image} = \exists_{\mathbf{hom}}[\star_{\mathbf{dom}}]$, and thus satisfying the equality:

$$\begin{tikzcd} & \mathbf{B} \arrow[d, "\star"] \\ \mathbf{B}^{\to} \arrow[ur, bend left, "\mathbf{dom}"] \arrow[dr, bend right, "\mathbf{cod}"'] \arrow[r, phantom, "\Downarrow\,\mathbf{hom}"] & \mathbf{E} \arrow[d, "p"] \\ & \mathbf{B} \end{tikzcd} \quad = \quad \begin{tikzcd} & \mathbf{B} \arrow[d, "\star"] \arrow[dl, phantom, "\Downarrow\,\lambda_{\mathbf{hom}}"] \\ \mathbf{B}^{\to} \arrow[ur, bend left, "\mathbf{dom}"] \arrow[dr, bend right, "\mathbf{cod}"'] \arrow[r, "\mathbf{image}"'] & \mathbf{E} \arrow[d, "p"] \\ & \mathbf{B} \end{tikzcd} \tag{25}$$

Moreover, the resulting image morphism makes the counterpart of diagram (22) commute for the same reason as in the specific case of the 2-category $\mathcal{K} = \mathbf{Cat}$. For that reason, the morphism **image** may be seen as a morphism

$$\mathbf{image} \quad : \quad (\mathbf{B}^{\to}, \mathbf{cod}) \to (\mathbf{E}, p) \tag{26}$$

in the slice 2-category $\mathcal{K}/\mathbf{B}$, defined as the expected sub-2-category of $\mathcal{K}/\mathcal{K}$ whose objects are the morphisms $p : \mathbf{E} \to \mathbf{B}$ with codomain $\mathbf{B}$. We are now in the position of defining the notion of *comprehension structure with image* at that 2-categorical level of generality.

▶ **Definition 25.** *Suppose given a morphism $p : \mathbf{E} \to \mathbf{B}$ with an image structure on an object $\mathbf{B}$ equipped with a path-object $(\mathbf{B}^{\to}, \beta)$. A comprehension structure with image on the morphism $p : \mathbf{E} \to \mathbf{B}$ is a right adjoint $\mathcal{P} : (\mathbf{E}, p) \to (\mathbf{B}^{\to}, \mathbf{cod})$ to the morphism $\mathbf{image} : (\mathbf{B}^{\to}, \mathbf{cod}) \to (\mathbf{E}, p)$ defined in (26) in the slice 2-category $\mathcal{K}/B$.*

A comprehension structure with image on $p : \mathbf{E} \to \mathbf{B}$ in the sense of Def. 25 comes equipped with a pair of adjunctions $\mathbf{id} : \mathbf{B} \leftrightarrows \mathbf{B}^{\to} : \mathbf{dom}$ and $\mathbf{image} : \mathbf{B}^{\to} \leftrightarrows \mathbf{E} : \mathcal{P}$ in the 2-category $\mathcal{K}$. From that, one easily deduces that

▶ **Proposition 26.** *Every comprehension structure with image (Def. 25) induces a comprehension structure with section defined as $\star = \mathbf{image} \circ \mathbf{id} : \mathbf{B} \to \mathbf{E}$ (Def. 16), where the right adjoint functor $[-] : \mathbf{E} \to \mathbf{B}$ is defined as the composite $[-] = \mathbf{dom} \circ \mathcal{P}$.*

We establish now the converse property which extends to every 2-category $\mathcal{K}$ the property stated in the introduction (Prop. 8) in the specific case of $\mathcal{K} = \mathbf{Cat}$. The statement extends [2], lemma 2.2.10 by relaxing the assumption that $p : \mathbf{E} \to \mathbf{B}$ is a bifibration.

▶ **Proposition 27.** *Suppose that $p : \mathbf{E} \to \mathbf{B}$ has an image structure in the 2-category $\mathcal{K}$ (in the sense of Def. 24). In that case, every comprehension structure with section (in the sense of Def. 16) defines a comprehension structure with image (in the sense of Def. 25).*

Proof. Suppose that $p : \mathbf{E} \to \mathbf{B}$ has an image structure and at the same time a comprehension structure with section $\star : \mathbf{E} \to \mathbf{B}$ in the 2-category $\mathcal{K}$. The 2-cell $\iota : [-] \to p : \mathbf{E} \to \mathbf{B}$ mentioned in Prop. 12 defines a morphism in the 2-category $\mathcal{K}(\mathbf{E}, \mathbf{B})$, and thus an object in the 2-category $\mathcal{K}(\mathbf{E}, \mathbf{B})^{\to}$. By definition of the path object $(\mathbf{B}^{\to}, \beta)$ in Def. 18, the 2-cell $\iota : [-] \to p : \mathbf{E} \to \mathbf{B}$ induces an object of the 2-category $\mathcal{K}(\mathbf{E}, \mathbf{B}^{\to})$, and thus a morphism noted $\mathcal{P} : \mathbf{E} \to \mathbf{B}^{\to}$ and characterized by the equation

$$\mathbf{hom} \circ \mathcal{P} \quad = \quad \iota \quad : \quad [-] \Longrightarrow p \quad : \quad \mathbf{E} \longrightarrow \mathbf{B}$$

We want to show that this morphism $\mathcal{P} : \mathbf{E} \to \mathbf{B}^{\to}$ is right adjoint to the morphism $\mathbf{image} : \mathbf{B}^{\to} \to \mathbf{E}$ in the 2-category $\mathcal{K}$. To that purpose, we consider an object $\mathbf{X}$ of the 2-category $\mathcal{K}$ and a pair of morphisms $u : \mathbf{X} \to \mathbf{B}^{\to}$ and $S : \mathbf{X} \to \mathbf{E}$, and we exhibit a one-to-one relationship (see [10], Section 5.11) between the 2-cells φ and ψ of the form:

$$\begin{tikzcd} & \mathbf{B}^{\to} \arrow[dd, "\mathbf{image}"] \\ \mathbf{X} \arrow[ur, bend left, "u"] \arrow[dr, bend right, "S"'] \arrow[r, phantom, "\Downarrow \varphi"] & \\ & \mathbf{E} \end{tikzcd} \qquad \begin{tikzcd} & \mathbf{B}^{\to} \\ \mathbf{X} \arrow[ur, bend left, "u"] \arrow[dr, bend right, "S"'] \arrow[r, phantom, "\Downarrow \psi"] & \\ & \mathbf{E} \arrow[uu, "\mathcal{P}"'] \end{tikzcd} \tag{27}$$

The key observation is that a 2-cell ψ of that form is the same thing as a 2-cell (ψ_1, ψ_2) in the 2-category $\mathcal{K}//\mathcal{K}$ between the composite morphisms:

$$\begin{tikzcd} \mathbf{X} \arrow[r, "u"] \arrow[d, "\mathrm{id}_{\mathbf{X}}"'] & \mathbf{B}^{\to} \arrow[r, "\mathbf{dom}"] \arrow[d, "\mathbf{cod}"'] \arrow[dr, phantom, "\overset{\mathbf{hom}}{\Longleftarrow}"] & \mathbf{B} \arrow[d, "\mathrm{id}_{\mathbf{B}}"] \\ \mathbf{X} \arrow[r, "\mathbf{cod} \circ u"'] & \mathbf{B} \arrow[r, "\mathrm{id}_{\mathbf{B}}"'] & \mathbf{B} \end{tikzcd} \quad \xRightarrow{(\psi_1, \psi_2)} \quad \begin{tikzcd} \mathbf{X} \arrow[r, "S"] \arrow[d, "\mathrm{id}_{\mathbf{X}}"'] & \mathbf{E} \arrow[r, "{[-]}"] \arrow[d, "p"'] \arrow[dr, phantom, "\overset{\iota}{\Longleftarrow}"] & \mathbf{B} \arrow[d, "\mathrm{id}_{\mathbf{B}}"] \\ \mathbf{X} \arrow[r, "p \circ S"'] & \mathbf{B} \arrow[r, "\mathrm{id}_{\mathbf{B}}"'] & \mathbf{B} \end{tikzcd}$$

It follows from the existence of the adjunction in $\mathcal{K}//\mathcal{K}$ established in Prop. 17 that there is a one-to-one relationship between the pairs of 2-cells (ψ_1, ψ_2) in the 2-category $\mathcal{K}$ of the form above, and the pairs (φ_1, φ_2) of 2-cells in the 2-category $\mathcal{K}$ defining a 2-cell (φ_1, φ_2) in the 2-category $\mathcal{K}//\mathcal{K}$ between the composite morphisms:

$$\begin{tikzcd} \mathbf{X} \arrow[r, "u"] \arrow[d, "\mathrm{id}_{\mathbf{X}}"'] & \mathbf{B}^{\to} \arrow[r, "\mathbf{dom}"] \arrow[d, "\mathbf{cod}"'] \arrow[dr, phantom, "\overset{\mathbf{hom}}{\Longleftarrow}"] & \mathbf{B} \arrow[r, "\star"] \arrow[d, "\mathrm{id}_{\mathbf{B}}"] & \mathbf{E} \arrow[d, "p"] \\ \mathbf{X} \arrow[r, "\mathbf{cod} \circ u"'] & \mathbf{B} \arrow[r, "\mathrm{id}_{\mathbf{B}}"'] & \mathbf{B} \arrow[r, "\mathrm{id}_{\mathbf{B}}"'] & \mathbf{B} \end{tikzcd} \quad \xRightarrow{(\varphi_1, \varphi_2)} \quad \begin{tikzcd} \mathbf{X} \arrow[r, "S"] \arrow[d, "\mathrm{id}_{\mathbf{X}}"'] & \mathbf{E} \arrow[d, "p"] \\ \mathbf{X} \arrow[r, "p \circ S"'] & \mathbf{B} \end{tikzcd}$$

The definition of the morphism $\mathbf{image} : \mathbf{B}^{\to} \to \mathcal{E}$ and the cartesianity of the 2-cell $\lambda_{\mathbf{hom}}$ in (25) with respect to the functor $\mathcal{K}(\mathbf{X}, p) : \mathcal{K}(\mathbf{X}, \mathbf{E}) \to \mathcal{K}(\mathbf{X}, \mathbf{B})$ implies that there is a one-to-one relationship between the pairs of 2-cells (φ_1, φ_2) in the 2-category $\mathcal{K}//\mathcal{K}$ above, and the 2-cells φ of the form (27) in the 2-category $\mathcal{K}$. The end of the proof is easy. ◀

8 Illustration: inductive reasoning on functor algebras and dually, coinductive reasoning on functor coalgebras

Suppose given a functor $p : \mathcal{E} \to \mathcal{B}$ between two categories $\mathcal{B}$ and $\mathcal{E}$ equipped with endofunctors $F : \mathcal{B} \to \mathcal{B}$ and $G : \mathcal{E} \to \mathcal{E}$ and a distributivity law of the form (10). A well-known result by Beck [1] states that the distributivity law $\delta : F \circ p \Rightarrow p \circ G$ describes one specific lifting of the functor $p : \mathcal{E} \to \mathcal{B}$ to a functor $p' : \mathbf{Alg}_G(\mathcal{E}) \to \mathbf{Alg}_F(\mathcal{B})$ between the underlying categories of algebras, in such a way that the diagram below commutes:

$$\begin{tikzcd}
\mathbf{Alg}_G(\mathcal{E}) \arrow[r, "p'"] \arrow[d, "U"'] & \mathbf{Alg}_F(\mathcal{B}) \arrow[d, "U"] \\
\mathcal{E} \arrow[r, "p"] & \mathcal{B}
\end{tikzcd}$$

where U denotes in both cases the forgetful functor. One main reason for working at a 2-categorical level as we do in the present paper is to provide us with a simple and elegant recipe to characterize in just the same spirit inherited from Beck [1] when a comprehension structure with section $\star : \mathcal{B} \to \mathcal{E}$ on the functor $p : \mathcal{E} \to \mathcal{B}$ lifts to a comprehension structure with section $\star : \mathbf{Alg}_F(\mathcal{B}) \to \mathbf{Alg}_G(\mathcal{E})$ on the functor $p : \mathbf{Alg}_G(\mathcal{E}) \to \mathbf{Alg}_F(\mathcal{B})$. To that purpose, we consider the 2-category $\mathbf{Endo(Cat)}$ with objects the categories equipped with endofunctors, and with morphisms the functors equipped with a distributivity law à la Beck. Note that $\mathbf{Endo(Cat)}$ may be defined as the full sub-2-category of $\mathbf{Cat}//\mathbf{Cat}$ whose objects are of the form $(\mathcal{C}, \mathcal{C}, G : \mathcal{C} \to \mathcal{C})$. This leads us the question of characterizing when a comprehension structure with section $\star : \mathcal{B} \to \mathcal{E}$ on the functor $p : \mathcal{E} \to \mathcal{B}$ lifts to a *comprehension structure with section* in the 2-category $\mathbf{Endo(Cat)}$, where this definition should be understood in the 2-categorical sense of §4, Def. 16. We establish that

▶ **Proposition 28.** *Suppose given a comprehension structure with section* $([-], \star)$ *in* $\mathbf{Cat}$ *on a functor* $p : \mathcal{E} \to \mathcal{B}$ *between categories* $\mathcal{B}$ *and* $\mathcal{E}$ *equipped with endofunctors* $S : \mathcal{B} \to \mathcal{B}$ *and* $T : \mathcal{E} \to \mathcal{E}$*. There is a one-to-one correspondence between the liftings to* $\mathbf{Endo(Cat)}$ *of the comprehension structure with section* $([-], \star)$ *and the pairs of distributivity laws*

$$\delta \ : \ F \circ p \Longrightarrow p \circ G \ : \ \mathcal{E} \longrightarrow \mathcal{B} \qquad\qquad \sigma \ : \ G \circ \star \Longrightarrow \star \circ F \ : \ \mathcal{B} \longrightarrow \mathcal{E}$$

such that (1) the composite natural transformation

$$F \xRightarrow{equal} F \circ p \circ \star \xRightarrow{\delta \circ \star} p \circ G \circ \star \xRightarrow{p \circ \sigma} p \circ \star \circ F \xRightarrow{equal} F$$

is the identity and (2) the natural transformation σ *is reversible.*

It is worth mentioning that, in this situation, the comprehension functor $[-] : \mathcal{E} \to \mathcal{B}$ lifts as the pair $([-], \tilde{\sigma}) : (\mathcal{E}, G) \to (\mathcal{B}, F)$ where the distributivity law $\tilde{\sigma} : F \circ [-] \Rightarrow [-] \circ G$ is defined as the mate of the inverse $\sigma^{-1} : \star \circ F \Rightarrow G \circ \star$. We obtain that in this situation

▶ **Corollary 29.** *The comprehension structure with section* $\star : \mathcal{B} \to \mathcal{E}$ *lifts to a comprehension structure with section* $(\star, \sigma) : (\mathcal{B}, F) \to (\mathcal{E}, G)$ *which is left adjoint to comprehension* $([-], \tilde{\sigma}) : (\mathcal{E}, G) \to (\mathcal{B}, F)$ *and thus transports the initial* F*-algebra* μF *to the initial* G*-algebra* $\mu G = \star_{\mu F}$*.*

The result extends the main soundness theorem established by Fumex, Ghani and Johann in [3], Thm 2.10. It establishes in their terminology (see [2], Def. 4.3.1) that G defines an induction scheme for μF in p. The approach translates immediately by duality to the case of quotient structures, and provides in just the same way necessary and sufficient conditions to be in the situation of Corollary 10 and to characterize the terminal F-coalgebra as $\nu F = \star_{\nu F}$.

9 Conclusion

Our main purpose and achievement in this paper is to exhibit the 2-categorical structures secretly at work in the 1-categorical approach to comprehension structures traditionally found in categorical logic. Our work was motivated by the fibered approach to induction on algebras and coinduction on coalgebras recently developed by Fumex, Ghani and Johann [3, 2]. We understand our 2-categorical approach and statements (Cor. 9, 10 and 29) as providing the clean conceptual foundations underlying their soundness theorems. For lack of space, we did not treat here the proof-theoretical aspects of our 2-categorical description of comprehension structures. A natural direction would be to start from the recent multicategorical approach to induction [7] developed in the fibered style of Melliès and Zeilberger's refinement systems [11]. We leave that for future work.

References

1 Jon Beck. Distributive laws. In *Seminar on triples and categorical homology theory*, pages 119–140. Springer, 1969.

2 Clément Fumex. *Induction and coinduction schemes in category theory*. PhD thesis, University of Strathclyde, 2012.

3 Clément Fumex, Neil Ghani, and Patricia Johann. Indexed induction and coinduction, fibrationally. In *Algebra and Coalgebra in Computer Science: 4th International Conference, CALCO 2011, Winchester, UK, August 30-September 2, 2011, Proceedings*, volume 6859, page 176. Springer Science & Business Media, 2011.

4 IEEE Computer Society. *A Categorical Semantics of Constructions*, 1988. URL: `https://www.irif.fr/~ehrhard/pub/categ-sem-constr.pdf`.

5 B. Jacobs. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam, 1999.

6 Bart Jacobs. Comprehension categories and the semantics of type dependency. *Theor. Comput. Sci.*, 107(2):169–207, 1993. `doi:10.1016/0304-3975(93)90169-T`.

7 Farzad Jafarrahmani. Induction in the Fibred Multicategory. Master's thesis, LSV of ENS Cachan, and Theory Group within the School of Computer Science of University of Birmingham, 2020. supervised by Gilles Dowek and Noam Zeilberger.

8 G. M. Kelly and Ross Street. Review of the elements of 2-categories. In GregoryM Kelly, editor, *Category Seminar*, volume 420 of *Lecture Notes in Mathematics*, pages 75–103. Springer Berlin / Heidelberg, 1974. `doi:10.1007/bfb0063101`.

9 Max Kelly. Doctrinal adjunction. In *Category Seminar*, pages 257–280. Springer, 1974.

10 Paul-André Melliès. *Categorical semantics of linear logic*, pages 1–196. Number 27 in Panoramas et Synthèses. Société Mathématique de France, 2009.

11 Paul-André Melliès and Noam Zeilberger. Functors are type refinement systems. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015.*, 2015.

12 Sean Moss. *The Dialectica Models of Type Theory*. PhD thesis, University of Cambridge, 2018.

13 Ross Street. The formal theory of monads. *Journal of Pure and Applied Algebra*, 2(2):149–168, 1972.

14 Lawvere William. Equality in hyperdoctrines and comprehension schema as an adjoint functor. In *Proceedings of the AMS Symposium on Pure Mathematics XVII*, pages 1–14. AMS, 1970.

15 Ernst Zermelo. Untersuchungen über die grundlagen der mengenlehre. i. *Mathematische Annalen*, 65(2):261–281, 1908.

A Four alternative notions of comprehension structures

For the sake of completeness, we give the list below of four well-recognized notions of comprehension structures appearing in the literature.

Jacobs comprehension categories

The notion of comprehension category was introduced by Jacobs ([6], Def 4.1, p. 181, and [5], chapter 10.4, page 613). A comprehension category is defined there as a functor $\mathcal{P} : \mathcal{E} \to \mathcal{B}^{\to}$ satisfying that

1. the functor $p : \mathcal{E} \to \mathcal{B}$ defined as the composite functor $p = \mathbf{cod} \circ \mathcal{P}$ is a Grothendieck fibration,
2. the functor $\mathcal{P} : \mathcal{E} \to \mathcal{B}^{\to}$ is cartesian in the sense that it transports every p-cartesian morphism of $\mathcal{E}$ to a **cod**-cartesian morphism of $\mathcal{B}^{\to}$, which may be equivalently defined as a *pullback square* in the category $\mathcal{B}$.

A comprehension category is thus the same thing as a comprehension structure in the sense of Def. 1 where the underlying functor $p : \mathcal{E} \to \mathcal{B}$ is a Grothendieck fibration, and where the functor $\mathcal{P} : \mathcal{E} \to \mathcal{B}^{\to}$ induced from the functor $[-] : \mathcal{E} \to \mathcal{B}$ and from the natural transformation $\iota : [-] \Rightarrow p$ transports every p-cartesian morphism of $\mathcal{E}$ to a pullback square in the category $\mathcal{B}$. Note that in that case, the equality $p = \mathbf{cod} \circ \mathcal{P}$ holds by construction.

Ehrhard D-categories

The notion of D-category was introduced by Ehrhard [4]. Ehrhard's D-categories are also called *comprehension category with units* by Jacobs [6] def. 4.12, and *Ehrhard comprehension category* by Moss [12], p 22. A pre-D-category is defined in [4] (Section 2.1, def. 5) as a functor $p : \mathcal{E} \to \mathcal{B}$ equipped with a right adjoint functor $\star : \mathcal{B} \to \mathcal{E}$ such that the counit of the adjunction $p \dashv \star$ an isomorphism, or equivalently, that the functor $\star$ is fully faithful. The functor $\star : \mathcal{B} \to \mathcal{E}$ may be thus seen as a section of the functor $p : \mathcal{E} \to \mathcal{B}$ up to isomorphism. In the definition of a pre-D-category, the functor $\star : \mathcal{B} \to \mathcal{E}$ should also come equipped with a right adjoint functor $[-] : \mathcal{E} \to \mathcal{B}$. Finally, a D-category is defined in [4] (Section 2.1, def. 5) as a pre-D-category where the functor $p : \mathcal{E} \to \mathcal{B}$ is a Grothendieck fibration.

In his later reformulation [6] of the notion of D-category, Jacobs makes the extra assumption that the counit of the adjunction $p \dashv \star$ is the identity, and not just an isomorphism. This implies in particular that the functor $\star : \mathcal{B} \to \mathcal{E}$ is a section of the functor $p : \mathcal{E} \to \mathcal{B}$. A D-category is thus defined in [6] Def. 4.12 as a Grothendieck fibration $p : \mathcal{E} \to \mathcal{B}$ equipped with a terminal object functor $\star : \mathcal{B} \to \mathcal{E}$ which has a right adjoint noted $[-] : \mathcal{E} \to \mathcal{B}$. Here, by terminal object functor $s : \mathcal{B} \to \mathcal{E}$, one means a section of the functor $p : \mathcal{E} \to \mathcal{B}$ which transports every object A of the basis category $\mathcal{B}$ to a terminal object of the fiber $\mathcal{E}_A$ of the object A with respect to the functor $p : \mathcal{E} \to \mathcal{B}$. Note in particular that the terminal object functor $\star : \mathcal{B} \to \mathcal{E}$ is fully faithful and right adjoint to the functor $p : \mathcal{E} \to \mathcal{B}$.

A D-category in that sense is thus the same thing as a comprehension structure with section (Def. 4) where the functor $p : \mathcal{E} \to \mathcal{B}$ is a Grothendieck fibration, and where the section $\star : \mathcal{B} \to \mathcal{E}$ is moreover right adjoint to $p : \mathcal{E} \to \mathcal{B}$. As mentioned above, this last point means that the section $\star : \mathcal{B} \to \mathcal{E}$ is the terminal object function which associates to every object A of the category $\mathcal{B}$ the terminal object in its fiber $\mathcal{E}_A$ with respect to the functor $p : \mathcal{E} \to \mathcal{B}$.

Fumex tC-opfibrations

The notion of tC-opfibration was introduced by Fumex in his PhD thesis, ([2] p. 38, def. 2.2.2.) A tC-category is defined there as a Grothendieck opfibration $p : \mathcal{E} \to \mathcal{B}$ with a fully faithful section $\star : \mathcal{B} \to \mathcal{E}$. The section $\star : \mathcal{B} \to \mathcal{E}$ is moreover required to have a right adjoint noted $[-] : \mathcal{E} \to \mathcal{B}$. Note that, given an object A of the basis category $\mathcal{B}$, one does not require that the object s_A is terminal in the fiber $\mathcal{E}_A$ of the object A. This is one main difference with Ehrhard's notion of D-category.

A tC-category is thus the same thing as a comprehension structure with section in the sense of Def. 4 where the functor $p : \mathcal{E} \to \mathcal{B}$ is a Grothendieck opfibration and where the section $\star : \mathcal{B} \to \mathcal{E}$ is moreover fully faithful. At this stage, it is important to observe that every Grothendieck opfibration has an image structure in the sense of Def. 21, or equivalently, in the sense of Def. 24 for the specific case $\mathcal{K} = \mathbf{Cat}$. From this follows, by Prop. 27, that a tC-category is in fact the same thing as a comprehension structure with image in sense of Def. 7, where the functor $p : \mathcal{E} \to \mathcal{B}$ is moreover a Grothendieck opfibration and where the section $\star : \mathcal{B} \to \mathcal{E}$ is fully faithful.

Lawvere categories

The notion of Lawvere category was introduced by Jacobs in [6], p 190, as a way to reflect the work by Lawvere [14] on hyperdoctrines in categorical logic. A Lawvere category is defined as a Grothendieck bifibration $p : \mathcal{E} \to \mathcal{B}$ with a terminal object in each fiber, defining a functor $\star : \mathcal{B} \to \mathcal{E}$, and such that the (ordinary) functor

$$f \mapsto \Sigma_f \star_{(dom f)} : \mathcal{B}^{\to} \to \mathcal{E}$$

induced by the left fibration structure has a right adjoint $[-] : \mathcal{E} \to \mathcal{B}^{\to}$, verifying $\mathbf{cod} \circ [-] = p$, and such that the unit and counit are vertical (their image by $\mathbf{cod}$ and p is the identity). Note that every Lawvere category is a tC-opfibration in the sense of Fumex [2]. A Lawvere category is thus the same thing as a comprehension structure with image in sense of Def. 7 where the functor $p : \mathcal{E} \to \mathcal{B}$ is a Grothendieck bifibration and where the section $\star : \mathcal{B} \to \mathcal{E}$ is the terminal object functor.

A Complete Normal-Form Bisimilarity for Algebraic Effects and Handlers

Dariusz Biernacki
University of Wrocław, Poland
dabi@cs.uni.wroc.pl

Sergueï Lenglet
Université de Lorraine, Nancy, France
serguei.lenglet@univ-lorraine.fr

Piotr Polesiuk
University of Wrocław, Poland
ppolesiuk@cs.uni.wroc.pl

Abstract

We present a complete coinductive syntactic theory for an untyped calculus of algebraic operations and handlers, a relatively recent concept that augments a programming language with unprecedented flexibility to define, combine and interpret computational effects. Our theory takes the form of a normal-form bisimilarity and its soundness w.r.t. contextual equivalence hinges on using so-called context variables to test evaluation contexts comprising normal forms other than values. The theory is formulated in purely syntactic elementary terms and its completeness demonstrates the discriminating power of handlers. It crucially takes advantage of the clean separation of effect handling code from effect raising construct, a distinctive feature of algebraic effects, not present in other closely related control structures such as delimited-control operators.

2012 ACM Subject Classification Theory of computation → Control primitives

Keywords and phrases algebraic effect, handler, behavioral equivalence, bisimilarity

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.7

Acknowledgements We would like to thank the anonymous reviewers for their helpful comments on the presentation of this work.

1 Introduction

Algebraic effects with handlers [22, 3] have become a popular technique of programming with computational effects such as exceptions, mutable state or nondeterminism. Their strength lies in their modularity, as it is possible to easily combine several effects thanks to the separation between syntax and semantics. Indeed, effects themselves are just syntactic constructs which do not carry any meaning; their semantics is given by the handlers, which come into play when an interpretation of an effect is needed for the computation to go through.

As an informal example, borrowed from [8], consider the reader effect ask, which returns a hidden value when triggered. An effect is used as a labeled operation, e.g., as in $\mathsf{do}_{\mathsf{ask}}\,() + \mathsf{do}_{\mathsf{ask}}\,() + 2$, and its meaning is given by a handler, as in

$$\mathsf{handle}\ \mathsf{do}_{\mathsf{ask}}\,() + \mathsf{do}_{\mathsf{ask}}\,() + 2\ \{\mathsf{ask}\colon x{,}k \to k\ 5;\ \mathsf{ret}\ y \to y\}$$

The handler specifies how it interprets the ask effect by the expression $x{,}k \to k\ 5$, where x stands for the value the effect operation is applied to (which is not used in this example), and k for its *continuation* or *resumption*, i.e., the rest of the computation, which includes the handler itself. Here, the handler simply passes 5 to the continuation, so that $\mathsf{do}_{\mathsf{ask}}\,() + \mathsf{do}_{\mathsf{ask}}\,() + 2$

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 7; pp. 7:1–7:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

eventually reduces to 12. Once the expression inside the handler is a value, it is passed to the *return clause* $\mathsf{ret}\ y \to y$, which in our case simply returns the result. Any expression can be used in an effect handler, including one making use of the continuation several times or not at all; for example, in

$$\mathsf{handle}\ \mathsf{do}_{\mathsf{ask}}\ () + \mathsf{do}_{\mathsf{ask}}\ () + 2\ \{\mathsf{ask}\colon x,k \to 13; \mathsf{ret}\ y \to y\}$$

the handler throws away the continuation when called the first time and returns 13, which is then the final result of the computation. Multiple effects can be used in an expression, which are then interpreted by a single handler, or by successive handlers enclosing the expression. The order of the handlers then specifies the semantics of all the effects combined.

While handlers make combining multiple effects programmer friendly, reasoning about the behavior of programs with effects and handlers appears to be inherently challenging, mainly due to the non-local transfer of control involved in effect handling. When it comes to the issue of program equivalence, the standard notion considered in calculi modeling programming languages, typically based on λ-calculi, is *contextual equivalence* [20], which requires program phrases to behave the same when plugged in any context. The quantification over all contexts makes this relation hard to use in practice, so one usually looks for more tractable characterizations of contextual equivalence, either in the form of logical relations [24] or coinductively defined bisimilarities [1, 17, 27].

In the presence of algebraic effects and handlers, the situation is even more interesting, because we have to take into account the possibility that the testing context may interpret any non-handled effects the two programs being tested might use. There exist some works on formal techniques for reasoning about program equivalence in calculi with algebraic effects, but they either do not include handlers in the language [16, 15, 14] or are directed by a type structure of the calculus [8] (we discuss related work in detail in Section 4). None of them, however, focuses on the control structure of a full calculus of algebraic effects and handlers (where effects are interpreted dynamically, unlike, e.g., in [14]) and in isolation from other concepts such as types. Algebraic effects are intimately related to delimited-control operators [12, 21], for which bisimulation theories have been studied extensively [4], yet they differ in a very essential way, as we argue in this work.

In this paper, we show that it is possible to characterize contextual equivalence in an untyped calculus with algebraic effects and handlers with one of the simplest notions of equivalence, namely *normal-form* (or *open*) bisimilarity [25, 17]. In a normal-form bisimilarity proof one compares open terms by reducing them to normal forms, which are then decomposed into bisimilar subterms. In a language with algebraic effects, we have to consider extra normal forms – programs with effects that have not been handled. More importantly, we have to observe how a context may handle an effect and its continuation. To this end, we introduce an extended calculus where contexts can be abstractly represented with *context variables*, a concept we used in our previous work on normal-form bisimulations for abortive continuations [7]. Such variables can be observed and discriminated upon by the bisimilarity that is defined for the extended calculus. Extending the calculus is a critical step in obtaining sound and complete bisimilarity, but it should be seen just as a tool for studying the plain calculus. When restricted to the plain calculus, the bisimilarity relates exactly those terms that are equivalent w.r.t. the contextual equivalence in the plain calculus.

In many calculi, the decomposition of normal forms as done in normal-form bisimilarity is usually too fine-grained and distinguishes programs that are in fact contextually equivalent [17]. The result of this paper shows that handlers contain sufficient discriminating power for normal-form bisimilarity to be complete w.r.t. contextual equivalence. It contrasts with other continuation-manipulating constructs such as (multi-prompted) delimited-control operators, for which finding a complete normal-form bisimilarity remains an open issue [4].

$$
\begin{array}{rcll}
\mathrm{Lbl} \ni l & & & \text{(effect labels)} \\
\mathrm{Var} \ni f, k, x, y, z & & & \text{(variables)} \\
\mathrm{Val} \ni u, v, w & ::= & x \mid \lambda x.e & \text{(values)} \\
\mathrm{Exp} \ni e & ::= & v \mid e_0\, e_1 \mid \mathsf{do}_l\, e \mid \mathsf{handle}\ e\, \{H; r\} & \text{(expressions)} \\
H & ::= & l_1 \colon h_1; \ldots; l_n \colon h_n & \\
h & ::= & x,k \to e & \text{(effect handlers)} \\
r & ::= & \mathsf{ret}\ x \to e & \text{(return clause)}
\end{array}
$$

Figure 1 Syntax of λ_{eff}.

The rest of this paper is organized as follows. In Section 2, we present the syntax, semantics, and contextual equivalence of the plain calculus λ_{eff}, the minimal calculus with effects and handlers we consider for our study. In Section 3, we define the normal-form bisimilarity for the extended calculus and prove its soundness and completeness. We also define *up-to techniques*, proof techniques meant to simplify equivalence proofs, and we illustrate how the bisimilarity and these techniques can be used on examples. Additionally, we pinpoint the difference between algebraic effects and delimited-control operators and how it affects the definition of a normal-form bisimulation. In Section 4, we discuss related work, and we conclude in Section 5. The appendix contains the soundness and completeness proof sketches.

2 The Calculus λ_{eff}

Syntax. The calculus λ_{eff}, whose syntax is given in Figure 1, extends the λ-calculus with labeled effects $\mathsf{do}_l\, e$ and handlers $\mathsf{handle}\ e\, \{H; r\}$, where H is a list of effect handlers $l_i \colon x_i,k_i \to e_i$ and r is a return clause $\mathsf{ret}\ x \to e'$. The order of the list is irrelevant, but we assume the labels $l_1 \ldots l_n$ to be pairwise distinct. In a handler $x_i,k_i \to e_i$, the variable x_i represents the argument of the effect, while k_i stands for its continuation (or *resumption*). We write $\mathsf{lbl}(e)$ for the set of effect labels l that label do expressions in e. The choice of having a handler interpret several effects at once makes writing examples easier, but does not affect the behavioral theory: the definitions of the equivalences are the same if the handler takes care of one effect only.

An effect handler $x_i,k_i \to e_i$ binds x_i and k_i in e_i, and a λ-abstraction $\lambda x.e$ or a return clause $\mathsf{ret}\ x \to e$ bind x in e. We use the standard notions of free variables ($\mathsf{fv}(e)$ is the set of free variables in e), closed and open expressions, and we work modulo α-conversion of the bound variables. A variable is called fresh if it does not occur in any of the entities under consideration.

We assume the standard call-by-value Church encoding of natural numbers, booleans (true, false, $\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$), unit ($()$), and the sequence expression ($e_1; e_2$) that we use in examples and in the proof of completeness.

Reduction semantics. We fix a call-by-value, left-to-right reduction strategy for λ_{eff} by defining the syntax of evaluation contexts as follows.

$$
\mathrm{ECtx} \ni E \quad ::= \quad \square \mid E\, e \mid v\, E \mid \mathsf{do}_l\, E \mid \mathsf{handle}\ E\, \{H; r\}
$$

We write $E[e]$ for the plugging of the expression e into the context E, and $e\{v/x\}$ for the usual capture-avoiding substitution of x by v in e. Given a context E, we define the set of effects it handles, written $\mathrm{hl}(E)$, as follows.

$$\begin{aligned}
\mathrm{hl}(\square) &\triangleq \emptyset \\
\mathrm{hl}(E\ e) &\triangleq \mathrm{hl}(E) \\
\mathrm{hl}(v\ E) &\triangleq \mathrm{hl}(E) \\
\mathrm{hl}(\mathsf{do}_l\ E) &\triangleq \mathrm{hl}(E) \\
\mathrm{hl}(\mathsf{handle}\ E\,\{l_1\!:h_1;\ldots;l_n\!:h_n;r\}) &\triangleq \mathrm{hl}(E) \cup \{l_1,\ldots,l_n\}
\end{aligned}$$

When writing expressions, we sometimes decorate a context with a label it does not handle, i.e., writing $E^{\bar{l}}$ if $l \notin \mathrm{hl}(E)$. Typically, we write $E^{\bar{l}}[\mathsf{do}_l\ v]$ for an expression where the effect l cannot be handled by E.

The reduction semantics of λ_{eff} is given by the following rules.

$$\begin{array}{rcll}
(\lambda x.e)\ v & \mapsto & e\{v/x\} & \\
\mathsf{handle}\ v\,\{H;\mathsf{ret}\ x \to e\} & \mapsto & e\{v/x\} & \\
E[\mathsf{do}_l\ v] & \mapsto & e\{v/x\}\{\lambda z.E[z]/k\} & \text{if } E = \mathsf{handle}\ E'^{\bar{l}}\,\{H;r\} \\
 & & & \text{and } l\!:x,k \to e \in H \\
 & & & \text{and } z \text{ is fresh} \\
E[e] & \to & E[e'] & \text{if } e \mapsto e'
\end{array}$$

We write $\to^*$ for the reflexive and transitive closure of $\to$. In the third rule, we see that the effect $\mathsf{do}_l\ v$ is interpreted by the first enclosing handler, as $E = \mathsf{handle}\ E'^{\bar{l}}\,\{H;r\}$ and E' does not handle l. The handler has access not only to the argument v of the effect, but also to its continuation, represented as a function $\lambda z.E[z]$. Note that the handler itself is part of the captured continuation, meaning that it can handle further effects when the continuation is resumed.[1] If a handler obtains a value (second rule), there are no more effects to handle and the value is passed to the return clause. The semantics is deterministic, as it can be shown that an expression is either a normal form or can be uniquely decomposed into a redex and an evaluation context.

▶ **Example 1.** Let us consider the example from the introduction:

$$e \triangleq \mathsf{handle}\ \mathsf{do}_{\mathsf{ask}}\ () + \mathsf{do}_{\mathsf{ask}}\ () + 2\,\{\mathsf{ask}\!: x,k \to k\ 5; \mathsf{ret}\ y \to y\}$$

If $E \triangleq \mathsf{handle}\ \square\,\{\mathsf{ask}\!: x,k \to k\ 5; \mathsf{ret}\ y \to y\}$ and z is a fresh variable, then e reduces as follows:

$$\begin{aligned}
e &\to (\lambda z.E[z + \mathsf{do}_{\mathsf{ask}}\ () + 2])\ 5 \\
&\to \mathsf{handle}\ 5 + \mathsf{do}_{\mathsf{ask}}\ () + 2\,\{\mathsf{ask}\!: x,k \to k\ 5; \mathsf{ret}\ y \to y\} \\
&\to (\lambda z.E[5 + z + 2])\ 5 \\
&\to \mathsf{handle}\ 5 + 5 + 2\,\{\mathsf{ask}\!: x,k \to k\ 5; \mathsf{ret}\ y \to y\} \\
&\to^* \mathsf{handle}\ 12\,\{\mathsf{ask}\!: x,k \to k\ 5; \mathsf{ret}\ y \to y\} \\
&\to 12
\end{aligned}$$

[1] Such handlers are known as *deep* handlers as opposed to *shallow* handlers also considered in the literature [21].

Normal forms and contextual equivalence. When considering open expressions, normal forms can be of the following kinds.

▶ **Lemma 2.** *An open expression e is a normal form iff e is a value, or $e = E[x\,v]$ for some E, x, and v, or $e = E^{\bar{l}}[\mathsf{do}_l\,v]$ for some E, l, and v.*

Values and expressions $E[x\,v]$ (referred to as *open-stuck terms*) are usual normal forms which can already be found in the plain λ-calculus. The expression $E^{\bar{l}}[\mathsf{do}_l\,v]$ cannot reduce further, as E cannot handle the effect l; we refer to such normal forms as *control-stuck terms*. Closed normal forms are either λ-abstractions or control-stuck terms.

Contextual equivalence equates expressions behaving the same in all contexts. In the presence of multiple closed normal forms as in λ_{eff}, several definitions of contextual equivalence are possible, depending on whether we observe termination of evaluation in general, or to specific, meaningful normal forms – usually values. It turns out that such a choice does not matter in λ_{eff}, as the definitions coincide; we explain why after presenting the definition we use in this paper. We let C range over arbitrary contexts, i.e., expressions with a hole $\square$. We write $e \Downarrow_{\mathrm{v}}$ if there is a value v, such that $e \to^* v$, and $e \Uparrow$ if e reduces infinitely, e.g., $\Omega \Uparrow$, where $\Omega = (\lambda x.x\,x)\,(\lambda x.x\,x)$.

▶ **Definition 3.** *Two expressions e_1 and e_2 are contextually equivalent, written $e_1 \equiv e_2$, if for all contexts C, such that $C[e_1]$ and $C[e_2]$ are closed, we have $C[e_1] \Downarrow_{\mathrm{v}}$ iff $C[e_2] \Downarrow_{\mathrm{v}}$.*

It can be shown that this definition introduces the same notion of contextual equivalence as the one in which we observe simply termination of evaluation, instead of evaluation to a value. The reason is that for any control-stuck term $e_1 = E_1{}^{\bar{l}}[\mathsf{do}_l\,v_1]$, taking

$$C = \mathsf{handle}\,\square\,\{l\colon x,k \to \Omega;\, \mathsf{ret}\ x \to x\}$$

we have $C[e_1] \Uparrow$, whereas $C[v_2] \Downarrow_{\mathrm{v}}$ for any value v_2, and taking

$$C' = \mathsf{handle}\,\square\,\{l\colon x,k \to x;\, \mathsf{ret}\ x \to x\}$$

we have $C'[e_1] \Downarrow_{\mathrm{v}}$, whereas $C'[e_2] \Uparrow$ for any e_2 such that $e_2 \Uparrow$. Thus, we can always build a context that preserves non-termination and evaluation to a value, but that at the same time coerces a control-stuck term to either a non-terminating expression (C) or to a value (C'). The two contextual equivalences therefore coincide, a situation which differs from other context-manipulating constructs such as delimited-control operators [4].

3 Normal-Form Bisimilarity

We first informally introduce our notion of normal-form bisimilarity, before giving its definition and discussing its soundness and completeness. We also explain why, in spite of the relationship between handlers and multi-prompted delimited continuations, it is more difficult to define a complete normal-form bisimilarity for the latter than for the former.

3.1 Informal Presentation

Normal-form bisimulation reduces expressions to normal forms and decomposes them into related subterms; for example, an open-stuck term $E_1[x\,v_1]$ is related to e_2 if e_2 reduces to a similar term such that the contexts and values are pairwise related. Compared to the plain λ-calculus [17, 7], we have to consider an extra normal form – control-stuck terms – but also take into account the fact that contexts may handle effects.

Dealing with control-stuck terms follows the same logic as for open-stuck terms: ${E_1}^{\overline{l}}[\mathsf{do}_l\ v_1]$ is related to e_2 if e_2 reduces to a control-stuck term with related values and contexts. Comparing contexts requires more care, as it depends how they are used. A context ${E_1}^{\overline{l}}$ surrounding a control-stuck term can only be captured and then plugged with a value, so it is enough to test them with a fresh variable representing that value. Such contexts represent resumptions (delimited continuations, really) that are bound to the continuation variable k in effect handlers and used to obtain suitable interpretation of the effect.

In contrast, in an open-stuck term $E_1[x\ v_1]$, the application may reduce to an effect which could be handled by E_1. Testing such contexts with only a fresh variable is not enough as it would relate $\square$ and $\mathsf{handle}\ \square\ \{H; \mathsf{ret}\ x \to x\}$, two contexts which behave differently as soon as they are plugged with an effect handled by H. We need to observe which handlers are surrounding the context holes, but without requiring the sequence of handled effects to be exactly the same. Indeed, successive "identity handlers" $h \overset{\triangle}{=} x,k \to k\ x$ should be related if they handle the same effects, even in a different order: the context $\mathsf{handle}\ \mathsf{handle}\ \square\ \{l_2 \colon h; \mathsf{ret}\ x \to x\}\ \{l_1 \colon h; \mathsf{ret}\ x \to x\}$ is expected to be equivalent to the context $\mathsf{handle}\ \mathsf{handle}\ \square\ \{l_1 \colon h; \mathsf{ret}\ x \to x\}\ \{l_2 \colon h; \mathsf{ret}\ x \to x\}$.

A simple way to compare the handlers behaviors is to plug the contexts with a control-stuck term $\mathsf{do}_l\ x$ for a fresh x and for any l (handled by the contexts). However, such a testing term is not strong enough, as it would relate a handler which throws away the continuation to one that does not, e.g., $E_1 = \mathsf{handle}\ \square\ \{l \colon x,k \to x; \mathsf{ret}\ x \to x\}$ and $E_2 = \mathsf{handle}\ \square\ \{l \colon x,k \to k\ x; \mathsf{ret}\ x \to x\}$. We need to account for the fact that control-stuck terms may be surrounded with a context without introducing a quantification over these contexts which would go against the principles behind normal-form bisimulation. We do so by extending the syntax of the calculus with *context variables*, a construct we introduced in previous works to track the whereabouts of contexts captured by control operators [7, 4]. In a control-stuck term $\alpha^{\overline{l}}[\mathsf{do}_l\ x]$, the context variable $\alpha^{\overline{l}}$ stands for a context which does not handle l, and its presence allows to distinguish between the two contexts E_1 and E_2.

Adding context variables to λ_{eff} generates new normal forms of the shape $E[\alpha^{\overline{l}}[v]]$ and $E[\alpha^{\overline{l}}[E'^{\overline{l'}}[\mathsf{do}_{l'}\ v]]]$ (with $l \neq l'$), where the computation is stuck because we do not know which context $\alpha^{\overline{l}}$ stands for. The bisimulation deals with these normal forms in a very regular way, simply asking to reduce to a normal form of the same shape with related contexts and values. In the end, the definition we obtain (Definition 5) follows the usual pattern of normal-form bisimulation – the only subtlety being in how to compare contexts – and yet the resulting bisimilarity is sound and complete w.r.t. the contextual equivalence of the extended calculus. More importantly, the restriction of the bisimilarity to plain calculus terms yields the contextual equivalence for the plain calculus.

3.2 Extended Calculus

As explained in the previous section, we extend the syntax of λ_{eff} with context variables in order to observe how contexts are captured when effects are triggered. We assume a set CVar of context variables, ranged over by α and β. Similar to evaluation contexts, we decorate these variables with an effect it does not handle: the variable $\alpha^{\overline{l}}$ is a context variable standing for a context which does not handle l. In particular, when considering a control-stuck term, the context variable is always decorated with an effect label. Moreover, we write $\alpha^{\overline{l}} \neq \beta^{\overline{l'}}$ if $l \neq l'$ or $\alpha \neq \beta$.

We extend the syntax of expressions and evaluation contexts as follows.

$$e ::= \ldots \mid \alpha^{\overline{l}}[e] \qquad\qquad E ::= \ldots \mid \alpha^{\overline{l}}[E]$$

We write $\mathsf{cv}(e)$ for the set of context variables occurring in e. We adapt the definition of hl so that $\mathrm{hl}(\alpha^{\bar{l}}[E]) \triangleq (\mathrm{Lbl} \setminus \{l\}) \cup \mathrm{hl}(E)$, as $\alpha^{\bar{l}}$ stands for a context not handling l but which may potentially handle any other label. While the reduction rules themselves are the same, the semantics of the extended calculus is still affected by the change in the grammar of evaluation contexts. In particular, it admits more normal forms than the plain λ_{eff}.

▶ **Lemma 4.** *An open expression e is a normal form in the extended calculus iff e is a value, or $e = E[x\ v]$ for some E, x, and v, or $e = E^{\bar{l}}[\mathsf{do}_l\ v]$ for some E, l, and v, or $e = E[\alpha^{\bar{l}}[v]]$ for some E, $\alpha^{\bar{l}}$, and v, or $e = E_1[\alpha^{\bar{l}}[E_2{}^{\overline{l'}}[\mathsf{do}_{l'}\ v]]]$ for some E_1, E_2, v, $\alpha^{\bar{l}}$ and l' such that $l \neq l'$.*

We refer to normal forms of the shape $E[\alpha^{\bar{l}}[v]]$ as *context-stuck terms* and those of the shape $E_1[\alpha^{\bar{l}}[E_2{}^{\overline{l'}}[\mathsf{do}_{l'}\ v]]]$ as *control/context-stuck terms*. The latter differ from control-stuck terms of the form $E^{\bar{l}}[\mathsf{do}_l\ v]$, because $\alpha^{\bar{l}}$ may be replaced by a context handling l', so even if E_1 does not handle l' we cannot consider $E_1[\alpha^{\bar{l}}[E_2{}^{\overline{l'}}]]$ as a context not handling l'.

A context variable cannot be bound, therefore an open term may contain context variables or free expression variables. In contrast, an expression or context is closed if it does not have any context variable or free expression variable.

Given an expression e, a context variable $\alpha^{\bar{l}}$ and a context $E^{\bar{l}}$, we define the *context substitution* $e\{E^{\bar{l}}/\alpha^{\bar{l}}\}$ so that $(\alpha^{\bar{l}}[e])\{E^{\bar{l}}/\alpha^{\bar{l}}\} \triangleq E^{\bar{l}}[e\{E^{\bar{l}}/\alpha^{\bar{l}}\}]$, and the substitution is recursively propagated to the sub-expressions in the other cases.

3.3 Definition

We define the bisimulation for the extended calculus using the notion of *diacritical progress* we developed in a previous work [2, 6], which distinguishes between *active* and *passive* clauses. Roughly, passive clauses are between simulation states which should be considered equal, while active clauses are between states where actual progress is taking place. This distinction does not change the notions of bisimulation or bisimilarity, but it simplifies the soundness proof of the bisimilarity. It also allows for the definition of powerful *up-to techniques*, functions on relations meant to simplify bisimilarity proofs. For normal-form bisimilarity, our framework enables up-to techniques which respect η-expansion [7], a necessary condition to reach completeness.

Given a relation $\mathcal{R}$ on expressions, we extend it to values and evaluation contexts in the following way.

$$\frac{v_1\ x\ \mathcal{R}\ v_2\ x \qquad x \text{ fresh}}{v_1\ \mathcal{R}^{\mathsf{v}}\ v_2} \qquad\qquad \frac{E_1[x]\ \mathcal{R}\ E_2[x] \qquad x \text{ fresh}}{E_1\ \mathcal{R}^{\mathsf{r}}\ E_2}$$

$$\frac{E_1[x]\ \mathcal{R}\ E_2[x] \qquad \forall l \in \mathrm{hl}(E_1) \cup \mathrm{hl}(E_2). E_1[\alpha^{\bar{l}}[\mathsf{do}_l\ x]]\ \mathcal{R}\ E_2[\alpha^{\bar{l}}[\mathsf{do}_l\ x]] \qquad x, \alpha^{\bar{l}} \text{ fresh}}{E_1\ \mathcal{R}^{\mathsf{c}}\ E_2}$$

The $\cdot^{\mathsf{v}}$ extension compares values by simply applying them to a fresh variable; such a test, compliant with η-expansion [7], is valid because λ-abstractions are the only values of our language. As explained in Section 3.1, we consider two extensions for evaluation contexts, as it depends how these are used: $\cdot^{\mathsf{r}}$ is used when we know the contexts are plugged only with values (resumptions), while $\cdot^{\mathsf{c}}$ assumes that they can be filled with any expression, including an effectful one. As a result, $\cdot^{\mathsf{c}}$ compares how the contexts deal with the effects they may handle (the ones in $\mathrm{hl}(E_1) \cup \mathrm{hl}(E_2)$), by testing them with an expression $\alpha^{\bar{l}}[\mathsf{do}_l\ x]$ built using a fresh context variable $\alpha^{\bar{l}}$ which can be observed during the bisimulation game.

We define progress, bisimulation and bisimilarity using these extensions.

▶ **Definition 5.** *A relation $\mathcal{R}$ progresses to $\mathcal{S}$, $\mathcal{T}$ written $\mathcal{R} \rightarrowtail \mathcal{S}, \mathcal{T}$, if $\mathcal{R} \subseteq \mathcal{S}$, $\mathcal{S} \subseteq \mathcal{T}$, and $e_1 \mathrel{\mathcal{R}} e_2$ implies:*

- *if $e_1 \rightarrow e_1'$, then there exists e_2' such that $e_2 \rightarrow^* e_2'$ and $e_1' \mathrel{\mathcal{T}} e_2'$;*
- *if $e_1 = v_1$, then there exists v_2 such that $e_2 \rightarrow^* v_2$ and $v_1 \mathrel{\mathcal{S}^{\mathsf{v}}} v_2$;*
- *if $e_1 = E_1[x\, v_1]$, then there exist E_2 and v_2 such that $e_2 \rightarrow^* E_2[x\, v_2]$, $E_1 \mathrel{\mathcal{T}^{\mathsf{c}}} E_2$, and $v_1 \mathrel{\mathcal{T}^{\mathsf{v}}} v_2$;*
- *if $e_1 = E_1{}^{\overline{l}}[\mathsf{do}_l\, v_1]$, then there exist E_2 and v_2 such that $e_2 \rightarrow^* E_2{}^{\overline{l}}[\mathsf{do}_l\, v_2]$, $E_1{}^{\overline{l}} \mathrel{\mathcal{T}^{\mathsf{r}}} E_2{}^{\overline{l}}$, and $v_1 \mathrel{\mathcal{T}^{\mathsf{v}}} v_2$;*
- *if $e_1 = E_1[\alpha^{\overline{l}}[v_1]]$, then there exist E_2 and v_2 such that $e_2 \rightarrow^* E_2[\alpha^{\overline{l}}[v_2]]$, $E_1 \mathrel{\mathcal{S}^{\mathsf{c}}} E_2$, and $v_1 \mathrel{\mathcal{S}^{\mathsf{v}}} v_2$;*
- *if $e_1 = E_1[\alpha^{\overline{l}}[E_1'{}^{\overline{l'}}[\mathsf{do}_{l'}\, v_1]]]$ with $l \neq l'$, then there exist E_2, $E_2'{}^{\overline{l'}}$, and v_2 such that $e_2 \rightarrow^* E_2[\alpha^{\overline{l}}[E_2'{}^{\overline{l'}}[\mathsf{do}_{l'}\, v_2]]]$, $E_1 \mathrel{\mathcal{T}^{\mathsf{c}}} E_2$, $E_1'{}^{\overline{l'}} \mathrel{\mathcal{T}^{\mathsf{r}}} E_2'{}^{\overline{l'}}$, and $v_1 \mathrel{\mathcal{T}^{\mathsf{v}}} v_2$;*
- *the symmetric of the above conditions on e_2.*

A normal-form bisimulation is a relation $\mathcal{R}$ such that $\mathcal{R} \rightarrowtail \mathcal{R}, \mathcal{R}$, and normal-form bisimilarity $\approx$ is the union of all normal-form bisimulations.

As pointed out before, the clauses dealing with normal forms are very similar, simply requiring e_2 to reduce to a normal form of the same kind, and then decomposing these normal forms into pairwise related subterms. We just have to be careful in using $\cdot^{\mathsf{r}}$ only for the contexts used as resumptions.

We progress towards $\mathcal{S}$ in the value and context-stuck term clauses and $\mathcal{T}$ in the others; the former are passive while the latter are active. Our framework prevents some up-to techniques from being applied after a passive transition. For values, we want to forbid the application of bisimulation up to context as it would be unsound: we could deduce that $v_1\, x$ and $v_2\, x$ are equivalent for all v_1 and v_2 just by building a candidate relation containing v_1 and v_2. Similarly, for context-stuck terms, we prevent the application of bisimulation up to substitution of context variables, as we could also relate any v_1 and v_2 from a candidate containing $\alpha^{\overline{l}}[v_1]$ and $\alpha^{\overline{l}}[v_2]$ by replacing the context variable with $\Box\, x$.

▶ **Example 6.** We consider the handler of Example 1 for the reader effect, where we generalize the hidden value 5 to a given variable z:

$$E_1 \triangleq \mathsf{handle}\, \Box\, \{\mathsf{ask}\colon x, k \rightarrow k\; z; \mathsf{ret}\; x \rightarrow x\}$$

Alternatively, the reader effect can be interpreted by the following handler obtained from the standard handler for mutable state:

$$E_2 \triangleq (\mathsf{handle}\, \Box\, \{\mathsf{ask}\colon x, k \rightarrow \lambda y.k\; y\; y; \mathsf{ret}\; x \rightarrow \lambda y.x\})\; z$$

The context E_2 applies the handler to the current value of the state and let the handling code of the operation(s) access it through a λ-abstraction. (We would obtain a standard handler for mutable state by adding the clause $\mathsf{set}\colon x, k \rightarrow \lambda y.k\; y\; x$ handling the operation set which sets the value of the state.)

We show that these two handlers for the reader effect are equivalent by establishing the equivalence between the contexts $E_1 \approx^{\mathsf{c}} E_2$.

Proof. Relating $E_1[x]$ and $E_2[x]$ for a fresh x is easy, as $E_1[x] \to x$ and $E_2[x] \to (\lambda y.x)\ z \to x$. Testing with $\alpha^{\bar{l}}[\mathsf{do}_l\ x]$ and defining $E_2' \triangleq \mathsf{handle}\ \square\ \{l{:}\ x,k \to \lambda y.k\ y\ y; \mathsf{ret}\ x \to \lambda y.x\}$, we get

$$E_1[\alpha^{\bar{l}}[\mathsf{do}_l\ x]] \to^2 E_1[\alpha^{\bar{l}}[z]]$$

$$E_2[\alpha^{\bar{l}}[\mathsf{do}_l\ x]] \to (\lambda y.(\lambda y'.E_2'[\alpha^{\bar{l}}[y']])\ y\ y)\ z \to^2 E_2'[\alpha^{\bar{l}}[z]]z = E_2[\alpha^{\bar{l}}[z]]$$

We obtain two context-stuck terms, for which we need to relate identical variables and the contexts E_1 and E_2 we want to equate in the first place. In the end, we can easily build a bisimulation $\mathcal{R}$ such that $E_1\ \mathcal{R}^{\mathsf{c}}\ E_2$. ◀

3.4 Soundness and Up-to Techniques

In our framework [6] as in the works we extend [18, 23], proving that the bisimilarity is *compatible* – preserved by contexts – amounts to showing that a form of bisimulation up to context is valid, as explained after Lemma 10. We slightly reformulate our most recent work [6] to make it simpler but expressive enough it can be applied to λ_{eff}.

In what follows, we use s, f, g to range over *monotone* functions on relations, i.e., functions such that $\mathcal{R} \subseteq \mathcal{S}$ implies $\mathsf{f}(\mathcal{R}) \subseteq \mathsf{f}(\mathcal{S})$ for any $\mathcal{R}$, $\mathcal{S}$. We extend $\cup$ to functions so that for all $\mathcal{R}$, $(\mathsf{f} \cup \mathsf{g})(\mathcal{R}) = \mathsf{f}(\mathcal{R}) \cup \mathsf{g}(\mathcal{R})$. We define an ordering $\sqsubseteq$ on functions so that $\mathsf{f} \sqsubseteq \mathsf{g}$ if for all $\mathcal{R}$, $\mathsf{f}(\mathcal{R}) \subseteq \mathsf{g}(\mathcal{R})$, which is itself extended pointwise to pairs of functions.

As pointed out before, because of the distinction between passive and active clauses, not all up-to techniques can be applied in all clauses. In fact, we decompose an up-to technique into a pair of functions (s, f), where s can be used in passive clauses while f cannot.

▶ **Definition 7.** *A pair of monotone functions* (s, f) *is an up-to technique if for all* $\mathcal{R}$*,* $\mathcal{R} \rightarrowtail \mathsf{s}(\mathcal{R}), \mathsf{f}(\mathcal{R})$ *implies* $\mathcal{R} \subseteq \approx$*.*

In an up-to technique (s, f), s is said *strong* while f is said *weak*. Instead of proving directly that a pair is an up-to technique, we consider a sufficient criterion based on *respectfulness*[2] and the largest respectful pair, called the *diacritical companion* (u, w): if a pair (s, f) is below the companion, then it is an up-to technique.

The diacritical companion is defined using notions of *evolution* on monotone functions which can be seen as the higher-order counterpart of progress on relations. We decompose diacritical progress $\mathcal{R} \rightarrowtail \mathcal{S}, \mathcal{T}$ into *passive* progress $\mathcal{R} \overset{\mathsf{p}}{\rightarrowtail} \mathcal{S}$ and *active* progress $\mathcal{R} \overset{\mathsf{a}}{\rightarrowtail} \mathcal{T}$ to define different kinds of evolution.

▶ **Definition 8.** *Let* f, g *be monotone functions.*

- f passively evolves *to* g*, written* $\mathsf{f} \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{g}$*, if for all* $\mathcal{R}$*,* $\mathcal{S}$*,* $\mathcal{R} \overset{\mathsf{p}}{\rightarrowtail} \mathcal{S}$ *implies* $\mathsf{f}(\mathcal{R}) \overset{\mathsf{p}}{\rightarrowtail} \mathsf{g}(\mathcal{S})$*;*
- f actively evolves *to* g*, written* $\mathsf{f} \overset{\mathsf{a}}{\rightsquigarrow} \mathsf{g}$*, if for all* $\mathcal{R}$*,* $\mathcal{S}$*,* $\mathcal{R} \overset{\mathsf{a}}{\rightarrowtail} \mathcal{S}$ *implies* $\mathsf{f}(\mathcal{R}) \overset{\mathsf{a}}{\rightarrowtail} \mathsf{g}(\mathcal{S})$*;*
- f restrictively evolves *to* g*, written* $\mathsf{f} \overset{\mathsf{p}|\mathsf{a}}{\rightsquigarrow} \mathsf{g}$*, if for all* $\mathcal{R}$*,* $\mathcal{S}$*,* $\mathcal{R} \overset{\mathsf{p}}{\rightarrowtail} \mathcal{R} \overset{\mathsf{a}}{\rightarrowtail} \mathcal{S}$ *implies* $\mathsf{f}(\mathcal{R}) \overset{\mathsf{a}}{\rightarrowtail} \mathsf{g}(\mathcal{S})$*.*

Passive and active evolutions express the idea that f becomes g in respectively passive and active clauses. Restricted evolution allows a relation $\mathcal{R}$ to do some administrative step (passive progress) before doing some active progress, as long as we stay in $\mathcal{R}$. For λ_{eff}, it means that we can reduce a term to a value before doing some active progress with it.

[2] Our previous work [6] is built on the notion of *compatibility*, but the notion of progress we use in this paper makes Definition 9 correspond to respectfulness instead. See [26, 23, 6] for a discussion on the difference between the two notions.

▶ **Definition 9.** *A pair of monotone functions* (s, f) diacritically evolves *to* $(\mathsf{s}', \mathsf{f}'), (\mathsf{s}'', \mathsf{f}'')$*, written* $(\mathsf{s}, \mathsf{f}) \rightsquigarrow (\mathsf{s}', \mathsf{f}'), (\mathsf{s}'', \mathsf{f}'')$ *if*

$$\mathsf{s} \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{s}' \qquad\qquad \mathsf{f} \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{f}' \qquad\qquad \mathsf{s} \overset{\mathsf{a}}{\rightsquigarrow} \mathsf{f}'' \qquad\qquad \mathsf{f} \overset{\mathsf{p}|\mathsf{a}}{\rightsquigarrow} \mathsf{f}''$$

A pair (s, f) *is* respectful *if* $(\mathsf{s}, \mathsf{f}) \rightsquigarrow (\mathsf{s}, \mathsf{f}), (\mathsf{s}, \mathsf{f})$*. The* diacritical companion (u, w) *is the largest respectful pair.*

In words, the bisimulations of diacritical evolution are exactly respectful pairs, and its bisimilarity is the diacritical companion. Among other properties, we can show that any pair below the companion (including the companion itself) is an up-to technique.

▶ **Lemma 10.** *The following hold:*

- *if* $(\mathsf{s}, \mathsf{f}) \sqsubseteq (\mathsf{u}, \mathsf{w})$*, then* (s, f) *is an up-to technique;*
- $\mathsf{u} \sqsubseteq \mathsf{w}$*;*
- $\mathsf{w}(\approx) = \approx$*.*

The second inequality implies that any strong function can also be used as a weak one, justifying why such a function is said "strong", as it can be applied without restriction in any clause. The last equality states that the weak companion preserves bisimilarity, so for any $\mathsf{f} \sqsubseteq \mathsf{w}$, we also have $\mathsf{f}(\approx) \subseteq \approx$. If f is a contextual closure function (if $e_1 \mathcal{R} e_2$ then $C[e_1]\ \mathsf{f}(\mathcal{R})\ C[e_2]$), showing that it is below w is enough to deduce that $\approx$ is compatible.

The remaining question is how to prove that a given pair (s, f) is below the companion. In this paper, we use a degenerate but sufficient version of a theorem in our previous work [6, Theorem 4.12]. Let id be the identity on relations. We define $\mathsf{S}(\mathsf{s})$ inductively as the smallest function verifying:

- for all $\mathsf{g} \in \{\mathsf{id}, \mathsf{s}, \mathsf{u}\}$, $\mathsf{g} \sqsubseteq \mathsf{S}(\mathsf{s})$;
- for all $\mathsf{g} \in \{\mathsf{id}, \mathsf{s}, \mathsf{u}\}$, $\mathsf{g} \circ \mathsf{S}(\mathsf{s}) \sqsubseteq \mathsf{S}(\mathsf{s})$, $\mathsf{S}(\mathsf{s}) \circ \mathsf{g} \sqsubseteq \mathsf{S}(\mathsf{s})$, $\mathsf{g} \cup \mathsf{S}(\mathsf{s}) \sqsubseteq \mathsf{S}(\mathsf{s})$, and $\mathsf{S}(\mathsf{s}) \cup \mathsf{g} \sqsubseteq \mathsf{S}(\mathsf{s})$;

and $\mathsf{W}(\mathsf{s}, \mathsf{f})$ inductively as the smallest function verifying:

- for all $\mathsf{g} \in \{\mathsf{id}, \mathsf{s}, \mathsf{f}, \mathsf{w}\}$, $\mathsf{g} \sqsubseteq \mathsf{W}(s)$;
- for all $\mathsf{g} \in \{\mathsf{id}, \mathsf{s}, \mathsf{f}, \mathsf{w}\}$, $\mathsf{g} \circ \mathsf{W}(\mathsf{s}, \mathsf{f}) \sqsubseteq \mathsf{W}(\mathsf{s}, \mathsf{f})$, $\mathsf{W}(\mathsf{s}, \mathsf{f}) \circ \mathsf{g} \sqsubseteq \mathsf{W}(\mathsf{s}, \mathsf{f})$, $\mathsf{g} \cup \mathsf{W}(\mathsf{s}, \mathsf{f}) \sqsubseteq \mathsf{W}(\mathsf{s}, \mathsf{f})$, and $\mathsf{W}(\mathsf{s}, \mathsf{f}) \cup \mathsf{g} \sqsubseteq \mathsf{W}(\mathsf{s}, \mathsf{f})$;

The function $\mathsf{S}(\mathsf{s})$ is the smallest function built from s, id, and u stable by composition and union, while $\mathsf{W}(\mathsf{s}, \mathsf{f})$ is the smallest function built from s, f, id, and w stable by composition and union. Including u and w in their definition means that any function already proved respectively strong or weak is below respectively $\mathsf{S}(\mathsf{s})$ or $\mathsf{W}(\mathsf{s}, \mathsf{f})$.

▶ **Theorem 11.** *Let* (s, f) *be monotone functions. If*

$$\mathsf{s} \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{S}(\mathsf{s}) \qquad\qquad \mathsf{f} \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{S}(\mathsf{s}) \circ \mathsf{f} \circ \mathsf{S}(\mathsf{s}) \qquad\qquad \mathsf{s} \overset{\mathsf{a}}{\rightsquigarrow} \mathsf{W}(\mathsf{s}, \mathsf{f}) \qquad\qquad \mathsf{f} \overset{\mathsf{p}|\mathsf{a}}{\rightsquigarrow} \mathsf{W}(\mathsf{s}, \mathsf{f})$$

then $(\mathsf{s}, \mathsf{f}) \sqsubseteq (\mathsf{u}, \mathsf{w})$ *and* (s, f) *is an up-to technique.*

The idea of the theorem is to see how s and f evolve and prove that the results of their evolutions is below what is on the right of the arrows. Any combination of weak functions can be obtained after an active or restricted evolution, but only strong functions can be used after a passive one, except that f can be used once. This constraint on f makes the soundness proofs of the most interesting up-to techniques of λ_{eff} more difficult (cf. Appendix A).

We define the up-to functions we consider for λ_{eff} in Figure 2. The first four are usual and can be found in many variants of the λ-calculus [7, 4]. The function red is the usual bisimulation up to reduction, where expressions can be related after some reduction steps, while refl equates any expression with itself. The function subst allows to replace a variable in related expressions with related values. Finally, lam is compatibility w.r.t. λ-abstraction.

Resumption position predicate.

$$\frac{}{\mathsf{resum}(\alpha^{\overline{l}}, x)} \qquad \frac{\mathsf{resum}(\alpha^{\overline{l}}, e)}{\mathsf{resum}(\alpha^{\overline{l}}, \lambda x.e)} \qquad \frac{\mathsf{resum}(\alpha^{\overline{l}}, e_1) \qquad \mathsf{resum}(\alpha^{\overline{l}}, e_2)}{\mathsf{resum}(\alpha^{\overline{l}}, e_1\ e_2)} \qquad \frac{\mathsf{resum}(\alpha^{\overline{l}}, e)}{\mathsf{resum}(\alpha^{\overline{l}}, \mathsf{do}_{l'}\ e)}$$

$$\frac{\mathsf{resum}(\alpha^{\overline{l}}, e) \qquad \forall l_i\!: x_i{,}k_i \to e_i \in H, \mathsf{resum}(\alpha^{\overline{l}}, e_i) \qquad \mathsf{resum}(\alpha^{\overline{l}}, e')}{\mathsf{resum}(\alpha^{\overline{l}}, \mathsf{handle}\ e\ \{H; \mathsf{ret}\ x \to e'\})} \qquad \frac{\mathsf{resum}(\alpha^{\overline{l}}, v)}{\mathsf{resum}(\alpha^{\overline{l}}, \alpha^{\overline{l}}[v])}$$

$$\frac{\mathsf{resum}(\alpha^{\overline{l}}, v)}{\mathsf{resum}(\alpha^{\overline{l}}, \alpha^{\overline{l}}[\mathsf{do}_l\ v])} \qquad \frac{\mathsf{resum}(\alpha^{\overline{l}}, e) \qquad \beta^{\overline{l'}} \neq \alpha^{\overline{l}}}{\mathsf{resum}(\alpha^{\overline{l}}, \beta^{\overline{l'}}[e])}$$

Up-to techniques.

$$\frac{e_1 \to^* e_1' \qquad e_2 \to^* e_2' \qquad e_1'\ \mathcal{R}\ e_2'}{e_1\ \mathsf{red}(\mathcal{R})\ e_2} \qquad \frac{}{e\ \mathsf{refl}(\mathcal{R})\ e}$$

$$\frac{e_1\ \mathcal{R}\ e_2 \qquad v_1\ \mathcal{R}^{\mathsf{v}}\ v_2}{e_1\{v_1/x\}\ \mathsf{subst}(\mathcal{R})\ e_2\{v_2/x\}} \qquad \frac{e_1\ \mathcal{R}\ e_2}{\lambda x.e_1\ \mathsf{lam}(\mathcal{R})\ \lambda x.e_2}$$

$$\frac{e_1\ \mathcal{R}\ e_2}{\alpha^{\overline{l}}[e_1]\ \mathsf{cvar}(\mathcal{R})\ \alpha^{\overline{l}}[e_2]} \qquad \frac{e_1\ \mathcal{R}\ e_2 \qquad E_1^{\overline{l}}\ \mathcal{R}^{\mathsf{c}}\ E_2^{\overline{l}}}{e_1\{E_1^{\overline{l}}/\alpha^{\overline{l}}\}\ \mathsf{csubst}(\mathcal{R})\ e_2\{E_2^{\overline{l}}/\alpha^{\overline{l}}\}}$$

$$\frac{e_1\ \mathcal{R}\ e_2 \qquad E_1^{\overline{l}}\ \mathcal{R}^{\mathsf{r}}\ E_2^{\overline{l}} \qquad \mathsf{resum}(\alpha^{\overline{l}}, e_1) \qquad \mathsf{resum}(\alpha^{\overline{l}}, e_2)}{e_1\{E_1^{\overline{l}}/\alpha^{\overline{l}}\}\ \mathsf{rsubst}(\mathcal{R})\ e_2\{E_2^{\overline{l}}/\alpha^{\overline{l}}\}}$$

Figure 2 Up-to functions for λ_{eff}.

The remaining functions are more specific to λ_{eff}. The function cvar plugs related terms into any context variable. This variable can then be replaced with contexts using either csubst or rsubst, depending whether the contexts behave as resumptions or not. In the latter case, the contexts should be related with $\cdot^{\mathsf{r}}$, and the context variable should be in *resumption position*, a condition we check with the predicate resum, defined in Figure 2. Roughly, $\mathsf{resum}(\alpha^{\overline{l}}, e)$ means that $\alpha^{\overline{l}}$ is about to be captured – i.e., plugged with an effect $\mathsf{do}_l\ v$ – or has already been captured, and is therefore plugged with a value.

The functions cvar, csubst, and rsubst can be used to define a more conventional bisimulation up to evaluation context, similar to the one of the plain λ-calculus [7].

▶ **Lemma 12.** *If $e_1\ \mathcal{R}\ e_2$ and $E_1\ \mathcal{R}^{\mathsf{c}}\ E_2$, then $E_1[e_1]\ \mathsf{csubst}(\mathsf{cvar}(\mathcal{R}) \cup \mathsf{id})\ E_2[e_2]$.*

We simply plug e_1 and e_2 into a fresh context variable which is then replaced with E_1 and E_2.

The functions we define are strong, except for csubst and rsubst.

▶ **Theorem 13.** *For all $\mathsf{s} \in \{\mathsf{refl}, \mathsf{id}, \mathsf{red}, \mathsf{subst}, \mathsf{lam}, \mathsf{cvar}\}$, we have $\mathsf{s} \sqsubseteq \mathsf{u}$. For all $\mathsf{f} \in \{\mathsf{csubst}, \mathsf{rsubst}\}$, we have $\mathsf{f} \sqsubseteq \mathsf{w}$.*

The proofs for the strong techniques are simple or as in the plain λ-calculus [7]; we sketch the proof for csubst and rsubst in the appendix. It is not surprising that these two functions are weak, as they essentially behave as bisimulation up to context, which is also weak in the plain λ-calculus. As explained in Section 3.3, they cannot be used in the passive clauses, i.e., when relating values or context-stuck terms.

Because cvar and csubst are up-to techniques, the bisimulation up to evaluation context is also sound, from which we deduce that $\approx$ is compatible w.r.t. evaluation contexts using Lemma 10. Thanks to lam, we know it is also preserved by λ-abstraction, so we can show the bisimilarity is compatible, from which we deduce it is a valid proof technique for the contextual equivalence of the plain calculus.

▶ **Corollary 14.** *Let e_1 and e_2 be expressions of the plain calculus. If $e_1 \approx e_2$, then $e_1 \equiv e_2$.*

Indeed, if $e_1 \approx e_2$, then for all contexts C, $C[e_1] \approx C[e_2]$ because $\approx$ is compatible. If $C[e_1] \Downarrow_{\mathrm{v}}$, then $C[e_2] \Downarrow_{\mathrm{v}}$ simply by definition of the bisimilarity.

The up-to techniques we define are useful beyond simply proving soundness of the bisimilarity; they can simplify the equivalence proof of two given terms, as illustrated by the following examples.

▶ **Example 15.** Dal Lago and Gavazzo [14] propose an example where two fixed-point combinators are signaling each β-reduction with a tick effect; we modify it so that the two expressions are equivalent with handlers (but the tick effect is now arbitrary). Let

$$e_1 \triangleq \lambda y.\mathsf{do}_{\mathsf{tick}}\,(\Delta_y\;\Delta_y) \qquad \Delta_y \triangleq \lambda x.(\mathsf{do}_{\mathsf{tick}}\,y)\;\lambda z.\mathsf{do}_{\mathsf{tick}}\,(x\;x\;z)$$

$$e_2 \triangleq \Theta\,\Theta \qquad \Theta \triangleq \lambda x.\lambda y.\mathsf{do}_{\mathsf{tick}}\,((\mathsf{do}_{\mathsf{tick}}\,y)\;\lambda z.\mathsf{do}_{\mathsf{tick}}\,(x\;x\;y\;z))$$

We prove these expressions are bisimilar up to, by building a candidate relation $\mathcal{R}$ incrementally, starting from e_1 and e_2.

Proof. The term e_1 is a value, and $e_2 \to \lambda y.\mathsf{do}_{\mathsf{tick}}\,((\mathsf{do}_{\mathsf{tick}}\,y)\;\lambda z.\mathsf{do}_{\mathsf{tick}}\,(\Theta\,\Theta\;y\;z))$, so we need to relate the bodies of the λ-abstractions. We have a reduction $\mathsf{do}_{\mathsf{tick}}\,(\Delta_y\;\Delta_y) \to \mathsf{do}_{\mathsf{tick}}\,((\mathsf{do}_{\mathsf{tick}}\,y)\;\lambda z.\mathsf{do}_{\mathsf{tick}}\,(\Delta_y\;\Delta_y\;z))$; the resulting term is control-stuck, which we relate to $\mathsf{do}_{\mathsf{tick}}\,((\mathsf{do}_{\mathsf{tick}}\,y)\;\lambda z.\mathsf{do}_{\mathsf{tick}}\,(\Theta\,\Theta\;y\;z))$ which is also control-stuck. The arguments of the effect are the same, and we need to relate the two contexts $\mathsf{do}_{\mathsf{tick}}\,(\Box\;\lambda z.\mathsf{do}_{\mathsf{tick}}\,(\Delta_y\;\Delta_y\;z))$ and $\mathsf{do}_{\mathsf{tick}}\,(\Box\;\lambda z.\mathsf{do}_{\mathsf{tick}}\,(\Theta\,\Theta\;y\;z))$.

Plugging them with a fresh variable, we obtain two open-stuck terms, meaning that we need to relate the two identical contexts $\mathsf{do}_{\mathsf{tick}}\,\Box$ and the values $\lambda z.\mathsf{do}_{\mathsf{tick}}\,(\Delta_y\;\Delta_y\;z)$ and $\lambda z.\mathsf{do}_{\mathsf{tick}}\,(\Theta\,\Theta\;y\;z)$. These last two values are related up to lambda and evaluation context if $\mathcal{R}$ contains $\Delta_y\;\Delta_y$ and $\Theta\,\Theta\;y$, and the bisimulation proof for these two expressions is the same as for e_1 and e_2. In the end, taking $\mathcal{R} \triangleq \{(e_1, e_2), (\Delta_y\,\Delta_y, \Theta\,\Theta\,y)\}$, we can show that $\mathcal{R}$ is a bisimulation up to refl, red, lam, and up to context, i.e., up to cvar and csubst. Note that we are allowed to use the latter weak technique when comparing open-stuck terms, as it is an active clause. ◀

▶ **Example 16.** We write E_R for the reader effect of Example 6, and consider the following handler to express backtracking.

$$E_{BT} \triangleq \mathsf{handle}\,\Box\,\{\mathsf{fail}{:}\,x{,}k \to ();\mathsf{flip}{:}\,x{,}k \to (\lambda z.k\;\mathsf{false})\;(k\;\mathsf{true});\,\mathsf{ret}\;x \to x\}$$

$$E_R \triangleq \mathsf{handle}\,\Box\,\{\mathsf{ask}{:}\,x{,}k \to k\;z;\,\mathsf{ret}\;x \to x\}$$

We prove that the two effects commute by showing that $E_{BT}[E_R] \approx^{\mathsf{c}} E_R[E_{BT}]$.

Sketch. We show that the relation $\mathcal{R}$ given by the following rules is a bisimulation up-to.

$$\frac{}{E_{BT}[E_R[v]] \mathrel{\mathcal{R}} E_R[E_{BT}[v]]} \qquad \frac{}{E_{BT}[E_R[\alpha^{\overline{l}}[\mathsf{do}_l\, x]]] \mathrel{\mathcal{R}} E_R[E_{BT}[\alpha^{\overline{l}}[\mathsf{do}_l\, x]]]}$$

$$\frac{e_1 \mathrel{\mathsf{red}(\mathcal{R})} E_R[e_2] \qquad z \notin \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2)}{(\lambda z.e_1)\ (E_{BT}[E_R[\alpha^{\overline{l}}[\mathsf{do}_l\, x]]]) \mathrel{\mathcal{R}} E_R[(\lambda z.e_2)\ (E_{BT}[\alpha^{\overline{l}}[\mathsf{do}_l\, x]])]}$$

The pair of the first rule is straightforward to check as each expression evaluates to v. For the second rule, the interesting cases are when l is an effect handled by E_{BT} or E_R. If $l = \mathsf{fail}$, the two expressions evaluate to (). If $l = \mathsf{ask}$, they evaluate to respectively $E_{BT}[E_R[\alpha^{\overline{l}}[z]]]$ and $E_R[E_{BT}[\alpha^{\overline{l}}[z]]]$, which are context-stuck terms and for which we can easily check the bisimulation requirements.

If $l = \mathsf{flip}$, then the expressions of the second rule reduce to respectively

$$(\lambda z.((\lambda y.E_{BT}[E_R[\alpha^{\overline{l}}[y]]])\ \mathsf{false}))\ E_{BT}[E_R[\alpha^{\overline{l}}[\mathsf{true}]]], \text{ and}$$

$$E_R[(\lambda z.((\lambda y.E_{BT}[\alpha^{\overline{l}}[y]])\ \mathsf{false}))\ E_{BT}[\alpha^{\overline{l}}[\mathsf{true}]]].$$

To compare these context-stuck terms, we plug the two contexts with a fresh variable and a fresh control-stuck terms. When plugged with a fresh variable, we obtain $E_{BT}[E_R[\alpha^{\overline{l}}[\mathsf{false}]]]$ and $E_R[E_{BT}[\alpha^{\overline{l}}[\mathsf{false}]]]$, for which we can again easily check the bisimulation clause. With control-stuck terms, we obtain expressions related by the third rule defining $\mathcal{R}$. Checking bisimulation for the third rule is done by a similar case analysis on l and concludes the proof. ◀

3.5 Completeness

In this section we show that for any two expressions e_1 and e_2 in the plain calculus, if $e_1 \equiv e_2$, then $e_1 \approx e_2$. To this end, we first observe that if $e_1 \equiv e_2$, then $e_1 \equiv_{\mathrm{E}} e_2$, where $\equiv_{\mathrm{E}}$ is a relation on expressions in the extended calculus, defined as follows.

▶ **Definition 17.** *We write $e_1 \equiv_{\mathrm{E}} e_2$ if for all evaluation contexts E (from the extended calculus), and substitutions σ (i.e., finite mappings from variables to values and from context variables to contexts), such that $E[e_1]\sigma$ and $E[e_2]\sigma$ are closed expressions in the plain calculus, we have $E[e_1]\sigma \Downarrow_{\mathrm{v}}$ iff $E[e_2]\sigma \Downarrow_{\mathrm{v}}$.*

▶ **Lemma 18.** *If $e_1 \equiv e_2$, then $e_1 \equiv_{\mathrm{E}} e_2$.*

Proof. Assume that $e_1 \equiv e_2$ and take any evaluation context E and closing substitution σ, such that $E[e_1]\sigma \Downarrow_{\mathrm{v}}$. Then, it must be the case that $E[e_2]\sigma \Downarrow_{\mathrm{v}}$ as well, since otherwise e_1 and e_2 would be distinguished by the following context:

$$C = (\lambda x_1 \ldots . \lambda x_n . E\sigma)\ v_1 \ldots v_n$$

assuming $\mathsf{dom}(\sigma) = \{x_1, \ldots x_n, \alpha_1, \ldots, \alpha_m\}$ and $\sigma(x_i) = v_i$ for $1 \leq i \leq n$. ◀

The main lemma of this section establishes that $\equiv_{\mathrm{E}}$ is a bisimulation, which, by Lemma 18, implies completeness of $\approx$ w.r.t. $\equiv$.

▶ **Lemma 19.** *$\equiv_{\mathrm{E}}$ is a bisimulation.*

Proof. The proof consists in a case-by-case verification of the conditions stated in Definition 5 for the candidate relation $\equiv_{\mathrm{E}}$. Here we present one of the most representative cases that, in our opinion, illustrates best the power of the calculus and the techniques used in the remaining cases.

Case: $\boldsymbol{e_1 = E_1[\alpha^{\bar{l}}[v_1]]}$ and $\boldsymbol{e_1 \equiv_{\mathrm{E}} e_2}$. We need to show that there exist E_2 and v_2 such that: (1) $e_2 \to^* E_2[\alpha^{\bar{l}}[v_2]]$, (2) $v_1 \equiv^{\mathsf{v}}_{\mathrm{E}} v_2$, and (3) $E_1 \equiv^{\mathsf{c}}_{\mathrm{E}} E_2$.

To prove (1), we take a fresh label l', and we define a substitution σ as follows:

$$\begin{array}{rcll} \sigma(x) & = & \lambda y.\Omega & \text{for } x \in \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\ \sigma(\beta^{\overline{l''}}) & = & \mathsf{handle}\,\square\,\{H_{l''}; \mathsf{ret}\ x \to \Omega\} & \text{for } \beta^{\overline{l''}} \in \mathsf{cv}(e_1) \cup \mathsf{cv}(e_2) \text{ and } \beta^{\overline{l''}} \neq \alpha^{\bar{l}} \\ \sigma(\alpha^{\bar{l}}) & = & \mathsf{do}_{l'}\,\square & \end{array}$$

where $H_{l''} = l_1\!: x,k \to \Omega; \ldots; l_n\!: x,k \to \Omega$ and $\{l_1, \ldots, l_n\} = \mathsf{lbl}(e_1) \cup \mathsf{lbl}(e_2) - \{l''\}$, and we consider a context $E = \mathsf{handle}\,\square\,\{l'\!: x,k \to x; \mathsf{ret}\ x \to \Omega\}$. It is easy to see that $E[e_1]\sigma \Downarrow_{\mathrm{v}}$ and that if e_2 evaluates to a normal form which is not $E_2[\alpha^{\bar{l}}[v_2]]$ for some E_2 and v_2, then either $E[e_2]\sigma \Uparrow$ or $E[e_2]\sigma$ reduces to a control-stuck term (the latter case occurs when e_2 itself reduces to a control-stuck term $E_2[\mathsf{do}_{l''}\ v_2]$).

To prove (2), we take a fresh variable z, a context E and a closing substitution σ, and we assume that $E[v_1\ z]\sigma \Downarrow_{\mathrm{v}}$. To see that $E[v_2\ z]\sigma \Downarrow_{\mathrm{v}}$ as well, we construct a substitution σ' and a context E' such that $E'[e_i]\sigma' \Downarrow_{\mathrm{v}}$ iff $E[v_i\ z]\sigma \Downarrow_{\mathrm{v}}$ for $i = 1, 2$. To this end we take fresh labels l', get and put (the latter two to encode a binary state as an algebraic effect), and we define σ' to be equal to σ everywhere, except for $\alpha^{\bar{l}}$:[3]

$$\sigma'(\alpha^{\bar{l}}) = \sigma(\alpha^{\bar{l}})[(\lambda x.\mathsf{if}\ \mathsf{do}_{\mathsf{get}}\ ()\ \mathsf{then}\ (\mathsf{do}_{\mathsf{put}}\ \mathsf{false};\ \mathsf{do}_{l'}\ x)\ \mathsf{else}\ x)\ \square]$$

along with

$$\begin{aligned} E'_b &= (\mathsf{handle}\ E''\ \{\mathsf{get}\!: x,k \to \lambda y.k\ y\ y;\ \mathsf{put}\!: x,k \to \lambda y.k\ ()\ x; \mathsf{ret}\ x \to \lambda y.x\})\ b \\ E'' &= \mathsf{handle}\,\square\,\{l'\!: x,k \to E[x\ z]; \mathsf{ret}\ x \to x\}. \end{aligned}$$

where $b \in \{\mathsf{true}, \mathsf{false}\}$. Let us notice that

$$E'_{\mathsf{true}}[e_i]\sigma' \to^* E'_{\mathsf{false}}[E_i[\mathsf{do}_{l'}\ v_i]]\sigma' \to^* E'_{\mathsf{false}}[E[v_i\ z]]\sigma'$$

The idea is to use $\alpha^{\bar{l}}$, the single synchronization point of e_1 and e_2 available, in such a way that the first time $\alpha^{\bar{l}}$ is used, $E'_{\mathsf{true}}[e_i]\sigma'$ reduces to an expression behaving like $E[v_i\ z]\sigma$. To ensure this, we make sure that any subsequent uses of $\alpha^{\bar{l}}$ (it could occur in v_i or E) actually mean $\sigma(\alpha^{\bar{l}})$. But when the state is set to false, the λ-abstraction in $\sigma'(\alpha^{\bar{l}})$ behaves like the identity, and filling the hole of $\sigma'(\alpha^{\bar{l}})$ with a value v simply passes v to $\sigma(\alpha^{\bar{l}})$. Filling it with a control-stuck term $E'^{\overline{l'}}[\mathsf{do}_{l'}\ v]$ allows $\sigma(\alpha^{\bar{l}})$ to eventually handle the effect, capturing a context equivalent to $(\lambda z.z)\ E'^{\overline{l'}}$. In the end, $E'_{\mathsf{false}}[E[v_i\ z]]\sigma'$ behaves like $E[v_i\ z]\sigma$, up to a few additional reduction steps.

To prove (3), we have to show: (a) $E_1[z] \equiv_{\mathrm{E}} E_2[z]$ for a fresh variable z, and (b) $E_1[\alpha^{\overline{l''}}[\mathsf{do}_{l''}\ z]] \equiv_{\mathrm{E}} E_2[\alpha^{\overline{l''}}[\mathsf{do}_{l''}\ z]]$ for any l'' and fresh $\alpha^{\overline{l''}}$ and z. Assuming we compare expressions using E and σ in both cases, we proceed as in (2), except that in (a) we take

$$E'' = \mathsf{handle}\,\square\,\{l'\!: x,k \to E[k\ z]; \mathsf{ret}\ x \to x\}$$

and in (b) we take

$$E'' = \mathsf{handle}\,\square\,\{l'\!: x,k \to E[k\ (\alpha^{\overline{l''}}[\mathsf{do}_{l''}\ z])]; \mathsf{ret}\ x \to x\}.$$

The remaining cases are proved similarly and can be found in Appendix B. ◀

▶ **Corollary 20.** *For any expressions e_1 and e_2 in the plain calculus, if $e_1 \equiv e_2$, then $e_1 \approx e_2$.*

[3] Strictly speaking, σ' additionally takes into account the free variables and context variables that occur in e_1 or e_2, but that have been reduced away and are not present in the resulting normal forms. The values and contexts σ' assigns to such variables are irrelevant.

3.6 Comparison with Multi-Prompted Delimited Continuations

Algebraic effects and handlers studied in the untyped setting, as in this work, diverge from their categorical origins [22], and can be considered a new form of delimited control [10, 11]. As a matter of fact, there exist mutual encodings of algebraic effects and (deep) handlers over a single operation and the control operator shift0 [28], both in an untyped [12] and polymorphically typed settings [21]. These encodings are not fully abstract and therefore they do not guarantee that a behavioral theory, such as the one presented in this work, would carry over to the corresponding calculus of delimited continuations. Given that we allow for multi-labeled algebraic operations, the corresponding calculus in our case would be a generalization of shift0 to its multi-prompted version $\mathsf{shift0}_l$ where the main reduction rule is:

$$\mathsf{prompt}_l E^{\bar{l}}[\mathsf{shift0}_l k.e] \quad \mapsto \quad e\{\lambda z.\mathsf{prompt}_l E^{\bar{l}}[z]/k\}$$

We can observe that in contrast to the calculus of algebraic effects, the party responsible for handling the effect is the same as the one that actually does the effect – it is not the prompt that handles it, but the expression e. The reversal of the roles makes algebraic effects considerably more programmer-friendly, but it also simplifies the theory, compared to the one for classical delimited-control operators. In particular, the techniques we propose in this work appear not to be sufficient for constructing a normal-form bisimulation theory for multi-prompted shift0.

The main obstacle is encountered when we relate evaluation contexts, say E_1 and E_2. The requirement that $E_1[z]$ and $E_2[z]$ (for a fresh z) be related is uncontroversial. However, how should we test E_1 and E_2 for control effects? We need a notion of an abstract control-stuck term and we do not know how to represent it in this calculus. We could introduce a syntactic category of control-stuck-term variables for this purpose, but this would lead nowhere – plugging E_1 and E_2 with such a variable would immediately result in control-stuck terms – there simply is no code that could test the contexts.

One could try to decompose the contexts E_1 and E_2 into some corresponding sub-contexts and relate those, following the approach that works for single-prompted control operators shift and reset for which there exists a sound normal-form bisimilarity [4]. Whether this could lead to a complete theory is not clear and requires further study. As for single-prompted control operators, be it shift or shift0, reaching completeness seems a tall order – notice that the completeness proof of Section 3.5 hinges on the existence of fresh effect labels (prompts).

4 Related Work

Up to now, most works studying the behavioral theory of a calculus with generic algebraic effects were not considering handlers, but interpretations of effects instead, usually in a monad. In such a setting, the behavior of an effect is therefore given for all programs once and for all by the interpretation. In contrast, with handlers, the behavior of an effect may change between programs or during the execution of a program as it depends on how it is handled. The calculus we consider is therefore more expressive than those of the works we list below, with a more discriminative contextual equivalence. It explains why we can reach completeness with a syntactic equivalence such as normal-form bisimilarity while previous works do not achieve completeness with more elaborate equivalences such as applicative bisimilarity. As a matter of fact, the completeness proof presented in this paper relies on an encoding of state and resembles the completeness proof we developed for higher-order state in a previous work [5]. The definition of the normal-form bisimilarity for state, unlike the one presented in this work, did not require any extensions of the calculus. However, its structure is considerably more involved since in the absence of control operators, to reach completeness, we had to explicitly handle deferred diverging terms and impose a stack-like discipline on the way evaluation contexts are tested.

Some recent works interpret effects in a monad and use *relators* which express how interpreted terms should be compared in the monad. Relators allow to develop the behavioral theory of a calculus with effects in a very abstract setting: e.g., one can get for free that the bisimilarity is a congruence provided that a relator exists for the interpretation monad. Relators have been studied for applicative bisimilarity in call-by-value [15] or call-by-name [16], and for normal-form bisimilarity in call-by-value [14]. As pointed out by the authors in [16], *"there is however little hope to prove a generic full-abstraction result [w.r.t. contextual equivalence] in such a setting, although for certain notions of an effect, full abstraction is already known to hold."* However, completeness can be obtained in some cases, as in an untyped call-by-name calculus with deterministic effects [16].

The other path to completeness in typed languages is through logic or logical relations. Johann et al. [13] propose a contextual equivalence and a logical relation characterizing it in a call-by-name calculus with effects. Their framework deals with different effects in a uniform way but with some limitations, as for instance nondeterminism, local store, or the combination of effects cannot be accounted for. Simpson and Voorneveld [29] present a modal logic for a call-by-value calculus which coincides with Dal Lago et al.'s applicative bisimilarity [15], but not with contextual equivalence, as demonstrated later [19]. Matache and Staton improve on these results by defining a logic for a calculus in continuation-passing style that coincides with both applicative bisimilarity and contextual equivalence [19]. Finally, Biernacki et al. [8] define a step-indexed logical relation for a call-by-value calculus with effects and handlers; to the best of our knowledge, it is the only previous work with handlers.

5 Conclusion

We present a sound and complete normal-form bisimilarity for a calculus with effects and handlers. The crucial point is to accurately observe how evaluation contexts may handle effects. First, we distinguish between resumptions, which are plugged only with values, from regular contexts, which may be plugged with any expressions, including effectful ones. We then test the latter contexts using control-stuck terms where the continuation is represented by a context variable, which allows to track how the captured continuation is handled. Extending the calculus with context variables introduces new normal forms which are compared by the bisimilarity in a very simple and regular way. The fact that such a simple notion of normal-form bisimilarity is complete shows the discriminating power of handlers. A consequence is that the examples of equivalent programs we provide are quite simple, as more complex effectful expressions are easily distinguished by handlers.

There are several directions for future work. As pointed out in Section 3.6, it remains an open question how to define complete normal-form bisimulations in the calculus of multi-prompted delimited-control operators corresponding to deep handlers studied in this work. Then, it would be worthwhile to investigate whether the results presented in this paper carry over to shallow handlers. Finally, there exist a number of type-and-effect systems for algebraic effects of varying complexity [8, 9, 21], and one can wonder how features such as effect polymorphism along with effect coercions would influence the theory of this paper.

References

1 Samson Abramsky. The lazy lambda calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, The University of Texas Year of Programming Series, chapter 4, pages 65–116. Addison-Wesley, 1990.

2 Andrés Aristizábal, Dariusz Biernacki, Sergueï Lenglet, and Piotr Polesiuk. Environmental Bisimulations for Delimited-Control Operators with Dynamic Prompt Generation. *Logical Methods in Computer Science*, 13(3), 2017.

3 Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.*, 84(1):108–123, 2015. doi:10.1016/j.jlamp.2014.02.001.

4 Dariusz Biernacki, Sergueï Lenglet, and Piotr Polesiuk. Bisimulations for delimited-control operators. *Logical Methods in Computer Science*, 15(2), 2019.

5 Dariusz Biernacki, Sergueï Lenglet, and Piotr Polesiuk. A complete normal-form bisimilarity for state. In Mikołaj Bojańczyk and Alex Simpson, editors, *Foundations of Software Science and Computation Structures - 22nd International Conference, FOSSACS 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 11425 of *Lecture Notes in Computer Science*, pages 98–114. Springer, 2019. doi:10.1007/978-3-030-17127-8_6.

6 Dariusz Biernacki, Sergueï Lenglet, and Piotr Polesiuk. Diacritical companions. In Barbara König, editor, *Proceedings of the Thirty-Fifth Conference on the Mathematical Foundations of Programming Semantics*, volume 347 of *Electronic Notes in Theoretical Computer Science*, pages 25–43, London, England, 2019.

7 Dariusz Biernacki, Sergueï Lenglet, and Piotr Polesiuk. Proving soundness of extensional normal-form bisimilarities. *Logical Methods in Computer Science*, 15(1), 2019.

8 Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Handle with care: relational interpretation of algebraic effects and handlers. *PACMPL*, 2(POPL):8:1–8:30, 2018.

9 Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Abstracting algebraic effects. *PACMPL*, 3(POPL):6:1–6:28, 2019. doi:10.1145/3290319.

10 Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

11 Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and Peter Mager, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, January 1988. ACM Press.

12 Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control. *PACMPL*, 1(ICFP):13:1–13:29, 2017.

13 Patricia Johann, Alex Simpson, and Janis Voigtländer. A generic operational metatheory for algebraic effects. In Jean-Pierre Jouannaud, editor, *Proceedings of the 25th IEEE Symposium on Logic in Computer Science (LICS 2010)*, pages 209–218, Edinburgh, UK, July 2010. IEEE Computer Society Press.

14 Ugo Dal Lago and Francesco Gavazzo. Effectful normal form bisimulation. 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, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 263–292, Prague, Czech Republic, April 2019. Springer.

15 Ugo Dal Lago, Francesco Gavazzo, and Paul Blain Levy. Effectful applicative bisimilarity: Monads, relators, and Howe's method. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017*, pages 1–12, Reykjavik, Iceland, June 2017. IEEE Computer Society.

16 Ugo Dal Lago, Francesco Gavazzo, and Ryo Tanaka. Effectful applicative similarity for call-by-name lambda calculi. In Dario Della Monica, Aniello Murano, Sasha Rubin, and Luigi Sauro, editors, *Joint Proceedings of the 18th Italian Conference on Theoretical Computer*

Science and the 32nd Italian Conference on Computational Logic co-located with the 2017 IEEE International Workshop on Measurements and Networking (2017 IEEE M&N), volume 1949 of *CEUR Workshop Proceedings*, pages 87–98, Naples, Italy, September 2017. CEUR-WS.org.

17 Søren B. Lassen. Eager normal form bisimulation. In Prakash Panangaden, editor, *Proceedings of the Twentieth Annual IEEE Symposium on Logic in Computer Science*, pages 345–354, Chicago, IL, June 2005. IEEE Computer Society Press.

18 Jean-Marie Madiot. *Higher-order languages: dualities and bisimulation enhancements.* PhD thesis, Université de Lyon and Università di Bologna, 2015.

19 Cristina Matache and Sam Staton. A sound and complete logic for algebraic effects. In Mikolaj Bojańczyk and Alex Simpson, editors, *Foundations of Software Science and Computation Structures - 22nd International Conference, FOSSACS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Proceedings*, volume 11425 of *Lecture Notes in Computer Science*, pages 382–399, Prague, Czech Republic, April 2019. Springer.

20 James H. Morris. *Lambda Calculus Models of Programming Languages.* PhD thesis, Massachusets Institute of Technology, 1968.

21 Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Typed equivalence of effect handlers and delimited control. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019*, volume 131 of *LIPIcs*, pages 30:1–30:16, Dortmund, Germany, June 2019. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

22 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`.

23 Damien Pous. Coinduction all the way up. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16*, pages 307–316, New York, NY, USA, July 2016. ACM.

24 John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Paris, France, 1983. IFIP.

25 Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. In Andre Scedrov, editor, *LICS'92*, pages 102–109, Santa Cruz, California, June 1992. IEEE Computer Society.

26 Davide Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8(5):447–479, October 1998.

27 Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. *ACM Transactions on Programming Languages and Systems*, 33(1):1–69, January 2011.

28 Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 20(4):371–401, 2007.

29 Alex Simpson and Niels F. W. Voorneveld. Behavioural equivalence via modalities for algebraic effects. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 300–326, Thessaloniki, Greece, April 2018. Springer.

A Soundness Proof Sketch

We only discuss the case of csubst and rsubst, as the others are proved as in the plain λ-calculus [7]. In particular, we use the fact that

▶ **Lemma 21.** $\mathsf{subst} \sqsubseteq \mathsf{u}$

We want to prove that csubst and rsubst are weak, but to circumvent the constraint that they cannot be composed twice in a passive clause, we combine csubst and rsubst in a single ssubst doing simultaneous substitutions.

$$\frac{e_1 \mathrel{\mathcal{R}} e_2 \qquad \sigma_1 \mathrel{\mathcal{R}^\sigma} \sigma_2}{e_1\sigma_1 \mathrel{\mathsf{ssubst}(\mathcal{R})} e_2\sigma_2}$$

We let σ ranges over simultaneous substitution, meaning that if $\sigma(\alpha^{\overline{l}}) = E^{\overline{l}}$, then $(\alpha^{\overline{l}}[e])\sigma \triangleq E^{\overline{l}}[e\sigma]$; we do not apply σ to $E^{\overline{l}}$. We define $\mathcal{R}^\sigma$ pairwise such that we have either $\sigma_1(\alpha^{\overline{l}}) \mathrel{\mathcal{R}^{\mathsf{c}}} \sigma_2(\alpha^{\overline{l}})$, or $\sigma_1(\alpha^{\overline{l}}) \mathrel{\mathcal{R}^{\mathsf{r}}} \sigma_2(\alpha^{\overline{l}})$ with $\mathsf{resum}(\alpha^{\overline{l}}, e_1)$ and $\mathsf{resum}(\alpha^{\overline{l}}, e_2)$.

▶ **Lemma 22.** $\mathsf{ssubst} \sqsubseteq \mathsf{w}$

Proof. Let $\mathcal{R} \rightarrowtail \mathcal{R}, \mathcal{S}$, $e_1\sigma_1 \mathrel{\mathsf{subst}(\mathcal{R})} e_2\sigma_2$ with $e_1 \mathrel{\mathcal{R}} e_2$ and $\sigma_1 \mathrel{\mathcal{R}^{\mathsf{c}}} \sigma_2$. We proceed by case analysis on the behavior of e_1. The cases where e_1 reduces, is a value, or is an open-stuck term are simple.

Suppose $e_1 = E_1'^{\overline{l}}[\mathsf{do}_l\, v_1]$, then there exist $E_2'^{\overline{l}}$ and v_2 such that $e_2 \rightarrow^* E_2'^{\overline{l}}[\mathsf{do}_l\, v_2]$, $E_1'^{\overline{l}} \mathrel{\mathcal{S}^{\mathsf{r}}} E_2'^{\overline{l}}$ and $v_1 \mathrel{\mathcal{S}^{\mathsf{v}}} v_2$. Any context variable surrounding the hole of $E_1'^{\overline{l}}$ can only be of the form $\alpha_i^{\overline{l}}$, meaning that $E_1'^{\overline{l}}\sigma_1$ still does not handle l, and the resulting terms are control-stuck. We progress to ssubst, so we can conclude.

Suppose $e_1 = E_1[\alpha^{\overline{l}}[v_1]]$ with $\alpha^{\overline{l}} \in \mathsf{dom}(\sigma_1)$ (the case where the variable is not in the domain is easily handled). There exist E_2 and v_2 such that $e_2 \rightarrow^* E_2[\alpha^{\overline{l}}[v_2]]$, $E_1 \mathrel{\mathcal{R}^{\mathsf{c}}} E_2$, and $v_1 \mathrel{\mathcal{R}^{\mathsf{v}}} v_2$. From $\sigma_1(\alpha^{\overline{l}}) \mathrel{\mathcal{R}^{\mathsf{c}}} \sigma_2(\alpha^{\overline{l}})$, we get in particular $\sigma_1(\alpha^{\overline{l}})[x] \mathrel{\mathcal{R}} \sigma_2(\alpha^{\overline{l}})[x]$ for a fresh x, therefore $\sigma_1(\alpha^{\overline{l}})[v_1] \mathrel{\mathsf{subst}(\mathcal{R})} \sigma_2(\alpha^{\overline{l}})[v_2]$. We have two special cases to consider, $\sigma_1(\alpha^{\overline{l}}) = \beta^{\overline{l}}[\square]$ and $\sigma_1(\alpha^{\overline{l}}) = \square$; in the other cases, $\sigma_1(\alpha^{\overline{l}})[v_1]$ is doing something active and we can conclude using Lemma 21.

If $\sigma_1(\alpha^{\overline{l}}) = \square$, we have $x \mathrel{\mathcal{R}} \sigma_2(\alpha^{\overline{l}})[x]$, from which we deduce that there exist w such that $\sigma_2(\alpha^{\overline{l}})[x] \rightarrow^* w$ and $x \mathrel{\mathcal{R}^{\mathsf{v}}} w$. As a result, $e_1\sigma_1 = E_1'\{_/\sigma\}1[v_1\sigma_1]$, and $e_2\sigma_2 \rightarrow^* E_2'\sigma_2[w\{v_2/x\}\sigma_2]$. Since we have $E_1'[v_1] \mathrel{\mathsf{subst}(\mathsf{subst}(\mathcal{R}))} E_2'[w\{v_2/x\}]$, we can conclude again with Lemma 21.

If $\sigma_1(\alpha^{\overline{l}}) = \beta^{\overline{l}}[\square]$, then from $\beta^{\overline{l}}[x] \mathrel{\mathcal{R}} \sigma_2(\alpha^{\overline{l}})[x]$, there exist E_2' and w such that $\sigma_2(\alpha^{\overline{l}})[x] \rightarrow^* E_2'[\beta^{\overline{l}}[w]]$, $\square \mathrel{\mathcal{R}^{\mathsf{c}}} E_2'$, and $x \mathrel{\mathcal{R}^{\mathsf{v}}} w$. Therefore we have $e_2\sigma_2 \rightarrow^* E_2\sigma_2[\sigma_2(\alpha^{\overline{l}})[v_2\sigma_2]] \rightarrow^* E_2\sigma_2[E_2'[\beta^{\overline{l}}[w\{v_2\sigma_2/x\}]]]$, yielding a context-stuck term that is to be related to $E_1\sigma_1[\beta^{\overline{l}}[v_1\sigma_1]]$. We are fine w.r.t. the values, as we have $v_1\sigma_1 \mathrel{\mathsf{ssubst}(\mathsf{subst}(\mathcal{R}))} w\{v_2\sigma_2/x\}$. For the contexts, we first relate $E_1\sigma_1[y]$ and $E_2\sigma_2[E_2'[y]]$ for a fresh y. Because $\square \mathrel{\mathcal{R}^{\mathsf{c}}} E_2'$, there exists w' such that $E_2'[y] \rightarrow^* w'$ and $y \mathrel{\mathcal{R}^{\mathsf{v}}} w'$. As a result, we have $E_2\sigma_2[E_2'[y]] \rightarrow^* E_2\sigma_2[w']$, and therefore $E_1\sigma_1[y] \mathrel{\mathsf{red}(\mathsf{ssubst}(\mathsf{subst}(\mathcal{R})))} E_2\sigma_2[E_2'[y]]$, which is what we need. Then we must relate $E_1\sigma_1[\gamma^{\overline{l'}}[\mathsf{do}_{l'}\, y]]$ and $E_2\sigma_2[E_2'[\gamma^{\overline{l'}}[\mathsf{do}_{l'}\, y]]]$ for any l' and fresh $\gamma^{\overline{l'}}$ and y. Because $\square \mathrel{\mathcal{R}^{\mathsf{c}}} E_2'$, there exist $E_2''^{\overline{l'}}$ and w' such that $E_2'[\gamma^{\overline{l'}}[\mathsf{do}_{l'}\, y]] \rightarrow^* E_2''^{\overline{l'}}[\mathsf{do}_{l'}\, w']$, $\gamma^{\overline{l'}}[\square] \mathrel{\mathcal{S}^{\mathsf{r}}} E_2''^{\overline{l'}}$, and $y \mathrel{\mathcal{S}^{\mathsf{v}}} w'$. From $E_1 \mathrel{\mathcal{R}^{\mathsf{c}}} E_2$, we get $E_1[\gamma^{\overline{l'}}[\mathsf{do}_{l'}\, y]] \mathrel{\mathcal{R}} E_2[\gamma^{\overline{l'}}[\mathsf{do}_{l'}\, y]]$, so if $E_1[\gamma^{\overline{l'}}[\mathsf{do}_{l'}\, y]] \rightarrow e_1'$ for some e_1' (the case where l' is not handled is not interesting), then there exists e_2' such that $E_2[\gamma^{\overline{l'}}[\mathsf{do}_{l'}\, y]] \rightarrow^* e_2'$ and $e_1' \mathrel{\mathcal{S}} e_2'$. Therefore, $E_2\sigma_2[E_2'[\gamma^{\overline{l'}}[\mathsf{do}_{l'}\, y]]] \rightarrow^* E_2\sigma_2[E_2''^{\overline{l'}}[\mathsf{do}_{l'}\, w']] \rightarrow^* e_2'\sigma_2'\{w'/y\}$ where $\sigma_2'(\gamma^{\overline{l'}}) = E_2''^{\overline{l'}}$ and is equal to σ_2 otherwise. Because $E_1\sigma_1[\gamma^{\overline{l'}}[\mathsf{do}_{l'}\, y]] \rightarrow e_1\sigma_1 = e_1\sigma_1'\{y/y\}$ where $\sigma_1'(\gamma^{\overline{l'}}) = \gamma^{\overline{l'}}[\square]$ and is equal to σ_1 otherwise, we deduce $E_1\sigma_1[\gamma^{\overline{l'}}[\mathsf{do}_{l'}\, y]] \mathrel{\mathsf{red}(\mathsf{subst}(\mathsf{ssubst}(\mathcal{S})))} E_2\sigma_2[E_2'[\gamma^{\overline{l'}}[\mathsf{do}_{l'}\, y]]]$ which is enough to conclude.

Suppose $e_1 = E_1[\alpha^{\overline{l}}[E_1'^{\overline{l'}}[\mathsf{do}_{l'}\, v_1]]]$ with $l \neq l'$ and $\alpha^{\overline{l}} \in \mathsf{dom}(\sigma_1)$; then there exist E_2, $E_2'^{\overline{l'}}$, and v_2 such that $e_2 \rightarrow^* E_2[\alpha^{\overline{l}}[E_2'^{\overline{l'}}[\mathsf{do}_{l'}\, v_2]]]$, $E_1 \mathrel{\mathcal{T}^{\mathsf{c}}} E_2$, $E_1'^{\overline{l'}} \mathrel{\mathcal{T}^{\mathsf{r}}} E_2'^{\overline{l'}}$, and $v_1 \mathrel{\mathcal{T}^{\mathsf{v}}} v_2$. From $\sigma_1(\alpha^{\overline{l}}) \mathrel{\mathcal{R}^{\mathsf{c}}} \sigma_2(\alpha^{\overline{l}})$, we get $\sigma_1(\alpha^{\overline{l}})[\gamma^{\overline{l'}}[\mathsf{do}_{l'}\, x]] \mathrel{\mathcal{R}} \sigma_2(\alpha^{\overline{l}})[\gamma^{\overline{l'}}[\mathsf{do}_{l'}\, x]]$ for fresh $\gamma^{\overline{l'}}$ and x. If

$\sigma_1(\alpha^{\overline{l}})[\gamma^{\overline{l'}}[\mathsf{do}_{l'}\ x]] \rightarrow e_1'$ for some e_1', then there exists e_2' such that $\sigma_2(\alpha^{\overline{l}})[\gamma^{\overline{l'}}[\mathsf{do}_{l'}\ x]] \rightarrow^* e_2'$ and $e_1' \ \mathcal{S}\ e_2'$. Then $e_1\sigma_1 \rightarrow E_1[e_1'\{v_1/x\}\{E_1'^{\overline{l'}}/\gamma^{\overline{l'}}\}]\sigma_1$ and $e_2\sigma_2 \rightarrow^* E_2[e_2'\{v_2/x\}\{E_2'^{\overline{l'}}/\gamma^{\overline{l'}}\}]\sigma_2$, and the resulting expressions are in $\mathsf{ssubst}(\mathsf{ssubst}(\mathsf{cvar}(\mathsf{ssubst}(\mathsf{subst}(\mathcal{S})))))$, which is fine, because we are in an active clause. ◀

B Completeness Proof Sketch

The proof proceeds as described in Section 3.5: given $e_1 \equiv_{\mathrm{E}} e_2$, we check that for each behavior of e_1, e_2 is able to match. If e_1 is a normal form, we verify that (1) e_2 evaluates to a normal form of the same kind, and the normal forms can be decomposed into related sub-parts. For each case, we give the substitution σ and the context E enforcing (1). Checking that related sub-parts are contextually equivalent relies in most cases on an encoding of a mutable state using handlers, as in Section 3.5. In all the subcases below, we assume the labels get and put to be fresh, and given a boolean b and a context E'', we define

$$E_b' = (\mathsf{handle}\ E''\ \{\mathsf{get}\colon z{,}k \rightarrow \lambda y.k\ y\ y;\ \mathsf{put}\colon z{,}k \rightarrow \lambda y.k\ ()\ z;\ \mathsf{ret}\ z \rightarrow \lambda y.z\})\ b$$

We define E'' in each subcase where the encoding is needed.

Case: $e_1 \rightarrow e_1'$. Because the reduction is deterministic, we still have $e_1' \equiv_{\mathrm{E}} e_2$.

Case: $e_1 = v_1$. To check (1), take σ as follows:

$$\begin{array}{rcll} \sigma(x) & = & \lambda y.\Omega & \text{for } x \in \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\ \sigma(\alpha^{\overline{l}}) & = & \mathsf{handle}\ \square\ \{H_l;\ \mathsf{ret}\ x \rightarrow \Omega\} & \text{for } \alpha^{\overline{l}} \in \mathsf{cv}(e_1) \cup \mathsf{cv}(e_2) \end{array}$$

where $H_l = l_1\colon x{,}k \rightarrow \Omega; \ldots; l_n\colon x{,}k \rightarrow \Omega$ with $\{l_1, \ldots, l_n\} = \mathsf{lbl}(e_1) \cup \mathsf{lbl}(e_2) \setminus \{l\}$, and $E = \square$. Hence, there exists v_2 such that $e_2 \rightarrow^* v_2$; we check that $v_1 \equiv_{\mathrm{E}}^{\mathsf{v}} v_2$.

Let x be a fresh variable, E a context, and σ a closing substitution such that $E[v_1\ x]\sigma \Downarrow_{\mathrm{v}}$. Then $E[e_2\ x]\sigma \rightarrow^* E[v_2\ x]\sigma$ and since $e_1 \equiv_{\mathrm{E}} e_2$, we also have $E[v_2\ x]\sigma \Downarrow_{\mathrm{v}}$.

Case: $e_1 = E_1[x\ v_1]$. To check (1), take σ as follows:

$$\begin{array}{rcll} \sigma(z) & = & \lambda y.\Omega & \text{for } z \in \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \setminus \{x\} \\ \sigma(x) & = & \lambda y.\mathsf{do}_{l'}\ \lambda z.z & \\ \sigma(\alpha^{\overline{l}}) & = & \mathsf{handle}\ \square\ \{H_l;\ \mathsf{ret}\ x \rightarrow \Omega\} & \text{for } \alpha^{\overline{l}} \in \mathsf{cv}(e_1) \cup \mathsf{cv}(e_2) \end{array}$$

where $l' \notin \mathsf{lbl}(e_1) \cup \mathsf{lbl}(e_2)$, $H_l = l_1\colon x{,}k \rightarrow \Omega; \ldots; l_n\colon x{,}k \rightarrow \Omega$ with $\{l_1, \ldots, l_n\} = \mathsf{lbl}(e_1) \cup \mathsf{lbl}(e_2) \setminus \{l\}$, and $E = \mathsf{handle}\ \square\ \{l'\colon y{,}k \rightarrow y;\ \mathsf{ret}\ x \rightarrow \Omega\}$. Hence, there exists $E_2[x\ v_2]$ such that $e_2 \rightarrow^* E_2[x\ v_2]$; we check that (2) $v_1 \equiv_{\mathrm{E}}^{\mathsf{v}} v_2$ and (3) $E_1 \equiv_{\mathrm{E}}^{\mathsf{c}} E_2$.

For (2), let y be a fresh variable and consider the *testing arguments* E and σ such that σ is a closing substitution and $E[v_1\ y]\sigma \Downarrow_{\mathrm{v}}$. Let l' be a fresh label, and define σ' to be equal to σ everywhere, except for x:

$$\sigma'(x) = \lambda z.\mathsf{if}\ \mathsf{do}_{\mathsf{get}}\ ()\ \mathsf{then}\ (\mathsf{do}_{\mathsf{put}}\ \mathsf{false};\ \mathsf{do}_{l'}\ z)\ \mathsf{else}\ \sigma(x)$$

and consider

$$E'' = \mathsf{handle}\ \square\ \{l'\colon z{,}k \rightarrow E[z\ y];\ \mathsf{ret}\ z \rightarrow z\}.$$

Then E_{true}' and σ' are the *discriminating arguments*, i.e., $E[v_1\ y]\sigma \Downarrow_{\mathrm{v}}$ iff $E_{\mathsf{true}}'[e_1]\sigma' \Downarrow_{\mathrm{v}}$ iff $E_{\mathsf{true}}'[e_2]\sigma' \Downarrow_{\mathrm{v}}$ iff $E[v_2\ y]\sigma \Downarrow_{\mathrm{v}}$.

Proving (3) requires (a) $E_1[y] \equiv_{\mathrm{E}} E_2[y]$ for a fresh y, and (b) $E_1[\alpha^{\overline{l''}}[\mathsf{do}_{l''}\, y]] \equiv_{\mathrm{E}} E_2[\alpha^{\overline{l}}[\mathsf{do}_l\, y]]$ for any l and fresh $\alpha^{\overline{l}}$ and y. Assuming the same testing arguments E and σ, both cases are proved as in (2), except that in (a) we take

$$E'' = \mathsf{handle}\,\square\,\{l'\!: z,k \rightarrow E[k\;y]; \mathsf{ret}\; z \rightarrow z\}$$

and in (b) we take

$$E'' = \mathsf{handle}\,\square\,\{l'\!: z,k \rightarrow E[k\;(\alpha^{\overline{l}}[\mathsf{do}_l\, y])]; \mathsf{ret}\; z \rightarrow z\}.$$

Case: $\boldsymbol{e_1 = E_1{}^{\overline{l}}[\mathsf{do}_l\, v_1]}$. To check (1), take σ as follows:

$$\begin{array}{rcll} \sigma(x) & = & \lambda y.\Omega & \text{for } x \in \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\ \sigma(\alpha^{\overline{l'}}) & = & \mathsf{handle}\,\square\,\{H_{l'}; \mathsf{ret}\; x \rightarrow \Omega\} & \text{for } \alpha^{\overline{l'}} \in \mathsf{cv}(e_1) \cup \mathsf{cv}(e_2) \end{array}$$

where $H_{l'} = l_1\!: x,k \rightarrow \Omega; \ldots; l_n\!: x,k \rightarrow \Omega$ with $\{l_1, \ldots, l_n\} = \mathsf{lbl}(e_1) \cup \mathsf{lbl}(e_2) \setminus \{l'\}$, and $E = \mathsf{handle}\,\square\,\{l\!: y,k \rightarrow y; \mathsf{ret}\; x \rightarrow \Omega\}$. Hence, there exists $E_2{}^{\overline{l}}[\mathsf{do}_l\, v_2]$ such that $e_2 \rightarrow^* E_2{}^{\overline{l}}[\mathsf{do}_l\, v_2]$; we check that (2) $v_1 \equiv^{\mathsf{v}}_{\mathrm{E}} v_2$ and (3) $E_1{}^{\overline{l}} \equiv^{\mathsf{r}}_{\mathrm{E}} E_2{}^{\overline{l}}$.

Assuming we use a fresh variable x and E, σ as testing arguments, we conclude in the former case by considering $E' = \mathsf{handle}\,\square\,\{l\!: z,k \rightarrow E[z\;x]; \mathsf{ret}\; z \rightarrow z\}$ and σ as discriminating arguments.

We prove (3) assuming x fresh and E, σ as testing arguments. Let l', l'' be fresh labels; we define

$$E'' = \mathsf{handle}\; E'''\;\{l'\!: z,k \rightarrow E_{l''}[z\;x]; \mathsf{ret}\; z \rightarrow z\}.$$

where

$$E''' = \mathsf{handle}\,\square\,\{l\!: z,k \rightarrow \mathsf{if}\;\mathsf{do}_{\mathsf{get}}\;()\;\mathsf{then}\;(\mathsf{do}_{\mathsf{put}}\;\mathsf{false};\;\mathsf{do}_{l'}\;k)\;\mathsf{else}\;k\;(\mathsf{do}_{l''}\;z); \mathsf{ret}\; z \rightarrow z\}.$$

and $E_{l''}$ is E where all the occurrences of l are replaced by l''. When l is handled first, we create the discriminating term; subsequent handlings are perfomed by E through l''. Renaming l into a fresh l'' in E is necessary to bypass the handler for l in E'''. The discriminating arguments are E'_{true} and σ.

Case: $\boldsymbol{e_1 = E_1[\alpha^{\overline{l}}[v_1]]}$. Described in details in Section 3.5.

Case: $\boldsymbol{e_1 = E_1[\alpha^{\overline{l'}}[E_1'{}^{\overline{l}}[\mathsf{do}_l\, v_1]]]}$. To check (1), take σ as follows:

$$\begin{array}{rcll} \sigma(x) & = & \lambda y.\Omega & \text{for } x \in \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\ \sigma(\beta^{\overline{l''}}) & = & \mathsf{handle}\,\square\,\{H_{l''}; \mathsf{ret}\; x \rightarrow \Omega\} & \text{for } \beta^{\overline{l''}} \in \mathsf{cv}(e_1) \cup \mathsf{cv}(e_2) \text{ and } \beta^{\overline{l''}} \neq \alpha^{\overline{l'}} \\ \sigma(\alpha^{\overline{l'}}) & = & \mathsf{handle}\,\square\,\{l\!: x,k \rightarrow \mathsf{do}_{l'''}\; x; \mathsf{ret}\; x \rightarrow \Omega\} & \end{array}$$

where $l''' \notin \mathsf{lbl}(e_1) \cup \mathsf{lbl}(e_2)$, $H_{l''} = l_1\!: x,k \rightarrow \Omega; \ldots; l_n\!: x,k \rightarrow \Omega$ with $\{l_1, \ldots, l_n\} = \mathsf{lbl}(e_1) \cup \mathsf{lbl}(e_2) \backslash \{l''\}$, and $E = \mathsf{handle}\,\square\,\{l'''\!: x,k \rightarrow x; \mathsf{ret}\; x \rightarrow \Omega\}$. Hence, there exists $E_2[\alpha^{\overline{l'}}[E_2'{}^{\overline{l}}[\mathsf{do}_l\, v_2]]]$ such that $e_2 \rightarrow^* E_2[\alpha^{\overline{l'}}[E_2'{}^{\overline{l}}[\mathsf{do}_l\, v_2]]]$; we check that (2) $v_1 \equiv^{\mathsf{v}}_{\mathrm{E}} v_2$, (3) $E_1'{}^{\overline{l}} \equiv^{\mathsf{r}}_{\mathrm{E}} E_2'{}^{\overline{l}}$, and (4) $E_1 \equiv^{\mathsf{c}}_{\mathrm{E}} E_2$. In each case, we assume x and l'' to be fresh and the testing arguments to be E and σ.

The discriminating arguments for (2) are σ', defined to be equal to σ everywhere, except for $\alpha^{\overline{l'}}$:

$$\sigma'(\alpha^{\overline{l'}}) = \sigma(\alpha^{\overline{l'}})[\mathsf{handle}\,\square\,\{l\colon z,k \to \mathsf{if}\ \mathsf{do}_{\mathsf{get}}\,()\ \mathsf{then}\ (\mathsf{do}_{\mathsf{put}}\,\mathsf{false};\, \mathsf{do}_{l''}\,z)\ \mathsf{else}\ k\,(\mathsf{do}_{l}\,x);\, \mathsf{ret}\ z \to z\}],$$

and E'_{true} assuming

$$E'' = \mathsf{handle}\,\square\,\{l''\colon z,k \to E[z\,x];\, \mathsf{ret}\ z \to z\}.$$

For (3), we prove $E_1'^{\overline{l}}[x] \equiv_{\mathrm{E}} E_2'^{\overline{l}}[x]$ as in (2), except that we take an extra fresh l''' and define

$$\sigma'(\alpha^{\overline{l'}}) = \sigma(\alpha^{\overline{l'}})[\mathsf{handle}\,\square\,\{l\colon z,k \to \mathsf{if}\ \mathsf{do}_{\mathsf{get}}\,()\ \mathsf{then}\ (\mathsf{do}_{\mathsf{put}}\,\mathsf{false};\, \mathsf{do}_{l''}\,k)\ \mathsf{else}\ k\,(\mathsf{do}_{l'''}\,x);\, \mathsf{ret}\ z \to z\}]$$

and

$$E'' = \mathsf{handle}\,\square\,\{l''\colon z,k \to E_{l'''}[z\,x];\, \mathsf{ret}\ z \to z\}$$

where $E_{l'''}$ is the context E where the occurrences of l are replaced with l'''.

Proving (4) requires (a) $E_1[x] \equiv_{\mathrm{E}} E_2[x]$ and (b) $E_1[\alpha^{\overline{l'''}}[\mathsf{do}_{l'''}\,z]] \equiv_{\mathrm{E}} E_2[\alpha^{\overline{l'''}}[\mathsf{do}_{l'''}\,x]]$ for any l''' and fresh $\alpha^{\overline{l'''}}$. Assuming the same testing arguments, both cases are proved as in (2), except that in (a) we take

$$E'' = \mathsf{handle}\,\square\,\{l''\colon z,k \to E[k\,x];\, \mathsf{ret}\ z \to z\}$$

and in (b) we take

$$E'' = \mathsf{handle}\,\square\,\{l''\colon z,k \to E[k\,(\alpha^{\overline{l'''}}[\mathsf{do}_{l'''}\,x])];\, \mathsf{ret}\ z \to z\}.$$

Pomsets with Boxes: Protection, Separation, and Locality in Concurrent Kleene Algebra

Paul Brunet
University College London, UK
paul.brunet-zamansky.fr
paul@brunet-zamansky.fr

David Pym
University College London, UK
www.cantab.net/users/david.pym/
d.pym@ucl.ac.uk

Abstract

Concurrent Kleene Algebra is an elegant tool for equational reasoning about concurrent programs. An important feature of concurrent programs that is missing from CKA is the ability to restrict legal interleavings. To remedy this we extend the standard model of CKA, namely pomsets, with a new feature, called boxes, which can specify that part of the system is protected from outside interference. We study the algebraic properties of this new model. Another drawback of CKA is that the language used for expressing properties of programs is the same as that which is used to express programs themselves. This is often too restrictive for practical purposes. We provide a logic, "pomset logic", that is an assertion language for specifying such properties, and which is interpreted on pomsets with boxes. In contrast with other approaches, this logic is not state-based, but rather characterizes the runtime behaviour of a program. We develop the basic metatheory for the relationship between pomset logic and CKA, including frame rules to support local reasoning, and illustrate this relationship with simple examples.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Semantics and reasoning; Theory of computation → Separation logic

Keywords and phrases Concurrent Kleene Algebra, Pomsets, Atomicity, Semantics, Separation, Local reasoning, Bunched logic, Frame rules

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.8

Related Version Since proofs are omitted from this paper, we refer the inquisitive reader to its extended version, available on arXiv: 1910.14384.

Supplementary Material A formalization of Section 2 in Coq is available on github: AtomicCKA.

Funding This work has been supported by UK EPSRC Research Grant EP/R006865/1: Interface Reasoning for Interacting Systems (IRIS).

Acknowledgements The authors are grateful to their colleagues at UCL and within IRIS project for their interest. We also thank the anonymous referees for their comments and suggestions.

1 Introduction

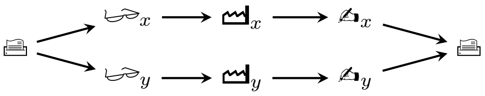
Concurrent Kleene Algebra (CKA) [11, 14, 15, 4] is an elegant tool for equational reasoning about concurrent programs. Its semantics is given in terms of pomsets languages; that is, sets of pomsets. Pomsets [8], also known as partial words [9], are a well-known model of concurrent behaviour, traditionally associated with runs in Petri nets [13, 4].

However, in CKA the language used for expressing properties of programs is the same as that which is used to express programs themselves. It is clear that this situation is not ideal for specifying and reasoning about properties of programs. Any language specifiable

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 8; pp. 8:1–8:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
        print(counter);
x:=counter; || y:=counter;
x:=x+1;     || y:=y+1;
counter:=x; || counter:=y;
        print(counter);
```

(a) Pseudo code.



(b) Graphical representation.

Figure 1 Distributed counter.

in CKA terms has bounded width (i.e., the number of processes in parallel; the size of a maximal independent set) and bounded depth (i.e., the number of alternations of parallel and sequential compositions)[17]. However, many properties of interest – for example, safety properties – are satisfied by sets of pomsets with both unbounded width and depth.

In this paper, we provide a logic, "pomset logic", that is an assertion language for specifying such properties. We develop the basic metatheory for the relationship between pomset logic and CKA and illustrate this relationship with simple examples. In addition, to the usual classical or intuitionistic connectives – both are possible – the logic includes connectives that characterize both sequential and parallel composition.

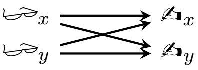
In addition, we note that CKA allows programs with every possible interleaving of parallel threads. However, to prove the correctness of such programs, some restrictions must be imposed on what are the legal interleavings. We provide a mechanism of "boxes" for this purpose. Boxes identify protected parts of the system, so restricting the possible interleavings. From the outside, one may interact with the box as a whole, as if the program inside was atomic. On the other hand, it is not possible to interact with its individual components, as that would intuitively require opening the box. However, boxes can be nested, with this atomicity observation holding at each level. Pomset logic has context and box modalities that characterize this situation.

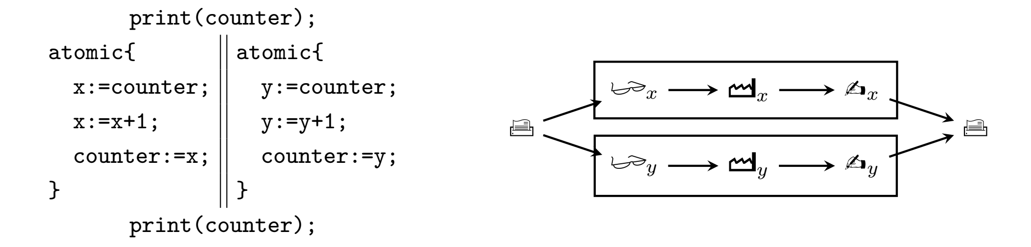
▶ Note. The term "Pomset logic" has already been used in work by Retoré [25]. We feel that reusing it does not introduce ambiguity, since the two frameworks arise in different contexts.

▶ **Example 1** (Running example: a distributed counter)**.** We consider here a program where a counter is incremented in parallel by two processes. The intention is that the counter should be incremented twice, once by each process. However, to do so each process has to first load the contents of the counter, then compute the increment, and finally commit the result to memory. A naive implementation is presented in Figure 1a. Graphically, we represent the print instruction `print(counter)`by 🖶, the read instruction `x:=counter` by 👓$_x$, the increment instruction `x:=x+1` by 🏭$_x$, and finally the write instruction `counter:=x` by ✍$_x$. We thus represent the previous program as displayed in Figure 1b.
This program does not comply with our intended semantics, since the following run is possible:

🖶 ⟶ 👓$_x$ ⟶ 👓$_y$ ⟶ 🏭$_x$ ⟶ 🏭$_y$ ⟶ ✍$_x$ ⟶ ✍$_y$ ⟶ 🖶

The result is that the counter has been incremented by one. We can identify a subset of instructions that indicate there is a fault: the problem is that both read instructions happened before both write instructions; i.e.,





```
          print(counter);
  atomic{          || atomic{
     x:=counter;   ||    y:=counter;
     x:=x+1;       ||    y:=y+1;
     counter:=x;   ||    counter:=y;
  }                || }
          print(counter);
```

(a) Pseudo code. **(b)** Graphical representation.

Figure 2 Distributed counter with atomic increment.

To preclude this problematic behaviour, a simple solution is to make the sequence "read;compute;write" *atomic*. This yields the program in Figure 2a. Diagrammatically, this can be represented by drawing solid boxes around the `atomic{}` blocks, as shown in Figure 2b. This paper shows how to make these ideas formal.

In Section 2, we extend pomsets with a new construct for protection, namely boxes. We provide a syntax for specifying such pomsets and characterize precisely its expressivity. This enables us, for example, to correctly represent the program from Example 1. We present a sound and complete axiomatization of these terms, with operators for boxing, sequential and parallel composition, and non-deterministic choice, as well as the constants *abort* and *skip*.

In Section 3, we introduce pomset logic. This logic comes in both classical and intuitionistic variants. In addition to the usual classical or intuitionistic connectives, this logic includes connectives corresponding to each of sequential and parallel composition. These two classes of connectives are combined to give the overall logics, in the same way as the additives and multiplicatives of BI (bunched implications logic) [21, 1, 23]. Just as in BI and its associated separation logics [21, 12, 26], pomset logic has both classical and intuitionistic variants. It also includes modalities that characterize, respectively, protection, and locality. These correspondences are made precise by van Benthem–Hennessy–Milner-type theorems asserting that two programs are (operationally) equivalent iff they satisfy the same formulae. We obtain such correspondences for several variants of our framework. In contrast to Hennessy–Milner logic, however, pomset logic is a logic of *behaviours* rather than of states and transitions.

In Section 4, we investigate local reasoning principles for our logic of program behaviours. We showcase the possibilities of our framework on an example. We conclude by briefly discussing future work in Section 5.

2 Algebra of Pomsets with Boxes

In this section, we define our semantic model, and the corresponding syntax. We characterize the expressivity of the syntax, and axiomatize its equational theory.

Throughout this paper, we will use Σ to denote a given set of atomic actions.

2.1 Pomsets with boxes

2.1.1 Definitions and elementary properties

▶ **Definition 2** (Poset with boxes). *A* poset with boxes *is a tuple* $P := \langle \mathcal{E}_P, \leq_P, \lambda_P, \mathcal{B}_P \rangle$*, where* $\mathcal{E}_P$ *is a finite set of* events*;* $\leq_P \subseteq \mathcal{E}_P \times \mathcal{E}_P$ *is a partial order;* $\lambda_P : \mathcal{E}_P \to \Sigma$ *is a labelling function;* $\mathcal{B}_P \subseteq \mathcal{P}(\mathcal{E}_P)$ *is a set of* boxes*, such that* $\emptyset \notin \mathcal{B}_P$*.*

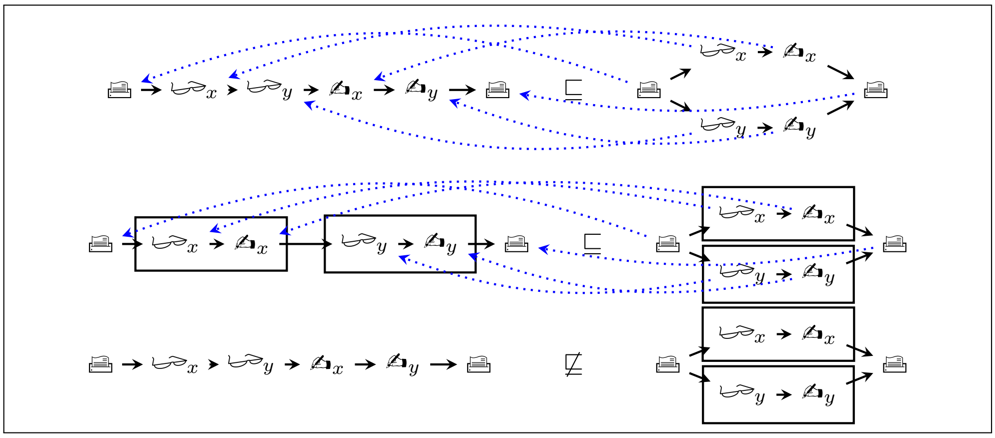


Figure 3 Poset subsumption.

The partial order should be viewed as a set of necessary dependencies: in any legal scheduling of the pomset, these dependencies have to be satisfied. We therefore consider that a stronger ordering – that is, one containing more pairs – yields a smaller pomset. The intuition is that the set of legal schedulings of the smaller pomset is contained in that of the larger one. The boxes are meant to further restrict the legal schedulings: no event from outside a box may be interleaved between the events inside the box. Subsequently, a pomset with more boxes is smaller than one with less boxes. This ordering between pomsets with boxes is formalized by the notion of homomorphism:

▶ **Definition 3** (Poset morphisms). *A* (poset with boxes) homomorphism *is a map between event-sets that is bijective, label respecting, order preserving, and box preserving. In other words, a map $\phi : \mathcal{E}_P \to \mathcal{E}_Q$ such that (i) ϕ is a bijection; (ii) $\lambda_Q \circ \phi = \lambda_P$; (iii) $\phi(\leq_P) \subseteq \leq_Q$; (iv) $\phi(\mathcal{B}_P) \subseteq \mathcal{B}_Q$. If in addition (iii) holds as an equality, ϕ is called* order-reflecting. *If on the other hand (iv) holds as an equality ϕ is* box-reflecting. *A homomorphism that is both order- and box-reflecting is a* (poset with boxes) isomorphism.

In Figure 3 are some examples and a non-example of subsumption between posets. We introduce some notations. $\mathbb{P}_\Sigma$ is the set of posets with boxes. If ϕ is a homomorphism from P to Q, we write $\phi : P \to Q$. If there exists such a homomorphism (respectively an isomorphism) from P to Q, we write $Q \sqsubseteq P$ (resp. $Q \cong P$).

▶ **Lemma 4.** *$\cong$ is an equivalence relation. $\sqsubseteq$ is a partial order with respect to $\cong$.*

▶ Remark 5. Note that the fact that $\sqsubseteq$ is antisymmetric with respect to $\cong$ relies on the finiteness of the posets considered here. Indeed, we can build infinite pomsets that are not isomorphic but have nevertheless homomorphisms between them in both directions.

▶ **Definition 6** (Pomsets with boxes). Pomsets with boxes *are equivalence classes of $\cong$. The set $\mathbf{Pom}_\Sigma$ of pomsets with boxes is defined as $\mathbb{P}_\Sigma/_{\cong}$.*

We now define some elementary poset-building operations.

▶ **Definition 7** (Constants). *Given a symbol $a \in \Sigma$, the* atomic poset *associated with a is defined as $\mathbb{a} := \left\langle \{0\}, [0 \mapsto a], Id_{\{0\}}, \emptyset \right\rangle \in \mathbb{P}_\Sigma$. The empty poset is defined as $\mathbb{e} := \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle \in \mathbb{P}_\Sigma$.*

▶ Remark 8. For any poset $P \in \mathbb{P}_\Sigma$, $P \sqsubseteq \mathbb{e} \Leftrightarrow P \sqsupseteq \mathbb{e} \Leftrightarrow P \cong \mathbb{e}$. This is because each of those relations imply there is a bijection between the events of P and $\mathcal{E}_{\mathbb{e}} = \emptyset$. So we know that P has no events, and since boxes cannot be empty, P has no boxes either. Hence $P \cong \mathbb{e}$.

▶ **Definition 9** (Compositions). *Let P, Q be two posets with boxes. The sequential composition $P \otimes Q$ and parallel composition $P \oplus Q$ are defined by:*

$$P \otimes Q := \langle \mathcal{E}_P \uplus \mathcal{E}_Q, \leq_P \cup \leq_Q \cup (\mathcal{E}_P \times \mathcal{E}_Q), \lambda_P \sqcup \lambda_Q, \mathcal{B}_P \cup \mathcal{B}_Q \rangle$$
$$P \oplus Q := \langle \mathcal{E}_P \uplus \mathcal{E}_Q, \leq_P \cup \leq_Q, \lambda_P \sqcup \lambda_Q, \mathcal{B}_P \cup \mathcal{B}_Q \rangle,$$

where the symbol $\sqcup$ denotes the union of two functions; that is, given $f : A \to C$ and $g : B \to C$, the function $f \sqcup g : A \uplus B \to C$ associates $f(a)$ to $a \in A$ and $g(b)$ to $b \in B$.

Intuitively, $P \oplus Q$ consists of disjoint copies of P and Q side by side. $P \otimes Q$ also contains disjoint copies of P and Q, but also orders every event in P before any event in Q.

▶ **Definition 10** (Boxing). *Given a poset P its* boxing *is denoted by $[P]$ and is defined by: $[P] := \langle \mathcal{E}_P, \leq_P, \lambda_P, \mathcal{B}_P \cup \{\mathcal{E}_P\} \rangle$.*

Boxing a pomset simply amounts to drawing a box around it.

In our running example, the pattern of interest is a subset of the events of the whole run. To capture this, we define the restriction of a poset to a subset of its events.

▶ **Definition 11** (Restriction, sub-poset). *For a given set of events $A \subseteq \mathcal{E}_P$, we define the* restriction of P to A *as $P{\downharpoonleft}_A := \langle A, \leq_P \cap (A \times A), \lambda_P{\downharpoonleft}_A, \mathcal{B}_P \cap \mathcal{P}(A) \rangle$. We say that P is a* sub-poset *of Q, and write $P \Subset Q$, if there is a set $A \subseteq \mathcal{E}_Q$ such that $P \cong Q{\downharpoonleft}_A$.*

Given a poset P, a set of events $A \subseteq \mathcal{E}_P$ is called:

- **nested** if for any box $\beta \in \mathcal{B}_P$ either $\beta \subseteq A$ or $A \cap \beta = \emptyset$;
- **prefix** if for any $e \in A$ and $f \notin A$ we have $e \leq_P f$; and
- **isolated** if for any $e \in A$ and $f \notin A$ we have $e \not\leq_P f$ and $f \not\leq_P e$.

These properties characterize sub-posets of particular interest to P. This is made explicit in the following observation:

▶ **Fact 12.** *Given a poset P and a set of events $A \subseteq \mathcal{E}_P$:*
(i) *A is prefix and nested iff $P \cong P{\downharpoonleft}_A \otimes P{\downharpoonleft}_{\overline{A}}$;*
(ii) *A is isolated and nested iff $P \cong P{\downharpoonleft}_A \oplus P{\downharpoonleft}_{\overline{A}}$.*
(Here $\overline{A}$ denotes the complement of A relative to $\mathcal{E}_P$; that is, $\overline{A} := \mathcal{E}_P \setminus A$.)

This fact is very useful as a way to "reverse-engineer" how a poset was built.

2.1.2 Series–parallel pomsets

In the sequel, we will often restrict our attention to series–parallel pomsets. These are of particular interest since they are defined as those pomsets that can be generated from constants using the operators we have defined.

▶ **Definition 13** (Pomset terms, SP-Pomsets). *A* (pomset) term *is a syntactic expression generated from the following grammar: $s, t \in \mathsf{SP}_\Sigma ::= 1 \mid a \mid s\,;t \mid s \parallel t \mid [s]$. By convention $;$ binds tighter than $\parallel$. A term is interpreted as a poset as follows:*

$$[\![a]\!] := \mathbb{a} \qquad [\![1]\!] := \mathbb{e} \qquad [\![\,[s]\,]\!] := [\,[\![s]\!]\,]$$
$$[\![s\,;t]\!] := [\![s]\!] \otimes [\![t]\!] \qquad [\![s \parallel t]\!] := [\![s]\!] \oplus [\![t]\!].$$

A pomset $[P]_{\cong}$ is called series–parallel *(or SP for short) if it is the interpretation of some term; that is, $\exists s \in \mathsf{SP}_\Sigma : [\![s]\!] \cong P$.*

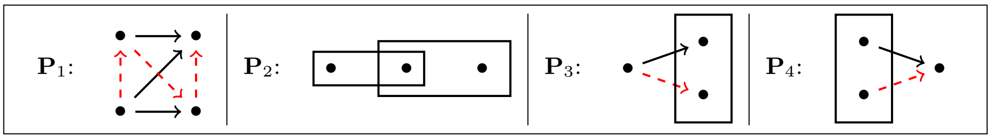


Figure 4 Forbidden patterns of SP-pomsets: dashed arrows (in red) are negated.

▶ **Example 14.** The program in Figure 1 of the running example corresponds to

$$[\![\text{🖶} ; (\text{👓}_x ; \text{🏭}_x ; \text{✍}_x \parallel \text{👓}_y ; \text{🏭}_y ; \text{✍}_y) ; \text{🖶}]\!] .$$

The corrected program, from Figure 2, corresponds to

$$[\![\text{🖶} ; ([\text{👓}_x ; \text{🏭}_x ; \text{✍}_x] \parallel [\text{👓}_y ; \text{🏭}_y ; \text{✍}_y]) ; \text{🖶}]\!] .$$

Finally, the problematic pattern we identified may be represented as $[\![(\text{👓}_x \parallel \text{👓}_y) ; (\text{✍}_x \parallel \text{✍}_y)]\!]$.

Series–parallel pomsets with boxes may also be defined by excluded patterns, in the same style as the characterization of series–parallel pomsets [27, 9, 8]. More precisely, one can prove that a pomset $[P]_{\cong}$ is series–parallel iff and only if it does not contain any of the patterns in Figure 4. These may be expressed formally as follows:

P_1 : $\exists e_1, e_2, e_3, e_4 \in \mathcal{E}_P : e_1 \leq_P e_3 \wedge e_2 \leq_P e_3 \wedge e_2 \leq_P e_4 \wedge e_1 \not\leq_P e_4 \wedge e_2 \not\leq_P e_1 \wedge e_4 \not\leq_P e_3$
P_2 : $\exists e_1, e_2, e_3 \in \mathcal{E}_P, \exists A, B \in \mathcal{B}_P : e_1 \in A \setminus B \wedge e_2 \in A \cap B \wedge e_3 \in B \setminus A$
P_3 : $\exists e_1, e_2, e_3 \in \mathcal{E}_P, \exists A \in \mathcal{B}_P : e_1 \notin A \wedge e_2, e_3 \in A \wedge e_1 \leq_P e_2 \wedge e_1 \not\leq_P e_3$
P_4 : $\exists e_1, e_2, e_3 \in \mathcal{E}_P, \exists A \in \mathcal{B}_P : e_1 \notin A \wedge e_2, e_3 \in A \wedge e_2 \leq_P e_1 \wedge e_3 \not\leq_P e_1$.

We omit the proof of this result here, but the interested reader may find it both in the Coq proof and in the online version referenced above.

These four patterns are invariant under isomorphism, since they only use the ordering between events and the membership of events to boxes. This is consistent with SP being a property of pomsets, rather than just posets. This result provides an alternative view of pomsets with boxes: one may see them as *hyper-pomsets*; that is, pomsets in which some events (the boxes) can be labelled with non-empty pomsets (the contents of the boxes). However, it seems that for our purposes the definition we provide is more convenient. In particular, the definition of hyper-pomset homomorphism is more involved.

2.2 Sets of posets

We now lift our operations and relations to sets of posets. This allows us to enrich our syntax with a non-deterministic choice operator.

▶ **Definition 15** (Orderings on sets of posets). *Let $A, B \subseteq \mathbb{P}_\Sigma$, we define the following:*
Isomorphic inclusion *$A \subsetsim B$ iff $\forall P \in A$, $\exists Q \in B$ such that $P \cong Q$*
Isomorphic equivalence *$A \cong B$ iff $A \subsetsim B \wedge B \subsetsim A$*
Subsumption *$A \sqsubseteq B$ iff $\forall P \in A$, $\exists Q \in B$ such that $P \sqsubseteq Q$.*

▶ Remark 16. Isomorphic inclusion and subsumption are partial orders with respect to isomorphic equivalence, which is an equivalence relation.

▶ **Definition 17** (Operations on sets of posets). *We will use the set-theoretic union of sets of posets, as well as the pointwise liftings of the two products of posets and the boxing operators:*

$$A \otimes B := \{P \otimes Q \mid \langle P, Q \rangle \in A \times B\} \qquad [A] := \{[P] \mid P \in A\}$$
$$A \oplus B := \{P \oplus Q \mid \langle P, Q \rangle \in A \times B\}.$$

Table 1 Equational and inequational logic.

$$\frac{e = f \in A}{A \vdash e = f} \qquad A \vdash e = e \qquad \frac{A \vdash e = f}{A \vdash f = e} \qquad \frac{A \vdash e = f \qquad A \vdash f = g}{A \vdash e = g}$$

$$\sigma, \tau : \Sigma \to \mathtt{T}_\Sigma,\ \frac{\forall a \in \Sigma,\ A \vdash \sigma(a) = \tau(a)}{A \vdash \hat{\sigma}(e) = \hat{\tau}(e)}$$

$$\frac{e = f \in A}{A \vdash e \leq f} \qquad \frac{f = e \in A}{A \vdash e \leq f} \qquad \frac{e \leq f \in A}{A \vdash e \leq f} \qquad A \vdash e \leq e \qquad \frac{A \vdash e \leq f \qquad A \vdash f \leq g}{A \vdash e \leq g}$$

$$\sigma, \tau : \Sigma \to \mathtt{T}_\Sigma,\ \frac{\forall a \in \Sigma,\ A \vdash \sigma(a) \leq \tau(a)}{A \vdash \hat{\sigma}(e) \leq \hat{\tau}(e)}$$

▶ **Definition 18** (Closure of a set of posets). *The* (downwards) closure *of a set of posets S is the smallest set containing S that is downwards closed with respect to the subsumption order; that is, $S{\downarrow} := \{P \in \mathbb{P}_\Sigma \mid \exists Q \in S : P \sqsubseteq Q\}$. Similarly, the* upwards closure *of S is defined as: $S{\uparrow} := \{P \in \mathbb{P}_\Sigma \mid \exists Q \in S : P \sqsupseteq Q\}$.*

▶ Remark 19. $(\_){\downarrow}$ and $(\_){\uparrow}$ are Kuratowski closure operators [16]; i.e., they satisfy the following properties:

$$\emptyset{\downarrow} = \emptyset \qquad A \subseteq A{\downarrow} \qquad A{\downarrow}{\downarrow} = A{\downarrow} \qquad (A \cup B){\downarrow} = A{\downarrow} \cup B{\downarrow}.$$

(And, similarly, for the upwards closure.) Using downwards-closures, we may express subsumption in terms of isomorphic inclusion:

$$A \sqsubseteq B \qquad \Leftrightarrow \qquad A \underset{\sim}{\subseteq} B{\downarrow} \qquad \Leftrightarrow \qquad A{\downarrow} \underset{\sim}{\subseteq} B{\downarrow}.$$

Similarly, the equivalence relation associated with $\sqsubseteq$, defined as the intersection of the relation and its converse, corresponds to the predicate $A{\downarrow} \cong B{\downarrow}$.

▶ **Definition 20.** *Terms are defined by the following grammar:*

$$e, f \in \mathtt{T}_\Sigma ::= 0 \mid 1 \mid a \mid e\,;f \mid e \parallel f \mid e + f \mid [e]\,.$$

Terms can be interpreted as finite sets of posets with boxes as follows:

$$[\![0]\!] := \emptyset \qquad [\![1]\!] := \{\mathbb{e}\} \qquad [\![a]\!] := \{\mathbb{a}\}$$

$$[\![\,[e]\,]\!] := [\,[\![e]\!]\,] \qquad [\![e\,;f]\!] := [\![e]\!] \otimes [\![f]\!] \qquad [\![e + f]\!] := [\![e]\!] \cup [\![f]\!] \qquad [\![e \parallel f]\!] := [\![e]\!] \oplus [\![f]\!]\,.$$

▶ Remark 21. Interpreted as a program, 0 represents failure: this is a program that aborts the whole execution. $+$, on the other hand, represents non-deterministic choice. It can be used to model conditional branching.

2.3 Axiomatic presentations of pomset algebra

We now introduce axioms to capture the various order and equivalence relations we introduced over posets and sets of posets. Given a set of axioms A (i.e., universally quantified identities), we write $A \vdash e = f$ to denote that the pair $\langle e, f\rangle$ belongs to the smallest congruence

Table 2 Axioms.

$s\,;(t\,;u) = (s\,;t)\,;u$	(A1)	$e+(f+g) = (e+f)+g$	(C1)
$s\parallel(t\parallel u) = (s\parallel t)\parallel u$	(A2)	$e+f = f+e$	(C2)
$s\parallel t = t\parallel s$	(A3)	$e+e = e$	(C3)
$1\,;s = s$	(A4)	$0+e = e$	(C4)
$s\,;1 = s$	(A5)	$0\,;e = e\,;0 = 0$	(C5)
$1\parallel s = s$	(A6)	$0\parallel e = 0$	(C6)
$[[s]] = [s]$	(A7)	$e\,;(f+g) = (e\,;f)+(e\,;g)$	(C7)
$[1] = 1$	(A8)	$(e+f)\,;g = (e\,;g)+(f\,;g)$	(C8)
		$e\parallel(f+g) = (e\parallel f)+(e\parallel g)$	(C9)
$(s\parallel t)\,;(u\parallel v) \leq (s\,;u)\parallel(t\,;v)$	(B1)	$[0] = 0$	(C10)
$[s] \leq s$	(B2)	$[e+f] = [e]+[f]$	(C11)

containing every axiom in A. Equivalently, $A \vdash e = f$ holds iff this statement is derivable in equational logic, as described in Table 1. Similarly, $A \vdash e \leq f$ is the smallest precongruence containing A, where equality axioms are understood as pairs of inequational axioms. An inference system is also provided in Table 1. We will consider the following sets of axioms:

$$
\begin{aligned}
\mathrm{BiMon}_{\square} &:= (\mathrm{A1}) - (\mathrm{A8}) && \text{(Bimonoid with boxes)}\\
\mathrm{CMon}_{\square} &:= \mathrm{BiMon}_{\square}, (\mathrm{B1}), (\mathrm{B2}) && \text{(Concurrent monoid with boxes)}\\
\mathrm{SR}_{\square} &:= \mathrm{BiMon}_{\square}, (\mathrm{C1}) - (\mathrm{C11}) && \text{(Bisemiring with boxes)}\\
\mathrm{CSR}_{\square} &:= \mathrm{SR}_{\square}, (\mathrm{B1}), (\mathrm{B2}). && \text{(Concurrent semiring with boxes)}
\end{aligned}
$$

In the last theory, inequational axioms $e \leq f$ should be read as $e + f = f$. Indeed one can show that for $A \in \{\mathrm{SR}_{\square}, \mathrm{CSR}_{\square}\}$, we have

$$A \vdash e \leq f \Leftrightarrow A \vdash e + f = f \qquad\qquad A \vdash e = f \Leftrightarrow A \vdash e \leq f \wedge A \vdash f \leq e.$$

These axioms capture the relations $\cong$ and $\sqsubseteq$:

▶ **Theorem 22.** *For any pair of terms $s, t \in \mathtt{SP}_\Sigma$, the following hold:*

$$\llbracket s \rrbracket \cong \llbracket t \rrbracket \Leftrightarrow \mathrm{BiMon}_{\square} \vdash s = t \tag{2.1}$$

$$\llbracket s \rrbracket \sqsubseteq \llbracket t \rrbracket \Leftrightarrow \mathrm{CMon}_{\square} \vdash s \leq t. \tag{2.2}$$

The following lemma allows us to extend seamlessly our completeness theorem from $\mathrm{BiMon}_{\square}$ to $\mathrm{SR}_{\square}$ and from $\mathrm{CMon}_{\square}$ to $\mathrm{CSR}_{\square}$.

▶ **Lemma 23.** *There exists a function $T\_ : \mathtt{T}_\Sigma \to \mathcal{P}_f\left(\mathtt{SP}_\Sigma\right)$ such that: $\mathrm{SR}_{\square} \vdash e = \sum_{s \in T_e} s$ and $\llbracket e \rrbracket \cong \{\llbracket s \rrbracket \mid s \in T_e\}$.*

From there, we can easily establish the following completeness results:

▶ **Theorem 24.** *For any pair of terms $e, f \in \mathtt{T}_\Sigma$, the following hold:*

$$\llbracket e \rrbracket \cong \llbracket f \rrbracket \Leftrightarrow \mathrm{SR}_{\square} \vdash e = f \tag{2.3}$$

$$\llbracket e \rrbracket\!\downarrow\, \cong \llbracket f \rrbracket\!\downarrow\, \Leftrightarrow \mathrm{CSR}_{\square} \vdash e = f. \tag{2.4}$$

3 Logic for pomsets with boxes

We introduce a logic for reasoning about pomsets with boxes, in the form of a bunched modal logic, in the sense of [21, 7, 5, 1, 23], with substructural connectives corresponding to each of sequential and concurrent composition. Modalities characterize boxes and locality. The logic is also conceptually related to Concurrent Separation Logic [19, 3].

In contrast with other work, pomset logic is a logic of *behaviours*. A behaviour is a run of some program, represented as a pomset. The logic describes such behaviours in terms of the order in which instructions are, or can be, executed, and the separation properties of sub-runs. Note, in particular, that we do not define any notion of *state*. On the contrary, existing approaches, such as dynamic logic and Hennessy–Milner logic for example, put the emphasis on the state of the machine before and after running the program. Typically, the assertion language describes the memory-states, and some accessibility relations between them. The semantics then relies on labelled transition systems to interpret action modalities.

Here, the satisfaction relation (given in Definition 26) directly defines a relation between sets of behaviours and formulas. An intuitionistic version of the semantics given in Definition 26 might be set up – cf. Tarski's semantics and the semantics of relevant logic – in terms of (ternary) relations on behaviours.

3.1 Pomset logic: definitions

We generate the set of formulas $\mathtt{F}_\Sigma$ and the set of positive formulas $\mathtt{F}^+_\Sigma$ as follows:

$$\phi,\psi \in \mathtt{F}^+_\Sigma ::= \bot \mid a \mid \phi \vee \psi \mid \phi \wedge \psi \mid \phi \blacktriangleright \psi \mid \phi \star \psi \mid [\phi] \mid (\!|\phi|\!)$$
$$\phi,\psi \in \mathtt{F}_\Sigma ::= \bot \mid a \mid \phi \vee \psi \mid \phi \wedge \psi \mid \phi \blacktriangleright \psi \mid \phi \star \psi \mid [\phi] \mid (\!|\phi|\!) \mid \neg\phi$$

▶ Remark 25. Here the atomic predicates are chosen to be exactly Σ. Another natural choice would be a separate set *Prop* of atomic predicates, together with a valuation $v : \mathit{Prop} \to \mathcal{P}(\Sigma)$ to indicate which actions satisfy which predicate. Both definitions are equivalent:

- to encode a formula over *Prop* as a formula over Σ, simply replace every predicate $p \in \mathit{Prop}$ with the formula $\bigvee_{a \in v(p)} a$
- to encode a formula over Σ as one over *Prop*, we need to make the customary assumption that $\forall a \in \Sigma,\ \exists p \in \mathit{Prop} :\ v(p) = \{a\}$.

These formulas are interpreted over posets. We define a satisfaction relation $\models_R$ that is parametrized by a relation $R \subseteq \mathbb{P}_\Sigma \times \mathbb{P}_\Sigma$ (to be instantiated later on with $\cong$, $\sqsubseteq$, and $\sqsupseteq$).

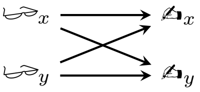
▶ **Definition 26.** *$P \models_R \phi$ is defined by induction on $\phi \in \mathtt{F}_\Sigma$:*

- $P \models_R \bot$ *iff* $R(P, \mathbb{\varepsilon})$
- $P \models_R a$ *iff* $R(P, \mathbb{a})$
- $P \models_R \neg\phi$ *iff* $P \not\models_R \phi$
- $P \models_R \phi \vee \psi$ *iff* $P \models_R \phi$ *or* $P \models_R \psi$
- $P \models_R \phi \wedge \psi$ *iff* $P \models_R \phi$ *and* $P \models_R \psi$
- $P \models_R \phi \blacktriangleright \psi$ *iff* $\exists P_1, P_2$ *such that* $R(P, P_1 \otimes P_2)$ *and* $P_1 \models_R \phi$ *and* $P_2 \models_R \psi$
- $P \models_R \phi \star \psi$ *iff* $\exists P_1, P_2$ *such that* $R(P, P_1 \oplus P_2)$ *and* $P_1 \models_R \phi$ *and* $P_2 \models_R \psi$
- $P \models_R [\phi]$ *iff* $\exists Q$ *such that* $R(P, [Q])$ *and* $Q \models_R \phi$
- $P \models_R (\!|\phi|\!)$ *iff* $\exists P', Q$ *such that* $R(P, P')$ *and* $P' ⫀ Q$ *and* $Q \models_R \phi$.

The operator $[-]$ describes the (encapsulated) properties of boxed terms. The operator $\llparenthesis - \rrparenthesis$ identifies a property of a term that is obtained by removing parts, including boxes and events, of its satisfying term (i.e., its world) such that remainder satisfies the formula that it guards. The meanings of these operators are discussed more fully in Section 4.2.

Note that $\models_{\sqsupseteq}$ and $\models_{\sqsubseteq}$ will only be used with positive formulas. Given a formula ϕ and a relation R, we may define the R-semantics of ϕ as $[\![\phi]\!]_R := \{P \in \mathbb{P}_\Sigma \mid P \models_R \phi\}$.

▶ **Example 27.** Recall the problematic pattern we saw in the running example; i.e.,



This pattern can be represented by the formula **conflict** $:= \llparenthesis (\text{👓}_x \star \text{👓}_y) \blacktriangleright (\text{✍}_x \star \text{✍}_y) \rrparenthesis$.

We may also interpret these formulas over sets of posets. We consider here two ways a set of posets X may satisfy a formula:

- X satisfies ϕ *universally* if every poset in X satisfies ϕ;
- X satisfies ϕ *existentially* if some poset in X satisfies ϕ.

Combined with our three satisfaction relations for pomsets, this yields six definitions:

$$X \models^{\forall}_{\cong} \phi \text{ iff } \forall P \in X, P \models_{\cong} \phi \qquad X \models^{\exists}_{\cong} \phi \text{ iff } \exists P \in X, P \models_{\cong} \phi$$
$$X \models^{\forall}_{\sqsupseteq} \phi \text{ iff } \forall P \in X, P \models_{\sqsupseteq} \phi \qquad X \models^{\exists}_{\sqsupseteq} \phi \text{ iff } \exists P \in X, P \models_{\sqsupseteq} \phi$$
$$X \models^{\forall}_{\sqsubseteq} \phi \text{ iff } \forall P \in X, P \models_{\sqsubseteq} \phi \qquad X \models^{\exists}_{\sqsubseteq} \phi \text{ iff } \exists P \in X, P \models_{\sqsubseteq} \phi.$$

For a term $e \in \mathtt{T}_\Sigma$, we write $e \models^{y}_{R} \phi$ to mean $[\![e]\!] \models^{y}_{R} \phi$. In terms of R-semantics, these definitions may be formalized as:

$$e \models^{\exists}_{R} \phi \Leftrightarrow [\![e]\!] \cap [\![\phi]\!]_R \neq \emptyset \qquad e \models^{\forall}_{R} \phi \Leftrightarrow [\![e]\!] \subseteq [\![\phi]\!]_R\,. \tag{3.1}$$

3.2 Properties of pomset logic

We now discuss some of the properties of pomset logic. First, notice that if the relation R is transitive, then for any posets P, Q and any formula $\phi \in \mathtt{F}^{+}_{\Sigma}$, we have that:

$$P\ R\ Q \text{ and } Q \models_R \phi \Rightarrow P \models_R \phi. \tag{3.2}$$

If, additionally, R is symmetric, this property may be strengthened to

$$\forall P, Q \in \mathbb{P}_\Sigma,\ \forall \phi \in \mathtt{F}_\Sigma,\ \text{if } P\ R\ Q, \text{ then } P \models_R \phi \Leftrightarrow Q \models_R \phi. \tag{3.3}$$

Furthermore, increasing the relation R increases the satisfaction relation as well:

$$R \subseteq R' \Rightarrow \forall \phi \in \mathtt{F}^{+}_{\Sigma}, \forall P \in \mathbb{P}_\Sigma,\ P \models_R \phi \Rightarrow P \models_{R'} \phi. \tag{3.4}$$

From these observations and (3.1), we obtain the following characterizations of the universal satisfaction relations for $R \in \{\cong, \sqsubseteq, \sqsupseteq\}$:

$$e \models^{\forall}_{\cong} \phi \Leftrightarrow [\![e]\!] \underset{\sim}{\subset} [\![\phi]\!]_{\cong} \tag{3.5}$$
$$e \models^{\forall}_{\sqsubseteq} \phi \Leftrightarrow [\![e]\!] \sqsubseteq [\![\phi]\!]_{\sqsubseteq} \tag{3.6}$$
$$e \models^{\forall}_{\sqsupseteq} \phi \Leftrightarrow \forall P \in [\![e]\!]\,,\ \exists Q \in [\![\phi]\!]_{\sqsupseteq} : P \sqsupseteq Q. \tag{3.7}$$

Additionally, the following preservation properties hold for sets of posets:

$$e \underset{\sim}{\subset} f \Rightarrow \forall \phi \in \mathtt{F}_\Sigma,\ \left(e \models^{\exists}_{\cong} \phi \Rightarrow f \models^{\exists}_{\cong} \phi\right) \wedge \left(f \models^{\forall}_{\cong} \phi \Rightarrow e \models^{\forall}_{\cong} \phi\right) \tag{3.8}$$

$$e \sqsubseteq f \Rightarrow \forall \phi \in \mathtt{F}^{+}_\Sigma,\ \left(e \models^{\exists}_{\sqsupseteq} \phi \Rightarrow f \models^{\exists}_{\sqsupseteq} \phi\right) \wedge \left(f \models^{\forall}_{\sqsubseteq} \phi \Rightarrow e \models^{\forall}_{\sqsubseteq} \phi\right). \tag{3.9}$$

We can build formulas from series–parallel terms: $\phi(a) := a$, $\phi(1) := \bot$, $\phi\left([s]\right) := [\phi(s)]$, $\phi(s\,;t) := \phi(s) \blacktriangleright \phi(t)$, and $\phi(s \parallel t) := \phi(s) \star \phi(t)$. Using $T\_$, we generalize this construction our full syntax: given a term $e \in \mathtt{T}_\Sigma$, we define the formula $\Phi(e) := \bigvee_{s \in T_e} \phi(s)$. These formulas are closely related to terms thanks to the following lemma:

▶ **Lemma 28.** *For any term $s \in \mathtt{SP}_\Sigma$ and any poset P, we have:*

$$P \models_{\cong} \phi(s) \Leftrightarrow P \cong [\![s]\!] \qquad P \models_{\sqsupseteq} \phi(s) \Leftrightarrow P \sqsupseteq [\![s]\!] \qquad P \models_{\sqsubseteq} \phi(s) \Leftrightarrow P \sqsubseteq [\![s]\!].$$

For a term $e \in \mathtt{T}_\Sigma$ and a set of posets $X \subseteq \mathbb{P}_\Sigma$, we have:

$$X \models^{\forall}_{\cong} \Phi(e) \Leftrightarrow X \underset{\sim}{\subset} [\![e]\!] \qquad X \models^{\forall}_{\sqsubseteq} \Phi(e) \Leftrightarrow X \sqsubseteq [\![e]\!].$$

As an immediate corollary, for any $e \in \mathtt{T}_\Sigma$ and any $s \in \mathtt{SP}_\Sigma$, we obtain that:

$$e \models^{\exists}_{\cong} \phi(s) \Leftrightarrow [\![s]\!] \in [\![e]\!] \qquad e \models^{\exists}_{\sqsupseteq} \phi(s) \Leftrightarrow [\![s]\!] \in [\![e]\!]\!\downarrow \tag{3.10}$$

We can now establish adequacy lemmas. These should be understood as appropriate formulations of the completeness theorems relating operational equivalence and logical equivalence in the sense of van Benthem [2] and Hennessy–Milner [10, 18] for this logic (cf. [1]). From the results we have established so far, we may directly prove the following:

▶ **Proposition 29.** *For a pair of series–parallel terms $s, t \in \mathtt{SP}_\Sigma$,*

$$\mathrm{BiMon}_{\square} \vdash s = t \Leftrightarrow \forall \phi \in \mathtt{F}_\Sigma,\ \left([\![s]\!] \models_{\cong} \phi \Leftrightarrow [\![t]\!] \models_{\cong} \phi\right) \tag{3.11}$$

$$\mathrm{CMon}_{\square} \vdash s \leq t \Leftrightarrow \forall \phi \in \mathtt{F}^{+}_\Sigma,\ \left([\![s]\!] \models_{\sqsupseteq} \phi \Rightarrow [\![t]\!] \models_{\sqsupseteq} \phi\right). \tag{3.12}$$

This extends to sets of pomsets in the following sense:

▶ **Proposition 30.** *Given two terms $e, f \in \mathtt{T}_\Sigma$, the following equivalences hold:*

$$\mathrm{SR}_{\square} \vdash e \leq f \Leftrightarrow \left(\forall \phi,\ e \models^{\exists}_{\cong} \phi \Rightarrow f \models^{\exists}_{\cong} \phi\right) \Leftrightarrow \left(\forall \phi,\ f \models^{\forall}_{\cong} \phi \Rightarrow e \models^{\forall}_{\cong} \phi\right) \tag{3.13}$$

$$\mathrm{SR}_{\square} \vdash e = f \Leftrightarrow \left(\forall \phi,\ e \models^{\exists}_{\cong} \phi \Leftrightarrow f \models^{\exists}_{\cong} \phi\right) \Leftrightarrow \left(\forall \phi,\ e \models^{\forall}_{\cong} \phi \Leftrightarrow f \models^{\forall}_{\cong} \phi\right) \tag{3.14}$$

$$\mathrm{CSR}_{\square} \vdash e \leq f \Leftrightarrow \left(\forall \phi,\ e \models^{\exists}_{\sqsupseteq} \phi \Rightarrow f \models^{\exists}_{\sqsupseteq} \phi\right) \Leftrightarrow \left(\forall \phi,\ f \models^{\forall}_{\sqsubseteq} \phi \Rightarrow e \models^{\forall}_{\sqsubseteq} \phi\right) \tag{3.15}$$

$$\mathrm{CSR}_{\square} \vdash e = f \Leftrightarrow \left(\forall \phi,\ e \models^{\exists}_{\sqsupseteq} \phi \Leftrightarrow f \models^{\exists}_{\sqsupseteq} \phi\right) \Leftrightarrow \left(\forall \phi,\ e \models^{\forall}_{\sqsubseteq} \phi \Leftrightarrow f \models^{\forall}_{\sqsubseteq} \phi\right). \tag{3.16}$$

4 Local Reasoning

Some of the discussions in this section do not rely on which satisfaction relation we pick. When this is the case, we use the symbol $\models$ to mean any of the relations $\models_{\cong}, \models_{\sqsupseteq}, \models_{\sqsubseteq}$.

4.1 Modularity

Pomset logic enjoys a high level of compositionality, much like algebraic logic. Formally, this comes from the following principle:

If $e \models \phi$ and $\forall a,\ \sigma a \models \tau a$, then $\hat{\sigma} e \models \hat{\tau} \phi$.

This makes possible the following verification scenario: Let P be a large program, involving a number of simpler sub-programs $P_1, \ldots, P_n$. We may simplify P by replacing the sub-programs by uninterpreted symbols $x_1, \ldots, x_n$. We then check that this simplified program satisfies a formula Φ, the statement of which might involve the x_i. We then separately determine for each sub-program P_i some specification ϕ_i. Finally, using the principle we just stated, we can show that the full program P satisfies the formula Φ', obtained by replacing the x_i with ϕ_i.

4.2 Frame rule

A key of objective of applied, modelling-oriented, work in logic and semantics is to understand systems – such as complex programs, large-scale distributed systems, and organizations – compositionally. That is, we seek understand how the system is made of components that can be understood independently of one another. A key aspect of this is what has become known as *local reasoning*. That is, that the pertinent logical properties of the components of a system should be independent of their context.

In the world of Separation Logic [26, 12, 19], for reasoning about how computer programs manipulate memory, O'Hearn, Reynolds, and Yang [22] suggest that

> "To understand how a program works, it should be possible for reasoning and specification to be confined to the cells that the program actually accesses. The value of any other cell will automatically remain unchanged."

In this context, a key idea is that of the "footprint" of a program; that is, that part of memory that is, in an appropriate sense, used by the program [24]. If, in an appropriate sense, a program executes correctly, or "safely", on its footprint, then the so-called "frame property" ensures that the resources present outside of the footprint and, by implication, their inherent logical properties, are unchanged by the program.

In the setting of Separation Logic, the frame property is usually represented by a Hoare-triple rule of the form

$$\frac{\{\phi\}C\{\psi\}}{\{\phi * \chi\}C\{\psi * \chi\}} \quad C \text{ is independent of } \chi.$$

That is, the formula χ does not include any variables (from the memory) that are modified by the program C.

In order to formulate the frame property in our framework, we first fix the notion of independence between a program and a formula. We say that a pomset P is *R-independent* of a formula ϕ, written $P \mathrel{\#_R} \phi$ if $P \not\models_R \llparenthesis [\phi] \rrparenthesis$. Since independence is meant to prevent overlap, the use of the $\llparenthesis - \rrparenthesis$ modality should come as no surprise.

To explain the need for the $[-]$ modality, first consider a pomset P satisfying $\llparenthesis \phi \rrparenthesis$. To extract a witness of this fact, we must remove parts of P, including boxes and events, such that the remainder satisfies ϕ. However, there are no restrictions on the relationship between the remaining events and those we have deleted. In a sequence of three events, we are allowed to keep the two extremities, and delete the middle one. In contrast, to get a witness of $\llparenthesis [\phi] \rrparenthesis$, we need to identify a box on P whose contents satisfy ϕ, and remove all events external to that box. The result is that the deleted events, that is, the context of our witness, can only appear outside the box, and must treat all events inside uniformly. In other words, these events can interact with the behaviour encapsulated in the box, but cannot interact with individual components inside. For this reason, the frame properties given in Proposition 31 are expressed using $[\phi]$ – that is, the encapsulation of ϕ – rather than ϕ.

$$\begin{aligned}
\mathtt{VoteProc} &:= \mathtt{Choose}\,;\mathtt{Publish}\\
\mathtt{Choose} &:= [\mathtt{Vote}\,(1)] \parallel \cdots \parallel [\mathtt{Vote}\,(n)]\\
\mathtt{Vote}\,(i) &:= \sum_{1\leqslant j\leqslant k} \text{✉}_{i,j}\,;\text{👓}_j\,;\text{🏭}\,;\text{✍}_j\\
\mathtt{Publish} &:= \text{☎}_1 \parallel \cdots \parallel \text{☎}_n
\end{aligned}$$

☎$_i$: the contents of counters $c_1, \ldots, c_k$ is sent to voter v_i
✉$_{i,j}$: voter v_i chooses counter c_j
👓$_j$: the content of counter c_j is loaded into a local variable
🏭 : the local variable is incremented
✍$_j$: the content of the local variable is stored in counter c_j

Figure 5 Voting protocol.

With this definition, we can now state three frame rules, enabling local reasoning with respect to the parallel product, sequential prefixing, and sequential suffixing.

▶ **Proposition 31** (Frame properties). *If $P \#_{\cong} \phi$, and $Q \models_{\cong} [\phi]$, then it holds that:*
(i) $\forall \psi \in \mathtt{F}_\Sigma,\ P \models_{\cong} \psi \Leftrightarrow P \oplus Q \models_{\cong} \psi \star [\phi]$*;*
(ii) $\forall \psi \in \mathtt{F}_\Sigma,\ P \models_{\cong} \psi \Leftrightarrow P \otimes Q \models_{\cong} \psi \blacktriangleright [\phi]$*;*
(iii) $\forall \psi \in \mathtt{F}_\Sigma,\ P \models_{\cong} \psi \Leftrightarrow Q \otimes P \models_{\cong} [\phi] \blacktriangleright \psi$*.*

▶ Remark 32. Note that this lemma does not hold for $\sqsubseteq$ or $\sqsupseteq$ instead of $\simeq$. The left-to-right implications always hold, but the converse may be fail. However, this principle may be extended to sets of pomsets. Indeed, if we define the independence relation for sets of pomsets as $A \#_R \phi := \forall P \in A,\ P \#_R \phi$, then Proposition 31 holds for both $\models^{\forall}_{\cong}$ and $\models^{\exists}_{\cong}$.

4.3 Example

In this section, we present an example program, and showcase reasoning principles of pomset logic. In particular, we will highlight the use of local reasoning when appropriate.

Consider the following voting protocol: a fixed number of voters, $v_1, \ldots, v_n$, are each asked to increment one of the counters $c_1, \ldots, c_k$. The tally is then sent to each of the v_i, to inform them of the result. The increment is implemented similarly to our running example of the distributed counter (Example 1). The implementation of the protocol is displayed in Figure 5, together with the intended semantics of the atomic actions.

Conflict

As in Example 1, if we forgo the boxes in `Choose`, we cannot enforce mutual exclusion. Recall that the undesirable behaviour is captured by the following formula:

$$\mathbf{conflict}_j := \llparenthesis(\text{👓}_j \star \text{👓}_j) \blacktriangleright (\text{✍}_j \star \text{✍}_j)\rrparenthesis$$

We may see this by defining an alternative (faulty) protocol:

$$\mathtt{VoteProc}' := (\mathtt{Vote}\,(1) \parallel \cdots \parallel \mathtt{Vote}\,(n))\,;\mathtt{Publish}$$

and then checking that this protocol displays the behaviour we wanted to avoid:

$$\mathtt{VoteProc}' \models^{\exists}_{\sqsupseteq} \mathbf{conflict}_j.$$

This statement should be read as “there is a pomset in $[\![\texttt{VoteProc}']\!]$ that is larger than one containing a conflict”. We can show the existence of this “bug” by local reasoning. We may first prove that $\texttt{Vote}\,(i)\parallel\texttt{Vote}\,(i')\models^{\exists}_{\sqsupseteq}\mathbf{conflict}_j$ (for some arbitrary $i\neq i'$). The properties of $\llparenthesis - \rrparenthesis$ then allow us to deduce that

$$\texttt{VoteProc}'\cong(\texttt{Vote}\,(i)\parallel\texttt{Vote}\,(i')\parallel\cdots)\,;\cdots\models^{\exists}_{\sqsupseteq}\llparenthesis\mathbf{conflict}_j\rrparenthesis\equiv\mathbf{conflict}_j.$$

The implementation in Figure 5 avoids this problem, and indeed it holds that:

$$\texttt{VoteProc}\not\models^{\exists}_{\sqsupseteq}\mathbf{conflict}_j.$$

However, showing that this formula is *not* satisfied by the program is less straightforward and, in particular, cannot be done locally: we have to enumerate all possible sub-pomsets, and check that none provide a suitable witness.

Sequential separation

In our protocol, the results of the vote are only communicated after every participant has voted. This is specified by the following statement:

$$\mathbf{SendAfterVote}:=\llparenthesis\neg\left(\bigvee_i\text{☎}_i\right)\rrparenthesis\blacktriangleright\llparenthesis\neg\left(\bigvee_{i,j}\text{✉}_{i,j}\right)\rrparenthesis.$$

This may be checked modularly. Indeed, one may prove by simple syntactic analysis that

$$\texttt{Choose}\models^{\forall}_{\cong}\llparenthesis\neg\left(\bigvee_i\text{☎}_i\right)\rrparenthesis\qquad\text{and}\qquad\texttt{Publish}\models^{\forall}_{\cong}\llparenthesis\neg\left(\bigvee_{i,j}\text{✉}_{i,j}\right)\rrparenthesis.$$

Therefore, we may combine these to get that:

$$\texttt{VoteProc}=\texttt{Choose}\,;\texttt{Publish}\models^{\forall}_{\cong}\llparenthesis\neg\left(\bigvee_i\text{☎}_i\right)\rrparenthesis\blacktriangleright\llparenthesis\neg\left(\bigvee_{i,j}\text{✉}_{i,j}\right)\rrparenthesis=\mathbf{SendAfterVote}.$$

For voter i, two of the most meaningful steps are $\text{✉}_{i,j}$ and ☎_i, i.e. when the vote is cast and when the result of the vote is forwarded to them. Using the macro $\mathbf{choose}_i:=\bigvee_j\text{✉}_{i,j}$, we can specify that during the protocol, each voter first votes, and then gets send the result:

$$\mathbf{VoteThenSend}:=\llparenthesis(\mathbf{choose}_1\blacktriangleright\text{☎}_1)\star\ldots\star(\mathbf{choose}_n\blacktriangleright\text{☎}_n)\rrparenthesis.$$

Unique votes

Another important feature of this protocol is that each voter may only cast a single vote. Knowing that each voter controls a single box, we express this property with the statement:

$$\texttt{VoteProc}\not\models^{\exists}_{\sqsubseteq}\bigvee_{j,j'}\llparenthesis[\llparenthesis\text{✍}_j\star\text{✍}_{j'}\rrparenthesis]\rrparenthesis.$$

Since we use the relation $\models^{\exists}_{\sqsubseteq}$ with the connective $\star$, we allow any possible ordering of the two write events. The only constraint is that there should be at least two of them in the same box. As for the “conflict” property, if the “bad” behaviour were to happen, one could prove it compositionally. However, disproving the existence of such a behaviour is a more global process, involving the exploration of all possible sub-pomsets.

Frame property

As we have seen in previous examples, proving that a formula does not hold can be challenging, because the non-existence of a local pattern is *not* a local property. We may circumvent this problem by adding more boxes in both programs and formulas. This is related to a common pattern in parallel programming: in a multi-threaded program, one may insert fences to "tame" concurrency. Doing so simplifies program analysis, at the cost of some efficiency. Similarly, since adding boxes restricts behaviours – thus disallowing some possible optimizations – the analysis of a program becomes simpler and more efficient.

We illustrate this with the following statement:

$$[\mathtt{Choose}]\,;[\mathtt{Publish}] \not\models_{\cong}^{\forall} \llparenthesis \text{✍} \blacktriangleright \text{✍} \rrparenthesis \blacktriangleright [\phi] \qquad \text{where } \phi := \neg\left(\bot \vee \llparenthesis [\llparenthesis \text{✍} \rrparenthesis] \rrparenthesis\right).$$

$\llparenthesis \text{✍} \blacktriangleright \text{✍} \rrparenthesis$ indicates that two "write" instructions can be executed in sequence, while ϕ denotes a non-empty pomset, not containing any boxes with a "write" event inside. We can first prove properties of the subprograms:

$$[\mathtt{Publish}] \models_{\cong}^{\forall} [\phi] \qquad [\mathtt{Choose}] \not\models_{\cong}^{\forall} \llparenthesis [\phi] \rrparenthesis \qquad [\mathtt{Choose}] \not\models_{\cong}^{\forall} \llparenthesis \text{✍} \blacktriangleright \text{✍} \rrparenthesis.$$

Since $[\mathtt{Choose}] \#_{\cong} \phi$ and $[\mathtt{Publish}] \models_{\cong}^{\forall} [\phi]$, we obtain from the frame rule that

$$[\mathtt{Choose}]\,;[\mathtt{Publish}] \models_{\cong}^{\forall} \llparenthesis \text{✍} \blacktriangleright \text{✍} \rrparenthesis \blacktriangleright [\phi] \Leftrightarrow [\mathtt{Choose}] \models_{\cong}^{\forall} \llparenthesis \text{✍} \blacktriangleright \text{✍} \rrparenthesis.$$

Since we have locally disproved the latter, we may deduce that the former does not hold.

5 Future work

In this paper, we have not considered the CKA operator $-^{\star}$. A natural further step would be to do so, with the corresponding need to consider versions of pomset logic with fixed points. Connections with Hoare-style program logics, such as Concurrent Separation Logic [3, 20] with its concrete semantics, should also be considered.

Our satisfaction relation over pomsets is defined inductively. However, the satisfaction relations we define for sets of pomsets is not: we define in terms of the former relation. For practical purposes, such as model-checking, it would be useful to have a similar inductive definition for sets of pomsets.

It is also worth noticing that the definitions and statements in Section 2 are straightforward generalizations of their counterparts in CKA (without boxes); even the proofs of those results follow a similar strategy. However, we could reuse almost no result from CKA: instead we had to reprove everything from scratch. This situation is deeply unsatisfactory, and we plan on investigating techniques to better "recycle" proofs in this context. Recent work on (C)KA with *hypotheses* [6, 14] seems to be a step towards this goal.

References

1 Gabrielle Anderson and David Pym. A calculus and logic of bunched resources and processes. *Theor. Comp. Sci.*, 614, 2016. doi:10.1016/j.tcs.2015.11.035.

2 Johan Van Benthem. *Logical Dynamics of Information and Interaction*. Cambridge University Press, 2014.

3 Stephen Brookes and Peter W. O'Hearn. Concurrent Separation Logic. *ACM SIGLOG News*, 3(3), August 2016. doi:10.1145/2984450.2984457.

4 Paul Brunet, Damien Pous, and Georg Struth. On Decidability of Concurrent Kleene Algebra. In *CONCUR*, 2017. doi:10.4230/LIPIcs.CONCUR.2017.28.

5 Matthew Collinson and David Pym. Algebra and Logic for Resource-Based Systems Modelling. *Math. Str. Comp. Sci.*, 19(5), 2009. doi:10.1017/S0960129509990077.

6 Amina Doumane, Denis Kuperberg, Damien Pous, and Pierre Pradic. Kleene Algebra with Hypotheses. In *FoSSaCS*, 2019. doi:10.1007/978-3-030-17127-8_12.

7 Didier Galmiche, Daniel Méry, and David Pym. The Semantics of BI and Resource Tableaux. *Math. Str. Comp. Sci.*, 15(6), December 2005. doi:10.1017/S0960129505004858.

8 Jay L. Gischer. The Equational Theory of Pomsets. *Theor. Comp. Sci.*, 61(2-3), 1988. doi:10.1016/0304-3975(88)90124-7.

9 Jan Grabowski. On partial languages. *Fund. Math.*, 4(2), 1981.

10 Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In *ICALP*, 1980. doi:10.1007/3-540-10003-2_79.

11 C.A.R. Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. Concurrent Kleene Algebra. In *CONCUR*, 2009. doi:10.1007/978-3-642-04081-8_27.

12 Samin S. Ishtiaq and Peter W. O'Hearn. BI as an Assertion Language for Mutable Data Structures. In *POPL*, 2001. doi:10.1145/373243.375719.

13 Lalita Jategaonkar and Albert R. Meyer. Deciding true concurrency equivalences on safe, finite nets. *Theor. Comp. Sci.*, 154(1), 1996. doi:10.1016/0304-3975(95)00132-8.

14 Tobias Kappé, Paul Brunet, Alexandra Silva, Jana Wagemaker, and Fabio Zanasi. Concurrent Kleene Algebra with Observations: From Hypotheses to Completeness. In *FoSSaCS*, 2020. doi:10.1007/978-3-030-45231-5_20.

15 Tobias Kappé, Paul Brunet, Alexandra Silva, and Fabio Zanasi. Concurrent Kleene Algebra: Free Model and Completeness. In *ESOP*, 2018. doi:10.1007/978-3-319-89884-1_30.

16 Casimir Kuratowski. Sur l'opération $\bar{A}$ de l'Analysis Situs. *Fund. Math.*, 3(1), 1922.

17 Michael R. Laurence and Georg Struth. Completeness Theorems for Bi-Kleene Algebras and Series-Parallel Rational Pomset Languages. In *RAMiCS*, 2014. doi:10.1007/978-3-319-06251-8_5.

18 Robin Milner. *Communication and Concurrency.* Prentice-Hall, Inc., 1989.

19 Peter W. O'Hearn. Resources, Concurrency, and Local Reasoning. *Theor. Comp. Sci.*, 375(1-3), May 2007. doi:10.1016/j.tcs.2006.12.035.

20 Peter W. O'Hearn, Rasmus L. Petersen, Jules Villard, and Akbar Hussain. On the Relation between Concurrent Separation Logic and Concurrent Kleene Algebra. *J. Log. Algebr. Methods Program.*, 84(3), May 2015. doi:10.1016/j.jlamp.2014.08.002.

21 Peter W. O'Hearn and David J. Pym. The Logic of Bunched Implications. *Bull. Symb. Log.*, 5(2), 1999. doi:10.2307/421090.

22 Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local Reasoning about Programs That Alter Data Structures. In *CSL*, 2001. doi:10.1007/3-540-44802-0_1.

23 David Pym. Resource Semantics: Logic as a Modelling Technology. *ACM SIGLOG News*, 6(2), April 2019. doi:10.1145/3326938.3326940.

24 Mohammad Raza and Philippa Gardner. Footprints in Local Reasoning. In *FoSSaCS*, 2008. doi:10.1007/978-3-540-78499-9_15.

25 Christian Retoré. Pomset logic: A non-commutative extension of classical linear logic. In *TLCA*, 1997. doi:10.1007/3-540-62688-3_43.

26 John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, July 2002. doi:10.1109/LICS.2002.1029817.

27 Jacobo Valdes, Robert E. Tarjan, and Eugene L. Lawler. The Recognition of Series Parallel Digraphs. *SIAM J. Comput.*, 11(2), 1982. doi:10.1137/0211023.

Undecidability of Semi-Unification on a Napkin

Andrej Dudenhefner
Saarland University, Saarbrücken, Germany
dudenhefner@ps.uni-saarland.de

Abstract

Semi-unification (unification combined with matching) has been proven undecidable by Kfoury, Tiuryn, and Urzyczyn in the 1990s. The original argument reduces Turing machine immortality via Turing machine boundedness to semi-unification. The latter part is technically most challenging, involving several intermediate models of computation.

This work presents a novel, simpler reduction from Turing machine boundedness to semi-unification. In contrast to the original argument, we directly translate boundedness to solutions of semi-unification and vice versa. In addition, the reduction is mechanized in the Coq proof assistant, relying on a mechanization-friendly stack machine model that corresponds to space-bounded Turing machines. Taking advantage of the simpler proof, the mechanization is comparatively short and fully constructive.

2012 ACM Subject Classification Theory of computation → Computability

Keywords and phrases undecidability, semi-unification, mechanization

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.9

Supplementary Material `https://github.com/uds-psl/2020-fscd-semi-unification`

1 Introduction

In the 1980s it was an actively studied, long-standing open problem whether the combination of first-order unification and matching, both of which are decidable problems, is decidable. This problem, called *semi-unification*, is: given a finite set of pairs (σ, τ) of first-order terms, is there a valuation φ of term variables such that for each pair (σ, τ) we have $\psi(\varphi(\sigma)) = \varphi(\tau)$ for some valuation ψ of term variables?

Semi-unification is directly related [10, 16] to type inference in an extension of the Hindley–Milner type system [11, 19] (cf. the standard **ML** [20] programming language), which allows for polymorphic recursion [21]. Therefore, computational properties of semi-unification translate to type inference capabilities for polymorphic functional programming languages, affecting programming language design. For a broad overview over properties of semi-unification the reader is referred to [17, 13].

In the 1990s Kfoury, Tiuryn, and Urzyczyn have shown that semi-unification is undecidable [15, 17]. This negative result motivated exploration of decidable fragments of semi-unification (for an overview see [18]). The original undecidability proof is quite sophisticated, reflecting the inherent intricacy of the semi-unification problem. It involves Turing machine immortality, symmetric intercell Turing machine boundedness, path equation derivability, and termination of a redex contraction procedure for semi-unification. Therefore, it is challenging to verify the original proof down to the last detail, let alone mechanize it in a proof assistant. Additionally, the original argument uses König's lemma and it is not obvious whether it can be presented constructively.

This work contributes to a better understanding of semi-unification in three aspects. First, we present a simpler proof for the undecidability of semi-unification. The presented technical argument connects an undecidable machine property (in immediate correspondence with Turing machine boundedness) to solutions of semi-unification in a direct way. The key contribution regarding this aspect is the function ζ (Definition 41) that constructs solutions

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 9; pp. 9:1–9:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

for semi-unification instances. Second, we mechanize [2] (ca. 1500 lines of code in the Coq [5] proof assistant) the presented argument, leaving little room for doubt regarding its correctness. Third, König's lemma in the original argument is replaced by the fan theorem. The provided mechanization reveals full constructivity of the remaining reasoning.

Proof Synopsis

First, we reduce Turing machine immortality [12] (is there a diverging configuration?) to a uniform boundedness problem for stack machines (is there a uniform bound on the number of reachable configurations?). The considered, restricted class of stack machines, which we call *simple*, is a mechanization-friendly presentation of space-bounded Turing machines.

Second, given a simple stack machine $\mathcal{M}$, we encode each instruction of $\mathcal{M}$ as a semi-unification constraint, thereby constructing a finite set of constraints $\mathcal{C}$. Each state of $\mathcal{M}$ is a variable in $\mathcal{C}$. The resulting constraints are of restricted shape, which we also call *simple*.

Third, if $\mathcal{M}$ is uniformly bounded, then we interpret configurations of $\mathcal{M}$ as first-order terms using an uncomplicated, computable function ζ. Most importantly, the interpretation of an empty stack configuration in each state of $\mathcal{M}$ is a solution for $\mathcal{C}$.

Fourth, if $\mathcal{C}$ has a solution φ, then we construct a uniform bound for $\mathcal{M}$ from the maximal depth of the syntax trees in the range of φ.

Fifth, the above constitutes an undecidability proof of semi-unification for simple constraints and immediately implies undecidability of semi-unification.

Key aspects of all of the above points, except the third, also appear in [17]. However, the technically most challenging aspect of [17], which we are able to simplify, is to show that a solution for a constructed semi-unification instance exists. Specifically, the function ζ is the main contribution of this work towards a better understanding of semi-unification.

Organization of the Paper

Section 2 contains preliminary properties of simple semi-unification (Problem 15), which is a restriction of semi-unification that transports undecidability (Theorem 1).

Section 3 contains preliminary properties of simple stack machines (Definition 16), which are equivalent to space-bounded Turing machines. Additionally, uniform boundedness of deterministic simple stack machines (Problem 26) is shown undecidable (Theorem 2).

Section 4 contains a reduction from uniform boundedness of deterministic simple stack machines to simple semi-unification. Correctness of the reduction (Lemma 48 and Lemma 45) results in undecidability of semi-unification (Theorem 4).

Section 5 provides an overview over the mechanization [2] of the presented reduction.

Section 6 concludes and lists potential future work.

2 Semi-unification Preliminaries

This section, following [17], recollects the basic definition and properties of semi-unification (Problem 3).

▶ **Definition 1** (Terms ($\mathbb{T}$)). Let $\mathbb{V}$ be a countably infinite set of *variables* ranged over by α, β, γ. The set of *terms* $\mathbb{T}$, ranged over by σ, τ, is given by the grammar

$$\sigma, \tau \in \mathbb{T} ::= \alpha \mid \sigma \to \tau$$

▶ **Definition 2** (Valuation $(\varphi), (\psi)$). A *valuation* $\varphi : \mathbb{V} \to \mathbb{T}$ assigns terms to variables, and is tacitly lifted to terms.

▶ **Problem 3** (Semi-unification (SU))**.** Given a finite set $\{s_1 \leq_1 t_1, \dots, s_n \leq_n t_n\}$ of *indexed inequalities*, do there exist valuations $\varphi, \psi_1, \dots, \psi_n : \mathbb{V} \to \mathbb{T}$ such that $\psi_i(\varphi(s_i)) = \varphi(t_i)$ holds for $i = 1 \dots n$?

Compared to first-order unification, semi-unification is non-structural. In a solvable instance, the left-hand side of an indexed inequality may even appear as subterm of the right-hand side (Example 4).

▶ **Example 4.** The indexed inequalities $\{\alpha \leq_1 \alpha \to \beta, \alpha \to \alpha \leq_2 \beta\}$ are solved by the valuations $\varphi = \{\alpha \mapsto \alpha, \beta \mapsto \alpha \to \alpha\}$, $\psi_1 = \{\alpha \mapsto \alpha \to (\alpha \to \alpha)\}$, and $\psi_2 = \{\alpha \mapsto \alpha\}$ because

$$\begin{aligned} \psi_1(\varphi(\alpha)) &= \alpha \to (\alpha \to \alpha) = \varphi(\alpha \to \beta) \\ \psi_2(\varphi(\alpha \to \alpha)) &= \alpha \to \alpha = \varphi(\beta) \end{aligned}$$

⌟

Next, we introduce the notion of constraints (Definition 6) (called *path equations* in [17]). Constraints play a key role connecting (constraint-based) semi-unification to the execution of a stack machine. Intuitively, a constraint $X \doteq Y$ reflects joinability of configurations X and Y in a stack machine (cf. Section 4).

▶ **Definition 5** (Binary Words ($\mathbb{B}^*$))**.** Let $\mathbb{B} = \{0, 1\}$ be ranged over by a, b. The set $\mathbb{B}^*$ of *words* is ranged over by s, t, v, w.

▶ **Definition 6** (Constraint ($s{\shortmid}\alpha{\shortmid}t \doteq v{\shortmid}\beta{\shortmid}w$))**.** A *constraint* has the shape $s{\shortmid}\alpha{\shortmid}t \doteq v{\shortmid}\beta{\shortmid}w$, where $\alpha, \beta \in \mathbb{V}$ and $s, t, v, w \in \mathbb{B}^*$.

A constraint is *simple* if it has the shape $a{\shortmid}\alpha{\shortmid}\epsilon \doteq \epsilon{\shortmid}\beta{\shortmid}b$, where $\alpha, \beta \in \mathbb{V}$, $a, b \in \mathbb{B}$, and ϵ is the empty word.

In order to connect words with valuations, we define valuation compositions (Definition 7) and path functions on terms (Definition 8).

▶ **Definition 7** (Valuation Composition (ψ_v))**.** Let $\psi_0, \psi_1 : \mathbb{V} \to \mathbb{T}$ be valuations. For a word $v \in \mathbb{B}^*$, the *composed valuation* $\psi_v : \mathbb{T} \to \mathbb{T}$ is such that

$$\psi_\epsilon(\sigma) = \sigma \qquad \psi_{wa}(\sigma) = \psi_w(\psi_a(\sigma))$$

▶ **Definition 8** (Path Function (π_v))**.** For a word $v \in \mathbb{B}^*$, the partial *path function* $\pi_v : \mathbb{T} \nrightarrow \mathbb{T}$ is such that

$$\pi_\epsilon(\sigma) = \sigma \quad \pi_{0w}(\sigma \to \tau) = \pi_w(\sigma) \quad \pi_{1w}(\sigma \to \tau) = \pi_w(\tau) \quad (\text{otherwise } \pi_v(\sigma) \text{ is undefined})$$

Intuitively, a simple constraint $a{\shortmid}\alpha{\shortmid}\epsilon \doteq \epsilon{\shortmid}\beta{\shortmid}b$ is satisfied by a valuation triple $(\varphi, \psi_0, \psi_1)$, if $\psi_a(\varphi(\alpha)) = \pi_b(\varphi(\beta))$. The absence of ψ_0 and ψ_1 on the right-hand side captures matching as part of semi-unification. Similarly to [17], the respective side $s{\shortmid}\alpha{\shortmid}t$ of a constraint is interpreted wrt. a valuation triple $(\varphi, \psi_0, \psi_1)$ by the term which arises when we apply ψ_s to $\varphi(\alpha)$ and then select a subterm via π_t. This interpretation is captured by the following model relation ($\models$).

▶ **Definition 9** (Model Relation ($\models$))**.** A valuation triple $(\varphi, \psi_0, \psi_1)$ *models* a constraint $s{\shortmid}\alpha{\shortmid}t \doteq v{\shortmid}\beta{\shortmid}w$, written $(\varphi, \psi_0, \psi_1) \models s{\shortmid}\alpha{\shortmid}t \doteq v{\shortmid}\beta{\shortmid}w$, if $\pi_t(\psi_s(\varphi(\alpha))) = \pi_w(\psi_v(\varphi(\beta)))$.

For a set $\mathcal{C}$ of constraints, we write $(\varphi, \psi_0, \psi_1) \models \mathcal{C}$ if $(\varphi, \psi_0, \psi_1) \models C$ for all $C \in \mathcal{C}$.

For a set $\mathcal{C}$ of constraints and a constraint C, we write $\mathcal{C} \models C$ if for all valuation triples $(\varphi, \psi_0, \psi_1)$ such that $(\varphi, \psi_0, \psi_1) \models \mathcal{C}$ we have $(\varphi, \psi_0, \psi_1) \models C$.

As a side note, path equation derivability of [17] is sound for ($\models$). The following Example 10, Example 11, and Example 13 illustrate positive and negative cases for models.

▶ **Example 10.** Let $\mathcal{C} = \{0{\shortmid}\alpha{\shortmid}\epsilon \doteq \epsilon{\shortmid}\beta{\shortmid}1, 1{\shortmid}\gamma{\shortmid}\epsilon \doteq \epsilon{\shortmid}\beta{\shortmid}1, 1{\shortmid}\alpha{\shortmid}\epsilon \doteq \epsilon{\shortmid}\gamma{\shortmid}0\}$ be a set of simple constraints. We have $(\varphi, \psi_0, \psi_1) \models \mathcal{C}$, where

$$\begin{aligned}
\varphi &= \{\alpha \mapsto \alpha, \beta \mapsto \beta_0 \to (\beta_{10} \to \beta_{11}), \gamma \mapsto \gamma_0 \to \gamma_1\}\\
\psi_0 &= \{\alpha \mapsto \beta_{10} \to \beta_{11}\}\\
\psi_1 &= \{\alpha \mapsto \gamma_0, \gamma_0 \mapsto \beta_{10}, \gamma_1 \mapsto \beta_{11}\}
\end{aligned}$$

▶ **Example 11.** Let $\mathcal{C} = \{0{\shortmid}\alpha{\shortmid}\epsilon \doteq \epsilon{\shortmid}\beta{\shortmid}1, 1{\shortmid}\gamma{\shortmid}\epsilon \doteq \epsilon{\shortmid}\beta{\shortmid}1, 1{\shortmid}\alpha{\shortmid}\epsilon \doteq \epsilon{\shortmid}\gamma{\shortmid}0\}$ be a set of simple constraints. We have $\mathcal{C} \models 0{\shortmid}\alpha{\shortmid}0 \doteq 11{\shortmid}\alpha{\shortmid}\epsilon$, because for any valuations φ, ψ_0, ψ_1 such that $(\varphi, \psi_0, \psi_1) \models \mathcal{C}$ we have

$$\pi_0(\psi_0(\varphi(\alpha))) = \pi_0(\pi_1(\varphi(\beta))) = \pi_0(\psi_1(\varphi(\gamma))) = \psi_1(\pi_0(\varphi(\gamma))) = \psi_1(\psi_1(\varphi(\alpha)))$$

The *depth* of a term is the maximal depth of its syntax tree, and is non-decreasing under substitution.

▶ **Definition 12** (Term Depth (depth))**.** The function depth : $\mathbb{T} \to \mathbb{N}$ is such that

$$\text{depth}(\alpha) = 0 \qquad \text{depth}(\sigma \to \tau) = 1 + \max\{\text{depth}(\sigma), \text{depth}(\tau)\}$$

▶ **Example 13.** There is no valuation triple $(\varphi, \psi_0, \psi_1)$ that models the simple constraint $1{\shortmid}\alpha{\shortmid}\epsilon \doteq \epsilon{\shortmid}\alpha{\shortmid}0$. Otherwise, we would have

$$\begin{aligned}
& & \pi_\epsilon(\psi_1(\varphi(\alpha))) &= \pi_0(\psi_\epsilon(\varphi(\alpha))) & \\
\Longrightarrow & & \psi_1(\varphi(\alpha)) &= \pi_0(\varphi(\alpha)) & \\
\Longrightarrow & & \psi_1(\sigma \to \tau) &= \sigma & &\text{where } \varphi(\alpha) = \sigma \to \tau\\
\Longrightarrow & & \text{depth}(\psi_1(\sigma \to \tau)) &= \text{depth}(\sigma) & \\
\Longrightarrow & & \text{depth}(\psi_1(\sigma)) &< \text{depth}(\sigma) & &\text{which is a contradiction}
\end{aligned}$$

Intuitively, the simple constraint $1{\shortmid}\alpha{\shortmid}\epsilon \doteq \epsilon{\shortmid}\alpha{\shortmid}0$ corresponds to an unbounded computation that transforms arbitrary many 1s on the left stack to 0s on the right stack (cf. Section 4).

The following Lemma 14 describes in which cases a simple constraint is modeled.

▶ **Lemma 14.** We have $(\varphi, \psi_0, \psi_1) \models a{\shortmid}\alpha{\shortmid}\epsilon \doteq \epsilon{\shortmid}\beta{\shortmid}b$ iff one of the following conditions holds

- $b = 0$ and $\psi_a(\varphi(\alpha)) \to \tau = \varphi(\beta)$ for some term $\tau \in \mathbb{T}$
- $b = 1$ and $\sigma \to \psi_a(\varphi(\alpha)) = \varphi(\beta)$ for some term $\sigma \in \mathbb{T}$

Finally, we identify the following semi-unification problem based on simple constraints. The importance of this restriction is pointed out in [17, Sec. 4], and its undecidability implies the undecidability of semi-unification (Theorem 1). Intuitively, we will use a simple constraint $a{\shortmid}\alpha{\shortmid}\epsilon \doteq \epsilon{\shortmid}\beta{\shortmid}b$ to represent a stack machine transition from state α to state β, removing the symbol a from the left stack and adding the symbol b to the right stack.

▶ **Problem 15** (Simple Semi-unification (SSU))**.** Given a finite set $\mathcal{C}$ of simple constraints, do there exist valuations $\varphi, \psi_0, \psi_1 : \mathbb{V} \to \mathbb{T}$ such that $(\varphi, \psi_0, \psi_1) \models \mathcal{C}$?

▶ **Theorem 1.** If simple semi-unification (Problem 15) is undecidable, then so is semi-unification (Problem 3).

Proof. Let $\mathcal{C} = \{0 \shortmid \alpha_i \shortmid \epsilon \doteq \epsilon \shortmid \beta_i \shortmid b_i \mid i = 1 \dots n\} \cup \{1 \shortmid \alpha_i \shortmid \epsilon \doteq \epsilon \shortmid \beta_i \shortmid b_i \mid i = n+1 \dots m\}$ be a set of simple constraints. We define an instance $\mathcal{D}$ of semi-unification that reflects solvability of $\mathcal{C}$ as follows.

Define $\sigma_i = \begin{cases} \alpha_i \to \gamma_i & \text{if } b_i = 0 \\ \gamma_i \to \alpha_i & \text{if } b_i = 1 \end{cases}$, where γ_i is fresh for $i = 1 \dots m$. Define $\mathcal{D}$ as (for convenience, we start indexing inequalities from 0)

$$\sigma_1 \to \dots \to \sigma_n \leq_0 \beta_1 \to \dots \to \beta_n$$
$$\sigma_{n+1} \to \dots \to \sigma_m \leq_1 \beta_{n+1} \to \dots \to \beta_m$$

First, by Lemma 14, if $\mathcal{D}$ has a solution φ, ψ_0, ψ_1, then $\psi_0(\varphi(\sigma_i)) = \varphi(\beta_i)$ for $i = 1 \dots n$, and $\psi_1(\varphi(\sigma_i)) = \varphi(\beta_i)$ for $i = n+1 \dots m$. Therefore, $(\varphi, \psi_0, \psi_1) \models \mathcal{C}$.

Second, assume $(\varphi, \psi_0, \psi_1) \models \mathcal{C}$. Define $\varphi' : \mathbb{V} \to \mathbb{T}$ such that $\varphi'(\gamma_i) = \gamma_i$ for $i = 1 \dots m$, and otherwise $\varphi'(\alpha) = \varphi(\alpha)$. For $a \in \mathbb{B}$, define $\psi'_a : \mathbb{V} \to \mathbb{T}$ such that $\psi'_a(\gamma_i) = \pi_{(1-b_i)}(\varphi(\beta_i))$ for $i = 1 \dots m$, and otherwise $\psi'_a(\alpha) = \psi_a(\alpha)$. By Lemma 14, $\varphi', \psi'_0, \psi'_1$ solve $\mathcal{D}$. ◀

3 Stack Machine Preliminaries

Instead of working with Turing machines (or symmetric intercell Turing machines of [17]), we use a more convenient computational model of simple stack machines (Definition 16). Intuitively, simple stack machines are a mechanization-friendly presentation of space-bounded Turing machines (cf. proof of Theorem 2).

▶ **Definition 16** (Simple Stack Machine ($\mathcal{M}$)). Let p, q range over a countably infinite set $\mathbb{S}$ of *states*. A *simple stack machine* $\mathcal{M}$ is a finite set of *instructions* of shape either $ap \longrightarrow qb$ or $pa \longrightarrow bq$, where $p, q \in \mathbb{S}$ and $a, b \in \mathbb{B}$.

A *configuration* is a triple $s \shortmid p \shortmid t$, where $p \in \mathbb{S}$ is a state, $s \in \mathbb{B}^*$ is the *left stack*, and $t \in \mathbb{B}^*$ is the *right stack*. The set of all configurations is denoted by $\mathbb{C}$.

The *step relation* $(\longrightarrow_{\mathcal{M}}) \subseteq \mathbb{C} \times \mathbb{C}$ on configurations is given by

- $sa \shortmid p \shortmid t \longrightarrow_{\mathcal{M}} s \shortmid q \shortmid bt$ if $(ap \longrightarrow qb) \in \mathcal{M}$
- $s \shortmid p \shortmid at \longrightarrow_{\mathcal{M}} sb \shortmid q \shortmid t$ if $(pa \longrightarrow bq) \in \mathcal{M}$

The *reachability relation* $(\longrightarrow^*_{\mathcal{M}}) \subseteq \mathbb{C} \times \mathbb{C}$ on configurations is the reflexive, transitive closure of $(\longrightarrow_{\mathcal{M}})$. For brevity, we say *machine* for simple stack machine.

▶ **Example 17.** Consider the machine $\mathcal{M} = \{(1p \longrightarrow p0)\}$, which pops 1s from the left stack and pushes 0s onto the right stack.

We have that from the configuration $X = 1^n \shortmid p \shortmid \epsilon$ the configurations $Y_m = 1^m \shortmid p \shortmid 0^{n-m}$ such that $m \leq n$ are reachable, i.e. $X \longrightarrow^*_{\mathcal{M}} Y_m$ for $m = 0 \dots n$.

▶ **Definition 18** (Deterministic). A machine $\mathcal{M}$ is *deterministic* if for all configurations $X, Y, Z \in \mathbb{C}$ such that $X \longrightarrow_{\mathcal{M}} Y$ and $X \longrightarrow_{\mathcal{M}} Z$ we have $Y = Z$.

▶ **Remark 19.** The step relation for Turing machines is naturally connected to the step relation for simple stack machines as follows. Say a Turing machine reading a symbol a in state x writes a symbol b, transitions into a state y, and moves right. This local behavior is described by the instructions $((x, a)0 \longrightarrow b(y, 0))$ and $((x, a)1 \longrightarrow b(y, 1))$, where $(x, a), (y, 0), (y, 1) \in \mathbb{S}$. The left (resp. right) stack describes the Turing machine tape left (resp. right) of the current head position.

A distinctive machine feature is preservation of total available space under reachability (Lemma 21).

▶ **Definition 20** (Word Length (length)). The function length : $\mathbb{B}^* \to \mathbb{N}$ is such that

$$\text{length}(\epsilon) = 0 \qquad \text{length}(av) = 1 + \text{length}(v)$$

▶ **Lemma 21.** If $s\shortmid p\shortmid t \longrightarrow^*_{\mathcal{M}} v\shortmid q\shortmid w$, then $\text{length}(s) + \text{length}(t) = \text{length}(v) + \text{length}(w)$.

Proof. Instructions preserve the sum of stack lengths. ◀

Since machines operate in bounded space (as opposed to Turing machines that operate on infinite tape), most machine properties, such as reachability (Lemma 22), are decidable. This is most useful for a fully constructive mechanization.

▶ **Lemma 22.** It is decidable, whether for a machine $\mathcal{M}$ and configurations $X, Y \in \mathbb{C}$, we have $X \longrightarrow^*_{\mathcal{M}} Y$.

Proof. By Lemma 21, the number of configurations reachable from X is finite and can be searched exhaustively. ◀

Although boundedness (is for any configuration X the number of configurations reachable from X finite?) is a trivially true machine property, uniform boundedness (Problem 26) is undecidable (Theorem 2).

▶ **Definition 23** (Uniformly Bounded). A machine $\mathcal{M}$ is *uniformly bounded* by a natural number $n \in \mathbb{N}$ if for all configurations $X \in \mathbb{C}$ we have

$$|\{Y \in \mathbb{C} \mid X \longrightarrow^*_{\mathcal{M}} Y\}| \leq n$$

For brevity, we say that $\mathcal{M}$ is uniformly bounded if $\mathcal{M}$ is uniformly bounded by some $n \in \mathbb{N}$.

The following Example 24 illustrates a uniformly bounded machine.

▶ **Example 24.** The machine $\mathcal{M} = \{(0p \longrightarrow q1), (q1 \longrightarrow 1p), (1p \longrightarrow q0), (q0 \longrightarrow 0p)\}$ is (by case analysis) uniformly bounded by $n = 4$. For instance, in case of a configuration $X = sa\shortmid p\shortmid t$, where $a \in \mathbb{B}$ and $s, t \in \mathbb{B}^*$, we have

$$|\{Y \in \mathbb{C} \mid X \longrightarrow^*_{\mathcal{M}} Y\}| = |\{sa\shortmid p\shortmid t, s\shortmid q\shortmid (1-a)t, s(1-a)\shortmid p\shortmid t, s\shortmid q\shortmid at\}| = 4 \leq n \qquad \lrcorner$$

Complementarily, the following Example 25 illustrates a machine that is not uniformly bounded. As will be shown in Section 4, this is because the simple constraint $1\shortmid\alpha\shortmid\epsilon \doteq \epsilon\shortmid\alpha\shortmid 0$ in Example 13 has no model.

▶ **Example 25.** The machine $\mathcal{M} = \{(1p \longrightarrow p0)\}$ from Example 17 is not uniformly bounded, because for any $n \in \mathbb{N}$ and the configuration $X = 1^n\shortmid p\shortmid\epsilon$ we have

$$|\{Y \in \mathbb{C} \mid X \longrightarrow^*_{\mathcal{M}} Y\}| = |\{1^m\shortmid p\shortmid 0^{n-m} \mid 0 \leq m \leq n\}| = n + 1 > n \qquad \lrcorner$$

▶ **Problem 26** (Uniform Boundedness of Deterministic Simple Stack Machines (UBDSSM)). Given a deterministic machine $\mathcal{M}$, is $\mathcal{M}$ is uniformly bounded?

The intuition in the above Remark 19 is used in the following Theorem 2 to connect unbounded simple stack machines to immortal Turing machines.

▶ **Theorem 2.** Uniform boundedness of deterministic simple stack machines (Problem 26) is undecidable.

Proof. Weak truth-table reduction from Turing machine mortality [12]. Let $\mathcal{T}$ be a Turing machine with moving tape over the alphabet $\mathbb{B}$ having states Q and transition function $\delta : Q \times \mathbb{B} \to Q \times \mathbb{B} \times \{L, R\}$. A generalized instantaneous description (GID)[1] of $\mathcal{T}$ is a pair $(x, T) \in Q \times \mathbb{B}^{\mathbb{Z}}$, where x is the current state and T is the current tape content with the currently scanned symbol $T(0)$.

Let $(Q \times \mathbb{B}) \subseteq \mathbb{S}$. Define a simple stack machine $\mathcal{M}$ having as instructions

- $(0(x,a) \longrightarrow (y,0)b)$ and $(1(x,a) \longrightarrow (y,1)b)$ if $\delta(x,a) = (y,b,L)$
- $((x,a)0 \longrightarrow b(y,0))$ and $((x,a)1 \longrightarrow b(y,1))$ if $\delta(x,a) = (y,b,R)$

If $\mathcal{T}$ is deterministic, then so is $\mathcal{M}$. Clearly, any finite number of $\mathcal{T}$-transitions corresponds to $\mathcal{M}$-steps for a large enough starting configuration.

We now show that if we can decide whether $\mathcal{M}$ is uniformly bounded, then we can decide whether $\mathcal{T}$ is immortal, i.e. that $\mathcal{T}$ has a GID which has no terminal successor.

First, assume that $\mathcal{M}$ is uniformly bounded by n. From a GID (x, T) we have that $\mathcal{T}$ cannot scan symbols initially positioned at i such that $i < -n$ or $i > n$. Therefore, $\mathcal{T}$ is immortal iff it loops in space $2n + 1$, which is decidable by exhaustive search.

Second, assume that every GID in $\mathcal{T}$ has a terminal successor. We use the fan theorem (as formulated by [4]) to show that $\mathcal{M}$ is uniformly bounded. Let $B = B_\top \cup B_\perp$, where $B_\top$ is the set of binary words that encode terminating computational histories (finite sequences of GIDs in bounded space) in $\mathcal{T}$, and let $B_\perp$ be the set of binary words that cannot be extended to encode a terminating computational history. Since every GID in $\mathcal{T}$ has a terminal successor, membership in B is decidable and B is a *bar*, i.e. every infinite binary sequence has a finite prefix in B. By the fan theorem, B is a uniform bar, i.e. there exists an $n \in \mathbb{N}$ such that any word in B has a prefix of length at most n that is in B. As a result, encoded terminating computational histories are of length at most n. Therefore, $\mathcal{M}$ is uniformly bounded by n. ◀

▶ **Remark 27.** In the above proof of Theorem 2, we deliberately use the fan theorem instead of König's lemma (used in [17, Corollary 5]). In constructive mathematics, the fan theorem, which is valid in Brouwer's intuitionism, is weaker than König's lemma (cf. [22]), which is valid classically.

▶ **Remark 28.** Peculiarly, for counter machines, as another model of computation, uniform boundedness is decidable (similarly to [14, Thm. 2]), whereas boundedness is not (similarly to [14, Thm. 1]). For simple stack machines it is vice versa.

3.1 Narrow Configurations

Clearly, a configuration from which no configuration with an empty left or right stack is reachable does not fully utilize the space it is provided. Therefore, key to boundedness are configurations that have an empty left or right stack, as such configurations may require additional space to reach further configurations. Extending this thought, in this section we identify a property of configurations, which we call *narrowness* (Definition 34) which plays a pivotal role in the overall argument and is part of the main contribution.

[1] An instantaneous description (ID) requires the tape content to be 0 except for finitely many positions.

We can view machine instructions as a restricted rewriting system. Such a view induces the notion of joinable configurations (Definition 29). For deterministic machines, configuration joinability is an equivalence relation (Lemma 30) with a system of representatives (Definition 31).

▶ **Definition 29** (Joinable $(\sim_{\mathcal{M}})$)**.** Two configurations $X, Y \in \mathbb{C}$ are *joinable* in a machine $\mathcal{M}$, written $X \sim_{\mathcal{M}} Y$, if there exists a configuration Z such that $X \longrightarrow^*_{\mathcal{M}} Z$ and $Y \longrightarrow^*_{\mathcal{M}} Z$.

▶ **Lemma 30.** If a machine $\mathcal{M}$ is deterministic, then $(\sim_{\mathcal{M}})$ is an equivalence (reflexive, symmetric, transitive) relation on configurations.

Proof. Clearly, $(\sim_{\mathcal{M}})$ is reflexive and symmetric. Since $\mathcal{M}$ is deterministic, we have that $(\longrightarrow_{\mathcal{M}})$ is confluent. Therefore, for any configurations X_1, X_2, X_3, Y_1, Y_2 such that $X_1 \longrightarrow^*_{\mathcal{M}} Y_1$, $X_2 \longrightarrow^*_{\mathcal{M}} Y_1$, $X_2 \longrightarrow^*_{\mathcal{M}} Y_2$, and $X_3 \longrightarrow^*_{\mathcal{M}} Y_2$ there exists a configuration Z such that $X_1 \longrightarrow^*_{\mathcal{M}} Y_1 \longrightarrow^*_{\mathcal{M}} Z$ and $X_3 \longrightarrow^*_{\mathcal{M}} Y_2 \longrightarrow^*_{\mathcal{M}} Z$. Therefore, $(\sim_{\mathcal{M}})$ is transitive. ◀

▶ **Definition 31** (Representative $([X]_{\mathcal{M}})$)**.** The *representative* of a configuration $X \in \mathbb{C}$ in a deterministic machine $\mathcal{M}$, written $[X]_{\mathcal{M}}$, is the lexicographically smallest configuration Y such that $X \sim_{\mathcal{M}} Y$.

▶ **Lemma 32.** For configurations $X, Y \in \mathbb{C}$, we have $[X]_{\mathcal{M}} = [Y]_{\mathcal{M}}$ iff $X \sim_{\mathcal{M}} Y$.

▶ **Remark 33.** By Lemma 21 and Lemma 22 the representative $[X]_{\mathcal{M}}$ of a configuration X in $\mathcal{M}$ is computable, and joinability $(\sim_{\mathcal{M}})$ is decidable.

Next, we identify a key property (Definition 34) of configurations, that connects machine computation with semi-unification (cf. Section 4).

▶ **Definition 34** (Narrow)**.** A configuration X is *narrow* in a machine $\mathcal{M}$, if there exists a state $p \in \mathbb{S}$ and a word $s \in \mathbb{B}^*$ such that $X \sim_{\mathcal{M}} s|p|\epsilon$.

▶ **Remark 35.** For a state $p \in \mathbb{S}$, the configuration $\epsilon|p|\epsilon$ is narrow in any machine $\mathcal{M}$.

▶ **Remark 36.** Similarly to Lemma 22, it is decidable, whether for a machine $\mathcal{M}$ and configuration $X \in \mathbb{C}$, we have that X is narrow in $\mathcal{M}$.

▶ **Example 37.** In the machine $\mathcal{M} = \{(p1 \longrightarrow 0r), (1q \longrightarrow r1)\}$ the configuration $0|p|11$ is narrow because $0|p|11 \longrightarrow^*_{\mathcal{M}} 00|r|1 \longleftarrow^*_{\mathcal{M}} 001|q|\epsilon$, that is we have $0|p|11 \sim_{\mathcal{M}} 001|q|\epsilon$.

Narrow configurations play a pivotal role for uniform boundedness (Lemma 38 and Lemma 39). Additionally, narrowness is *the* decisive property which we use to construct solutions for semi-unification instances (Definition 41 and Definition 42).

▶ **Lemma 38.** If a machine $\mathcal{M}$ is uniformly bounded, then there exists $m \in \mathbb{N}$ such that for all narrow in $\mathcal{M}$ configurations $s|p|t \in \mathbb{C}$ we have $\text{length}(t) \leq m$.

Proof. If $s|p|t$ is narrow in $\mathcal{M}$, then there are a configurations $s'|p'|\epsilon$ and $v|q|w$ such that $s|p|t \longrightarrow^*_{\mathcal{M}} v|q|w$ and $s'|p'|\epsilon \longrightarrow^*_{\mathcal{M}} v|q|w$. If $\mathcal{M}$ is uniformly bounded by n, then we have $|\,\text{length}(t) - \text{length}(w)| \leq n$ and $|\,\text{length}(\epsilon) - \text{length}(w)| \leq n$. Therefore, $\text{length}(t) \leq 2n$. ◀

▶ **Lemma 39.** Let $\mathcal{M}$ be a deterministic machine. If there exists $m \in \mathbb{N}$ such that for all narrow in $\mathcal{M}$ configurations $\epsilon\shortmid p\shortmid t \in \mathbb{C}$ we have $\text{length}(t) \leq m$, then $\mathcal{M}$ is uniformly bounded.

Proof. Let $m \in \mathbb{N}$ be such that for all narrow in $\mathcal{M}$ configurations $\epsilon\shortmid p\shortmid t \in \mathbb{C}$ we have $\text{length}(t) \leq m$. Let $n \in \mathbb{N}$ and let $X = s\shortmid p\shortmid t$ reach at least n configurations such that $(\text{length}(s) + \text{length}(t))$ is minimal. We show that $\mathcal{M}$ is uniformly bounded by showing

$$n \leq 1 + |\{s'\shortmid p'\shortmid t' \in \mathbb{C} \mid \text{length}(s') + \text{length}(t') \leq m \text{ and } p' \text{ occurs in } \mathcal{M}\}| \qquad (\star)$$

We have $X \longrightarrow^*_{\mathcal{M}} \epsilon\shortmid q\shortmid w$ for some state $q \in \mathbb{S}$ and word $w \in \mathbb{B}^*$. Otherwise, left stacks of all configurations reachable from X would have the same prefix, which could be removed. Similarly, we have $X \longrightarrow^*_{\mathcal{M}} v\shortmid r\shortmid \epsilon$ for some state $r \in \mathbb{S}$ and word $v \in \mathbb{B}^*$. Since $\mathcal{M}$ is deterministic, $(\longrightarrow_{\mathcal{M}})$ is confluent. Therefore, the configuration $\epsilon\shortmid q\shortmid w$ is narrow in $\mathcal{M}$.

Finally, by Lemma 21, for any configuration $s'\shortmid p'\shortmid t'$ such that $X \longrightarrow^*_{\mathcal{M}} s'\shortmid p'\shortmid t'$ we have $\text{length}(s') + \text{length}(t') = \text{length}(s) + \text{length}(t) = \text{length}(w) \leq m$, showing $(\star)$. ◀

4 Undecidability of Semi-unification

In this section we fix a deterministic machine $\mathcal{M}$. Our goal is to construct a *specific instance* $\mathcal{C}_{\mathcal{M}}$ (Definition 40) of simple semi-unification such that the machine $\mathcal{M}$ is uniformly bounded if (Lemma 48) and only if (Lemma 45) $\mathcal{C}_{\mathcal{M}}$ is solvable.

For brevity, we omit $\mathcal{M}$ in notations in this section, i.e. we write $(\sim)$ for $(\sim_{\mathcal{M}})$, write $\mathcal{C}$ for $\mathcal{C}_{\mathcal{M}}$, say narrow for narrow in $\mathcal{M}$, etc. All definitions in this section tacitly depend on $\mathcal{M}$.

Let us tacitly inject $\mathbb{S}$ into $\mathbb{V}$, i.e. $\mathbb{S} \subseteq \mathbb{V}$. Additionally, for each configuration $X \in \mathbb{C}$ we fix a distinct variable $\alpha_X \in \mathbb{V}$.

▶ **Definition 40** (Specific instance $\mathcal{C}$)**.** The set $\mathcal{C}$ of simple constraints is given by

$$\mathcal{C} = \{a\shortmid p\shortmid \epsilon \doteq \epsilon\shortmid q\shortmid b \mid (ap \longrightarrow qb) \in \mathcal{M}\} \cup \{b\shortmid q\shortmid \epsilon \doteq \epsilon\shortmid p\shortmid a \mid (pa \longrightarrow bq) \in \mathcal{M}\}$$

4.1 Uniform Boundedness of $\mathcal{M}$ to Solvability of $\mathcal{C}$

In this subsection we assume that $\mathcal{M}$ is uniformly bounded and construct a solution φ, ψ_0, ψ_1 (Definition 42) for $\mathcal{C}$. Surprisingly, this can be done directly via the following function ζ (Definition 41), based on the notion of narrow configurations (Definition 34).

▶ **Definition 41** (ζ)**.** If $\mathcal{M}$ is uniformly bounded, then the function $\zeta : \mathbb{C} \to \mathbb{T}$ is given by

$$\zeta(s\shortmid p\shortmid t) = \begin{cases} \zeta(s\shortmid p\shortmid t0) \to \zeta(s\shortmid p\shortmid t1) & \text{if } s\shortmid p\shortmid t \text{ is narrow} \\ \alpha_{[s\shortmid p\shortmid t]} & \text{otherwise} \end{cases}$$

By Lemma 38, ζ is well-defined and computable (cf. Remark 36 and Remark 33). Computability of ζ is essential for a fully constructive argument.

▶ **Definition 42** (Valuations φ, ψ_0, ψ_1)**.** The valuation $\varphi : \mathbb{V} \to \mathbb{T}$ is such that

$$\varphi(p) = \zeta(\epsilon\shortmid p\shortmid \epsilon) \qquad (\text{otherwise } \varphi(\alpha) = \alpha)$$

For $a \in \mathbb{B}$, the valuation $\psi_a : \mathbb{V} \to \mathbb{T}$ is such that

$$\psi_a(\alpha_{s\shortmid p\shortmid t}) = \zeta(as\shortmid p\shortmid t) \qquad (\text{otherwise } \psi_a(\alpha) = \alpha)$$

The function ζ respects joinability (Lemma 43), i.e. it can be lifted to $(\sim)$ equivalence classes.

▶ **Lemma 43.** For configurations $X, Y \in \mathbb{C}$ such that $X \sim Y$ we have $\zeta(X) = \zeta(Y)$.

Proof. We show $\zeta(s \shortmid p \shortmid t) = \zeta(v \shortmid q \shortmid w)$ by induction on $\mathrm{depth}(\zeta(s \shortmid p \shortmid t))$.

Case $s \shortmid p \shortmid t$ is narrow: By Lemma 30, the configuration $v \shortmid q \shortmid w$ is narrow. Therefore,

$$\zeta(s \shortmid p \shortmid t) = \zeta(s \shortmid p \shortmid t0) \to \zeta(s \shortmid p \shortmid t1) \overset{\text{(IH)}}{=} \zeta(v \shortmid q \shortmid w0) \to \zeta(v \shortmid q \shortmid w1) = \zeta(v \shortmid q \shortmid w)$$

Case $s \shortmid p \shortmid t$ is not narrow: By Lemma 30, the configuration $v \shortmid q \shortmid w$ is not narrow. Therefore,

$$\zeta(s \shortmid p \shortmid t) = \alpha_{[s \shortmid p \shortmid t]} \overset{\text{Lem. 32}}{=} \alpha_{[v \shortmid q \shortmid w]} = \zeta(v \shortmid q \shortmid w)$$ ◀

Since the function ζ respects joinability, it absorbs ψ_0 and ψ_1 (Lemma 44).

▶ **Lemma 44.** For $a \in \mathbb{B}$ and configuration $s \shortmid p \shortmid t \in \mathbb{C}$, we have $\psi_a(\zeta(s \shortmid p \shortmid t)) = \zeta(as \shortmid p \shortmid t)$.

Proof. We show $\psi_a(\zeta(s \shortmid p \shortmid t)) = \zeta(as \shortmid p \shortmid t)$ by induction on $\mathrm{depth}(\zeta(s \shortmid p \shortmid t))$.

Case $s \shortmid p \shortmid t$ is narrow: We have that $as \shortmid p \shortmid t$ is narrow, and

$$\begin{aligned}\psi_a(\zeta(s \shortmid p \shortmid t)) &= \psi_a(\zeta(s \shortmid p \shortmid t0) \to \zeta(s \shortmid p \shortmid t1)) = \psi_a(\zeta(s \shortmid p \shortmid t0)) \to \psi_a(\zeta(s \shortmid p \shortmid t1)) \\ &\overset{\text{(IH)}}{=} \zeta(as \shortmid p \shortmid t0) \to \zeta(as \shortmid p \shortmid t1) = \zeta(as \shortmid p \shortmid t)\end{aligned}$$

Case $s \shortmid p \shortmid t$ is not narrow: Let $v \shortmid q \shortmid w = [s \shortmid p \shortmid t]$. We have

$$\psi_a(\zeta(s \shortmid p \shortmid t)) = \psi_a(\alpha_{[s \shortmid p \shortmid t]}) = \zeta(av \shortmid q \shortmid w) \overset{\text{Lem. 43}}{=} \zeta(as \shortmid p \shortmid t)$$ ◀

As a result, the valuations φ, ψ_0, ψ_1 solve $\mathcal{C}$ (Lemma 45).

▶ **Lemma 45.** If $\mathcal{M}$ is uniformly bounded, then $(\varphi, \psi_0, \psi_1) \models \mathcal{C}$.

Proof. Configuration where both stacks are empty are trivially narrow (Remark 35).

Case $a \shortmid p \shortmid \epsilon \doteq \epsilon \shortmid q \shortmid b \in \mathcal{C}$: We have $(ap \longrightarrow qb) \in \mathcal{M}$, therefore $a \shortmid p \shortmid \epsilon \sim \epsilon \shortmid q \shortmid b$. We have

$$\psi_a(\varphi(p)) = \psi_a(\zeta(\epsilon \shortmid p \shortmid \epsilon)) \overset{\text{Lem. 44}}{=} \zeta(a \shortmid p \shortmid \epsilon) \overset{\text{Lem. 43}}{=} \zeta(\epsilon \shortmid q \shortmid b) = \pi_b(\zeta(\epsilon \shortmid q \shortmid \epsilon)) = \pi_b(\varphi(q))$$

Case $b \shortmid q \shortmid \epsilon \doteq \epsilon \shortmid p \shortmid a \in \mathcal{C}$: We have $(pa \longrightarrow bq) \in \mathcal{M}$, therefore $b \shortmid q \shortmid \epsilon \sim \epsilon \shortmid p \shortmid a$. We have

$$\psi_b(\varphi(q)) = \psi_b(\zeta(\epsilon \shortmid q \shortmid \epsilon)) \overset{\text{Lem. 44}}{=} \zeta(b \shortmid q \shortmid \epsilon) \overset{\text{Lem. 43}}{=} \zeta(\epsilon \shortmid p \shortmid a) = \pi_a(\zeta(\epsilon \shortmid p \shortmid \epsilon)) = \pi_a(\varphi(p))$$ ◀

Essentially, the function ζ interprets machine configurations as terms from which the solution $(\varphi, \psi_0, \psi_1)$ of $\mathcal{C}$ is constructed. Traditionally, this step in the overall argument [17] relies on on a more complicated path equation derivability and termination of a redex contraction procedure for semi-unification. Arguably, the function ζ is the main insight of this work, as it contributes to a simpler, fully constructive translation of machine boundedness to solvability of semi-unification.

4.2 Solvability of $\mathcal{C}$ to Uniform Boundedness of $\mathcal{M}$

In this subsection we assume that there exist valuations φ, ψ_0, ψ_1 such that $(\varphi, \psi_0, \psi_1) \models \mathcal{C}$, and we show that $\mathcal{M}$ is uniformly bounded.

Intuitively, we show that joinability is sound for constraint semantics (Corollary 47) based on soundness of the step relation for constraint semantics (Lemma 46).

▶ **Lemma 46.** For configurations $X, Y \in \mathbb{C}$ such that $X \longrightarrow Y$ we have $\mathcal{C} \models X \doteq Y$.

Proof. Let φ, ψ_0, ψ_1 be valuations such that $(\varphi, \psi_0, \psi_1) \models \mathcal{C}$.
Case $sa|p|t \longrightarrow s|q|bt$**:** We have $a|p|\epsilon \doteq \epsilon|q|b \in \mathcal{C}$. Therefore, $\psi_a(\varphi(p)) = \pi_b(\varphi(q))$ and

$$\pi_t(\psi_{sa}(\varphi(p))) = \pi_t(\psi_s(\psi_a(\varphi(p)))) = \pi_t(\psi_s(\pi_b(\varphi(q)))) = \pi_{bt}(\psi_s(\varphi(q)))$$

Case $s|p|at \longrightarrow_{\mathcal{M}} sb|q|t$**:** We have $b|q|\epsilon \doteq \epsilon|p|a \in \mathcal{C}$. Therefore, $\psi_b(\varphi(q)) = \pi_a(\varphi(p))$ and

$$\pi_{at}(\psi_s(\varphi(p))) = \pi_t(\psi_s(\pi_a(\varphi(p)))) = \pi_t(\psi_s(\psi_b(\varphi(q)))) = \pi_t(\psi_{sb}(\varphi(q)))$$ ◀

▶ **Corollary 47.** For configurations $X, Y \in \mathbb{C}$ such that $X \sim Y$ we have $\mathcal{C} \models X \doteq Y$.

As a result, narrow configurations $\epsilon|p|t$ do not admit arbitrary long right stacks t, because $\pi_t(\varphi(p))$ is undefined if length(t) exceeds depth$(\varphi(p))$. The bound on depth for the range of φ immediately induces a uniform bound for $\mathcal{M}$ (Lemma 48).

▶ **Lemma 48.** If there exist valuations φ, ψ_0, ψ_1 such that $(\varphi, \psi_0, \psi_1) \models \mathcal{C}$, then $\mathcal{M}$ is uniformly bounded.

Proof. Let $\epsilon|p|t \in \mathbb{C}$ be narrow, i.e. $\epsilon|p|t \sim s|q|\epsilon$ for some state $q \in \mathbb{S}$ and word $s \in \mathbb{B}^*$. By Corollary 47, we have $\pi_t(\varphi(p)) = \psi_s(\varphi(q)) \in \mathbb{T}$. Therefore,

$$\text{length}(t) \leq \max\{\text{depth}(\varphi(r)) \mid r \in \mathbb{S} \text{ and } r \text{ occurs in } \mathcal{M}\}$$

By Lemma 39, $\mathcal{M}$ is uniformly bounded. ◀

Key to the construction of a uniform bound in the above proof is the characterization of uniform boundedness via narrow configurations (Lemma 39).

4.3 Main Result

Overall, we obtain undecidability of semi-unification (Theorem 4) via undecidability of simple semi-unification (Theorem 3).

▶ **Theorem 3.** Simple semi-unification (Problem 15) is undecidable.

Proof. By Theorem 2, uniform boundedness of deterministic machines (UBDSSM) is undecidable. Section 4 gives a reduction from UBDSSM to simple semi-unification, for which correctness is shown by Lemma 45 and Lemma 48. ◀

▶ **Theorem 4.** Semi-unification is undecidable.

Proof. By Theorem 3 and Theorem 1. ◀

Let us illustrate the construction, revisiting the uniformly bounded machine of Example 24.

▶ **Example 49.** Let $\mathcal{M} = \{(0p \longrightarrow q1), (q1 \longrightarrow 1p), (1p \longrightarrow q0), (q0 \longrightarrow 0p)\}$. Then, $\mathcal{C} = \{0{\shortmid}p{\shortmid}\epsilon \doteq \epsilon{\shortmid}q{\shortmid}1, 1{\shortmid}p{\shortmid}\epsilon \doteq \epsilon{\shortmid}q{\shortmid}1, 1{\shortmid}p{\shortmid}\epsilon \doteq \epsilon{\shortmid}q{\shortmid}0, 0{\shortmid}p{\shortmid}\epsilon \doteq \epsilon{\shortmid}q{\shortmid}0\}$. Narrow in $\mathcal{M}$ configurations are $s{\shortmid}r{\shortmid}\epsilon$ for words $s \in \mathbb{B}^*$ and states $r \in \mathbb{S}$, and $s{\shortmid}q{\shortmid}a$ for words $s \in \mathbb{B}^*$ and symbols $a \in \mathbb{B}$. Therefore (not writing out representatives), we have

$$\begin{aligned}
\varphi(p) &= \zeta(\epsilon{\shortmid}p{\shortmid}\epsilon) = \zeta(\epsilon{\shortmid}p{\shortmid}0) \to \zeta(\epsilon{\shortmid}p{\shortmid}1) = \alpha_{[\epsilon{\shortmid}p{\shortmid}0]} \to \alpha_{[\epsilon{\shortmid}p{\shortmid}1]}\\
\varphi(q) &= \zeta(\epsilon{\shortmid}q{\shortmid}\epsilon) = \zeta(\epsilon{\shortmid}q{\shortmid}0) \to \zeta(\epsilon{\shortmid}q{\shortmid}1)\\
&= \big(\zeta(\epsilon{\shortmid}q{\shortmid}00) \to \zeta(\epsilon{\shortmid}q{\shortmid}01)\big) \to \big(\zeta(\epsilon{\shortmid}q{\shortmid}10) \to \zeta(\epsilon{\shortmid}q{\shortmid}11)\big)\\
&= (\alpha_{[\epsilon{\shortmid}q{\shortmid}00]} \to \alpha_{[\epsilon{\shortmid}q{\shortmid}01]}) \to (\alpha_{[\epsilon{\shortmid}q{\shortmid}10]} \to \alpha_{[\epsilon{\shortmid}q{\shortmid}11]})\\
\psi_a(\alpha_{\epsilon{\shortmid}p{\shortmid}b}) &= \zeta(a{\shortmid}p{\shortmid}b) = \alpha_{[a{\shortmid}p{\shortmid}b]} && \text{for } a, b \in \mathbb{B}
\end{aligned}$$

Overall, the valuations φ, ψ_0, ψ_1 model $\mathcal{C}$, i.e. $(\varphi, \psi_0, \psi_1) \models \mathcal{C}$. For example, we have $(\varphi, \psi_0, \psi_1) \models 0{\shortmid}p{\shortmid}\epsilon \doteq \epsilon{\shortmid}q{\shortmid}1$ because $0{\shortmid}p{\shortmid}1 \sim \epsilon{\shortmid}q{\shortmid}11$ and $0{\shortmid}p{\shortmid}1 \sim \epsilon{\shortmid}q{\shortmid}11$ imply

$$\begin{aligned}
\psi_0(\varphi(p)) &= \psi_0(\alpha_{[\epsilon{\shortmid}p{\shortmid}0]} \to \alpha_{[\epsilon{\shortmid}p{\shortmid}1]}) = \psi_0(\alpha_{\epsilon{\shortmid}p{\shortmid}0} \to \alpha_{\epsilon{\shortmid}p{\shortmid}1}) = \alpha_{[0{\shortmid}p{\shortmid}0]} \to \alpha_{[0{\shortmid}p{\shortmid}1]}\\
&= \alpha_{[\epsilon{\shortmid}q{\shortmid}10]} \to \alpha_{[\epsilon{\shortmid}q{\shortmid}11]} = \pi_1(\varphi(q))
\end{aligned}$$

5 Mechanization

This section provides an overview over the mechanization [2] in the Coq proof assistant of the reduction presented in Section 4.

The mechanization can be considered self-contained code supporting the mathematical argument and its constructivity. In addition, it is compatible with the framework of *synthetic undecidability results* [9, 8, 7] in *synthetic computability theory* [3].

5.1 Semi-unification

Terms (Definition 1) are mechanized in `SemiU/SemiU_prelim.v` as the inductive type

```
Inductive term : Set :=
  | atom : nat -> term
  | arr : term -> term -> term.
```

Correspondingly, application of valuations is mechanized as

```
Definition valuation : Set := nat -> term.

Fixpoint substitute (f: valuation) (t: term) : term :=
  match t with
  | atom n => f n
  | arr s t => arr (substitute f s) (substitute f t)
  end.
```

Solvability of semi-unification inequalities is mechanized as

```
Definition inequality : Set := (term * term).

Definition solution (φ : valuation) : inequality -> Prop :=
  fun '(s, t) => exists (ψ : valuation),
    substitute ψ (substitute φ s) = substitute φ t.
```

Correspondingly, semi-unification is mechanized in `SemiU/SemiU.v` as the predicate

```
Definition SemiU (p: list inequality) := exists (φ: valuation),
  forall (c: inequality), In c p -> solution φ c.
```

5.2 Simple Stack Machines

Machines (`ssm`) are mechanized in `SM/SSM_prelim.v` as lists of instructions.

```
Definition stack : Set := list bool.
Definition state : Set := nat.
Definition config : Set := stack * state * stack.
Definition dir : Set := bool.
Definition symbol : Set := bool.
Definition instruction : Set := state * state * symbol * symbol * dir.
Definition ssm : Set := list instruction.
```

For example, `(p, q, a, b, true) : instruction` corresponds to the instruction $(ap \longrightarrow qb)$, and `(p, q, b, a, false) : instruction` corresponds to the instruction $(pb \longrightarrow aq)$. This is captured by the inductive predicate `Inductive step (M : ssm) : config -> config -> Prop`, that mechanizes the step relation.

Deterministic machines (`dssm`) admit only functional step predicates and reachability (`reachable`) is the reflexive, transitive closure of `step`.

```
Definition deterministic (M: ssm) := forall (X Y Z: config),
  step M X Y -> step M X Z -> Y = Z.

Definition dssm := { M : ssm | deterministic M }.

Definition reachable (M: ssm) : config -> config -> Prop :=
  clos_refl_trans config (step M).
```

Uniform boundedness (`bounded`) of deterministic machines (`dssm`) is mechanized in `SM/DSSM_UB.v` as the predicate `DSSM_UB`.

```
Definition bounded (M: ssm) (n: nat) : Prop :=
  forall (X: config), exists (L: list config),
    (forall (Y: config), reachable M X Y -> In Y L) /\ length L <= n.

Definition DSSM_UB (M: dssm) := exists (n: nat), bounded (proj1_sig M) n.
```

5.3 Main Result

Many-one reducibility ($\preceq$) of a predicate `p : X -> Prop` to a predicate `q : Y -> Prop` is mechanized in `Reduction.v` as

```
Definition reduces X Y (p : X -> Prop) (q : Y -> Prop) :=
  exists f : X -> Y, forall x, p x <-> q (f x).
Notation "p ⪯ q" := (reduces p q) (at level 50).
```

The main result is mechanized in `SemiU/DSSM_UB_to_SemiU.v` as

```
Theorem DSSM_UB_to_SemiU : DSSM_UB ⪯ SemiU.
Proof.
  apply (reduces_transitive DSSM_UB_to_SSemiU).
  exact SSemiU_to_SemiU.
Qed.
```

The above shows that we first reduce `DSSM_UB` to simple semi-unification (mechanized in `SemiU/SSemiU.v` as the predicate `SSemiU`) and then reduce `SSemiU` to `SemiU`. Mechanization details of `DSSM_UB_to_SSemiU` are found in `SemiU/SSemiU/DSSM_UB_to_SSemiU_argument.v`.

Informative decidability of narrowness is mechanized in `DSM/DSSM/DSSM_facts.v` as

```
Lemma narrow_dec (X: config) : decidable (narrow X).
```

Based on decidability of narrowness, the key function ζ (Definition 41) is mechanized as

```
Fixpoint ζ (n: nat) (X: config) : term :=
  match n with
  | 0 => atom (embed (nf X))
  | S n =>
      match X with
      | (A, x, B) =>
        if narrow_dec (A, x, B) then
          arr (ζ n (A, x, B++[false])) (ζ n (A, x, B++[true]))
        else atom (embed (nf X))
      end
  end.
```

where `nf X` mechanizes the representative of the mechanized configuration `X` (Definition 31). The parameter `n` is initialized with a uniform bound of the underlying machine.

Finally, Lemma 45 and Lemma 48 are mechanized as

```
Lemma soundness {M: dssm} :
  DSSM_UB M -> SSemiU (SM_to_SUcs (proj1_sig M)).

Lemma completeness {M: dssm}:
  SSemiU (SM_to_SUcs (proj1_sig M)) -> DSSM_UB M.
```

Overall, the mechanization encompasses 1500 lines of code, where two thirds show machine properties (such as decidability of narrowness) and one third is dedicated to the main argument of Section 4.

6 Conclusion

Traditionally, the association of an undecidable property for Turing machines with solvability of semi-unification is, arguably, opaque. It is established via the symmetric closure of intercell Turing machines, path equation derivability, and termination of a redex contraction procedure for semi-unification [17]. The main novelty of the presented approach is the direct association of an undecidable boundedness property with solutions of semi-unification via certain (narrow) machine configurations. As a consequence, we obtain a simpler argument for the undecidability of semi-unification. Additionally, this allows for a fully constructive mechanization of a reduction from uniform boundedness of deterministic simple stack machines (Problem 26) to semi-unification (Problem 3).

There are at least two reasonable goals to pursue next.

First, there exists a larger Coq framework [9] containing various undecidability results. The mechanization presented in Section 5 is a significant part of the ongoing effort to mechanize a reduction from the Turing machine halting problem to semi-unification. It is unclear whether a comprehensive reduction can be given fully constructively, as the presented mechanization starts with uniform boundedness. The reduction from the Turing machine halting problem (as of now) requires the fan theorem (which is part of Brouwer's constructivism, but is not considered fully constructive by Bishop). Nevertheless, it is an improvement over König's lemma used in [17]. There is reason to believe, that eliminating immortality as an intermediate step may allow for a fully constructive reduction. This is why the mechanization in Section 5 starts with boundedness as opposed to immortality.

Second, related work on semi-unification mostly follows the original approach (e.g. [1, 6]). We anticipate that the more direct argument, presented in this work, can be adapted to the related scenarios. Specifically, the presented approach seems promising to realize a fully constructive mechanization of the undecidability of unification modulo synchronous distributivity [1].

References

1 Siva Anantharaman, Serdar Erbatur, Christopher Lynch, Paliath Narendran, and Michaël Rusinowitch. Unification modulo synchronous distributivity. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 14–29. Springer, 2012. doi:10.1007/978-3-642-31365-3_4.

2 Andrej Dudenhefner. Mechanization of a Reduction from Uniform Boundedness of Deterministic Simple Stack Machines to Semi-unification. https://github.com/uds-psl/2020-fscd-semi-unification. Accessed: 2019-12-19.

3 Andrej Bauer. First steps in synthetic computability theory. *Electr. Notes Theor. Comput. Sci.*, 155:5–31, 2006. doi:10.1016/j.entcs.2005.11.049.

4 Josef Berger. The fan theorem and uniform continuity. In S. Barry Cooper, Benedikt Löwe, and Leen Torenvliet, editors, *New Computational Paradigms, First Conference on Computability in Europe, CiE 2005, Amsterdam, The Netherlands, June 8-12, 2005, Proceedings*, volume 3526 of *Lecture Notes in Computer Science*, pages 18–22. Springer, 2005. doi:10.1007/11494645_3.

5 The Coq Proof Assistant. https://coq.inria.fr/. Accessed: 2019-12-19.

6 Lucılia Figueiredo and Carlos Camarao. Semi-unification and periodicity of turing machines. URL: https://www.researchgate.net/publication/268426611_Semi-unification_and_Periodicity_of_Turing_Machines.

7 Yannick Forster, Edith Heiter, and Gert Smolka. Verification of PCP-related computational reductions in Coq. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, pages 253–269, 2018. doi:10.1007/978-3-319-94821-8_15.

8 Yannick Forster, Dominik Kirst, and Gert Smolka. On synthetic undecidability in Coq, with an application to the entscheidungsproblem. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 38–51. ACM, 2019. doi:10.1145/3293880.3294091.

9 Yannick Forster, Dominique Larchey-Wendling, Andrej Dudenhefner, Edith Heiter, Dominik Kirst, Fabian Kunze, Gert Smolka, Simon Spies, Dominik Wehr, and Maximilian Wuttke. A Coq Library of Undecidable Problems. In *The Sixth International Workshop on Coq for Programming Languages (CoqPL 2020)*, 2020. URL: https://github.com/uds-psl/coq-library-undecidability.

10 Fritz Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, 1993. doi:10.1145/169701.169692.

11 Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

12 Philip K. Hooper. The Undecidability of the Turing Machine Immortality Problem. *J. Symb. Log.*, 31(2):219–234, 1966. doi:10.2307/2269811.

13 Said Jahama and Assaf J. Kfoury. A general theory of semi-unification. Technical report, Boston University Computer Science Department, 1993.

14 Jarkko Kari and Nicolas Ollinger. Periodicity and immortality in reversible computing. In Edward Ochmanski and Jerzy Tyszkiewicz, editors, *Mathematical Foundations of Computer Science 2008, 33rd International Symposium, MFCS 2008, Torun, Poland, August 25-29, 2008, Proceedings*, volume 5162 of *Lecture Notes in Computer Science*, pages 419–430. Springer, 2008. doi:10.1007/978-3-540-85238-4_34.

15 Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. The Undecidability of the Semi-Unification Problem (Preliminary Report). In Harriet Ortiz, editor, *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 468–476. ACM, 1990. doi:10.1145/100216.100279.

16 Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):290–311, 1993. doi:10.1145/169701.169687.

17 Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. The undecidability of the semi-unification problem. *Inf. Comput.*, 102(1):83–101, 1993. doi:10.1006/inco.1993.1003.

18 Hans Leiß and Fritz Henglein. A decidable case of the semi-unification problem. In Andrzej Tarlecki, editor, *Mathematical Foundations of Computer Science 1991, 16th International Symposium, MFCS'91, Kazimierz Dolny, Poland, September 9-13, 1991, Proceedings*, volume 520 of *Lecture Notes in Computer Science*, pages 318–327. Springer, 1991. doi:10.1007/3-540-54345-7_75.

19 Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978. doi:10.1016/0022-0000(78)90014-4.

20 Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML*. MIT Press, 1990.

21 Alan Mycroft. Polymorphic type schemes and recursive definitions. In Manfred Paul and Bernard Robinet, editors, *International Symposium on Programming, 6th Colloquium, Toulouse, France, April 17-19, 1984, Proceedings*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer, 1984. doi:10.1007/3-540-12925-1_41.

22 Helmut Schwichtenberg. A direct proof of the equivalence between brouwer's fan theorem and könig's lemma with a uniqueness hypothesis. *J. UCS*, 11(12):2086–2095, 2005. doi:10.3217/jucs-011-12-2086.

Conditional Bisimilarity for Reactive Systems

Mathias Hülsbusch
Universität Duisburg-Essen, Germany

Barbara König
Universität Duisburg-Essen, Germany
barbara_koenig@uni-due.de

Sebastian Küpper
FernUniversität in Hagen, Germany
sebastian.kuepper@fernuni-hagen.de

Lara Stoltenow
Universität Duisburg-Essen, Germany
lara.stoltenow@uni-due.de

Abstract

Reactive systems à la Leifer and Milner, an abstract categorical framework for rewriting, provide a suitable framework for deriving bisimulation congruences. This is done by synthesizing interactions with the environment in order to obtain a compositional semantics.

We enrich the notion of reactive systems by conditions on two levels: first, as in earlier work, we consider rules enriched with application conditions and second, we investigate the notion of conditional bisimilarity. Conditional bisimilarity allows us to say that two system states are bisimilar provided that the environment satisfies a given condition. We present several equivalent definitions of conditional bisimilarity, including one that is useful for concrete proofs and that employs an up-to-context technique, and we compare with related behavioural equivalences. We instantiate reactive systems in order to obtain DPO graph rewriting and consider a case study in this setting.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Program reasoning

Keywords and phrases conditional bisimilarity, reactive systems, up-to context, graph transformation

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.10

Related Version A full version of the paper [17] is available at `https://arxiv.org/abs/2004.11792`.

Funding Research partially supported by DFG Project BEMEGA.

1 Introduction

Behavioural equivalences, such as bisimilarity, relate system states with the same behaviour. Here, we are in particular interested in conditional bisimilarity, which allows us to say that two states a, b are bisimilar provided that the environment satisfies a condition $\mathcal{C}$. Work on such conditional bisimulations appears somewhat scattered in the literature (see for instance [21, 15, 11, 3]). They also play a role in the setting of featured transition systems for modelling software product lines [7], where the behaviour of many products is specified in a single transition system. In this setting it is possible to state that two states are bisimilar for certain products, but not for others.

We believe that conditional notions of behavioural equivalence are worthy of further study. In practice it may easily happen that two sub-systems are only ever used in restricted environments and it is too much to ask that they behave equivalently under all possible contexts. Furthermore, instead of giving a simple yes/no-answer, bisimulation checks can answer in a more fine-grained way, specifying conditions which ensure bisimilarity.

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 10; pp. 10:1–10:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We state our results in a very general setting: reactive systems à la Leifer and Milner [22], a categorical abstract framework for rewriting, which provides a suitable framework for deriving bisimulation congruences. In particular, this framework allows to synthesize labelled transitions from plain reaction rules, such that the resulting bisimilarity is automatically a congruence. Intuitively, the label is the minimal context that has to be borrowed from the environment in order to trigger a reduction. (Transitions labelled with such a minimal context will be called representative steps in the sequel. They are related to the idem pushout steps of [22].) Here, we rely on the notion of saturated bisimilarity introduced in [5] and we consider reactive system rules with application conditions, generalizing [16].

Important instances of reactive systems are process calculi with contextualization, bigraphs [18] and double-pushout graph rewriting [8], or in general rewriting in adhesive categories [20]. Hence we can use our results to reason about process calculi as well as dynamically evolving graphs and networks for various different types of graphs (node- or edge-labelled graphs, hypergraphs, etc.). Our contributions in this paper can be summarized as follows:

- We define the notion of conditional bisimilarity, in fact we provide three equivalent definitions: two notions are derived from saturated bisimilarity, where a context step (or a representative step) can be mimicked by several answering steps. Third, we compare with the notion of conditional environment congruence, which is based on the idea of annotating transitions with passive environments enabling a step.
- Conditional bisimulation relations tend to be very large – often infinite in size. In order to handle conditional bisimulation, we propose an up-to context technique that allows to replace infinite conditional bisimulations by possibly finite bisimulations up-to context, which provide witnesses for bisimilarity.
- We compare conditional bisimilarity with related notions of behavioural equivalence.
- To illustrate our concepts, we work out a small case study in the context of double-pushout graph rewriting, where we model message passing over reliable and unreliable channels.

The article is structured as follows: First, in Section 2 we recite the fundamental ideas for reactive systems without conditions, including all preliminary definitions and techniques developed for reactive systems relevant to our work. In Section 3, we consider the refinement to conditional reactive systems, before we turn towards our main contribution in Section 4, which is conditional bisimulation and its up-to variant in Section 5. In Section 6 we give an alternative characterization of conditional bisimilarity and compare to related notions of behavioural equivalence and we conclude in Section 7. All proofs for the theorems in Sections 4 to 6, as well as additional examples can be found in the full version [17].

2 Reactive Systems

2.1 Reactive Systems without Conditions

We denote the composition of arrows $f\colon A \to B$, $g\colon B \to C$ by $f;g\colon A \to C$.

We now define reactive systems, introduced in [22] and extended in [16] with application conditions for rules:

▶ **Definition 2.1** (Reactive system rules, reaction). *Let* $\mathbf{C}$ *be a category with a distinguished object* 0 *(not necessarily initial). A* rule *is a pair* (ℓ, r) *of arrows* $\ell, r\colon 0 \to I$ *(called left-hand side and right-hand side). A* reactive system *is a set of rules.*

Let $\mathcal{R}$ *be a reactive system and* $a, a'\colon 0 \to J$ *be arrows. We say that* a reduces to a' *(*$a \rightsquigarrow a'$*) whenever there exists a rule* $(\ell, r) \in \mathcal{R}$ *with* $\ell, r\colon 0 \to I$ *and an arrow* $c\colon I \to J$ *(the* reactive context*) such that* $a = \ell;c$ *and* $a' = r;c$*.*

Using a notation closer to process caluli, we could write $C[P] \rightsquigarrow C[P']$ whenever there is a reaction rule $P \to P'$ and a context $C[\_]$. Fixing a distinguished object 0 means that we consider only ground reaction rules (as opposed to open reactive systems [19]).

An important instance are reactive systems where the arrows are cospans in a base category $\mathbf{D}$ with pushouts [27, 28]. A *cospan* is a pair of arrows $f_L\colon A \to C$, $f_R\colon B \to C$. A cospan is *input linear* if its left arrow f_L is mono.

$$\begin{tikzcd}
 & & B \arrow[dl, "f_R"'] \arrow[dr, "g_L"] \arrow[rr, bend left, "g"] & & \\
A \arrow[r, "f_L"] \arrow[urr, bend left, "f"] \arrow[rrrr, bend right=50, "f;g"'] & X \arrow[dr, "p_L"'] & (\mathrm{PO}) & Y \arrow[dl, "p_R"] & C \arrow[l, "g_R"'] \\
 & & Z & &
\end{tikzcd}$$

Figure 1 Composition of cospans via pushouts.

Two cospans $f\colon A \xrightarrow{f_L} X \xleftarrow{f_R} B$, $g\colon B \xrightarrow{g_L} Y \xleftarrow{g_R} C$ are composed by taking the pushout (p_L, p_R) of (f_R, g_L) as shown in Figure 1. The result is the cospan $f;g\colon A \xrightarrow{f_L;p_L} Z \xleftarrow{g_R;p_R} C$, where Z is the pushout object of f_R, g_L. For adhesive categories [20], the composition of input linear cospans again yields an input linear cospan (by applying [20, Lemma 12] to the cospan composition diagram). Given an adhesive category $\mathbf{D}$, $ILC(\mathbf{D})$ is the category where the objects are the objects of $\mathbf{D}$, the arrows $f\colon A \to C$ are input linear cospans $f\colon A \to B \leftarrow C$ of $\mathbf{D}$ and composition is performed via pushouts as above. We see an arrow $f\colon A \to C$ of $ILC(\mathbf{D})$ as an object B of $\mathbf{D}$ equipped with two interfaces A, C, and composition glues the inner objects of two cospans via their interfaces. Input linearity is required since we rely on adhesive categories where pushouts along monos are well-behaved and are stable under pullbacks.

$$\begin{tikzcd}
0 \arrow[r, tail] \arrow[rr, bend left, "\ell"] \arrow[dr, tail] \arrow[ddrr, bend right, "a"'] & L \arrow[d, tail] & I \arrow[l, tail] \arrow[r, tail] \arrow[d, tail] \arrow[dd, bend right, "c"'] & R \arrow[d, tail] & 0 \arrow[l, tail] \arrow[ll, bend right, "r"'] \arrow[dl, tail] \arrow[ddll, bend left, "a'"] \\
 & G & C \arrow[l, tail] \arrow[r, tail] & H & \\
 & & 0 \arrow[ul, tail] \arrow[u, tail] \arrow[ur, tail] & &
\end{tikzcd}$$

Figure 2 DPO graph transformation as reactive system steps.

In this article, as a running example we consider $\mathbf{Graph_{fin}}$, which is the category of finite graphs (we use directed multigraphs with node and edge labels) and total graph morphisms as arrows. In $\mathbf{Graph_{fin}}$, monos are exactly the injective graph morphisms. We then use reactive systems over $ILC(\mathbf{Graph_{fin}})$ (input-linear cospans of graphs), i.e. we rewrite graphs with interfaces. If the distinguished object 0 is the empty graph (the initial object of $\mathbf{Graph_{fin}}$), such reactive systems coincide [27] with the well-known *double pushout (DPO) graph transformation* approach [10, 13] when used with injective matches. As shown in Figure 2, a DPO rewrite step $G \Rightarrow H$ can be expressed as a reactive system reaction $a \rightsquigarrow a'$ where the pushouts of the DPO step are obtained from cospan compositions $\ell;c$ and $r;c$.

2.2 Deriving Bisimulation Congruences

The reduction relation $\rightsquigarrow$ generates an unlabelled transition system, on reactive agents (in our example, graphs) as states. A disadvantage of bisimilarity on $\rightsquigarrow$ is that it usually is not a congruence: it is easy to construct an example where neither a nor b can perform a step since no complete left-hand side is present. However, by adding a suitable context c, $a;c$ could contain a full left-hand side and can reduce, whereas $b;c$ can not.

Therefore, to check whether two components can be exchanged, they have to be combined with every possible context and bisimilarity has to be shown for each.

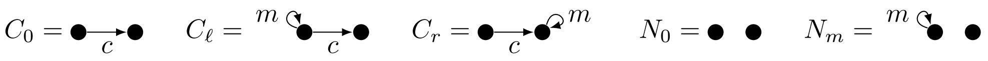
In order to obtain a congruence, we can resort to defining bisimulation on labelled transitions, using as labels the additional contexts that allow an agent to react [22, 16].

▶ **Definition 2.2** (Context step (without conditions) [16]). *Let $\mathcal{R}$ be a reactive system and $a\colon 0 \to J$, $f\colon J \to K$, $a'\colon 0 \to K$ be arrows. We write $a \xrightarrow{f}_C a'$ whenever $a;f \rightsquigarrow a'$ (i.e. there exists a rule $(\ell, r) \in \mathcal{R}$ and an arrow c such that $a;f = \ell;c$, $a' = r;c$). Such steps are called* context steps.

$$\begin{tikzcd}
0 \arrow[r, "\ell"] \arrow[d, "a"'] & I \arrow[d, "c"] & 0 \arrow[l, "r"'] \arrow[dl, "a'"] \\
J \arrow[r, "f"] & K &
\end{tikzcd}$$

The name *context step* stems from the fact that a cannot do a reaction on its own, but requires an additional context f. This can be seen in the following example:

▶ **Example 2.3** (Context step (without conditions)). Consider the following reactive system over $ILC(\mathbf{Graph_{fin}})$, where we model a network of nodes that pass messages (represented by m-loops) over communication channels. Let the following graphs be given:



We can now represent the transmission of a message from the left node to the right node using the rule $P = (\emptyset \to C_\ell \leftarrow C_0,\ \emptyset \to C_r \leftarrow C_0)$. All graph morphisms are induced by edge labels and position of nodes, i.e. the left node is always mapped to the left node.

Observe that a channel by itself ($a = \emptyset \to C_0 \leftarrow N_0$) cannot do a reaction, since there is no message to be transferred. However, if a message on the left node is borrowed ($f = N_0 \to N_m \leftarrow N_0$), the example rule can be applied. As a result, we obtain the context step $(\emptyset \to C_0 \leftarrow N_0) \xrightarrow{(N_0 \to N_m \leftarrow N_0)}_C (\emptyset \to C_r \leftarrow N_0)$.

A bisimulation relation over $\to_C$ is called *saturated bisimulation*, as it checks all contexts. Consequently, saturated bisimilarity $\sim_C$ ($\sim_{SAT}$ in [16]) is a congruence [5, 16], i.e., it is closed under contextualization. In other words $a{\sim_C}b$ implies $a;c{\sim_C}b;c$ for all contexts c.

2.3 Representative Squares

Checking bisimilarity of context steps is impractical: usually, f can be chosen from an infinite set of possible contexts, which all have to be checked. Most of these contexts are larger than necessary, that is, they contain elements that do not actively participate in the reduction. (In Example 2.3, contexts can be arbitrarily large, as long as they have an m-loop on the left node.) An improvement would be to check only the minimal contexts from which all other context steps can be derived.

When checking which contexts are required to make a rule applicable, in the reaction diagram (Definition 2.2) the arrows a, ℓ are given and we need to check for possible values of f (which generate matching c, a'). To derive a set of contexts f which is as small as possible – preferably finite – [6, 16] introduced the notion of representative squares, which describe methods to produce squares from a pair a, ℓ in a representative way.

▶ **Definition 2.4** (Representative squares [6]). *A class κ of commuting squares in a category* $\mathbf{C}$ *is* representative *if κ satisfies the following condition: for each commuting square $(\alpha_1, \alpha_2, \delta_1, \delta_2)$ in* $\mathbf{C}$ *there exists a commuting square $(\alpha_1, \alpha_2, \beta_1, \beta_2)$ in κ and an arrow γ, such that $\delta_1 = \beta_1;\gamma$, $\delta_2 = \beta_2;\gamma$. This situation is depicted in Figure 3.*

$$\begin{tikzcd} A \arrow[r, "\alpha_1"] \arrow[d, "\alpha_2"'] & B \arrow[d, "\delta_1"] \\ C \arrow[r, "\delta_2"'] & D' \end{tikzcd} \quad \rightarrow \quad \begin{tikzcd} A \arrow[r, "\alpha_1"] \arrow[dd, "\alpha_2"'] & B \arrow[d, "\beta_1"'] \arrow[dd, bend left, "\delta_1"] \\ & D \arrow[d, "\gamma"] \\ C \arrow[ur, "\beta_2"] \arrow[r, "\delta_2"'] & D' \end{tikzcd}$$

Figure 3 Every commuting square of the category (left) can be reduced to a representative square in κ and an arrow γ which extends the representative square to the original square (right).

For two arrows $\alpha_1 \colon A \to B$, $\alpha_2 \colon A \to C$, we define $\kappa(\alpha_1, \alpha_2)$ as the set of pairs of arrows (β_1, β_2) which, together with α_1, α_2, form representative squares in κ.

The original paper on reactive systems [22] used the (more restrictive) notion of idem pushouts instead of representative squares. Unfortunately, the universal property of idem pushouts leads to complications, in particular for cospan categories, where one has to resort to the theory of bicategories in order to be able to express this requirement. For the purposes of this paper, we stick to the simpler notion of representative squares, in order to keep our results independent of the concrete class of squares chosen.

The question arises which constructions yield suitable classes of representative squares, ideally with finite $\kappa(\alpha_1, \alpha_2)$, in order to represent all possible contexts δ_1, δ_2 with a finite set of representative contexts β_1, β_2. Pushouts can be used when they exist [16], however, they do not exist for $\mathit{ILC}(\mathbf{Graph_{fin}})$.

For adhesive categories, borrowed context diagrams – initially introduced as an extension of DPO rewriting [9] – can be used as representative squares. Before we can introduce such diagrams, we first need the notion of jointly epi.

▶ **Definition 2.5** (Jointly epi). *A pair of arrows $f \colon B \to D$, $g \colon C \to D$ is* jointly epi *(JE) if for each pair of arrows $d_1, d_2 \colon D \to E$ the following holds: if $f;d_1 = f;d_2$ and $g;d_1 = g;d_2$, then $d_1 = d_2$.*

In $\mathbf{Graph_{fin}}$ jointly epi equals jointly surjective, meaning that each node or edge of D is required to have a preimage under f or g or both (it contains only elements from B or C).

▶ **Definition 2.6** (Borrowed context diagram [16]). *A commuting diagram in the category $\mathit{ILC}(\mathbf{C})$, where* $\mathbf{C}$ *is adhesive, is a* borrowed context diagram *whenever it has the form of the diagram shown below, and the four squares in the base category* $\mathbf{C}$ *are jointly epi (JE), pushout (PO) or pullback (PB) as indicated.*

$$\begin{tikzcd} 0 \arrow[r, tail] \arrow[d, tail] \arrow[rr, bend left, "\ell"] \arrow[dd, bend right, "a"'] \arrow[dr, phantom, "\text{JE}"] & L \arrow[d, tail] \arrow[dr, phantom, "\text{PO}"] & I \arrow[l] \arrow[d, tail] \arrow[dd, bend left, "c"] \\ G \arrow[r, tail] & G^+ & C \arrow[l] \\ J \arrow[u] \arrow[r, tail] \arrow[rr, bend right, "f"'] \arrow[ur, phantom, "\text{PO}"] & F \arrow[u] \arrow[ur, phantom, "\text{PB}"] & K \arrow[l] \arrow[u] \end{tikzcd}$$

The top left jointly epi square and the bottom left pushout ensure that the borrowed context f is not larger than necessary [9]. We will discuss an example below (Example 2.9). For additional examples, we refer to [9].

For adhesive categories, borrowed context diagrams form a representative class of squares [16]. Furthermore, for some categories (such as $\mathbf{Graph_{fin}}$), there are – up to isomorphism – only finitely many jointly epi squares for a given span of monos and hence only finitely many borrowed context diagrams given a, ℓ (since pushout complements along monos in adhesive categories are unique up to isomorphism).

This motivates the following finiteness assumption that we will refer to in this paper: given a, ℓ, we require that $\kappa(a, \ell)$ is finite. (FIN)

2.4 Representative Steps

It is possible to define a reaction relation based on representative squares. By requiring that the left square is representative, we ensure that the contexts $\hat{f}$ are not larger than necessary:

▶ **Definition 2.7** (Representative step (without conditions) [16]). *Let $a\colon 0 \to J$, $\hat{f}\colon J \to K$, $a'\colon 0 \to K$ be arrows. We write $a \xrightarrow{\hat{f}}_R a'$ if a context step $a \xrightarrow{\hat{f}}_C a'$ is possible (i.e. $a;\hat{f} \rightsquigarrow a'$, i.e. for some rule (ℓ, r) and some arrow $\hat{c}$ we have $a;\hat{f} = \ell;\hat{c}$ and $r;\hat{c} = a'$) and additionally $\kappa(a, \ell) \ni (\hat{f}, \hat{c})$ (i.e. the arrows $(a, \ell, \hat{f}, \hat{c})$ form a representative square). Such steps are called* representative steps*.*

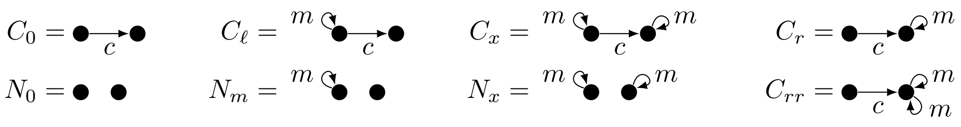
$$\begin{tikzcd}
0 \arrow[r, "\ell"] \arrow[d, "a"'] & I \arrow[d, "c"] & 0 \arrow[l, "r"'] \arrow[dl, "a'"] \\
J \arrow[r, "f"] & K &
\end{tikzcd}$$

$$\begin{tikzcd}
0 \arrow[r, "\ell"] \arrow[dd, "a"'] & I \arrow[d, "\hat{c}"'] \arrow[dd, bend left, "c"] & 0 \arrow[l, "r"'] \arrow[ddl, "a'"] \\
& K' \arrow[d, "\hat{g}"] & \\
J \arrow[ur, "\hat{f}"] \arrow[r, "f"'] & K &
\end{tikzcd}$$

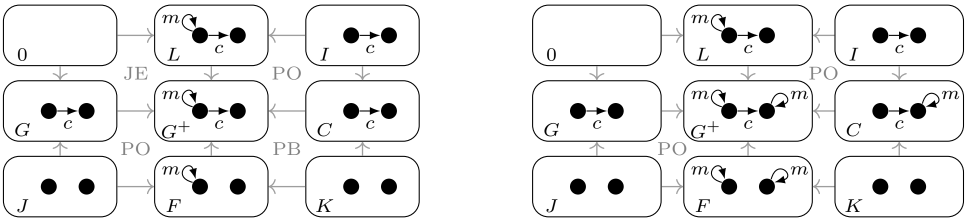
▶ Remark 2.8. Definitions 2.2 and 2.4 imply that every context step $a \xrightarrow{f}_C a'$ (top diagram) can be reduced to a representative step $a \xrightarrow{\hat{f}}_R r;\hat{c}$ (bottom diagram), a fact used in the proofs.

For this, we construct the representative square $(a, \ell, \hat{f}, \hat{c}) \in \kappa$ (which, according to Definition 2.4, always exists) from the square (a, ℓ, f, c) describing the context step. We obtain arrows $\hat{f}, \hat{c}$ and an arrow $\hat{g}$ which completes $\hat{f}, \hat{c}$ to f, c (i.e. $\hat{f};\hat{g} = f$, $\hat{c};\hat{g} = c$).

▶ **Example 2.9** (Representative steps). Let the following graphs be given:



As before (Example 2.3), the rule $P = (\emptyset \to C_\ell \leftarrow C_0,\ \emptyset \to C_r \leftarrow C_0)$ transfers a message. One possible context step allows a channel C_0 to borrow a message N_m and do a transfer: $(\emptyset \to C_0 \leftarrow N_0) \xrightarrow{(N_0 \to N_m \leftarrow N_0)}_C (\emptyset \to C_r \leftarrow N_0)$.



(a) Borrowed context diagram for $C_0 \xrightarrow{N_m}_R C_r$. **(b)** Commuting diagram for $C_0 \xrightarrow{N_x}_C C_{rr}$.

Figure 4 Diagrams for the two steps described in Example 2.9.

Another possible context step is $(\emptyset \to C_0 \leftarrow N_0) \xrightarrow{(N_0 \to N_x \leftarrow N_0)}_C (\emptyset \to C_{rr} \leftarrow N_0)$, i.e. an additional message on the right node is borrowed. Clearly, this is a valid context step, but the right message is not required by the rule, and we do not want to consider such steps in our analysis (by adding yet more messages, we obtain infinitely many context steps).

However, the second context step is not a representative step. We try to construct a borrowed context diagram: First we fill in the graphs given by a, f and ℓ, then we construct the bottom left pushout, we obtain $G^+ = C_x$ as depicted in Figure 4b. Then however the top left square is not jointly epi, since neither C_ℓ (from ℓ) nor C_0 (from a) provide a preimage for the right m-loop.

On the other hand, the first context step is representative, since there $G^+ = C_\ell$ does not contain the problematic right m-loop and it is possible to complete the borrowed context diagram as shown in Figure 4a. (To obtain the result of the step, the right-hand side a' is constructed just as for context steps (see Example 2.3), which is not depicted here.)

In a *semi-saturated bisimulation*, $\to_R$-steps are answered by $\to_C$-steps (for every $(a,b) \in R$ and step $a \xrightarrow{f}_R a'$ there is $b \xrightarrow{f}_C b'$ such that $(a',b') \in R$). The resulting bisimilarity $\sim_R$ is identical [16] to saturated bisimilarity (i.e. $\sim_R = \sim_C$) and therefore also a congruence. Whenever (Fin) holds, $\sim_R$ is amenable to mechanization, since we have to consider only finitely many $\to_R$-steps ($\to_R$ is finitely branching).

Note that answering $\to_R$-steps with $\to_R$-steps gives a different, finer notion of behavioural equivalence, which we do not treat here [16].

3 Conditions for Reactive Systems

The reactive systems defined so far cannot represent rules where a certain component is required to be absent: whenever a reaction $a \rightsquigarrow a'$ is possible, a reaction $a;c \rightsquigarrow a';c$ (with additional context c) is also possible, with no method to prevent this. Restricting rule applications can be useful, e.g. to model access to a shared resource, which may only be accessed if no other entity is currently using it.

For graph transformation systems, application conditions with a first-order logic flavour have been studied extensively (e.g. in [12, 14]) and generalized to reactive systems in [6]. If we interpret such conditions in $\mathit{ILC}(\mathbf{Graph_{fin}})$, we obtain a logic that subsumes first-order logic (for more details on expressiveness see [6]).

In this section, we summarize the definitions from [6] and define shifting of conditions as partial evaluation. We then summarize the changes that are necessary to extend reactive systems with conditions. An example for conditional reactive systems will be discussed later (Example 4.3). For further examples, we refer to the full version and to [6].

3.1 Conditions and Satisfiability

▶ **Definition 3.1** (Condition [6]). *Let* $\mathbf{C}$ *be a category. The set of conditions* $\mathrm{Cond}(A)$ *over an object* A *is defined inductively as:*

- $\mathrm{true}_A := (A, \forall, \emptyset) \in \mathrm{Cond}(A)$*,* $\mathrm{false}_A := (A, \exists, \emptyset) \in \mathrm{Cond}(A)$ *(base case)*
- $\mathcal{A} = (A, \mathcal{Q}, S) \in \mathrm{Cond}(A)$*, where* $A = \mathrm{Ro}(\mathcal{A})$ *is the* root object *of* $\mathcal{A}$*,*
  - $\mathcal{Q} \in \{\forall, \exists\}$ *is a quantifier and*
  - S *is a finite set of pairs* $(h, \mathcal{A}')$*, where* $h\colon A \to A'$ *is an arrow and* $\mathcal{A}' \in \mathrm{Cond}(A')$*.*

Note that conditions can be represented as finite trees.

▶ **Definition 3.2** (Satisfiability of conditions [6]). *Let* $\mathcal{A} \in \mathrm{Cond}(A)$*. For an arrow* $a\colon A \to B$ *and a condition* $\mathcal{A}$ *we define the satisfaction relation* $a \models \mathcal{A}$ *as follows:*

- $a \models (A, \forall, S)$ *iff for every pair* $(h, \mathcal{A}') \in S$ *and every arrow* $g\colon \mathrm{Ro}(\mathcal{A}') \to B$ *we have: if* $a = h;g$*, then* $g \models \mathcal{A}'$*.*
- $a \models (A, \exists, S)$ *iff there exists a pair* $(h, \mathcal{A}') \in S$ *and an arrow* $g\colon \mathrm{Ro}(\mathcal{A}') \to B$ *such that* $a = h;g$ *and* $g \models \mathcal{A}'$*.*

We write $\mathcal{A} \models \mathcal{B}$ *(*$\mathcal{A}$ *implies* $\mathcal{B}$*) if for every arrow* c *with* $\mathrm{dom}(c) = \mathrm{Ro}(\mathcal{A}) = \mathrm{Ro}(\mathcal{B})$ *we have: if* $c \models \mathcal{A}$*, then* $c \models \mathcal{B}$*. Two conditions are equivalent (*$\mathcal{A} \equiv \mathcal{B}$*) if* $\mathcal{A} \models \mathcal{B}$ *and* $\mathcal{B} \models \mathcal{A}$*.*

▶ **Proposition 3.3** (Boolean operations). *We define the following Boolean operations on conditions:*

- $\neg(A, \forall, S) := (A, \exists, \{(h, \neg\mathcal{A}') \mid (h, \mathcal{A}') \in S\})$ *and*
  $\neg(A, \exists, S) := (A, \forall, \{(h, \neg\mathcal{A}') \mid (h, \mathcal{A}') \in S\})$
- $\mathcal{A} \vee \mathcal{B} := (A, \exists, \{(\mathrm{id}_A, \mathcal{A}), (\mathrm{id}_A, \mathcal{B})\})$ *for two conditions* $\mathcal{A}, \mathcal{B} \in \mathrm{Cond}(A)$
- $\mathcal{A} \wedge \mathcal{B} := (A, \forall, \{(\mathrm{id}_A, \mathcal{A}), (\mathrm{id}_A, \mathcal{B})\})$ *for two conditions* $\mathcal{A}, \mathcal{B} \in \mathrm{Cond}(A)$

These operations satisfy the standard laws of propositional logic, i.e. true_A *is satisfied by every arrow with domain* A*,* false_A *is satisfied by no arrow;* $a \models \neg\mathcal{A}$ *if and only if* $a \not\models \mathcal{A}$*;* $a \models (\mathcal{A} \vee \mathcal{B})$ *if and only if* $a \models \mathcal{A} \vee a \models \mathcal{B}$*, analogously for* $\mathcal{A} \wedge \mathcal{B}$*.*

3.2 Shifting as Partial Evaluation of Conditions

When evaluating conditions, it is sometimes known that a given context is guaranteed to be present. In this case, a condition can be rewritten, using representative squares, under the assumption that this context is provided by the environment. This operation is known as shift [14]:

▶ **Definition 3.4** (Shift of a condition [6]). *Given a fixed class of representative squares* κ*, the* shift of a condition $\mathcal{A} = (A, \mathcal{Q}, S)$ along an arrow $c\colon A \to B$ *is inductively defined as follows:*

$$\mathcal{A}_{\downarrow c} := \Big(B, \mathcal{Q}, \Big\{(\beta, \mathcal{A}'_{\downarrow \alpha}) \,\Big|\, (h, \mathcal{A}') \in S,\ (\alpha, \beta) \in \kappa(h, c)\Big\}\Big)$$

The shift operation can be understood as a partial evaluation of $\mathcal{A}$ *under the assumption that* c *is already present. It satisfies* $c;d \models \mathcal{A} \iff d \models \mathcal{A}_{\downarrow c}$*.*

If we assume that (Fin) holds, shifting a finite condition will again result in a finite condition. Representative squares as well as shift play a major role in the diagrammatic proofs.

3.3 Conditional Reactive Systems

We now extend reactive systems with application conditions:

▶ **Definition 3.5** (Conditional reactive system [6]). *A* rule with condition *is a triple* $(\ell, r, \mathcal{B})$ *where* $\ell, r \colon 0 \to I$ *are arrows and* $\mathcal{B}$ *is a condition with root object* I*. A* conditional reactive system *is a set of rules with conditions.*

As the root object I of the condition is the codomain of the rule arrow, it is also the domain of the reactive context, which has to satisfy the rule condition in order to be able to apply the rule:

▶ **Definition 3.6** (Reaction). *Let* a, a' *be arrows of a conditional reactive system with rules* $\mathcal{R}$*. We say that* a reduces to a' *(*$a \rightsquigarrow a'$*) whenever there exists a rule* $(\ell, r, \mathcal{B}) \in \mathcal{R}$ *with* $\ell, r \colon 0 \to I$ *and a reactive context* $c \colon I \to J$ *such that* $a = \ell;c$*,* $a' = r;c$ *and additionally* $c \models \mathcal{B}$*.*

In order to define a bisimulation for conditional reactive systems that is also a congruence, it is necessary to enrich labels with conditions derived from the application conditions. Since we can not assume that the full context is present, the application condition might refer to currently unknown parts of the context and this has to be suitably integrated into the label.

▶ **Definition 3.7** (Context/representative step with conditions [16]). *Let* $\mathcal{R}$ *be a conditional reactive system, let* $a \colon 0 \to J$*,* $f \colon J \to K$*,* $a' \colon 0 \to K$ *be arrows and* $\mathcal{A} \in \mathrm{Cond}(K)$ *be a condition. We write* $a \xrightarrow{f, \mathcal{A}}_C a'$ *whenever there exists a rule* $(\ell, r, \mathcal{B}) \in \mathcal{R}$ *and an arrow* c *such that* $a;f = \ell;c$*,* $a' = r;c$ *(i.e. the reaction is possible without conditions) and furthermore* $\mathcal{A} \models \mathcal{B}_{\downarrow c}$ *(an additional context has to satisfy a condition* $\mathcal{A}$ *which is at least as strong as the rule condition* $\mathcal{B}$*, shifted over* c*). Such steps are called* context steps*.*

We write $a \xrightarrow{f, \mathcal{A}}_R a'$ *whenever* $a \xrightarrow{f, \mathcal{A}}_C a'$*,* $\kappa(a, \ell) \ni (f, c)$ *and* $\mathcal{A} = \mathcal{B}_{\downarrow c}$*. Such steps are called* representative steps*.*

$$\begin{tikzcd}
 & \mathcal{B} & \\
0 \arrow[r, "\ell"] \arrow[d, "a"'] & I \arrow[d, "c"'] & 0 \arrow[l, "r"'] \arrow[dl, "a'"] \\
J \arrow[r, "f"] & K & \\
 & \mathcal{A} &
\end{tikzcd}$$

Conditions are represented graphically in the form of "arrowhead shapes" depicted next to the root object. Intuitively $a \xrightarrow{f, \mathcal{A}}_C a'$ means that a can make a step to a' when borrowing f, if the yet unknown context beyond f satisfies condition $\mathcal{A}$ (since this context does not directly participate in the reduction, we call it *passive context*). In the case of a representative step, we require that a context step is possible, the borrowed context is minimal, and the condition on the passive context is not stronger than necessary.

▶ Remark 3.8. Definitions 2.4 and 3.7 imply, analogously to Remark 2.8, that every context step $a \xrightarrow{f, \mathcal{A}}_C a'$ can be reduced to a representative step $a \xrightarrow{\hat{f}, \mathcal{B}_{\downarrow \hat{c}}}_R r;\hat{c}$.

We now extend (semi-)saturated bisimilarity to rules with conditions:

▶ **Definition 3.9** ((Semi-)Saturated bisimilarity [16]). *A* saturated bisimulation *is a symmetric relation* R*, relating pairs of arrows* $a, b \colon 0 \to J$*, such that: for all* $(a, b) \in R$ *and for every context step* $a \xrightarrow{f, \mathcal{A}}_C a'$ *there exist answering moves* $b \xrightarrow{f, \mathcal{B}_i}_C b'_i$*,* $i \in I$*, such that* $(a', b'_i) \in R$ *and* $\mathcal{A} \models \bigvee_{i \in I} \mathcal{B}_i$*.*

Two arrows a, b are called saturated bisimilar *($(a,b) \in \sim_C$) whenever there exists a saturated bisimulation R with $(a,b) \in R$. Similarly, for* semi-saturated bisimilarity *we require that $\rightarrow_R$-steps of a can be answered by $\rightarrow_C$-steps of b. Saturated and semi-saturated bisimilarity agree and both are congruences [16].*

The logic does not support infinite disjunctions, so $\mathcal{A} \models \bigvee_{i\in I} \mathcal{B}_i$ means that for every d with $d \models \mathcal{A}$, there exists $i \in I$ such that $d \models \mathcal{B}_i$.

4 Conditional Bisimilarity

We will now introduce our new results on conditional bisimilarity: as stated earlier, our motivation is to extend the notion of saturated bisimilarity, which is often too strict, since it requires that two system states behave identically in all possible contexts. However, sometimes it is enough to ensure behavioural equivalence only in specific environments.

Hence we now replace standard bisimilarity, which is a binary relation, by a ternary relation – called conditional relation – with tuples of the form $(a, b, \mathcal{C})$, which can be read as: a, b are bisimilar in all contexts satisfying $\mathcal{C}$.

4.1 Definition, Properties and Examples

▶ **Definition 4.1** (Conditional relation, closure under contextualization, conditional congruence)**.** *A* conditional relation *is a set of triples $(a, b, \mathcal{C})$, where $a, b \colon 0 \to J$ are arrows with identical target and $\mathcal{C}$ is a condition over J. A conditional relation R is* reflexive *if $(a, a, \mathcal{C}) \in R$ for all $a, \mathcal{C}$ with $\mathrm{codom}(a) = \mathrm{Ro}(\mathcal{C})$;* symmetric *if $(a, b, \mathcal{C}) \in R$ implies $(b, a, \mathcal{C}) \in R$;* transitive *if $(a, b, \mathcal{C}) \in R$ and $(b, c, \mathcal{C}) \in R$ implies $(a, c, \mathcal{C}) \in R$. R is* closed under contextualization *if $(a, b, \mathcal{C}) \in R$ implies $(a;d, b;d, \mathcal{C}_{\downarrow d}) \in R$. R is a* conditional congruence *if it is additionally an equivalence (reflexive, symmetric, transitive).*

Closure under contextualization means that whenever a, b are related under a context satisfying $\mathcal{C}$, then they are still related when we contextualize under d, where however the condition has to be shifted since we commit to the fact that the context is of the form $d;c$ for some c.

Note that the root object of the condition is not the source of a (as is the case for satisfiability), but the target $\mathrm{codom}(a)$. This is because we do not state a condition on the arrows a, b themselves, but on the context in which they are embedded ($a;f$ resp. $b;f$ for some context f), so the condition is over $\mathrm{dom}(f) = \mathrm{codom}(a)$.

▶ **Definition 4.2** (Conditional bisimulation)**.** *A conditional bisimulation R is a symmetric conditional relation such that the following holds: for each triple $(a, b, \mathcal{C}) \in R$ and each context step $a \xrightarrow{f,\mathcal{A}}_C a'$, there are answering steps $b \xrightarrow{f,\mathcal{B}_i}_C b'_i$, $i \in I$, and conditions $\mathcal{C}'_i$ such that $(a', b'_i, \mathcal{C}'_i) \in R$ and $\mathcal{A} \wedge \mathcal{C}_{\downarrow f} \models \bigvee_{i\in I} (\mathcal{C}'_i \wedge \mathcal{B}_i)$. Two arrows are* conditionally bisimilar under $\mathcal{C}$ *($(a, b, \mathcal{C}) \in \overset{\circ}{\sim}_C$) whenever a conditional bisimulation R with $(a, b, \mathcal{C}) \in R$ exists.*[1]

The condition is to be understood as follows: For every step, we have a borrowed context f and an additional passive context d (as explained below Definition 3.7). The condition $\mathcal{C}$ from the triple refers to the full context of a (hence $f;d \models \mathcal{C}$ or equivalently $d \models \mathcal{C}_{\downarrow f}$), while $\mathcal{A}$, coming from the context step, only refers to the passive context (hence $d \models \mathcal{A}$).

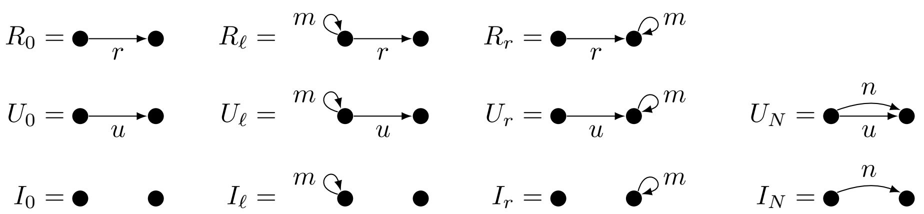
[1] Note that since conditional bisimulations are closed under union, $\overset{\circ}{\sim}_C$ is itself a conditional bisimulation.

If these two are satisfied (left-hand side of the implication), we require answering steps which also impose conditions on the passive context ($d \models \mathcal{B}_i$). Additionally, we choose conditions $\mathcal{C}'_i$, which ensure that the chosen answering steps yield pairs a', b'_i which are bisimilar under $\mathcal{C}'_i$. As for saturated bisimilarity [16, remark after Definition 15], we need to allow several answering moves for a single step of a: the answering step taken by b might depend on the context, using different rules for contexts satisfying different conditions $\mathcal{B}_i$. We just have to ensure that all answering step conditions together (disjunction on the right-hand side) fully cover the conditions under which the step of a is feasible (left-hand side).

▶ **Example 4.3** (Message passing over unreliable channels)**.** We now work in the category of input-linear cospans of graphs, i.e., $ILC(\mathbf{Graph_{fin}})$.

We extend our previous example (cf. Example 2.3) of networked nodes, introducing different types of channels. A channel can be reliable or unreliable, indicated by an r-edge or u-edge respectively. Sending a message over a reliable channel always succeeds (rule P_R), while an unreliable channel only transmits a message if there is no noise (indicated by a parallel n-edge) in the environment that disturbs the transmission (rule P_U).

To represent this situation as a reactive system, let the following graphs be given:



We can now represent the transmission of a message using the following rules with application conditions, where $\mathcal{A}_U$ states that no n-edge exists:

$$\begin{aligned} P_R &= (\emptyset \to R_\ell \leftarrow R_0,\ \emptyset \to R_r \leftarrow R_0,\ \mathrm{true}_{R_0}) \\ P_U &= (\emptyset \to U_\ell \leftarrow U_0,\ \emptyset \to U_r \leftarrow U_0,\ \mathcal{A}_U) \\ \mathcal{A}_U &= \big(U_0, \forall, \{(U_0 \to U_N \leftarrow U_0,\ \mathrm{false}_{U_0})\}\big) \end{aligned}$$

Hence the application condition $\mathcal{A}_U$ says that the context must not be decomposable into $U_0 \to U_N \leftarrow U_0$ and some other cospan, i.e., the u-edge in the interface has no parallel n-edge. In other words: there is no noise.

We compare the behaviour of a reliable channel ($r := \emptyset \to R_0 \leftarrow I_0$) to that of an unreliable channel ($u := \emptyset \to U_0 \leftarrow I_0$). It is easy to see that they are not saturated bisimilar: r can do a step by borrowing a message on the left ($f := I_0 \to I_\ell \leftarrow I_0$) without further restrictions (i.e. using an environment condition $\mathcal{A} = \mathrm{true}$). But u is unable to answer this step, because the corresponding rule is only applicable if no n-edge is present.

However, r and u are conditionally bisimilar under the assumption that no n-edge is present ($\mathcal{C} = \mathcal{A}_C = \big(I_0, \forall, \{(I_0 \to I_N \leftarrow I_0,\ \mathrm{false}_{I_0})\}\big)$), i.e. there exists a conditional bisimulation that contains $(r, u, \mathcal{A}_C)$. A direct proof is hard, since the proof involves checking infinitely many context steps, since messages accumulate on the right-hand side. However, in Example 4.9 we will use an argument based on representative steps to construct a proof.

▶ Remark 4.4 (Condition strengthening). It holds that $(a, b, \mathcal{C}') \in \overset{\circ}{\sim}_C$, $\mathcal{C} \models \mathcal{C}'$ implies $(a, b, \mathcal{C}) \in \overset{\circ}{\sim}_C$. (This is due to the fact that $\mathcal{C} \models \mathcal{C}'$ implies $\mathcal{C}_{\downarrow f} \models \mathcal{C}'_{\downarrow f}$ which, in Definition 4.2, implies $\mathcal{A} \wedge \mathcal{C}_{\downarrow f} \models \mathcal{A} \wedge \mathcal{C}'_{\downarrow f}$ for any condition $\mathcal{A}$ and arrow f.)

▶ Remark 4.5. It can be shown that conditional bisimilarity $\overset{\circ}{\sim}_C$ is a conditional congruence, this follows as a corollary of Theorem 6.3 which will be shown later. This is an important plausibility check, since reactive systems have been introduced with the express purpose to define and reason about bisimulation congruences.

Our motivation for introducing the notion of conditional bisimilarity was to check whether two systems are behaviourally equivalent, when they are put into a context that satisfies some condition $\mathcal{C}$. It is not immediately obvious that our definition can be used for this purpose, since all context steps are checked, not just the ones that actually satisfy $\mathcal{C}$.

Hence we now show that our definition is sound, i.e. if two systems are conditionally bisimilar, then they show identical behaviour under all contexts that satisfy $\mathcal{C}$.

▶ **Theorem 4.6.** *Let R be a conditional bisimulation. Then $R' = \{(a;d, b;d) \mid (a, b, \mathcal{C}) \in R \wedge d \models \mathcal{C}\}$ is a bisimulation for the reaction relation $\rightsquigarrow$.*

Note that the converse of Theorem 4.6 (if R' is a bisimulation, then R is a conditional bisimulation) does not hold. For a counterexample, we refer to the full version.

4.2 Representative Conditional Bisimulations

Checking whether two arrows are conditionally bisimilar, or whether a given relation is a conditional bisimulation, can be hard in practice, since we have to check all possible context steps, of which there are typically infinitely many.

For saturated bisimilarity, we used representative steps instead of context steps (cf. Sections 2.3 and 2.4) to reduce the number of contexts to be checked. In this section, we extend our definition of conditional bisimulation to use representative steps and prove that the resulting bisimilarity is identical to the one previously defined.

▶ **Definition 4.7** (Representative conditional bisimulation). *A* representative conditional bisimulation *R is a symmetric conditional relation such that the following holds: for each triple $(a, b, \mathcal{C}) \in R$ and each representative step $a \xrightarrow{f,\mathcal{A}}_R a'$, there are answering context steps $b \xrightarrow{f,\mathcal{B}_i}_C b'_i$ and conditions $\mathcal{C}'_i$ such that $(a', b'_i, \mathcal{C}'_i) \in R$ and $\mathcal{A} \wedge \mathcal{C}_{\downarrow f} \models \bigvee_{i \in I} (\mathcal{C}'_i \wedge \mathcal{B}_i)$. Two arrows are* representative conditionally bisimilar under $\mathcal{C}$ *($(a, b, \mathcal{C}) \in \overset{\circ}{\sim}_R$) whenever a representative conditional bisimulation R with $(a, b, \mathcal{C}) \in R$ exists.*

We now show that these two conditional bisimilarities are equivalent.

▶ **Theorem 4.8.** *Conditional bisimilarity and representative conditional bisimilarity coincide, that is, $\overset{\circ}{\sim}_C = \overset{\circ}{\sim}_R$.*

▶ **Example 4.9** (Message passing over unreliable channels, continued). Consider the reactive system of Example 4.3. There exists a representative conditional bisimulation R such that $(\emptyset \to R_0 \leftarrow I_0,\ \emptyset \to U_0 \leftarrow I_0,\ \mathcal{A}_C) \in R$.

We consider the representative steps that are possible from either R_0 or U_0 and only explain the most interesting cases:

- R_0 can do a step using rule P_R by borrowing a message on the left node, that is, $f = I_0 \to I_\ell \leftarrow I_0$, and reacting to R_r. No further restrictions on the environment are necessary, so $\mathcal{A} = \text{true}$. U_0 can answer this step using P_U and reacts to U_r, but only if no noise is present (environment satisfies $\mathcal{B}_i = \mathcal{A}_C$). We evaluate the implication $\mathcal{A} \wedge \mathcal{C}_{\downarrow f} \equiv \text{true} \wedge \mathcal{A}_{C\downarrow f} \equiv \mathcal{A}_C \models \bigvee_{i \in I} (\mathcal{C}'_i \wedge \mathcal{A}_C) \equiv \bigvee_{i \in I} (\mathcal{C}'_i \wedge \mathcal{B}_i)$, setting $\mathcal{C}'_i = \mathcal{A}_C$. (Note that $\mathcal{A}_{C\downarrow f} \equiv \mathcal{A}_C$ since $\mathcal{A}_C$ forbids the existence of an n-edge between the two interface nodes and f is unrelated, providing an m-loop on the left-hand node.) We now require $(\emptyset \to R_r \leftarrow I_0,\ \emptyset \to U_r \leftarrow I_0,\ \mathcal{A}_C) \in R$.

- Symmetrically, U_0 can do a step using P_U by borrowing a message on the left node, reacting to U_r in an environment without noise ($\mathcal{A} = \mathcal{A}_C$). R_0 can answer this step under any condition $\mathcal{B}_i$. Then, the implication is satisfied if we set $\mathcal{C}'_i = \mathcal{A}_C$, so we require again $(\emptyset \to R_r \leftarrow I_0,\ \emptyset \to U_r \leftarrow I_0,\ \mathcal{A}_C) \in R$.
- There are additional representative steps that differ in how much of the left-hand side is borrowed, but can be proven analogously to the two previously discussed steps.

This means that we have to add the pair $(\emptyset \to R_r \leftarrow I_0,\ \emptyset \to U_r \leftarrow I_0,\ \mathcal{A}_C)$ to R and to continue adding pairs until we obtain a bisimulation: with every step, a new triple with an additional m-loop on the right node is added to the relation, therefore, the smallest conditional bisimulation has infinite size. However, except for the additional m-loop on the right node, which does not affect rule application, this pair is identical to the initial one and we can hence use a similar argument. In Section 5 we show how to make this formal, using up-to techniques, and thus obtain a completely mechanized proof. In summary, we conclude that R_0 is conditionally bisimilar to U_0 under the condition $\mathcal{A}_C$.

▶ **Example 4.10** (Unreliable channel vs. no channel)**.** For Examples 4.3 and 4.9, it can also be shown that under the condition $\neg\mathcal{A}_C$, the unreliable channel $\emptyset \to U_0 \leftarrow I_0$ is conditionally bisimilar to not having a channel between the two nodes ($\emptyset \to I_0 \leftarrow I_0$).

In this case, U_0 can still do a reaction under $\mathcal{A}_C$. Then, I_0 can answer with an empty set of steps. The implication $\mathcal{A}_C \wedge \mathcal{C}_{\downarrow f} \models \bigvee_{i\in I}(\mathcal{C}'_i \wedge \mathcal{B}_i)$ is then simplified to $\mathcal{A}_C \wedge \neg\mathcal{A}_C \models \text{false}$, which is easily seen to be valid.

5 Up-to Techniques for Proving Conditional Bisimilarity

Our optimizations so far involved replacing context steps by representative steps, which ensure finite branching and thus greatly reduce the proof obligations for a single step. However, it can still happen very easily that the smallest possible bisimulation is of infinite size, in which case automated proving of conditional bisimilarity becomes impossible. For instance, in Example 4.9, the least conditional bisimulation relating the two cospans u, r (representing (un)reliable channels) contains infinitely many triples $(u;m^n,\ r;m^n,\ \mathcal{A}_C)$ for any number n of messages on the right node ($m = I_0 \to I_r \leftarrow I_0$).

On the other hand, conditional bisimilarity is closed under contextualization, hence if u, r are related, we can conclude that $u;m$ and $r;m$ must be related as well. Intuitively the relation $R = \{(u, r, \mathcal{A}_C)\}$ is a sufficient witness, since after one step we reach the triple $(u;m,\ r;m,\ \mathcal{A}_C)$, from which we can "peel off" a common context m to obtain a triple already contained in R.

This is an instance of an *up-to technique*, which can be used to obtain smaller witness relations by identifying and removing redundant elements from a bisimulation relation. Instead of requiring the redundant triple $(u;m,\ r;m,\ \mathcal{A}_C)$ to be contained in the relation, it is sufficient to say that *up to* the passive context m, the triple is represented by $(u, r, \mathcal{A}_C)$, which is already contained in the relation. In particular, this specific up-to technique is known as *up-to context* [25], a well-known proof technique for process calculi.

Note that in general, a bisimulation up-to context is not a bisimulation relation. However, it can be converted into a bisimulation by closing it under all contexts.

In this section, we show how to adapt this concept to conditional bisimilarity and in particular discuss how to deal with the conditions in a conditional bisimulation up-to context.

5.1 Conditional Bisimilarity Up-To Context

We start our investigation of conditional bisimilarity up-to context with the idea of a relation that can be extended to a conditional bisimulation. To show, using such a conditional bisimulation up-to context R, that a pair of arrows is conditionally bisimilar, one cannot necessarily find the pair in R, but instead extends a pair in R to the pair under review. As this extension might provide parts of the context that the original condition referred to, it is necessary to shift the associated condition over the extension.

▶ **Definition 5.1** (Conditional bisimulation up-to context (CBUC)). *A symmetric conditional relation R is a* conditional bisimulation up-to context *if the following holds: for each triple $(a, b, \mathcal{C}) \in R$ and each context step $a \xrightarrow{f,\mathcal{A}}_C a'$, there are answering steps $b \xrightarrow{f,\mathcal{B}_i}_C b'_i$, $i \in I$, and conditions $\mathcal{C}''_i$ such that for each $i \in I$ there exists $(a''_i, b''_i, \mathcal{C}''_i) \in R$ with $a' = a''_i;j_i$, $b'_i = b''_i;j_i$ for some arrow j_i and additionally $\mathcal{A} \wedge \mathcal{C}_{\downarrow f} \models \bigvee_{i \in I} (\mathcal{C}''_{i\downarrow j_i} \wedge \mathcal{B}_i)$.*

$$\begin{tikzcd}
 & & \mathcal{D} & & \\
 & 0 \arrow[r, "\ell"] \arrow[d, "a"'] & I \arrow[d, "c"] & 0 \arrow[l, "r"'] \arrow[d, "a''_i"] \arrow[dl, "a'"] & \\
\mathcal{C} & J \arrow[r, "f"] & K & J' \arrow[l, "j_i"'] & \mathcal{C}''_i \\
 & 0 \arrow[u, "b"] \arrow[r, "\ell_i"'] & I_i \arrow[u, "e_i"] & 0 \arrow[l, "r_i"] \arrow[u, "b''_i"'] \arrow[ul, "b'_i"] & \\
 & & \mathcal{D}_i & &
\end{tikzcd}$$

Figure 5 A single answer step in conditional bisimulation up-to context.

The situation for one answer step is depicted in Figure 5. The conditions $\mathcal{A}, \mathcal{B}_i$ over K are not shown in the diagram. The weakest possible $\mathcal{A}, \mathcal{B}_i$ can be derived from the rule conditions as $\mathcal{A} = \mathcal{D}_{\downarrow c}$, $B_i = \mathcal{D}_{i\downarrow e_i}$.

Compared to a regular conditional bisimulation, which directly relates the results of the answering steps $(a', b'_i, \mathcal{C}'_i)$, in a CBUC it is sufficient to relate some pair $(a''_i, b''_i, \mathcal{C}''_i)$, where a''_i, b''_i are obtained from a', b'_i by removing an identical context j_i.

We now show that this up-to technique is useful or *sound*, that is, all elements recognized as bisimilar by the up-to technique are actually bisimilar [26, 25].

▶ **Theorem 5.2** (Characterization of CBUC). *A symmetric conditional relation R satisfies Definition 5.1 (is a CBUC) iff its closure under contextualization $\hat{R} := \{(a;d, b;d, \mathcal{C}_{\downarrow d}) \mid (a, b, \mathcal{C}) \in R,\ a, b \colon 0 \to J,\ d \colon J \to K\}$ is a conditional bisimulation.*

▶ Remark 5.3. From Theorem 5.2 we easily obtain as a corollary that every CBUC R is contained in $\stackrel{\circ}{\sim}_C$ ($R \subseteq \stackrel{\circ}{\sim}_C$), i.e. all elements contained in some CBUC are indeed conditionally bisimilar. This follows from the fact that $R \subseteq \hat{R}$ (set $d = \mathrm{id}_J$) and $\hat{R} \subseteq \stackrel{\circ}{\sim}_C$ (since by Theorem 5.2 $\hat{R}$ is a conditional bisimulation).

Note that while Theorem 5.2 gives a more accessible definition of CBUCs than Definition 5.1, the latter definition is amenable to mechanization, since R might be finite, whereas $\hat{R}$ is infinite.

5.2 Conditional Bisimilarity Up-To Context with Representative Steps

CBUCs allow us to represent certain infinite bisimulation relations in a finite way. For instance, we can use a finite CBUC in Example 4.9. However, automated checking for conditional bisimilarity up-to context is still hard, since all possible context steps have to be checked, of which there can be infinitely many.

For conditional bisimulations, we introduced an alternative definition using representative steps (Definition 4.7) and showed that it yields an equivalent notion of conditional bisimilarity (Theorem 4.8). We will show that the same approach can be used for CBUCs.

▶ **Definition 5.4** (CBUC with representative steps). *A* CBUC with representative steps *is a symmetric conditional relation R such that the following holds: for each triple $(a, b, \mathcal{C}) \in R$ and each representative step $a \xrightarrow{f,\mathcal{A}}_R a'$, there are answering steps $b \xrightarrow{f,\mathcal{B}_i}_C b'_i$ and conditions $\mathcal{C}''_i$ such that for each answering step there exists $(a''_i, b''_i, \mathcal{C}''_i) \in R$ with $a' = a''_i;j_i$, $b'_i = b''_i;j_i$ for some arrow j_i per answering step, and additionally $\mathcal{A} \wedge \mathcal{C}_{\downarrow f} \models \bigvee_{i\in I} (\mathcal{C}''_{i\downarrow j_i} \wedge \mathcal{B}_i)$.*

▶ **Theorem 5.5.** *A conditional relation is a CBUC (Definition 5.1) if and only if it is a CBUC with representative steps (Definition 5.4).*

▶ **Example 5.6.** Consider again Examples 4.3 and 4.9. We have previously seen that it is possible to repeatedly borrow a message on the left-hand node and transfer it to the right-hand node, which leads to more and more received messages accumulating at the right-hand node. We now show that the two types of channels are conditionally bisimilar by showing that $R = \{(\emptyset \to R_0 \leftarrow I_0,\ \emptyset \to U_0 \leftarrow I_0,\ \mathcal{A}_C)\}$ is a CBUC, i.e. it satisfies Definition 5.4. We consider the same steps as in Example 4.9:

- R_0 can do a step using rule P_R by borrowing a message on the left node, with environment condition $\mathcal{A} = \text{true}$, and reduces to $a' = \emptyset \to R_r \leftarrow I_0$. U_0 can answer this step using P_U under $\mathcal{B}_i = \mathcal{A}_C$ (no noise) and reacts to $b'_i = \emptyset \to U_r \leftarrow I_0$.

  Now set $j_i = I_0 \to I_r \leftarrow I_0$, i.e. we consider the m-loop on the right node as irrelevant context. Then, using $a''_i = \emptyset \to R_0 \leftarrow I_0$, $b''_i = \emptyset \to U_0 \leftarrow I_0$, $\mathcal{C}''_i = \mathcal{A}_C$ we have $a' = a''_i;j_i$, $b'_i = b''_i;j_i$, and we find that the triple without the irrelevant context j_i, that is $(a''_i, b''_i, \mathcal{C}''_i)$ (which happens to be the same as our initial triple), is contained in R. As before, the implication $\mathcal{A} \wedge \mathcal{C}_{\downarrow f} \models \bigvee_{i\in I} (\mathcal{C}''_i \wedge \mathcal{B}_i)$ holds.
- Symmetrically, U_0 borrows a message on the left node and reacts to U_r under $\mathcal{A} = \mathcal{A}_C$. Analogously to the previous case and to Example 4.9, R_0 answers this step, using $\mathcal{C}''_i = \mathcal{A}_C$ and $j_i = I_0 \to I_r \leftarrow I_0$.
- Again, the remaining representative steps can be proven in an analogous way.

Note that instead of working with an infinite bisimulation, we now have a singleton.

6 Comparison and An Alternative Characterization

6.1 An Equivalent Characterization Based on Environment Steps

We will now give a more natural characterization of conditional bisimilarity, in order to justify Definitions 4.2 and 4.7. This alternative definition is more elegant since it characterizes $\overset{\circ}{\sim}_C$ as the largest conditional congruence that is a conditional environment bisimulation. On the other hand, this definition is not directly suitable for mechanization.

In [16], environment steps, which capture the idea that a reaction is possible under some *passive* context d, have been defined to obtain a more natural characterization of saturated bisimilarity. Unlike the borrowed context f, the passive context d does not participate in the reaction itself, but we refer to it to ensure that the application condition of the rule holds.

▶ **Definition 6.1** (Environment step [16]). *Let $\mathcal{R}$ be a set of reactive system rules and $a\colon 0 \to K,\ a'\colon 0 \to K,\ d\colon K \to J$ be arrows. We write $a \overset{d}{\rightsquigarrow} a'$ whenever there exists a rule $(\ell, r, \mathcal{B}) \in \mathcal{R}$ and an arrow c such that $a = \ell;c,\ a' = r;c$ and $c;d \models \mathcal{B}$.*

Environment steps and context steps are related: they can be transformed into each other. Furthermore saturated bisimilarity is the coarsest bisimulation relation over environment steps that is also a congruence [16]. We now give a characterization of conditional bisimilarity based on environment steps:

▶ **Definition 6.2** (Conditional environment congruence). *A symmetric conditional relation R is a* conditional environment bisimulation *if whenever $(a, b, \mathcal{C}) \in R$ and $a \overset{d}{\rightsquigarrow} a'$ for some $d \models \mathcal{C}$, then $b \overset{d}{\rightsquigarrow} b'$ and $(a', b', \mathcal{C}') \in R$ for some condition $\mathcal{C}'$ such that $d \models \mathcal{C}'$. We denote by $\overset{\circ}{\sim}_E$ the largest conditional environment bisimulation that is also a conditional congruence and call it* conditional environment congruence.

▶ **Theorem 6.3.** *Conditional bisimilarity and conditional environment congruence coincide, that is, $\overset{\circ}{\sim}_C = \overset{\circ}{\sim}_E$.*

6.2 Comparison to Other Equivalences

We conclude this section by considering $\overset{\circ}{\sim}_T := \{(a, b) \mid (a, b, \mathrm{true}) \in \overset{\circ}{\sim}_C\}$, a binary relation derived from conditional bisimilarity, which is ternary. Intuitively it contains pairs (a, b), where a, b are system states that behave equivalently in every possible context. We investigate how $\overset{\circ}{\sim}_T$ compares to other behavioural equivalences that also check for identical behaviour in all contexts. First, we consider saturated bisimilarity ($\sim_C$), which has been characterized in [16] as the coarsest relation which is a congruence as well as a bisimilarity:

▶ **Theorem 6.4.** *Saturated bisimilarity implies* true-*conditional bisimilarity ($\sim_C \subseteq \overset{\circ}{\sim}_T$). However,* true-*conditional bisimilarity does* not *imply saturated bisimilarity ($\overset{\circ}{\sim}_T \nsubseteq \sim_C$).*

For saturated bisimilarity, if a step of a is answered by b with multiple steps, all b'_i reached in this way must be saturated bisimilar to a' (that is, show the same behaviour even if the environment is later changed to one which did not allow the given b'_i to be reached). In fact, it was an explicit goal in the design of saturated bisimilarity to account for external modification of the environment.

On the other hand, for conditional bisimilarity, each b'_i is only required to be conditionally bisimilar to a' under the condition which allowed this particular answering step – that is, after a step, the environment is fixed (or, depending on the system, can only assume a subset of all possible environments, cf. Definition 6.2 and Theorem 6.3).

Next, we compare $\overset{\circ}{\sim}_T$ to id-congruence, the coarsest congruence contained in bisimilarity over the reaction relation $\rightsquigarrow$. It simply relates two agents whenever they are bisimilar in all contexts, i.e. $\sim_{\mathrm{id}} := \{(a, b) \mid \text{for all contexts } d,\ a;d,\ b;d \text{ are bisimilar wrt. } \rightsquigarrow\}$.

▶ **Theorem 6.5.** *It holds that* true-*conditional bisimilarity implies* id-*congruence ($\overset{\circ}{\sim}_T \subseteq \sim_{\mathrm{id}}$). However,* id-*congruence does* not *imply* true-*conditional bisimilarity ($\sim_{\mathrm{id}} \nsubseteq \overset{\circ}{\sim}_T$).*

Intuitively, true-conditional bisimilarity allows to observe whether some item is consumed and recreated (by including it in both sides of a rule) or whether it is simply required (using an existential rule condition, cf. Theorem 6.5). On the other hand, id-congruence does not recognize this and simply checks whether reactions are possible in the same set of contexts.

Hence we have $\sim_C \subsetneq \overset{\circ}{\sim}_T \subsetneq \sim_{\mathrm{id}}$, which implies that checking for identical behaviour in all contexts using conditional bisimilarity gives rise to a new kind of behavioural equivalence, which does not allow arbitrary changes to the environment (as $\sim_C$ does), yet allows distinguishing borrowed and passive context (which $\sim_{\mathrm{id}}$ does not). For two of those equivalences ($\sim_C$, $\overset{\circ}{\sim}_T$) we can mechanize bisimulation proofs.

7 Conclusion, Related and Future Work

As stated earlier, there are some scattered approaches to notions of behavioural equivalence that can be compared to conditional bisimilarity. The concept of behaviour depending on a context is also present in Larsen's PhD thesis [21]. There, the idea is to embed an LTS into an environment, which is modelled as an action transducer, an LTS that consumes transitions of the system under investigation – similar to CCS synchronization. He then defines environment-parameterized bisimulation by considering only those transitions that are consumed in a certain environment. In [15], Hennessy and Lin describe symbolic bisimulations in the setting of value-passing processes, where Boolean expressions restrict the interpretations for which one shows bisimilarity. Instead in [2], Baldan, Bracciali and Bruni propose bisimilarity on open systems, specified by terms with a hole or place-holder. Instead of imposing conditions on the environment, they restrict the components that are filling the holes.

In [11], Fitting studies a matrix view of unlabelled transition system, annotated by Boolean conditions. In [3] we have shown that such systems can alternatively be viewed as conditional transition systems, where activation of transitions depends on conditions of the environment and one can state the bisimilarity of two states provided that the environment meets certain requirements. This view is closely tied to featured transition systems, which have been studied extensively in the software engineering literature. The idea here is to specify system behaviour dependent on the features that are present in the product (see for instance [7] for simulations on featured transition systems).

Our contribution in this paper is to consider conditional bisimilarity based on contextualization in a rule-based setting. That is, system behaviour is specified by generic rewriting rules, system states can be composed with a context specifying the environment and we impose restrictions on those contexts. By viewing both system states and contexts as arrows of a category, we can work in the framework of reactive systems à la Leifer and Milner and define a general theory of conditional bisimilarity. While in [16] conditions were only used to restrict applicability of the rules and bisimilarity was checked for all contexts, we here additionally use conditions to establish behavioural equivalence only in specific contexts.

As future work we want to take a closer look at the logic that we used to specify conditions. Conditional bisimilarity is defined in a way that is largely independent of the kind of logic, provided that the logic supports Boolean operators and shift. It is unclear and worth exploring whether the logic considered by us is expressive enough to characterize all contexts that ensure bisimilarity of two given arrows.

Up-to techniques can be elegantly stated in a lattice-theoretical framework [24] and it is not difficult to reframe the results of Section 5 in this setting, using the notion of compatibility. This view might help to incorporate further optimizations into the up-to technique.

Furthermore, it is an open question whether there is an alternative characterization of the id-congruence of Theorem 6.5 that is amenable to mechanization.

We have already implemented label derivation and bisimulation checking in the borrowed context approach, see for instance [23], however without taking conditions into account. Our aim is to obtain an efficient implementation for the scenario described in this paper. Note that our conditions subsume first-order logic [6] and hence in order to come to terms with the undecidability of implication we have to resort to simpler conditions or use approximative methods.

One natural question is whether our results can be stated in a coalgebraic setting, since coalgebra provides a generic framework for behavioural equivalences. We have already studied a much simplified coalgebraic version of conditional systems (without considering contextualization) in [1], using coalgebras living in Kleisli categories. Reactive systems can also be viewed as coalgebras (see [4]). However, a combination of these features has not yet been considered as far as we know.

References

1 Jiří Adámek, Filippo Bonchi, Mathias Hülsbusch, Barbara König, Stefan Milius, and Alexandra Silva. A coalgebraic perspective on minimization and determinization. In *Proc. of FOSSACS '12*, pages 58–73. Springer, 2012. LNCS/ARCoSS 7213.

2 Paolo Baldan, Andrea Bracciali, and Roberto Bruni. Bisimulation by unification. In *Proc. of AMAST '02*, pages 254–270. Springer, 2002. LNCS 2422.

3 Harsh Beohar, Barbara König, Sebastian Küpper, and Alexandra Silva. Conditional transition systems with upgrades. In *Proc. of TASE '17 (Theoretical Aspects of Software Engineering)*. IEEE Xplore, 2017.

4 Filippo Bonchi. *Abstract Semantics by Observable Contexts*. PhD thesis, Università degli Studi di Pisa, Dipartimento di Informatica, May 2008.

5 Filippo Bonchi, Barbara König, and Ugo Montanari. Saturated semantics for reactive systems. In *Proc. of LICS '06*, pages 69–80. IEEE, 2006.

6 H.J. Sander Bruggink, Raphaël Cauderlier, Mathias Hülsbusch, and Barbara König. Conditional reactive systems. In *Proc. of FSTTCS '11*, volume 13 of *LIPIcs*. Schloss Dagstuhl – Leibniz Center for Informatics, 2011.

7 Maxime Cordy, Andreas Classen, Gilles Perrouin, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Simulation-based abstractions for software product-line model checking. In *Proc. of ICSE '12*, pages 672–682. IEEE, 2012.

8 Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation—part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*, chapter 3. World Scientific, 1997.

9 Hartmut Ehrig and Barbara König. Deriving bisimulation congruences in the DPO approach to graph rewriting. In *Proc. of FOSSACS '04*, pages 151–166. Springer, 2004. LNCS 2987.

10 Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph-grammars: An algebraic approach. In *14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*, pages 167–180, October 1973.

11 Melvin Fitting. Bisimulations and boolean vectors. In *Advances in Modal Logic*, volume 4, pages 1–29. World Scientific Publishing, 2002.

12 Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3,4):287–313, December 1996.

13 Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, October 2001.

14 Annegret Habel and Karl-Heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009.

15 Matthew Hennessy and Huimin Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.

16 Mathias Hülsbusch and Barbara König. Deriving bisimulation congruences for conditional reactive systems. In *Proc. of FOSSACS '12*, pages 361–375. Springer, 2012. LNCS/ARCoSS 7213.

17 Mathias Hülsbusch, Barbara König, Sebastian Küpper, and Lara Stoltenow. Conditional bisimilarity for reactive systems, 2020. arXiv:2004.11792. URL: `https://arxiv.org/abs/2004.11792`.

18 Ole Høgh Jensen and Robin Milner. Bigraphs and transitions. In *Proc. of POPL 2003*, pages 38–49. ACM, 2003.

19 Bartek Klin, Vladimiro Sassone, and Paweł Sobociński. Labels from reductions: towards a general theory. In *Proc. of CALCO '05*, pages 30–50. Springer, 2005. LNCS 3629.

20 Stephen Lack and Paweł Sobociński. Adhesive and quasiadhesive categories. *RAIRO – Theoretical Informatics and Applications*, 39(3):511–545, 2005.

21 Kim Guldstrand Larsen. *Context-Dependent Bisimulation between Processes*. PhD thesis, University of Edinburgh, 1986.

22 James J. Leifer and Robin Milner. Deriving bisimulation congruences for reactive systems. In *CONCUR 2000 — Concurrency Theory: 11th International Conference University Park, PA, USA, August 22–25, 2000 Proceedings*, pages 243–258. Springer Berlin Heidelberg, 2000.

23 Dennis Nolte. Automatischer Nachweis von Bisimulationsäquivalenzen bei Graphtransformationssystemen. Master's thesis, Universität Duisburg-Essen, November 2012.

24 Damien Pous. Complete lattices and up-to techniques. In *Proc. of APLAS '07*, pages 351–366. Springer, 2007. LNCS 4807.

25 Damien Pous and Davide Sangiorgi. Enhancements of the coinductive proof method. In Davide Sangiorgi and Jan Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*. Cambridge University Press, 2011.

26 Davide Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8(5):447–479, 1998.

27 Vladimiro Sassone and Paweł Sobociński. Reactive systems over cospans. In *Proc. of LICS '05*, pages 311–320. IEEE, 2005.

28 Paweł Sobociński. *Deriving process congruences from reaction rules*. PhD thesis, University of Aarhus, 2004.

A Fast Decision Procedure For Uniqueness of Normal Forms w.r.t. Conversion of Shallow Term Rewriting Systems

Masaomi Yamaguchi
Tohoku University, Sendai, Japan
masaomi.yamaguchi.t4@dc.tohoku.ac.jp

Takahito Aoto
Niigata University, Niigata, Japan
aoto@ie.niigata-u.ac.jp

Abstract

Uniqueness of normal forms w.r.t. conversion (UNC) of term rewriting systems (TRSs) guarantees that there are no distinct convertible normal forms. It was recently shown that the UNC property of TRSs is decidable for shallow TRSs (Radcliffe et al., 2010). The existing procedure mainly consists of testing whether there exists a counterexample in a finite set of candidates; however, the procedure suffers a bottleneck of having a sheer number of such candidates. In this paper, we propose a new procedure which consists of checking a smaller number of such candidates and enumerating such candidates more efficiently. Correctness of the proposed procedure is proved and its complexity is analyzed. Furthermore, these two procedures have been implemented and it is experimentally confirmed that the proposed procedure runs much faster than the existing procedure.

2012 ACM Subject Classification Theory of computation → Equational logic and rewriting

Keywords and phrases uniqueness of normal forms w.r.t. conversion, shallow term rewriting systems, decision procedure

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.11

Supplementary Material Our implementations as well as the details of the experiments can be found in the webpage `http://www.nue.ie.niigata-u.ac.jp/experiments/fscd20/`.

Funding This work is partially supported by JSPS KAKENHI No. 18K11158.

Acknowledgements Thanks are due to the anonymous reviewers for many valuable comments.

1 Introduction

A term rewriting systems (TRS for short) is a well-known model of computation, which plays many roles in equational deduction and formal verification. A key property of the computation of TRSs that it is non-deterministic, which enables flexible computations in TRSs as well as flexible transformations between TRSs and equational axioms. Due to the non-determinism in the computations, however, it is not always guaranteed that results of computations are unique. Thus, properties ensuring unique results of computations are important topics in the study of TRSs. The most well-known such a property is *confluence* (CR), meaning that two convertible terms are joinable. Less known such properties include *uniqueness of normal forms w.r.t. conversion* (UNC) meaning that there are no distinct convertible normal forms. The UNC property[1] of TRSs has been studied in e.g. [3, 4, 11, 12, 13, 19, 10, 20]. Furthermore, interests in automation of proving these properties initiated to start *Confluence Competition* [1, 2, 14] among software tools for proving such properties; there the category of the UNC property has been started from the 2016 edition of the Competition.

[1] The UNC property have been also studied under the name of UN or $\mathrm{UN}^{=}$. We use UNC, following the convention employed in the Confluence Competition.

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 11; pp. 11:1–11:23
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

One of the important topics concerning these properties is (un)decidability. It is known that the first-order theory of rewriting is decidable for left-linear right-ground TRSs [9]. Indeed, an implementation of the decision procedure of such theory have been reported in [18], and has been applied for (dis)proving these properties of left-linear and right-ground TRSs. An implementation of more efficient decision procedures of these properties for ground TRSs (a subclass of left-linear and right-ground TRSs) have been also reported in [11]. Another line of criteria for such (un)decidability is shallowness and flatness. Shallowness or flatness restricts the depth of (variable) occurrences in the rewrite rules. It is known that confluence is undecidable for flat (and hence shallow) TRSs [15]. In contrast, a polynomial algorithm for deciding the UNC property of linear shallow TRSs have been shown in [20], and it was recently shown that the UNC property of shallow TRSs is decidable [16, 17].

The existing procedure of [17] mainly consists of testing whether there exists a counterexample in a finite set of candidates. However, the procedure suffers a bottleneck of having a sheer number of such candidates even for small examples. In this paper, we propose a new procedure which reduces the number of such candidates to be checked and also enumerates such candidates more efficiently. The proposed procedure has the same structure and is based on the same ideas as the one of [17]; the difference is in the ways of checking the two main cases (whether or not there exists a counterexample to UNC in which the convertible normal forms are convertible to a constant in $\widehat{E_{\mathcal{R}}}$, a complete equations set for TRS $\mathcal{R}$ [8]).

The idea of the proposed method is to construct normal forms which can be reached by minimal constant expansion steps of $\widehat{E_{\mathcal{R}}}$. Based on this idea, we introduce constant propagation algorithm that incrementally constructs normal forms of each constant. Using this algorithm, we can determine whether there exists any minimal counterexample that is equivalent to a constant efficiently. If there exists no such a counterexample, we can check the UNC property efficiently by using the normal forms obtained by the algorithm.

We prove correctness of the proposed decision procedure and analyze its complexity. Furthermore, we implement two UNC decision procedures those based on existing method [17] and those based on proposed method, and experimentally confirm that proposed method runs much faster than existing one.

The rest of the paper is organized as follows: In Section 2, we present basic notions and notations used in this paper, and recall some preliminary backgrounds on our decision procedure. In addition, we overview the existing procedure [17]. In Section 3, we present our new decision procedure, together with its main ingredients – construction of two key sets CP_{NF} and CW – illustrating them through concrete examples. In Section 4, a notion of constant propagation class is introduced; it is used to show the correctness of checking the existence of a minimal witness that is equivalent to a constant by CP_{NF} in Section 5. Section 6 is devoted to show the correctness of checking the existence of a minimal witness that is *not* equivalent to a constant by CW. The correctness theorem and complexity analysis of our decision procedure are given in Section 7. In Section 8, we report our implementation and experiments. In Section 9, we conclude.

2 Preliminaries

In this section, we fix notations that will be used in this paper. Familiarity with term rewriting systems are assumed (see e.g. [6]).

2.1 Term rewriting systems

We denote by $\mathcal{V}$ a countably infinite set of *variables*, and by $\mathcal{F}$ the finite set of (arity-fixed) *function symbols*, which includes the set $\mathcal{C}$ of constants; variables are denoted by $x, y, z, \ldots$, function symbols by $f, g, h, \ldots$, and constants by $a, b, c, \ldots$. The set of terms is denoted by $\mathrm{T}(\mathcal{F},\mathcal{V})$ and the set of non-constant non-variable terms by $\mathrm{T}_f(\mathcal{F},\mathcal{V})$; they may be abbreviated to T and T_f, respectively. We define $\mathrm{height}(t) = 0$ for $t \in \mathcal{C} \cup \mathcal{V}$, and $\mathrm{height}(f(t_1,\dots,t_n)) = 1 + \max\{\mathrm{height}(t_1),\dots,\mathrm{height}(t_n)\}$ $(n \geq 1)$. The *size* of a term t, denoted by $|t|$, is 1 if $t \in \mathcal{V}$, and is $1 + \sum_{i=1}^{n} |t_i|$ if $t = f(t_1,\dots,t_n)$. The set of variables in a term t is denoted by $\mathcal{V}(t)$. The *root symbol* of a term t is denoted by $\mathrm{root}(t)$.

A *substitution* σ is a mapping $\sigma\colon \mathcal{V} \to \mathrm{T}(\mathcal{F},\mathcal{V})$ such that the set $\mathrm{dom}(\sigma) = \{x \mid \sigma(x) \neq x\}$ is finite. When $\mathrm{dom}(\sigma) \subseteq \{x_1,\dots,x_n\}$, we also write it as $\{x_1 := \sigma(x_1),\dots,x_n := \sigma(x_n)\}$. A substitution is identified with its homomorphic extension; we write $t\sigma$ for $\sigma(t)$. We write $s \leqslant t$ if $\sigma(s) = t$ for some substitution σ. A renaming substitution is a substitution that is a permutation on variables (i.e. a bijective mapping from $\mathcal{V}$ to $\mathcal{V}$); renaming substitutions are denoted by $\sigma_\alpha, \rho_\alpha, \dots$. The symbol $\mathbb{N}$ ($\mathbb{N}^*$) stands for the set of (resp. the finite sequences of) natural numbers. We denote the set of *(variable) positions* in a term t by $\mathrm{Pos}(t)$ (resp. $\mathrm{Pos}_{\mathcal{V}}(t)$). The *root position* is denoted by ϵ and the *subterm* at a position p by $t|_p$. A subterm $t|_p$ is a *direct* (*variable*) subterm of t if $p \in \mathbb{N}$ (resp. $t|_p \in \mathcal{V}$). A *hole* is a special constant, denoted by $\Box$. A *context* is a term containing exactly one hole. For a context C and a term t, we denote by $C[t]$ the term obtained by replacing the hole in C by t. A context C is also written as $C[\,]$. Especially, we write $C[\,]_p$ to specify the position of the hole in $C[\,]$. We write $t[\,]_p$ to denote the context obtained by substituting the hole at the position p in a term t.

A *rewrite rule* $l \to r$ satisfies $l \notin \mathcal{V}$; we don't assume, however, the other usual variable restriction $\mathcal{V}(r) \subseteq \mathcal{V}(l)$ in this paper. A *term rewriting system* (*TRS*) is given by $\langle \mathcal{F},\mathcal{R}\rangle$ where $\mathcal{R}$ is a finite set of rewrite rules over $\mathcal{F}$. When $\langle \mathcal{F},\mathcal{R}\rangle$ is abbreviated to $\mathcal{R}$, some appropriate $\mathcal{F}$ is fixed. Let $\mathcal{R}$ be a TRS. If there exist $l \to r \in \mathcal{R}$, a substitution σ and a context $C[\,]_p$ such that $s = C[l\sigma]_p$ $t = C[r\sigma]_p$, we have a *rewrite step* $s \to_{\mathcal{R}} t$. The subscript $\mathcal{R}$ may be abbreviated when it is clear from the context. When we need to make (some of) $p, l \to r, \sigma$ explicit, we write $s \to^{p,l\to r,\sigma} t$, etc. A rewrite step $s \to^{p} t$ is *root* if $p = \epsilon$. A non-root rewrite step is denoted by $s \to^{>\epsilon} t$. A term s is a *normal form* if $s \to t$ for no t; the set of normal forms is denoted by *NF*. The symmetric closure of $\to$ is denoted by $\leftrightarrow$, its transitive closure by $\xrightarrow{+}$, its reflexive transitive closure by $\xrightarrow{*}$. its equivalence closure by $\xleftrightarrow{*}$. A successive composition of rewrite steps $s_1 \to \cdots \to s_n$ is called a *rewrite sequence*, which may be abbreviated as $s_1 \xrightarrow{*} s_n$. These notations are reused for other similar relations as well and could be combined. Terms s and t are *convertible* if $s \xleftrightarrow{*} t$. A TRS $\mathcal{R}$ satisfies *uniqueness of normal forms w.r.t. conversion* (UNC) if there are no convertible distinct normal forms, i.e. $s \xleftrightarrow{*} t$ with $s,t \in \mathit{NF}$ implies $s = t$. A finite set of equations is called an equational system (ES for short). We identify equations $l \approx r$ and $r \approx l$. A rewrite step $s \leftrightarrow_E t$ by an equation $l \approx r \in E$ is defined in the same way as for a rewrite rule. For a TRS $\mathcal{R}$, the associated ES $\{l \approx r \mid l \to r \in \mathcal{R}\}$ is denoted by $E_{\mathcal{R}}$.

2.2 UNC of shallow TRSs

A term t is *shallow* if $\mathrm{Pos}_{\mathcal{V}}(t) \subseteq \{\epsilon\} \cup \mathbb{N}$, i.e. $t \in \mathcal{C} \cup \mathcal{V}$ or t contains a variable only as a direct subterm. For example, terms $x, \mathsf{a}, \mathsf{g}(y), \mathsf{f}(x,\mathsf{g}(\mathsf{a}))$ are shallow but $\mathsf{f}(x,\mathsf{g}(y))$ is not. A TRS $\mathcal{R}$ is shallow if l, r are shallow for all $l \to r \in \mathcal{R}$. In [17], a decision procedure for the UNC property of shallow TRSs is given. We now explain some crucial characterizations of UNC, some notions and notations in [17] that will be also used in our decision procedure.

The first step of our decision procedure, as well as that of [17], is to translate shallow TRSs to flat TRSs. A term t is *flat* if $\mathit{height}(t) \leq 1$; a TRS $\mathcal{R}$ is flat if l, r are flat for all $l \to r \in \mathcal{R}$. Clearly, flat TRSs are shallow. On the other hand, a term $\mathsf{f}(x,\mathsf{g}(\mathsf{a}))$ is shallow but not flat. It is known that one can transform shallow TRSs into flat TRSs preserving (non-)UNC; we refer details to [20].

▶ **Example 1.** Let $\mathcal{R}_{shallow} = \{\mathsf{f}(x,y) \to \mathsf{g}(\mathsf{h}(\mathsf{a}))\}$. As $\mathit{height}(\mathsf{g}(\mathsf{h}(\mathsf{a}))) = 2$, the TRS $\mathcal{R}_{shallow}$ is not flat. Now, by the UNC-preserving flatting translation [20], we obtain a flat TRS $\mathcal{R} = \{\mathsf{b} \to \mathsf{h}(\mathsf{a}), \mathsf{f}(x,y) \to \mathsf{g}(\mathsf{b})\}$ from $\mathcal{R}_{shallow}$. Here, b is a newly introduced constant.

Our procedure, as well as the one of [17], employs this transformation. *Henceforth, we focus on flat TRSs.*

The *pattern* of direct subterms in a term $t = f(t_1, \dots, t_n) \in \mathrm{T}_f$, denoted by $\mathit{Patt}(t)$, is the set $\{\{i,j\} \mid 1 \leq i,j \leq n, i \neq j, t|_i = t|_j\}$; its subset $\{\{i,j\} \in \mathit{Patt}(t) \mid t|_i \in \mathcal{V}\}$ is denoted by $\mathit{Patt}_{\mathcal{V}}(t)$. The following property of root rewrite steps of flat TRSs will be heavily used:

▶ **Lemma 2** ([17]). *Let $\mathcal{R}$ be a flat TRS and t a term. Then, $t \to_{\mathcal{R}}^{\epsilon, l \to r} t'$ for some t' iff (1)* $\mathrm{root}(l) = \mathrm{root}(t)$*, (2) $l|_i = t|_i$ for all $i \in \mathbb{N}$ with $l_i \in \mathcal{C}$, and (3) $\mathit{Patt}_{\mathcal{V}}(l) \subseteq \mathit{Patt}(t)$.*

An ES is flat if so are all equations in it. An important ingredient of the decision procedure is the *completion* of flat ESs [8]: Given a flat ES E, one can construct a closure $\widehat{E}$ of E with respect to the following rules:

$$\frac{g \approx d, l \approx r}{d\sigma \approx r\sigma} \quad \text{if } l, g \notin \mathcal{V},\ \sigma = \mathrm{mgu}(l, g) \tag{1}$$

$$\frac{l \approx d, y \approx r}{d \approx r\sigma} \quad \text{if } y \in \mathcal{V}, l \in \mathcal{C} \cup \mathcal{V}, \sigma = \{y := l\} \tag{2}$$

$$\frac{C[a] \approx d, a \approx b}{C[b] \approx d} \quad \text{if } a, b \in \mathcal{C} \tag{3}$$

Here, mgu stands for a most general unifier. Then, $\widehat{E}$ is a flat ES that is equivalent to E (i.e. $\overset{*}{\leftrightarrow}_E = \overset{*}{\leftrightarrow}_{\widehat{E}}$) and is ground complete w.r.t. the ordered rewriting [7]. We won't go into the detail of the latter property, but remark that only we concern in this paper is that, from the latter property, for any given terms s, t, it is decidable whether $s \overset{*}{\leftrightarrow}_{\widehat{E}} t$ holds. Our procedure, as well as the one of [17], heavily uses the completion $\widehat{E_{\mathcal{R}}}$ of $E_{\mathcal{R}}$ $(= \{l \approx r \mid l \to r \in \mathcal{R}\})$.

▶ **Example 3.** Let $\mathcal{R} = \{\mathsf{a} \to \mathsf{b}, \mathsf{a} \to \mathsf{f}(x,\mathsf{c},\mathsf{d}), \mathsf{c} \to \mathsf{g}(\mathsf{d}), \mathsf{h}(\mathsf{a}) \to \mathsf{d}, \mathsf{e} \to \mathsf{g}(\mathsf{e})\}$. Then, for example, one obtains $\widehat{E_{\mathcal{R}}} = \{\mathsf{a} \approx \mathsf{b}, \mathsf{a} \approx \mathsf{f}(x,\mathsf{c},\mathsf{d}), \mathsf{b} \approx \mathsf{f}(x,\mathsf{c},\mathsf{d}), \mathsf{f}(x,\mathsf{c},\mathsf{d}) \approx \mathsf{f}(y,\mathsf{c},\mathsf{d}), \mathsf{c} \approx \mathsf{g}(\mathsf{d}), \mathsf{h}(\mathsf{a}) \approx \mathsf{d}, \mathsf{h}(\mathsf{b}) \approx \mathsf{d}, \mathsf{h}(\mathsf{a}) \approx \mathsf{h}(\mathsf{b}), \mathsf{e} \approx \mathsf{g}(\mathsf{e})\}$. It is decidable whether $s \overset{*}{\leftrightarrow}_{\widehat{E_{\mathcal{R}}}} t$ (equivalently, $s \overset{*}{\leftrightarrow}_{\mathcal{R}} t$) for any given terms s, t.

We have one further point to explain about the use of $\widehat{E_{\mathcal{R}}}$ in the decision procedures. A TRS $\mathcal{R}$ (or an ES E) is *inconsistent* if $x \overset{*}{\leftrightarrow}_{\mathcal{R}} y$ (resp. $x \overset{*}{\leftrightarrow}_E y$) for some distinct variables x, y; it is *consistent* if it is not inconsistent. Clearly, UNC of a TRS $\mathcal{R}$ implies consistency of $\mathcal{R}$. It is also easy to see that a TRS (or an ES) is inconsistent iff there exists a term t convertible to $x \in \mathcal{V} \setminus \mathcal{V}(t)$. For a flat ES E, this characterization can be strengthened as follows [8]: E is inconsistent iff there exists $x \approx t \in \widehat{E}$ such that $x \notin \mathcal{V}(t)$. Thus, one can check whether $\mathcal{R}$ is consistent using $\widehat{E_{\mathcal{R}}}$.

In the beginning of the decision procedures, one computes a completion $\widehat{E_{\mathcal{R}}}$ of $E_{\mathcal{R}}$. Then, one checks if there exists an equation $x \approx t \in \widehat{E_{\mathcal{R}}}$ such that $x \notin \mathcal{V}(t)$. If this is the case, one knows that $\widehat{E_{\mathcal{R}}}$ is inconsistent, and hence so is $\mathcal{R}$. As inconsistency implies non-UNC, one can conclude $\mathcal{R}$ is not UNC. Thus, the rest of the procedure only deals with the case $\mathcal{R}$ is consistent. For this reason, *we focus on the case that $\mathcal{R}$ is consistent in Sections 4–6.*

The following properties are easily obtained using the definition of $\widehat{E}$ [8]. These properties will be used in subsequent sections without mentioning.

▶ **Proposition 4** ([8]). *Let E be a consistent flat ES, and $\widehat{E}$ its completion. (1) If $s \overset{*}{\leftrightarrow}_E t$, then there exists a rewrite sequence $s \overset{*}{\leftrightarrow}_{\widehat{E}} t$ that has at most one root rewrite step. (2) For any $c, c' \in \mathcal{C}$ with $c \neq c'$, $c \overset{*}{\leftrightarrow}_{\widehat{E}} c'$ iff $c \approx c' \in \widehat{E}$. (3) Suppose $c \overset{*}{\leftrightarrow}_{\widehat{E}} c'$. Then, $c \approx r \in \widehat{E}$ iff $c' \approx r \in \widehat{E}$.*

As we focus on the UNC property of $\mathcal{R}$, a pair $\langle s, t\rangle$ of distinct normal forms s, t such that $s \overset{*}{\leftrightarrow}_{\mathcal{R}} t$ is called a *witness* (of non-UNC). We call a witness $\langle s, t\rangle$ is *equivalent to a constant* c if $s \overset{*}{\leftrightarrow} c$ and $t \overset{*}{\leftrightarrow} c$. If $|s| + |t|$ is minimal among all witnesses, $\langle s, t\rangle$ is a *minimal witness*. The set of all subterms of any minimal witness is denoted by $\mathit{SubMinWit}_{\mathcal{R}}$. Clearly, $\mathcal{R}$ has UNC property iff there exists no (minimal) witness. It is not at all easy to see what actually $\mathit{SubMinWit}_{\mathcal{R}}$ is, but it satisfies the following useful claims that will be used later.

▶ **Proposition 5** ([17]). *Let $f(t_1, \ldots, t_n) \in \mathit{SubMinWit}_{\mathcal{R}}$. If $t_i \overset{*}{\leftrightarrow} t_j$ then $t_i = t_j$.*

▶ **Lemma 6.** *Let $f(t_1, \ldots, t_n) \in \mathit{SubMinWit}_{\mathcal{R}}$. If $t_i \overset{*}{\leftrightarrow} c \in \mathcal{C} \cap \mathit{NF}$ then $t_i = c$.*

2.3 Existing Decision Procedure

Here, we briefly describe the decision procedure of [17]. It is based on these two lemmas:

▶ **Lemma 7** ([17]). *Let $\mathcal{R}$ be a TRS. One can add a finite number of constants to $\mathcal{R}$ to get the TRS $\mathcal{R}'$ which meets the following condition: a witness exists in $\mathcal{R}$ iff a ground witness exists in $\mathcal{R}'$.*

▶ **Lemma 8** ([17]). *Let $\mathcal{R}$ be a flat TRS. If there exists a witness, there exists a witness $\langle s, t\rangle$ such that* $\mathrm{height}(s), \mathrm{height}(t) \leq \max(1, |\mathcal{C}|)$.

Think of a given shallow TRS $\mathcal{R}_{shallow}$. As we explained above, one can get a flat TRS $\mathcal{R}$ from $\mathcal{R}_{shallow}$ preserving (non-)UNC, and its completion $\widehat{E_{\mathcal{R}}}$. From Lemmas 7 and 8, we simply need to check there exists a ground witness $\langle s, t\rangle$ s.t. $\mathrm{height}(s), \mathrm{height}(t) \leq \max(1, |\mathcal{C}|)$, adding a finite number of constants to $\mathcal{R}$. Since there are only finitely many ground terms that satisfy height $\leq \max(1, |\mathcal{C}|)$, one can construct all of them. Thus, it remains to check there exists any pair of such terms that is a witness – this can be decided using $\mathcal{R}$ (whether its components are normal forms) and $\widehat{E_{\mathcal{R}}}$ (whether it consists of convertible terms).

3 New Decision Procedure

In this section, we describe our new decision procedure for the UNC property of shallow TRSs and motivate later sections where we prove its correctness.

3.1 The Whole Procedure

Below, the rewrite step $\leftrightarrow$ of $\widehat{E_{\mathcal{R}}}$ will be abbreviated as $\leftrightarrow$ and NF denotes the set of normal forms w.r.t. $\to_{\mathcal{R}}$.

The whole decision procedure is given in Figure 1. Apart from the same part as the existing procedure (Step 1) and simple checking, the procedure contains two main ingredients – construction of the set $\mathit{CP}_{\mathit{NF}}$ and that of the set CW. We are going to explain the details of these constructions shortly. Actually, these two steps are closely related to the correctness proofs of the previous algorithm [17]. In [17], the authors divide the UNC problem into two main cases according to whether there exists a witness equivalent to a constant. The Step 3 checks whether there exists such a witness and the Step 4 checks whether there exists a witness that is not equivalent to any constant.

Input: a shallow TRS
Output: UNC or Non-UNC

Step 1 Transform the input shallow TRS into a flat TRS $\mathcal{R}$ preserving the UNC property, and calculate its completion $\widehat{E_{\mathcal{R}}}$. If $\mathcal{R}$ is inconsistent (this can be detected when calculating $\widehat{E_{\mathcal{R}}}$), then return Non-UNC.
Step 2 Calculate CP_{NF} by the Constant Propagation Algorithm.
Step 3 If there exists $(\hat{c}, \hat{r}, t, h) \in CP_{NF}$ such that t has a direct variable subterm, then return Non-UNC. If there exist $(\hat{c}, \hat{r}_s, s, h_s), (\hat{c}, \hat{r}_t, t, h_t) \in CP_{NF}$ such that $s \neq t$, then return Non-UNC.
Step 4 Calculate CW. If there exist $\langle s, t \rangle \in CW$ such that $s \neq t$ and $s, t \in NF$, then return Non-UNC.
Step 5 Return UNC.

Figure 1 Proposed decision procedure for UNC of shallow TRSs.

3.2 Constant Propagation Algorithm

Here, we describe how to construct CP_{NF} by Constant Propagation Algorithm, which determine whether there exists any minimal counterexample that is equivalent to a constant.

We first need a couple of notion and notation.

▶ **Definition 9** (equivalence relation $\simeq$ on flat terms). *We define an equivalence relation $\simeq$ on flat terms like this: $s \simeq t$ iff either (1) $s, t \in \mathcal{V}$ and $s = t$, (2) $s, t \in \mathcal{C}$ and $s \overset{*}{\leftrightarrow} t$, or (3) $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_n)$ $(n \geq 1)$ such that (a) $s_i \in \mathcal{C}$ iff $t_i \in \mathcal{C}$ for all $1 \leq i \leq n$, (b) $s_i \overset{*}{\leftrightarrow} t_i$ for all $s_i \in \mathcal{C}$, and (c) there exists a renaming substitution σ_α such that $\sigma_\alpha(s_i) = t_i$ for all $s_i \in \mathcal{V}$.*

It is easy to see that $\simeq$ is indeed an equivalence relation. The $\simeq$-equivalence class of a flat term t is denoted by $[\![t]\!]$.

▶ **Definition 10.** *We fix a representative element of $[\![c]\!]$ $(c \in \mathcal{C})$ and denote it by $\hat{c}$. We denote the set $\{\hat{c} \mid c \in \mathcal{C}\}$ by $\widehat{\mathcal{C}}$. For an arbitrary flat term t, we define $\hat{x} = x$ and $\hat{t} = f(\hat{t}_1, \ldots, \hat{t}_n)$ for $t = f(t_1, \ldots, t_n)$, in addition.*

The idea of our algorithms comes from the observation that, for any term t which is equivalent to a constant c, we have $t \overset{*}{\to} c$ by the ordered rewriting of $\widehat{E_{\mathcal{R}}}$, and c (or maybe another constant that is equivalent to c) is a $\widehat{E_{\mathcal{R}}}$-normal form of t. Our algorithm incrementally searches normal forms of each constant, tracing the inverse direction of the $\widehat{E_{\mathcal{R}}}$-rewriting sequences.

▶ **Definition 11** (Constant Propagation (CP) Algorithm). *Suppose a flat TRS $\mathcal{R}$ and its completion $\widehat{E_{\mathcal{R}}}$ are given. The algorithm incrementally computes a set of quadruples CP_{NF} by the pseudo-code presented in Figure 2.*

Actually, the fourth element of quadruples is unused to compute the result; the sole purpose of adding auxiliary parameter H is to use it in our proof below.

▶ **Example 12.** Let $\mathcal{R}$ and $\widehat{E_{\mathcal{R}}}$ be given as in Example 3. The constant propagation algorithm runs as follows:

```
Input: a flat TRS $\mathcal{R}$ and its completion $\widehat{E_{\mathcal{R}}}$
Output: $CP_{NF}$

Step 1: $H := 0$;   $\widehat{\mathcal{C}} := \{\hat{c}_1, \ldots, \hat{c}_m\}$
        For each $i = 1, \ldots, m$ :  $Y_{\hat{c}_i} := \{\hat{r} \mid r \in \mathrm{T}_f,\ \hat{c}_i \approx r \in \widehat{E_{\mathcal{R}}}\}$
Step 2: $CP_{NF} := \{(\hat{c}, \hat{c}, c, 0) \mid c \in \mathcal{C} \cap NF\}$
Step 3: Repeat the following (main loop):
    $H := H + 1$;   $X_{tmp} := \emptyset$
    Calculate a function $\chi_H : \widehat{\mathcal{C}} \cup \mathcal{V} \to \mathcal{P}(\mathrm{T})$ as follows :

        $\chi_H(\hat{c}) = \begin{cases} \{u \mid (\hat{c}, \_, u, \_) \in CP_{NF}\} & \text{if } \exists(\hat{c}, \_, u, \_) \in CP_{NF} \\ \{\hat{c}\} & \text{otherwise} \end{cases}$

        $\chi_H(x) = \{x\}$

    For each $i = 1, \ldots, m$, calculate as follows :
        For each $\hat{r} = f(\hat{u}_1, \ldots, \hat{u}_n) \in Y_{\hat{c}_i}$, calculate as follows :
            $X_{i,\hat{r}} := \left\{ (\hat{c}_i, \hat{r}, f(u'_1, \ldots, u'_n), H) \;\middle|\; \begin{array}{l} u'_j \in \chi_H(\hat{u}_j)\ (1 \leq j \leq n) \\ f(u'_1, \ldots, u'_n) \in NF \end{array} \right\}$
            $Y_{\hat{c}_i} := \begin{cases} Y_{\hat{c}_i} \backslash \{\hat{r}\} & \text{if } X_{i,\hat{r}} \neq \emptyset \\ Y_{\hat{c}_i} & \text{otherwise} \end{cases}$
            $X_{tmp} := X_{i,\hat{r}} \cup X_{tmp}$
    $CP_{NF} := X_{tmp} \cup CP_{NF}$
    If $X_{tmp} = \emptyset$, exit the main loop
```

Figure 2 Constant Propagation Algorithm.

1. Choose a representative element among convertible constants. As we have $\mathsf{a} \overset{*}{\leftrightarrow} \mathsf{b}$, let us pick up a as their representative, i.e. $\hat{\mathsf{a}} = \hat{\mathsf{b}} = \mathsf{a}$ (picking b leads no problem). Since there are no other distinct convertible constants, we have $\widehat{\mathcal{C}} = \{\mathsf{a}, \mathsf{c}, \mathsf{d}, \mathsf{e}\}$. Thus, we set $Y_{\mathsf{a}} = \{\mathsf{f}(x, \mathsf{c}, \mathsf{d})\}$, $Y_{\mathsf{c}} = \{\mathsf{g}(\mathsf{d})\}$, $Y_{\mathsf{d}} = \{\mathsf{h}(\mathsf{a})\}$ and $Y_{\mathsf{e}} = \{\mathsf{g}(\mathsf{e})\}$ in Step 1.
2. In Step 2, as $\mathcal{C} \cap NF = \{\mathsf{b}, \mathsf{d}\}$, we initialize $CP_{NF} := \{(\mathsf{a}, \mathsf{a}, \mathsf{b}, 0), (\mathsf{d}, \mathsf{d}, \mathsf{d}, 0)\}$. Intuitively, this expresses that a term b (d) is one of the convertible normal forms of the constant a (resp. d).
3. Now we run into the first loop of the Step 3. We have $\chi_1(\mathsf{a}) = \{\mathsf{b}\}$ and $\chi_1(x) = \{x\}$ for $x \neq \mathsf{a}$. Now, we check whether this replacement mapping χ_1 can make elements of $Y_{\mathsf{a}} \cup Y_{\mathsf{c}} \cup Y_{\mathsf{d}} \cup Y_{\mathsf{e}}$ a normal form. Then, we find that $\mathsf{g}(\mathsf{d}) \in NF$ is obtained from $\mathsf{g}(\mathsf{d}) \in Y_{\mathsf{c}}$ and $\mathsf{h}(\mathsf{b}) \in NF$ is obtained from $\mathsf{h}(\mathsf{a}) \in Y_{\mathsf{d}}$. Thus, we updates the sets as: $CP_{NF} := CP_{NF} \cup \{(\mathsf{c}, \mathsf{g}(\mathsf{d}), \mathsf{g}(\mathsf{d}), 1), (\mathsf{d}, \mathsf{h}(\mathsf{a}), \mathsf{h}(\mathsf{b}), 1)\}$, $Y_{\mathsf{c}} := \emptyset$ and $Y_{\mathsf{d}} := \emptyset$. Intuitively, in this step, we found a normal form $\mathsf{g}(\mathsf{d})$ ($\mathsf{h}(\mathsf{b})$) equivalent to a constant c (resp. d).
4. Now we run into the second loop of the Step 3. We have $\chi_2(\mathsf{a}) = \{\mathsf{b}\}$, $\chi_2(\mathsf{c}) = \{\mathsf{g}(\mathsf{d})\}$, $\chi_2(\mathsf{d}) = \{\mathsf{d}, \mathsf{h}(\mathsf{b})\}$ and $\chi_2(x) = \{x\}$ for $x \notin \{\mathsf{a}, \mathsf{c}, \mathsf{d}\}$. Again, we check whether this replace mapping can make remaining elements of $Y_{\mathsf{a}} \cup Y_{\mathsf{e}}$ a normal form. Then, we find that $\mathsf{f}(x, \mathsf{g}(\mathsf{d}), \mathsf{d}), \mathsf{f}(x, \mathsf{g}(\mathsf{d}), \mathsf{h}(\mathsf{b})) \in NF$ are obtained from $\mathsf{f}(x, \mathsf{c}, \mathsf{d}) \in Y_{\mathsf{a}}$. Thus, we update the sets as: $CP_{NF} := CP_{NF} \cup \{(\mathsf{a}, \mathsf{f}(x, \mathsf{c}, \mathsf{d}), \mathsf{f}(x, \mathsf{g}(\mathsf{d}), \mathsf{d}), 2), (\mathsf{a}, \mathsf{f}(x, \mathsf{c}, \mathsf{d}), \mathsf{f}(x, \mathsf{g}(\mathsf{d}), \mathsf{h}(\mathsf{b})), 2)\}$ and $Y_{\mathsf{a}} := \emptyset$.
5. The third round of the loop of the Step 3 finds no new normal forms, and $X_{tmp} = \emptyset$.

Thus, we exit the loop.
Finally, we have $CP_{NF} =$

$$\left\{\begin{array}{lll} (\mathsf{a},\mathsf{a},\mathsf{b},0), & (\mathsf{a},\mathsf{f}(x,\mathsf{c},\mathsf{d}),\mathsf{f}(x,\mathsf{g}(\mathsf{d}),\mathsf{d}),2), & (\mathsf{a},\mathsf{f}(x,\mathsf{c},\mathsf{d}),\mathsf{f}(x,\mathsf{g}(\mathsf{d}),\mathsf{h}(\mathsf{b})),2), \\ (\mathsf{c},\mathsf{g}(\mathsf{d}),\mathsf{g}(\mathsf{d}),1), & (\mathsf{d},\mathsf{d},\mathsf{d},0), & (\mathsf{d},\mathsf{h}(\mathsf{a}),\mathsf{h}(\mathsf{b}),1) \end{array}\right\}.$$

Observe that, for any quadruples $(c, r, u, h) \in CP_{NF}$, we have $c \overset{*}{\leftrightarrow} u$ and $u \in NF$. Thus, from the final CP_{NF}, one easily see some witnesses equivalent to a constant: e.g. $\langle \mathsf{d}, \mathsf{h}(\mathsf{b})\rangle$ (equivalent to d), $\langle \mathsf{b}, \mathsf{f}(x, \mathsf{g}(\mathsf{d}), \mathsf{d})\rangle$ and $\langle \mathsf{f}(x, \mathsf{g}(\mathsf{d}), \mathsf{d}), \mathsf{f}(x, \mathsf{g}(\mathsf{d}), \mathsf{h}(\mathsf{b}))\rangle$ (equivalent to a). One also obtains a witness $\langle \mathsf{f}(x, \mathsf{g}(\mathsf{d}), \mathsf{d}), \mathsf{f}(y, \mathsf{g}(\mathsf{d}), \mathsf{d})$ (equivalent to a), since renaming x to y leads $\mathsf{f}(x, \mathsf{g}(\mathsf{d}), \mathsf{d}) \overset{*}{\leftrightarrow} \mathsf{a} \overset{*}{\leftrightarrow} \mathsf{f}(y, \mathsf{g}(\mathsf{d}), \mathsf{d})$.

To characterize the situation where we couldn't find any witness from CP_{NF}, we introduce the following property.

▶ **Definition 13** (consistency of CP_{NF})**.** *CP_{NF} is* consistent *if (i) there exists no $(\hat{c}, \hat{r}, t, h) \in CP_{NF}$ such that t has a direct variable subterm, and (ii) no $(\hat{c}, \hat{r}_s, s, h_s), (\hat{c}, \hat{r}_t, t, h_t) \in CP_{NF}$ such that $s \neq t$. It is* inconsistent *if it is not consistent.*

Before ending this subsection, we introduce one notation that will be used below.

▶ **Definition 14.** *We define $\mathrm{T}^{CP} = \{t \mid (\hat{c}, \hat{r}, t, h) \in CP_{NF}\}$.*

3.3 Construction of CW

CP_{NF} finds only constant-equivalent witnesses, so we need to know yet whether there exist any minimal witnesses that are *not* equivalent to a constant.

From Step 3, we may suppose that CP_{NF} is consistent. Assume a term t is equivalent to a constant c. We denote a term $\tilde{t}$ such that $\tilde{t} \in \mathrm{T}^{CP}$ and $c \overset{*}{\leftrightarrow} \tilde{t}$. If such a term exists, it must be unique from the assumption.

▶ **Definition 15.** *(1) Define $\delta''(t)$ for $t \in \mathcal{C} \cup \mathcal{V}$ as follows: $\delta''(t) = \tilde{t}$ if $t \in \mathcal{C}$, and $\delta''(t) = t$ otherwise. (2) Define $\psi''(t)$ for non-constant flat terms t as follows: $\psi''(t) = t$ if $t \in \mathcal{V}$, and $\psi''(t) = f(\delta''(t_1), \dots, \delta''(t_n))$ if $t = f(t_1, \dots, t_n)$ $(n \geq 1)$. (3) Finally, define the set CW:*

$$CW = \{\langle \psi''(l), \psi''(r)\rangle \mid l \approx r \in \widehat{E_{\mathcal{R}}},\ l, r \notin \mathcal{C}\}.$$

▶ **Example 16.** Let us consider $\mathcal{R}$ of Example 1. Through the Step 1 of the Figure 1, we obtain a completion of $\mathcal{R}$ as $\widehat{E_{\mathcal{R}}} = \{\mathsf{b} \approx \mathsf{h}(\mathsf{a}), \mathsf{f}(x, y) \approx \mathsf{g}(\mathsf{b}), \mathsf{f}(x, y) \approx \mathsf{f}(x_1, y_1)\}$. By the Step 2, we obtain $CP_{NF} := \{(\mathsf{a}, \mathsf{a}, \mathsf{a}, 0), (\mathsf{b}, \mathsf{h}(\mathsf{a}), \mathsf{h}(\mathsf{a}), 1)\}$. The conditions of Step 3 fails, and thus we run into Step 4. Note here that from CP_{NF}, we have $\tilde{\mathsf{a}} = \mathsf{a}$ and $\tilde{\mathsf{b}} = \mathsf{h}(\mathsf{a})$. Thus, we obtain $CW = \{\langle \mathsf{f}(x, y), \mathsf{g}(\mathsf{h}(\mathsf{a}))\rangle, \langle \mathsf{f}(x, y), \mathsf{f}(x_1, y_1)\rangle\}$. Since none of $\mathsf{f}(x, y), \mathsf{g}(\mathsf{h}(\mathsf{a})), \mathsf{f}(x, y), \mathsf{f}(x_1, y_1)$ is a normal form of $\mathcal{R}$, we conclude that $\mathcal{R}$ (and hence $\mathcal{R}_{shallow}$) is UNC.

In the subsequent sections, we prove the correctness of our decision procedure, and report its complexity analysis and the result of experiments.

4 Constant Propagation Class

In the previous section, we introduced CP_{NF} that finds witnesses that are equivalent to constants. In this section, we introduce a notion of constant propagation class, a key notion that acts as a mediator between CP_{NF} and constant-equivalent witnesses.

In subsequent sections, we fix a TRS $\mathcal{R}$ that is consistent and flat, and its completion $\widehat{E_{\mathcal{R}}}$. For convenience, we also use $\mathcal{R}$ and $\widehat{E_{\mathcal{R}}}$ as if they are closed under renaming, i.e. we assume $\mathcal{R}$ ($\widehat{E_{\mathcal{R}}}$) includes all the rules (resp. equations) whose variables are renamed, and use $\widehat{E_{\mathcal{R}}}$ as if it contains trivial equations $c \approx c$ ($c \in \mathcal{C}$).

▶ **Definition 17** (constant expansion). *A term t is obtained from s by a* constant expansion*, written as $s \rightharpoonup_{\widehat{E_{\mathcal{R}}}} t$, if there exists $c \in \mathcal{C}$ with $c \approx r \in \widehat{E_{\mathcal{R}}}$ and a position p such that $s|_p = c$, $t = s[r]_p$ and $\mathcal{V}(r) \cap \mathcal{V}(s) = \emptyset$.*

Henceforth, we will omit the subscript $\widehat{E_{\mathcal{R}}}$ of $s \rightharpoonup_{\widehat{E_{\mathcal{R}}}} t$. Clearly, $t \rightharpoonup t'$ implies $t \leftrightarrow t'$.

▶ **Example 18.** Let $\mathcal{R}$ and $\widehat{E_{\mathcal{R}}}$ be given as in Example 3 (enhanced by trivial equations and renamed rules, as explained). Then we have $\mathsf{a} \rightharpoonup \mathsf{a}$, $\mathsf{a} \rightharpoonup \mathsf{b}$, $\mathsf{a} \rightharpoonup \mathsf{f}(x, \mathsf{c}, \mathsf{d}) \rightharpoonup \mathsf{f}(x, \mathsf{c}, \mathsf{h}(\mathsf{a})) \rightharpoonup \mathsf{f}(x, \mathsf{c}, \mathsf{h}(\mathsf{f}(y, \mathsf{c}, \mathsf{d})))$, and $\mathsf{a} \rightharpoonup \mathsf{f}(x, \mathsf{c}, \mathsf{d}) \rightharpoonup \mathsf{f}(x, \mathsf{c}, \mathsf{h}(\mathsf{b})) \rightharpoonup \mathsf{f}(x, \mathsf{c}, \mathsf{h}(\mathsf{a})) \rightharpoonup \mathsf{f}(x, \mathsf{g}(\mathsf{d}), \mathsf{h}(\mathsf{a}))$.

For any $r \in \mathrm{T}_f$, we have $r \overset{*}{\rightharpoonup} t$ iff $r \overset{*}{\rightharpoonup}_{>\epsilon} t$, and hence $r \overset{*}{\rightharpoonup} t$ implies $\mathrm{root}(r) = \mathrm{root}(t)$. This motivates us to introduce a term class parameterized by $c \in \mathcal{C}$ and $r \in \mathrm{T}_f$ as follows.

▶ **Definition 19** (constant propagation class). *Let $c \in \mathcal{C}, r \in T$. Define the* constant propagation class (CPC) *of the pair $\langle c, r \rangle$ as follows:*

$$\mathrm{CP}(c, r) = \{t \mid r \in \mathrm{T}_f,\ c \approx r \in \widehat{E_{\mathcal{R}}},\ \exists t' \textit{ s.t. } c \rightharpoonup r \overset{*}{\rightharpoonup} t' \leqslant t\}$$

Remark that $t \in \mathrm{CP}(c, r)$ implies that $t \in \mathrm{T}_f$ and $\mathrm{root}(t) = \mathrm{root}(r)$.

▶ **Example 20.** Let $\widehat{E_{\mathcal{R}}}$ be as in Example 18. Then, $\mathsf{f}(\mathsf{b}, \mathsf{c}, \mathsf{h}(\mathsf{f}(x, \mathsf{c}, \mathsf{d}))) \in \mathrm{CP}(\mathsf{a}, \mathsf{f}(x, \mathsf{c}, \mathsf{d}))$ as $\mathsf{a} \rightharpoonup \mathsf{f}(x, \mathsf{c}, \mathsf{d}) \overset{*}{\rightharpoonup} \mathsf{f}(x, \mathsf{c}, \mathsf{h}(\mathsf{f}(y, \mathsf{c}, \mathsf{d}))) \leqslant \mathsf{f}(\mathsf{b}, \mathsf{c}, \mathsf{h}(\mathsf{f}(x, \mathsf{c}, \mathsf{d})))$.

Next lemma shows that the class $\mathrm{CP}(c, r)$ is invariant under renaming.

▶ **Lemma 21.** *Let $c \in \mathcal{C}$, $r \in \mathrm{T}_f$, and assume $c \approx r \in \widehat{E_{\mathcal{R}}}$. Then, $\mathrm{CP}(c, r) = \{t \mid \exists t', \sigma_\alpha$ s.t. $c \rightharpoonup \sigma_\alpha(r) \overset{*}{\rightharpoonup} t' \leqslant t\}$, and hence $\mathrm{CP}(c, r) = \mathrm{CP}(c, \sigma_\alpha(r))$.*

The invariance of the $\mathrm{CP}(c, r)$ can be extended further than renaming. Firstly, $\mathrm{CP}(c, r)$ and $[\![c]\!]$ share the following property.

▶ **Lemma 22.** *Let $t \in \mathrm{CP}(c, r) \cup [\![c]\!]$. Then, (1) $c \overset{*}{\rightharpoonup} t' \leqslant t$ for some t', and (2) $c \overset{*}{\leftrightarrow} t$.*

We are now going to show that any CPC is preserved under $\simeq$ (Theorem 25).

▶ **Lemma 23.** *Let $c, c' \in \mathcal{C}$ and $r, r' \in \mathrm{T}_f$. (1) $c \simeq c'$ iff $c \approx c' \in \widehat{E_{\mathcal{R}}}$. (2) If $r \simeq r'$ then $c \approx r \in \widehat{E_{\mathcal{R}}}$ iff $c \approx r' \in \widehat{E_{\mathcal{R}}}$.*

▶ **Lemma 24.** *Let $c, c' \in \mathcal{C}$ and $r, r' \in \mathrm{T}_f$. (1) If $c \simeq c'$ then $c \rightharpoonup c'$. (2) If $c \simeq c'$, then $c \rightharpoonup r$ iff $c' \rightharpoonup r$. (3) If $c \simeq c'$ and $r \simeq r'$, then $c \rightharpoonup r$ iff $c' \rightharpoonup r'$. (4) If $r \simeq r'$ then there exists σ_α such that $\sigma_\alpha(r') \overset{*}{\rightharpoonup} r$.*

▶ **Theorem 25** (preservation of CPCs by $\simeq$). *CPCs are preserved by $\simeq$, i.e. if $c \simeq c'$ and $r \simeq r'$ then $\mathrm{CP}(c, r) = \mathrm{CP}(c', r')$. In particular, $\mathrm{CP}(c, r) = \mathrm{CP}(\hat{c}, \hat{r})$.*

Proof. Use above four lemmas and Proposition 4. See appendix for the detail. ◀

Before ending the section, we relate $\mathit{SubMinWit}_{\mathcal{R}}$ and CPCs.

▶ **Theorem 26.** *Let $c \in \mathcal{C}$ and $t \in \mathit{SubMinWit}_{\mathcal{R}}$ such that $c \overset{*}{\leftrightarrow} t$. Then, there exists a term r such that $t \in \mathrm{CP}(c, r) \cup [\![c]\!]$. Hence, $t \in \mathrm{CP}(\hat{c}, \hat{r}) \cup [\![\hat{c}]\!]$.*

Proof. The proof proceeds by induction on $h = \mathit{height}(t)$. Let $c \in \mathcal{C}$ and $t \in \mathit{SubMinWit}_{\mathcal{R}}$, and assume $c \overset{*}{\leftrightarrow} t$.

1. (B.S.) Suppose $h = 0$. Then $t \in \mathcal{C} \cup \mathcal{V}$ holds. It follows from the consistency of $\mathcal{R}$ that $t \in \mathcal{C}$. Since $c \overset{*}{\leftrightarrow} t$, we have $c \simeq t$. Thus $t \in [\![c]\!]$.
2. (I.S.) By $h > 0$, we have $t \notin [\![c]\!]$. From $c \overset{*}{\leftrightarrow} t$ and Proposition 4, it follows that we have

$$c \leftrightarrow^{\epsilon} f(s'_1, \ldots, s'_n) \overset{*}{\leftrightarrow}{}^{>\epsilon} f(t_1, \ldots, t_n) = t.$$

Let $c \leftrightarrow f(s_1, \ldots, s_n) \in \widehat{E_{\mathcal{R}}}$ be the rule used at the rewrite step $c \leftrightarrow^{\epsilon} f(s'_1, \ldots, s'_n)$. Then, there exists a substitution σ such that $f(\sigma(s_1), \ldots, \sigma(s_n)) = f(s'_1, \ldots, s'_n)$ and $\sigma(s_i) \overset{*}{\leftrightarrow} t_i$ $(1 \leq i \leq n)$. Now, define u_i and σ_i such that $s_i \overset{*}{\rightharpoonup} u_i$, $\sigma_i(u_i) = t_i$ $(i = 1, \ldots, n)$ so that $\bigcup_i \sigma_i$ is well-defined, according to the following case distinction:

   a. Case $s_i = c_i \in \mathcal{C}$. Then, $\sigma(s_i) = c_i \overset{*}{\leftrightarrow} t_i$ holds. From consistency of $\mathcal{R}$, we know $t_i \notin \mathcal{V}$. By $c_i \overset{*}{\leftrightarrow} t_i$ and $t_i \in \mathit{SubMinWit}_{\mathcal{R}}$, it follows from induction hypothesis that there exists r_i such that $t_i \in \mathrm{CP}(c_i, r_i) \cup [\![c_i]\!]$.

      i. Case $t_i \in [\![c_i]\!]$. Put $u_i = t_i$ and $\sigma_i = \{\}$ (the identity substitution). Then $s_i = c_i \simeq t_i = u_i$, and hence we have $s_i \rightharpoonup u_i$ by Lemma 24. Clearly, $\sigma_i(u_i) = t_i$ and $\mathcal{V}(u_i) = \emptyset$.

      ii. Case $t_i \in \mathrm{CP}(c_i, r_i)$. Then, $r_i \in \mathrm{T}_f$ holds. From Lemma 21, $c_i \rightharpoonup \sigma_{\alpha_i}(r_i) \overset{*}{\rightharpoonup} t'_i \leqslant t_i$ for some t'_i and σ_{α_i} such that the variables in $\sigma_{\alpha_i}(r_i)$ or t'_i are fresh. Put $u_i = t'_i$ and take σ_i as a substitution such that $\sigma_i(t'_i) = t_i$. Clearly, we have $s_i = c_i \overset{*}{\rightharpoonup} u_i$ and $\sigma_i(u_i) = t_i$. Furthermore, variables in $\mathrm{dom}(\sigma_i)$ and $\mathcal{V}(u_i)$ are fresh.

   b. Case $r_i \in \mathcal{V}$. Put $u_i = r_i$ and $\sigma_i = \{u_i := t_i\}$. Clearly, we have $r_i \overset{*}{\rightharpoonup} u_i$ and $\sigma_i(u_i) = t_i$.

   Now, we show the substitution $\sigma = \bigcup_{1 \leq i \leq n} \sigma_i$ is well-defined. From the construction, it is clear that it suffices to show $t_p = t_q$ whenever $s_p = s_q \in \mathcal{V}$ $(1 \leq p, q \leq n)$. If $s_p = s_q$, then $t_p \overset{*}{\leftrightarrow} \sigma(s_p) = \sigma(s_q) \overset{*}{\leftrightarrow} t_q$ holds. Then, since $t \in \mathit{SubMinWit}_{\mathcal{R}}$, we have $t_p = t_q$ by Proposition 5. Hence, σ is well-defined. Now we have $c \rightharpoonup f(s_1, \ldots, s_n) \overset{*}{\rightharpoonup} f(u_1, \ldots, u_n)$ and $\sigma(f(u_1, \ldots, u_n)) = t$. Thus, $t \in \mathrm{CP}(c, f(s_1, \ldots, s_n))$.

Thus, there exists r such that $t \in \mathrm{CP}(c, r) \cup [\![c]\!]$. Also, $t \in \mathrm{CP}(\hat{c}, \hat{r}) \cup [\![\hat{c}]\!]$ by Theorem 25. ◀

5 Correctness of Constant Propagation Algorithm

In this section, we describe the correctness of Constant Propagation Algorithm given in Figure 2, which checks whether there exists a minimal witness that is equivalent to a constant.

Because of the main loop, termination of the algorithm needs to be clarified.

▶ **Lemma 27.** *CP algorithm terminates.*

The following properties of elements in CP_{NF} are immediate from the definition.

▶ **Lemma 28.** *Let $t \in \mathcal{C} \cap \mathit{NF}$. Then, $t \in [\![\hat{c}]\!]$ iff $(\hat{c}, \hat{c}, t, 0) \in \mathit{CP}_{NF}$.*

▶ **Lemma 29.** *Let $(\hat{c}, \hat{r}, t, h) \in \mathit{CP}_{NF}$. Then, (1) $\mathrm{height}(t) = h$, (2) $\mathrm{root}(\hat{r}) = \mathrm{root}(t)$, (3) $\hat{r}|_i \in \mathcal{V} \Rightarrow \hat{r}|_i = t|_i$ for each $i \in \mathbb{N}$, (4) $t \in \mathit{NF}$, and (5) $t \notin \mathcal{V}$.*

Further properties are established as well.

▶ **Lemma 30.** *Let $(\hat{c}, \hat{r}, t, h) \in \mathit{CP}_{NF}$. Then, (1) $\hat{c} \rightharpoonup \hat{r} \overset{*}{\rightharpoonup} t$, (2) if $h > 0$ then $t \in \mathrm{CP}(\hat{c}, \hat{r})$, and (3) $\hat{c} \overset{*}{\leftrightarrow} t$.*

From the previous lemma and Lemma 28, $(\hat{c}, \hat{r}, t, h) \in \mathit{CP}_{\mathit{NF}}$ implies $t \in \mathrm{CP}(\hat{c}, \hat{r}) \cup [\![\hat{c}]\!]$. We now consider the reverse direction. In fact, we have already shown the case $t \in [\![\hat{c}]\!]$ in Lemma 28. For the case $t \in \mathrm{CP}(\hat{c}, \hat{r})$, we need a further assumption that $t \in \mathit{SubMinWit}_{\mathcal{R}}$.

▶ **Lemma 31.** *Let $t \in \mathit{SubMinWit}_{\mathcal{R}} \cap \mathrm{CP}(\hat{c}, \hat{r})$. Then, $(\hat{c}, \hat{r}, t', h') \in \mathit{CP}_{\mathit{NF}}$ for some t' and $h' \leq \mathrm{height}(t)$.*

A final property of elements in $\mathit{CP}_{\mathit{NF}}$ we need is a kind of injectivity.

▶ **Lemma 32.** *Suppose $(\hat{c}_s, \hat{r}_s, s, h_s), (\hat{c}_t, \hat{r}_t, t, h_t) \in \mathit{CP}_{\mathit{NF}}$. Then $s = t$ implies $\hat{c}_s = \hat{c}_t$ and $\hat{r}_s = \hat{r}_t$.*

Now we arrive at the main result of this section – $\mathit{CP}_{\mathit{NF}}$ gives a necessary and sufficient criteria to find a minimal witness equivalent to a constant.

▶ **Theorem 33.** *There exists a witness that is equivalent to a constant iff $\mathit{CP}_{\mathit{NF}}$ is inconsistent.*

Proof. ($\Rightarrow$) Let $\langle u, v \rangle$ be a minimal witness that is equivalent to a constant c. We have $c \xleftrightarrow{*} u$, $c \xleftrightarrow{*} v$, $u \neq v$ and $u, v \in \mathit{NF}$. From Theorem 26, there exists r_u, r_v such that $u \in \mathrm{CP}(\hat{c}, \hat{r}_u) \cup [\![\hat{c}]\!]$ and $v \in \mathrm{CP}(\hat{c}, \hat{r}_v) \cup [\![\hat{c}]\!]$. We distinguish four cases:

1. Case $u \in [\![\hat{c}]\!]$ and $v \in [\![\hat{c}]\!]$. From Lemma 28, we have $(\hat{c}, \hat{c}, u, 0), (\hat{c}, \hat{c}, v, 0) \in \mathit{CP}_{\mathit{NF}}$. Since $u \neq v$, the claim holds.
2. Case $u \in \mathrm{CP}(\hat{c}, \hat{r}_u)$ and $v \in [\![\hat{c}]\!]$. From Lemma 31, there exists $u' \in \mathrm{CP}(\hat{c}, \hat{r}_u)$ such that $(\hat{c}, \hat{r}_u, u', h'_u) \in \mathit{CP}_{\mathit{NF}}$. From Lemma 28, we have $(\hat{c}, \hat{c}, v, 0) \in \mathit{CP}_{\mathit{NF}}$. Since $\hat{r}_u \notin \mathcal{C}$, $\hat{c} \neq \hat{r}_u$ holds. Hence Lemma 32 leads $u' \neq v$.
3. Case $u \in [\![\hat{c}]\!]$ and $v \in \mathrm{CP}(\hat{c}, \hat{r}_v)$. Similar to the previous case.
4. Case $u \in \mathrm{CP}(\hat{c}, \hat{r}_u)$ and $v \in \mathrm{CP}(\hat{c}, \hat{r}_v)$. If $(\hat{c}, \hat{r}_u) \neq (\hat{c}, \hat{r}_v)$, then $u \neq v$ by Lemma 32. So, suppose otherwise, i.e. $(\hat{c}, \hat{r}_u) = (\hat{c}, \hat{r}_v)$. By $\hat{r}_u = \hat{r}_v$, $\mathrm{root}(u) = \mathrm{root}(\hat{r}_u) = \mathrm{root}(\hat{r}_v) = \mathrm{root}(v)$. Thus, one can let $u = f(t_{u,1}, \ldots, t_{u,n})$, $v = f(t_{v,1}, \ldots, t_{v,n})$ and $\hat{r}_u = \hat{r}_v = f(s_1, \ldots, s_n)$. Then, there exist $u', v' \in \mathrm{T}_f$, $\sigma_u, \sigma_v \in \Sigma$ such that

$$c \rightharpoonup f(s_1, \ldots, s_n) \overset{*}{\rightharpoonup} f(u_1, \ldots, u_n) = u', \quad \sigma_u(u') = u,$$
$$c \rightharpoonup f(s_1, \ldots, s_n) \overset{*}{\rightharpoonup} f(v_1, \ldots, v_n) = v', \quad \sigma_v(v') = v$$

   where $s_i \overset{*}{\rightharpoonup} u_i$, $s_i \overset{*}{\rightharpoonup} v_i$ for all $1 \leq i \leq n$. Since $u \neq v$, $\sigma_u(u_i) \neq \sigma_v(v_i)$ for some $1 \leq i \leq n$. Assume $\sigma_u(u_i) \neq \sigma_v(v_i)$. Suppose $s_i \in \mathcal{C}$. Then, we have $\sigma_u(u_i) \xleftrightarrow{*} s_i \xleftrightarrow{*} \sigma_v(v_i)$. Since $\sigma_u(u_i), \sigma_v(v_i) \in \mathit{NF}$, $\langle \sigma_u(u_i), \sigma_v(v_i) \rangle$ is a witness equivalent to the constant s_i. This violates the minimality of $\langle u, v \rangle$. Hence $s_i \in \mathcal{V}$. Now, as $u \in \mathit{SubMinWit}_{\mathcal{R}} \cap \mathrm{CP}(\hat{c}, \hat{r}_u)$, there exists $u' \in \mathrm{CP}(\hat{c}, \hat{r}_u) \cup [\![\hat{c}]\!]$ such that $(\hat{c}, \hat{r}_u, u', h'_u) \in \mathit{CP}_{\mathit{NF}}$ by Lemma 31. Because of $s_i \in \mathcal{V}$, we have $s_i = \hat{r}_u|_i = u'|_i$ by Lemma 29. Hence u' has a direct variable subterm.

($\Leftarrow$) Suppose (i) of the definition of $\mathit{CP}_{\mathit{NF}}$ holds. One can take σ_α such that $\sigma_\alpha(t) \neq t$. Then $\langle t, \sigma_\alpha(t) \rangle$ is a witness as $t, \sigma_\alpha(t) \in \mathit{NF}$ and $t \xleftrightarrow{*} c \xleftrightarrow{*} \sigma_\alpha(t)$. Suppose (ii) holds. Then $s, t \in \mathit{NF}$ by Lemma 29 and $s \xleftrightarrow{*} \hat{c} \xleftrightarrow{*} t$ by Lemma 30. Thus $\langle s, t \rangle$ is a witness. ◀

Before ending the section, we present a property regarding T^{CP} (see Definition 14).

▶ **Lemma 34.** *Suppose $\mathit{CP}_{\mathit{NF}}$ is consistent. Then, (1) for any $s, t \in \mathrm{T}^{CP} \cup \mathcal{V}$, $s \xleftrightarrow{*} t$ implies $s = t$. (2) Suppose $c \xleftrightarrow{*} t$ for $c \in \mathcal{C}$ and $t \in \mathit{SubMinWit}_{\mathcal{R}}$. Then, there exists a unique $s \in \mathrm{T}^{CP}$ such that $c \xleftrightarrow{*} s$.*

6 Minimal Witness that is Not Equivalent to a Constant

In the previous section, a sufficient criteria for having a minimal witness that is equivalent a constant is obtained. In this section, we turn our attention to the check whether there exists a witness that is not equivalent to a constant.

We use the following result of [17] as our starting point. For each term t, one can assign a variable $x_{\overline{t}}$ in such a way that $x_{\overline{s}} = x_{\overline{t}}$ if and only if $s \overset{*}{\leftrightarrow} t$. Using this convention, the following definition is given.

▶ **Definition 35** ([17]). *Let δ and ψ be defined as follows: (1) $\delta(t) = t$ if t is equivalent to a constant, and $\delta(t) = x_{\overline{t}}$ otherwise. (2) $\psi(t) = t$ if $t \in \mathcal{C} \cup \mathcal{V}$, and $\psi(t) = f(\delta(t_1), \ldots, \delta(t_n))$ if $t = f(t_1, \ldots, t_n)$ $(n \geq 1)$.*

▶ **Proposition 36** ([17]). *Let $\langle s, t\rangle$ be a minimal witness that is not equivalent to a constant. Then, either $\langle \psi(s), y\rangle$, $\langle y, \psi(t)\rangle$ or $\langle \psi(s), \psi(t)\rangle$ is a witness for some variable y.*

We first refine δ so that the candidates of $\delta(t)$ form a smaller set. As we focus the case that there is no witness that is equivalent a constant, for the rest of the section, we suppose CP_{NF} is consistent. We refine δ to δ' by substituting a unique term $\tilde{t}$ for $\delta'(t)$ (see section 3.3); the existence of such a term is guaranteed for $t \in \mathit{SubMinWit}_{\mathcal{R}}$ by our assumption just given and Lemma 34.

▶ **Definition 37.** *Let δ' and ψ' be defined as follows: (1) $\delta'(t) = \tilde{t}$ if t is equivalent to a constant, and $\delta'(t) = x_{\overline{t}}$ otherwise. (2) $\psi'(t) = t$ if $t \in \mathcal{C} \cup \mathcal{V}$, and $\psi'(t) = f(\delta'(t_1), \ldots, \delta'(t_n))$ if $t = f(t_1, \ldots, t_n)$ $(n \geq 1)$.*

The following lemma is readily checked.

▶ **Lemma 38.** *Let $t \in \mathit{SubMinWit}_{\mathcal{R}}$. (1) Then, $\mathrm{root}(t) = \mathrm{root}(\psi(t)) = \mathrm{root}(\psi'(t))$. (2) If $t \in \mathrm{T}_f$ and $\psi(t)|_i \in \mathcal{V}$ then $\psi(t)|_i = \psi'(t)|_i$ for all $i \in \mathbb{N}$. (3) If $t \in \mathrm{T}_f$, then $\psi(t)|_i \overset{*}{\leftrightarrow} \psi'(t)|_i$ for all $i \in \mathbb{N}$. (4) We have $\psi(t) \overset{*}{\leftrightarrow} \psi'(t)$.*

A further property of ψ' is as follows.

▶ **Lemma 39.** *Let $\langle s, t\rangle$ be a minimal witness that is not equivalent to a constant. Then, $s \notin \mathcal{V}$ $(t \notin \mathcal{V})$ implies $\psi'(s) \in NF$ (resp. $\psi'(t) \in NF$).*

We can now refine Proposition 36 as follows.

▶ **Lemma 40.** *Let $\langle s, t\rangle$ be a minimal witness that is not equivalent to a constant. Then, either $\langle \psi'(s), y\rangle$, $\langle y, \psi'(t)\rangle$ or $\langle \psi'(s), \psi'(t)\rangle$ is a witness for some variable y.*

The definition of ψ' gives rise to the following characterization of terms.

▶ **Definition 41.** *We define $\mathrm{T}^{CP}_f = \{f(t_1, \ldots, t_n) \mid t_i \in \mathrm{T}^{CP} \cup \mathcal{V}$ for all $i\}$.*

The following lemma will be used later.

▶ **Lemma 42.** *Let $\langle s, t\rangle$ be a witness such that $s, t \in \mathrm{T}^{CP}_f \cup \mathcal{V}$. Then, there exists a proof $s \overset{*}{\leftrightarrow} t$ which has precisely one root rewrite step using a non-trivial equation.*

CW (Definition 15) is now used to further restrict the witnesses class. Because $\widehat{E_{\mathcal{R}}}$ is finite, CW is a finite set, and it can be checked whether an element of CW is a witness:

▶ **Lemma 43.** *Let $\langle s, t\rangle \in CW$. Then, $\langle s, t\rangle$ is a witness if and only if $s, t \in NF$ and $s \neq t$.*

We now arrive at the main result of this section that a witness (if it exists) can be found in CW, if there is no minimal witness equivalent to a constant.

▶ **Theorem 44.** *Suppose CP_{NF} is consistent. If there exists a minimal witness that is not equivalent to a constant, then there exists a witness in CW.*

Proof. We show that if there exists a witness $\langle s, t\rangle$ such that $s, t \in (\mathrm{T}_f^{CP} \cap NF) \cup \mathcal{V}$ then there exists a witness $\langle s', t'\rangle \in CW$. Then the claim follows from Lemma 40, as $\psi'(s), \psi'(t) \in \mathrm{T}_f^{CP} \cap NF$ by Lemma 39. Let $\langle s, t\rangle$ be a witness such that $s, t \in (\mathrm{T}_f^{CP} \cap NF) \cup \mathcal{V}$. Then, w.l.o.g. one can suppose (a) $s = f(u_1, \ldots, u_m)$ and $t = g(u_{m+1}, \ldots, u_{m+n})$, or (b) $s = f(u_1, \ldots, u_m)$ and $t = u_{m+1} \in \mathcal{V}$. Now, we repeatedly refine the witness $\langle s, t\rangle$ until we get a desired witness $\langle s', t'\rangle \in CW$.

We first describe one step refinement from $\langle s, t\rangle$ to $\langle s', t'\rangle$ for the case (a). Suppose $s, t \in \mathrm{T}_f^{CP} \cap NF$ with $s = f(u_1, \ldots, u_m)$ and $t = g(u_{m+1}, \ldots, v_{m+n})$.

By Lemma 42, there exist $f(\mu_1, \ldots, \mu_m)$, $g(\mu_{m+1}, \ldots, \mu_{m+n})$ such that

$$s \overset{*}{\leftrightarrow}{}^{>\epsilon} f(\mu_1, \ldots, \mu_m) \leftrightarrow^{\epsilon} g(\mu_{m+1}, \ldots, \mu_{m+n}) \overset{*}{\leftrightarrow}{}^{>\epsilon} t$$

Let $f(s_1, \ldots, s_m) \approx g(s_{m+1}, \ldots, s_{m+n}) \in \widehat{E_{\mathcal{R}}}$ be the equation used in the root rewrite step. Then, there exists a substitution σ such that $u_i \overset{*}{\leftrightarrow} \mu_i = \sigma(s_i)$ for all $1 \leq i \leq m+n$. Suppose there exists $s_i \in \mathcal{V}$ either $u_i = \mu_i \in \mathcal{V}$ does not hold, or there exists j such that $s_i \neq s_j$ and $\mu_i = \mu_j$. If there is no such s_i, the refining step stops.

Let $\{k_1, \ldots k_p\} = \{j \in \mathbb{N} \mid s_i = s_j\}$. Then, we have $u_{k_1} \overset{*}{\leftrightarrow} u_{k_2} \overset{*}{\leftrightarrow} \cdots \overset{*}{\leftrightarrow} u_{k_p}$. Since $s, t \in \mathrm{T}_f^{CP}$, $u_{k_1}, \ldots, u_{k_p} \in \mathrm{T}^{CP} \cup \mathcal{V}$. Thus, Lemma 34 yields $u_{k_1} = \cdots = u_{k_p}$. Now, take a fresh variable x, and let u'_j, μ'_j be x for all $j \in \{k_1, \ldots k_p\}$ and u_j, μ_j, respectively, otherwise. Then we obtain a proof

$$s' \overset{*}{\leftrightarrow}{}^{>\epsilon} f(\mu'_1, \ldots, \mu'_m) \leftrightarrow^{\epsilon} g(\mu'_{m+1}, \ldots, \mu'_{m+n}) \overset{*}{\leftrightarrow}{}^{>\epsilon} t'$$

where $s' = f(u'_1, \ldots, u'_m)$ and $t' = g(u'_{m+1}, \ldots, u'_{m+n})$.

By construction, it is clear that $s', t' \in \mathrm{T}_f^{CP}$. Since $s \in NF$ and $Patt(s') \subseteq Patt(s)$, we have $s' \in NF$ by Lemma 2. Similarly, $t' \in NF$. Thus, $\langle s', t'\rangle$ is a witness such that $s', t' \in \mathrm{T}_f^{CP} \cap NF$.

Next, we describe one step refinement from $\langle s, t\rangle$ to $\langle s', t'\rangle$ for the case (b). So, assume $s = f(u_1, \ldots, u_m)$, $t = u_{m+1} \in \mathcal{V}$ with $s \in \mathrm{T}_f^{CP} \cap NF$.

By Lemma 42, there exists $f(\mu_1, \ldots, \mu_m)$ such that

$$s = f(u_1, \ldots, u_m) \overset{*}{\leftrightarrow}{}^{>\epsilon} f(\mu_1, \ldots, \mu_m) \leftrightarrow^{\epsilon} t$$

Suppose the equation $f(s_1, \ldots, s_m) \approx s_{m+1} \in \widehat{E_{\mathcal{R}}}$ was used for the root rewrite. Suppose there exists $s_i \in \mathcal{V}$ such that either $u_i = \mu_i \in \mathcal{V}$ does not hold, or there exists j such that $s_i \neq s_j$ and $\mu_i = \mu_j$. Then, similar to the case (a), one can obtain a witness $\langle s', t'\rangle$ such that $s' \in \mathrm{T}_f^{CP} \cap NF$ and $t' \in \mathcal{V}$.

Since $\langle s', t'\rangle$ is a witness such that $s', t' \in (\mathrm{T}_f^{CP} \cap NF) \cup \mathcal{V}$, we can repeatedly apply the refinement step above. Since one can iterates the refining steps at most $n+m$-times, eventually one obtains a witness $\langle s'', t''\rangle$ so that, for some equation $f(s_1, \ldots, s_m) \approx g(t_1, \ldots, t_n) \in \widehat{E_{\mathcal{R}}}$ (note the all renaming equations are in $\widehat{E_{\mathcal{R}}}$), $s''|_i \in \mathrm{T}^{CP} \cup \mathcal{V}$, $s''|_i = s_i$ if $s_i \in \mathcal{V}$, and $s''|_i \overset{*}{\leftrightarrow} s_i$ if $s_i \in \mathcal{C}$ for all $1 \leq i \leq m$, and similarly for all t_i's.

Furthermore, since there are no $(\hat{c}, \hat{r}_s, s, h_s), (\hat{c}, \hat{r}_t, t, h_t) \in CP_{NF}$ such that $s \neq t$, we have $s'' = \psi''(f(s_1, \ldots, s_m))$, $t'' = \psi''(g(t_1, \ldots, t_n))$. Hence, $\langle s'', t''\rangle \in CW$. ◀

Table 1 Construction of CP_{NF} by CP algorithm.

$\hat{c}$	$\hat{r}$ (or $\hat{c}$)	normal forms	h
a	a	b	0
	$\mathsf{f}(x, \mathsf{c}, \mathsf{d})$		
c	c	-	-
	$\mathsf{g}(\mathsf{d})$		
d	d	d	0
	$\mathsf{h}(\mathsf{a})$		
e	e	-	-
	$\mathsf{g}(\mathsf{e})$		

(CP table after Step 2)

$\hat{c}$	$\hat{r}$ (or $\hat{c}$)	normal forms	h
a	a	b	0
	$\mathsf{f}(x, \mathsf{c}, \mathsf{d})$	$\mathsf{f}(x, \mathsf{g}(\mathsf{d}), \mathsf{d})$	2
		$\mathsf{f}(x, \mathsf{g}(\mathsf{d}), \mathsf{h}(\mathsf{b}))$	2
c	c	-	-
	$\mathsf{g}(\mathsf{d})$	$\mathsf{g}(\mathsf{d})$	1
d	d	d	0
	$\mathsf{h}(\mathsf{a})$	$\mathsf{h}(\mathsf{b})$	1
e	e	-	-
	$\mathsf{g}(\mathsf{e})$		

(Final CP table)

7 Correctness of the Decision Procedure and Its Complexity

Combining preparations in the previous sections, we now show the correctness of the decision procedure for the UNC property of shallow TRSs in Figure 1. The following is immediate.

▶ **Lemma 45.** *The procedure given in Figure 1 terminates.*

Our main theorem follows from Theorems 33 and 44.

▶ **Theorem 46.** *It can be decided whether a given shallow TRS is UNC or not, by the procedure given in Figure 1.*

We now analyze the complexity of our algorithm. Following [17], the complexity of the algorithm is evaluated in terms of the number of rules in the flat TRS $\mathcal{R}$, and we omit the cost of constructing $\widehat{E_{\mathcal{R}}}$.

In Section 3, we give a *set-based* description of the constant propagation algorithm. To evaluate the complexity, we introduce a data structure *CP table* as illustrated in the following example.

▶ **Example 47.** Let $\mathcal{R}$ and $\widehat{E_{\mathcal{R}}}$ be as in Example 12. One can obtain CP_{NF} as in Table 1 according to following procedure:

1. Enumerate all constants in $\hat{\mathcal{C}}$ and fill the first column of the table.
2. For all $\hat{c} \in \hat{\mathcal{C}}$, enumerate all elements of $[\![\hat{c}]\!]$ and all equations $\hat{c} \approx \hat{r} \in \widehat{E_{\mathcal{R}}}$ $(r \notin \mathcal{C})$ to fill the second column.
3. Fill the 1st, 3rd, 5th and 7th rows according to $CP_{NF} := \{(\hat{c}, \hat{c}, c, 0) \mid c \in \mathcal{C} \cap NF\}$.
4. When an element $(\hat{c}, \hat{r}, t, H)$ is added to CP_{NF} in the main loop of the algorithm, fill the third and forth columns with t, H whose first and second columns correspond to $\hat{c} \approx \hat{r}$.

The table is referred to as a *CP table.* A row of the CP table with non-empty third column corresponds an element of CP_{NF}. Thus, $\chi_H(\hat{c})$ is given by look up of the third columns of the rows with $h < H$ and having $\hat{c}$ at the first column.

▶ **Theorem 48.** *The procedure in Figure 1 runs in $O(\alpha|\mathcal{R}|^{4\alpha+5})$, where α is the maximal arity of function symbols and $|\mathcal{R}|$ is the number of rules in $\mathcal{R}$.*

Proof. Let $\mathcal{R}$ be the flat TRS obtained by the transformation from the input shallow TRS, and $\widehat{E_{\mathcal{R}}}$ the completion of $E_{\mathcal{R}}$. Let $N = |\mathcal{R}|$ and $M = |\widehat{E_{\mathcal{R}}}|$.

Let us first evaluate the CP algorithm. If the set X_{tmp} is non-empty, then $X_{i,\hat{r}}$ is non-empty for some $i, \hat{r}$. Thus in each iteration of the main loop, the number of $Y_{\hat{c}_i}$ reduces for some i. As $\sum_i |Y_{\hat{c}_i}| \leq M$, the number of iterations of the main loop is bounded by M.

In each iteration, the calculation of χ_H is replaced with the look up of the CP table for calculating CP_{NF} (Table 1) Each pair $\langle \hat{c}, \hat{r} \rangle$ in the table is from an equation $c \approx r \in \widehat{E_{\mathcal{R}}}$. If one finds two normal forms for the pair $\langle \hat{c}, \hat{r} \rangle$, during the construction of the CP table, then one can stop the construction and output a counter example. Thus, the height of the CP table is bounded by the number of such pairs, i.e. by M. The calculation of a candidate of new normal form $f(u'_1, \dots, u'_n)$ is done by representing non-variable direct subterms of $\hat{r}$ as a pointer to an entry of the CP table. Then, one has to check the candidate $f(u'_1, \dots, u'_n)$ is whether a normal form; as as $u'_1, \dots, u'_n$ are normal forms and $\mathcal{R}$ is flat, this is done in $O(\alpha N)$. Note that each non-variable u'_i can be identified as a pointer to an entry of the CP table. Thus, the calculation of each entry of the CP table is done in $O(\alpha N)$.

Thus, each iteration of the main loop is bounded by $O(\alpha NM)$, and hence the CP algorithm is bounded by $O(\alpha NM^2)$. During the construction of the CP table the checks in Step 3 can be done in $O(1)$. Thus, this accounts the complexity of the Steps 2, 3.

For the Step 4, first note that computing $\tilde{c}$ costs M. Thus, computing $\psi''(t)$ costs at most αM. Since the size of CW is at most M, one needs $O(\alpha M^2)$ for computing the set CW. For each $\langle s, t \rangle \in CW$, checking whether $s, t \in NF$ needs $O(\alpha N)$, and checking whether $s \neq t$ needs $O(\alpha)$, Since the size of CW is at most M, checking the existence of a witness in CW costs $O(\alpha NM)$. Thus, the Step 4 runs in $O(\alpha M^2 + \alpha NM)$.

Thus, the complexity is dominated by Step 1, that is, $O(\alpha NM^2)$. Now it is known that M is bounded by $O(N^{2\alpha+2})$ [17]. Hence, we conclude that the complexity of the algorithm is $O(\alpha N^{4\alpha+5})$. ◀

▶ Remark 49. It is shown in [17] that the complexity of the algorithm given there is $O(|\mathcal{R}|^{2\alpha+2}(|\mathcal{R}| + (|\mathcal{F}| + \beta + 1)^{O(\beta\alpha^{|\mathcal{C}|})})$, where $\beta = O(\max(\alpha, |\mathcal{C}| - 1)))$. This is of the form $O(N(M + L))$ where L is the number of candidates for witnesses (and N, M as in the proof above). This complexity comes from check $s \neq t$ of the candidates $\langle s, t \rangle$ of the witnesses. In their algorithm, the complexity of the candidates construction part $O(\alpha ML)$ does not affect the final complexity. In contrast, our complexity comes from the candidates construction part $O(\alpha NM^2)$. In this view, a large set L of candidates is reduced to a set of size $O(NM)$ in our algorithm, and the witness checking part is omittable in contrast.

8 Implementations and Experiments

We have implemented our decision procedure, as well as the existing one described in [17]. We use the functional programming language SML/NJ for the implementations.

We have prepared 13 shallow TRSs which covers various situations of the algorithm for our experiments. The timeout was set to 300 seconds. When the execution exceeds 300 seconds, we regard the decision was failed, and we write the execution time as ∞. Our computer used for the experiments has Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz and 4GB memory. Standard ML of New Jersey of v110.79 has been used.

A summary of the experiments is shown in Table 2. Here, “YES” stands for UNC and “NO” stands for Non-UNC; when it cannot judge UNC by timeout, the result is shown as “ - ”. WEC stands for “a witness equivalent to a constant.” The results show that our procedure can judge UNC for all examples 1–13. On the other hand, the existing procedure cannot for examples 7–9,11, because the execution time sharply increased as the rules become complicated. Furthermore, except for very simple examples, the proposed procedure was able to run significantly faster than the existing method.

Table 2 A summary of experiments on examples for various situations.

	Procedure of [17]		Proposed procedure		
TRS	**Result**	**Time(s)**	**Result**	**Time(s)**	features of the TRS
$\mathcal{R}_1$	NO	0.003	NO	0.001	signature with only constants
$\mathcal{R}_2$	NO	0.177	NO	0.002	flat with WEC
$\mathcal{R}_3$	NO	0.001	NO	0.000	flat without WEC
$\mathcal{R}_4$	YES	0.001	YES	0.001	flat, simple, UNC
$\mathcal{R}_5$	NO	0.000	NO	0.000	inconsistent
$\mathcal{R}_6$	YES	0.868	YES	0.000	shallow, simple, height 2
$\mathcal{R}_7$	-	∞	YES	0.000	shallow, simple, height 3
$\mathcal{R}_8$	-	∞	NO	0.014	shallow, complex, with WEC
$\mathcal{R}_9$	-	∞	NO	0.013	shallow, complex, without WEC
$\mathcal{R}_{10}$	YES	0.004	YES	0.001	shallow, simple, UNC
$\mathcal{R}_{11}$	-	∞	YES	0.011	shallow, complex, UNC
$\mathcal{R}_{12}$	NO	0.003	NO	0.001	non-linear, Non-UNC
$\mathcal{R}_{13}$	YES	0.900	YES	0.000	non-linear, UNC

Table 3 A summary of experiments on problems from Cops.

	Procedure of [17]	Proposed procedure
Result	**Num of examples**	**Num of examples**
YES	38	94
NO	18	45
timeout	90	7

We have also tested how procedures fare for the problems from the Cops (confluence problems) database[2]. At the time of the experiment, the database consists of 1137 problems, containing 146 shallow TRSs in it. The timeout was set to 60 sec., which is the timeout used in the Confluence Competition. A summary of the experiments is shown in Table 3. Our procedure succeeds 139 examples and have 7 timeouts, while the previous procedure succeeds 56 examples and have 90 timeouts. Our decision procedure has been also incorporated to the confluence tool ACP [5], which have won the category of UNC in the 2019 edition of Confluence Competition (CoCo 2019)[3].

Our implementations as well as the details of the experiments can be found in the webpage `http://www.nue.ie.niigata-u.ac.jp/experiments/fscd20/`.

9 Conclusion

In this paper, we have proposed a new decision procedure for the UNC property of shallow TRSs. We have introduced a constant propagation algorithm that efficiently constructs candidates of counter examples that are equivalent to a constant. Those candidates have been also used to construct candidates of counter examples that are not equivalent to a constant either. Thus, a large enumeration of candidates for counter examples have been avoided, in contrast to the existing algorithm of [17]. The correctness of the proposed procedure has

[2] `https://cops.uibk.ac.at/`
[3] `http://project-coco.uibk.ac.at/2019/`

been proved and its complexity has been analyzed. Furthermore, we have implemented the proposed decision procedure and the existing one, and it has been experimentally confirmed that the proposed procedure runs much faster than the existing procedure.

References

1 T. Aoto, M. Hamana, N. Hirokawa, A. Middeldorp, J. Nagele, N. Nishida, K. Shintani, and H. Zankl. Confluence competition 2018. In *Proc. of 3rd FSCD*, volume 108 of *LIPIcs*, pages 32:1–32:5. Schloss Dagstuhl, 2018. doi:10.4230/LIPIcs.FSCD.2018.32.

2 T. Aoto, N. Hirokawa, J. Nagele, N. Nishida, and H. Zankl. Confluence competition 2015. In *Proc. of 25th CADE*, volume 9195 of *LNCS*, pages 101–104. Springer-Verlag, 2015. doi:10.1007/978-3-319-21401-6_5.

3 T. Aoto and Y. Toyama. On composable properties of term rewriting systems. In *Proc. of 6th ALP and 3rd HOA*, volume 1298 of *LNCS*, pages 114–128. Springer-Verlag, 1997. doi:10.1007/BFb0027006.

4 T. Aoto and Y. Toyama. Automated proofs of unique normal forms w.r.t. conversion for term rewriting systems. In *Proc. of 12th FroCoS*, volume 11715 of *LNAI*, pages 330–347. Springer-Verlag, 2019. doi:10.1007/978-3-030-29007-8_19.

5 T. Aoto, Y. Yoshida, and Y. Toyama. Proving confluence of term rewriting systems automatically. In *Proc. of 20th RTA*, volume 5595 of *LNCS*, pages 93–102. Springer-Verlag, 2009. doi:10.1007/978-3-642-02348-4_7.

6 F. Baader. and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998. doi:10.1017/CBO9781139172752.

7 L. Bachmair, N. Dershowitz, and D. A. Plaisted. Completion without failure. In H. Aït Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2, pages 1–30. Academic Press, 1989. doi:10.1016/B978-0-12-046371-8.50007-9.

8 H. Comon, M. Haberstrau, and J. P. Jouannaud. Syntacticness, cycle-syntacticness, and shallow theories. *Information and Computation*, 111(1):154–191, 1994. doi:10.1006/inco.1994.1043.

9 M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. In *Proc. of 5th LICS*, pages 242–248. IEEE Computer Society Press, 1990. doi:10.1109/LICS.1990.113750.

10 R. C. de Vrijer. Conditional linearization. *Indagationes Mathematicae*, 10(1):145–159, 1999. doi:10.1016/S0019-3577(99)80012-3.

11 B. Felgenhauer. Deciding confluence and normal form properties of ground term rewrite systems efficiently. *Logical Methods in Computer Science*, 14(4:7):1–35, 2018. doi:10.23638/LMCS-14(4:7)2018.

12 S. Kahrs and C. Smith. Non-ω-overlapping TRSs are UN. In *Proc. of 1st FSCD*, volume 52 of *LIPIcs*, pages 22:1–17. Schloss Dagstuhl, 2016. doi:10.4230/LIPIcs.FSCD.2016.22.

13 A. Middeldorp. *Modular Properties of Term Rewriting Systems.* PhD thesis, Vrije Universiteit, Amsterdam, 1990.

14 A. Middeldorp, J. Nagele, and K. Shintani. Confluence competition 2019. In *Proc. of 25th TACAS*, volume 11429 of *LNCS*, pages 25–40. Springer-Verlag, 2019. doi:10.1007/978-3-030-17502-3_2.

15 I. Mitsuhashi, M. Oyamaguchi, and F. Jacquemard. The confluence problem for flat TRSs. In *Proc. of 8th AISC*, volume 4120 of *LNAI*, pages 68–81. Springer-Verlag, 2006. doi:10.1007/11856290_8.

16 N. Radcliffe and R. M. Verma. Uniqueness of normal forms is decidable for shallow term rewrite systems. In *Proc. of 30th FSTTCS*, volume 8 of *LIPIcs*, pages 284–295. Schloss Dagstuhl, 2010. doi:10.4230/LIPIcs.FSTTCS.2010.284.

17 N. R. Radcliffe, L. F. T. Moreas, and R. M. Verma. Uniqueness of normal forms for shallow term rewrite systems. *ACM Transactions on Computational Logic*, 18(2):17:1–17:20, 2017. doi:10.1145/3060144.

18 F. Rapp and A. Middeldorp. Automating the first-order theory of rewriting for left-linear right-ground rewrite systems. In *Proc. of 1st FSCD*, volume 52 of *LIPIcs*, pages 36:1–17. Schloss Dagstuhl, 2016. doi:10.4230/LIPIcs.FSCD.2016.36.

19 Y. Toyama and M. Oyamaguchi. Conditional linearization of non-duplicating term rewriting systems. *IEICE Transactions on Information and Systems*, E84-D(4):439–447, 2001.

20 J. Zinn and R. M. Verma. A polynomial algorithm for uniqueness of normal forms of linear shallow term rewrite systems. *Applicable Algebra in Engineering, Communication and Computing*, 21(6):459–485, 2010. doi:10.1007/s00200-010-0133-1.

A Omitted Proofs

Proof of Lemma 6. Let $t = f(t_1, \ldots, t_n)$. Then there exists a minimal witness $\langle u, v\rangle$ such that t is a subterm of u or v. Assume $t_i \overset{*}{\leftrightarrow} c \in \mathcal{C} \cap \mathit{NF}$ with $t_i \neq c$. From $u, v \in \mathit{NF}$, we know $t_i \in \mathit{NF}$. Thus, $\langle t_i, c\rangle$ is a witness. But, as $|t_i| + |c| = |t_i| + 1 < |t| + 1 \leq |u| + |v|$, this contradicts the minimality of $\langle u, v\rangle$. ◀

Proof of Lemma 21. Note that, by our convention, $c \approx r \in \widehat{E_{\mathcal{R}}}$ iff $c \approx \sigma_\alpha(r) \in \widehat{E_{\mathcal{R}}}$. ($\subseteq$) Clear. ($\supseteq$) Observe $\sigma_\alpha(r) \overset{*}{\rightharpoonup} t'$ implies $r \overset{*}{\rightharpoonup} \sigma_\alpha^{-1}(t') \leqslant t' \leqslant t$. ◀

Prof of Lemma 22. (1) For $t \in [\![c]\!]$, we have $t \in \mathcal{C}$ by the definition, and thus, $c \leftrightarrow t$ or $c = t$ by Proposition 4 (1). For $t \in \mathrm{CP}(c, r)$, it is clear from the definition. (2) From (1), we have $c \overset{*}{\rightharpoonup} t' \leqslant t$. Hence we have $c \overset{*}{\leftrightarrow} t'$ and $\sigma(t') = t$ for some σ. Therefore, $c = c\sigma \overset{*}{\leftrightarrow} t'\sigma = t$. ◀

Proof of Lemma 23. (1) follows immediately from Proposition 4 and our convention that $c \approx c \in \widehat{E_{\mathcal{R}}}$ for $c \in \mathcal{C}$. (2) is a consequence of (1) and the inference rule (3) for $\widehat{E}$ ([8], p. 160) and by our convention that $\widehat{E_{\mathcal{R}}}$ is closed under renaming. ◀

Proof of Lemma 24. (1) By Lemma 23. (2) By Proposition 4. (3) Use (2) and Lemma 23. (4) Then $r = f(s_1, \ldots, s_n)$, $r' = f(t_1, \ldots, t_n)$ $(n \geq 1)$, and there exists σ_α such that $s_i = \sigma_\alpha(t_i)$ for all $s_i \in \mathcal{V}$, and $s_i \simeq t_i$ for all $s_i \in \mathcal{C}$. From (1), $t_i \rightharpoonup s_i$ for all $s_i \in \mathcal{C}$. Thus, $\sigma_\alpha(r') \overset{*}{\rightharpoonup} r$. ◀

Proof of Theorem 25. By the definition of CPC, we only consider the case $c, c' \in \mathcal{C}$. Since $r \simeq r'$, we have either (a) $r, r' \in V$, (b) $r, r' \in \mathcal{C}$ or (c) $r, r' \in \mathrm{T}_f$. For the cases (a), (b), we have $\mathrm{CP}(c, r) = \emptyset = \mathrm{CP}(c, r)$ by the definition. Thus, assume furthermore, $r, r' \in \mathrm{T}_f$. It suffices to show that (1) $c \simeq c'$ implies $\mathrm{CP}(c, r) = \mathrm{CP}(c', r)$ for any $r \in \mathrm{T}_f$, and (2) $r \simeq r'$ implies $\mathrm{CP}(c, r) = \mathrm{CP}(c, r')$ for any $c \in \mathcal{C}$. (1) follows from Proposition 4. (2) Suppose $r \simeq r'$. Then, by Lemma 24, $\sigma_\alpha(r') \overset{*}{\rightharpoonup} r$ for some σ_α. Furthermore, from $r \simeq r'$, we have $c \approx r \in \widehat{E_{\mathcal{R}}}$ iff $c \approx r' \in \widehat{E_{\mathcal{R}}}$ by Lemma 23. Suppose $t \in \mathrm{CP}(c, r)$. Then, we have $c \rightharpoonup r \overset{*}{\rightharpoonup} t' \leqslant t$ for some t'. Hence $c \rightharpoonup \sigma_\alpha(r') \overset{*}{\rightharpoonup} t' \leqslant t$. Thus, by Lemma 21, we obtain $t \in \mathrm{CP}(c, r')$. ◀

Proof of Lemma 27. $k = \sum_{i=1}^{m} |Y_{\hat{c}_i}|$ is finite for $H = 0$, and the algorithm decreases k in each iteration of the main loop. ◀

Proof of Lemma 30. (1) The proof proceeds by induction on h. (B.S.) Then $\hat{r} = \hat{c}$ and $t = c$. Thus $\hat{c} \simeq t$ and the claim follows by Lemma 24. (I.S.) By Lemma 29, let $\hat{r} = f(\hat{u}_1, \ldots, \hat{u}_n)$, $t = f(u'_1, \ldots, u'_n)$. As $\hat{r} \in Y_{\hat{c}}$, we have $\hat{c} \approx r \in \widehat{E_{\mathcal{R}}}$, and thus, $\hat{c} \approx \hat{r} \in \widehat{E_{\mathcal{R}}}$ by Lemma 24. Hence $\hat{c} \rightharpoonup \hat{r}$. Consider the relation between $\hat{u}_i$ and u'_i, according to the following case distinction:

1. Case $\hat{u}_i \in \mathcal{V}$. Then, by Lemma 29, $u'_i = \hat{u}_i$.

2. Case $\hat{u}_i \in \mathcal{C}$. Then, by the definition of χ^H, either (a) $(\hat{u}_i, \hat{r}_i, u'_i, h'_i) \in \mathit{CP}_{\mathit{NF}}$ for some $\hat{r}_i$ and $h'_i < h_i$ or (b) $\hat{u}_i = u'_i$. In the former case, we have $\hat{u}_i \rightharpoonup \hat{r}_i \overset{*}{\rightharpoonup} u'_i$ by induction hypothesis. In the latter case, $\hat{u}_i \overset{*}{\rightharpoonup} u'_i$ trivially.

Thus, $\hat{c} \rightharpoonup \hat{r} = f(\hat{u}_1, \ldots, \hat{u}_n) \overset{*}{\rightharpoonup} f(u'_1, \ldots, u'_n) = t$. (2) If $h > 0$, then $\hat{r} \in \mathrm{T}_f$ by Lemma 29, and thus the claim follows from (1). (3) is also clear from (1). ◀

Proof of Lemma 31. The proof proceeds by induction on $h = \mathrm{height}(t)$. Let $t \in \mathit{SubMinWit}_{\mathcal{R}} \cap \mathrm{CP}(\hat{c}, \hat{r})$. (B.S.) By $t \in \mathrm{CP}(\hat{c}, \hat{r})$, we have $h > 0$. Thus, the claim trivially holds. (I.S.) By the definition of CPC, one can let $\hat{r} = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$, and for some $u = f(u_1, \ldots, u_n)$,

$$c \rightharpoonup^{\epsilon} f(s_1, \ldots, s_n) \overset{*}{\rightharpoonup}{}^{>\epsilon} f(u_1, \ldots, u_n) \leqslant f(t_1, \ldots, t_n).$$

Let $\sigma(f(u_1, \ldots, u_n)) = f(t_1, \ldots, t_n)$. By $s_i \overset{*}{\leftrightarrow} u_i$, we have $\sigma(s_i) \overset{*}{\leftrightarrow} \sigma(u_i) = t_i$. Now, define t'_i and h_i $(1 \leq i \leq n)$ according to the following case distinction:

1. Case $s_i = c_i \in \mathcal{C}$. Then, $\sigma(s_i) = c_i \overset{*}{\leftrightarrow} t_i$ holds. Since $\mathcal{R}$ is consistent, $t_i \notin \mathcal{V}$. Since $t_i \in \mathit{SubMinWit}_{\mathcal{R}}$, there exists $\hat{r}_i$ such that $t_i \in \mathrm{CP}(\hat{c}_i, \hat{r}_i) \cup [\![\hat{c}_i]\!]$ by Theorem 26.
   - Case $t_i \in [\![\hat{c}_i]\!]$. Put $t'_i = t_i$ and $h_i = 0$. Since $t_i \in \mathcal{C} \cap \mathit{NF}$, we have $(\hat{c}_i, \hat{c}_i, t'_i, h_i) \in \mathit{CP}_{\mathit{NF}}$ by Lemma 28.
   - Case $t_i \in \mathrm{CP}(\hat{c}_i, \hat{r}_i)$. By induction hypothesis, there exists $t'_i \in \mathrm{CP}(\hat{c}_i, \hat{r}_i)$ such that $(\hat{c}_i, \hat{r}_i, t'_i, h_i) \in \mathit{CP}_{\mathit{NF}}$ with $h_i = \mathrm{height}(t'_i) \leq \mathrm{height}(t_i) < h$. By Lemmas 22 and 30, we have $t_i \overset{*}{\leftrightarrow} \hat{c}_i \overset{*}{\leftrightarrow} t'_i$. Also, by Lemma 29, $t'_i \in \mathit{NF}$.
2. Case $s_i \in \mathcal{V}$. Put $t'_i = s_i$ and $h_i = 0$.

Let $t' = f(t'_1, \ldots, t'_n)$. We now derive $t' \in \mathit{NF}$ from $t \in \mathit{NF}$ using Lemma 2. From the definition, $t'_1, \ldots, t'_n \in \mathit{NF}$. Clearly, $t'|_i = t|_i$ whenever $t'|_i \in \mathcal{C}$. Also, $\mathrm{root}(t) = \mathrm{root}(t')$. Thus, it remains to show $\mathit{Patt}(t') \subseteq \mathit{Patt}(t)$, i.e. $t_i = t_j$ whenever $t'_i = t'_j$. Suppose $t'_i = t'_j$.

- Case $t'_i = t'_j \notin \mathcal{V}$. Then $s_i, s_j \in \mathcal{C}$, and thus, $t_i \overset{*}{\leftrightarrow} t'_i$ and $t_j \overset{*}{\leftrightarrow} t'_j$. Hence, $t_i \overset{*}{\leftrightarrow} t'_i = t'_j \overset{*}{\leftrightarrow} t_j$. Then, since $t \in \mathit{SubMinWit}_{\mathcal{R}}$, Proposition 5 leads $t_i = t_j$.
- Case $t'_i = t'_j \in \mathcal{V}$. Then, $s_i = t'_i = t'_j = s_j \in \mathcal{V}$. Thus, we have $t_i \overset{*}{\leftrightarrow} \sigma(s_i) = \sigma(s_j) \overset{*}{\leftrightarrow} t_j$. Since $t \in \mathit{SubMinWit}_{\mathcal{R}}$, Proposition 5 leads $t_i = t_j$.

Thus, we obtain $t' \in \mathit{NF}$. Also, $h' = \mathrm{height}(t') = 1 + \max\{\mathrm{height}(t'_i)\}_i \leq h$.

Since $h' = 1 + \max_i h'_i$, there exists $1 \leq j \leq n$ such that $h_j = h' - 1$. Then $(\hat{c}_j, \hat{r}_j, t_j, h_j)$ is added to $\mathit{CP}_{\mathit{NF}}$ at $H = h' - 1$ by Lemma 29. Thus, the main loop the CP algorithm is performed at $H = h'$. Consider the main loop for $H = h'$. If $\hat{r} \notin Y_{\hat{c}}$ already, it is clear that there exists t'' and $h'' \in \mathbb{N}$ such that $(\hat{c}, \hat{r}, t'', h'') \in \mathit{CP}_{\mathit{NF}}$ $(h'' < h')$. From Lemma 30, we have $t'' \in \mathrm{CP}(\hat{c}, \hat{r})$. Hence the claim established in this case. Suppose $\hat{r} \in Y_{\hat{c}_i}$. Then, by the construction above, $s_i = t'_i$ whenever $s_i \in \mathcal{V}$, and $(\hat{s}_i, \hat{r}_i, t'_i, h_i) \in \mathit{CP}_{\mathit{NF}}$ $(h_i < h)$ whenever $s_i \in \mathcal{C}$. Also, $t' \in \mathit{NF}$. Therefore, the main loop of the CP algorithm yields $(\hat{c}, \hat{r}, t', h') \in \mathit{CP}_{\mathit{NF}}$. ◀

Proof of Lemma 32. Suppose $(\hat{c}_s, \hat{r}_s, s, h_s), (\hat{c}_t, \hat{r}_t, t, h_t) \in \mathit{CP}_{\mathit{NF}}$ and $s = t$. From Lemma 30, we have $\hat{c}_s \overset{*}{\leftrightarrow} s = t \overset{*}{\leftrightarrow} \hat{c}_t$. Thus, $\hat{c}_s \approx \hat{c}_t$, and hence $\hat{c}_s = \hat{c}_t$. By Lemma 30, $s \in \mathrm{CP}(\hat{c}_s, \hat{r}_s) \cup [\![\hat{c}_s]\!]$ and $t \in \mathrm{CP}(\hat{c}_s, \hat{r}_t) \cup [\![\hat{c}_s]\!]$. We distinguish four cases:

1. Case $s \in [\![\hat{c}_s]\!]$ and $t \in [\![\hat{c}_s]\!]$. Then, $s, t \in \mathcal{C}$. Thus, $h_s = h_t = 0$ and $\hat{r}_s = \hat{c}_s = \hat{c}_t = \hat{r}_t$.
2. Case $s \in [\![\hat{c}_s]\!], t \in \mathrm{CP}(\hat{c}_s, \hat{r}_t)$. Then $\mathrm{height}(s) > 0 = \mathrm{height}(t)$, which contradicts $s = t$.
3. Case $s \in \mathrm{CP}(\hat{c}_s, \hat{r}_s), t \in [\![\hat{c}_s]\!]$. Similar to the previous case.

4. Case $s \in \mathrm{CP}(\hat{c}_s, \hat{r}_s), t \in \mathrm{CP}(\hat{c}_s, \hat{r}_t)$. From Lemma 30, we have $\hat{c}_s \rightharpoonup \hat{r}_s \overset{*}{\rightharpoonup} s$ and $\hat{c}_s \rightharpoonup \hat{r}_t \overset{*}{\rightharpoonup} t$. Thus $\mathrm{root}(\hat{r}_s) = \mathrm{root}(s) = \mathrm{root}(t) = \mathrm{root}(\hat{r}_t)$. Thus, one can let $\hat{r}_s = f(u_1, \ldots, u_n)$, $\hat{r}_t = f(v_1, \ldots, v_n)$, $s = f(s_1, \ldots, s_n) = t$. Furthermore, we have $u_i \overset{*}{\rightharpoonup} s_i$ and $v_i \overset{*}{\rightharpoonup} s_i$ for all $1 \le i \le n$. If $s_i \in \mathcal{V}$ then $u_i = s_i = v_i$ by Lemma 29. Suppose $s_i \notin \mathcal{V}$. Then, $u_i, v_i \in \mathcal{C}$, and thus $\hat{u}_i = u_i$ and $\hat{v}_i = v_i$ (as they are subterms of $\hat{r}_s, \hat{r}_t$). Since $u_i \overset{*}{\leftrightarrow} s_i \overset{*}{\leftrightarrow} v_i$, we have $u_i \approx v_i$, and hence $u_i = v_i$. Thus, we obtain $\hat{r}_s = f(u_1, \ldots, u_n) = f(v_1, \ldots, v_n) = \hat{r}_t$. ◀

Proof of Lemma 34. (1) Suppose $s \overset{*}{\leftrightarrow} t$. By Lemma 29, $\mathrm{T}^{CP} \cap \mathcal{V} = \emptyset$. Thus, we can distinguish four cases. The case $s, t \in \mathrm{T}^{CP}$ follows from the assumption, and the case $s, t \in \mathcal{V}$ follows from the consistency of $\mathcal{R}$. If $s \in \mathrm{T}^{CP}$ and $t \in \mathcal{V}$ then $(\hat{c}, \hat{r}, s, h_s) \in CP_{NF}$ for some $\hat{c}, \hat{r}, h_s$, and thus $\hat{c} \overset{*}{\leftrightarrow} s \overset{*}{\leftrightarrow} t \in \mathcal{V}$ by Lemma 30. This contradicts the consistency of $\mathcal{R}$. The case $t \in \mathrm{T}^{CP}$ and $s \in \mathcal{V}$ follows similarly. (2) Suppose $c \overset{*}{\leftrightarrow} t$ for $c \in \mathcal{C}$ and $t \in \mathit{SubMinWit}_{\mathcal{R}}$. Then, by Theorem 26, $t \in \mathrm{CP}(c, r) \cup [\![c]\!]$. Then, by Lemma 31, $(\hat{c}, \hat{r}, s, \_) \in CP_{NF}$ for some s. Then, we have $s \in \mathrm{T}^{CP}$ and $c \overset{*}{\leftrightarrow} \hat{c} \overset{*}{\leftrightarrow} s$ by Lemma 30. If $s' \in \mathrm{T}^{CP}$ and $c \overset{*}{\leftrightarrow} s'$, then $s \overset{*}{\leftrightarrow} c \overset{*}{\leftrightarrow} s'$, and thus $s = s'$ by (1). ◀

Proof of Lemma 38. (1), (2) are immediate. (3) If $t|_i$ is not equivalent to a constant, from definitions of ψ and ψ', we have $\psi(t)|_i = \psi'(t)|_i$. If $t|_i$ is equivalent to a constant c, we have $c \overset{*}{\leftrightarrow} t|_i$. From the definition of ψ, $t|_i = \psi(t)|_i$. Also, from the definition of ψ', $c \overset{*}{\leftrightarrow} \psi'(t)|_i$. Hence $\psi(t)|_i \overset{*}{\leftrightarrow} \psi'(t)|_i$. (4) Clear from (1) and (3). ◀

Proof of Lemma 39. From Lemma 29, we have $t'_1, \ldots, t'_n \in NF$. Thus it suffices to show there's no root rewrite step from $\psi'(s)$. It is known that $\psi(s) \in NF$ [17]; thus (as $\mathcal{R}$ is flat) the claim follows if we have: (a) $t'_i \in \mathcal{C}$ implies $t'_i = t_i$ for all $1 \le i \le n$ and (b) $\mathit{Patt}(\psi'(s)) \subseteq \mathit{Patt}(\psi(s))$. To show (a), suppose $t'_i \in \mathcal{C}$. Then $t'_i \in \mathcal{C} \cap NF$. From Lemma 38, $t_i \overset{*}{\leftrightarrow} t'_i$ holds. Also, by definition of ψ, $u_i = t_i$ holds. Thus, $u_i = t_i \overset{*}{\leftrightarrow} t'_i \in \mathcal{C} \cap NF$. Since $s = f(u_1, \ldots, u_n) \in \mathit{SubMinWit}_{\mathcal{R}}$, $t_i = u_i = t'_i$ holds by Lemma 6. Next, we show (b). Suppose $t'_i = t'_j$. We distinguish two cases:

- Case $t'_i = t'_j \notin \mathcal{V}$. Then u_i, u_j are equivalent to constants, and hence $u_i = t_i$ and $u_j = t_j$ by the definition of ψ. By $u_i = t_i \overset{*}{\leftrightarrow} t'_i = t'_j \overset{*}{\leftrightarrow} t_j = u_j$ and $s \in \mathit{SubMinWit}_{\mathcal{R}}$, we obtain $t_i = u_i = u_j = t_j$ by Proposition 5.
- Case $t'_i = t'_j \in \mathcal{V}$. Then $t_i = t'_i = t'_j = t_j$ by definitions of ψ and ψ'. ◀

Proof of Lemma 40. Let $\langle s, t\rangle$ be a minimal witness that is not equivalent to a constant. By Proposition 36, either $\langle \psi(s), y\rangle$, $\langle y, \psi(t)\rangle$ or $\langle \psi(s), \psi(t)\rangle$ is a witness for some variable y. By Lemma 38, $\psi(s) \overset{*}{\leftrightarrow} \psi'(s)$ and $\psi(t) \overset{*}{\leftrightarrow} \psi'(t)$ hold. We distinguish three cases.

- Case $\langle \psi(s), y\rangle$ is a witness. Then $\psi(s) \notin \mathcal{V}$ as $\mathcal{R}$ is consistent. By the definitions, we have $\psi(s) \notin \mathcal{V}$ iff $s \notin \mathcal{V}$ iff $\psi'(s) \notin \mathcal{V}$, and thus $\psi'(s) \in NF \setminus \mathcal{V}$ by Lemma 39. Since $\psi'(s) \overset{*}{\leftrightarrow} \psi(s) \overset{*}{\leftrightarrow} y$, $\langle \psi'(s), y\rangle$ is also a witness.
- Case $\langle y, \psi(t)\rangle$ is a witness. Same as the previous case.
- Case $\langle \psi(s), \psi(t)\rangle$ is a witness. If $\psi(s) \in \mathcal{V}$ or $\psi(t) \in \mathcal{V}$, then one can use the same argument as above. So, suppose $\psi(s), \psi(t) \notin \mathcal{V}$. Then, $\psi'(s), \psi'(t) \notin \mathcal{V}$ as above, and thus $\psi'(s), \psi'(t) \in NF$ by Lemma 39. Since $\psi'(s) \overset{*}{\leftrightarrow} \psi(s) \overset{*}{\leftrightarrow} \psi(t) \overset{*}{\leftrightarrow} \psi'(t)$, it remains to show $\psi'(s) \neq \psi'(t)$. If $\mathrm{root}(\psi(s)) \neq \mathrm{root}(\psi(t))$ or if $\psi(s)|_i \neq \psi(t)|_i$ with $\psi(s)|_i \in \mathcal{V}$ or $\psi(t)|_i \in \mathcal{V}$ for some $i \in \mathbb{N}$, then it follows from Lemma 38 that $\psi'(s) \neq \psi'(t)$. Consider the case where roots of $\langle \psi(s), \psi(t)\rangle$ and all direct variable subterms are same. Then, we have $\psi(s)|_i \neq \psi(t)|_i$ and $\psi(s)|_i, \psi(t)|_i \notin \mathcal{V}$ for some $i \in \mathbb{N}$. Then by definition of ψ, we have $s|_i = \psi(s)|_i$ and $t|_i = \psi(t)|_i$. Thus, $s|_i \neq t|_i$. Then, if $\psi'(s)|_i = \psi'(t)|_i$, then $s|_i = \psi(s)|_i \overset{*}{\leftrightarrow} \psi'(s)|_i = \psi'(t) \overset{*}{\leftrightarrow} \psi'(t)|_i = t|_i$ by Lemma 38. Then $\langle s|_i, t|_i\rangle$ becomes a witness, which contradicts the minimaliy of $\langle s, t\rangle$. ◀

Proof of Lemma 42. From Proposition 4, there exists a proof $s \xleftrightarrow{*} t$ which has at most one root rewrite step. Clearly, one can assume the root step by a trivial equation have been removed. Suppose $s \xleftrightarrow{*} t$ does not have a root rewrite step. Then, $s = f(s_1, \dots s_n)$, $t = f(t_1, \dots t_n)$ and $s_i \xleftrightarrow{*} t_i$ for all $1 \leq i \leq n$ for some f and s_i, t_i $(1 \leq i \leq n)$. Then, $s, t \in \mathrm{T}_f^{CP}$ and hence $s_i, t_i \in \mathrm{T}^{CP} \cup \mathcal{V}$ for all $1 \leq i \leq n$. Then, from Lemma 34, we know $s_i = t_i$ for all $1 \leq i \leq n$. Thus, $s = f(s_1, \dots s_n) = f(t_1, \dots t_n) = t$, but this contradicts that $\langle s, t \rangle$ is a witness. ◀

Proof of Lemma 43. Let $\langle s, t \rangle \in CW$. Then, there exists $l \approx r \in \widehat{E_{\mathcal{R}}}$ such that $\psi''(l) = s$ and $\psi''(r) = t$. It immediately follows from the definition of ψ'' that $\psi''(l) \xleftrightarrow{*} l$ and $r \xleftrightarrow{*} \psi''(r)$. Hence $s \xleftrightarrow{*} t$. The claim is an easy consequence of this. ◀

Proof of Lemma 45. Step 1 terminates and a flat TRS $\mathcal{R}$ and a finite set $\widehat{E_{\mathcal{R}}}$ is computed [8, 20]. Step 2 terminates by Lemma 27, and a finite set CP_{NF} is obtained. As the set CP_{NF} is finite, Step 3 terminates. As a consequence of Step 3, $\tilde{t}$ is defined uniquely. Thus, by the finiteness of CP_{NF} and $\widehat{E_{\mathcal{R}}}$, a finite set CW can be computed. By Lemma 43, one can check whether there exists a witness in CW. Thus, Step 4 terminates. ◀

Proof of Theorem 46. It suffices to decide that the flat TRS $\mathcal{R}$ obtained by the transformation has the UNC property or not. If $\mathcal{R}$ is inconsistent, then $\mathcal{R}$ is not UNC. In this case, UNC is returned at the Step 1 of the procedure. For the rest of the procedure, one can assume $\mathcal{R}$ is consistent. Suppose there exists $(\hat{c}, \hat{r}, t, h) \in CP_{NF}$ such that t has a direct variable subterm or there exist $(\hat{c}, \hat{r}_s, s, h_s), (\hat{c}, \hat{r}_t, t, h_t) \in CP_{NF}$ such that $s \neq t$. Then $\mathcal{R}$ is not UNC by Theorem 33. In this case, Non-UNC is returned as the Step 3 of the procedure. Suppose this does not hold. Then, by Theorem 33, there exists no minimal witness equivalent to a constant. If there exists a minimal witness that is not equivalent to a constant, then there exists a witness in CW by Theorem 44. Thus, in this case, Non-UNC is returned as the Step 4 of the procedure. Suppose otherwise. Clearly, if a witness exists, then there exists a minimal witness. Thus, one can conclude that there is no witness, and hence $\mathcal{R}$ has the UNC property. In this case, UNC is returned as the Step 5 of the procedure. ◀

B Examples in our Experiments

Bellow we present examples used in our experiments given in Table 2.

$$
\begin{aligned}
\mathcal{R}_1 &= \{\mathsf{a} \to \mathsf{b},\ \mathsf{a} \to \mathsf{c},\ \mathsf{c} \to \mathsf{c},\ \mathsf{d} \to \mathsf{c},\ \mathsf{d} \to \mathsf{e}\}\\
\mathcal{R}_2 &= \{\mathsf{f}(x,\mathsf{a}) \to \mathsf{a},\ \mathsf{a} \to \mathsf{b}\}\\
\mathcal{R}_3 &= \{\mathsf{g}(y) \to \mathsf{f}(x,y)\}\\
\mathcal{R}_4 &= \{\mathsf{a} \to \mathsf{b},\ \mathsf{b} \to \mathsf{a},\ \mathsf{f}(x,y) \to \mathsf{a}\}\\
\mathcal{R}_5 &= \{\mathsf{a} \to \mathsf{b},\ \mathsf{b} \to \mathsf{a},\ \mathsf{f}(x,y) \to \mathsf{a},\ \mathsf{f}(x,y) \to z\}\\
\mathcal{R}_6 &= \{\mathsf{f}(\mathsf{g}(\mathsf{a}),y) \to y\}\\
\mathcal{R}_7 &= \{\mathsf{f}(\mathsf{g}(\mathsf{h}(\mathsf{a})),y) \to y\}\\
\mathcal{R}_8 &= \left\{\begin{array}{l} \mathsf{f}(x,y) \to \mathsf{a},\ \mathsf{f}(\mathsf{k}(\mathsf{l}(\mathsf{a}_1,\mathsf{a}_2),\mathsf{a}_1,\mathsf{a}_3),y) \to \mathsf{a},\ \mathsf{f}(x,\mathsf{u}) \to \mathsf{a},\ \mathsf{a} \to \mathsf{g}(\mathsf{b}_1,\mathsf{u},x),\ \mathsf{b}_1 \to \mathsf{b},\\ \mathsf{b} \to \mathsf{h}(x,\mathsf{d}),\ \mathsf{h}(y,\mathsf{d}) \to \mathsf{i}(\mathsf{c}),\ \mathsf{u} \to \mathsf{j}(\mathsf{v}) \end{array}\right\}\\
\mathcal{R}_9 &= \left\{\begin{array}{l} \mathsf{f}(x,y) \to \mathsf{a},\ \mathsf{f}(\mathsf{k}(\mathsf{l}(\mathsf{a}_1,\mathsf{a}_2),\mathsf{a}_1,\mathsf{a}_3),y) \to \mathsf{a},\ \mathsf{f}(x,\mathsf{u}) \to \mathsf{a},\ \mathsf{a} \to \mathsf{g}(\mathsf{b}_1,\mathsf{u}),\ \mathsf{b}_1 \to \mathsf{b}\\ \mathsf{b} \to \mathsf{h}(x,\mathsf{d}),\ \mathsf{h}(y,\mathsf{d}) \to \mathsf{i}(\mathsf{c}),\ \mathsf{u} \to \mathsf{j}(\mathsf{v}),\ \mathsf{m}(x,x) \to \mathsf{n}(y) \end{array}\right\}\\
\mathcal{R}_{10} &= \left\{\ \mathsf{f}(x,y) \to \mathsf{g}(\mathsf{h}(\mathsf{a}))\ \right\}\\
\mathcal{R}_{11} &= \left\{\begin{array}{l} \mathsf{f}(x,y) \to \mathsf{a},\ \mathsf{f}(\mathsf{k}(\mathsf{l}(\mathsf{a}_1,\mathsf{a}_2),\mathsf{a}_1,\mathsf{a}_3),y) \to \mathsf{a},\ \mathsf{f}(x,\mathsf{u}) \to \mathsf{a},\ \mathsf{a} \to \mathsf{g}(\mathsf{b}_1,\mathsf{u}),\ \mathsf{b}_1 \to \mathsf{b}\\ \mathsf{b} \to \mathsf{h}(x,\mathsf{d}).\ \mathsf{h}(y,\mathsf{d}) \to \mathsf{i}(\mathsf{c}),\ \mathsf{u} \to \mathsf{j}(\mathsf{v}) \end{array}\right\}\\
\mathcal{R}_{12} &= \{\mathsf{f}(x) \to \mathsf{g}(\mathsf{a},x),\ \mathsf{f}(x) \to \mathsf{g}(x,\mathsf{a}),\ \mathsf{g}(x,x) \to \mathsf{f}(x)\}\\
\mathcal{R}_{13} &= \{\mathsf{f}(x,x) \to \mathsf{g}(x),\ \mathsf{f}(\mathsf{a},\mathsf{b}) \to \mathsf{g}(\mathsf{a})\}
\end{aligned}
$$

C Implementation of Existing Decision Procedure

Here, we briefly explain our implementation of the existing decision procedure [17] and illustrate why it suffers a bottleneck of having a sheer number of candidates for the witness.

We use the following algorithm:

1. Transform a shallow TRS into a flat TRS $\mathcal{R}$ preserving UNC.
2. Calculate $\widehat{E_{\mathcal{R}}}$. (Here, the program also judges whether $\mathcal{R}$ is consistent.)
3. Add new constants $\mathcal{C}_{new}$ to $\mathcal{F}$ of the size $|\mathcal{C}_{new}| = 2 * \alpha^{h-1}$, where $h = \max(1, |\mathcal{C}|)$ and $\alpha = \max\{\text{arity}(f) \mid f \in \mathcal{F}\}$.
4. Make all ground terms over $\mathcal{F} \cup \mathcal{C}_{new}$ of height $\leq h$.
5. Check whether there exists a pair of such terms that is a witness.

Below, we provide a (straight) estimation of the number of candidates for the witness for our examples $\mathcal{R}_6$ and $\mathcal{R}_7$. These examples are very similar (see the previous section) but our implementation of the existing procedure succeeds for $\mathcal{R}_6$ but fails for $\mathcal{R}_7$.

▶ **Example 50.** Consider the TRS $\mathcal{R}_6$. The UNC-preserving flatting translation makes the following flat TRS $\{\mathsf{f}(\mathsf{c}_0, x) \to x, \mathsf{g}(\mathsf{a}) \to \mathsf{c}_0\}$. Since we have $h = |\mathcal{C}| = 2$ and $\alpha = 2$, we add $2 * 2^1 = 4$ new constants. Hence, we consider ground terms over function symbols f, g and $2 + 4 = 6$ constants. We have 6 ground terms of height 0. There are 6 ground terms of height 1 having root g and $6 \times 6 = 36$ ground terms of height 1 having root f. Thus, the number of ground terms of height ≤ 2 to be used to constructing the candidates is $6 + 6 + 42 = 48$. This means there are $48 \times 47 = 2{,}256$ candidates to be checked.

▶ **Example 51.** Consider the TRS $\mathcal{R}_7$. The UNC-preserving flatting translation makes the following flat TRS $\{\mathsf{f}(\mathsf{c}_1, x) \to x, \mathsf{g}(\mathsf{c}_0) \to \mathsf{c}_1, \mathsf{h}(\mathsf{a}) \to \mathsf{c}_0\}$. Since we have $h = |\mathcal{C}| = 3$ and

$\alpha = 2$, we add $2 * 2^2 = 8$ new constants. Hence, we consider ground terms over function symbols f, g and $3 + 8 = 11$ constants. We have 11 ground terms of height 0. There are 11 ground terms of height 1 having root g, and $11 \times 11 = 121$ ground terms of height 1 having root f. Thus, the number of ground terms of height 1 to be used is $11 + 121 = 132$. There are 132 ground terms of height 2 having root g. Now we calculate the number of ground terms of height 2 having root f distinguishing three cases. The number of terms $\mathsf{f}(s, t)$ with $\mathit{height}(s) = 0$, $\mathit{height}(t) = 1$ is $11 \times 121 = 1331$; so is the number of terms $\mathsf{f}(s, t)$ with $\mathit{height}(s) = 1$, $\mathit{height}(t) = 0$. The number of terms $\mathsf{f}(s, t)$ with $\mathit{height}(s) = \mathit{height}(t) = 1$ is $121 \times 121 = 14,642$. Thus, we have $132 + 1331 + 1331 + 14,641 = 17,435$ ground terms of height 2. Hence,the number of ground terms of height ≤ 2 be used to constructing the candidates is $11 + 121 + 17435 = 17567$. This means there are 17567×17566 (≈ 300 millions) candidates to be checked.

Modules over Monads and Operational Semantics

André Hirschowitz
Université Côte d'Azur, CNRS, Nice, France
https://math.unice.fr/~ah

Tom Hirschowitz
Univ. Grenoble Alpes, Univ. Savoie Mont Blanc, CNRS, LAMA, 73000 Chambéry, France
https://www.lama.univ-savoie.fr/pagesmembres/hirschowitz

Ambroise Lafont
University of New South Wales, Sydney, Australia
https://amblafont.github.io

Abstract

This paper is a contribution to the search for efficient and high-level mathematical tools to specify and reason about (abstract) programming languages or calculi. Generalising the reduction monads of Ahrens et al., we introduce transition monads, thus covering new applications such as $\bar{\lambda}\mu$-calculus, π-calculus, Positive GSOS specifications, differential λ-calculus, and the big-step, simply-typed, call-by-value λ-calculus. Finally, we design a suitable notion of signature for transition monads.

2012 ACM Subject Classification Theory of computation → Semantics and reasoning; Theory of computation → Categorical semantics; Theory of computation → Operational semantics

Keywords and phrases Operational semantics, Category theory

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.12

Acknowledgements We thank the anonymous referees for their constructive comments.

1 Introduction

The search for a mathematical notion of programming language goes back at least to Turi and Plotkin [25], who coined the name "Mathematical Operational Semantics", and explained how known classes of well-behaved rules for structural operational semantics, such as GSOS [7], can be categorically understood and specified via distributive laws and bialgebras. Their initial framework did not cover variable binding, and several authors have proposed variants which do [14, 13, 24], treating examples like the π-calculus. However, none of these approaches covers higher-order languages like the λ-calculus.

In recent work, following previous work on modules over monads for syntax with binding [18, 2] (see also [1]), Ahrens et al. [3] introduce **reduction monads**, and show how they cover several standard variants of the λ-calculus. Furthermore, as expected in similar contexts, they propose a mechanism for specifying reduction monads by suitable signatures.

Our starting point is the fact that already the call-by-value λ-calculus does not form a reduction monad. Indeed, in this calculus, variables are placeholders for values but not for λ-terms; in other words, reduction, although it involves general terms, is stable under substitution by values only.

In the present work, we generalise reduction monads to what we call **transition monads**. The main new ingredients of our generalisation are as follows.

- We now have two kinds of terms, called **placetakers** and **states**: variables are placeholders for our placetakers, while transitions relate states. Typically, in call-by-value, small-step λ-calculus, placetakers are values, while states are general terms.
- We also have a set of types for placetakers, and a possibly different set of types for states. Typically, in call-by-value, simply-typed λ-calculus, both sets of types coincide and are

given by simple types, while in $\overline{\lambda}\mu$-calculus, we have two placetaker types, one for terms and one for stacks, and one state type, for processes.

- We in fact have two possibly different kinds of states, **source** states and **target** states, so that a transition now relates a source state to a target state. Typically, in call-by-value, big-step λ-calculus, source states are general terms, while target states are values.
- The relationship between placetakers and states is governed by two functors S_1 and S_2, as follows: given an object X (for variables), we have an object $T(X)$ of placetakers ("with free variables in X"), and the corresponding objects of source and target states are respectively $S_1(T(X))$ and $S_2(T(X))$ (see §2.2).

Reduction monads correspond to the untyped case with $S_1 = S_2 = \mathrm{Id}_{\mathbf{Set}}$. In §2.1, after giving a "monadic" definition of transition monads in terms of **relative monads** [4], we provide a "modular" definition (in terms of modules over monads), which we prove equivalent in Proposition 6. From the modular point of view, a transition monad consists of a **placetaker** monad T, two **state** functors S_1, S_2, a **transition T-module** R, and two T-module morphisms $src : R \to S_1T$ and $tgt : R \to S_2T$. Such a triple (R, src, tgt) is thus an object of the slice category of T-modules over $S_1T \times S_2T$.

In §2.2, we present a series of examples of transition monads: $\overline{\lambda}\mu$-calculus, simply-typed λ-calculus (in its call-by-value, big-step variant), π-calculus (as an unlabelled transition system), and differential λ-calculus.

Finally, in §2.3, we organise transition monads into categories. For the category of transition monads over a fixed triple (T, S_1, S_2), we take the slice category of T-modules alluded to above. Then, we wrap together these "little" slice categories into what we call a **record** category of transition monads.

We then proceed to the main concern of this work: the specification of transition monads via suitable signatures. For this, we start in §3 by proposing a new, abstract notion of **semantic signature** over a category $\mathbf{C}$. A semantic signature $S = (\mathbf{E}, U)$ over $\mathbf{C}$ consists of a category $\mathbf{E}$ of **algebras**, together with a **forgetful**[1] functor $U : \mathbf{E} \to \mathbf{C}$, such that $\mathbf{E}$ has an initial object $S^{\circledast}$: we think of such a semantic signature as **specifying** the object $S^* := U(S^{\circledast})$ underlying the initial algebra. Abstracting over this generating procedure, we introduce **registers** in §3. A register R for the category $\mathbf{C}$ consists of a class $\mathbf{Sig}_R$ of **signatures**, together with a map associating to each signature S a semantic signature $[\![S]\!]_R$, say $\mathbf{U}_S : S\text{-alg} \to \mathbf{C}$. Just as for semantic signatures, omitting $[\![-]\!]_R$ for readability, we think of a signature S as specifying the object $S^* = \mathbf{U}_S(S^{\circledast})$.

We may now state our achievement properly: we construct a register for transition monads, containing signatures specifying the desired examples. Towards this goal, we start in §4 by designing registers for monads and functors, relying on Ahrens et al. [2] and Fiore and Hur [11]. This will allow us to efficiently specify the base components (T, S_1, S_2) of the desired example transition monads, separately. We continue in §5 by presenting some general constructions of registers, whose combination will yield a register for transition monads. First, the product construction allows us to group the signatures of T, S_1, and S_2 into a single signature for the triple (T, S_1, S_2). Then, we introduce in §5.2 a register for a slice category of modules over a monad. This yields a register for transition monads over a fixed triple (T, S_1, S_2), since these form such a slice category. Finally, in §5.3 we address the task of grouping into a single signature the signatures for the triple (T, S_1, S_2) and for the transition module (R, s, t) over it. For this, we propose a **record** construction for registers, which binds together registers on the base and on fibres of a record category. Applying this

[1] Here "algebra" and "forgetful" have no technical meaning and are chosen by analogy.

to the previously constructed registers for our base product of three categories and our fibre slice categories of modules, we give in Definition 63 our final register for the category of transition monads (with fixed sets of types). This register covers all examples of transition monads from §2.2, as we demonstrate in the appendix.

Related work

Beyond the already evoked related work [3, 25, 11], there is a solid body of work on categorical approaches to rewriting with variable binding, which only covers transition relations that are stable under arbitrary contexts, e.g., Hamana [16], T. Hirschowitz [19], and Ahrens [1]. Regarding signatures, Fiore [12], Altenkirch et al. [5], and Garner [15] use notions of signatures for languages with dependent types, which may provide an alternative approach to the specification of operational semantics systems. Finally, let us mention that a preliminary account of the present work appears in the third author's PhD thesis [20, Chapter 6].

Notations

In the following, **Set** denotes the category of sets, $[\mathbf{Set}^{\mathbb{P}}, \mathbf{Set}^{\mathbb{Q}}]_f$ denotes the locally small category of finitary functors $\mathbf{Set}^{\mathbb{P}} \to \mathbf{Set}^{\mathbb{Q}}$ for any sets $\mathbb{P}$ and $\mathbb{Q}$.

The category of finitary monads on $\mathbf{C}$ is denoted by $\mathbf{Mnd}_f(\mathbf{C})$, or sometimes just $\mathbf{Mnd}_f$ when $\mathbf{C}$ is clear from context. Given a monad T on $\mathbf{C}$, the category of $\mathbf{D}$-valued (finitary) T-modules is denoted by $T\text{-}\mathbf{Mod}_f(\mathbf{D})$, where we recall [18] that such a T-module consists of a finitary functor $M : \mathbf{C} \to \mathbf{D}$ equipped with a right T-action $M \circ T \to M$ satisfying some coherence conditions.

For any sequence $p_1, \ldots, p_n$ in a set $\mathbb{P}$, for any monad T on $\mathbf{Set}^{\mathbb{P}}$ and $\mathbf{D}$-valued T-module M, we denote by $M^{(p_1,\ldots,p_n)}$ the D-valued T-module defined by $M^{(p_1,\ldots,p_n)}(X) = M(X + \mathbf{y}_{p_1} + \cdots + \mathbf{y}_{p_n})$, where $\mathbf{y} : \mathbb{P} \to \mathbf{Set}^{\mathbb{P}}$ is the embedding defined by $\mathbf{y}_p(q) = 1$ if $p = q$ and $\emptyset$ otherwise. If $\mathbb{P}$ is a singleton, we abbreviate this to $M^{(n)}$.

2 Transition monads

2.1 Definition of transition monads

In this section, we introduce the main new mathematical notion of the paper which was already motivated by the case of the call-by-value, simply-typed, big-step λ-calculus in §1: transition monads. We first describe the various components of a transition monad. Then we give the monadic definition. And finally we give a modular description, which is better suited for later use.

Placetakers and states. In standard λ-calculus, we have terms, variables are placeholders for terms, and transitions relate a source term to a target term. In a general transition monad we still have variables and transitions, but placetakers for variables and endpoints of transitions can be of a different nature, which we phrase as follows: variables are placeholders for **placetakers**, while transitions relate a **source state** with a **target state**.

The categories for placetakers and for states. In standard λ-calculi, we have a set $\mathbb{T}$ of types for terms (and variables); for instance in the untyped version, $\mathbb{T}$ is a singleton. Accordingly, terms form a **monad** on the category $\mathbf{Set}^{\mathbb{T}}$.

Similarly, in a general transition monad we have a set $\mathbb{P}$ of placetaker types, and a set $\mathbb{S}$ of state types. For example, for simply-typed λ-calculus, $\mathbb{P} = \mathbb{S}$ is the set of simple types.

Placetakers form a **monad** on the category $\mathbf{Set}^{\mathbb{P}}$.

The object of variables. In our (monadic) view of the untyped λ-calculus, there is a (variable!) set of variables and everything is parametric in this "variable set". Similarly, in a general transition monad R, there is a "variable object" V in $\mathbf{Set}^{\mathbb{P}}$ and everything is functorial in this variable object. In particular, we have a placetaker object $T_R(V)$ in $\mathbf{Set}^{\mathbb{P}}$ and a source (resp. target) state object in $\mathbf{Set}^{\mathbb{S}}$, both depending upon the variable object.

The state functors S_1 and S_2. While in the λ-calculus, states are the same as placetakers, in a general transition monad, they may differ, and more precisely both state objects are derived from the placetaker object by applying the **state functors** $S_1, S_2 : \mathbf{Set}^{\mathbb{P}} \to \mathbf{Set}^{\mathbb{S}}$.

The transitions. In standard λ-calculi, there is a (typed!) set of transitions, which yields a graph on the set of terms. That is to say, if V is the variable object, and $LC(V)$ the term object, there is a transition object $\mathit{Trans}(V)$ equipped with two morphisms $\mathit{src}_V, \mathit{trg}_V : \mathit{Trans}(V) \to LC(V)$. Note that we consider "proof-relevant" transitions here, in the sense that two different transitions may have the same source and target. (Appendix G discusses how proof irrelevance can be recovered.)

In a general transition monad R, we still have a transition object $\mathit{Trans}_R(V)$, which now lives in $\mathbf{Set}^{\mathbb{S}}$, together with state objects $S_1(T_R(V))$ and $S_2(T_R(V))$, so that src_V and trg_V form a span $S_1(T_R(V)) \leftarrow \mathit{Trans}_R(V) \to S_2(T_R(V))$.

The S-graph of transitions. Now we rephrase the previous status of transitions in terms of a graph-like notion which we call S-graph: here $S := (S_1, S_2)$ is the pair of state functors. In the untyped λ-calculus, $\mathit{Trans}(V)$ and the maps src_V and trg_V turn the term object $LC(V)$ into a graph (which depends functorially on the variable object V). For an analogous statement in a general transition monad, we will use the following.

▶ **Definition 1.** *For any pair $S = (S_1, S_2)$ of functors $\mathbf{Set}^{\mathbb{P}} \to \mathbf{Set}^{\mathbb{S}}$, an S-**graph** over an object $V \in \mathbf{Set}^{\mathbb{P}}$ consists of*

- *an object E (of **edges**) in $\mathbf{Set}^{\mathbb{S}}$, and*
- *a span $S_1(V) \leftarrow E \to S_2(V)$, which we alternatively view as a morphism $\partial : E \to S_1(V) \times S_2(V)$.*

*An S-**graph** consists of an object $V \in \mathbf{Set}^{\mathbb{P}}$ and an S-graph over V.*

Now we can say that in a general transition monad, transitions form an S-graph over the placetaker object (the whole thing depending upon the variable object…).

The category of S-graphs. A reduction monad (in particular the untyped λ-calculus) is just a monad relative to the "discrete graph" functor from sets to graphs [3]. In order to have a similar definition for transition monads, the last missing piece is the category of S-graphs, which we now describe. A morphism $G \to G'$ of S-graphs consists of a morphism for vertices $f : V_G \to V_{G'}$ together with a morphism for edges $g : E_G \to E_{G'}$ making the following diagram commute.

$$\begin{tikzcd}
E_G \arrow[r, "g"] \arrow[d, "\partial_G"'] & E_{G'} \arrow[d, "\partial_{G'}"] \\
S_1(V_G) \times S_2(V_G) \arrow[r, "S_1(f)\times S_2(f)"'] & S_1(V_{G'}) \times S_2(V_{G'})
\end{tikzcd}$$

▶ **Proposition 2.** *For any pair $S = (S_1, S_2)$ of functors $\mathbf{Set}^{\mathbb{P}} \to \mathbf{Set}^{\mathbb{S}}$, S-graphs form a category S-$\mathbf{Gph}$.*

Monadic definition of transition monad. First of all, let us recall [4] that, given any functor $J : \mathbf{C} \to \mathbf{D}$, a **monad relative to** J, or J**-relative monad**, consists of

- an object mapping $T : \mathbf{ob}(\mathbf{C}) \to \mathbf{ob}(\mathbf{D})$, together with
- morphisms $\eta_X : J(X) \to T(X)$, and
- for each morphism $f : J(X) \to T(Y)$, an **extension** $f^\star : T(X) \to T(Y)$,

satisfying coherence conditions. Any J-relative monad T has an underlying functor $\mathbf{C} \to \mathbf{D}$, and is said **finitary** when this functor is.

We will consider monads relative to functors of the following form.

▶ **Definition 3.** *For any functors $S_1, S_2 : \mathbf{Set}^{\mathbb{P}} \to \mathbf{Set}^{\mathbb{S}}$, letting $S = (S_1, S_2)$, the* ***discrete*** *S**-graph** functor $J_S : \mathbf{Set}^{\mathbb{P}} \to \mathbf{Set}^{\mathbb{S}}$ maps any $V \in \mathbf{Set}^{\mathbb{P}}$ to the S-graph on V with no edges.*

Now we are ready to deliver a first, monadic definition of transition monad.

▶ **Definition 4.** *A* ***monadic transition monad*** *over* $(\mathbb{P}, \mathbb{S})$ *consists of*

- *two finitary functors $S_1, S_2 : \mathbf{Set}^{\mathbb{P}} \to \mathbf{Set}^{\mathbb{S}}$, and*
- *a finitary J_S-relative monad, where $S = (S_1, S_2)$.*

Given any J_S-relative monad T, we think of $T(X)$ as having terms with free variables in X as vertices, with all transitions between them as edges. A morphism $\sigma : J_S(X) \to T(Y)$ amounts to a mapping from X to terms in $T(Y)$, i.e., a substitution, and its extension $T(X) \to T(Y)$ models the action of σ both on terms and on transitions.

Modular definition of transition monad. The monadic definition just given does not mention explicitly one crucial feature we had mentioned earlier: the monad of placetakers. In order to clarify this point, we give an alternative "modular" definition.

▶ **Definition 5.** *A* ***transition monad*** *over* $(\mathbb{P}, \mathbb{S})$ *consists of*

- *two finitary functors $S_1, S_2 : \mathbf{Set}^{\mathbb{P}} \to \mathbf{Set}^{\mathbb{S}}$,*
- *a finitary monad T on $\mathbf{Set}^{\mathbb{P}}$, called the* ***placetaker*** *monad,*
- *a finitary T-module $R : \mathbf{Set}^{\mathbb{P}} \to \mathbf{Set}^{\mathbb{S}}$, called the* ***transition*** *module,*
- *a* ***source*** *T-module morphism $src : R \to S_1 T$,*
- *a* ***target*** *T-module morphism $tgt : R \to S_2 T$.*

This is the definition that we use in the rest of the paper.

▶ **Proposition 6.** *Modular and monadic transition monads are in one-to-one correspondence.*

Proof. See Appendix F. ◀

2.2 Examples of transition monads

In this section, we introduce informally a few example transition monads, which will be more rigorously defined in the appendix.

2.2.1 $\overline{\lambda}\mu$-calculus

The $\overline{\lambda}\mu$-calculus [17] is an example with two placetaker types. Its grammar is given by

$$\begin{array}{ccc} \text{Processes} & \text{Programs} & \text{Stacks} \\ c ::= \langle e|\pi\rangle & e ::= x \mid \mu\alpha.c \mid \lambda x.e & \pi ::= \alpha \mid e\cdot\pi, \end{array}$$

where x and α range over two disjoint sets of variables, called **stack** and **program** variables respectively. Both constructions μ and λ bind their variable in the body. There are two transition rules: $\langle\mu\alpha.c|\pi\rangle \to c[\alpha\mapsto\pi]$ $\langle\lambda x.e|e'\cdot\pi\rangle \to \langle e[x\mapsto e']|\pi\rangle$.

Let us show how this calculus gives rise to a transition monad. First, there are two placetaker types, for programs and stacks, so $\mathbb{P} = 2 = \{\mathbf{p},\mathbf{s}\}$. A variable object is an element of $\mathbf{Set}^{\mathbb{P}}$, that is, a pair of sets: the first one gives the available free program variables, and the second one the available free stack variables. The syntax may be viewed as a monad $T : \mathbf{Set}^2 \to \mathbf{Set}^2$: given a variable object $X = (X_{\mathbf{p}}, X_{\mathbf{s}}) \in \mathbf{Set}^2$, the placetaker object $(T(X)_{\mathbf{p}}, T(X)_{\mathbf{s}}) \in \mathbf{Set}^2$ consists of the sets of program and stack terms with free variables in X, up to bound variable renaming. As usual, monad multiplication is given by capture-avoiding substitution.

For transitions, source and target states are processes, so there is only one state type: $\mathbb{S} = 1$. Furthermore, processes are pairs of a program and a stack, so that, setting $S_1(A) = S_2(A) = A_{\mathbf{p}} \times A_{\mathbf{s}}$, we get $S_i(T(X)) = T(X)_{\mathbf{p}} \times T(X)_{\mathbf{s}}$ for $i = 1, 2$ as desired. Finally, transitions with free variables in X form a graph with vertices in $T(X)_{\mathbf{p}} \times T(X)_{\mathbf{s}}$, which we model as a map $\langle src_X, tgt_X\rangle : Trans(X) \to (T(X)_{\mathbf{p}} \times T(X)_{\mathbf{s}})^2$. This family is natural in X and commutes with substitution, hence forms a T-module morphism. We thus have a transition monad.

2.2.2 The π-calculus

For an example involving equations on placetakers, let us recall the following standard presentation of π-calculus [23]. The syntax for **processes** is given by

$$P, Q ::= 0 \mid (P|Q) \mid \nu a.P \mid \overline{a}\langle b\rangle.P \mid a(b).P,$$

where a and b range over **channel names**, and b is bound in $a(b).P$ and in $\nu b.P$. Processes are identified when related by the smallest context-closed equivalence relation $\equiv$ satisfying

$$0|P \equiv P \qquad P|Q \equiv Q|P \qquad P|(Q|R) \equiv (P|Q)|R \qquad (\nu a.P)|Q \equiv \nu a.(P|Q),$$

where in the last equation a should not occur free in Q. Transition is then given by the rules

$$\frac{}{\overline{a}\langle b\rangle.P|a(c).Q \longrightarrow P|(Q[c\mapsto b])} \qquad \frac{P \longrightarrow Q}{P|R \longrightarrow Q|R} \qquad \frac{P \longrightarrow Q}{\nu a.P \longrightarrow \nu a.Q}\ .$$

The π-calculus gives rise to a transition monad as follows. Again, we consider two placetaker types, one for channels and one for processes. Hence, $\mathbb{P} = 2 = \{\mathbf{c},\mathbf{p}\}$. Then, the syntax may be viewed as a monad $T : \mathbf{Set}^2 \to \mathbf{Set}^2$: given a variable object $X = (X_{\mathbf{c}}, X_{\mathbf{p}}) \in \mathbf{Set}^2$, the placetaker object $T(X) = (X_{\mathbf{c}}, T(X)_{\mathbf{p}}) \in \mathbf{Set}^2$ consists of the sets of channels and processes with free variables in X (modulo $\equiv$). Note that $T(X)_{\mathbf{c}} = X_{\mathbf{c}}$ as there is no operation on channels. Transitions relate processes, so we take $\mathbb{S} = 1$ and $S_1(X) = S_2(X) = X_{\mathbf{p}}$. Transitions are stable under substitution, hence form a transition monad.

2.2.3 Positive GSOS rules

An example involving labelled transitions (and $S_1 \neq S_2$) is given by Positive GSOS rules [7]. They specify labelled transitions $e \xrightarrow{a} f$. For any set O of operations with arities in $\mathbb{N}$, Positive GSOS rules have the shape $\dfrac{x_i \xrightarrow{a_{i,j}} y_{i,j}}{op(x_1, \ldots, x_n) \xrightarrow{c} e}$, where the variables x_i and $y_{i,j}$ are all distinct, $op \in O$ is an operation with arity n, and e is an expression potentially depending on all the variables.

Each family of operations and rules yields a transition monad with $\mathbb{P} = 1$, because we are in an untyped setting, and $\mathbb{S} = 1$ because states are terms. The syntax gives the term monad T. For transitions, in order to take labels into account, we take $S_1(X) = X$ and $S_2(X) = \mathbb{A} \times X$, where $\mathbb{A}$ denotes the set of labels. Transitions thus form a set over $X \times (\mathbb{A} \times X)$ as desired.

2.2.4 Differential λ-calculus

The differential λ-calculus [9] provides a further example with $S_1 \neq S_2$. Its syntax may [26, §6] be defined by

$$\begin{array}{rcll} e, f & ::= & x \mid \lambda x.e \mid e\ U \mid De \cdot f & \text{(terms)} \\ U, V & ::= & \langle e_1, \ldots, e_n \rangle & \text{(multiterms),} \end{array}$$

where $\langle e_1, \ldots, e_n \rangle$ denotes a (possibly empty) multiset, i.e., the ordering is irrelevant. Terms induce a monad T on **Set**, which we take as the placetaker monad (hence $\mathbb{P} = 1$).

Transitions relate terms to multiterms, hence $\mathbb{S} = 1$, S_1 is the identity, and $S_2 = !$ is the functor mapping any set X to the set of (finite) multisets over X.

The definition of transition is based on two intermediate notions:

1. **Unary multiterm substitution** $e[x \mapsto U]$ of a multiterm U for a variable x in a term e, which returns a multiterm (not to be confused with unary monadic substitution, which handles the particular case where U is just a singleton).
2. **Partial derivative** $\frac{\partial e}{\partial x} \cdot U$ of a term e w.r.t. a term variable x along a multiterm U. This again returns a multiterm.

Both are defined by induction on e (see [26]) and induce T-module morphisms $T^{(1)} \times\, ! \circ T \to\, ! \circ T$.

Unary multiterm substitution and partial derivation are used to define the transition relation as the smallest context-closed relation satisfying the rules below.

$$(\lambda x.e)\ U \to e[x \mapsto U] \qquad\qquad D(\lambda x.e) \cdot f \to \lambda x. \left(\frac{\partial e}{\partial x} \cdot f \right)$$

The second rule relies on the abbreviation $\lambda x.\langle e_1, \ldots, e_n \rangle := \langle \lambda x.e_1, \ldots, \lambda x.e_n \rangle$.

One can show that transitions are stable under substitution by terms, hence we again have a transition monad.

2.2.5 Call-by-value, simply-typed λ-calculus, big-step style

Let us finally organise the simply-typed, call-by-value, big-step λ-calculus into a transition monad. Most often, big-step semantics describes evaluation of closed terms. Our approach requires to treat open terms as well, so we consider a variant describing the evaluation of open terms [21]. In this setting, the main subtlety lies in the fact that variables are only placeholders for values.

Because variables and values are indexed by (simple) types, we take $\mathbb{P} = \mathbb{S}$ to be the set of types (generated from some fixed set of type constants). The monad T over $\mathbf{Set}^{\mathbb{P}}$ is then given by values: given a variable object $X \in \mathbf{Set}^{\mathbb{P}}$, the placetaker object $T(X) \in \mathbf{Set}^{\mathbb{P}}$ assigns to each type τ the set $T(X)_\tau$ of values of type τ taking free (typed) variables in X.

In big-step semantics, transition relates terms to values. Hence, we are seeking state functors $S_1, S_2 : \mathbf{Set}^{\mathbb{P}} \to \mathbf{Set}^{\mathbb{P}}$ such that $S_1(T(X))_\tau$ is the set of λ-terms of type τ with free variables in X, and $S_2(T(X))_\tau$ is the subset of values therein. For S_2, we should clearly take the identity functor. For S_1, we first observe that λ-terms can be described as application binary trees whose leaves are values (internal nodes being typed applications). Thus, we define $S_1(X)_\tau$ to be the set of application binary trees of type τ with leaves in X.

Finally, transitions are stable under value substitution, so we obtain a transition monad.

2.3 Categories of transition monads

In the next sections, we show how to generate transition monads such as the examples of the previous section from basic data. For this, we follow the recipe of initial semantics; this requires as input a category of "models" equipped with a "forgetful" functor to the category of transition monads, and it outputs the image of the initial model by this functor (of course, the existence of an initial model is also required). In order to do this for transition monads, we need to organise them into a category. We start with a particular case.

▶ **Definition 7.** *For any sets $\mathbb{P}$ and $\mathbb{S}$, finitary monad T over $\mathbf{Set}^{\mathbb{P}}$, and finitary functors $S_1, S_2 : \mathbf{Set}^{\mathbb{P}} \to \mathbf{Set}^{\mathbb{S}}$, let $\mathbf{TMnd}_{\mathbb{P},\mathbb{S}}(T, S_1, S_2)$ denote the slice category $T\text{-}\mathbf{Mod}_f(\mathbf{Set}^{\mathbb{S}})/S_1T \times S_2T$.*

This gives a first family of categories of transition monads, that we will integrate through a simple construction[2]:

▶ **Definition 8.** *A **record category** is a category of the form $\sum_{B \in \mathbf{ob}(\mathbf{B})} \mathbf{P}_B$ where B ranges over the objects of a **base category** $\mathbf{B}$, and each $\mathbf{P}_B$, called the **fibre over** B, is a category. In other words, it is given by a (base) category $\mathbf{B}$ equipped with a map $\mathbf{P} : \mathbf{ob}(\mathbf{B}) \to \mathbf{CAT}$.*

The relevant example for the present work is the following.

▶ **Definition 9.** *Given two sets $\mathbb{P}$ and $\mathbb{S}$, let $\mathbf{TMnd}_{\mathbb{P},\mathbb{S}}$ denote the following record category of transition monads with $\mathbb{P}$ and $\mathbb{S}$ as sets of types for placetakers and states:*

- *its base category is the product $\mathbf{Mnd}_f(\mathbf{Set}^{\mathbb{P}}) \times [\mathbf{Set}^{\mathbb{P}}, \mathbf{Set}^{\mathbb{S}}]_f^2$ of the category of monads on $\mathbf{Set}^{\mathbb{P}}$ with two copies of the functor category $[\mathbf{Set}^{\mathbb{P}}, \mathbf{Set}^{\mathbb{S}}]_f$;*
- *the fibre over a triple (T, S_1, S_2) is the category $\mathbf{TMnd}_{\mathbb{P},\mathbb{S}}(T, S_1, S_2)$ of Definition 7.*

3 Signatures and registers

The rest of the paper is devoted to the specification of transition monads via suitable signatures. More concretely, each of our example transition monads may be characterised as underlying the initial object in the category of models associated to a suitable signature.

We start in §3.1 by introducing a general notion of semantic signature over a category. In §3.2, we define registers: a register is just a family of semantic signatures. Our main goal (achieved in Definition 63) is to propose a register for transition monads.

[2] There is a more comprehensive construction, obtained by observing that the assignment $(T, S_1, S_2) \mapsto \mathbf{TMnd}_{\mathbb{P},\mathbb{S}}(T, S_1, S_2)$ forms a pseudofunctor and applying the so-called Grothendieck construction. Signatures for the earlier definitions of transition monads presented in [3] and [20, Chapter 6] fully acknowledge this fact, but here we choose to ignore it in order to make the development simpler.

3.1 Semantic signatures

Our notion of semantic signature is an abstract counterpart of usual signatures.

▶ **Definition 10.** *A **semantic signature** S over a given category $\mathbf{C}$ consists of*

- *a category S-alg **of models of** S **(or algebras)**, which admits an initial object, denoted by $S^{\circledast}$, and*
- *a **forgetful** functor $\mathbf{U}_s : S\text{-alg} \to \mathbf{C}$.*

▶ Remark 11. The term "forgetful functor" is merely the name of the corresponding component of a semantic signature; it does not impose any further constraint on it.

▶ **Terminology 12.** *Given a semantic signature S over a category $\mathbf{C}$, we say that S is a **signature for** $S^* := \mathbf{U}_S(S^{\circledast})$, or alternatively that S **specifies** S^*.*

▶ Notation 13. When convenient, we introduce a semantic signature over $\mathbf{C}$ as $u : \mathbf{E} \to \mathbf{C}$, to be understood as the semantic signature S with $S\text{-alg} := \mathbf{E}$ and $\mathbf{U}_S := u$.

▶ **Example 14.** Any object c of any given category $\mathbf{C}$ is specified by the following signatures:

- the functor $1 \to \mathbf{C}$ mapping the only object of the final category (with one object and one morphism) to c;
- the codomain functor $c/\mathbf{C} \to \mathbf{C}$ from the coslice category.

▶ **Example 15.** Consider the standard endofunctor $F : \mathbf{Set} \to \mathbf{Set}$ with $F(X) = X + 1$. We define a semantic signature over $\mathbf{Set}$ for which the category of models is the category of F-algebras, and the forgetful functor sends any F-algebra to its carrier. In order to complete the definition of this example, we should prove that the category of F-algebras has an initial object. This is well-known and the carrier of the initial model is $\mathbb{N}$.

▶ **Definition 16.** *We denote by $UR_{\mathbf{C}}$ the class of semantic signatures over the category $\mathbf{C}$ (UR stands for "universal register", as later justified by Definition 19, Section 3.2).*

▶ **Proposition 17.** *The assignment $\mathbf{C} \mapsto UR_{\mathbf{C}}$ extends to a functor $\mathbf{CAT} \to \mathbf{SET}$. The action of a functor $F : \mathbf{C} \to \mathbf{D}$, denoted by UR_F, is given by postcomposition.*

3.2 Registers of signatures

In this section, we introduce *registers* of signatures for a category $\mathbf{C}$, which are (possibly large) families of semantic signatures over $\mathbf{C}$. Roughly speaking, each register allows to write down specific signatures, gives the recipe for the corresponding semantic signature, hence yielding a notion of model together with the existence of an initial one.

▶ **Definition 18.** *A **register** R for a given category $\mathbf{C}$ consists of*

- *a class $\mathbf{Sig}_R$ (of **signatures**), and*
- *a **semantics** map $[\![-]\!]_R : \mathbf{Sig}_R \to UR_{\mathbf{C}}$.*

We can now motivate the notation UR_C above:

▶ **Definition 19.** *For a given category $\mathbf{C}$, the* universal register *$UR_{\mathbf{C}}$ is defined as follows:*

- *its signatures are semantic signatures for $\mathbf{C}$, and*
- *the map $[\![-]\!]_{UR_{\mathbf{C}}}$ is the identity (on $UR_{\mathbf{C}}$).*

▶ Notation 20. When convenient, we introduce a register as $u : S \to UR_{\mathbf{C}}$ to be understood as the register R with $\mathbf{Sig}_R := S$ and $[\![-]\!]_R := u$. Moreover, we sometimes implicitly identify a signature s in a register with its associated semantic signature $[\![s]\!]_R$.

We can now translate the slogan *Endofunctors are signatures* with a register, using a well-known initiality result [22, p62].

▶ **Definition 21.** *For a given cocomplete category* $\mathbf{C}$*, the* universal finitary endofunctorial register $UFE_{\mathbf{C}}$ *is defined as the map* $[\mathbf{C},\mathbf{C}]_f \to UR_{\mathbf{C}}$ *sending any finitary endofunctor* F *to the forgetful functor* F-alg $\to \mathbf{C}$ *from its category of algebras.*

Let us now define simple constructions of registers. Recalling Proposition 17, we have:

▶ **Definition 22.** *For any register* R *for* $\mathbf{C}$ *and functor* $F : \mathbf{C} \to \mathbf{D}$*, postcomposition with* UR_F *induces a register* $F_!(R) := (\mathbf{Sig}_R, UR_F \circ [\![-]\!]_R)$ *for* $\mathbf{D}$*.*

▶ **Definition 23.** *For any register* R *for* $\mathbf{C}$ *and map* $f : \mathbf{S} \to \mathbf{Sig}_R$*, precomposition with* f *induces a register* $f^*(R)$ *for* $\mathbf{C}$ *whose signatures are elements of* $\mathbf{S}$*. We say that* $f^*(R)$ *is a* ***subregister*** *of* R*.*

Here is an important application.

▶ **Definition 24.** *We call* ***endofunctorial*** *all registers of the form* $f^*(UFE_{\mathbf{C}})$*, for some map* $f : \mathbf{S} \to \mathbf{Sig}_{UFE_{\mathbf{C}}}$*.*

A useful fact is that endofunctorial registers are closed under the family construction:

▶ **Definition 25.** *For any endofunctorial register* R*, we denote by* R^* *the endofunctorial register whose signatures are families of signatures in* $\mathbf{Sig}_R$*, and whose semantics maps any family to the coproduct of associated endofunctors.*

4 Basic registers

In this section, we construct registers for monads and functors. Both of our initiality proofs follow from Fiore and Hur's theory of **equational systems** [11].

4.1 A register for monads

In this section, we fix a set $\mathbb{P}$ and construct a register $\mathbf{MndReg}(\mathbb{P})$ for monads on $\mathbf{Set}^{\mathbb{P}}$, generalising [2] to the simply-typed setting (see also Fiore and Hur [10]).

Let us first construct a naive register $\mathbf{MndReg}^0(\mathbb{P})$ which only allows us to specify operations. We will then deal with equations.

4.1.1 A naive register for specifying operations

We first describe signatures for $\mathbf{MndReg}^0(\mathbb{P})$. The basic idea for specifying operations is that the arity of an operation is a pair of ($\mathbf{Set}$-valued) **parametric** modules, in the sense of modules that are definable for any monad on $\mathbf{Set}^{\mathbb{P}}$.

▶ **Definition 26.** *Given a category* $\mathbf{D}$*, Let* $\mathbf{Mod}(\mathbf{D})$ *denote the category*

- *whose objects are pairs* (T, M) *of a finitary monad* T *on* $\mathbf{Set}^{\mathbb{P}}$ *and a finitary* T*-module* $M : \mathbf{Set}^{\mathbb{P}} \to \mathbf{D}$*,*
- *and whose morphisms* $(T, M) \to (U, N)$ *are pairs* (α, β) *of a monad morphism* $\alpha : T \to U$ *and a natural transformation* $\beta : M \to N$ *commuting with action.*

The first projection yields a forgetful functor $\mathbf{p} : \mathbf{Mod}(\mathbf{D}) \to \mathbf{Mnd}_f$*.*

▶ **Definition 27.** *A (*$\mathbf{D}$*-valued)* ***parametric*** *module is a section of* $\mathbf{p}$*, i.e., a functor* $s : \mathbf{Mnd}_f \to \mathbf{Mod}(\mathbf{D})$ *such that* $\mathbf{p} \circ s = id_{\mathbf{Mnd}_f}$*.*

▶ **Terminology 28.** *In the following, parametric modules are implicitly* **Set***-valued by default.*

▶ **Example 29.** Let us start by a few basic constructions of parametric modules:

- we denote by Θ the $\mathbf{Set}^{\mathbb{P}}$-valued parametric module mapping a monad T to itself, as a module over itself;
- for any $p_1, \ldots, p_n \in \mathbb{P}$ and $\mathbf{D}$-valued parametric module M, let $M^{(p_1,\ldots,p_n)}$ associate to each monad T the T-module $M(T)^{(p_1,\ldots,p_n)}$ as in the notations of §1, i.e., $M^{(p_1,\ldots,p_n)}(T)(X) = M(T)(X + \mathbf{y}_{p_1} + \cdots + \mathbf{y}_{p_n})$; when $\mathbb{P} = 1$, we merely count the p_i's and write $M^{(n)}$;
- for any finitary functor $F : \mathbf{D} \to \mathbf{E}$ and $\mathbf{D}$-valued parametric module M, the $\mathbf{E}$-parametric module $F \circ M$ maps any monad T to the T-module $F \circ M(T)$; as particular cases:
  - the terminal $\mathbf{Set}$-valued parametric module $1 = 1 \circ \Theta$ maps any monad T to the constant T-module 1;
  - for any $p \in \mathbb{P}$ and $\mathbf{Set}^{\mathbb{P}}$-valued parametric module M, we denote by M_p the $\mathbf{Set}$-valued parametric module mapping any monad T to the T-module $X \mapsto M(X)_p$ (see § 2.2.1);
  - given a finite family $(M_i)_{i \in I}$ of $\mathbf{Set}$-valued parametric modules, I, let $\prod_i M_i$ associate to any monad T the T-module $\prod_i M_i(T)$.

▶ **Example 30.** An operation will be specified by two parametric modules, one for the source and one for the target. Let us give the parametric modules for a few operations from our examples.

Language	Operation	Source	Target
Pure $\overline{\lambda}\mu$	Push	$\Theta_{\mathbf{p}} \times \Theta_{\mathbf{s}}$	$\Theta_{\mathbf{s}}$
Pure $\overline{\lambda}\mu$	Abstraction	$\Theta_{\mathbf{p}}^{(1)}$	$\Theta_{\mathbf{p}}$
π-calculus	Input $a(b).P$	$\Theta_{\mathbf{c}} \times \Theta_{\mathbf{p}}^{(\mathbf{c})}$	$\Theta_{\mathbf{p}}$

Morally, a signature (without equations) should be a family of pairs of parametric modules. However, in order to ensure existence of an initial model, we restrict this as follows.

▶ **Definition 31.** *A signature of* $\mathbf{MndReg}^0(\mathbb{P})$ *is a family of pairs* (d, c) *of parametric modules, in which*

- c *has the shape* Θ_p *for some* $p \in \mathbb{P}$*, and*
- d *is* ***elementary****, in the sense that it is a finite product of parametric modules of the shape* $(F \circ \Theta)^{(p_1,\ldots,p_n)}$ *for some* $p_1, \ldots, p_n \in \mathbb{P}$ *and finitary functor* $F : \mathbf{Set}^{\mathbb{P}} \to \mathbf{Set}$.

▶ **Example 32.** Typically, an elementary parametric module is a finite product of parametric modules of the shape $\Theta_p^{(p_1,\ldots,p_n)}$, for some $p, p_1, \ldots, p_n \in \mathbb{P}$.

▶ **Definition 33.** *The category of models associated to a signature* $(d_i, c_i)_{i \in I}$ *is defined by:*

- *A model is a monad* T *equipped with module morphisms* $d_i(T) \to c_i(T)$ *for all* $i \in I$.
- *A model morphism is a monad morphism commuting with these morphisms.*

▶ **Lemma 34.** *Any such category of models admits an initial object.*

With the obvious forgetful functor to the category of monads, this defines the semantic signature associated to a signature of $\mathbf{MndReg}^0(\mathbb{P})$, as a register for monads on $\mathbf{Set}^{\mathbb{P}}$.

4.1.2 A register for specifying operations and equations

Let us now define our register $\mathbf{MndReg}(\mathbb{P})$, following Ahrens et al.'s approach to specifying equations [2]. A signature of $\mathbf{MndReg}(\mathbb{P})$ will consist of a signature of $\mathbf{MndReg}^0(\mathbb{P})$, plus a family of "equations". An equation is essentially a pair of "metaterms", which may have "metavariables". The idea is that metavariables are given by a parametric module.

▶ **Example 35.** Consider associativity of parallel composition in π-calculus, $P|(Q|R) \equiv (P|Q)|R$: the metavariables are P, Q, and R. The corresponding parametric module is $\Theta_{\mathbf{p}}^3$.

Intuitively, a metaterm will be a parametric module morphism from metavariables to some Θ_p. However, it should potentially rely on constructions from the considered signature Σ of $\mathbf{MndReg}^0(\mathbb{P})$, as in Example 35. We thus consider a modified notion of parametric module morphism, which is parametric in models of Σ instead of mere monads.

▶ **Definition 36.** *Given any signature Σ for $\mathbf{MndReg}^0(\mathbb{P})$,*

- *a* ***Σ-module morphism*** *$M \to N$ between parametric modules M and N is a natural family of morphisms $(\alpha_T : M(T) \longrightarrow N(T))_{T \in \Sigma\text{-alg}}$, such that α_T is a T-module morphism, for each Σ-model T;*
- *a* ***Σ-equation*** *consists of an elementary parametric module V, called the* ***metavariable module****, and two parallel Σ-module morphisms $V \longrightarrow \theta_p$, for some $p \in \mathbb{P}$, called the* ***metaterms (of type** p**)****.*

▶ **Definition 37.** *A signature of $\mathbf{MndReg}(\mathbb{P})$ is a pair of a signature Σ of $\mathbf{MndReg}^0(\mathbb{P})$ and a family of Σ-equations.*

▶ **Definition 38.** *The category of models associated to a signature (Σ, E) is defined as follows.*

- *A model is a model T of Σ such that for all equations $(L, R) \in E$, $L(T) = R(T)$.*
- *A morphism of models of (Σ, E) is a morphism of models of Σ.*

The following generalises [2, Theorem 32]:

▶ **Lemma 39.** *Any such category of models admits an initial object.*

With the obvious forgetful functor to the category of monads, this defines the semantic signature associated to a signature of $\mathbf{MndReg}(\mathbb{P})$, as a register for monads on $\mathbf{Set}^{\mathbb{P}}$.

▶ **Example 40.** Let us revisit Example 35: the relevant signature Σ has in particular an operation $par : \Theta_{\mathbf{p}}^2 \to \Theta_{\mathbf{p}}$ for parallel composition, which gives our two metaterms

$$\Theta_{\mathbf{p}}^3 \xrightarrow{par\times\Theta_{\mathbf{p}}} \Theta_{\mathbf{p}}^2 \xrightarrow{par} \Theta_{\mathbf{p}} \qquad \text{and} \qquad \Theta_{\mathbf{p}}^3 \xrightarrow{\Theta_{\mathbf{p}}\times par} \Theta_{\mathbf{p}}^2 \xrightarrow{par} \Theta_{\mathbf{p}}.$$

▶ Notation 41 (Format for equations). We have already started to write pairs (d, c) of parametric modules as $d \to c$. Given any signature Σ for $\mathbf{MndReg}^0(\mathbb{P})$, we write any Σ-equation $\begin{array}{rcl} V & \to & \Theta_p^2 \\ x & \mapsto & (L, R) \end{array}$ as $x : V \vdash L \equiv R : \Theta_p$, or just $L \equiv R$ when the rest may be inferred.

▶ **Example 42.** We write associativity from Example 40 as just $par(P, par(Q, R)) \equiv par(par(P, Q), R)$. In this case, the argument x is the triple (P, Q, R).

4.2 A register for state functors

In this section, we sketch a register $\mathbf{FunReg}(\mathbb{P}, \mathbb{S})$, which is an adaptation of $\mathbf{MndReg}(\mathbb{P})$ to the case of state functors. Parametric modules are replaced with parametric premodules:

▶ **Definition 43.** *A* ***parametric premodule*** *is a functor $[\mathbf{Set}^{\mathbb{P}}, \mathbf{Set}^{\mathbb{S}}]_f \to [\mathbf{Set}^{\mathbb{P}}, \mathbf{Set}]_f$.*

We introduce the following notations:

▶ Notation 44. We denote by Θ the identity endofunctor on $[\mathbf{Set}^{\mathbb{P}}, \mathbf{Set}^{\mathbb{S}}]_f$, and by $\Gamma : [\mathbf{Set}^{\mathbb{P}}, \mathbf{Set}^{\mathbb{S}}]_f \to [\mathbf{Set}^{\mathbb{P}}, \mathbf{Set}^{\mathbb{P}}]_f$ the constant functor mapping anything to the identity endofunctor.

The constructions $\prod_i M_i$ and $M^{(p_1,\ldots,p_n)}$ carry over essentially verbatim.

▶ **Example 45.** We have seen in §2.2.5 that the source state functor S_1 for call-by-value, simply-typed λ-calculus is built with application binary trees. Intuitively, it has two (type-indexed families of) operations: the first one injects values, thus maps X_t, into $S_1(X)_t$, and the second one forms application binary trees, with components $X_t \to S_1(X)_t$ and $S_1(X)_{t\to t'} \times S_1(X)_t \to S_1(X)_{t'}$.
This yields a specification with two families of operations $\Gamma_t \to \Theta_t$ and $\Theta_{t\to t'} \times \Theta_t \to \Theta_{t'}$.

Operations, equations, and models are defined exactly as for monads, and a signature in $\mathbf{FunReg}(\mathbb{P}, \mathbb{S})$ again consists of families of operations and equations[3]. The only difference lies in the notion of elementary parametric premodule, which becomes the following:

▶ **Definition 46.** *A parametric premodule is **elementary** iff it is a finite product of parametric premodules of the shape $(F \circ \langle\Gamma, \Theta\rangle)^{(p_1,\ldots,p_n)}$ for some $p_1, \ldots, p_n \in \mathbb{P}$ and finitary functor $F : \mathbf{Set}^{\mathbb{P}} \times \mathbf{Set}^{\mathbb{S}} \to \mathbf{Set}$.*

▶ **Example 47.** Typically, an elementary parametric premodule is a product of parametric premodules of the shape $\Gamma_p^{(p_1,\ldots,p_n)}$ or $\Theta_\sigma^{(p_1,\ldots,p_n)}$, for some $p, p_1, \ldots, p_n \in \mathbb{P}$ and $\sigma \in \mathbb{S}$.

▶ Remark 48. Any finitary functor F admits a trivial signature consisting of the family $((F_\sigma \circ \Gamma) \to \Theta_\sigma)_{\sigma\in\mathbb{S}}$ of operations. Here are a few examples from §2.2:

Language	State functor	Specification
$\bar{\lambda}\mu$	$S_1(X) = S_2(X) = X_{\mathbf{p}} \times X_{\mathbf{s}}$	$\langle -\vert-\rangle : \Gamma_{\mathbf{p}} \times \Gamma_{\mathbf{s}} \to \Theta$
π	$S_1(X) = S_2(X) = X_{\mathbf{p}}$	$\Gamma_{\mathbf{p}} \to \Theta$
Call-by-value, simply-typed λ	$S_2(X) = X$	$\eta_t : \Gamma_t \to \Theta_t$ (for all t)
Positive GSOS specifications	$S_1(X) = X$ $S_2(X) = \mathbb{A} \times X$	$\Gamma \to \Theta$ $\mathbb{A} \times \Gamma \to \Theta$

▶ Notation 49. We adopt Notation 41 for state functors. E.g., Example 42 applies verbatim for associativity of multiset union in the target state functor for differential λ-calculus.

5 Constructions of registers

In this section, we provide constructions of new registers out of existing ones.

5.1 Product registers

Let us start by considering products. We first describe the product of semantic signatures, and then, based on that, we define product registers.

Given semantic signatures for some family of categories, we want to construct a semantic signature for the product category. The application we have in mind is the product category $\mathbf{Mnd}_f(\mathbf{Set}^{\mathbb{P}}) \times [\mathbf{Set}^{\mathbb{P}}, \mathbf{Set}^{\mathbb{S}}]_f^2$ (Definition 9), which is the base category of our record category of transition monads (see below Example 53).

▶ **Lemma 50.** *Given a set I and functors $U_i : \mathbf{E}_i \to \mathbf{C}_i$ for $i \in I$, if each $\mathbf{E}_i$ has an initial object, then so does the product $\prod_i \mathbf{E}_i$.*

[3] Existence of an initial object in the category of models relies on the theory of equational systems, as mentioned above.

▶ **Definition 51.** *Given a family $\mathbf{C} := (\mathbf{C}_i)_{i \in I}$ of categories, and a corresponding family of semantic signatures $u_i : \mathbf{E}_i \to \mathbf{C}_i$, the product $\prod_i u_i : \prod_i \mathbf{E}_i \to \prod_i \mathbf{C}_i$ is a semantic signature. This defines our (external) product of signatures $\prod_{\mathbf{C}} : \prod_i UR_{\mathbf{C}_i} \to UR_{\prod_i \mathbf{C}_i}$.*

Let us now define the product of a family of registers.

▶ **Definition 52.** *The **product** of a family $(u_i : S_i \to UR_{\mathbf{C}_i})_{i \in I}$ of registers is obtained by post-composing $\prod_i u_i$ with the product of semantic signatures:*

$$\prod_i S_i \xrightarrow{\prod_i u_i} \prod_i UR_{\mathbf{C}_i} \xrightarrow{\prod_{\mathbf{C}}} UR_{\prod_i \mathbf{C}_i}.$$

▶ **Example 53.** The product $\mathbf{MndReg}(\mathbb{P}) \times \mathbf{FunReg}(\mathbb{P}, \mathbb{S})^2$ of the register $\mathbf{MndReg}(\mathbb{P})$ with two copies of the register $\mathbf{FunReg}(\mathbb{P}, \mathbb{S})$.

5.2 Registers for slice module categories

In this section, we fix two sets $\mathbb{P}$ and $\mathbb{S}$, a monad T on $\mathbf{Set}^{\mathbb{P}}$, and a $\mathbf{Set}^{\mathbb{S}}$-valued T-module M. We then define an endofunctorial register $\mathbf{Rule}(T, M)$ for the category $T\text{-}\mathbf{Mod}_f(\mathbf{Set}^{\mathbb{S}})/M$. Later on, we will use the register $\mathbf{Rule}^*(T, M)$ (recalling Definition 25) with $M := S_1T \times S_2T$, i.e., for the category of transition monads over (T, S_1, S_2).

5.2.1 The naive register $\mathbf{Rule}^0$

For expository purposes, we start by defining a naive endofunctorial register, $\mathbf{Rule}^0(T, M)$. A signature of $\mathbf{Rule}^0(T, M)$ consists of

- a **metavariable Set**-valued T-module V,
- a **conclusion** module morphism $t : V \to M_\tau$ for some **conclusion** state type $\tau \in \mathbb{S}$, and
- a list of **premise** module morphisms $s_i : V \to M_{\sigma_i}$, for some **premise** state types $\sigma_i \in \mathbb{S}$.

▶ **Example 54.** For the left application congruence rule of pure λ-calculus $\dfrac{e \to e'}{e\ f \to e'\ f}$, there are three metavariables e, e', and f, so the metavariable module V is T^3. The conclusion and premise are respectively defined as the module morphisms

$$\begin{array}{rcl} T^3 & \to & T^2 \\ (e, e', f) & \mapsto & (e\ f, e'\ f) \end{array} \quad \text{and} \quad \begin{array}{rcl} T^3 & \to & T^2 \\ (e, e', f) & \mapsto & (e, e'). \end{array}$$

Now, the endofunctor Σ_S associated to any signature $S := (\tau, V, t, (\sigma_i, s_i)_{i \in n})$ is a composite

$$\begin{tikzcd} T\text{-}\mathbf{Mod}_f(\mathbf{Set})/\prod_i M_{\sigma_i} \arrow[r, "\Delta_{\langle s_i \rangle_i}"] & T\text{-}\mathbf{Mod}_f(\mathbf{Set})/V \arrow[r, "\sum_t"] & T\text{-}\mathbf{Mod}_f(\mathbf{Set})/M_\tau \arrow[d, hook] \\ T\text{-}\mathbf{Mod}_f(\mathbf{Set}^{\mathbb{S}})/M \arrow[u, "\prod_i (-)_{\sigma_i}"] & & T\text{-}\mathbf{Mod}_f(\mathbf{Set}^{\mathbb{S}})/M, \end{tikzcd} \tag{1}$$

of four functors, where

- $\prod_i (\partial : R \to M)_{\sigma_i}$ denotes $\prod_i \partial_{\sigma_i} : \prod_i R_{\sigma_i} \to \prod_i M_{\sigma_i}$,
- $\Delta_{\langle s_i \rangle_i}$ is defined by pullback along the tupling $\langle s_i \rangle_i : V \to \prod_i M_{\sigma_i}$ of all premises,
- $\sum_i$ is defined by postcomposition with the conclusion $t : V \to M_\tau$, and

- the last functor is the canonical embedding, which maps any $R \to M_\tau$ to $R \cdot \mathbf{y}_\tau \to M$, where $R \cdot \mathbf{y}_\tau$ is defined for every X by $(R \cdot \mathbf{y}_\tau)(X)_\tau = R(X)$ and $(R \cdot \mathbf{y}_\tau)(X)_\sigma = \emptyset$ for $\sigma \neq \tau$.

▶ Remark 55. The embedding $(-) \cdot \mathbf{y}_\tau$ is left adjoint to evaluation at τ: $(-) \cdot \mathbf{y}_\tau \dashv (-)_\tau$. Thus Σ_S maps any $\partial : R \to M$ to the transpose of the right-hand composite q below.

$$\begin{tikzcd}
\prod_i R_{\sigma_i} \arrow[d, "\prod_i \partial_{\sigma_i}"'] & P \arrow[l] \arrow[d] \arrow[dl, phantom, very near start, "\lrcorner"] \arrow[dr, "q"] & \\
\prod_i M_{\sigma_i} & V \arrow[l, "\langle s_i \rangle_i"] \arrow[r, "t"'] & M_\tau
\end{tikzcd} \tag{2}$$

By Lemma 59 below, each Σ_S is finitary, which completes the definition of our register $\mathbf{Rule}^0(T, M)$ for $T\text{-}\mathbf{Mod}_f(\mathbf{Set}^{\mathbb{S}})/M$.

▶ **Example 56.** Consider the endofunctor associated to the left application rule of Example 54. Because $\mathbb{S} = 1$, the functor $(-) \cdot \mathbf{y}_\tau$ is the identity functor, so the endofunctor maps any $\partial : R \to T^2$:

- to the pullback P, where $P(X)$ is the set of 4-tuples $(r, e, e', f) \in R(X) \times T(X)^3$ such that r is a transition $e \to e'$,
- with projection to T^2 mapping any (r, e, e', f) to $(e\ f, e'\ f)$.

An algebra is thus such a $\partial : R \to T^2$ which, to each such tuple (r, e, e', f) associates a transition over $(e\ f, e'\ f)$, as desired.

5.2.2 The register Rule

In this section, we define the endofunctorial register $\mathbf{Rule}(T, M)$, refining the naive register $\mathbf{Rule}^0(T, M)$ of the previous section. The motivation lies in rules whose premises have additional free variables.

▶ **Example 57.** Consider the ξ rule of pure λ-calculus: $\dfrac{e \to f}{\lambda x.e \to \lambda x.f}$.

The metavariables and conclusion may remain the same; the problem is with the premise, which cannot be a morphism $V \to T^2$, but should rather have type $V \to T^{(1)} \times T^{(1)}$. We thus generalise $\mathbf{Rule}^0(T, M)$ to let rules have premises of this shape:

▶ **Definition 58.** *The endofunctorial register* $\mathbf{Rule}(T, M)$ *for* $T\text{-}\mathbf{Mod}_f(\mathbf{Set}^{\mathbb{S}})/M$ *is defined by:*

- *signatures are just as in* $\mathbf{Rule}^0(T, M)$*, except that the premises now have the shape* $s : V \to M_\sigma^{(\vec{p})}$*, for* $\sigma \in \mathbb{S}$ *and* $\vec{p}$ *a list of elements of* $\mathbb{P}$*; and*
- *the induced endofunctor is defined exactly as for naive rules, replacing* $\prod_i R_i$ *with* $\prod_i R_i^{(\vec{p^i})}$.

This register is well defined thanks to the following lemma (proved in Appendix H):

▶ **Lemma 59.** *Given a signature* S*, the endofunctor* Σ_S *is finitary.*

Using Definition 25, we obtain a register $\mathbf{Rule}^*(T, M)$ for $T\text{-}\mathbf{Mod}_f(\mathbf{Set}^{\mathbb{S}})/M$, whose signatures are families of signatures in $\mathbf{Rule}(T, M)$.

5.2.3 A format for signatures in Rule and Rule*

When $M = S_1T \times S_2T$, we adopt the following notational conventions for signatures in $\mathbf{Rule}(T, M)$:

- for each premise or conclusion $\begin{array}{ccc} V & \to & W \\ x & \mapsto & (L,R) \end{array}$ of a rule, we write $x : V \vdash L \rightsquigarrow R : W$,
- we organise the premises and conclusion as usual:

$$\frac{x : V \vdash L_1 \rightsquigarrow R_1 : W_1 \qquad \ldots \qquad x : V \vdash L_n \rightsquigarrow R_n : W_n}{x : V \vdash L \rightsquigarrow R : W},$$

or just $\dfrac{L_1 \rightsquigarrow R_1 \qquad \ldots \qquad L_n \rightsquigarrow R_n}{L \rightsquigarrow R}$ when the rest may be inferred from context.

▶ Remark 60. The module V is often a product and thus x is a tuple.

▶ Remark 61 ([3]). In practice, there are several choices for building the transition rule out of such a schematic presentation, depending on the order of metavariables. This order is irrelevant: all interpretations yield isomorphic semantics, in the obvious sense.

5.3 The record construction for registers

The registers introduced in the previous sections allow us to design registers for the various components of our transition monad, separately: we may specify the underlying monad T and state functors S_1 and S_2 using signatures from the registers for monads and functors previously defined. We may even assemble these signatures into a single signature Σ for the product register of Definition 52. Then, we may specify the desired transition monad as an object of the fibre $\mathbf{TMnd}_{\mathbb{P},\mathbb{S}}(T, S_1, S_2)$, using a signature $\mathbf{R}$ of the register $\mathbf{Rule}^*(T, M)$ from Section 5.2.2, with $M = S_1T \times S_2T$.

In this section, we show how to assemble Σ and $\mathbf{R}$ into a single signature of some compound register for the record category $\mathbf{TMnd}_{\mathbb{P},\mathbb{S}}$. Our construction can be performed in general for an arbitrary record category.

▶ **Definition 62.** *Consider any record category $K = \sum_{B \in \mathbf{ob}(\mathbf{B})} \mathbf{P}(B)$, with $\mathbf{P} : \mathbf{ob}(\mathbf{B}) \to \mathbf{CAT}$, together with*

- *a **base** register R_b for $\mathbf{B}$, and*
- *for each signature S in $\mathbf{Sig}_{R_b}$, a **fibre** register $R_f(S)$ for the fibre $\mathbf{P}_{S^*}$ over the initial S-algebra.*

*The **record register** $\sum(R_b, R_f)$ for the record category K is defined as follows.*

- *Signatures are pairs (S, F) with $S \in \mathbf{Sig}_{R_b}$ and $F \in \mathbf{Sig}_{R_f(B)}$.*
- *The semantic signature associated to any (S, F) is the composite $F\text{-alg} \xrightarrow{\mathbf{U}_F} \mathbf{P}_{S^*} \hookrightarrow K$.*

Our main example application is:

▶ **Definition 63.** *Let $\mathbf{TMndReg} := \sum(R_b, R_f)$, where*

- *R_b is the product register $\mathbf{MndReg}(\mathbb{P}) \times \mathbf{FunReg}(\mathbb{P}, \mathbb{S})^2$ of Example 53 for monads and state functors, and*
- *for all signatures S of R_b, the register $R_f(S)$ is defined as $\mathbf{Rule}^*(T, S_1T \times S_2T)$, where $(T, S_1, S_2) = S^*$.*

We provide signatures for all examples from §2.2 in the appendix.

6 Conclusion and perspectives

We have introduced transition monads as a generalisation of reduction monads, and demonstrated that they cover relevant new examples. We have introduced a register of signatures for specifying them. In future work, we plan to investigate more general forms of state modules. E.g., using an arbitrary module covers the subtle labelled transition system for π-calculus. We also plan to generalise the Grothendieck construction to signatures/registers along the line in [3]. In the longer term, we plan to refine our register in a way ensuring that the generated transition system satisfies important properties like congruence of observational equivalences, confluence, or type soundness. In this direction, a result on congruence of applicative bisimilarity for a simpler register has recently been obtained by Borthelle et al. [8]. Finally, quantitative (e.g., probabilistic [6]) operational semantics would be worth investigating in our setting.

References

1 Benedikt Ahrens. Modules over relative monads for syntax and semantics. *MSCS*, 26:3–37, 2016.

2 Benedikt Ahrens, André Hirschowitz, Ambroise Lafont, and Marco Maggesi. Modular specification of monads through higher-order presentations. In Herman Geuvers, editor, *Proc. 4th Int. Conf. on Formal Structures for Comp. and Deduction*, LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

3 Benedikt Ahrens, André Hirschowitz, Ambroise Lafont, and Marco Maggesi. Reduction monads and their signatures. *PACMPL*, 4(POPL):31:1–31:29, 2020. doi:10.1145/3371099.

4 Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Monads need not be endofunctors. *Logical Methods in Computer Science*, 11(1), 2015. doi:10.2168/LMCS-11(1:3)2015.

5 Thorsten Altenkirch, Peter Morris, Fredrik Nordvall Forsberg, and Anton Setzer. A categorical semantics for inductive-inductive definitions. In Andrea Corradini, Bartek Klin, and Corina Cîrstea, editors, *Proc. 4th Alg. and Coalgebra in Comp. Sci.*, volume 6859 of *LNCS*, pages 70–84. Springer, 2011. doi:10.1007/978-3-642-22944-2_6.

6 Falk Bartels. GSOS for probabilistic transition systems: (extended abstract). *Electronic Notes in Theoretical Computer Science*, 65(1):29–53, 2002. CMCS '2002, Coalgebraic Methods in Computer Science. doi:10.1016/S1571-0661(04)80358-X.

7 Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced. *J. ACM*, 42(1):232–268, 1995. doi:10.1145/200836.200876.

8 Peio Borthelle, Tom Hirschowitz, and Ambroise Lafont. A cellular Howe theorem. In *Proc. 35th ACM/IEEE Logic in Comp. Sci.* ACM, 2020. doi:10.1145/3373718.3394738.

9 Thomas Ehrhard and Laurent Regnier. The differential lambda-calculus. *TCS*, 309:1–41, 2003.

10 M. P. Fiore and C.-K. Hur. Second-order equational logic. In *Proceedings of the 19th EACSL Annual Conference on Computer Science Logic (CSL 2010)*, 2010.

11 Marcelo Fiore and Chung-Kil Hur. On the construction of free algebras for equational systems. *TCS*, 410:1704–1729, 2009.

12 Marcelo P. Fiore. Second-order and dependently-sorted abstract syntax. In *Proc. 23rd Logic in Comp. Sci.*, pages 57–68. IEEE, 2008. doi:10.1109/LICS.2008.38.

13 Marcelo P. Fiore and Sam Staton. A congruence rule format for name-passing process calculi from mathematical structural operational semantics. In *Proc. 21st Logic in Comp. Sci.*, pages 49–58. IEEE, 2006. doi:10.1109/LICS.2006.7.

14 Marcelo P. Fiore and Daniele Turi. Semantics of name and value passing. In *Proc. 16th Logic in Comp. Sci.*, pages 93–104. IEEE, 2001. doi:10.1109/LICS.2001.932486.

15 Richard Garner. Combinatorial structure of type dependency. *Journal of Pure and Applied Algebra*, 219(6):1885 – 1914, 2015. doi:10.1016/j.jpaa.2014.07.015.

16 Makoto Hamana. Term rewriting with variable binding: An initial algebra approach. In *Proc. 5th Princ. and Practice of Decl. Prog.* ACM, 2003. doi:10.1145/888251.888266.

17 Hugo Herbelin. *Séquents qu'on calcule: de l'interprétation du calcul des séquents comme calcul de lambda-termes et comme calcul de stratégies gagnantes.* PhD thesis, Paris Diderot University, France, 1995. URL: https://tel.archives-ouvertes.fr/tel-00382528.

18 André Hirschowitz and Marco Maggesi. Modules over monads and linearity. In *WoLLIC*, volume 4576 of *LNCS*, pages 218–237. Springer, 2007. doi:10.1007/3-540-44802-0_3.

19 Tom Hirschowitz. Cartesian closed 2-categories and permutation equivalence in higher-order rewriting. *LMCS*, 9(3), 2013. doi:10.2168/LMCS-9(3:10)2013.

20 Ambroise Lafont. *Signatures and models for syntax and operational semantics in the presence of variable binding.* PhD thesis, École Nationale Superieure Mines – Telecom Atlantique Bretagne Pays de la Loire – IMT Atlantique, 2019. URL: https://arxiv.org/abs/1910.09162v2.

21 Luca Paolini and Simona Ronchi Della Rocca. Call-by-value solvability. *RAIRO - Theor. Inf. and Applic.*, 33(6):507–534, 1999. doi:10.1051/ita:1999130.

22 Jan Reiterman. A left adjoint construction related to free triples. *JPAA*, 10:57–71, 1977.

23 Davide Sangiorgi and David Walker. *The π-calculus - a theory of mobile processes.* Cambridge University Press, 2001.

24 Sam Staton. General structural operational semantics through categorical logic. In *Proc. 23rd Logic in Comp. Sci.*, pages 166–177. IEEE, 2008. doi:10.1109/LICS.2008.43.

25 Daniele Turi and Gordon D. Plotkin. Towards a mathematical operational semantics. In *Proc. 12th Logic in Comp. Sci.*, pages 280–291, 1997. doi:10.1109/LICS.1997.614955.

26 Lionel Vaux. *λ-calcul différentiel et logique classique : interactions calculatoires.* PhD thesis, Université Aix-Marseille 2, 2007.

A Specifying the call-by-value, simply-typed, big-step λ-calculus

In the setting of §2.2.5, let $F : \mathbf{Set}^{\mathbb{P}} \to \mathbf{Set}^{\mathbb{S}}$ be specified by two families of operations $app_{t,t'} : \Theta_{t\to t'} \times \Theta_t \to \Theta_{t'}$ and $val_t : \Gamma_t \to \Theta_t$. Our signature for call-by-value, simply-typed, big-step λ-calculus is presented in the following table

Monad and state functors	T	S_1	S_2
	$\lambda_{t,t'} : (F_{t'} \circ \Theta)^{(t)} \to \Theta_{t\to t'}$	F	Id
Rules	$\dfrac{}{val_t(v) \rightsquigarrow v}$	$\dfrac{e_1 \rightsquigarrow \lambda_{t,t'}(e_3) \quad e_2 \rightsquigarrow w \quad e_3[w] \rightsquigarrow v}{app_{t,t'}(e_1, e_2) \rightsquigarrow v}$	

where

- $-[-] : (S_1T)^{(t)}_{t'} \times T_t \to (S_1T)_{t'}$ denotes the substitution morphism;
- $S_1 = F$ and $S_2 = \mathrm{Id}$ are specified by easy signatures as in Remark 48;
- the rules should be understood as families of rules indexed by suitable types.

In a bit more detail, the first rule is indexed by the type t of v. The second one is indexed by two types t and t'. There are five metavariables, e_1, e_2, e_3, v, and w. We thus take $V := (S_1T)_{t\to t'} \times (S_1T)_t \times (S_1T)^{(t)}_{t'} \times T_{t'} \times T_t$.

B Specifying the $\overline{\lambda}\mu$-calculus

For $\overline{\lambda}\mu$-calculus, the state functor has been specified in Remark 48. The monad is specified by operations

$$\mu : \Theta^{(\mathbf{s})}_{\mathbf{p}} \times \Theta^{(\mathbf{s})}_{\mathbf{s}} \to \Theta_{\mathbf{p}} \qquad \lambda : \Theta^{(\mathbf{p})}_{\mathbf{p}} \to \Theta_{\mathbf{p}} \qquad \cdot : \Theta_{\mathbf{p}} \times \Theta_{\mathbf{s}} \to \Theta_{\mathbf{s}},$$

with no equation.

The transition rules are almost as usual:

$$\frac{}{\langle \mu\langle e|\pi'\rangle|\pi\rangle \to \langle e[\pi],\pi'[\pi]\rangle} \qquad \frac{}{\langle \lambda(e)|e'\cdot\pi\rangle \to \langle e[e']|\pi\rangle}$$

The first rule has metavariable module $V := T_{\mathbf{p}}^{(\mathbf{s})} \times T_{\mathbf{s}}^{(\mathbf{s})} \times T_{\mathbf{s}}$, the argument being (e, π', π). The second rule has $V := T_{\mathbf{p}}^{(\mathbf{p})} \times T_{\mathbf{p}} \times T_{\mathbf{s}}$.

C Specifying the π-calculus

For π-calculus, the state functor has been specified in Remark 48. The placetaker monad T is specified by operations $0 : 1 \to \Theta_{\mathbf{p}} \quad | : \Theta_{\mathbf{p}} \times \Theta_{\mathbf{p}} \to \Theta_{\mathbf{p}} \quad \nu : \Theta_{\mathbf{p}}^{(\mathbf{c})} \to \Theta_{\mathbf{p}} \quad \mathit{out} : \Theta_{\mathbf{c}}^2 \times \Theta_{\mathbf{p}} \to \Theta_{\mathbf{p}} \quad \mathit{in} : \Theta_{\mathbf{c}} \times \Theta_{\mathbf{p}}^{(\mathbf{c})} \to \Theta_{\mathbf{p}}$
with equations $\quad 0|P \equiv P \quad P|Q \equiv Q|P \quad P|(Q|R) \equiv (P|Q)|R \quad \nu(P)|Q \equiv \nu(P|\mathbf{w_c}(Q))$, almost copied verbatim from §2.2.2, where $\mathbf{w_c}(Q)$ denotes the action of $T(X) \to T(X + \mathbf{y_c})$ on Q. Finally, the transition rules are

$$\frac{}{\mathit{out}(a,b,P)|\mathit{in}(a,Q) \longrightarrow P|(Q[b])} \qquad \frac{P \longrightarrow Q}{P|R \longrightarrow Q|R} \qquad \frac{P \longrightarrow Q}{\nu(P) \longrightarrow \nu(Q)}\ .$$

In particular, the third rule has as metavariable module $V := (\Theta_{\mathbf{p}}^{(\mathbf{c})})^2$.

D The register $GSOS^+$

In this section, we define a register $GSOS^+$ for specifying positive GSOS systems [7]. This is a subregister of our record register **TMndReg**, for untyped ($\mathbb{P} = \mathbb{S} = 1$) transition monads. Let us recall that signatures in this register consist of pairs (B, F) where B is a signature in the product register of Example 53, and F is a signature in $\mathbf{Rule}^*(B)$.

In order to describe this subregister, we have to describe its class of signatures, and then assign to each such signature a pair (B, F) as above. Before performing this task we recall the standard format of a $GSOS^+$ rule:

$$\frac{\ldots \qquad V_i \xrightarrow{a_{i,j}} V_{i,j} \qquad \ldots}{op(V_1,\ldots,V_n) \xrightarrow{c} e}.$$

A signature of the register $GSOS^+$ consist of

- three sets O (for operations), $\mathbb{A}$ (for labels), and R (for rules),
- for each element o of O, a number m_o (the arity),
- for each rule,
  - an operation $o \in O$ (for the source of the conclusion),
  - a label $c \in \mathbb{A}$ (the label of the conclusion),
  - for each $i \leq m_o$,
    - a number n_i (the number of premises for this argument),
    - for each $j \leq n_i$, an element a_{ij} of $\mathbb{A}$ (for the label of the premise),
    - a term e in the syntax generated by O, potentially depending on $m_o + \sum_i n_i$ variables.

We now describe the pair (B, F) associated to a signature as above:

- the signatures for both state functors have been given in Remark 48;

- the signature for the underlying monad is $\sum_{o \in O} \Theta^{m_o} \to \Theta$ (following §4.1).

These three signatures yield our base signature B. Finally, each Positive GSOS rule yields a rule $\dfrac{\dots \qquad V_i \rightsquigarrow (a_{i,j}, V_{i,j}) \qquad \dots}{op_o(V_1, \dots, V_{m_o}) \rightsquigarrow (c, e)}$ in our fibre signature F (in the register $\mathbf{Rule}^*(T, S_1 T \times S_2 T)$).

E Specifying the differential λ-calculus

In this section, we present in some detail the signature for differential λ-calculus, as a transition monad with $\mathbb{P} = \mathbb{S} = 1$, introduced in §2.2.4. A signature in the register of transition monads consists of two components: a (product) signature for the state functors and monad, given in §E.1, and a signature for the β and ∂-transition rules. Both are straightforwardly modelled by a signature over as explained in §5.2, but they first require us to construct some intermediate operations $-[x \mapsto -]$ and $\frac{\partial -}{\partial x} \cdot -$. We tackle this task in §E.2.

E.1 State functors and monad of differential λ-calculus

The first state functor is the identity functor $\mathrm{Id} : \mathbf{Set} \to \mathbf{Set}$, and thus is specified by the arity $\Gamma \to \Theta$. The second state functor is $!$, the multiset functor, and is specified by three arities $1 \to \Theta$ (for the empty multiset), $\Gamma \to \Theta$ (for the singleton multiset), and $\Theta \times \Theta \to \Theta$ (for the union operation), subject to commutativity, associativity, and unitality.

Next, the monad of differential λ-calculus is specified by the arities $\Theta^{(1)} \to \Theta$, $\Theta \times !\Theta \to \Theta$, and $\Theta \times \Theta \to \Theta$, modelling the operations $\lambda x.-$, $-\ -$, and $D - \cdot -$. No equation is required.

E.2 Intermediate constructions for differential λ-calculus

Specifying the transition rules requires two intermediate constructions: **unary multiterm substitution** $-[x \mapsto -]$, and **partial derivation** $\frac{\partial -}{\partial x} \cdot -$, which we both model as T-module morphisms $T^{(1)} \times !T \to !T$, or equivalently $T^{(1)} \to (!T)^{!T}$. [4]

In [26, §6], the underlying maps are defined by induction. Here, we define them by using a special induction principle for building module morphisms out of $T^{(1)}$, that we now describe. Let us denote by $\Sigma^{\uparrow}$ the endofunctor on T-modules defined by the same formula than the parametric module specifying the differential λ-calculus monad T (see Section E.1): $\Sigma^{\uparrow}(M) = M^{(1)} + M \times !M + M \times M$. Note that T has a canonical $\Sigma^{\uparrow}$-algebra structure, $T^{(1)}$ a canonical $(\Sigma^{\uparrow} + 1)$-algebra structure, and the embedding $T \to T^{(1)}$ is a $\Sigma^{\uparrow}$-algebra morphism.

The following lemma induces a useful induction principle:

▶ **Lemma 64.** *Given any $(\Sigma^{\uparrow} + 1)$-algebra M and $\Sigma^{\uparrow}$-algebra morphism $m : T \to M$, there exists a unique $(\Sigma^{\uparrow} + 1)$-algebra morphism $i : T^{(1)} \to M$ making the following diagram commute.*

$$\begin{tikzcd}
T \arrow[r, "j"] \arrow[dr, "\forall m"'] & T^{(1)} \arrow[d, dashed, "\exists ! i"] \\
 & M
\end{tikzcd}$$

[4] The category of finitary $\mathbf{Set}$-valued T-modules is equivalent to the category of presheaves on the full subcategory of the Kleisli category of T consisting of finite sets. As such, it has exponentials.

Proof. By [11], T is the initial algebra of $\Sigma^{\uparrow} + \mathrm{Id}$, as a (finitary) endofunctor on $[\mathbf{Set}, \mathbf{Set}]_f$. By inspecting the colimit of the relevant initial chains, it can be shown that $T^{(1)}$ is the initial algebra of $\Sigma^{\uparrow} + \mathrm{Id} + 1$. Given the data, M has $(\Sigma^{\uparrow} + \mathrm{Id} + 1)$-algebra structure: the natural transformation $\mathrm{Id} \to M$ is merely the composite $\mathrm{Id} \to T \to M$ with the unit of T.

By initiality, we have a unique $(\Sigma^{\uparrow} + \mathrm{Id} + 1)$-algebra morphism $i : T^{(1)} \to M$ as functors, such that $i \circ j = m$. At this stage, we can already deduce uniqueness of the desired morphism.

It remains to show that i is a T-module morphism. We omit the proof for lack of space. ◀

The lemma legitimates the following general recipe for constructing a T-module morphism $T^{(1)} \to M$:

1. provide M with a $(\Sigma^{\uparrow} + 1)$-algebra structure,
2. provide a T-module morphism $T \to M$, and
3. check that this morphism is a $\Sigma^{\uparrow}$-algebra morphism.

Indeed, by the above adjunction, we get a $(\Sigma^{\uparrow} + 1)$-algebra morphism $T^{(1)} \to M$, and thus in particular a T-module morphism $T^{(1)} \to M$.

We now apply this recipe to define unary multiterm substitution (the case of partial derivation is similar).

1. We first equip $(!T)^{!T}$ with $(\Sigma^{\uparrow} + 1)$-algebra structure. This structure should reflect the recursive equations in [26, Definition 6.3] defining unary multiterm substitution. In fact, the recursive equations will follow from the fact that the constructed morphism $T^{(1)} \to (!T)^{!T}$ is a $(\Sigma^{\uparrow} + 1)$-algebra morphism.
   By universal property of exponential and coproduct, a $(\Sigma^{\uparrow} + 1)$-algebra structure on $(!T)^{!T}$ decomposes into a 4-tuple of maps to $!T$, each one corresponding to a recursive equation. We define these maps, recalling the corresponding recursive equation, in the following table.

Inductive case	Recursive equation	T-module morphism
Abstraction	$(\lambda x.t)[x \mapsto U] = \lambda x.t[x \mapsto U]$	$((!T)^{!T})^{(1)} \times !T \to !T$ $(t, U) \mapsto \lambda.t(U)$
Application	$((s)\ V)[x \mapsto U] = (s[x \mapsto U])\ V[x \mapsto U]$	$(!T)^{!T} \times !(!T)^{!T} \times !T \to !T$ $(s, V, U) \mapsto (s(U))\ V(U)$
Differential application	$(Ds \cdot u)[x \mapsto U] = D(s[x \mapsto U]) \cdot u[x \mapsto U]$	$(!T)^{!T} \times (!T)^{!T} \times !T \to !T$ $(s, u, U) \mapsto D(s(U)) \cdot u(U)$
Substituted variable	$x[x \mapsto U] = U$	$!T \to !T$ $U \mapsto U$

   This covers four out of five recursive equations. The missing one is $y[x \mapsto U] = y$ when $x \neq y$; it corresponds to the morphism $T \to (!T)^{!T}$, which more generally deals with any term not depending on x.
   Let us explain some cases, beginning with the last one. The morphism $!T \to !T$ corresponds to the mapping $U \mapsto x[x \mapsto U]$, thus we choose the identity morphism.
   Next, e.g., the morphism for application is a composite
   $$(!T)^{!T} \times !(!T)^{!T} \times !T \quad \to \quad !T \times !!T \quad \to \quad !(T \times !T) \quad \xrightarrow{!app} \quad !T,$$
   where
   - the first morphism duplicates $!T$ and evaluates both exponentials, and
   - the second follows from the well-known fact that ! is a commutative monad.

   The other cases are similar, and require to lift the other operations to the level of multiterms.

2. Now, following our recipe, we need to give a T-module morphism from T to $(!T)^{!T}$, or equivalently, a T-module morphism $m : T \times !T \to !T$. This morphism corresponds to the mapping $(t, U) \mapsto t[x \mapsto U]$, when t does not depend on x. Thus, we define $m_X(t, U) = t$. More formally, m is the composite $T \times !T \xrightarrow{\pi_1} T \xrightarrow{\eta_! T} !T$.
3. It remains to check that the induced morphism $T \to (!T)^{!T}$ is a $\Sigma^{\uparrow}$-algebra morphism, that is, that this morphism is compatible with each operation, which is routine.

F Proof of Proposition 6

In this section, we show that the modular and the monadic definitions of transition monads are equivalent. The proof consists merely in unfolding the definitions.

Consider the modular definition. We have:

- a finitary monad T on $\mathbf{Set}^{\mathbb{P}}$, that is:
  - an object mapping $T : \mathbf{ob}(\mathbf{Set}^{\mathbb{P}}) \to \mathbf{ob}(\mathbf{Set}^{\mathbb{P}})$, together with
  - morphisms $X \to T(X)$ for the variables, and
  - for each morphism $f : X \to T(Y)$, an extension $f^\star : T(X) \to T(Y)$, subject to the usual equations;
- a finitary T-module $R : \mathbf{Set}^{\mathbb{P}} \to \mathbf{Set}^{\mathbb{S}}$, called the **transition** module, that is:
  - an object mapping $R : \mathbf{ob}(\mathbf{Set}^{\mathbb{P}}) \to \mathbf{ob}(\mathbf{Set}^{\mathbb{S}})$, together with
  - for each morphism $f : X \to T(Y)$, an extension $f^\star : R(X) \to R(Y)$ subject to the usual equations;
- two T-module morphisms $s_i : R \to S_i T$, that is, families of morphisms $R(X) \to S_i(T(X))$ commuting with T-substitution.

The monadic definition was already detailed after Definition 4. It is straightforward to check that the assignement $X \mapsto (src_X, tgt_X : R(X) \to S_i(T(X)))$ defines a monadic transition monad. Conversely, given a monadic transition monad mapping X to some $(src_X, tgt_X : R(X) \to S_i(T(X)))$, the assignement $X \mapsto T(X)$ defines a monad T, the assignment $X \mapsto R(X)$ defines a T-module R, and src and tgt induce T-module morphisms $R \to S_1 T$ and $R \to S_2 T$, respectively. Hence we get a modular transition monad.

G Proof-irrelevant variant

▶ **Proposition 65.** *Let $\mathbf{ITMnd}_{\mathbb{P},\mathbb{S}}(T, S_1, S_2)$ denote the full subcategory of transition monads $\langle src, tgt\rangle : R \to S_1 T \times S_2 T$ such that $\langle src, tgt\rangle$ is a pointwise inclusion. Then, the embedding $U : \mathbf{ITMnd}_{\mathbb{P},\mathbb{S}}(T, S_1, S_2) \hookrightarrow \mathbf{TMnd}_{\mathbb{P},\mathbb{S}}(T, S_1, S_2)$ is reflective.*

Proof. The left adjoint $L : \mathbf{TMnd}_{\mathbb{P},\mathbb{S}}(T, S_1, S_2) \to \mathbf{ITMnd}_{\mathbb{P},\mathbb{S}}(T, S_1, S_2)$ maps a transition monad $\partial : R \to S_1 T \times S_2 T$ to the monomorphism $\overline{R} \hookrightarrow S_1 T \times S_2 T$ obtained from the (strong epi)-mono factorisation[5] of ∂. Then, the natural bijection $\mathbf{TMnd}_{\mathbb{P},\mathbb{S}}(T, S_1, S_2)(T_1, UT_2) \cong \mathbf{ITMnd}_{\mathbb{P},\mathbb{S}}(T, S_1, S_2)(LT_1, T_2)$ follows from the lifting property of strong epimorphisms. ◀

Thanks to Definition 22, we then get a register for proof-irrelevant transition monads from the register **TMndReg** of Definition 63.

[5] As mentioned before, the category of finitary **Set**-valued T-modules is a presheaf category, and thus has (strong epi)-mono factorisations.

H Proof of Lemma 59

In this section, we fix two sets $\mathbb{P}$ and $\mathbb{S}$, a monad T on $\mathbf{Set}^{\mathbb{P}}$, and a $\mathbf{Set}^{\mathbb{S}}$-valued T-module M. We then show that given any signature ρ in $\mathbf{Rule}(T, M)$, Σ_ρ on $T\text{-}\mathbf{Mod}_f(\mathbf{Set}^{\mathbb{S}})/M$ is finitary.

Now, Σ_ρ is a composite of four functors as in (1) with $\prod_i (-)_{\sigma_i}$ replaced by $\mathcal{D}^\rho := \prod_i (-)^{(\vec{p_i})}_{\sigma_i}$ as explain in §5.2.2. The last three of these functors are left adjoints (because we restrict to finitary modules), hence readily finitary. It remains to show that the fourth factor, $\mathcal{D}^\rho/M : T\text{-}\mathbf{Mod}_f(\mathbf{Set}^{\mathbb{S}})/M \to T\text{-}\mathbf{Mod}_f(\mathbf{Set})/\mathcal{D}^\rho(M)$, is finitary. Because the domain functors $T\text{-}\mathbf{Mod}_f(\mathbf{Set}^{\mathbb{S}})/M \to T\text{-}\mathbf{Mod}_f(\mathbf{Set}^{\mathbb{S}})$ and $T\text{-}\mathbf{Mod}_f(\mathbf{Set})/\mathcal{D}^\rho(M) \to T\text{-}\mathbf{Mod}_f(\mathbf{Set})$ create colimits, this reduces to $\mathcal{D}^\rho$ being finitary. But finitary functors are closed under finite products, so, because colimits are pointwise in presheaf categories, this in turn reduces to each $(-)^{(p)}$ being finitary, which follows from their being left adjoints. (They may be viewed as precomposition with an endofunctor of $\mathbf{Kl}(T)$, hence admit a right adjoint given by right Kan extension.)

Type Safety of Rewrite Rules in Dependent Types

Frédéric Blanqui
Université Paris-Saclay, ENS Paris-Saclay, CNRS, Inria, France
Laboratoire Spécification et Vérification, Cachan, France
http://rewriting.gforge.inria.fr/

Abstract

The expressiveness of dependent type theory can be extended by identifying types modulo some additional computation rules. But, for preserving the decidability of type-checking or the logical consistency of the system, one must make sure that those user-defined rewriting rules preserve typing. In this paper, we give a new method to check that property using Knuth-Bendix completion.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Equational logic and rewriting; Theory of computation → Logic and verification

Keywords and phrases subject-reduction, rewriting, dependent types

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.13

Supplementary Material https://github.com/wujuihsuan2016/lambdapi/tree/sr

Acknowledgements The author thanks Jui-Hsuan Wu for his prototype implementation and his comments on a preliminary version of this work.

1 Introduction

The $\lambda\Pi$-calculus, or LF [12], is an extension of the simply-typed λ-calculus with dependent types, that is, types that can depend on values like, for instance, the type $V n$ of vectors of dimension n. And two dependent types like $V n$ and $V p$ are identified as soon as n and p are two expressions having the same value (modulo the evaluation rule of λ-calculus, β-reduction).

In the $\lambda\Pi$-calculus modulo rewriting, function and type symbols can be defined not only by using β-reduction but also by using rewriting rules [19]. Hence, types are identified modulo β-reduction and some user-defined rewriting rules. This calculus has been implemented in a tool called Dedukti [9].

Adding rewriting rules adds a lot of expressivity for encoding logical or type systems. For instance, although the $\lambda\Pi$-calculus has no native polymorphism, one can easily encode higher-order logic or the calculus of constructions by using just a few symbols and rules [8]. As a consequence, various tools have been developed for translating actual terms and proofs from various systems (Coq, OpenTheory, Matita, Focalize, ...) to Dedukti, and back, opening the way to some interoperability between those systems [1]. The Agda system recently started to experiment with rewriting too [7].

To preserve the decidability of type-checking and the logical consistency, it is however essential that the rules added by the user preserve typing, that is, if an expression e has some type T and a rewriting rule transforms e into a new expression e', then e' should have type T too. This property is also very important in programming languages, to avoid some errors (a program declared to return a string should not return an integer).

When working in the simply-typed λ-calculus, it is not too difficult to ensure this property: it suffices to check that, for every rewriting rule $l \hookrightarrow r$, the right-hand side (RHS) r has the same type as the left-hand side (LHS) l, which is decidable.

The situation is however much more complicated when working with dependent types modulo user-defined rewriting rules. As type-checking requires one to decide the equivalence of two expressions, it is undecidable in general to say whether a rewriting rule preserves typing, even β-reduction alone [17].

Note also that, in the $\lambda\Pi$-calculus modulo rewriting, the set of well-typed terms is not fixed but depends on the rewriting rules themselves (it grows when one adds rewriting rules).

Finally, the technique used in the simply-typed case (checking that both the LHS and the RHS have the same type) is not satisfactory in the case of dependent types, as it often forces rule left-hand sides to be non-linear [6], making other important properties (namely confluence) more difficult to establish and the implementation of rewriting less efficient (if it does not use sharing).

▶ **Example 1.** As already mentioned, Dedukti is often used to encode logical systems and proofs coming from interactive or automated theorem provers. For instance, one wants to be able to encode the simply-typed λ-calculus in Dedukti. Using the new Dedukti syntax[1], this can be done as follows (rule variables must be prefixed by `$` to distinguish them from function symbols with the same name):

```
constant symbol T: TYPE // Dedukti type for representing simple types
constant symbol arr: T → T → T // arrow simple type constructor

injective symbol τ: T → TYPE// interprets T elements as Dedukti types
rule τ (arr $x $y) ↪ τ $x → τ $y // (Curry-Howard isomorphism)

// representation of simply-typed λ-terms
symbol lam: Π a b, (τ a → τ b) → τ (arr a b)
symbol app: Π a b, τ (arr a b) → (τ a → τ b)

rule app $a $b (lam $a' $b' $f) $x ↪ $f $x // β-reduction
```

Proving that the above rule preserves typing is not trivial as it is equivalent to proving that β-reduction has the subject-reduction property in the simply-typed λ-calculus. And, indeed, the previous version of Dedukti was unable to prove it.

The LHS is typable if f is of type $\tau(\mathtt{arr}\,a'\,b')$, $\tau(\mathtt{arr}\,a'\,b') \simeq \tau(\mathtt{arr}\,a\,b)$, and x is of type τa. Then, in this case, the LHS is of type τb.

Here, one could be tempted to replace a' by a, and b' by b, so that these conditions are satisfied but this would make the rewriting rule non left-linear and the proof of its confluence problematic [13].

Fortunately, this is not necessary. Indeed, we can prove that the RHS is typable and has the same type as the LHS by using the fact that $\tau(\mathtt{arr}\,a'\,b') \simeq \tau(\mathtt{arr}\,a\,b)$ when the LHS is typable. Indeed, in this case, and thanks to the rule defining τ, f is of type $\tau a \to \tau b$. Therefore, the RHS has type τb as well.

In this paper, we present a new method for doing this kind of reasoning automatically. By using Knuth-Bendix completion [15, 18], the equations holding when a LHS is typable are turned into a convergent (i.e. confluent and terminating) set of rewriting rules, so that the type-checking algorithm of Dedukti itself can be used to check the type of a RHS modulo these equations.

[1] https://github.com/Deducteam/lambdapi

Outline. The paper is organized as follows. In Section 2, we recall the definition of the $\lambda\Pi$-calculus modulo rewriting. In Section 3, we recall what it means for a rewriting rule to preserve typing. In Section 4, we describe a new algorithm for checking that a rewriting rule preserves typing and provide general conditions for ensuring its termination. Finally, in Section 5, we compare this new approach with previous ones and conclude.

2 $\lambda\Pi$-calculus modulo rewriting

Following Barendregt's book on typed λ-calculus [4], the $\lambda\Pi$-calculus is a Pure Type System (PTS) on the set of sorts $\mathcal{S} = \{\star, \Box\}$:[2]

▶ **Definition 2** ($\lambda\Pi$-term algebra). *A $\lambda\Pi$-term algebra is defined by:*

- *a set $\mathcal{F}$ of function symbols,*
- *an infinite set $\mathcal{V}$ of variables,*

such that $\mathcal{V}$, $\mathcal{F}$ and $\mathcal{S}$ are pairwise disjoint.

The set $\mathcal{T}(\mathcal{F},\mathcal{V})$ of $\lambda\Pi$-terms is then inductively defined as follows:

$$t, u := s \in \mathcal{S} \mid x \in \mathcal{V} \mid f \in \mathcal{F} \mid \lambda x : t, u \mid tu \mid \Pi x : t, u$$

where $\lambda x : t, u$ is called an abstraction, tu an application, and $\Pi x : t, u$ a (dependent) product (simply written $t \to u$ if x does not occur in u). As usual, terms are identified modulo renaming of bound variables (x is bound in $\lambda x : t, u$ and $\Pi x : t, u$). We denote by $\mathrm{FV}(t)$ the free variables of t. A term is said to be closed if it has no free variables.

A substitution is a finite map from $\mathcal{V}$ to $\mathcal{T}(\mathcal{F},\mathcal{V})$. It is written as a finite set of pairs. For instance, $\{(x,a)\}$ is the substitution mapping x to a.

Given a substitution σ and a term t, we denote by $t\sigma$ the capture-avoiding replacement of every free occurrence of x in t by its image in σ.

▶ **Definition 3** ($\lambda\Pi$-calculus). *A $\lambda\Pi$-calculus on $\mathcal{T}(\mathcal{F},\mathcal{V})$ is given by:*

- *a function $\Theta : \mathcal{F} \to \mathcal{T}(\mathcal{F},\mathcal{V})$ mapping every function symbol f to a term Θ_f called its type (we will often write $f : A$ instead of $\Theta_f = A$),*
- *a function $\Sigma : \mathcal{F} \to \mathcal{S}$ mapping every function symbol f to a sort Σ_f,*
- *a set $\mathcal{R}$ of rewriting rules $(l, r) \in \mathcal{T}^2$, written $l \hookrightarrow r$, such that $\mathrm{FV}(r) \subseteq \mathrm{FV}(l)$.*

We then denote by $\simeq$ the smallest equivalence relation containing $\hookrightarrow = \hookrightarrow_{\mathcal{R}} \cup \hookrightarrow_\beta$ where $\hookrightarrow_{\mathcal{R}}$ is the smallest relation stable by context and substitution containing $\mathcal{R}$, and $\hookrightarrow_\beta$ is the usual β-reduction relation.

▶ **Example 4.** For representing natural numbers, we can use the function symbols $N : \star$ of sort $\Box$, and the function symbols $0 : N$ and $s : N \to N$ of sort $\star$. Addition can be represented by $+ : N \to N \to N$ of sort $\star$ together with the following set of rules:

$$\begin{array}{rcl} 0 + y & \hookrightarrow & y \\ x + 0 & \hookrightarrow & x \\ x + (sy) & \hookrightarrow & s(x+y) \\ (sx) + y & \hookrightarrow & s(x+y) \\ (x+y)+z & \hookrightarrow & x + (y+z) \end{array}$$

[2] PTS sorts should not be confused with the notion of sort used in first-order logic. The meaning of these sorts will be explained after the definition of typing (Definition 5). Roughly speaking, $\star$ is the type of objects and proofs, and $\Box$ is the type of set families and predicates.

$$\begin{array}{lcl}
\text{(ax)} & \vdash \star : \Box & \\[1ex]
\text{(fun)} & \dfrac{\vdash \Theta_f : \Sigma_f}{\vdash f : \Theta_f} & \\[2ex]
\text{(var)} & \dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} & (x \notin \Gamma) \\[2ex]
\text{(weak)} & \dfrac{\Gamma \vdash t : T \quad \Gamma \vdash A : s}{\Gamma, x : A \vdash t : T} & (x \notin \Gamma) \\[2ex]
\text{(prod)} & \dfrac{\Gamma \vdash A : \star \quad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A, B : s} & \\[2ex]
\text{(app)} & \dfrac{\Gamma \vdash t : \Pi x : A, B \quad \Gamma \vdash a : A}{\Gamma \vdash ta : B\{(x, a)\}} & \\[2ex]
\text{(abs)} & \dfrac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash \Pi x : A, B : s}{\Gamma \vdash \lambda x : A, b : \Pi x : A, B} & \\[2ex]
\text{(conv)} & \dfrac{\Gamma \vdash t : T \quad \Gamma \vdash U : s}{\Gamma \vdash t : U} & (T \simeq U)
\end{array}$$

Figure 1 Typing rules of the $\lambda\Pi$-calculus modulo rewriting.

Note that Dedukti allows overlapping LHS and matching on defined symbols like in this example. (It also allows higher-order pattern matching like in Combinatory Reduction Systems (CRS) [14] but we do not consider this feature in the current paper.)

Throughout the paper, we assume a given $\lambda\Pi$-calculus $\Lambda = (\mathcal{F}, \mathcal{V}, \Theta, \Sigma, \mathcal{R})$.

▶ **Definition 5** (Well-typed terms). *A typing environment is a possibly empty ordered sequence of pairs (x_1, A_1), ..., (x_n, A_n), written $x_1 : A_1, \ldots, x_n : A_n$, where the x_i's are distinct variables and the A_i's are terms.*

A term t has type A in a typing environment Γ if the judgment $\Gamma \vdash t : A$ is derivable from the rules of Figure 1. An environment Γ is valid if some term is typable in it.

A substitution σ is a well-typed substitution from an environment Γ to an environment Γ', written $\Gamma' \vdash \sigma : \Gamma$, if, for all $x : A \in \Gamma$, we have $\Gamma' \vdash x\sigma : A\sigma$.

Note that well-typed substitutions preserve typing: if $\Gamma \vdash t : T$ and $\Gamma' \vdash \sigma : \Gamma$, then $\Gamma' \vdash t\sigma : T\sigma$ [5].

A type-checking algorithm for the $\lambda\Pi$-calculus modulo (user-defined) rewriting rules is implemented in the Dedukti tool [9].

We first recall a number of basic properties that hold whatever $\mathcal{R}$ is and can be easily proved by induction on $\vdash$ [5]:

▶ **Lemma 6.**

(a) *If t is typable, then every subterm of t is typable.*

(b) *$\Box$ is not typable.*

(c) *If $\Gamma \vdash t : T$ then either $T = \Box$ or $\Gamma \vdash T : s$ for some sort s.*

(d) *If $\Gamma \vdash t : \Box$ then t is a kind, that is, of the form $\Pi x_1 : T_1, \ldots, \Pi x_n : T_n, \star$.*

(e) *If $\Gamma \vdash t : T$, $\Gamma \subseteq \Gamma'$ and Γ' is valid, then $\Gamma' \vdash t : T$.*

Throughout the paper, we assume that, for all f, $\vdash \Theta_f : \Sigma_f$. Indeed, if $\vdash \Theta_f : \Sigma_f$ does not hold, then no well-typed term can contain f. (This assumption is implicit in the presentations of LF using signatures [12].)

More importantly, we will assume that $\hookrightarrow$ is confluent on the set $\mathcal{T}(\mathcal{F},\mathcal{V})$ of untyped terms, that is, for all terms $t, u, v \in \mathcal{T}(\mathcal{F},\mathcal{V})$, if $t \hookrightarrow^* u$ and $t \hookrightarrow^* v$, then there exists a term $w \in \mathcal{T}(\mathcal{F},\mathcal{V})$ such that $u \hookrightarrow^* w$ and $v \hookrightarrow^* w$, where $\hookrightarrow^*$ is the reflexive and transitive closure of $\hookrightarrow$.

This condition is required for ensuring that conversion behaves well with respect to products (if $\Pi x : A, B \simeq \Pi x : A', B'$ then $A \simeq A'$ and $B \simeq B'$), which in particular implies subject-reduction for $\hookrightarrow_\beta$.

This last assumption may look strong, all the more so since confluence is undecidable. However this property is satisfied by many systems in practice. For instance, $\hookrightarrow$ is confluent if the left-hand sides of $\mathcal{R}$ are algebraic (Definition 8), linear and do not overlap with each other [21]. This is in particular the case of the rewriting systems corresponding to the function definitions allowed in functional programming languages such as Haskell, Agda, OCaml or Coq. But confluence can be relaxed in some cases: when there are no type-level rewriting rules [3] or when the right-hand sides of type-level rewriting rules are not products [6].

When $\hookrightarrow$ is confluent, the typing relation satisfies additional properties. For instance, the set of typable terms can be divided into three disjoint classes:

- the terms of type $\Box$, called kinds, of the form $\Pi x_1 : A_1, \ldots, \Pi x_n : A_n, \star$;
- the terms whose type is a kind, called predicates;
- the terms whose type is a predicate, called objects.

3 Subject-reduction

A relation $\rhd$ preserves typing (subject-reduction property) if, for all environments Γ and all terms t, u and A, if $\Gamma \vdash t : A$ and $t \rhd u$, then $\Gamma \vdash u : A$.

One can easily check that $\hookrightarrow_\beta$ preserves typing when $\hookrightarrow$ is confluent [5]. Our aim is therefore to check that $\hookrightarrow_\mathcal{R}$ preserves typing too. To this end, it is enough to check that every rule $l \hookrightarrow r \in \mathcal{R}$ preserves typing, that is, for all environments Γ, substitutions σ and terms A, if $\Gamma \vdash l\sigma : A$, then $\Gamma \vdash r\sigma : A$.

A first idea is to require that:

$$(*) \text{ there exist } \Delta \text{ and } B \text{ such that } \Delta \vdash l : B \text{ and } \Delta \vdash r : B.$$

But this condition is not sufficient in general as shown by the following example:

▶ **Example 7.** Consider the rule $f(xy) \hookrightarrow y$ with $f : B \to B$. In the environment $\Delta = x : B \to B, y : B$, we have $\Delta \vdash l : B$ and $\Delta \vdash r : B$. However, in the environment $\Gamma = x : A \to B, y : A$, we have $\Gamma \vdash l : B$ and $\Gamma \vdash r : A$.

The condition (*) is sufficient if the rule left-hand side is a non-variable simply-typed first-order term [3], a notion that we slightly generalize as follows:

▶ **Definition 8** (Pattern). *We assume that the set of variables is split in two disjoint sets, the algebraic variables and the non-algebraic ones, and that there is an injection* $\hat{}$ *from algebraic variables to non-algebraic variables.*

A term is algebraic if it is an algebraic variable or of the form $ft_1 \ldots t_n$ *with each* t_i *algebraic and* f *a function symbol whose type is of the form* $\Pi x_1 : A_1, \ldots, x_n : A_n, B$.

A term is an object-level algebraic term if it is algebraic and all its function symbols are of sort $\star$.

A pattern is an algebraic term of the form $ft_1 \ldots t_n$ *where each* t_i *is an object-level algebraic term.*

The distinction between algebraic and non-algebraic variables is purely technical: for generating equations (Definition 10), we need to associate a type $\widehat{x}$ to every variable x, and we need those variables $\widehat{x}$ to be distinct from one another and distinct from the variables used in rules. To do so, we split the set of variables into two disjoint sets. The ones used in rules are called algebraic, and the others are called non-algebraic. Finally, we ask the function $\widehat{\ }$ to be an injection from the set of algebraic variables to the set of non-algebraic variables.

In the rest of the paper, we also assume that rule left-hand sides are patterns. Hence, every rule is of the form $f\, l_1 \dots l_n \hookrightarrow r$, and we say that a symbol $f \in \mathcal{F}$ is defined if there is in $\mathcal{R}$ a rule of the form $f\, l_1 \dots l_n \hookrightarrow r$.

However, the condition (*) is not satisfactory in the context of dependent types. Indeed, when function symbols have dependent types, it often happens that a term is typable only if it is non-linear. And, with non-left-linear rewriting rules, $\hookrightarrow$ is generally not confluent on untyped terms [13], while there exist many confluence criteria for left-linear rewriting systems [21].

Throughout the paper, we will use the following simple but paradigmatic example to illustrate how our new algorithm works:

▶ **Example 9.** Consider the following rule to define the *tail* function on vectors:

$$\mathit{tail}\ n\ (\mathit{cons}\ x\ p\ v) \hookrightarrow v$$

where $\mathit{tail} : \Pi n : N, V(sn) \to Vn$, $V : N \to \star$, $\mathit{nil} : V0$, $\mathit{cons} : R \to \Pi n : N, Vn \to V(sn)$ and $R : \star$.

For the left-hand side to be typable, we need to take $p = n$, because $\mathit{tail}\ n$ expects an argument of type $V(sn)$, but $\mathit{cons}\ x\ p\ v$ is of type $V(sp)$.

Yet, the rule with $p \neq n$ preserves typing. Indeed, assume that there is an environment Γ, a substitution σ and a term A such that $\Gamma \vdash \mathit{tail}\ n\sigma\ (\mathit{cons}\ x\sigma\ p\sigma\ v\sigma) : A$. By inversion of typing rules, we get $V(n\sigma) \simeq A$, $\Gamma \vdash A : s$ for some sort s, $V(sp\sigma) \simeq V(sn\sigma)$ and $\Gamma \vdash v\sigma : Vp\sigma$. Assume now that V and s are undefined, that is, there is no rule of $\mathcal{R}$ of the form $Vt \hookrightarrow u$ or $st \hookrightarrow u$. Then, by confluence, $p\sigma \simeq n\sigma$. Therefore, $Vp\sigma \simeq A$ and $\Gamma \vdash v\sigma : A$.

Hence, that a rewriting rule $l \hookrightarrow r$ preserves typing does not mean that its left-hand side l must be typable [6]. Actually, if no instance of l is typable, then $l \hookrightarrow r$ trivially preserves typing (since it can never be applied)! The point is therefore to check that any typable instance of $l \hookrightarrow r$ preserves typing.

4 A new subject-reduction criterion

The new criterion that we propose for checking that $l \hookrightarrow r$ preserves typing proceeds in two steps. First, we generate conversion constraints that are satisfied by every typable instance of l (Figure 2). Then, we try to check that r has the same type as l in the type system where the conversion relation is extended with the equational theory generated by the conversion constraints inferred in the first step. For type-checking in this extended type theory to be decidable and implementable using Dedukti itself, we use Knuth-Bendix completion [15] to replace the set of conversion constraints by an equivalent but convergent (i.e. terminating and confluent) set of rewriting rules.

4.1 Inference of typability constraints

We first define an algorithm for inferring typability constraints and then prove its correctness and completeness.

▶ **Definition 10** (Typability constraints). *For every algebraic term t, we assume given a valid environment $\Delta_t = \widehat{y_1} : \star, y_1 : \widehat{y_1}, \ldots, \widehat{y_k} : \star, y_k : \widehat{y_k}$ where $y_1, \ldots, y_k$ are the free variables of t.*

Let $\uparrow$ be the partial function defined in Figure 2. It takes as input a term t and returns a pair $(A, \mathcal{E})$, written $A[\mathcal{E}]$, where A is a term and $\mathcal{E}$ is a set of equations, an equation being a pair of terms (l, r) usually written $l = r$.

A substitution σ satisfies a set $\mathcal{E}$ of equations, written $\sigma \models \mathcal{E}$, if for all equations $a = b \in \mathcal{E}$, $a\sigma \simeq b\sigma$.

$$\frac{}{y \uparrow \widehat{y}[\emptyset]}$$

$$\frac{f : \Pi x_1 : T_1, \ldots, \Pi x_n : T_n, U \qquad t_1 \uparrow A_1[\mathcal{E}_1] \quad t_n \uparrow A_n[\mathcal{E}_n]}{f t_1 \ldots t_n \uparrow U\sigma[\mathcal{E}_1 \cup \ldots \cup \mathcal{E}_n \cup \{A_1 = T_1\sigma, \ldots, A_n = T_n\sigma\}]}$$

$$\text{where } \sigma = \{(x_1, t_1), \ldots, (x_n, t_n)\}$$

Figure 2 Typability constraints.

▶ **Example 11.** In our running example $tail\ n\ (cons\ x\ p\ v) \hookrightarrow v$, we have $cons\ x\ p\ v \uparrow V(sp)[\mathcal{E}_1]$ with $\mathcal{E}_1 = \{\widehat{x} = T, \widehat{p} = N, \widehat{v} = Vp\}$, and $tail\ n\ (cons\ x\ p\ v) \uparrow Vn[\mathcal{E}_2]$ with $\mathcal{E}_2 = \mathcal{E}_1 \cup \{\widehat{n} = N, V(sp) = V(sn)\}$.

▶ **Lemma 12.** *If $\Gamma \vdash \Pi x_1 : T_1, \ldots, \Pi x_n : T_n, U : s$ then, for all i, $\Gamma^{i-1} \vdash T_i : \star$ and $\Gamma^n \vdash U : s$, where $\Gamma^i = \Gamma, x_1 : T_1, \ldots, x_i : T_i$.*

Proof. Since $\hookrightarrow$ is confluent and left-hand sides are patterns, $s \simeq s'$ iff $s = s'$. The result follows then by inversion of typing rules and weakening. ◀

In particular, because $\vdash \Theta_f : \Sigma_f$ for all f, we have:

▶ **Corollary 13.** *For all function symbols $f : \Pi x_1 : T_1, \ldots, \Pi x_n : T_n, U$ and integer i, we have $\Gamma_f^{i-1} \vdash T_i : \star$ and $\Gamma_f^n \vdash U : \Sigma_f$ where $\Gamma_f^i = x_1 : T_1, \ldots, x_i : T_i$.*

▶ **Lemma 14.** *For all environments Γ, terms $t, x_1, T_1, \ldots, x_n, T_n, U, T$ and substitutions σ for $x_1, \ldots, x_n$, if $\Gamma \vdash t : \Pi x_1 : T_1, \ldots, \Pi x_n : T_n, U$ and $\Gamma \vdash t x_1\sigma \ldots x_n\sigma : T$, then $U\sigma \simeq T$ and $\Gamma \vdash \sigma : \Delta^n$ where $\Delta^n = x_1 : T_1, \ldots, x_n : T_n$.*

Proof. Let $\sigma_i = \{(x_1, t_1), \ldots, (x_{i-1}, t_{i-1})\}$. We proceed by induction on n.

- Case $n = 0$. By equivalence of types.
- Case $n > 0$. By inversion of typing rules and weakening, $\Gamma, \Delta^{n-1} \vdash T_n : \star$, $\Gamma \vdash t x_1\sigma_{n-1} \ldots x_{n-1}\sigma_{n-1} : \Pi x_n : A, B$, $\Gamma \vdash x_n\sigma : A$ and $B\{(x_n, x_n\sigma)\} \simeq T$. By induction hypothesis, $\Gamma \vdash \sigma_{n-1} : \Delta^{n-1}$ and $(x_n : T_n\sigma_{n-1})U\sigma_{n-1} \simeq \Pi x_n : A, B$. By substitution, $\Gamma \vdash T_n\sigma_{n-1} : \star$. By confluence , $T_n\sigma_{n-1} \simeq A$ and $U\sigma_{n-1} \simeq B$. Therefore, by conversion, $\Gamma \vdash x_n\sigma : T_n\sigma$ and $\Gamma \vdash \sigma : \Delta^n$. Now, x_n can always be chosen so that $U\sigma = U\sigma_{n-1}\{(x_n, x_n\sigma)\}$. Therefore, $U\sigma \simeq B\{(x_n, x_n\sigma)\} \simeq T$. ◀

▶ **Lemma 15.**

- *(Correctness) For all algebraic terms t, terms T and sets of equations $\mathcal{E}$, if $t \uparrow T[\mathcal{E}]$ then, for all valid environments Γ, substitutions $\widehat{\theta}$ such that $\Gamma \vdash \widehat{\theta} : \Delta_t$ and $\widehat{\theta} \models \mathcal{E}$, we have $\Gamma \vdash t\widehat{\theta} : T\widehat{\theta}$.*
- *(Completeness) For all environments Γ, patterns t, substitutions θ and terms A, if $\Gamma \vdash t\theta : A$, then there are a term T, a set of equations $\mathcal{E}$ and a substitution $\widehat{\theta}$ extending θ such that $t \uparrow T[\mathcal{E}]$, $\widehat{\theta} \models \mathcal{E}$, $\Gamma \vdash \widehat{\theta} : \Delta_t$ and $A \simeq T\widehat{\theta}$.*

Proof.

- (Correctness) By induction on t.
  - Case $t = y$. Then, $T = \widehat{y}$ and $\mathcal{E} = \emptyset$. By assumption, we have $\Gamma \vdash y\theta : \widehat{y}\theta$. Therefore, $\Gamma \vdash t\theta : T\theta$.
  - Case $t = ft_1 \dots t_n$ with $f : \Pi x_1 : T_1, \dots, x_n : T_n, U$, $t_1 \uparrow A_1[\mathcal{E}_1]$, $\dots$, $t_n \uparrow A_n[\mathcal{E}_n]$. Then, $T = U\sigma$ and $\mathcal{E} = \mathcal{E}_1 \cup \dots \cup \mathcal{E}_n \cup \{A_1 = T_1\sigma, \dots, A_n = T_n\sigma\}$ where $\sigma = \{(x_1, t_1), \dots, (x_n, t_n)\}$.

    By Lemma 12, we have $\Gamma_f^{i-1} \vdash T_i : \star$.

    By induction hypothesis, for all i, we have $\Gamma \vdash t_i\widehat{\theta} : A_i\widehat{\theta}$ and $A_i\widehat{\theta} \simeq T_i\sigma\widehat{\theta}$.

    We now prove that, for all i, $\Gamma \vdash T_i\sigma\widehat{\theta} : \star$ and $\Gamma \vdash x_i\sigma\widehat{\theta} : T_i\sigma\widehat{\theta}$, hence that $\Gamma \vdash \sigma : \Gamma_f^i$, by induction on i.
    - * Case $i = 1$. Since $\vdash T_1 : \star$, T_1 is closed and $T_1\sigma\widehat{\theta} = T_1$. Therefore, by weakening, $\Gamma \vdash T_1\sigma\widehat{\theta} : \star$ and, by conversion, $\Gamma \vdash x_1\sigma\widehat{\theta} : T_1\sigma\widehat{\theta}$.
    - * Case $i > 1$. By induction hypothesis, $\Gamma \vdash \sigma\widehat{\theta} : \Gamma_f^{i-1}$. Since $\Gamma_f^{i-1} \vdash T_i : \star$, by substitution, we get $\Gamma \vdash T_i\sigma\widehat{\theta} : \star$. Therefore, by conversion, $\Gamma \vdash x_i\sigma\widehat{\theta} : T_i\sigma\widehat{\theta}$.

    Hence, $\Gamma \vdash \sigma\widehat{\theta} : \Gamma_f^n$. Now, since $\Gamma_f^n \vdash fx_1 \dots x_n : U$, by substitution, we get $\Gamma \vdash t : U\sigma\widehat{\theta}$.
- (Completeness) We first prove completeness for object-level algebraic terms t such that $\Gamma \vdash A : \star$, by induction on t.
  - Case $t = y$. We take $T = \widehat{y}$, $\mathcal{E} = \emptyset$ and $\widehat{\theta} = \theta \cup \{(\widehat{y}, A)\}$. We have $t \uparrow T[\mathcal{E}]$, $\widehat{\theta} \models \mathcal{E}$ and $A \simeq T\widehat{\theta}$. Now, $\Gamma \vdash y\widehat{\theta} : \widehat{y}\widehat{\theta}$ and $\Gamma \vdash \widehat{y}\widehat{\theta} : \star$. Therefore, $\Gamma \vdash \widehat{\theta} : \Delta_t$.
  - Case $t = ft_1 \dots t_n$ with $f : \Pi x_1 : T_1, \dots, x_n : T_n, U$. By Lemma 12, for all i, we have $\Gamma_f \vdash x_i : T_i$ and $\Gamma_f \vdash T_i : \star$, where $\Gamma_f = x_1 : T_1, \dots, x_n : T_n$. By Lemma 14 because $\vdash \Theta_f : \Sigma_f$ for all f, we have $A \simeq U\sigma\theta$ and $\Gamma \vdash \sigma\theta : \Gamma_f$. Hence, by substitution, for all i, we have $\Gamma \vdash t_i\theta : T_i\sigma\theta$ and $\Gamma \vdash T_i\sigma\theta : \star$. Therefore, by induction hypothesis, there are A_i, $\mathcal{E}_i$ and $\widehat{\theta}_i$ extending θ such that $t_i \uparrow A_i[\mathcal{E}_i]$, $\widehat{\theta}_i \models \mathcal{E}_i$, $\Gamma \vdash \widehat{\theta}_i : \Delta_{t_i}$ and $T_i\sigma\theta \simeq A_i\widehat{\theta}_i$. Then, let $T = U\sigma$, $\mathcal{E} = \mathcal{E}_1 \cup \dots \cup \mathcal{E}_n \cup \{(A_1, T_1\sigma), \dots, (A_n, T_n\sigma)\}$, $y\widehat{\theta} = y\theta$ if $y \in \mathrm{FV}(t)$, and $\widehat{y}\widehat{\theta} = \widehat{y}\widehat{\theta}_i$ where i is the smallest integer such that $y \in \mathrm{FV}(t_i)$. Then, we have $t \uparrow T[\mathcal{E}]$ and $A \simeq U\sigma\theta = T\widehat{\theta}$.

    If $y \in \mathrm{FV}(t_i) \cap \mathrm{FV}(t_j)$, then $y\widehat{\theta}_i = y\theta = y\widehat{\theta}_j$ since $\widehat{\theta}_i$ and $\widehat{\theta}_j$ are both extensions of θ. Now, if $\Gamma \vdash y\widehat{\theta}_i : \widehat{y}\widehat{\theta}_i$ and $\Gamma \vdash y\widehat{\theta}_j : \widehat{y}\widehat{\theta}_j$ then, by equivalence of types, $\widehat{y}\widehat{\theta}_i \simeq \widehat{y}\widehat{\theta}_j$. Therefore, $\widehat{\theta} \models \mathcal{E}$ and $\Gamma \vdash \widehat{\theta} : \Delta_t$.

  Let now t be a pattern. By definition, t is of the form $ft_1 \dots t_n$ with $f : \Pi x_1 : T_1, \dots, x_n : T_n, U$ and each t_i an object-level algebraic term. As we have seen above, for all i, we have $\Gamma \vdash t_i\theta : T_i\sigma\theta$ and $\Gamma \vdash T_i\sigma\theta : \star$. Therefore, by completeness for object level algebraic terms, there are A_i, $\mathcal{E}_i$ and $\widehat{\theta}_i$ extending θ such that $t_i \uparrow A_i[\mathcal{E}_i]$, $\widehat{\theta}_i \models \mathcal{E}_i$, $\Gamma \vdash \widehat{\theta}_i : \Delta_{t_i}$ and $T_i\sigma\theta \simeq A_i\widehat{\theta}_i$. We can now conclude like in the previous case. ◀

▶ **Example 16.** In our running example *tail* n (*cons* x p v) $\hookrightarrow v$, we have seen that *cons* x p $v \uparrow V(sp)[\mathcal{E}_1]$ with $\mathcal{E}_1 = \{\widehat{x} = T, \widehat{p} = N, \widehat{v} = Vp\}$, and *tail* n (*cons* x p v) $\uparrow Vn[\mathcal{E}_2]$ with $\mathcal{E}_2 = \mathcal{E}_1 \cup \{\widehat{n} = N, V(sp) = V(sn)\}$. This means that, if σ is a substitution and (*tail* n (*cons* x p v))σ is typable, then $\sigma \models \mathcal{E}_2$. In particular, $V(sp\sigma) \simeq V(sn\sigma)$.

4.2 Type-checking modulo typability constraints

For checking that the right-hand side of a rewriting rule $l \hookrightarrow r$ has the same type as the left-hand side modulo the typability constraints $\mathcal{E}$ of the left hand-side, we introduce a new $\lambda\Pi$-calculus as follows:

▶ **Definition 17.** *Given a pattern l and a set of equations $\mathcal{E}$ such that $\mathcal{R}$ contains no variable of $\{x \mid x \in \mathrm{FV}(l)\} \cup \{\widehat{x} \mid x \in \mathrm{FV}(l)\}$*[3]*, we define a new $\lambda\Pi$-calculus $\Lambda_{l,\mathcal{E}} = (\mathcal{F}', \mathcal{V}', \Theta', \Sigma', \mathcal{R}')$ where:*

- $\mathcal{F}' = \mathcal{F} \cup \{x \mid x \in \mathrm{FV}(l)\} \cup \{\widehat{x} \mid x \in \mathrm{FV}(l)\}$
- $\mathcal{V}' = \mathcal{V} - (\{x \mid x \in \mathrm{FV}(l)\} \cup \{\widehat{x} \mid x \in \mathrm{FV}(l)\})$
- $\Theta' = \Theta \cup \{(x, \widehat{x}) \mid x \in \mathrm{FV}(l)\} \cup \{(\widehat{x}, \star) \mid x \in \mathrm{FV}(l)\}$
- $\Sigma' = \Sigma \cup \{(x, \star) \mid x \in \mathrm{FV}(l)\} \cup \{(\widehat{x}, \Box) \mid x \in \mathrm{FV}(l)\}$
- $\mathcal{R}' = \mathcal{R} \cup \mathcal{E} \cup \mathcal{E}^{-1}$*, where* $l = r \in \mathcal{E}^{-1}$ *iff* $r = l \in \mathcal{E}$*.*

We denote by $\simeq_{l,\mathcal{E}}$ the conversion relation of $\Lambda_{l,\mathcal{E}}$, and by $\vdash_{l,\mathcal{E}}$ its typing relation.

$\Lambda_{l,\mathcal{E}}$ is similar to Λ except that the symbols of $\{x \mid x \in \mathrm{FV}(l)\} \cup \{\widehat{x} \mid x \in \mathrm{FV}(l)\}$ are not variables but function symbols, and that the set of rewriting rules is extended by $\mathcal{E} \cup \mathcal{E}^{-1}$ which, in $\Lambda_{l,\mathcal{E}}$, is a set of closed rewriting rules (rules and equations are synonyms: they both are pairs of terms).

▶ **Lemma 18.** *For all patterns l, sets of equations $\mathcal{E}$ and substitutions σ in Λ, and for all terms t, u in $\Lambda_{l,\mathcal{E}}$, if $\sigma \models \mathcal{E}$ and $t \simeq_{l,\mathcal{E}} u$, then $t\sigma \simeq u\sigma$.*[4]

Proof. Immediate as each application of an equation $(a, b) \in \mathcal{E} \cup \mathcal{E}^{-1}$ can be replaced by a conversion $a \simeq b$. ◀

▶ **Theorem 19.** *For all patterns l, sets of equations $\mathcal{E}$, and terms T, r in Λ, if $l \uparrow T[\mathcal{E}]$ and $\vdash_{l,\mathcal{E}} r : T$, then $l \hookrightarrow r$ preserves typing in Λ.*

Proof. Let Δ be an environment, σ be a substitution and A be a term of Λ such that $\Delta \vdash l\sigma : A$. By Lemma 15 (completeness), there are a term T', a set of equations $\mathcal{E}'$ and a substitution $\widehat{\sigma}$ extending σ such that $t \uparrow T'[\mathcal{E}']$, $\widehat{\sigma} \models \mathcal{E}'$, $\Delta \vdash \widehat{\sigma} : \Delta_t$ and $A \simeq T'\widehat{\sigma}$. Since $\uparrow$ is a function, we have $T' = T$ and $\mathcal{E}' = \mathcal{E}$.

We now prove that, if $\Gamma \vdash_{l,\mathcal{E}} t : T$, then $\Delta, \Gamma\widehat{\sigma} \vdash t\widehat{\sigma} : T\widehat{\sigma}$, by induction on $\vdash_{l,\mathcal{E}}$ (note that $\widehat{\sigma}$ replaces function symbols by terms).

(fun) $\dfrac{\vdash_{l,\mathcal{E}} \Theta'_f : \Sigma'_f}{\vdash_{l,\mathcal{E}} f : \Theta'_f}$. By induction hypothesis, we have $\Delta \vdash \Theta'_f\widehat{\sigma} : \Sigma'_f\widehat{\sigma} = \Sigma'_f$.

- Case $f \in \mathcal{F}$. Then, $f\widehat{\sigma} = f$, $\Theta'_f\widehat{\sigma} = \Theta'_f = \Theta_f$ and $\Sigma'_f = \Sigma_f$. By inverting typing rules, we get $\vdash \Theta_f : \Sigma_f$. Therefore, by (fun) and (weak), $\Delta \vdash f : \Theta_f$, that is, $\Delta \vdash f\widehat{\sigma} : \Theta_f\widehat{\sigma}$.
- Case $f = x \in \mathrm{FV}(l)$. Then, $f\widehat{\sigma} = x\sigma$ and $\Theta'_f\widehat{\sigma} = \widehat{x}\widehat{\sigma}$. Therefore, $\Delta \vdash f\widehat{\sigma} : \Theta_f\widehat{\sigma}$ since $\Delta \vdash x\widehat{\sigma} : \widehat{x}\widehat{\sigma}$.
- Case $f = \widehat{x}$ with $x \in \mathrm{FV}(l)$. Then, $f\widehat{\sigma} = \widehat{x}\widehat{\sigma}$ and $\Theta'_f\widehat{\sigma} = \star$. Therefore, $\Delta \vdash f\widehat{\sigma} : \Theta_f\widehat{\sigma}$ since $\Delta \vdash \widehat{x}\widehat{\sigma} : \star$.

(conv) $\dfrac{\Gamma \vdash_{l,\mathcal{E}} t : T \quad T \simeq_{l,\mathcal{E}} U \quad \Gamma \vdash_{l,\mathcal{E}} U : s}{\Gamma \vdash_{l,\mathcal{E}} t : U}$. By induction hypothesis, $\Delta, \Gamma\widehat{\sigma} \vdash t\widehat{\sigma} : T\widehat{\sigma}$ and $\Delta, \Gamma\widehat{\sigma} \vdash U\widehat{\sigma} : s$. By Lemma 18, $T\widehat{\sigma} \simeq U\widehat{\sigma}$ since $\widehat{\sigma} \models \mathcal{E}$. Hence, by (conv), $\Delta, \Gamma\widehat{\sigma} \vdash t\widehat{\sigma} : U\widehat{\sigma}$.

- The other cases follow easily by induction hypothesis.

[3] This can always be done by renaming variables.

[4] Note that, here, we extend the notion of substitution by taking maps on $\mathcal{V} \cup \mathcal{F}$.

Hence, we have $\Delta \vdash r\widehat{\sigma} : T\widehat{\sigma}$. Since $\mathrm{FV}(r) \subseteq \mathrm{FV}(l)$, we have $r\widehat{\sigma} = r\sigma$. Since $\Delta \vdash l\sigma : A$, by Lemma 6, either $\Delta \vdash A : s$ for some sort s, or $A = \Box$ and $l\sigma$ is of the form $\Pi x_1 : A_1, \dots, \Pi x_k : A_k, \star$. Since l is a pattern, l is of the form $f l_1 \dots l_n$. Therefore, $\Delta \vdash A : s$ and, by (conv), $\Delta \vdash r\sigma : A$. ◀

▶ **Example 20.** We have seen that $cons\ x\ p\ v \uparrow V(sp)[\mathcal{E}_1]$ with $\mathcal{E}_1 = \{\widehat{x} = T, \widehat{p} = N, \widehat{v} = Vp\}$, and $tail\ n\ (cons\ x\ p\ v) \uparrow Vn[\mathcal{E}_2]$ with $\mathcal{E}_2 = \mathcal{E}_1 \cup \{\widehat{n} = N, V(sp) = V(sn)\}$. After the previous theorem, the rewriting rule defining *tail* preserves typing if we can prove that $\vdash_{l,\mathcal{E}_2} v : Vn$ where, in $\Lambda_{l,\mathcal{E}_2}$, v is a function symbol of type $\widehat{v}$ and sort $\star$, and types are identified modulo $\simeq$ and the equations of $\mathcal{E}_2$. But this is not possible since $v : Vp$ and $Vp \not\simeq_{l,\mathcal{E}_2} Vn$. Yet, if $\sigma \models V(sp) = V(sn)$ and V and s are undefined then, by confluence, $\sigma \models p = n$ and thus $\sigma \models Vp = Vn$. We therefore need to simplify the set of equations before type-checking the right-hand side.

4.3 Simplification of typability constraints

In this section, we show that Theorem 19 can be generalized by using any valid simplification relation, and give an example of such a relation.

▶ **Definition 21** (Valid simplification relation). *A relation $\rightsquigarrow$ on sets of equations is valid if, for all sets of equations $\mathcal{D}, \mathcal{D}'$ and substitutions σ, if $\sigma \models \mathcal{D}$ and $\mathcal{D} \rightsquigarrow \mathcal{D}'$, then $\sigma \models \mathcal{D}'$.*

Theorem 19 can be easily generalized as follows:

▶ **Theorem 22** (Preservation of typing). *For all patterns l, sets of equations $\mathcal{D}, \mathcal{E}$, and terms T, r in Λ, if $l \uparrow T[\mathcal{D}]$, $\mathcal{D} \rightsquigarrow^* \mathcal{E}$ and $\vdash_{l,\mathcal{E}} r : T$, then $l \hookrightarrow r$ preserves typing in Λ.*

We have seen in the previous example that, thanks to confluence, $\sigma \models p = n$ whenever $\sigma \models sp = sn$ and s is undefined. But this last condition is a particular case of a more general property:

▶ **Definition 23** (I-injectivity). *Given $f : \Pi x_1 : T_1, \dots, \Pi x_n : T_n, U$ and a set $I \subseteq \{1, \dots, n\}$, we say that f is I-injective when, for all $t_1, u_1, \dots, t_n, u_n$, if $f t_1 \dots t_n \simeq f u_1 \dots u_n$ and, for all $i \notin I$, $t_i \simeq u_i$, then, for all $i \in I$, $t_i \simeq u_i$.*

For instance, f is $\{1, \dots, n\}$-injective if f is undefined. The new version of Dedukti allows users to declare if a function symbol is I-injective (like the function τ in Example 1), and a procedure for checking I-injectivity of function symbols defined by rewriting rules has been developed and implemented in Dedukti [22]. For instance, the function symbol τ of Example 1, which is defined by the rule $\tau(\mathrm{arr}\, x\, y) \hookrightarrow \tau x \to \tau y$, can be proved to be $\{1\}$-injective.

Clearly, I-injectivity can be used to define a valid simplification relation. In fact, one can easily check that the following simplification rules are valid too:

▶ **Lemma 24.** *The relation defined in Figure 3 is a valid simplification relation.*

Proof. We only detail the first rule which says that, if some substitution σ validates some equation $t = u$, that is, if $t\sigma \simeq u\sigma$, then σ validates any equation $t' = u'$ where t' and u' are reducts of t and u respectively. Indeed, since t' is a reduct of t, $t' \simeq t$. Similarly, $u' \simeq u$. Therefore, by stability of conversion by substitution and transitivity, $t'\sigma \simeq u'\sigma$. ◀

▶ **Example 25.** We can now handle our running example. We have $cons\ x\ p\ v \uparrow V(sp)[\mathcal{E}_1]$ with $\mathcal{E}_1 = \{\widehat{x} = T, \widehat{p} = N, \widehat{v} = Vp\}$, $l = tail\ n\ (cons\ x\ p\ v) \uparrow Vn[\mathcal{E}_2]$ with $\mathcal{E}_2 = \mathcal{E}_1 \cup \{\widehat{n} = N, V(sp) = V(sn)\}$, and $\mathcal{E}_2 \rightsquigarrow^* \mathcal{E}_2' = \mathcal{E}_1 \cup \{\widehat{n} = N, p = n\}$ since V and s are $\{1\}$-injective. Therefore, $\vdash_{l,\mathcal{E}_2'} v : Vn$ and $l \hookrightarrow v$ preserves typing.

$$\begin{array}{rcl}
\mathcal{D} \uplus \{t = u\} & \rightsquigarrow & \mathcal{D} \cup \{t' = u'\} \text{ if } t \hookrightarrow^* t' \text{ and } u \hookrightarrow^* u' \\
\mathcal{D} \uplus \{\Pi x : t_1, t_2 = \Pi x : u_1, u_2\} & \rightsquigarrow & \mathcal{D} \cup \{t_1 = u_1, t_2 = u_2\} \text{ if } x \text{ is fresh} \\
\mathcal{D} \uplus \{f t_1 \dots t_n = f u_1 \dots u_n\} & \rightsquigarrow & \mathcal{D} \cup \{t_i = u_i \mid i \in I\} \\
 & & \text{if } f \text{ is } I\text{-injective and } \forall i \notin I, t_i \simeq_{l,\mathcal{D}} u_i
\end{array}$$

Figure 3 Some valid simplification rules on typability constraints.

The above simplification relation works for the rewriting rule defining *tail* but may not be sufficient in more general situations:

▶ **Example 26.** Let $\mathcal{D}$ be the set of equations $\{fct = ga, fcu = gb, a = b\}$ and assume that f is $\{2\}$-injective. Then the equation $t = u$ holds as well, but $\mathcal{D}$ cannot be simplified by the above rules because it contains no equation of the form $fct = fcu$.

We leave for future work the development of more general simplification relations.

4.4 Decidability conditions

We now discuss the decidability of type-checking in $\Lambda_{l,\mathcal{E}}$ and of the simplification relation based on injectivity, assuming that $\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R}$ is terminating and confluent so that type-checking is decidable in Λ. In both cases, we have to decide $\simeq_{l,\mathcal{E}}$, the reflexive, symmetric and transitive closure of $\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R} \cup \hookrightarrow_\mathcal{E} \cup \hookrightarrow_{\mathcal{E}^{-1}}$, where $\mathcal{E}$ is a set of closed equations.

As it is well known, an equational theory is decidable if there exists a convergent (i.e. terminating and confluent) rewriting system having the same equational theory: to decide whether two terms are equivalent, it suffices to check that their normal forms are identical.

In [15], Knuth and Bendix introduced a procedure to compute a convergent rewriting system included in some termination ordering, when equations are algebraic. Interestingly, this procedure always terminates when equations are closed, if one takes a termination ordering that is total on closed terms like the lexicographic path ordering $>_{lpo}$ wrt any total order $>$ on function symbols (for more details, see for instance [2]).

For the sake of self-contentness, we recall in Figure 4 a rule-based definition of closed completion. These rules operate on a pair $(\mathcal{E}, \mathcal{D})$ made of a set of equations $\mathcal{E}$ and a set of rules $\mathcal{D}$. Starting from $(\mathcal{E}, \emptyset)$, completion consists in applying these rules as long as possible. This process necessarily ends on $(\emptyset, \mathcal{D})$ where $\mathcal{D}$ is terminating (because $\mathcal{D} \subseteq >_{lpo}$) and confluent (because it has no critical pairs).

$$\begin{array}{rcl}
(\mathcal{E} \uplus \{l = r\}, \mathcal{D}) & \rightsquigarrow & (\mathcal{E}, \{l \hookrightarrow r\} \cup \mathcal{D}) \text{ if } l > r \\
(\mathcal{E} \uplus \{l = r\}, \mathcal{D}) & \rightsquigarrow & (\mathcal{E}, \{r \hookrightarrow l\} \cup \mathcal{D}) \text{ if } l < r \\
(\mathcal{E} \uplus \{t = t\}, \mathcal{D}) & \rightsquigarrow & (\mathcal{E}, \mathcal{D}) \\
(\mathcal{E}, \{l[g] \hookrightarrow r, g \hookrightarrow d\} \uplus \mathcal{D}) & \rightsquigarrow & (\mathcal{E} \cup \{l[d] = r\}, \{g \hookrightarrow d\} \cup \mathcal{D}) \\
(\mathcal{E}, \{l \hookrightarrow r[g], g \hookrightarrow d\} \uplus \mathcal{D}) & \rightsquigarrow & (\mathcal{E}, \{l \hookrightarrow r[d], g \hookrightarrow d\} \cup \mathcal{D})
\end{array}$$

Figure 4 Rules for closed completion.

We leave for future work the extension of this procedure to the case of non-algebraic, and possibly higher-order, equations.

If we apply this procedure to the set $\mathcal{E}$ of equations (assuming that they are algebraic), we get that $\simeq_{l,\mathcal{E}}$ is the reflexive, symmetric and transitive closure of $\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R} \cup \hookrightarrow_\mathcal{D}$, where

$\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R}$ and $\hookrightarrow_\mathcal{D}$ are both terminating and confluent. However, termination is not modular in general, even when combining two systems having no symbols in common [20].

There exists many results on the modularity of confluence and termination of first-order rewriting systems when these systems have no symbols in common, or just undefined symbols (see for instance [11] for some survey). But, here, we have higher-order rewriting rules that may share defined symbols.

So, instead, we may try to apply general modularity results on abstract relations [10]. In particular, for all terminating relations P and Q, $P \cup Q$ terminates if P steps can be postponed, that is, if $PQ \subseteq QP^*$. In our case, we may try to postpone the $\mathcal{D}$ steps:

▶ **Lemma 27.** *For all sets of higher-order rewriting rules $\mathcal{R}$ and $\mathcal{D}$, we have that $\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R} \cup \hookrightarrow_\mathcal{D}$ terminates if:*

(a) *$\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R}$ and $\hookrightarrow_\mathcal{D}$ terminate,*

(b) *$\mathcal{R}$ is left-linear,*

(c) *$\mathcal{D}$ is closed,*

(d) *no right-hand side of $\mathcal{D}$ is $\beta\mathcal{R}$-reducible or headed by an abstraction,*

(e) *no right-hand side of $\mathcal{D}$ unifies with a non-variable subterm of a left-hand side of $\mathcal{R}$.*

Proof. As usual, we define positions in a term as words on $\{1,2\}$: $\mathrm{Pos}(s) = \mathrm{Pos}(x) = \mathrm{Pos}(f) = \{\varepsilon\}$, the empty word representing the root position, and $\mathrm{Pos}(tu) = \mathrm{Pos}(\lambda x : t, u) = \mathrm{Pos}(\Pi x : t, u) = 1 \cdot \mathrm{Pos}(t) \cup 2 \cdot \mathrm{Pos}(u)$.

Assume that $t \hookrightarrow_\mathcal{D} u$ at position p and $u \hookrightarrow_{\beta\mathcal{R}} v$ at position q. If p and q are disjoint, then these reductions can be trivially permuted: $t \hookrightarrow_{\beta\mathcal{R}}\hookrightarrow_\mathcal{D} v$. The case $p \leq q$ (p prefix of q) is not possible since $\mathcal{D}$ is closed (c) and no right-hand side of $\mathcal{D}$ is $\beta\mathcal{R}$-reducible (d). So, we are left with the case $q < p$:

- Case $u \hookrightarrow_\beta v$. The case $p = q1$ is not possible since no right-hand side of $\mathcal{D}$ is headed by an abstraction (d). So, $t|_q$ is of the form $(\lambda x : A, b)a$ and the $\mathcal{D}$ step is in A, b or a. Therefore, $t \hookrightarrow_\beta \hookrightarrow_\mathcal{D}^* v$.
- Case $u \hookrightarrow_\mathcal{R} v$, that is, when $u|_q = l\sigma$ where l is a left-hand side of a rule of $\mathcal{R}$. The case $p = qs$ where s is a non-variable position of l is not possible because no non-variable subterm of a left-hand side of $\mathcal{R}$ unifies with a right-hand side of $\mathcal{D}$ (e). Therefore, since l is left-linear (b), $t|_q$ is of the form $l\theta$ for some substitution θ, and the $\mathcal{D}$ step occurs in some $x\theta$. Hence, $t \hookrightarrow_\mathcal{R} \hookrightarrow_\mathcal{D}^* v$. ◀

▶ **Example 28.** As we have already seen, the typability conditions of $l = \mathit{tail}\ n\ (\mathit{cons}\ x\ p\ v)$ is the set of equations $\mathcal{E} = \{\widehat{x} = T, \widehat{p} = N, \widehat{v} = Vp, \widehat{n} = N, V(sp) = V(sn)\}$. By taking $\widehat{x} > \widehat{v} > \widehat{p} > \widehat{n} > V > T > N > s > p > n$ as total order on function symbols, the Knuth-Bendix completion procedure yields with $>_{lpo}$ the rewriting system $\mathcal{D} = \{\widehat{x} \hookrightarrow T, \widehat{p} \hookrightarrow N, \widehat{v} \hookrightarrow Vp, \widehat{n} \hookrightarrow N, V(sp) \hookrightarrow V(sn)\}$. After Lemma 27, $\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R} \cup \hookrightarrow_\mathcal{D}$ is convergent if $\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R}$ is convergent, $\mathcal{R}$ is left-linear and V and s are undefined. This works as well if, instead of $\mathcal{E}$, we use its simplification $\mathcal{E}' = \{\widehat{x} = T, \widehat{p} = N, \widehat{v} = Vp, \widehat{n} = N, p = n\}$. In this case, we get the rewriting system $\mathcal{D} = \{\widehat{x} \hookrightarrow T, \widehat{p} \hookrightarrow N, \widehat{v} \hookrightarrow Vn, \widehat{n} \hookrightarrow N, p \hookrightarrow n\}$.

▶ **Example 29.** Finally, let's come back to the rewriting rule $\mathrm{app}\, a\, b\, (\mathrm{lam}\, a'\, b'\, f)\, x \hookrightarrow f\, x$ of Example 1 encoding the β-reduction of simply-type λ-calculus. As already mentioned, the previous version of Dedukti was unable to prove that this rule preserves typing. Thanks to our new algorithm, the new version of Dedukti[5] can now do it.

[5] `https://github.com/Deducteam/lambdapi`

The computability constraints of the LHS are $\widehat{f} = \tau a' \to \tau b'$, $\tau a' \to \tau b' = \tau(\mathrm{arr}\, a\, b)$ and $\widehat{x} = \tau a$. Preservation of typing cannot be proved without simplifying this set of equations to $\widehat{f} = \tau a' \to \tau b'$, $\tau a' = \tau a$, $\tau b' = \tau b$ and $\widehat{x} = \tau a$.

Then, any total order on function symbols allows to prove preservation of typing. For instance, by taking $\widehat{f} > \to > a' > a > b' > b$, we get the rewriting rules $\widehat{f} \hookrightarrow \tau a \to \tau b$, $\tau a' \hookrightarrow \tau a$, $\tau b' \hookrightarrow \tau b$ and $\widehat{x} \hookrightarrow \tau a$, so that one can easily check that, modulo these rewriting rules, $f\, x$ has type τb. Therefore, $\mathrm{app}\, a\, b\, (\mathrm{lam}\, a'\, b'\, f)\, x \hookrightarrow f\, x$ preserves typing.

Note that the result does not depend on the total order taken on function symbols. For instance, if one takes $\widehat{f} > \to > a > a' > b' > b$ (flipping the order of a and a'), we get the rewriting rules $\widehat{f} \hookrightarrow \tau a' \to \tau b$, $\tau a \hookrightarrow \tau a'$, $\tau b' \hookrightarrow \tau b$ and $\widehat{x} \hookrightarrow \tau a'$. In this case, $f\, x$ has type τb as well. Flipping the order of b and b' would work as well.

5 Related works and conclusion

The problem of type safety of rewriting rules in dependent type theory modulo rewriting has been first studied for simply-typed function symbols by Barbanera, Fernández and Geuvers in [3]. In [6], the author extended these results to polymorphically and dependently typed function symbols, and showed that rule left-hand sides do not need to be typable for rewriting to preserve typing. This was later studied in more details and implemented in Dedukti by Saillard [17]. In this approach, one first extracts a substitution ρ (called a pre-solution in Saillard's work) from the typability constraints of the left-hand side l and check that, if l is of type A, then the right-hand side r is of type $A\rho$ (in the same system). For instance, from the simplified set of constraints $\mathcal{E}' = \{\widehat{x} = T, \widehat{p} = N, \widehat{v} = Vp, \widehat{n} = N, p = n\}$ of our running example, one can extract the substitution $\rho = \{(n, p)\}$ and check that v has type $(Vn)\rho = Vp$. However, it is not said how to compute useful pre-solutions (note that we can always take the identity as pre-solution). In practice, the pre-solution is often given by the user thanks to annotations in rules. A similar mechanism called inaccessible or "dot" patterns exists in Agda too [16].

An inconvenience of this approach is that, in some cases, no useful pre-solution can be extracted. For instance, if, in the previous example, we take the original set of constraints $\mathcal{E} = \{\widehat{x} = T, \widehat{p} = N, \widehat{v} = Vp, \widehat{n} = N, V(sp) = V(sn)\}$ instead of its simplified version $\mathcal{E}'$, then we cannot extract any useful pre-solution.

In this paper, we proposed a more general approach where we check that the right-hand side has the same type as the left-hand side modulo the equational theory generated by the typability constraints of the left-hand side seen as closed equations (Theorem 19). A prototype implementation is available on:
`https://github.com/wujuihsuan2016/lambdapi/tree/sr`.

To ensure the decidability of type-checking in this extended system, we propose to replace these equations by an equivalent but convergent rewriting system using Knuth-Bendix completion [15] (which always terminates on closed equations), and provide conditions for preserving the termination and confluence of the system when adding these new rules (Lemma 27). This approach has also the advantage that Dedukti itself can be used to check the type safety of user-defined Dedukti rules.

We also showed that, for the algorithm to work, the typability constraints sometimes need to be simplified first, using the fact that some function symbols are injective (Theorem 22). It would be interesting to be able to detect or check injectivity automatically (see [22] for preliminary results on this topic), and also to find a simplification procedure more general than the one of Figure 3.

References

1 A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. Dedukti: a logical framework based on the $\lambda\Pi$-calculus modulo theory, 2019. Draft.

2 F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

3 F. Barbanera, M. Fernández, and H. Geuvers. Modularity of strong normalization in the algebraic-λ-cube. *Journal of Functional Programming*, 7(6):613–660, 1997.

4 H. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of logic in computer science. Volume 2. Background: computational structures*, pages 117–309. Oxford University Press, 1992.

5 F. Blanqui. *Théorie des types et récriture.* PhD thesis, Université Paris-Sud, France, 2001.

6 F. Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005.

7 J. Cockx and A. Abel. Sprinkles of Extensionality for Your Vanilla Type Theory (abstract). Presented at TYPES'2016.

8 D. Cousineau and G. Dowek. Embedding pure type systems in the lambda-Pi-calculus modulo. In *Proceedings of the 8th International Conference on Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science 4583, 2007.

9 Dedukti. `https://deducteam.github.io/`, 2018.

10 H. Doornbos and B. von Karger. On the union of well-founded relations. *Logic Journal of the Interest Group in Pure and applied Logic*, 6(2):195–201, 1998.

11 B. Gramlich. Modularity in term rewriting revisited. *Theoretical Computer Science*, 464:3–19, 2012.

12 R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

13 J. W. Klop. *Combinatory reduction systems.* PhD thesis, Utrecht Universiteit, NL, 1980. Published as Mathematical Center Tract 129.

14 J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.

15 D. Knuth and P. Bendix. Simple word problems in universal algebra. In *Computational problems in abstract algebra, Proceedings of a Conference held at Oxford in 1967*, pages 263–297. Pergamon Press, 1970. Reproduced in [18].

16 U. Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers University of Technology, Sweden, 2007.

17 R. Saillard. *Type checking in the Lambda-Pi-calculus modulo: theory and practice.* PhD thesis, Mines ParisTech, France, 2015.

18 J. H. Siekmann and G. Wrightson, editors. *Automation of reasoning. 2: classical papers on computational logic 1967-1970.* Symbolic computation. Springer, 1983.

19 TeReSe. *Term rewriting systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 2003.

20 Y. Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25(3):141–143, 1987.

21 V. van Oostrom. *Confluence for abstract and higher-order rewriting.* PhD thesis, Vrije Universiteit Amsterdam, NL, 1994.

22 J.-H. Wu. Checking the type safety of rewrite rules in the $\lambda\pi$-calculus modulo rewriting. `https://hal.inria.fr/hal-02288720`, 2019. Internship report.

Refining Constructive Hybrid Games

Rose Bohrer
Carnegie Mellon University, Pittsburgh, PA, USA
rose.bohrer.cs@gmail.com

André Platzer
Carnegie Mellon University, Pittsburgh, PA, USA
Technische Universität München, Germany
`http://symbolaris.com/`
aplatzer@cs.cmu.edu

Abstract

We extend the constructive differential game logic (CdGL) of hybrid games with a refinement connective that relates two hybrid games. In addition to CdGL's ability to prove the existence of winning strategies for specific postconditions of hybrid games, game refinements relate two games to one another. That makes it possible to prove that *any* winning strategy for *any* postcondition of one game carries over to a winning strategy for the other. Since CdGL is constructive, a computable winning strategy can be extracted from a proof that a player wins a game. A folk theorem says that any such winning strategy for a hybrid game gives rise to a corresponding hybrid system satisfying the same property. We make this precise using CdGL's game refinements and prove correct the construction of hybrid systems from winning strategies of hybrid games.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Type theory

Keywords and phrases Hybrid Games, Constructive Logic, Refinement, Game Logic

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.14

Related Version A companion report [12] with proofs is available at `https://arxiv.org/abs/2002.02576`.

Funding This research was sponsored by the AFOSR under grant number FA9550-16-1-0288.
Rose Bohrer: The first author was also funded by the NDSEG Fellowship.

Acknowledgements We thank Jon Sterling for suggestions regarding our choice of type theory and for references to the literature. We thank the anonymous reviewers for their helpful feedback.

1 Introduction

Cyber-physical systems (CPSs) such as transportation systems, medical devices, and power systems are often modeled with hybrid systems and hybrid games. Hybrid systems combine discrete computation with continuous differential equations (ODEs), to which hybrid games add adversarial dynamics. *Differential Game Logic* (dGL) [44] and its systems fragment dL [46] provide formal proofs of correctness properties such as safety and liveness for hybrid games and systems. Theorems of dGL answer: does a winning strategy exist for a given player to achieve a given postcondition in a given game? Because safety-critical CPSs must remain reliable even in adversarial environments, these rigorous correctness guarantees for adversarial models are essential. Despite the importance of games, verification and synthesis technology for hybrid games are less mature than for hybrid systems. For example, the end-to-end verified monitor synthesizer VeriPhy [13] only supports systems.

This paper studies the gap between hybrid games and systems and proposes a reduction, which in principle enables hybrid game synthesis via existing hybrid system tools. To study this gap, we use a program refinement calculus, which is the fundamental tool for comparing programs. Refinement and equivalence reasoning have repeatedly proven fruitful for programs generally and for CPS models specifically:

- Equivalences of programs are the fundamental building block for KAT [34].
- Differential refinement logic dR$\mathcal{L}$ [36] has reduced the human labor required for verification of classical hybrid systems by relating one hybrid system to another.
- Differential game refinements in dGL [45] provide a relational reasoning technique for *differential* games (as opposed to hybrid games).

Because we are motivated by code synthesis, we develop our refinement calculus for Constructive Differential Game Logic (CdGL) [11], which ensures winning strategies are *computable*. Having defined refinements for CdGL, we bridge the gap between hybrid games and systems by defining an operation we call *reification*. The reification operation takes as its input a hybrid game α, its correctness condition ϕ, and a CdGL proof that α satisfies ϕ. The reified output is a hybrid *system* which implements the strategy expressed in the correctness proof. Refinement allows us to prove the relationship between a game and its reified system: the output system refines the input game so that every safety theorem of the output is a theorem of the input. Conversely, the output also satisfies ϕ. To our knowledge, prior works [5] only feature ad-hoc discussions of reification; we give the first rigorous algorithm and correctness theorems. It may be surprising that game strategies can be reduced to systems, because games are known [44] to be more expressive than systems. Our result does not contradict this fact: only once a winning strategy is known can we bridge this expressiveness gap. Reification makes several practical applications possible:

i) End-to-end correctness for hybrid game synthesis could be implemented by reifying winning strategies as a preprocessing step to VeriPhy [13].
ii) Interactive proof languages are better understood for systems than for games. A proof language for CdGL could be developed which combines systems reasoning with refinement. Experience with dR$\mathcal{L}$ [36] suggests refinement-based proof may be more productive.
iii) Reification and refinement give an intensional view of strategy equality: two strategies are “the same” if their reification produces equivalent systems.
iv) Refinement may enable comparing the efficacy of two controllers: does one controller always achieve its goal faster?

In Section 2, we discuss additional related work. In Section 3, we recall the syntax of CdGL, demonstrate the syntax with a toy example, and add a refinement connective. In Section 4, we recall the semantics of CdGL, generalizing them to support refinement. In Section 5, we give a calculus for CdGL refinements. In Section 6, we discuss theoretical results about soundness and reification. The paper concludes with Section 7.

2 Related Work

Our most closely related works are refinement logics. Other related works include constructive modal logics, synthesis, and games in logic.

Refinement

Refinement calculi have been studied extensively, and previous studies [5] have given examples of how programs can be refined from games, but have not given an explicit reification algorithm let alone its correctness theorems. We give an explicit *reification* algorithm and prove that it captures the winning strategy of a game in a system. Our proofs are not formalized because they rely on features which are unsupported or experimental in prominent proof assistants such as Coq. Specifically, we construct inductive families whose well-foundedness requires inductive proof and we make significant use of universe polymorphism.

We build directly on classical refinement reasoning for hybrid systems from $\mathsf{dR}\mathcal{L}$ [36] and also on (propositional discrete) game algebra [28]. Game algebra equivalences are not contextual because contextual reasoning is uniquely challenging for games, which are subnormal. $\mathsf{dR}\mathcal{L}$ supports contextual reasoning but not games. Our refinement calculus subsumes both game algebra and $\mathsf{dR}\mathcal{L}$ by mixing games rules with contextual rules for systems. Even for rules which look the same as prior work, our constructive semantics demand novel soundness proofs. Event-B [2] uses refinement for practical verification by verifying simpler models, then refining them to more complex models. Extensions of Event-B have been proposed for hybrid systems [6] but not hybrid games.

Games in Logic

Propositional GL was introduced by Parikh [41]. The first-order GL of hybrid games is dGL [44]. We build on CdGL [10, 11], the constructive dialect of dGL. GL formulas have been reduced to μ-calculus [41, 33] and game algebras have been reduced [28] to propositional modal logic. In contrast, we translate game logic *proofs*, which lets us translate CdGL into a *less expressive* logical fragment.

GLs are unique in their clear delegation of strategy to the *proof* language rather than the *model* language, allowing succinct, trustworthy game specifications with sophisticated winning strategies. Relatives without this separation of concerns include Constructive Concurrent Dynamic Logic [59], SL [16], ATL [4], CATL [55], SDGL [27], structured strategies [48], DEL [52, 54, 51], evidence logic [53], and Angelic Hoare Logic [38].

Completeness of game logics is a notoriously difficult problem, which has recently been shown in the propositional case [22]. dGL is undecidable, but is *relatively* complete [44]. Game logics can be expressed in game refinement logics, so game refinement completeness is at least as difficult. Game algebra is complete for games containing only choices, sequencing, and duality [28]. This paper does not pursue a completeness theorem, but we subsume game algebra, incorporate dGL-like rules, and exercise a broad range of refinement reasoning in our theorems on reification. These facts bode well for the expressiveness of our calculus. As with other refinement calculi, we expect that limitations arise when reasoning with ghost variables or reasoning about two games whose structures are entirely different.

Constructive Modal Logics

The task of assigning a semantics to games should not be confused with game semantics [1], which give a semantics to programs *in terms of* games. The main semantic approaches for constructive modal logics are intuitionistic Kripke semantics [58] and realizability semantics [56, 35]. We follow the type theoretic semantics which were introduced for CdGL [11]. A related approach to our type-theoretic semantics is Hoare Type Theory [40], which provides a type-theoretic connective for Hoare triples, but does not consider games or refinement.

Constructive (modal) program logics are less studied than classical ones. A few authors [10, 31] develop a Curry-Howard correspondence with proof terms, the latter for a simple fragment of dynamic logic. Other works [59, 20, 15] address only fragments and do not explore Curry-Howard in the same depth. In contrast to these, we support constructive refinement, which is also of interest for constructive program logics generally. We do not discuss proof terms here for the sake of space. Our treatment of constructive real arithmetic follows CdGL, which follows Bishop [8, 14] using constructive formalizations [19, 37].

Hybrid Systems Synthesis

Synthesis for hybrid systems is an active research area. Fully automated synthesis relies on restrictions such as simple fragments [32, 49] or discrete abstractions [25, 24]. ModelPlex [39] exploits interactive safety proofs in dL [46], the systems fragment of dGL, for monitor synthesis. Not only can proof-based synthesis synthesize *every* provable model, but it gives the user more control: to generate a less restrictive monitor, simply revise the proof to use less restrictive assumptions. ModelPlex supports an especially rigorous end-to-end verification approach [13]. We aim to provide a reduction through which ModelPlex could support games. Synthesis of high-level plans is also studied [7, 23].

3 Constructive Differential Game Logic

We recall the language of CdGL [11], introduce refinement formulas, and give an example.

3.1 Syntax

The language of CdGL consists of terms f, g, games α, β, and formulas ϕ, ψ, φ. Games are perfect-information, zero-sum, and with two players. Take note of our terminology for players, which is particularly subtle for constructive games. We use the name *Angel* for the player whose choices are quantified existentially ("us") and *Demon* for the player whose choices are quantified universally ("them"). The players alternate turns, and at any moment one player is *active* (making decisions) while the opponent is *dormant* (waiting for their turn). In an unfortunate subtlety, a formula, proof, refinement, etc. is called Angelic whenever it is existential and Demonic whenever it is universal, regardless of which player is active. The simplest terms are (game) *variables* $x, y \in \mathcal{V}$ where $\mathcal{V}$ is the set of variable identifiers. All variables are mutable and globally scoped. Their values correspond to the state of the game. For every base game variable x there is a primed counterpart x' whose purpose within an ODE is to track the time-derivative of x. The state consists of reals, which are uncountable, but the value of a term is computable as a function of the state.

▶ **Definition 1** (Terms). *We define scalar terms f, g inductively, where $c \in \mathbb{R}$ is a real literal, x a game variable, $f + g$ a sum, and $f \cdot g$ a product:*

$$f, g \ ::= \ \cdots \mid c \mid x \mid f + g \mid f \cdot g \mid (f)'$$

In a practical implementation within a theorem prover, one might prefer to reuse terms from the metalogic. It suffices that the interpretation of every term is a (Type-2 [57]) computable function. Type-2 computability means the interpretation of f must be computable to arbitrary precision when the values of variables are represented as streams of bits. We occasionally use terms which return tuples of reals, which are computable when every component is computable. The total spatial differential of term f is written $(f)'$ and agrees with the time derivative of f during an ODE.

Because CdGL is constructive, strategies must represent Angel's choices computably. While Demon is playing, Angel simply monitors whether Demon's choices obey the rules of the game, and does not care whether choices were computable. We informally discuss how a game is played here, then give full winning conditions in Section 4. The definitions of games and formulas are simultaneously inductive.

▶ **Definition 2** (Games). *The language of* games α, β *is defined recursively as such:*

$$\alpha, \beta \ ::= \ ?\phi \mid x := f \mid x := * \mid x' = f \,\&\, \psi \mid \alpha \cup \beta \mid \alpha ; \beta \mid \alpha^* \mid \alpha^d$$

The *test game* $?\phi$, is a no-op if the active player can present a proof of ϕ, else the dormant player wins by default since the active player "broke the rules". A deterministic assignment $x := f$ updates variable x to the value of term f. Nondeterministic assignments $x := *$ ask the active player to compute the new value of $x : \mathbb{R}$, i.e. they are witnessed by a term that computes a new value of x. The ODE game $x' = f \,\&\, \psi$ evolves the ODE $x' = f$ for some duration $d \geq 0$ chosen by the active player such that the active player proves ψ throughout.

All terms f are effectively locally Lipschitz continuous, meaning that a neighborhood N and real L for each state can be constructed such that L is a Lipschitz constant of f on N. Effective local Lipschitz continuity ensures that constructive Picard-Lindelöf [37] can construct the unique solution of each ODE, which need not have a closed form. ODEs are explicit-form, meaning that f and ψ do not mention any primed variables y'. Except when otherwise stated, we present ODEs with a single equation $x' = f$ for the sake of readability. In the choice game $\alpha \cup \beta$, the active player chooses whether to play game α or game β. In the sequential game $\alpha; \beta$, game α is played first, then β from the resulting state (unless a player broke the rules during α). In the repetition game α^*, the active player chooses after each repetition of α whether to continue playing, but repetitions must be well-founded and thus terminating. The exact number of iterations does not need to be computed in advance but can depend on the opponent's moves. The dual game α^d plays α with the active and dormant roles reversed. We parenthesize games with braces $\{\alpha\}$ when necessary.

▶ **Definition 3** (CdGL Formulas)**.** *The language of CdGL* formulas ϕ, ψ, φ *is given recursively by the following grammar, where* $\sim \in \{\leq, <, =, \neq, >, \geq\}$ *are comparison predicates:*

$$\phi \ ::= \ \langle\alpha\rangle\phi \mid [\alpha]\phi \mid f \sim g \mid \alpha \leq^{i}_{[]} \beta$$

Modalities $\langle\alpha\rangle\phi$ and $[\alpha]\phi$ say Angel wins α with postcondition ϕ, starting as the active or dormant player respectively. Modality $\langle\alpha\rangle\phi$ is Angelic in the sense that decisions are resolved Angelically: Angel is the one currently making choices. Modality $[\alpha]\phi$ is Demonic in the sense that decisions are resolved Demonically: Angel has no control until a dual operator is encountered. We will deal mainly in box modalities $[\alpha]\phi$, with *Angel*'s moves appearing inside dualities α^d and *Demon*'s moves outside dualities.

To define refinements, we introduce the rank $\mathfrak{R}(\alpha \text{ or } \phi)$ of a game or formula, a technical device which represents the smallest predicative universe in which α has a semantics, see Section 4. Game refinements come in two standard [28] kinds: Angelic and Demonic. *Demonic refinement* $\alpha \leq^{i}_{[]} \beta$ of *rank* i holds if for every ϕ with $\mathfrak{R}(\phi) \leq i$, dormant winning strategies of $[\alpha]\phi$ can be mapped constructively into winning strategies of $[\beta]\phi$. *Angelic refinement* $\alpha \leq^{i}_{\langle\rangle} \beta$ maps active winning strategies of $\langle\alpha\rangle\phi$ constructively into winning strategies of $\langle\beta\rangle\phi$. Note this difference carefully: Angelic refinement may be more familiar to the reader, but we take the Demonic presentation as primary, in large part because the theorems we wish to prove are Demonic. Angelic and Demonic refinement are interdefinable: $\alpha \leq^{i}_{\langle\rangle} \beta \leftrightarrow \alpha^d \leq^{i}_{[]} \beta^d$ and vice versa. You may wish to ignore rank on the first reading: it can be inferred automatically, and we write $\alpha \leq_{[]} \beta$ when rank is unimportant.

The standard connectives of first-order constructive logic are definable from games and comparisons. Verum ($\mathtt{tt}$) is defined $1 > 0$ and falsum ($\mathtt{ff}$) is $0 > 1$. Conjunction $\phi \wedge \psi$ is defined $\langle ?\phi\rangle\psi$, disjunction $\phi \vee \psi$ is $\langle ?\phi \cup ?\psi\rangle\mathtt{tt}$, implication $\phi \to \psi$ is $[?\phi]\psi$, universal quantification $\forall x\, \phi$ is defined $[x := *]\phi$, and existential quantification $\exists x\, \phi$ is $\langle x := *\rangle\phi$. Equivalence $\phi \leftrightarrow \psi$ is $(\phi \to \psi) \wedge (\psi \to \phi)$. As usual in constructive logics, negation $\neg\phi$ is defined $\phi \to \mathtt{ff}$, and inequality is defined by $f \neq g \equiv \neg(f = g)$. The defined game skip is the trivial test $?\mathtt{tt}$. While these constructs are derivable, and thus it suffices to provide semantics and proof rules for the core constructs, we find it useful to consider the core and derived forms syntactically distinct. It will also aid in understanding of the semantics to keep the definitions above in mind, because the semantics for many first-order programs mirror those from their counterpart in first-order constructive logic.

3.2 Example Game

As a simple example, consider a *push-pull cart* [46] on a 1 dimensional playing field with boundaries $x_l \leq x \leq x_r$ where x is the position of the cart and $x_l < x_r$ strictly. The initial position is written x_0. These preconditions are in formula pre. Demon is at the left of the cart and Angel at its right. Each player chooses to pull or push the cart, then the (oversimplified) physics say velocity is proportional to the sum of forces. Physics can evolve so long as the boundary $x_l \leq x \leq x_r$ is respected, with duration chosen by Demon.

$$\mathsf{pre} \equiv x_l < x_r \land x_l \leq x_0 = x \leq x_r$$
$$\mathsf{PP} \equiv \{\{L := -1 \cup L := 1\}; \{R := -1 \cup R := 1\}^d; \{x' = L + R \,\&\, x_l \leq x \leq x_r\}\}^*$$

A simple safety theorem for the push-pull game says that Angel has a strategy to ensure position x remains constant ($x = x_0$) no matter how Demon plays:

$$\mathsf{pre} \rightarrow [\mathsf{PP}]x = x_0 \tag{1}$$

The winning strategy that proves (1) is a simple mirroring strategy: Angel observes Demon's choice of L and plays the opposite value of R so that $L + R = 0$. Because $L + R = 0$, the ODE simplifies to $x' = 0 \,\&\, x_l \leq x \leq x_r$, which has the trivial solution $x(t) = x(0)$ for all times $t \in \mathbb{R}_{\geq 0}$. Angel shows the safety theorem by replacing the ODE with its solution and observing that $x = x_0$ holds for all possible durations.

In addition to solution reasoning, CdGL supports *differential invariant* [46] reasoning which appeals to the derivative of a term and *differential ghost* [46] reasoning which augments an ODE with a new continuous variable. Solution reasoning suffices for this toy example, but invariant reasoning is essential for CdGL games whose ODEs have non-polynomial, even non-elementary solutions. Ghost reasoning allows proving differential invariants which are not inductive [47], which cannot be proved otherwise [43]. For these reasons, our proof calculus (Section 5) includes solution, invariant, and ghost rules.

In contrast to a safety theorem, a liveness theorem would be shown by a progress argument. Suppose that Angel could set $L = 2$ but Demon can only choose $R \in \{-1, 1\}$. Then Angel's liveness theorem might say she can achieve $x = x_r$ if she as allowed to choose ODE duration, because the choice $L = 2$ ensures at least 1 unit of progress in x for each unit of time.

4 Type-theoretic Semantics

We generalize the type-theoretic semantics of CdGL [11]. We define the semantics of the new refinement formulas $\alpha \leq^i_{[]} \beta$ and employ an infinite tower of type universes, in support of refinements. We first give our assumptions on the underlying type theory.

4.1 Type Theory Assumptions

We assume a Calculus of Inductive and Coinductive Constructions (CIC)-like type theory [17, 18, 50] with dependency and an infinite tower of cumulative predicative universes. Predicativity is essential because our semantics are a large elimination, which would interact dangerously with impredicative quantification. We assume first-class anonymous constructors for (indexed [21]) inductive and coinductive types. We write M, N for type-theoretic terms, τ for type families, and κ for kinds (those type families inhabited by other type families).

We write $\Pi x : \tau_1.\, \tau_2$ for a dependent function type with argument named x of type τ_1 where return type τ_2 may mention x. We write $\Sigma x : \tau_1.\, \tau_2$ for a dependent pair type with left component named x of type τ_1 and right component of type τ_2, possibly mentioning x. These specialize to the simple types $\tau_1 \Rightarrow \tau_2$ and $\tau_1 * \tau_2$ respectively when x is not mentioned in τ_2. Lambdas $(\lambda x : \tau.\ M)$ inhabit dependent function types. Pairs (M, N) inhabit dependent pair types. Let-binding unpacks pairs and $\pi_L M$ and $\pi_R M$ are left and right projection. We write $\tau_1 + \tau_2$ for disjoint unions inhabited by $\ell \cdot M$ and $r \cdot M$, and write $\mathsf{case}\ A\ \mathsf{of}\ \ell \Rightarrow B \mid r \Rightarrow C$ for case analysis for $\tau_1 + \tau_2$, where ℓ and r are variables over proofs.

We assume a type $\mathbb{R}$ for real numbers and type $\mathfrak{S}$ for Euclidean state vectors supporting scalar and vector sums, products, scalar inverses, and units. States $s, \hat{s} : \mathfrak{S}$ assign values to every variable $x \in \mathcal{V}$ and support the operations $s\ x$ for *retrieving* the value of x and $\mathsf{set}\ s\ x\ v$ for *updating* the value of x to v. Likewise, $\mathsf{set}\ s\ (x, y)\ (v, w)$ sets both x and y to v and w, respectively. The usual axioms of setters and getters [26] are satisfied.

We write $\mathbb{T}_i$ for the i'th predicative universe. We use $P, Q : \mathfrak{S} \Rightarrow \mathbb{T}_i$ for variables over *regions*, e.g., the interpretations of formulas. Inductive type families are written $\mu P : \kappa.\, \tau$, which denotes the *smallest* solution $\mathtt{ty}$ of kind κ to the fixed-point equation $\mathtt{ty} = [\mathtt{ty}/P]\tau$. Coinductive type families are written $\rho P : \kappa.\, \tau$, which denotes the *largest* solution $\mathtt{ty}$ of kind κ to the fixed-point equation $\mathtt{ty} = [\mathtt{ty}/P]\tau$. The type expression τ must be monotone in P to ensure that smallest and largest solutions exist, per Knaster-Tarski [29, Thm. 1.12]. Monotonicity of τ will require inductive proof in our case.

4.2 Semantics of CdGL

The interpretation of terms f, g as functions of type $\mathfrak{S} \Rightarrow \mathbb{R}$ is standard. Games α and formulas ϕ require a notion of *rank* $\mathfrak{R}(\alpha \text{ or } \phi)$ indicating the smallest universe where α or ϕ has semantics. Universes are cumulative, so the semantics also belong to all universes $\mathbb{T}_i$ such that $i \geq \mathfrak{R}(\alpha)$. Refinement quantifies over types of a lower universe, which is predicative. A refinement formula's rank is given by its annotation: $\mathfrak{R}(\alpha \leq^i_{[]} \beta) = 1 + i$, requiring $\mathfrak{R}(\alpha), \mathfrak{R}(\beta) \leq i$. In all other cases, the rank is the maximum of ranks of subexpressions.

Formulas ϕ are interpreted as predicates over states, i.e., type families $\ulcorner \phi \urcorner : \mathfrak{S} \Rightarrow \mathbb{T}_{\mathfrak{R}(\phi)}$. We say the formula ϕ is *valid* if there exists a CIC term $M : (\Pi s : \mathfrak{S}.\, \ulcorner \phi \urcorner\ s)$. CIC term M is allowed to inspect state s, but only using computable operations. A natural deduction sequent $(\Gamma \vdash \phi)$ is valid iff implication formula $\bigwedge \Gamma \rightarrow \phi$ with conjunction $\bigwedge \Gamma$ is valid. The formula semantics are defined in terms of the active and dormant semantics of games, which determine how Angel wins a game α whose postcondition is a formula ϕ whose semantics are the goal region $\ulcorner \phi \urcorner$ (variable P in Definition 5). We write $\langle\langle \alpha \rangle\rangle : (\mathfrak{S} \Rightarrow \mathbb{T}_{\mathfrak{R}(\alpha)}) \Rightarrow (\mathfrak{S} \Rightarrow \mathbb{T}_{\mathfrak{R}(\alpha)})$ for the active semantics of α and $[[\alpha]] : (\mathfrak{S} \Rightarrow \mathbb{T}_{\mathfrak{R}(\alpha)}) \Rightarrow (\mathfrak{S} \Rightarrow \mathbb{T}_{\mathfrak{R}(\alpha)})$ for its dormant semantics, which capture Angel's winning strategies when Angel is active or dormant, respectively. In contrast to classical game logics, the diamond and box modalities are *not* interdefinable constructively. The rank of an expression is only relevant in the refinement cases.

▶ **Definition 4** (Formula semantics). *Interpretation* $\ulcorner \phi \urcorner : \mathfrak{S} \Rightarrow \mathbb{T}_{\mathfrak{R}(\phi)}$ *is defined by*

$$\ulcorner [\alpha]\phi \urcorner\ s = [[\alpha]]\ \ulcorner \phi \urcorner\ s \qquad \ulcorner \langle\alpha\rangle\phi \urcorner\ s = \langle\langle \alpha \rangle\rangle\ \ulcorner \phi \urcorner\ s \qquad \ulcorner f \sim g \urcorner\ s = \big((f\ s) \sim (g\ s)\big)$$

$$\ulcorner \alpha \leq^i_{[]} \beta \urcorner\ s = \big(\Pi P : (\mathfrak{S} \Rightarrow \mathbb{T}_i).\ \big([[\alpha]]\ P\ s \Rightarrow [[\beta]]\ P\ s\big)\big)$$

The modality $\langle\alpha\rangle\phi$ is true in state s when active Angel has a strategy $\langle\langle \alpha \rangle\rangle\ \ulcorner \phi \urcorner\ s$ for game α from state s to reach the region $\ulcorner \phi \urcorner$ on which ϕ has a proof. The modality $[\alpha]\phi$ is true in state s when dormant Angel has a strategy $[[\alpha]]$ for game α from state s to reach the region $\ulcorner \phi \urcorner$ on which ϕ has a proof. For comparison operators $\sim \in \{\leq, <, =, \neq, >, \geq\}$, the values of f and

g are compared at state s. Game α demonically refines β ($\alpha \leq_{[]} \beta$) from a state s if for *all* goal regions P there exists an effective mapping from dormant strategies $[[\alpha]]\ P\ s$ to dormant strategies $[[\beta]]\ P\ s$. The (defined) meaning of angelic refinement $\alpha \leq_{\langle\rangle} \beta$ is symmetric using diamond semantics $\langle\langle\alpha\rangle\rangle$. That is, refinements may depend on the state (they are *local* or *contextual*), but must hold for all goal regions P, as refinements consider the general game form itself, not a game fixed to a particular postcondition. Because refinement formulas are first-class, quantifiers may appear nested and in arbitrary positions, not necessarily prenex form. We ensure predicativity by requiring that refinements quantify only over postconditions of lower rank. Rank can be inferred in practice by inspecting a proof: each rank annotation need only be as large as the rank of every postcondition in every application of rules R$\langle\cdot\rangle$ and R$[\cdot]$ from Section 5.

The semantics of games are simultaneously inductive with those for formulas and with one another. In each case, the connectives which define $[[\alpha]]$ and $\langle\langle\alpha\rangle\rangle$ are duals, because $[\alpha]\phi$ and $\langle\alpha\rangle\phi$ are dual. Below, P is the goal region and s is the initial state.

▶ **Definition 5** (Active semantics). *The active interpretation $\langle\langle\alpha\rangle\rangle$ of the hybrid game α has kind $(\mathfrak{S} \Rightarrow \mathbb{T}_{\mathfrak{R}(\alpha)}) \Rightarrow (\mathfrak{S} \Rightarrow \mathbb{T}_{\mathfrak{R}(\alpha)})$ and is defined by*

$$\begin{aligned}
\langle\langle ?\psi \rangle\rangle\ P\ s &= \ulcorner\psi\urcorner\ s * P\ s\\
\langle\langle x := f \rangle\rangle\ P\ s &= P\ (\textsf{set}\ s\ x\ (f\ s))\\
\langle\langle x := * \rangle\rangle\ P\ s &= \Sigma v : \mathbb{R}. (P\ (\textsf{set}\ s\ x\ v))\\
\langle\langle \alpha \cup \beta \rangle\rangle\ P\ s &= \langle\langle\alpha\rangle\rangle\ P\ s\ + \langle\langle\beta\rangle\rangle\ P\ s\\
\langle\langle \alpha ; \beta \rangle\rangle\ P\ s &= \langle\langle\alpha\rangle\rangle\ (\langle\langle\beta\rangle\rangle\ P)\ s
\end{aligned}$$

$$\begin{aligned}
\langle\langle \alpha^d \rangle\rangle\ P\ s &= [[\alpha]]\ P\ s\\
\langle\langle x' = f \,\&\, \psi \rangle\rangle\ P\ s &= \Sigma d : \mathbb{R}_{\geq 0}. \Sigma sol : ([0, d] \Rightarrow \mathbb{R}).\\
&\quad (sol, s, d \vDash x' = f)\\
&\quad *(\Pi t : [0, d]. \ulcorner\psi\urcorner\ (\textsf{set}\ s\ x\ (sol\ t)))\\
&\quad *P\ \big(\textsf{set}\ s\ (x, x')\\
&\qquad (sol\ d, f\ (\textsf{set}\ s\ x\ (sol\ d)))\big)
\end{aligned}$$

$$\langle\langle \alpha^* \rangle\rangle\ P\ s = \big(\mu Q : (\mathfrak{S} \Rightarrow \mathbb{T}_{\mathfrak{R}(\alpha)}). \lambda\hat{s} : \mathfrak{S}.\ (P\ \hat{s} \Rightarrow Q\ \hat{s})\ +(\langle\langle\alpha\rangle\rangle\ Q\ \hat{s} \Rightarrow Q\ \hat{s})\big)\ s$$

Angel wins $?\psi$ by proving both ψ and P at s. Angel wins the deterministic assignment $x := f$ by executing it, then proving P. Angel wins nondeterministic assignment $x := *$ by choosing a new value v, then proving P. Angel wins $\alpha \cup \beta$ by choosing to play game α or β, then winning it. Angel wins $\alpha; \beta$ by winning α with the postcondition of winning β. Angel wins α^d if she wins α in the dormant role. Angel wins ODE game $x' = f \,\&\, \psi$ by choosing some solution y of some duration d for which she proves domain constraint ψ throughout and the goal region P at time d. While top-level postconditions rarely mention x', intermediate proof steps do, thus x and x' are both updated in the postcondition. The construct $(sol, s, d \vDash x' = f)$ says sol solves $x' = f$ from state s for time d [12, App. A]. Active Angel strategies for α^* are inductively defined: either stop the loop and prove P now, else play a round of α and repeat inductively. By Knaster-Tarski [29, Thm. 1.12], this least fixed point exists since games' semantics are monotone in the postcondition [11, Lem. 7].

▶ **Definition 6** (Dormant semantics). *The dormant interpretation $[[\alpha]]$ of the hybrid game α has kind $(\mathfrak{S} \Rightarrow \mathbb{T}_{\mathfrak{R}(\alpha)}) \Rightarrow (\mathfrak{S} \Rightarrow \mathbb{T}_{\mathfrak{R}(\alpha)})$ and is defined by*

$$\begin{aligned}
[[?\psi]]\ P\ s &= \ulcorner\psi\urcorner\ s \Rightarrow P\ s\\
[[x := f]]\ P\ s &= P\ (\textsf{set}\ s\ x\ (f\ s))\\
[[x := *]]\ P\ s &= \Pi v : \mathbb{R}. (P\ (\textsf{set}\ s\ x\ v))\\
[[\alpha \cup \beta]]\ P\ s &= [[\alpha]]\ P\ s\ * [[\beta]]\ P\ s\\
[[\alpha ; \beta]]\ P\ s &= [[\alpha]]\ ([[\beta]]\ P)\ s
\end{aligned}$$

$$\begin{aligned}
[[\alpha^d]]\ P\ s &= \langle\langle\alpha\rangle\rangle\ P\ s\\
[[x' = f \,\&\, \psi]]\ P\ s &= \Pi d : \mathbb{R}_{\geq 0}. \Pi sol : ([0, d] \Rightarrow \mathbb{R}).\\
&\quad (sol, s, d \vDash x' = f)\\
&\quad \Rightarrow\big(\Pi t : [0, d]. \ulcorner\psi\urcorner\ (\textsf{set}\ s\ x\ (sol\ t))\big)\\
&\quad \Rightarrow P\ \big(\textsf{set}\ s\ (x, x')\\
&\qquad (sol\ d, f\ (\textsf{set}\ s\ x\ (sol\ d)))\big)
\end{aligned}$$

$$[[\alpha^*]]\ P\ s = \big(\rho Q : (\mathfrak{S} \Rightarrow \mathbb{T}_{\mathfrak{R}(\alpha)}). \lambda\hat{s} : \mathfrak{S}.\ (Q\ \hat{s} \Rightarrow [[\alpha]]\ Q\ \hat{s})\ *(Q\ \hat{s} \Rightarrow P\ \hat{s})\big)\ s$$

Angel wins $?\psi$ by proving P under assumption ψ, which Demon must provide. Deterministic assignment is unchanged. Angel wins $x := *$ by proving P for *every* choice of x. Angel wins $\alpha \cup \beta$ with a pair of winning strategies, since Demon chooses whether to play α or β. Angel wins $\alpha; \beta$ by winning α with a postcondition of winning β. Angel wins α^d if she can win α actively. Angel wins $x' = f \,\&\, \psi$ if for an arbitrary duration and arbitrary solution which satisfy the domain constraint, Angel can prove the postcondition. Dormant repetition strategies are coinductive using some invariant region Q. When Demon decides to stop the loop, Angel responds by proving P from Q. Whenever Demon chooses to continue, Angel proves that Q is preserved. Greatest fixed points exist by Knaster-Tarski [29, Thm. 1.12] using monotonicity [11, Lem. 7].

In general, Angel strategies are constructive but permit Demon to play classically. In the cyber-physical setting, Demon is indeed rarely a computer.

5 Refinement Proof Calculus

We give a natural deduction calculus for hybrid game refinements. Refinement is relative to a context Γ of CdGL formulas, which may include refinements. All rules are expressed as Demonic refinements $\alpha \leq^i_{[]} \beta$, but an Angelic refinement $\alpha \leq^i_{\langle\rangle} \beta$ is supported by refining the duals $\alpha^d \leq^i_{[]} \beta^d$. Remember that in a Demonic refinement, the Angelic (existential) connectives appear under dualities α^d. We write $\alpha \cong \beta$ for $\alpha \leq_{[]} \beta \land \beta \leq_{[]} \alpha$. Recall that hybrid *systems* are hybrid games which do not contain the dual operator α^d.

The refinement elimination rules $\text{R}\langle\cdot\rangle$ and $\text{R}[\cdot]$ say every true postcondition ϕ of a game α is a true postcondition of every β which α refines. The side condition for $\text{R}\langle\cdot\rangle$ and $\text{R}[\cdot]$ is that $\mathfrak{R}(\phi) \leq i$ where i is the rank annotation of the refinement. These are the only rules which care about rank, so ranks can be inferred from proofs by inspecting the uses of these rules. While rank is of little practical import, it ensures a predicative formal foundation.

Figure 1 gives the refinement rules for discrete connectives. Soundness of game refinement rules is subtle because games are subnormal. Sound game refinement rules can be divided into two classes: either refine games globally by requiring an empty context (;G), or restrict some subgames to be systems (;S). The formal approach follows game algebra [28] and is required when comparing two games. The latter approach generalizes dR$\mathcal{L}$ [36] and is necessary when reifying sequential games. By combining both approaches, our calculus is strictly more complete than both game algebra and dR$\mathcal{L}$. Unlike dR$\mathcal{L}$ [36], we face the challenge that game logics are subnormal and subregular [30]: For a game α, formula $[\alpha](\phi \land \psi)$ need not hold when both $[\alpha]\phi$ and $[\alpha]\psi$ do.

Bold variables only range over systems, e.g., $\boldsymbol{\alpha}$ and $\boldsymbol{\alpha}_1$ in rules ;S and un$*$. One sequence refines another piecewise in the ;S rule, which is *contextual*: refinement of the second component exploits the fact that the first component has been executed. Rule ;G is a variant of ;S which says α_1 can be an arbitrary game, but only if $\beta_1 \leq_{[]} \beta_2$ holds in the empty context. System $\boldsymbol{\alpha}_1$ in the second premiss of ;S could soundly be α_2, but in practical proofs it is often more convenient to work with $[\boldsymbol{\alpha}_1]$ because it is a *system* modality, which is normal. Rules $\langle?\rangle$ and $[?]$ refine tests by weakening or strengthening test conditions. The left and right rules for choices are dual. Rules $\langle\cup\rangle$R1 and $\langle\cup\rangle$R2 say each branch refines an Angelic choice, while $[\cup]$R says a Demonic choice is refined by refining both branches. Rules $\langle:*\rangle$ and $[:*]$ say that deterministic assignments refine nondeterministic ones. Rule un$*$ compares loops by comparing their bodies and roll$_l$ allows unrolling a loop before refining. Rules skip^d, $:=^d$, and $;^d$ say skip and $x := f$ are self-dual and the dual of a sequence is a sequence of duals. Double duals cancel by DDE.

$$\mathrm{R}\langle\cdot\rangle\quad \frac{\Gamma \vdash \langle\alpha\rangle\phi \quad \Gamma \vdash \alpha \leq^{i}_{\langle\rangle} \beta}{\Gamma \vdash \langle\beta\rangle\phi}\ {}^{1}$$

$$;\mathrm{S}\quad \frac{\Gamma \vdash \boldsymbol{\alpha}_1 \leq_{[]} \alpha_2 \quad \Gamma \vdash [\boldsymbol{\alpha}_1]\beta_1 \leq_{[]} \beta_2}{\Gamma \vdash \boldsymbol{\alpha}_1; \beta_1 \leq_{[]} \alpha_2; \beta_2}\ {}^{2}$$

$$;\mathrm{G}\quad \frac{\Gamma \vdash \alpha_1 \leq_{[]} \alpha_2 \quad \cdot \vdash \beta_1 \leq_{[]} \beta_2}{\Gamma \vdash \alpha_1; \beta_1 \leq_{[]} \alpha_2; \beta_2}$$

$$[\cup]\mathrm{L1}\quad \Gamma \vdash \alpha \cup \beta \leq_{[]} \alpha$$

$$[\cup]\mathrm{L2}\quad \Gamma \vdash \alpha \cup \beta \leq_{[]} \beta$$

$$[\cup]\mathrm{R}\quad \frac{\Gamma \vdash \alpha \leq_{[]} \beta \quad \Gamma \vdash \alpha \leq_{[]} \gamma}{\Gamma \vdash \alpha \leq_{[]} \beta \cup \gamma}$$

$$\langle{:}{*}\rangle\quad \Gamma \vdash x := f^d \leq_{[]} x := *^d$$

$$[{:}{*}]\quad \Gamma \vdash x := * \leq_{[]} x := f$$

$$\mathrm{R}[\cdot]\quad \frac{\Gamma \vdash [\alpha]\phi \quad \Gamma \vdash \alpha \leq^{i}_{[]} \beta}{\Gamma \vdash [\beta]\phi}\ {}^{2}$$

$$\langle ? \rangle\quad \frac{\Gamma \vdash \phi \to \psi}{\Gamma \vdash ?\phi^d \leq_{[]} ?\psi^d}$$

$$[?]\quad \frac{\Gamma \vdash \psi \to \phi}{\Gamma \vdash ?\phi \leq_{[]} ?\psi}$$

$$\langle\cup\rangle\mathrm{R1}\quad \Gamma \vdash \alpha^d \leq_{[]} \{\alpha \cup \beta\}^d$$

$$\langle\cup\rangle\mathrm{R2}\quad \Gamma \vdash \beta^d \leq_{[]} \{\alpha \cup \beta\}^d$$

$$\langle\cup\rangle\mathrm{L}\quad \frac{\Gamma \vdash \alpha^d \leq_{[]} \gamma \quad \Gamma \vdash \beta^d \leq_{[]} \gamma}{\Gamma \vdash \{\alpha \cup \beta\}^d \leq_{[]} \gamma}$$

$$\mathrm{un}{*}\quad \frac{\Gamma \vdash [\boldsymbol{\alpha}^*](\boldsymbol{\alpha} \leq_{[]} \beta)}{\Gamma \vdash \boldsymbol{\alpha}^* \leq_{[]} \beta^*}\ {}^{2}$$

$$\mathrm{roll}_l\quad \Gamma \vdash \textsf{skip} \cup \{\alpha; \alpha^*\} \cong \alpha^*$$

$$\mathrm{skip}^d\quad \textsf{skip}^d \cong \textsf{skip} \qquad ;^d\quad \{\alpha;\beta\}^d \cong \alpha^d;\beta^d \qquad :=^d\quad x := f^d \cong x := f \qquad \mathrm{DDE}\quad \{\alpha^d\}^d \cong \alpha$$

[1] assuming $\mathfrak{R}(\phi) \leq i$

[2] $\boldsymbol{\alpha}_1$ respectively $\boldsymbol{\alpha}$ is a hybrid system

Figure 1 Refinement of discrete connectives.

$$\mathrm{trans}\quad \frac{\Gamma \vdash \alpha \leq_{[]} \beta \quad \Gamma \vdash \beta \leq_{[]} \gamma}{\Gamma \vdash \alpha \leq_{[]} \gamma}$$

$$\mathrm{refl}\quad \Gamma \vdash \alpha \leq_{[]} \alpha$$

$$;\mathrm{id}_l\quad \Gamma \vdash \{\textsf{skip}; \alpha\} \cong \alpha$$

$$;\mathrm{id}_r\quad \Gamma \vdash \{\alpha; \textsf{skip}\} \cong \alpha$$

$$\mathrm{annih}_l\quad \Gamma \vdash ?\mathtt{ff}; \alpha \cong ?\mathtt{ff}$$

$$:=\mathrm{nop}\quad \Gamma \vdash \{x := x\} \cong \textsf{skip}$$

$$;\mathrm{d}_r\quad \Gamma \vdash \{\alpha \cup \beta\}; \gamma \cong \{\alpha; \gamma\} \cup \{\beta; \gamma\}$$

$$;\mathrm{A}\quad \Gamma \vdash \{\alpha; \beta\}; \gamma \cong \alpha; \{\beta; \gamma\}$$

$$:=:=\quad \Gamma \vdash x := f; x := g \cong x := g\ {}^{1}$$

$$\cup\mathrm{A}\quad \Gamma \vdash \{\alpha \cup \beta\} \cup \gamma \cong \alpha \cup \{\beta \cup \gamma\}$$

$$\cup\mathrm{c}\quad \Gamma \vdash \alpha \cup \beta \cong \beta \cup \alpha$$

$$\cup\mathrm{idem}\quad \Gamma \vdash \alpha \cup \alpha \cong \alpha$$

[1] for $x \notin \mathrm{FV}(g)$

Figure 2 Algebraic rules (selected).

$$\text{DC}\quad \frac{\Gamma \vdash [x' = f \,\&\, \phi]\psi}{\Gamma \vdash \{x' = f \,\&\, \phi\} \cong \{x' = f \,\&\, \phi \land \psi\}} \qquad \text{DW}\quad \Gamma \vdash \{x := *; x' := f; ?\psi\} \leq_{[\,]} \{x' = f \,\&\, \psi\}$$

$$\text{solve}\quad \frac{\Gamma \vdash [t := *; ?0 \leq t \leq d; x := sln]\psi}{\Gamma, t = 0, d \geq 0 \vdash \{t := d; x := sln; t' := 1; x' := f\} \leq_{[\,]} \{t' = 1, x' = f \,\&\, \psi\}^d}\,^1$$

$$\text{DG}\quad \Gamma \vdash \{y := f_0; x' = f, y' = a(x)y + b(x) \,\&\, \psi\} \leq_{[\,]} \{x' = f \,\&\, \psi; \{y := *; y' := *\}^d\}$$

[1] sln solves ODE, $\{t, t', x, x'\} \cap FV(d) = \emptyset$

Figure 3 Differential equation refinements.

The rules in Figure 2 are selected algebraic properties which will be used in the proof of Theorem 10. These rules generalize known game equalities [28] to refinement. Some rules of dRL [36] are reused here, but other rules of dRL, such as those for repetitions α^* are not sound for arbitrary games. Rules refl and trans say refinement is a partial order. Sequential composition has identities (;id$_l$ and ;id$_r$). Rule :=:= deduplicates a double assignment if the first assignment does not influence the second: FV(f) are the *free variables* mentioned in f. Choice (∪A) and sequence (;A) are associative, and choice is commutative (∪c) and idempotent (∪idem), while sequence is right-distributive (;d$_r$). Impossible tests can annihilate any following program (annih$_l$). Assigning a variable to itself is a no-op (:=nop).

Figure 3 gives the ODE refinement rules. *Differential cut* DC says the domain constraints ϕ and $\phi \land \psi$ are equivalent if ψ holds as a postcondition under domain constraint ϕ. *Differential weakening* DW says an ODE is overapproximated by the program which assumes only the domain constraint. *Differential solution* solve says that a solvable Angelic ODE $x' = f \,\&\, \psi$ with syntactic solution term sln is refined by a deterministic program which assigns the solution to x after through which the domain constraint holds, specified by a term d which is constant throughout the ODE. Here $sln = (\lambda s : \mathfrak{S}.\ (sol\ (s\ t)))$ is the term corresponding to the semantic solution sol at time t. *Differential ghosts* DG soundly augments an ODE with a fresh dimension y so long as the solution for y exists as long as that of x, and is known [47] to enable proofs of otherwise unprovable [43] properties. The right-hand side for y is required to be linear in y because this suffices to ensure sufficient duration. Axiom DG is not an equivalence because linear ODEs do not suffice to reach every of the nondeterministically assigned final values for y and y' [42].

6 Theory

We develop theoretical results about CdGL refinements: soundness and the relationship between games and systems. Proofs are in a companion report [12].

6.1 Soundness

The *sine qua non* condition of any logic is soundness. We show that every formula provable in the CdGL refinement calculus is true in the type-theoretic semantics.

▶ **Theorem 7** (Soundness). *If $\Gamma \vdash \phi$ is provable then the sequent $(\Gamma \vdash \phi)$ is valid.*

6.2 Reification

A game α describes what actions are allowed for each player but not how Angel selects among them given an adversarial Demon. Every game modality proof, whether of $[\alpha]\phi$ or $\langle\alpha\rangle\phi$, lets Demon make arbitrary (universally-quantified) moves within the confines of the game, and describes Angel's strategy to achieve a given postcondition ϕ. Whereas a given game can contain both Angelic and Demonic choices, a *system* can only contain one or the other: modality $[\boldsymbol{\alpha}]\phi$ treats a system $\boldsymbol{\alpha}$ as Demonic while $\langle\boldsymbol{\alpha}\rangle\phi$ treats a system as Angelic.

A folklore theorem describes the relation between hybrid games and hybrid systems: given a proof (winning strategy) for a hybrid game, one can *reify* Angel's strategy to produce a hybrid *system* which implements that strategy. The constructivity of CdGL ensures that Angel's choices are implementable by computable functions. Since Demonic choices survive reification, it is simplest to work with Demonic game modalities $[\alpha]\phi$ here, but every Angelic game modality $\langle\alpha\rangle\phi$ could equivalently be expressed as $[\alpha^d]\phi$. In this section, we formally define the reification operation and prove its relation to the source game using refinements and a derivation $\mathcal{A}$ in the CdGL (non-refinement) proof calculus. For the sake of space, we present CdGL rules informally as we define reification. Our presentation differs in insignificant ways from the full calculus from prior work [11]; it is convenient for our purposes that premisses eliminate as many connectives as possible, as is common in natural-deduction style. Let $\mathcal{A}$ be a CdGL proof of some CdGL formula $[\alpha]\phi$ in context Γ, i.e., let $\Gamma \vdash \mathcal{A} \colon [\alpha]\phi$. We then write $\mathcal{A} \rightsquigarrow \boldsymbol{\alpha}$ to say the (unique) result of reifying the strategy given by $\mathcal{A}$ into hybrid game α is the hybrid *system* $\boldsymbol{\alpha}$. The system $\boldsymbol{\alpha}$ needs to commit to Angel's strategy according to $\mathcal{A}$ while retaining all available choices of Demon. What properties ought $\boldsymbol{\alpha}$ satisfy?

Committing to a safe Angel strategy should never make the system less safe. The safety postcondition ϕ should *transfer* to $\boldsymbol{\alpha}$, i.e., the following property should hold:

If $\Gamma \vdash \mathcal{A} \colon [\alpha]\phi$ and $\mathcal{A} \rightsquigarrow \boldsymbol{\alpha}$ then $(\Gamma \vdash [\boldsymbol{\alpha}]\phi)$ is provable.

Transfer alone does not capture reification, e.g., defining $\boldsymbol{\alpha} = ?\mathtt{ff}$ for all α and $\mathcal{A}$ would vacuously satisfy the transfer property but certainly not capture the meaning of strategy $\mathcal{A}$.

We, thus, guarantee a converse direction. The reified hybrid system $\boldsymbol{\alpha}$ is a *safety refinement* of hybrid game α, so *every* postcondition ψ satisfying $[\boldsymbol{\alpha}]\psi$ also satisfies $[\alpha]\psi$:

If $\Gamma \vdash \mathcal{A} \colon [\alpha]\phi$ and $\mathcal{A} \rightsquigarrow \boldsymbol{\alpha}$ then $(\Gamma \vdash \boldsymbol{\alpha} \leq_{[\,]} \alpha)$ is provable.

Intuitively, $[\boldsymbol{\alpha}]\psi$ says postcondition ψ holds for every Demon behavior of $\boldsymbol{\alpha}$, while $[\boldsymbol{\alpha}]\psi$ holds if there *exists* an Angel strategy that ensures ψ for every Demon behavior of α. Since derivation $\mathcal{A}$ is designed to satisfy ψ, there certainly exists a strategy that satisfies ψ. Refinement captures the notion that Angelic choices in $\boldsymbol{\alpha}$ are made more strictly than in α, while Demonic choices are only made more loosely.

Even transfer *and* refinement do not fully validate the reification operation, since defining $\boldsymbol{\alpha} = \alpha$ suffices to ensure both. This leads to a third, most obvious property: $\boldsymbol{\alpha}$ must be a system when α is a game. Not only are systemhood, transfer, and refinement all desirable properties for reification, but their combination is an appealing specification because there is no trivial operation which satisfies all three. If the above three properties hold, they also imply a sound version of the normal modal logic axiom K that is elusive in games: If $\Gamma \vdash \mathcal{A} : [\alpha]\phi$ and $\Gamma \vdash [\boldsymbol{\alpha}]\psi$ is provable for $\mathcal{A} \rightsquigarrow \boldsymbol{\alpha}$, then $\Gamma \vdash [\alpha](\phi \wedge \psi)$ is provable. Additionally, transfer and systemhood suggest that game synthesis can "export" a game proof to a systems proof, for which synthesis tools already exist [39, 13]. We discuss some technicalities first.

Technicalities

Reification accepts a CdGL proof and returns a system. In each case of its inductive definition, we write $\mathcal{A}, \mathcal{B}, \mathcal{C}$ for the proofs of each premiss and $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}$ for corresponding output systems. For simplicity, we reify *nested* modalities: reifying a proof of $[\alpha_1][\alpha_2]\phi$ results in a system $\boldsymbol{\alpha}$

which refines $\alpha_1; \alpha_2$. Angelic programs are represented by duality α^d and the reification of first-order ϕ is a no-op skip. This style is interchangeable with normal-form CdGL proofs; we elide the duality ($[\alpha^d]\phi \leftrightarrow \langle\alpha\rangle\phi$) and skip ($[\mathsf{skip}]\phi \leftrightarrow \phi$) steps which convert between the two. Prior work [10] shows case-analysis, which is not *canonical*, is sometimes *normal* because state-dependent cases are decided only at runtime. Normal case analyses are analogous to case-tree normal forms in lambda calculi with coproducts [3]. Normal forms of (classical) ODE proofs have been characterized [9]. We call a game *system-test* if all its tests and domain constraints are system-test formulas. A formula is *system-test* if all modalities it mentions are box system modalities, while a proof is *system-test* if every context in its proof tree contains only system-test formulas. For the sake of defining system-test, the first-order propositional connectives are considered distinct from game modalities, rather than defined, i.e., first-order arithmetic expressions are permissible in the system-test fragment. Restricting reification to the system-test fragment ensures the reification of the hypothesis rule hyp is a system. System-test is stronger than *weak-test* (no modalities in tests) but weaker than *strong-test* (arbitrary modalities in tests). In the proof rules that follow, $\cdot\frac{y}{x}$ is the renaming of variable x to (usuallly fresh) variable y in a term, formula, game, or context.

Definitions

We define reification. Reification $\mathcal{A} \rightsquigarrow \boldsymbol{\alpha}$ is defined inductively on box CdGL proofs of system-test games. We first give the reification of case-analysis and hypothesis proofs, the only two normal proofs which are not introduction forms. In $\vee$E, $\mathcal{A}$ proves some first-order disjunction $\phi \vee \psi$, since proper choice game modalities $\langle\alpha \cup \beta\rangle$ are not permitted in system-test, normal-form proofs. In $\vee$E, the reified systems for the second and third premiss are $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$, likewise in every rule.

$$\text{hyp}\,\frac{(\text{if } [\alpha]\phi \in \Gamma)}{\Gamma \vdash [\alpha]\phi} \rightsquigarrow \alpha \qquad \vee\text{E}\,\frac{\Gamma \vdash \phi \vee \psi \quad \Gamma, \phi \vdash \varphi \quad \Gamma, \psi \vdash \varphi}{\Gamma \vdash \varphi} \rightsquigarrow \{?\phi; \boldsymbol{\beta}\} \cup \{?\psi; \boldsymbol{\gamma}\}$$

Hypothesis proofs do not give a concrete strategy for α and thus trivially refine α to itself. Case analysis allows Demon to choose either branch, so long as it is provable. The output is nondeterministic if ϕ and ψ are not mutually exclusive. Both ϕ and ψ are game-free in the system-test fragment and, in practical proofs, even quantifier-free first-order arithmetic. We first give the discrete *Angelic* cases, which plug in the specific Angel strategy from proof $\mathcal{A}$.

$$\langle\!\![:=]\!\!\rangle\text{I}\,\frac{\Gamma\frac{y}{x}, x = f\frac{y}{x} \vdash \phi}{\Gamma \vdash [\{x := f\}^d]\phi} \rightsquigarrow x := f; \boldsymbol{\alpha} \qquad \langle:*\rangle\text{I}\,\frac{\Gamma\frac{y}{x}, x = f\frac{y}{x} \vdash \phi}{\Gamma \vdash [\{x := *\}^d]\phi} \rightsquigarrow x := f; \boldsymbol{\alpha} \qquad \langle?\rangle\text{I}\,\frac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash [\{?\phi\}^d]\psi} \rightsquigarrow \boldsymbol{\beta}$$

$$\langle\!\![;]\!\!\rangle\text{I}\,\frac{\Gamma \vdash [\alpha^d][\beta^d]\phi}{\Gamma \vdash [\{\alpha; \beta\}^d]\phi} \rightsquigarrow \boldsymbol{\alpha} \qquad \langle\cup\rangle\text{IL}\,\frac{\Gamma \vdash [\alpha^d]\phi}{\Gamma \vdash [\{\alpha \cup \beta\}^d]\phi} \rightsquigarrow \boldsymbol{\alpha} \qquad \langle\cup\rangle\text{IR}\,\frac{\Gamma \vdash [\beta^d]\phi}{\Gamma \vdash [\{\alpha \cup \beta\}^d]\phi} \rightsquigarrow \boldsymbol{\alpha}$$

$$\langle*\rangle\text{I}\,\frac{\begin{array}{c}\Gamma \vdash \varphi \\ \varphi, \mathcal{M} \succ \mathbf{0} \wedge \mathcal{M}_0 = \mathcal{M} \vdash [\alpha^d](\varphi \wedge \mathcal{M}_0 \succ \mathcal{M}) \quad \varphi, \mathbf{0} \not\succ \mathcal{M} \vdash \phi\end{array}}{\Gamma \vdash [\{\alpha^*\}^d]\phi} \rightsquigarrow \{?\mathcal{M} \succ \mathbf{0}; \boldsymbol{\beta}\}^*; ?\mathbf{0} \not\succ \mathcal{M}; \boldsymbol{\gamma}$$

Discrete assignments remain in the output. Nondeterministic assignments are proved by providing a witness term f (rule $\langle:*\rangle$), which is preserved by the inductive call $\boldsymbol{\alpha}$. Subtly, Angelic tests can be eliminated by $\langle?\rangle$ because they are *proven to succeed* and because we wish only to keep tests which *Demon* is required to pass. A normal-form proof for a sequential composition $\alpha; \beta$ proves α with β in the postcondition. Normal Angelic choice proofs are injections, so Angelic proofs reify by $\langle\cup\rangle$R1 or $\langle\cup\rangle$R2 according to one branch or the other. Normal Angelic repetition proofs are by convergence: some metric $\mathcal{M}$ decreases to terminal value $\mathbf{0}$ while maintaining invariant formula φ. Variable $\mathcal{M}_0$ remembers the value of $\mathcal{M}$ at the start of each loop iteration for comparison purposes. Hybrid systems loops are nondeterministic, so Demon chooses the loop duration, but the Demonic test $\mathcal{M} \succ \mathbf{0}$ must pass at each repetition and $\mathbf{0} \not\succ \mathcal{M}$ must pass at the end, determinizing the loop duration.

To reify a discrete *Demonic* connective, we do not restrict Demon's capabilities, but recursively traverse the proof so that Angelic proofs can be reified.

$$\text{⟦}{:=}\text{⟧I}\,\frac{\Gamma\frac{y}{x}, x = f\frac{y}{x} \vdash \phi}{\Gamma \vdash [x := f]\phi} \leadsto x := f; \boldsymbol{\alpha} \qquad [{:}*]\,\frac{\Gamma\frac{y}{x} \vdash \phi}{\Gamma \vdash [x := *]\phi} \leadsto x := *; \boldsymbol{\alpha} \qquad \text{⟦};\text{⟧I}\,\frac{\Gamma \vdash [\alpha][\beta]\phi}{\Gamma \vdash [\alpha; \beta]\phi} \leadsto \boldsymbol{\alpha}$$

$$[?]\text{I}\,\frac{\Gamma, \psi \vdash \phi}{\Gamma \vdash [?\psi]\phi} \leadsto ?\psi; \boldsymbol{\alpha} \quad [\cup]\text{I}\,\frac{\Gamma \vdash [\alpha]\phi \quad \Gamma \vdash [\beta]\phi}{\Gamma \vdash [\alpha \cup \beta]\phi} \leadsto \boldsymbol{\alpha} \cup \boldsymbol{\beta} \quad [*]\text{I}\,\frac{\Gamma \vdash \psi \quad \psi \vdash [\alpha]\psi \quad \psi \vdash \phi}{\Gamma \vdash [\alpha^*]\phi} \leadsto \boldsymbol{\beta}^*; \boldsymbol{\gamma}$$

Nondeterministic Demonic assignments, unlike Angelic ones, are not modified during reification, because Demon retains the power to choose any value. Demonic tests introduce assumptions, and must continue to do so in the reification system to avoid changing the acceptable behavior. Demonic sequential compositions are like Angelic ones. Demonic choices refine each branch. Note that reification of games with form $\{\alpha \cup \beta\}; \gamma$ follows distributive normal forms $\{\alpha; \gamma\} \cup \{\beta; \gamma\}$, which are equivalent by $;\text{d}_r$. Demonic repetitions keep the loop, recalling that the coinductive loop invariant ψ justifies the postcondition by premiss $\mathcal{C}$.

We give the reification cases for ODEs. The reification of an invariant-based Demonic proof (dc and dw) is a *relaxation* of the ODE: the reified system need not follow the precise behavior of the ODE so long as all invariants required for the proof are obeyed. Indeed, this is where proof-based synthesis in ModelPlex [39] gains much of its power: real implementations never follow an ODE with perfect precision, but usually do follow its invariant-based relaxation.

$$\langle{}'\rangle\,\frac{\begin{array}{c}\Gamma \vdash d \geq 0 \quad \Gamma\frac{y}{x}, 0 \leq t \leq d, x = sln\frac{y}{x}, x' = f \vdash \psi \\ \Gamma\frac{y}{x}, 0 \leq t = d, x = sln\frac{y}{x}, x' = f \vdash \phi\end{array}}{\Gamma \vdash [t := 0; \{t' = 1, x' = f \,\&\, \psi\}^d]\phi} \leadsto t := d; x := sln; x' := f; \boldsymbol{\gamma}$$

$$[{}']\,\frac{\Gamma\frac{y}{x}, t \geq 0, \hat{\psi}, x = sln\frac{y}{x}, x' = f \vdash \phi}{\Gamma \vdash [t := 0; \{t' = 1, x' = f \,\&\, \psi\}]\phi} \leadsto t := 0; \{t' = 1, x' = f \,\&\, \psi\}; \boldsymbol{\alpha}$$

$$\text{dw}\,\frac{\Gamma\frac{y}{x}, \psi \vdash \phi}{\Gamma \vdash [x' = f \,\&\, \psi]\phi} \leadsto x := *; x' := f; ?\psi; \boldsymbol{\alpha}$$

$$\text{dc}\,\frac{\Gamma \vdash [x' = f \,\&\, \psi]\varphi \quad \Gamma \vdash [x' = f \,\&\, \psi \wedge \varphi]\phi}{\Gamma \vdash [x' = f \,\&\, \psi]\phi} \leadsto \boldsymbol{\beta}$$

$$\text{dg}\,\frac{\Gamma, y = f_0 \vdash [x' = f, y' = a(x)y + b(x) \,\&\, \psi]\phi}{\Gamma \vdash [x' = f \,\&\, \psi; \{y := *; y' := *\}^d]\phi} \leadsto y := f_0; \boldsymbol{\alpha}$$

Variable y is fresh in $\langle{}'\rangle$, $[{}']$, and dg. In $\langle{}'\rangle$, the side condition requires that y is fresh, term sln is the unique solution of the ODE, and chosen duration term d is constant throughout the ODE. The Angelic domain constraint is comparable to an Angelic test: it is soundly omitted in the reification because it is proven to pass. In Demonic ODE solutions ($[{}']$), the duration and domain constraint are *assumptions*, and formula $\hat{\psi} \equiv \forall 0 \leq s \leq t\, [t := s; x := sln]\psi$ says the domain constraint ψ holds through time t where s is fresh. Since our ODEs are Lipschitz, they have unique solutions and Demon could reify the unique solution of the ODE, as does case ($\langle{}'\rangle$). There is no obvious benefit to doing so, except that the reified system would fall within *discrete* dynamic logic. Differential Cut (dc) reification introduces an assumption in the domain constraint, and is sound by DC. By itself, dc *strengthens* a program, but in combination with dw enables relaxation of ODEs. Differential Weakening (dw) relaxes an

ODE by allowing x and x' to change *arbitrarily* so long as the domain constraint ψ (and thus invariants introduced by dc) remain true. Differential Ghost (dg) introduces a dimension to the ODE and reifies according to the recursive call. The introduced dimension is linear in order to soundly preserve the duration of the ODE. Assignment $y := f_0$ sets the initial value of the ghost variable to a chosen term.

Reification Example

Recall example PP and its safety property (1). Let $\mathcal{A}_{\mathsf{PP}}$ be the proof of (1) with a mirroring strategy described in Section 3.2. Then the reified result $\boldsymbol{\alpha}_{\mathsf{PP}}$ is

$$\begin{aligned}\boldsymbol{\alpha}_{\mathsf{PP}} = \big\{&\{L := -1; R := 1; x' = L + R \,\&\, x_l \le x \le x_r\}\\ &\cup\{L := 1; R := -1; x' = L + R \,\&\, x_l \le x \le x_r\}\big\}^*\end{aligned}$$

which we discuss step-by-step. Demonic repetition reification just repeats the body. Reifying a Demonic choice follows the structure of the proof, not the source program, hence the ODE occurs for each branch. Each branch commits to a choice of L, and each branch of $\mathcal{A}_{\mathsf{PP}}$ resolves the Angelic choice R to balance out L. When reifying an Angelic choice, only the branch taken is emitted. In $\boldsymbol{\alpha}_{\mathsf{PP}}$, we assume that $\mathcal{A}_{\mathsf{PP}}$ proves the ODE $x' = L + R \,\&\, x_l \le x \le x_r$ by replacing it with its solution, which is why the ODE appears verbatim in the refined system. A differential invariant proof could also be used with a differential cut (DC) of $x = x_0$, in which case physics are represented by the program $x := *; x' := *; ?x_l \le x \le x_r \wedge x = x_0$ in the result of reification. Different proofs generally give rise to different systems, some of which are less restrictive than others. Differential invariants, especially inequational invariants, ($x \ge x_0$ vs. $x = x_0$) can be more easily monitored with finite-precision numbers.

Note that the system $\boldsymbol{\alpha}_{\mathsf{PP}}$ is a refinement of PP and satisfies the same safety theorem $\mathsf{pre} \to [\boldsymbol{\alpha}_{\mathsf{PP}}]x = x_0$. Next, we show that this is the case for all reified strategies.

Metatheoretic Results

We state theorems (proven in our report [12]) showing how the reification of a game α refines α. Recall that Γ, α, ϕ, and $\mathcal{A}$ are in the *system-test* fragment of CdGL.

▶ **Theorem 8** (Systemhood). *If $\Gamma \vdash \mathcal{A} : [\alpha]\phi$ for system-test $\Gamma, \mathcal{A}$, and hybrid game α and $\mathcal{A} \rightsquigarrow \boldsymbol{\alpha}$ then $\boldsymbol{\alpha}$ is a system, i.e., it does not contain dualities.*

▶ **Theorem 9** (Reification transfer). *If $\Gamma \vdash \mathcal{A} : [\alpha]\phi$ for system-test $\Gamma, \mathcal{A}$, and hybrid game α and $\mathcal{A} \rightsquigarrow \boldsymbol{\alpha}$ then $\Gamma \vdash [\boldsymbol{\alpha}]\phi$ is provable in CdGL.*

▶ **Theorem 10** (Reification refinement). *If $\Gamma \vdash \mathcal{A} : [\alpha]\phi$ for system-test $\Gamma, \mathcal{A}$, and hybrid game α and $\mathcal{A} \rightsquigarrow \boldsymbol{\alpha}$ then $\Gamma \vdash \boldsymbol{\alpha} \le_{[\,]} \alpha$ is provable in CdGL.*

Theorem 8 is proven by trivial induction on $\mathcal{A}$. Theorem 9 is proven by inducting on $\mathcal{A}$, reusing its contents in a proof for $\boldsymbol{\alpha}$. Theorem 10 inducts on $\mathcal{A}$ and in each case appeals to the corresponding refinement rule. The fact that Theorem 10 could be proved validates the strength of CdGL's refinement rules.

7 Conclusion

We developed a refinement calculus for Constructive Differential Game Logic (CdGL). Technical challenges in this development included the facts that game logic is subnormal and that the *constructive* box and diamond modalities $[\alpha]\phi$ and $\langle\alpha\rangle\phi$ are not interdefinable.

We introduced a new constructive semantics for refinement and proved soundness. We formalized a reification operation and folklore theorem which reduce verified hybrid games to hybrid systems by specializing a game to the commitments made by its winning strategy. The immediate applications are synthesis tools and refinement-based proof tools for hybrid games. Theorem 8 and Theorem 9 support synthesis by ensuring that the reified system is a *system* which satisfies the *same* safety condition as the input game, which are respectively required in order to use existing synthesis tools and to ensure an end-to-end safety guarantee. Theorem 10 supports refinement-based proof technology: because the reified system refines the input game, game safety can be shown by choosing a strategy and showing the strategy safe. Once these tools are implemented, there are a wide array of applications studied in the hybrid systems and hybrid games literature which would benefit from the modeling power and synthesis guarantees that are possible with CdGL.

Our refinement calculus is of theoretical and practical interest beyond reducing games to systems. We expect that refinements can be used to provide shorter proofs, to compare the efficacy (dominance) of two strategies for the same game, and to determine when two strategies or programs should be considered "the same". These questions are worth pursuing both for hybrid games and for games in general.

References

1 Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Inf. Comput.*, 163(2):409–470, 2000. doi:10.1006/inco.2000.2930.

2 Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010. doi:10.1017/CBO9781139195881.

3 Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip J. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *LICS*, pages 303–310. IEEE, 2001. doi:10.1109/LICS.2001.932506.

4 Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002. doi:10.1145/585265.585270.

5 Ralph-Johan Back and Joakim von Wright. *Refinement Calculus - A Systematic Introduction.* Graduate Texts in Computer Science. Springer, 1998. doi:10.1007/978-1-4612-1674-2.

6 Richard Banach, Michael J. Butler, Shengchao Qin, Nitika Verma, and Huibiao Zhu. Core Hybrid Event-B I: Single Hybrid Event-B machines. *Sci. Comput. Program.*, 105:92–123, 2015. doi:10.1016/j.scico.2015.02.003.

7 Amit Bhatia, Lydia E. Kavraki, and Moshe Y. Vardi. Motion planning with hybrid dynamics and temporal goals. In *Conference on Decision and Control*, pages 1108–1115. IEEE, 2010. doi:10.1109/CDC.2010.5717440.

8 Errett Bishop. *Foundations of constructive analysis.* McGraw-Hill, 1967.

9 Rose Bohrer and André Platzer. Toward structured proofs for dynamic logics. *CoRR*, abs/1908.05535, 2019. arXiv:1908.05535.

10 Rose Bohrer and André Platzer. Constructive game logic. In Peter Müller, editor, *ESOP*, volume 12075 of *LNCS*. Springer, 2020. arXiv:2002.08523.

11 Rose Bohrer and André Platzer. Constructive hybrid games. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *IJCAR*, LNCS. Springer, 2020.

12 Rose Bohrer and André Platzer. Refining constructive hybrid games. *CoRR*, abs/2002.02576, 2020. arXiv:2002.02576.

13 Rose Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. VeriPhy: Verified controller executables from verified cyber-physical system models. In Dan Grossman, editor, *PLDI*, pages 617–630. ACM, 2018. doi:10.1145/3192366.3192406.

14 Douglas S Bridges and Luminita Simona Vita. *Techniques of constructive analysis.* Springer, 2007.

15 Sergio A. Celani. A fragment of intuitionistic dynamic logic. *Fundam. Inform.*, 46(3):187–197, 2001. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi46-3-01.

16 Krishnendu Chatterjee, Thomas A. Henzinger, and Nir Piterman. Strategy logic. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR*, volume 4703 of *LNCS*, pages 59–73. Springer, 2007. doi:10.1007/978-3-540-74407-8_5.

17 Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988. doi:10.1016/0890-5401(88)90005-3.

18 Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG*, volume 417 of *LNCS*, pages 50–66. Springer, 1988. doi:10.1007/3-540-52335-9_47.

19 Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-CoRN, the constructive Coq repository at Nijmegen. In Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors, *MKM*, volume 3119 of *LNCS*. Springer, 2004. Accessed: commits 9c44dae and 6411967. doi:10.1007/978-3-540-27818-4_7.

20 JW Degen and JM Werner. Towards intuitionistic dynamic logic. *Log. and Log. Philosophy*, 15(4):305–324, 2006. doi:10.12775/LLP.2006.018.

21 Peter Dybjer. Inductive families. *Formal Asp. Comput.*, 6(4):440–465, 1994. doi:10.1007/BF01211308.

22 Sebastian Enqvist, Helle Hvid Hansen, Clemens Kupke, Johannes Marti, and Yde Venema. Completeness for game logic. In *LICS*, pages 1–13. IEEE, 2019. doi:10.1109/LICS.2019.8785676.

23 Georgios E. Fainekos, Antoine Girard, Hadas Kress-Gazit, and George J. Pappas. Temporal logic motion planning for dynamic robots. *Automatica*, 45(2):343–352, 2009. doi:10.1016/j.automatica.2008.08.008.

24 Ioannis Filippidis, Sumanth Dathathri, Scott C. Livingston, Necmiye Ozay, and Richard M. Murray. Control design for hybrid systems with TuLiP: The temporal logic planning toolbox. In *Conference on Control Applications*, pages 1030–1041. IEEE, 2016. doi:10.1109/CCA.2016.7587949.

25 Cameron Finucane, Gangyuan Jing, and Hadas Kress-Gazit. LTLMoP: Experimenting with language, temporal logic and robot control. In *IROS*, pages 1988–1993. IEEE, 2010. doi:10.1109/IROS.2010.5650371.

26 John Nathan Foster. Bidirectional programming languages. Technical Report MS-CIS-10-08, Department of Computer & Information Science, University of Pennsylvania, Philadelphia, PA, March 2010.

27 Sujata Ghosh. Strategies made explicit in dynamic game logic. *Workshop on Logic and Intelligent Interaction at ESSLLI, Hamburg*, pages 74–81, 2008.

28 Valentin Goranko. The basic algebra of game equivalences. *Studia Logica*, 75(2):221–238, 2003. doi:10.1023/A:1027311011342.

29 David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic logic*. MIT Press, 2000. doi:10.1145/568438.568456.

30 George Edward Hughes and Max Cresswell. *A new introduction to modal logic*. Routledge, 1996.

31 Norihiro Kamide. Strong normalization of program-indexed lambda calculus. *Bull. Sect. Log. Univ. Łódź*, 39(1-2):65–78, 2010.

32 Marius Kloetzer and Calin Belta. A fully automated framework for control of linear systems from temporal logic specifications. *IEEE Trans. Automat. Contr.*, 53(1):287–297, 2008. doi:10.1109/TAC.2007.914952.

33 Dexter Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983. doi:10.1016/0304-3975(82)90125-6.

34 Dexter Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, 1997.

35 James Lipton. Constructive Kripke semantics and realizability. In Y. N. Moschovakis, editor, *Logic from Computer Science*, pages 319–357. Springer, 1992. doi:10.1017/978-1-4612-2822-6_13.

36 Sarah M. Loos and André Platzer. Differential refinement logic. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *LICS*, pages 505–514. ACM, 2016. doi:10.1145/2933575.2934555.

37 Evgeny Makarov and Bas Spitters. The Picard algorithm for ordinary differential equations in Coq. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *ITP*, volume 7998 of *LNCS*. Springer, 2013. doi:10.1007/978-3-642-39634-2_34.

38 Konstantinos Mamouras. Synthesis of strategies using the Hoare logic of angelic and demonic nondeterminism. *Log. Methods Comput. Sci.*, 12(3):1–41, 2016. doi:10.2168/LMCS-12(3:6)2016.

39 Stefan Mitsch and André Platzer. ModelPlex: Verified runtime validation of verified cyber-physical system models. *Form. Methods Syst. Des.*, 49(1):33–74, 2016. doi:10.1007/s10703-016-0241-z.

40 Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008. doi:10.1017/S0956796808006953.

41 Rohit Parikh. Propositional game logic. In *FOCS*, pages 195–200. IEEE, 1983. doi:10.1109/SFCS.1983.47.

42 André Platzer. Differential dynamic logic for hybrid systems. *J. Autom. Reas.*, 41(2):143–189, 2008. doi:10.1007/s10817-008-9103-8.

43 André Platzer. The structure of differential invariants and differential cut elimination. *Log. Methods Comput. Sci.*, 8(4):1–38, 2012. doi:10.2168/LMCS-8(4:16)2012.

44 André Platzer. Differential game logic. *ACM Trans. Comput. Log.*, 17(1):1:1–1:51, 2015. doi:10.1145/2817824.

45 André Platzer. Differential hybrid games. *ACM Trans. Comput. Log.*, 18(3):19:1–19:44, 2017. doi:10.1145/3091123.

46 André Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer, Switzerland, 2018. doi:10.1007/978-3-319-63588-0.

47 André Platzer and Yong Kiam Tan. Differential equation invariance axiomatization. *J. ACM*, 67(1), 2020. doi:10.1145/3380825.

48 Ramaswamy Ramanujam and Sunil Easaw Simon. Dynamic logic on games with structured strategies. In Gerhard Brewka and Jérôme Lang, editors, *Knowledge Representation*, pages 49–58. AAAI Press, 2008. URL: http://www.aaai.org/Library/KR/2008/kr08-006.php.

49 Ankur Taly and Ashish Tiwari. Switching logic synthesis for reachability. In Luca P. Carloni and Stavros Tripakis, editors, *EMSOFT*, pages 19–28. ACM, 2010. doi:10.1145/1879021.1879025.

50 The Coq development team. The Coq proof assistant reference manual, 2019. URL: https://coq.inria.fr/.

51 Johan Van Benthem. Games in dynamic-epistemic logic. *Bull. Econ. Research*, 53(4):219–248, 2001.

52 Johan van Benthem. Logic of strategies: What and how? In Johan van Benthem, Sujata Ghosh, and Rineke Verbrugge, editors, *Models of Strategic Reasoning - Logics, Games, and Communities*, volume 8972 of *LNCS*, pages 321–332. Springer, 2015. doi:10.1007/978-3-662-48540-8_10.

53 Johan van Benthem and Eric Pacuit. Dynamic logics of evidence-based beliefs. *Studia Logica*, 99(1-3):61–92, 2011. doi:10.1007/s11225-011-9347-x.

54 Johan van Benthem, Eric Pacuit, and Olivier Roy. Toward a theory of play: A logical perspective on games and interaction. *Games*, 2011. doi:10.3390/g2010052.

55 Wiebe van der Hoek, Wojciech Jamroga, and Michael J. Wooldridge. A logic for strategic reasoning. In Frank Dignum, Virginia Dignum, Sven Koenig, Sarit Kraus, Munindar P. Singh, and Michael J. Wooldridge, editors, *AAMAS*. ACM, 2005. doi:10.1145/1082473.1082497.

56 Jaap van Oosten. Realizability: A historical essay. *Math. Structures Comput. Sci.*, 12(3):239–263, 2002. doi:10.1017/S0960129502003626.

57 Klaus Weihrauch. *Computable Analysis - An Introduction.* Texts in Theoretical Computer Science. Springer, 2000. doi:10.1007/978-3-642-56999-9.

58 Duminda Wijesekera. Constructive modal logics I. *Ann. Pure Appl. Log.*, 50(3):271–301, 1990. doi:10.1016/0168-0072(90)90059-B.

59 Duminda Wijesekera and Anil Nerode. Tableaux for constructive concurrent dynamic logic. *Ann. Pure Appl. Log.*, 135(1-3):1–72, 2005. doi:10.1016/j.apal.2004.12.001.

Data-Flow Analyses as Effects and Graded Monads

Andrej Ivašković
Department of Computer Science and Technology, University of Cambridge, UK
andrej.ivaskovic@cst.cam.ac.uk

Alan Mycroft
Department of Computer Science and Technology, University of Cambridge, UK
alan.mycroft@cst.cam.ac.uk

Dominic Orchard
School of Computing, University of Kent, UK
d.a.orchard@kent.ac.uk

Abstract

In static analysis, two frameworks have been studied extensively: monotone data-flow analysis and type-and-effect systems. Whilst both are seen as general analysis frameworks, their relationship has remained unclear. Here we show that monotone data-flow analyses can be encoded as effect systems in a uniform way, via algebras of transfer functions. This helps to answer questions about the most appropriate structure for general effect algebras, especially with regards capturing control-flow precisely. Via the perspective of capturing data-flow analyses, we show the recent suggestion of using *effect quantales* is not general enough as it excludes non-distributive analyses e.g., *constant propagation*. By rephrasing the McCarthy transformation, we then model monotone data-flow effects via *graded monads*. This provides a model of data-flow analyses that can be used to reason about analysis correctness at the semantic level, and to embed data-flow analyses into type systems.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases data-flow analysis, effect systems, graded monads, correctness

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.15

Supplementary Material Code and additional proofs: `https://doi.org/10.5281/zenodo.3784967`

Funding *Andrej Ivašković*: funded by Trinity College, Cambridge (Internal Graduate Scholarship). *Dominic Orchard*: funded in part by EPSRC grant EP/T013516/1.

1 Introduction

Static program analysis is the bedrock of optimising compilation, extracting program properties from syntax to inform semantics-preserving program transformations. Throughout the history of program analysis it has been repeatedly noticed that various analyses have similar forms and can thus be unified into more general frameworks. Notably, the early *data-flow analyses* performed on control-flow graphs (e.g., for live variables, available expressions, reaching definitions etc.) were unified by the notion of *monotone data-flow frameworks* [8] (Khedker et al. [10] give a wider perspective). Such analyses are formalised as scanning program statements forwards or backwards to obtain data-flow equations over some algebraic structure, which are then solved. Another major class of analyses are *effect systems* [5, 7, 15, 24], typically applied in a functional setting (but also notably for Java's checked exceptions). Effect systems typically augment type systems with information about possible side-effects, drawn from a particular algebraic structure. Such approaches evolved into a framework for general static analysis [16]. Another general class of static analysis is *abstract interpretation* given by Galois connections or related structures [3], though this is not our focus here.

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 15; pp. 15:1–15:23
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Despite claims of effect systems' generality, it has been unclear whether they have sufficient expressive power to capture classical data-flow analyses, due in part to the functional-style bias of effect systems but also due to a lack of clarity about how effect systems interact with control-flow. Various approaches have developed effect-system-like systems for capturing particular data-flow analyses, but typically in an *ad hoc* manner. For example, Nielson, Nielson, and Hankin [17] presented an effect-system-like annotated type system for reaching definitions analysis, but the approach was not clearly linked to a general algebraic characterisation of effects seen elsewhere. Laud et al. [11] introduced several type systems that represent data-flow analyses, but these are not effect systems and the approach is not unified.

In this work, we study the general relationship of dataflow frameworks to effect systems, and through this investigate the most appropriate algebraic characterisation of effects to capture known analyses in a uniform way. While Gifford-Lucassen-style [5] effect annotations were originally seen as mere subsets of the space of possible effect operations along with a single composition operator, Amtoft and the Nielsons [1] showed how distinct sequencing and alternation operators for composing effects gave better expressivity, capturing various other analyses. Recently, Katsumata [9] and Orchard et al. [20] linked effect systems to the mathematical notion of *graded monads*, using graded monads to model languages with effect systems. The graded monad model characterises the algebraic structure of effect systems by the structure of its *grades* which constrain the model of a computation's side effects. In this setting, Katsumata offers the most-general framework for effect systems: an *effect algebra* is a pre-ordered monoid $(D, \sqsubseteq, \rhd, 1)$, where $(D, \sqsubseteq)$ is a pre-ordered set and $(D, \rhd, 1)$ a monoid with $\rhd$ monotonic with respect to $\sqsubseteq$ [9]. Binary least upper bounds on D, if they exist, give a natural (if partial) alternation operator. Gordon [6] by contrast aims, in recent work, at a more precise axiomatisation using *effect quantales* which enforce composition and alternation to be total, if necessary by adding an additional top (or error) element to D, and adding distributivity requirements. Mycroft et al. [15] also split effect algebras into separate operators for sequencing and alternation, with a graded monad model.

Two questions arise. Firstly, how related are the theories of data-flow analysis and effect systems, and their interpretation as graded monads? Secondly, what is the most natural structure for an effect algebra that covers common analyses?

Contributions and structure. Section 2 begins by summarising various background material about data-flow analyses and effect systems. We then contribute three main results:

1. We show that monotone data-flow frameworks can be captured via a kind of effect system on control-flow graphs (CFGs) with effect algebras of transfer functions (Section 3). The approach ends up resembling Kam and Ullman's monotone data-flow analysis frameworks. The novelty is that the approach unifies several classical data-flow analyses.
2. We adapt McCarthy's transformation [12] to translate the CFG-effect system of Section 3 into a *graded monad* rendering of effect systems (Section 4). This gives a semantic model equipped with data-flow analysis information which can be used to reason about analysis correctness or to capture dataflow as types, which we demonstrate via a Haskell encoding.
3. We discuss how effect quantales are too restrictive to capture non-distributive data-flow analyses such as constant propagation (Section 5).

There are several interesting lines of further work that follow from the perspective of this paper. For example, computational complexity of data-flow analysis algorithms is well understood and results from this field may provide valuable insight in constructing efficient type-and-effect inference algorithms. We also aim to contribute towards finding a 'best' model for effect algebras – one that imposes just enough restrictions that every static analysis can be modelled, while disallowing models which correspond to no (known) static analysis.

2 Background

2.1 Analysis structure: partial orders and lattices

Program analysis generally captures program properties as elements of a partially ordered set (*poset*). Often this poset forms a complete lattice, but this places strong requirements on the existence of least upper and greatest lower bounds, which are not always needed or desired. For example, in type inference we may infer that two expressions e_1 and e_2 have respective types Int and Bool, but then say that a conditional selecting between e_1 and e_2 is ill-typed. There seems to be a tacit understanding that static type inference is usually partial while static determination of other properties is total – perhaps because we are happy for a program to be rejected as ill-typed, but not for a compiler to reject our program just because a static analysis says it is unfit for a given optimisation. In general this distinction between partial and total analyses affects "formal presentation" more than "conceptual understanding" as we can make any partial analysis total by adding a $\top$ element to its poset of values.

A poset $(D, \sqsubseteq)$ is a set D with a reflexive, antisymmetric and transitive relation $\sqsubseteq$. Given two posets, $(D_1, \sqsubseteq_1)$ and $(D_2, \sqsubseteq_2)$ then their product $D_1 \times D_2$ has the induced product order: $(x, y) \sqsubseteq (x', y')$ whenever $x \sqsubseteq_1 x'$ and $y \sqsubseteq_2 y'$. Similarly, given any set X then $X \to D$ becomes a poset with induced ordering $f \sqsubseteq g$ whenever $\forall x \in X. f(x) \sqsubseteq g(x)$.

A poset $(D, \sqsubseteq)$ is a *(bounded) join-semilattice* if all finite (including empty) subsets $X \subseteq D$ have a least upper bound with respect to $\sqsubseteq$. It is a *(bounded) lattice* if such subsets also have a greatest lower bound. It is a *complete lattice* if all subsets have least upper bounds and greatest lower bounds. We write $\bot$ for $\bigsqcup\{\}$ and $\top$ for $\bigsqcup D$ when these exist. Much work on program analysis is done on posets of *finite height* (every totally ordered subset is finite) so completeness adds no additional requirements.

A join-semilattice is often axiomatised via an operator $(D, \sqcup)$ because this gives an algebraic characterisation; the relation $\sqsubseteq$ can be recovered by taking $x \sqsubseteq y \Leftrightarrow x \sqcup y = y$.

For data-flow analysis of Turing-complete languages we generally need $(D, \sqsubseteq)$, or $(D, \sqcup)$ to be *bounded* or *pointed*, i.e. to have a least element $\bot$ which can represent the data-flow values resulting from a non-terminating expression, and also serves as the initial value for a Tarski fixed-point iteration when solving data-flow equations.

2.2 Control-flow graphs

Classical compiler optimisations usually deal with simple imperative programs, represented as *control-flow graphs* (CFGs). Here statements S appearing within the flow graph and possibly containing branches to labels ℓ are given by:

$$\begin{array}{rcll} v & ::= & X \mid k & \text{(syntactic values)} \\ e & ::= & v \mid v_1 \ op \ v_2 & \text{(expressions)} \\ S & ::= & X := e; \ \mathsf{goto}\ \ell \mid \mathsf{if}\ v \geq 0\ \mathsf{then\ goto}\ \ell'\ \mathsf{else\ goto}\ \ell'' \mid \mathsf{halt}\ v & \text{(statements)} \end{array}$$

where k are assumed to be integers, op ranges over arithmetic operators ($+$, $-$, $\times$ etc.), and X ranges over $Vars$, a finite set of integer-valued mutable variables.

A CFG $(N, E \subseteq N \times N, \mathcal{L} : N \to S)$ is a directed graph whose nodes N are labelled with 3-address arithmetic and control-flow statements. We use n (and ℓ when thinking of a node as a label) to range over N. As usual, we write $succ(n)$ and $pred(n)$ for the sets of E-successors and E-predecessors of n, and require the number of successors of a node to respect the labelling $\mathcal{L}$. We write $(\ell : S)$ to indicate that node ℓ is labelled with a given statement or, in programming terms, that statement S has label ℓ.

2.3 Classical data-flow analysis

Data-flow analysis refers to static analysis approaches commonly used in optimising compilers. These analyses infer facts about how data is used in the program, including constant propagation, live variables and pointer analysis.

Liveness. In a CFG, a variable x is *live* at node n if there is a (possibly infeasible) path of edges starting at n along which the value of X is read before being written to. The sets of variables live on entry and exit of n respectively satisfy the following data-flow equations:

$$\mathit{LiveIn}(n) = (\mathit{LiveOut}(n) \setminus \mathit{LiveKill}(n)) \cup \mathit{LiveGen}(n) \qquad \mathit{LiveOut}(n) = \bigcup_{s \in \mathit{succ}(n)} \mathit{LiveIn}(s)$$

Sets $\mathit{LiveKill}(n)$ and $\mathit{LiveGen}(n)$ are determined by the statement at node n. For statement $X := e$ they are $\mathit{LiveKill}(n) = \{X\}$ and $\mathit{LiveGen}(n) = \mathit{fv}(e)$ (the set of free variables in e). For halt v and if $v \geq 0$ they are $\mathit{LiveKill}(n) = \emptyset$ and $\mathit{LiveGen}(n) = \mathit{fv}(v)$.

We consider $\mathit{LiveIn}(n)$ to be the set of live variables just before the statement at node n, and $\mathit{LiveOut}(n)$ to be the live variables immediately after this statement. The notation $\mathit{Live}(n)$, $\mathit{gen}(n)$, and $\mathit{kill}(n)$ are used as synonyms for $\mathit{LiveIn}(n)$, $\mathit{LiveGen}(n)$, and $\mathit{LiveKill}(n)$.

Monotone data-flow analysis frameworks. Liveness, along with several other analyses, can be seen as examples of Kam and Ullman's *monotone data-flow analysis frameworks* [8]. Roughly speaking, a monotone data-flow analysis framework[1] *instance* is specified by a lattice $(\mathit{DFValues}, \sqcup)$ with:

- the set $\mathit{DFValues}$ of all possible data-flow values with a *least element* $\bot$;
- the direction of the analysis, *forwards* or *backwards* (liveness is backwards since $\mathit{LiveOut}$ is calculated from successors);
- Gen and Kill sets for every statement;
- the *merge* operation $\sqcup$ (for liveness and reaching definitions this is $\cup$, whereas for available expressions and very busy expressions it is $\cap$).

As with liveness, such instances give a set of equations whose solutions give the data-flow values at every node in the CFG. An exception is made in the cases of incoming data-flow values for entry nodes in forwards analysis and outgoing data-flow in exit nodes in backwards analysis – they do not depend on other data-flow values, they are instead equal to the *boundary information* (BI, typically $\bot$ or $\top$). When there are multiple solutions, we take the *least* one (which exists because of the lattice assumption and the existence of $\bot$ in $\mathit{DFValues}$). An iterative *work-list algorithm* is used to compute data-flow values at every node.

Every node in a CFG determines a *transfer function* (or *flow function*) from the set of data-flow values at one end of the node to that at the other end (*In* to *Out* for forwards analyses and *Out* to *In* for backwards analyses). Transfer functions propagate data-flow values around a program. For backwards analyses, the transfer function ϕ satisfies $\mathit{DF}_{\mathit{In}}(n) = \phi(\mathit{DF}_{\mathit{Out}}(n))$; for forward analyses $\mathit{DF}_{\mathit{Out}}(n) = \phi(\mathit{DF}_{\mathit{In}}(n))$ (where $\mathit{DF}_{\mathit{In}}$ and $\mathit{DF}_{\mathit{Out}}$ map nodes to the data-flow values at entry and exit, like LiveIn and $\mathit{LiveOut}$ previously).

We can extend the idea of transfer functions for single statements to sequences of statements (or paths in a CFG). Consider, for example, statements S_1 and S_2 with associated transfer functions ϕ_1 and ϕ_2. Then the transfer function for the sequence "S_1 then S_2" is

[1] The original work calculated maximal fixed points by iterating from a $\top$ value. We use the dual formulation (least fixed points and $\bot$ value). Our data-flow examples use complete lattices.

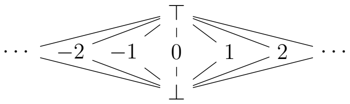


Figure 1 Lattice of integers $\mathbb{Z}_\bot^\top$ with $\top$ and $\bot$ whose product lattice is the lattice of data-flow values in constant propagation.

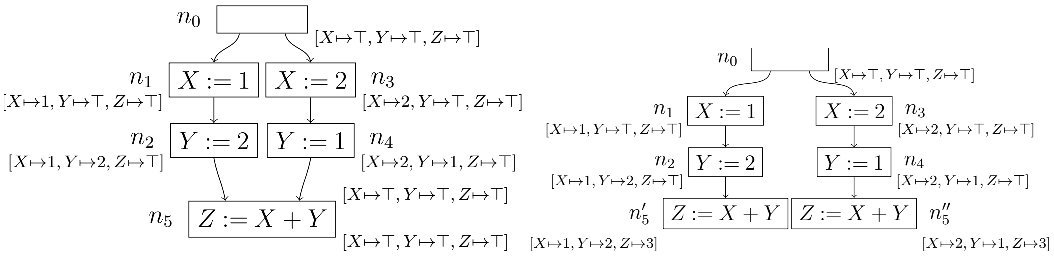

n_0
$[X{\mapsto}\top, Y{\mapsto}\top, Z{\mapsto}\top]$
n_1
$X := 1$
$X := 2$
n_3
$[X{\mapsto}1, Y{\mapsto}\top, Z{\mapsto}\top]$
$[X{\mapsto}2, Y{\mapsto}\top, Z{\mapsto}\top]$
n_2
$Y := 2$
$Y := 1$
n_4
$[X{\mapsto}1, Y{\mapsto}2, Z{\mapsto}\top]$
$[X{\mapsto}2, Y{\mapsto}1, Z{\mapsto}\top]$
$[X{\mapsto}\top, Y{\mapsto}\top, Z{\mapsto}\top]$
n_5
$Z := X + Y$
$[X{\mapsto}\top, Y{\mapsto}\top, Z{\mapsto}\top]$
n_0
$[X{\mapsto}\top, Y{\mapsto}\top, Z{\mapsto}\top]$
n_1
$X := 1$
$X := 2$
n_3
$[X{\mapsto}1, Y{\mapsto}\top, Z{\mapsto}\top]$
$[X{\mapsto}2, Y{\mapsto}\top, Z{\mapsto}\top]$
n_2
$Y := 2$
$Y := 1$
n_4
$[X{\mapsto}1, Y{\mapsto}2, Z{\mapsto}\top]$
$[X{\mapsto}2, Y{\mapsto}1, Z{\mapsto}\top]$
n_5'
$Z := X + Y$
$Z := X + Y$
n_5''
$[X{\mapsto}1, Y{\mapsto}2, Z{\mapsto}3]$
$[X{\mapsto}2, Y{\mapsto}1, Z{\mapsto}3]$

Figure 2 Non-distributivity of constant propagation. Assume data-flow into n_0 is such that multiple values can be associated with each of X, Y, Z and so these are mapped to $\top$. Non-distributivity manifests at node n_5 in the CFG on the left: whichever execution path is taken after n_0, Z has value 3 at n_5, but constant-propagation analysis gives $Z \mapsto \top$. Splitting the statement into two and performing constant-propagation analysis on these paths separately (as in the CFG on the right) gives $Z \mapsto 3$ which is more precise than $Z \mapsto \top$ in the left CFG, violating distributivity.

$\phi_1 \circ \phi_2$ for a backwards analysis, whereas for a forwards analysis it is $\phi_2 \circ \phi_1$ (this reverse composition is natural for forwards analysis, since $\phi_2 \circ \phi_1$ first applies ϕ_1 to the input, then applies ϕ_2 to the result).

Data-flow analyses may also have a notion of *distributivity* relating to the merging operator. A forwards or backwards analysis is *distributive* if it satisfies the following corresponding property for all nodes n:

$$DF_{Out}(n) = \bigsqcup_{n' \in pred(n)} \phi_n(DF_{Out}(n')) \;\; (\mathit{forward}) \qquad DF_{In}(n) = \bigsqcup_{n' \in succ(n)} \phi_n(DF_{In}(n')) \;\; (\mathit{backward})$$

where ϕ_n is the transfer function for node n. Live variable analysis is a distributive analysis.

Constant propagation. Some data-flow analyses are not distributive. One such example is *constant propagation*: a forwards analysis that associates with each program point a mapping (ranged over by s here) from variables to data-flow values which are either an integer or one of two special symbols $\bot$ or $\top$. The mapping $X \mapsto \top$ means that variable X potentially takes multiple values and so is not (known to be) a constant, whereas $Y \mapsto \bot$ means that the value of variable Y has not been explored yet in the analysis (this is needed for loops where the analysis uses fixed-point iteration). For integer variables, this gives a lattice of data-flow values of integers, along with $\bot$ and $\top$, shown in Figure 1.

This lattice naturally gives rise to a lattice of mappings which is just a *product lattice* with the subtlety that if one variable maps to $\bot$ then all do (a so-called $\bot$-*coalesced product*), with the partial order $\sqsubseteq$ lifted to the product space, e.g. $[X \mapsto 1,\ Y \mapsto 5] \sqsubseteq [X \mapsto 1,\ Y \mapsto \top]$ but $[X \mapsto 1,\ Y \mapsto 5] \not\sqsubseteq [X \mapsto 3,\ Y \mapsto 5]$. The formula $s_1 \sqsubseteq s_2$ can be read as "s_1 is more precise than s_2". The result of the analysis should be a least solution of the data-flow equations. Merging is via least upper bounds ($\sqcup$) taken component-wise, e.g. $[X \mapsto 1,\ Y \mapsto 3] \sqcup [X \mapsto 2,\ Y \mapsto 3] = [X \mapsto \top,\ Y \mapsto 3]$.

Figure 2 shows via an example that the analysis is not distributive.

2.4 Effect systems and effect algebras

Type-and-effect systems extend type systems to analyse impure concepts such as IO, exceptions and mutable state [5, 7, 24, 15]. Type-and-effect judgements are often written as $\Gamma \vdash e : \tau \,\&\, F$ for an expression e of type τ in context Γ with potential effects described by F. (For the remainder of Section 2 we take e as ranging over general programming-language expressions.) For example, F might be the set of exceptions the expression e may throw. Type-and-effect systems also introduce *latent effect* annotations in functions, for example $\tau_1 \xrightarrow{F} \tau_2$ is the type of a function which has effect F when applied.

The simplest effect systems use powersets of symbols representing possible impure program actions, ignoring control-flow and statement order by using $\cup$ to combine effect information (e.g., [5, 24]). Consider the effect system that captures the set of exceptions that an expression may raise. In this case, the inference rule for conditionals is:

$$(\text{IF})\ \frac{\Gamma \vdash e_1 : \mathsf{Bool} \,\&\, F_1 \qquad \Gamma \vdash e_2 : \tau \,\&\, F_2 \qquad \Gamma \vdash e_3 : \tau \,\&\, F_3}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau \,\&\, F_1 \cup F_2 \cup F_3}$$

However, this only covers 'may' analyses, and not 'must' analyses, and furthermore it only allows for commutative effect combination. Amtoft et al. [1] therefore introduced separate operators for sequencing ($\rhd$) and combining alternate effects ($\sqcup$) e.g., in if-then-else-style conditionals. The meaning of $\rhd$ is such that $F_1 \rhd F_2$ is the cumulative effect of two sequenced operations, where the first has the effect F_1 and the second F_2. This sequential composition of effects, in a space D, is generally modelled as a monoid $(D, \rhd, 1)$ where $\rhd$ is an associative operation with identity element 1. The previous inference rule now becomes:

$$(\text{IF})\ \frac{\Gamma \vdash e_1 : \mathsf{Bool} \,\&\, F_1 \qquad \Gamma \vdash e_2 : \tau \,\&\, F_2 \qquad \Gamma \vdash e_3 : \tau \,\&\, F_3}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau \,\&\, F_1 \rhd (F_2 \sqcup F_3)}$$

The effect system now distinguishes sequencing from branching and allows the former to be non-commutative. Such effect systems, which take control flow into account, are sometimes referred to as *sequential* (or *flow-sensitive*) *effect systems* [26].

One algebraic characterisation of these effect-system operators, due to Katsumata [9], is as a partially-ordered[2] monoid (*pomonoid*), which we write as a quadruple $(D, \sqsubseteq, \rhd, 1)$ where D is both a poset (ordered by $\sqsubseteq$) and a monoid (with $\sqsubseteq$-monotonic operation $\rhd$).

Gordon argues for a special case of this model[3] called *effect quantales* [6]. An effect quantale $(D, \sqcup, \rhd)$ is a bounded join-semilattice where $\rhd$ distributes over $\sqcup$ on both sides: $x \rhd (y \sqcup z) = (x \rhd y) \sqcup (x \rhd z)$ and $(y \sqcup z) \rhd x = (y \rhd x) \sqcup (z \rhd x)$. Gordon adds the requirement that $(D, \sqcup)$ has a $\top$ element (but this holds whenever D is finite-height) and also that $\top$ is a left- and right-zero for $\rhd$.

Distributivity of $\rhd$ over $\sqcup$ implies its monotonicity w.r.t. $\sqsubseteq$, but not vice versa.

Both Katsumata's pomonoids and Gordon's effect quantales form bases for sequential effect systems. Effect quantales are a special case of pomonoids, but have the laudable aim to be closer to modelling only those effect algebras which are useful in practice. We however argue that the distributivity requirement of effect quantales is too strong (Section 3.4).

[2] Katsumata proposed a *pre-ordered monoid*, but this becomes a partially ordered monoid after quotienting by equivalence classes hence our slight re-characterisation here to match the partial-order setting of data-flow analysis.

[3] Here we consider only sequential composition and alternation; Gordon's work also considers iteration.

$$(\textsc{if})\ \frac{\Phi(\ell) = \langle\!\langle v \rangle\!\rangle_{\mathrm{TF}} \rhd (\Phi(\ell_1) \sqcup \Phi(\ell_2))}{\Phi \vdash (\ell : \mathsf{if}\ v \geq 0\ \mathsf{then\ goto}\ \ell_1\ \mathsf{else\ goto}\ \ell_2) : \mathsf{Int}\ \&\ \Phi(\ell)}$$

$$(\textsc{assign})\ \frac{\Phi(\ell) = \langle\!\langle \ell : X := e \rangle\!\rangle_{\mathrm{TF}} \rhd \Phi(\ell')}{\Phi \vdash (\ell : X := e;\ \mathsf{goto}\ \ell') : \mathsf{Int}\ \&\ \Phi(\ell)} \qquad (\textsc{halt})\ \frac{\Phi(\ell) = \langle\!\langle v \rangle\!\rangle_{\mathrm{TF}}}{\Phi \vdash (\ell : \mathsf{halt}\ v) : \mathsf{Int}\ \&\ \Phi(\ell)}$$

Figure 3 Data-flow effect system for the imperative language of CFGs.

3 An effect system for data-flow analysis

As discussed in Section 1, specific data-flow analyses have sometimes been given *ad hoc* characterisations as effect-system-like analyses (e.g., Nielson et al.'s annotated type system for reaching definitions [17]). Here we introduce a more general, unifying approach based on a type-and-effect system for CFGs in which statements in the language of Section 2.2 are given effect annotations corresponding to transfer functions. Since assignments and branches contain goto ℓ, their overall ("run to completion") effect does not directly correspond to traditional transfer functions of CFG nodes. Section 3.1 explores the details, introducing an effect system for liveness. Section 3.2 considers inference. Section 3.3 then generalises the system to classical dataflow analyses and constant propagation, which is non-distributive.

3.1 Type-and-effect system and inference rules for liveness

Recall the language of CFGs introduced in Section 2.2. We wrote $(\ell : S)$ to mean node ℓ is associated with statement S. We introduce judgements capturing the type and effect of running to completion a CFG program starting at a given statement. The judgement form is $\Phi \vdash (\ell : S) : \tau\,\&\,\phi$, where τ is a type and ϕ is an effect annotation given by a transfer function that is a *combination* of transfer functions on paths from ℓ up to a halt. More precisely: for liveness, applying ϕ to the live variable set at program exit (the boundary information) gives the live set at ℓ.

The role of Φ (which is a map from labels to transfer functions) is more subtle. Normally, type-and-effect systems are given in a syntax-directed manner. But loops in programs behave like recursive functions, requiring finding a fixed point (with potentially multiple solutions). Here, we posit a solution Φ giving the data-flow value at each program point, and use inference rules to assert this is consistent. There may be multiple possible Φ (fixed points).

The type-and-effect system of this form that describes liveness is given in Figure 3. It uses various functions and symbols. The operators $\sqcup$ and $\rhd$ are $\cup$ and $\circ$ (function composition), respectively. The notation $\langle\!\langle \ell : X := e \rangle\!\rangle_{\mathrm{TF}}$ and $\langle\!\langle v \rangle\!\rangle_{\mathrm{TF}}$ represents transfer functions corresponding respectively to assignments $X{:=}e$ at label ℓ, and variable references in halt v and if $v \geq 0$ statements. They are respectively $\lambda s.\ (s \setminus \mathit{kill}(\ell : X := e)) \cup \mathit{gen}(\ell : X := e)$ and $\lambda s.\ s \cup \mathit{fv}(v)$. We interpret these rules inductively and we are interested in the *least* solution (in terms of Φ in the partial order of functions from labels to transfer functions).

▶ **Theorem 1.** *Let $\hat{\Phi}$ be the least solution for a CFG that contains an instruction with label ℓ. Then $\hat{\Phi}(\ell)(\emptyset)$ is equal to the set of live variables at node ℓ of the CFG.*

Proof. This is restating a well-known fact about transfer functions by Sharir and Pnueli [22]. A statement and proof of it can be found in, for example, Theorem 7-3.4 in Muchnick and Jones [14]. Using their notation, the expression $\hat{\Phi}(\ell)(\emptyset)$ corresponds to $z_\ell = \chi_\ell(\emptyset)$ and the live set at ℓ is x_ℓ, and the theorem states that $x_\ell = z_\ell$. See Appendix A for details. ◀

Effect systems are traditionally applied to functional languages to analyse impure code. Thus calling our approach here an "effect system" may seem unorthodox. Our justification is that the inference system in Figure 3 is the pre-image of the translation in Section 4 (based on the McCarthy transformation) from CFGs into functional code with a type-and-effect system (via graded monads) mapping transfer functions to type-based effect information.

Our effect system here resembles Nielson et al.'s "annotated type system" [17] capturing reaching-definitions analysis for a simple imperative `while` language. The main difference is that we operate with transfer functions on CFGs (which, to the best of our knowledge, is a novel approach), unifying several monotone data-flow analyses (shown in Section 3.3).

3.2 Inferring effects

Given a labelled imperative program as in Section 3.1, we want to find the effects associated with every single label. We present a method to infer the *principal* solution to this problem.

Statements in a CFG are uniquely labelled. Thus we can see a CFG as a set of tuples $(\ell : S)$, where ℓ is a label and S is a statement. Let ϕ_ℓ be the effect associated with the label ℓ, so that $\Phi \vdash (\ell : S) : \tau \,\&\, \phi_\ell$ holds. For every statement there is an associated set of constraints involving its effect. These constraints resemble the rules given in Figure 3. They are given in the form of inequalities that use a subeffecting relation $\sqsubseteq$, which in the case of liveness is just $\subseteq$ lifted to the function space. Each statement form below emits the indicated constraint (these are conventionally expressed using $\sqsupseteq$, the converse of $\sqsubseteq$):

$$\begin{array}{lcl}
(\ell : \mathsf{halt}\ v) & \Longrightarrow & \phi_\ell \sqsupseteq \langle\!\langle v \rangle\!\rangle_{\mathrm{TF}} \\
(\ell : \mathsf{if}\ v \geq 0\ \mathsf{then\ goto}\ \ell_1\ \mathsf{else\ goto}\ \ell_2) & \Longrightarrow & \phi_\ell \sqsupseteq \langle\!\langle v \rangle\!\rangle_{\mathrm{TF}} \rhd (\phi_{\ell_1} \sqcup \phi_{\ell_2}) \\
(\ell : X := e;\ \mathsf{goto}\ \ell') & \Longrightarrow & \phi_\ell \sqsupseteq \langle\!\langle \ell : X := e \rangle\!\rangle_{\mathrm{TF}} \rhd \phi_{\ell'}
\end{array}$$

We seek the least solution (w.r.t. $\sqsubseteq$) for this set of constraints. Since the domain of the constraints is the lattice of transfer functions, finding the least solution is done by using a simple work-list algorithm: Initially all ϕ_ℓ are set to $\bot_{\mathit{DFValues} \to \mathit{DFValues}}$ (the transfer function that maps any set of data-flow values to $\emptyset$). Then the solution is iteratively improved until we reach a tuple of transfer functions that satisfies all the constraints.

Since every transfer function appears on the left-hand side of exactly one constraint, the value in the next iteration is updated according to this constraint. For example, if there is a constraint $\phi \sqsupseteq \langle\!\langle v \rangle\!\rangle_{\mathrm{TF}} \rhd (\phi_1 \sqcup \phi_2)$, this update step sets the new estimate of ϕ to exactly $\langle\!\langle v \rangle\!\rangle_{\mathrm{TF}} \rhd (\phi_1' \sqcup \phi_2')$, where ϕ_1' and ϕ_2' are the current estimates of ϕ_1 and ϕ_2. These steps are monotonic with respect to $\sqsubseteq$, and thus this iteration converges to the least fixed-point solution for our finite-height lattices.

3.3 Generalising to other data-flow analyses

Our CFG-based effect system for liveness can be generalised to a single framework capturing the four classical data-flow analyses (live variables, reaching definitions, very busy expressions, available expressions). The generalised form is parameterised by the following algebra:

- a set of data-flow values *DFValues*, effects are then transfer functions drawn from $\mathit{DFValues} \to \mathit{DFValues}$;
- a subeffecting relation $\sqsubseteq$ on transfer functions;
- a sequencing operator $\rhd$ on transfer functions;
- a function $\langle\!\langle \ell : X := e \rangle\!\rangle_{\mathrm{TF}}$ mapping labelled assignment statements to transfer functions;
- a function $\langle\!\langle v \rangle\!\rangle_{\mathrm{TF}}$ mapping a potential variable appearing in $\mathsf{halt}\ v$ or in $\mathsf{if}\ v \geq 0$ to a transfer function representing its being read.

	$\mathit{DFValues}$	$\sqsubseteq$	$\sqcup$	$\rhd$	$\mathit{kill}(\ell : X := e)$	$\mathit{gen}(\ell : X := e)$	$\mathit{vgen}(v)$
LVA	$\mathcal{P}(\mathit{Vars})$	$\subseteq$	$\cup$	$\circ$	$\{X\}$	$\mathit{fv}(e)$	$\mathit{fv}(v)$
RD	$\mathcal{P}(\mathit{Vars} \times \mathit{Labels})$	$\subseteq$	$\cup$	$\hat{\circ}$	$\{(X, l) \mid l \in \mathit{Labels}\}$	$\{(X, \ell)\}$	$\emptyset$
VBE	$\mathcal{P}(\mathit{Expressions})$	$\supseteq$	$\cap$	$\circ$	$\{e' \mid X \in \mathit{fv}(e')\}$	$\{e\}$	$\mathit{fv}(v)$
AVAIL	$\mathcal{P}(\mathit{Expressions})$	$\supseteq$	$\cap$	$\hat{\circ}$	$\{e' \mid X \in \mathit{fv}(e')\}$	$\{e\}$	$\mathit{fv}(v)$

(Recall $v ::= X \mid k$ therefore $\mathit{fv}(v)$ in the rightmost column is either a singleton or empty set.)

Figure 4 CFG effect-system instantiations for classical data-flow analyses.

Sets of transfer functions equipped with $\sqsubseteq$ are lattices, therefore a $\sqcup$ operator (join) exists. For all four classical analysis, $\langle\!\langle \ell : X := e \rangle\!\rangle_{\mathrm{TF}}$ and $\langle\!\langle v \rangle\!\rangle_{\mathrm{TF}}$ can be expressed:

$$\begin{aligned} \langle\!\langle \ell : X := e \rangle\!\rangle_{\mathrm{TF}} &= (\lambda d.\ (d \setminus \mathit{kill}(\ell : X := e)) \cup \mathit{gen}(\ell : X := e)) &&: \mathit{DFValues} \to \mathit{DFValues} \\ \langle\!\langle v \rangle\!\rangle_{\mathrm{TF}} &= (\lambda d.\ d \cup \mathit{vgen}(v)) &&: \mathit{DFValues} \to \mathit{DFValues} \end{aligned}$$

The space of data-flow values $\mathit{DFValues}$ along with its $\sqsubseteq$, $\sqcup$, gen, kill and vgen operators are variously parameterised for the four data-flow analyses as shown in Figure 4. The effect system that then describes all of these is precisely the one given in Figure 3.

In these instantiations, the $\rhd$ operator is particularly interesting. The algebra $(\mathit{DFValues} \to \mathit{DFValues}, \sqsubseteq, \rhd, \mathit{id})$ is a partially ordered monoid, with id as the unit element. We consider two possibilities for $\rhd$ depending on the direction of the analysis

- For backwards analysis, $\rhd$ is function composition $\circ$;
- For forwards analysis, $\rhd$ is reverse function composition $\hat{\circ}$ – defined as $f \mathbin{\hat{\circ}} g \overset{\mathrm{def}}{=} g \circ f$.

3.4 Constant propagation as a non-distributive example

Constant propagation from Section 2.3 (not one of the four classical analyses) also fits into the above framework. We take $\mathit{DFValues}$ to be the lattice of mappings s from variables to $\mathbb{Z}_\bot^\top$ with the $\sqsubseteq$ relation being lifted component-wise. This lattice of transfer functions becomes a pomonoid by taking $\rhd$ to be reverse composition $\hat{\circ}$ (since constant propagation is a forwards analysis). Transfer functions for assignment and variable access are:

$$\langle\!\langle \ell : X := e \rangle\!\rangle_{\mathrm{TF}} = \lambda s.\ s[X \mapsto s(e)] \qquad \text{and} \qquad \langle\!\langle v \rangle\!\rangle_{\mathrm{TF}} = \lambda s.\ s$$

where we abusively write $s(e)$ to mean the value in $\mathbb{Z}_\bot^\top$ obtained by substituting variables in e as specified by s and simplifying. Variable access does not update variables so $\langle\!\langle v \rangle\!\rangle_{\mathrm{TF}} = \mathit{id}$.

As an example, sequencing the effects of $X := 1$ and $Y := X + 2$ gives the effect:

$$\begin{aligned} (\lambda s.\ s[X \mapsto 1]) \mathbin{\hat{\circ}} (\lambda s.\ s[Y \mapsto s(X) + 2]) &= (\lambda s.\ s[Y \mapsto s(X) + 2]) \circ (\lambda s.\ s[X \mapsto 1]) \\ &= \lambda s.\ s[X \mapsto 1, Y \mapsto 3] \end{aligned}$$

Thus the inference system of Figure 3 can be used also for constant propagation.

With constant propagation, $\sqcup$ and $\hat{\circ}$ do not satisfy distributivity, as seen previously in the example of Figure 2. In this algebra, that example illustrates the fact that for:

$$\phi_1 = \lambda s.\ s[X \mapsto 1, Y \mapsto 2] \qquad \phi_2 = \lambda s.\ s[X \mapsto 2, Y \mapsto 1] \qquad \phi_3 = \lambda s.\ s[Z \mapsto s(X) + s(Y)]$$

distributivity is violated – that is, $(\phi_1 \mathbin{\hat{\circ}} \phi_3) \sqcup (\phi_2 \mathbin{\hat{\circ}} \phi_3) \neq (\phi_1 \sqcup \phi_2) \mathbin{\hat{\circ}} \phi_3$. Thus, the idea of basing effect-systems on the distributive structure of quantales (as in [6]) would exclude this common static analysis. We therefore advocate that distributivity is not imposed (Section 5).

4 Translating to a graded monadic setting

We now formulate a graded monadic model of the effect system given in Section 3, exploring the use of graded structures to encode liveness analysis in programming and semantic modelling. We describe translations from our CFGs into a pure functional language (e.g., Haskell, Agda, Coq, or a pure subset of ML).

We briefly overview graded monads in Section 4.1. We go on to define a monadic variant of the McCarthy transformation in Section 4.2; this is generalised to a graded monadic McCarthy transformation in Section 4.3.1. For the graded monadic case, the data-flow equations get represented as typing constraints in the target language's type system. Section 4.3.2 gives a graded monad which further refines a semantic model of state by liveness information. Section 4.4 considers a concrete translation into Haskell, details of which are in Appendix B. Lastly, Section 4.5 explains how to generalise this approach to other data-flow analyses.

4.1 Graded monads

Monads are common in pure functional programming languages (such as Haskell) for embedding and structuring effectful computations [27] and for semantic models of effects [13]. We recall a programming oriented definition: a monad is a triple $(M, \gg\!=, \mathsf{return})$ where M is a type constructor, $\gg\!=$ (bind) is an infix operator, and return is a function, with the types:

$$\mathsf{return} : \forall\alpha.\ \alpha \to M\alpha \qquad\qquad (\gg\!=) : \forall\alpha\forall\beta.\ M\alpha \to (\alpha \to M\beta) \to M\beta$$

Following Moggi [13], we use the word *computation* for values of type $M\tau$, just like we use *function* for values of type $\sigma \to \tau$.

In addition, these operations should satisfy the following axioms:

$$\begin{aligned} m \gg\!= \mathsf{return} &= m && \text{(right identity)} \\ \mathsf{return}\ x \gg\!= f &= f\ x && \text{(left identity)} \\ (m \gg\!= f) \gg\!= g &= m \gg\!= (\lambda x.\ f\ x \gg\!= g) && \text{(associativity)} \end{aligned}$$

Wadler and Thiemann [28] showed that monads and effect systems can be united by transposing effect systems into an equivalent monadic system with effect annotations in types: for an expression of type τ with effect F, there is an monad M annotated with F (written M_F) such that there is an equivalent expression of type $M_F\tau$. This annotated monad represents the possible effects of an impure expression e, described by F. *Graded monads* essentially systematise and generalise this idea so that a model or embedding of the effectful computation may depend on the effect information F, which has some algebraic structure (the *effect algebra*). In this way, graded monads can capture effect information in types (e.g., for fine-grained effect and resource reasoning) or make effect semantics more fine-grained.

Graded monads generalise monads to an indexed family of type constructors whose indices range over elements of a given algebraic structure [4, 9, 15]. The operations of this structure then mediate the operations of the graded monad. The structure of grades is usually a pomonoid $(D, \sqsubseteq, \rhd, 1)$, giving a graded monad $(\{M^r\}_{r\in D}, \mathsf{sub}, \gg\!=, \mathsf{return})$, where $\{M^r\}_{r\in D}$ is a family of type constructors indexed by D-elements and $\gg\!=$, return, and sub have types:

$$\begin{aligned} &\gg\!= &&: \forall r\forall s\forall\alpha\forall\beta.\ M^r\alpha \to (\alpha \to M^s\beta) \to M^{r\rhd s}\beta \\ &\mathsf{return} &&: \forall\alpha.\ \alpha \to M^1\alpha \\ &\mathsf{sub} &&: \forall r\forall s\forall\alpha.\ M^r\alpha \to M^s\alpha && \textit{if } r \sqsubseteq s \end{aligned}$$

Here $\gg\!=$ and sub are polymorphic in types and grades. We use Greek letters for types and Roman letters for elements (grades) of the algebra in order to avoid clutter in type signatures.

A graded monad satisfies axioms analogous to those of a monad, but with the addition of grades in such a way that the graded monad laws depend on the associativity and identity properties of the monoid, where for all $x : \alpha, m : M^r\alpha, f : \alpha \to M^s\beta$ and $g : \beta \to M^t\gamma$:

$$\begin{array}{rcll} m \gg\!=_{r,1} \mathsf{return} & = & m & : M^r\alpha \qquad \text{(right identity)} \\ \mathsf{return}\ x \gg\!=_{1,s} f & = & f\ x & : M^s\beta \qquad \text{(left identity)} \\ (m \gg\!=_{r,s} f) \gg\!=_{r\triangleright s,t} g & = & m \gg\!=_{r,s\triangleright t} (\lambda x.\ f\ x \gg\!=_{s,t} g) & : M^{r\triangleright s\triangleright t}\gamma \qquad \text{(associativity)} \end{array}$$

We subscript the operations with the instantiation of the grades here for clarity.

The sub satisfies the following $\forall r, s, r', s', m : M^r\alpha, f : \alpha \to M^s\beta$ where $r \sqsubseteq r'$ and $s \sqsubseteq s'$:

$$\mathsf{sub}_{r,r'} m \gg\!=_{r',s'} (\mathsf{sub}_{s,s'} \circ f) = \mathsf{sub}_{r\triangleright s, r'\triangleright s'}(m \gg\!=_{r,s} f) : M^{r'\triangleright s'}\beta \qquad \text{(monotonicity)}$$

Categorically, graded monads correspond to lax monoidal functors between a pomonoid (viewed as a category) and a category of endofunctors (essentially type constructors) [9, 20]. This categorical construction embodies the idea that graded monads match the structure of some analysis domain (a pomonoid on D) to the structure of a semantic domain (type constructors modelling computations). The resulting operations ($\gg\!=$, return, sub) propagate the pomonoid structure with them via the grades, describing the structure of a computation.

4.2 Monadic McCarthy transformation

McCarthy's transformation [12] maps CFG statements to mutually recursive function definitions using an m-tuple of functionally updated variables to represent the state. For example, the node ($\ell_1 : Y := X + Z$; goto ℓ_2) in a CFG containing variables X, Y, Z can be translated into the function $f_1(x, y, z) = f_2(x, x + z, z)$ where f_2 is the function corresponding to the CFG node with label ℓ_2. We define a variant, using a monad to represent state, and call it the *monadic McCarthy transformation*.

The standard state monad [27] models a single mutable memory cell with type constructor State α parameterised by the type of values that can be stored α, and two operations for manipulating the state: get : $\forall\alpha.$State α α and put : $\forall\alpha.\alpha \to$ State α Unit. We can thus represent m integer variables by the monad State(Int, . . . , Int) with access to each variable provided by get and put and projections. An alternative is to use a monad transformer stack. For brevity, we instead assume an equivalent monad MultiState which holds the state of m integer variables and has m operations get_i : MultiState Int and put_i : Int $\to$ MultiState Unit one for each CFG variable ($X, Y, Z, \ldots \in$ *Vars*). In examples these are written getX, putY etc. Thus MultiState τ is the type of computations over mutable variables that return type τ.

We use Haskell's do $\{\ldots\}$ notation as syntactic sugar for monadic computations,[4] equivalent to Moggi's monadic metalanguage [13]. For example, do $\{x \leftarrow e_1;\ e_2\}$ sugars $e_1 \gg\!= (\lambda x.\ e_2)$, and do $\{e_1;\ e_2\}$ sugars $e_1 \gg\!= (\lambda\_.\ e_2)$. The desugaring is recursively applied.

Our monadic McCarthy transformation produces a set of mutually recursive definitions of computation values of monadic type (in our case MultiState Int) instead of a set of mutually recursive functions. Each labelled statement $(\ell : S_\ell)$ maps to a definition $[\![\ell : S_\ell]\!]_{\mathrm{MM}}$ as specified in Figure 5: for assignment $X := e$, the variables of e are read into temporary (pure) variables using get, followed by a putX to write to X; variables read within conditionals and halts are treated similarly. The resultant monadic definitions can be directly read as a Haskell program (or an ML program after desugaring into recursive function definitions).

[4] There is an additional assumption of monad *strength* which allows monadic computations to close over variables in scope. Strength holds for all monads in Cartesian-closed categories (and in programming settings). The notion of strength extends to graded monads [9] and is provided for all our examples.

$\ell : S$	$[\![\ell : S]\!]_{\mathrm{MM}}$
$\ell : X := Y + Z;\ \mathsf{goto}\ \ell'$	$g_\ell = \mathsf{do}\ \{\ y \leftarrow \mathsf{getY};\ z \leftarrow \mathsf{getZ};\ \mathsf{putX}\ (y+z);\ g_{\ell'}\ \}$
$\ell : X := k;\ \mathsf{goto}\ \ell'$	$g_\ell = \mathsf{do}\ \{\ \mathsf{putX}\ k;\ g_{\ell'}\ \}$
$\ell : \mathsf{if}\ X \geq 0\ \mathsf{then\ goto}\ \ell'\ \mathsf{else\ goto}\ \ell''$	$g_\ell = \mathsf{do}\ \{\ x \leftarrow \mathsf{getX};\ \mathsf{if}\ x \geq 0\ \mathsf{then}\ g_{\ell'}\ \mathsf{else}\ g_{\ell''}\ \}$
$\ell : \mathsf{halt}\ X$	$g_\ell = \mathsf{do}\ \{\ x \leftarrow \mathsf{getX};\ \mathsf{return}\ x\ \}$

Figure 5 The monadic McCarthy transformation $[\![-]\!]_{\mathrm{MM}}$. For assignment, we only give the cases $X := k$ and $X := Y + Z$; other cases, e.g., $X := Y + 1$, are similar. Conditional and halt forms that have k (constants) instead of variables X are analogous.

$$\begin{array}{lcl}
\ell_0 : X := 100;\ \mathsf{goto}\ \ell_1 & & g_0 = \mathsf{do}\ \{\ \mathsf{putX}\ 100;\ g_1\ \} \\
\ell_1 : \mathsf{if}\ X \geq 0\ \mathsf{then\ goto}\ \ell_2\ \mathsf{else\ goto}\ \ell_4 & & g_1 = \mathsf{do}\ \{\ x \leftarrow \mathsf{getX};\ \mathsf{if}\ x \geq 0\ \mathsf{then}\ g_2\ \mathsf{else}\ g_4\ \} \\
\ell_2 : X := X - 1;\ \mathsf{goto}\ \ell_3 & & g_2 = \mathsf{do}\ \{\ x \leftarrow \mathsf{getX};\ \mathsf{putX}\ (x-1);\ g_3\ \} \\
\ell_3 : Y := Y + 1;\ \mathsf{goto}\ \ell_1 & \xrightarrow{[\![-]\!]_{\mathrm{MM}}} & g_3 = \mathsf{do}\ \{\ y \leftarrow \mathsf{getY};\ \mathsf{putY}\ (y+1);\ g_1\ \} \\
\ell_4 : R := Y + Z;\ \mathsf{goto}\ \ell_5 & & g_4 = \mathsf{do}\ \{\ y \leftarrow \mathsf{getY};\ z \leftarrow \mathsf{getZ};\ \mathsf{putR}\ (y+z);\ g_5\ \} \\
\ell_5 : \mathsf{halt}\ R & & g_5 = \mathsf{do}\ \{\ r \leftarrow \mathsf{getR};\ \mathsf{return}\ r\ \}
\end{array}$$

Figure 6 Example monadic McCarthy transformation. CFG code (left) is translated into mutually recursive definitions of computation values (right).

Figure 6 exemplifies the monadic McCarthy transformation converting an imperative program (left) to a set of mutually recursive computation definitions (right).

The monadic McCarthy transformation produces a program with equivalent behaviour to the original CFG (by a straightforward refactoring of McCarthy's transformation into the state monad). Next, we show that a more refined model can be given by targeting a *graded monad* instead of a monad. This allows the target of the translation to capture the same data-flow information as the CFG effect system's judgements.

4.3 Graded monadic McCarthy transformation for liveness

The above monadic McCarthy transformation maps CFG terms to state monad computations, i.e., $[\![l : S]\!]_{\mathrm{MM}} : \mathsf{MultiState}\ \mathsf{Int}$. Instead, given a graded monad $\mathsf{MultiState}^{\phi}$ which provides state monad-like behaviour (graded by our pomonoid of transfer functions ϕ), we give a *graded monadic* McCarthy transformation $[\![\ell : S]\!]_{\mathrm{GM}} : \mathsf{MultiState}^{\Phi(\ell)}\ \tau$ whenever $\Phi \vdash (\ell : S) : \tau \,\&\, \Phi(\ell)$.

We describe this graded monadic McCarthy transformation (Section 4.3.1) by first taking the usual MultiState monad and wrapping into a trivial graded monad: one whose grades only decorate the types but do not affect the operations and thus have no semantic meaning. We then replace this graded monad with one whose grades have semantic meaning, refining the types and operations of the former to give a semantic account of liveness (Section 4.3.2).

4.3.1 Transformation to a trivial graded monad

Given a monad M and a pomonoid $(D, \sqsubseteq, \rhd, 1)$ one can construct a *trivial* graded monad with type constructors $M^{d}_{\mathsf{triv}}\tau = M\tau$ for $d \in D$. In this construction M^{d}_{triv} simply wraps M and thus the grades have no bearing on the computation encoded by the monad. The monad operations of M provide the graded monad operations of M_{triv} via this wrapping, and the required graded monad laws follow from the laws of the monoid $(D, \sqsubseteq, \rhd, 1)$ and monad M.

We use this construction on the MultiState monad to form a graded monad written $\mathsf{MultiState}^{\phi}_{\mathsf{triv}}$ graded by the pomonoid of transfer functions $(\mathit{DFValues} \to \mathit{DFValues}, \sqsubseteq, \rhd, \mathit{id})$ from Section 3.1. Thus a value of type $\mathsf{MultiState}^{\phi}_{\mathsf{triv}}\ \tau$ is a stateful computation that returns a value of type τ with some transfer-function grade ϕ associated to it by its operations.

The graded monadic McCarthy transformation $[\![-]\!]_{\mathrm{GM}}$ enriches $[\![-]\!]_{\mathrm{MM}}$ (Fig. 5) in only two ways: (i) applying sub to both $g_{\ell'}$ and $g_{\ell''}$ in the translation of if, and (ii) using the *graded* monad operations below in the body of do (and for $\gg\!=$ when desugaring it):

$$\begin{array}{lcl}
\mathsf{getX} & : & \mathsf{MultiState}^{\mathit{gen}_X}_{\mathsf{triv}}\ \mathsf{Int} \qquad\qquad \text{where} \quad \mathit{gen}_X \overset{\mathrm{def}}{=} \lambda d.\ d \cup \{X\} \\
\mathsf{putX} & : & \mathsf{Int} \to \mathsf{MultiState}^{\mathit{kill}_X}_{\mathsf{triv}}\ \mathsf{Unit} \qquad\qquad \mathit{kill}_X \overset{\mathrm{def}}{=} \lambda d.\ d \setminus \{X\} \\
\mathsf{return} & : & \forall \alpha.\alpha \to \mathsf{MultiState}^{\mathit{id}}_{\mathsf{triv}}\ \alpha \\
\gg\!= & : & \forall \phi, \phi', \alpha, \beta.\ \mathsf{MultiState}^{\phi}_{\mathsf{triv}}\ \alpha \to (\alpha \to \mathsf{MultiState}^{\phi'}_{\mathsf{triv}}\ \beta) \to \mathsf{MultiState}^{\phi \rhd \phi'}_{\mathsf{triv}}\ \beta
\end{array}$$

Since the transformation operates on the syntax of CFGs, rather than judgements of the CFG effect system, the grades on each computation type must be inferred by generating a set of typing constraints which are then solved (as was done in Section 3.2), by the host language's type system (we consider the feasibility of this in Section 4.4).

The syntactic translation results in graded monadic computations whose grades match exactly the analysis of our CFG effect-system from Section 3:

▶ **Lemma 2** (Soundness of the graded monadic McCarthy transformation)**.** *If* $\Phi \vdash (\ell : S) : \tau \& \Phi(\ell)$ *and* $\forall \ell' \in \mathsf{dom}(\Phi).(g_{\ell'} : \mathsf{MultiState}^{\Phi(\ell')}_{\mathsf{triv}}\ \mathsf{Int})$ *then* $[\![\ell : S]\!]_{GM} : \mathsf{MultiState}^{\Phi(\ell)}_{\mathsf{triv}}\ \tau$*.*

▶ **Example 3.** Let g, of type $\mathsf{MultiState}^{\phi}_{\mathsf{triv}}\ \mathsf{Int}$, represent a liveness transfer function 'for the rest of the program'. Now consider the following expression (effectively prefixing g with the statement $Z := X + Y$ and applying the graded McCarthy transformation $[\![-]\!]_{\mathrm{GM}}$ above):

$$\mathsf{do}\ \{\quad x \leftarrow \mathsf{getX};\ \ y \leftarrow \mathsf{getY};\ \ \mathsf{putZ}\ (x+y);\ \ g \quad\}$$

By construction, its type is $\mathsf{MultiState}^{\phi'}_{\mathsf{triv}}\ \mathsf{Int}$ where $\phi' = \lambda d.\ (\phi(d) \setminus \{Z\}) \cup \{X, Y\}$ represents the liveness transfer function for $Z := X + Y$ followed by the 'rest of the program' because

$$\phi' = \mathit{gen}_X \rhd \mathit{gen}_Y \rhd \mathit{kill}_Z \rhd \phi = \lambda d.\ (\phi(d) \setminus \{Z\}) \cup \{X, Y\}$$

As in Section 3.1, $\phi'(\emptyset)$ gives us the liveness information the start of the 'body' of do {} because the boundary information is that the set of live variables is empty at program exit.

▶ **Example 4.** In Figure 6, we converted an imperative program into mutually recursive computation values $g_0, \ldots, g_5$. Let $\phi_0, \ldots, \phi_5$ stand for the transfer function grades of the graded monadic types of $g_0, \ldots, g_5$, so that g_i is of type $\mathsf{MultiState}^{\phi_i}_{\mathsf{triv}}\ \mathsf{Int}$. Then these transfer functions must satisfy the following constraints (coming from the type system):

$$\begin{array}{lll lll lll}
\phi_0 & \sqsupseteq & \mathit{kill}_X \rhd \phi_1 & \phi_1 & \sqsupseteq & \mathit{gen}_X \rhd \phi_2 & \phi_1 & \sqsupseteq & \mathit{gen}_X \rhd \phi_4 \\
\phi_2 & \sqsupseteq & \mathit{gen}_X \rhd \mathit{kill}_X \rhd \phi_3 & \phi_3 & \sqsupseteq & \mathit{gen}_Y \rhd \mathit{kill}_Y \rhd \phi_1 & & & \\
\phi_4 & \sqsupseteq & \mathit{gen}_Y \rhd \mathit{gen}_Z \rhd \mathit{kill}_R \rhd \phi_5 & \phi_5 & \sqsupseteq & \mathit{gen}_R \rhd \mathit{id} & & &
\end{array}$$

The usual fixed-point iteration gives the principal (least) solution: $\phi_0 = \lambda d.(d \setminus \{X, R\}) \cup \{Y, Z\}$, $\phi_1 = \phi_2 = \phi_3 = \phi_4 = \lambda d.\ (d \setminus \{R\}) \cup \{X, Y, Z\}$ and $\phi_5 = \lambda d.\ d \cup \{R\}$. The set of live variables at the program start (i.e. at g_0) is therefore $\phi_0(\emptyset) = \{Y, Z\}$.

4.3.2 Analysis-directed semantics via grade-based refinement

The previous section constructed the graded monad $\mathsf{MultiState}^{\phi}_{\mathsf{triv}}\,\tau$ as a simple wrapper over the usual state monad; grade ϕ was a transfer function but it had no semantic meaning: the grades were merely decorations on types and did not affect the operations. We can instead use grades to *refine* the types and operations of the usual state monad by the liveness information, so that graded monad operations actually depend on the grades. In this case, refinement means restricting stores to subsets of the variables involved in a program. This paves the way to ensuring that only semantically valid analyses can be encoded as grades. The translation $[\![\ell : S]\!]_{\mathrm{GM}}$ remains the same but we instead replace the operations of $\mathsf{MultiState}^{\phi}_{\mathsf{triv}}$ with the operations of a new graded monad $\mathsf{MultiState}^{\phi}$ (such that Lemma 2 holds for $\mathsf{MultiState}^{\phi}$).

Previously, $\mathsf{MultiState}$ and $\mathsf{MultiState}^{\phi}_{\mathsf{triv}}\,\tau$ represented their stores as m-tuples of Ints where $m = |\,\mathit{Vars}|$ (the CFG variables), i.e. $\mathsf{MultiState}\,\tau = (\mathsf{Int}, \ldots, \mathsf{Int}) \to \tau \times (\mathsf{Int}, \ldots, \mathsf{Int})$. Now, given any subset $V \subseteq \mathit{Vars}$, we define the *$V$-refined store* $\mathsf{Store}(V)$ to be $V \to \mathsf{Int}$, writing $\hat{\emptyset}$ for the only member of $\mathsf{Store}(\emptyset)$. Note, $\mathsf{Store}(\mathit{Vars})$ recovers (up to isomorphism) the previous $(\mathsf{Int}, \ldots, \mathsf{Int})$.

We now define $\mathsf{MultiState}^{\phi}$ whose input and output stores are computed from ϕ:

$$\mathsf{MultiState}^{\phi}\tau = \mathsf{Store}(\mathit{reads}(\phi)) \to \tau \times \mathsf{Store}(\mathit{footprint}(\phi))$$

For the input store, $\mathit{reads}(\phi) = \phi(\emptyset)$ gives us the subset of variables which are live-in and thus read by a computation of type $\mathsf{MultiState}^{\phi}\tau$. For the output store, $\mathit{footprint}(\phi) = \phi(\emptyset) \cup (\mathit{Vars} \setminus \phi(\mathit{Vars}))$ gives the *footprint* (borrowing terminology from separation logic [18]) containing those variables read or written by this computation.[5] For example, $\mathsf{do}\ \{x \leftarrow \mathsf{getX};\ y \leftarrow \mathsf{getY};\ \mathsf{putZ}\ (x+y);\}$ has grade $\phi = \mathit{gen}_X \rhd \mathit{gen}_Y \rhd \mathit{kill}_Z = \lambda d.\ (d \setminus \{Z\}) \cup \{X, Y\}$ (akin to Example 3) and thus $\mathit{reads}(\phi) = \phi(\emptyset) = \{X, Y\}$ and $\mathit{footprint}(\phi) = \{X, Y, Z\}$.

The $\mathsf{MultiState}^{\phi}$ type is a graded monad with refined return and state operations:

$$\begin{array}{llll}
\mathsf{return} : \forall \alpha.\alpha \to \mathsf{MultiState}^{\mathit{id}}\alpha & = \lambda x.\lambda s.(x, \hat{\emptyset}) & : \forall \alpha.\alpha \to (\mathsf{Store}(\emptyset) \to \alpha \times \mathsf{Store}(\emptyset)) \\
\mathsf{getX} : \mathsf{MultiState}^{\mathit{gen}_X}\,\mathsf{Int} & = \lambda s.(s(X), s) & : \mathsf{Store}(\{X\}) \to \mathsf{Int} \times \mathsf{Store}(\{X\}) \\
\mathsf{putX} : \mathsf{Int} \to \mathsf{MultiState}^{\mathit{kill}_X}\,\mathsf{Unit} & = \lambda x.\lambda s.((), [X \mapsto x]) & : \mathsf{Int} \to (\mathsf{Store}(\emptyset) \to \mathsf{Unit} \times \mathsf{Store}(\{X\}))
\end{array}$$

On the right, we repeat the type of the operations, expanding the definition of $\mathsf{MultiState}^{\phi}$. The input and output stores of return are both the empty map as $\mathit{reads}(\mathit{id}) = \mathit{footprint}(\mathit{id}) = \emptyset$ representing that no variables are read or written by return. Thus, return represents a pure computation as isomorphic to the identity. The getX and putX operators are similarly refined.

The graded monad $\gg\!=$ resembles the usual state monad $\gg\!=$ but with three auxiliary operations ($\blacktriangleleft$, $\lhd$, and $\downarrow$ below) to manage the variously refined stores:

$$\begin{array}{rll}
\gg\!= & : \forall \phi, \phi', \alpha, \beta.\ \mathsf{MultiState}^{\phi}\alpha \to (\alpha \to \mathsf{MultiState}^{\phi'}\beta) \to \mathsf{MultiState}^{\phi \rhd \phi'}\beta & \\
& = \lambda m.\lambda f.\lambda s.\ \mathsf{let}\ (a, s') = m(\downarrow_{\phi,\phi'} s)\ \mathsf{in} & \\
& \qquad\qquad \mathsf{let}\ (b, s'') = (f\ a)(s \blacktriangleleft_{\phi,\phi'} s')\ \mathsf{in}\ (b,\ s' \lhd_{\phi,\phi'} s'') & \\
\mathit{where}\ \downarrow_{\phi,\phi'} & : \mathsf{Store}(\mathit{reads}(\phi \rhd \phi')) \to \mathsf{Store}(\mathit{reads}(\phi)) & (\mathit{restrict}) \\
\blacktriangleleft_{\phi,\phi'} & : \mathsf{Store}(\mathit{reads}(\phi \rhd \phi')) \times \mathsf{Store}(\mathit{footprint}(\phi)) \to \mathsf{Store}(\mathit{reads}(\phi')) & (\mathit{merge}_1) \\
\lhd_{\phi,\phi'} & : \mathsf{Store}(\mathit{footprint}(\phi)) \times \mathsf{Store}(\mathit{footprint}(\phi')) \to \mathsf{Store}(\mathit{footprint}(\phi \rhd \phi')) & (\mathit{merge}_2)
\end{array}$$

Here $\downarrow_{\phi,\phi'} s$ restricts the incoming store $s : \mathsf{Store}(\mathit{reads}(\phi \rhd \phi'))$ to $\mathsf{Store}(\mathit{reads}(\phi))$, i.e., just those variables live in computation $m : \mathsf{MultiState}^{\phi}\alpha$. The operation $s \blacktriangleleft_{\phi,\phi'} s'$ pads the

[5] A more refined graded state monad would return an output store containing only those variables that are written-to (e.g. as in [15]). However, liveness analysis alone does not allow us to compute just the variables written-to. The footprint is therefore a safe over-approximation of the written-to set.

domain of store $s' : \mathsf{Store}(\mathit{footprint}(\phi))$ (resulting from m) with the variables in store s, to produce a store whose domain is just the live variables required for computation $(f\ a)$. The $s' \lhd_{\phi,\phi'} s''$ operation similarly pads the domain of store $s'' : \mathsf{Store}(\mathit{footprint}(\phi'))$ (resulting from $(f\ a)$) with the variables in store s' to give the final updated store its required domain.

The resulting $\gg\!=$ operation thus 'filters' input stores by what is live, and output stores by the footprint enabling, e.g., soundness of dead-code removal to be proved (future work). The usual state monad $\gg\!=$ is recovered by redefining $\downarrow_{\phi,\phi'} s = s$ and $s \blacktriangleleft_{\phi,\phi'} s' = s \lhd_{\phi,\phi'} s' = s'$.

Appendix C provides more details and the proof that this is indeed a graded monad. For brevity, we omit the definition of sub.

4.4 Targeting a host language and applications

We have used Haskell-like do-notation as syntactic sugar for the (graded) monad operations in some functional language. We can take this a step further, concretely targeting GHC/Haskell, leveraging its combination of a practical functional language with an advanced type system. Appendix B gives more details, showing how the Section 4.3.1 can be captured in Haskell.

This approach works well for sequential code, but reaches its limits with branching and recursion as GHC does not have an appropriate notion of subtyping nor can it compute fixed-points of type equations. A system with subtyping and equirecursive types (e.g., OCaml) may fare better. An alternate approach is to make the graded monadic McCarthy transformation not just syntax directed but *type-and-effect directed*. In this approach, solutions to the data-flow equations can be computed (e.g, by work-list algorithm) *before* applying the graded monadic McCarthy transformation. The resulting (least) transfer functions can then be used in the translation to specialise the types of the resultant graded monadic program.

Whilst Haskell is shown as a target here, our approach is likely to be more useful in the setting of a proof assistant when formalising language semantics or a compiler and its optimisations (e.g., the CakeML verified compiler [25]).

4.5 Generalising to other data-flow analyses

So far we focused on liveness, where assignment statements are decomposed into sequences of get and put operations. For other data-flow analyses, we cannot perform the same translation of assignment as it may not be similarly decomposable. For example, for available expressions we cannot associate *kill* with put nor *gen* with get. To capture these other data-flow analyses, we can parameterise our graded monadic McCarthy transformation by a specialised interpretation for assignments $[\ell : x := e]_{\mathrm{GM}} : M^{\langle\!\langle \ell : x := e \rangle\!\rangle_{\mathrm{TF}}}\mathsf{Unit}$, graded by the assignment transfer function. The translation is then the same as Section 4.3, but with assignments translated as:

$$[\![\ell : x := e;\ \mathsf{goto}\ \ell']\!]_{\mathrm{GM}} = \mathsf{do}\,\{\,[\ell : x := e]_{\mathrm{GM}};\ g_{\ell'}\,\}$$

We then require that $[\ell : x := e]_{\mathrm{GM}}$ simulates the behaviour of assignment in the non-graded monadic McCarthy transformation $[\![-]\!]_{\mathrm{MM}}$ on the MultiState monad. Using this generalisation for different analyses with specialised graded monads akin to Section 4.3.2 is further work.

5 Conclusions and discussion

We demonstrated that a type-and-effect system based on transfer functions can be used to compute data-flow values at any point in a CFG, and in particular can be used for liveness analysis. Furthermore we have shown that the McCarthy transformation can be adapted into a (graded) monadic form which embeds live variable analysis using control-flow graphs into functional programs, where transfer functions are grades of a graded monad.

This not only unifies two separately developed fields, but also contributes to the evolving discussion of "what properties do we expect of the effect algebras used as grades". In particular, it shows that the distributivity axiom posited for effect quantales is over-restrictive in that it does not allow representation of non-distributive data-flow problems such as constant propagation. Using a pomonoid (or even pre-ordered monoid as originally phrased by Katsumata) seems to impose *minimal requirements* on a model and so is most general. However, it then admits partial orders in which there is no concept of a least (or principal) solution and which do not seem to model any known static analysis. We suggest an appropriate model should be a pomonoid $(D, \sqsubseteq, \triangleright)$ which satisfies the following requirements:

- $\triangleright$ is monotonic w.r.t. $\sqsubseteq$ (following Katsumata and allowing distributivity);
- D is *bounded complete*: whenever set $X \subseteq D$ has some upper bound then it has a least upper bound.

An advantage of our graded-monadic approach compared to classical data-flow analysis is the potential for a *correct-by-construction property*; correctness of an analysis can be established at the semantic level, either denotationally or by showing a graded-type-preservation property in a reduction-style operational semantics. Correctness then follows from a number of results:

1. that live-variable analysis is achieved by fixed-point calculation over equations on transfer functions (Theorem 1);
2. soundness of the McCarthy transformation (established in [12]); soundness of replacing McCarthy's explicit state passing with the state monad (well-known); and soundness of our novel transformation to a graded state monad (Lemma 2);
3. that our graded monad $\mathsf{MultiState}^{\phi}$ really is a graded monad (Section 4.3.2 / Appendix C);
4. that reduction in our (graded monad) calculus exhibits progress and preservation.

The last point is the subject of future work, which we wish to explore in the context of general data-flow analyses and proving the correctness of program transformations.

Related work. Benton et al. [2] use a graded-monad-based effect system to model non-determinism in an otherwise pure functional language and then use this information in a logical relation semantics to prove program transformations correct, whereas our focus is on embedding general data-flow analyses for imperative languages into graded monads.

Dijkstra monads [23] are a generalisation of monads used for verifying program conditions, where the annotation carries the precondition and postcondition of an expression. While more general, they achieve their full power in a dependently typed language. By contrast, we manage to get far in a graded monadic setting without dependent types.

Further work. As discussed in Section 4.5, further work is to study the graded monadic McCarthy approach in more detail for analyses other than liveness, which was our focus here.

The current work embeds *intra-procedural* program analyses on control-flow graphs as grading inference problems in graded-monadic forms of effect systems. Further work might include showing how notions from inter-procedural analysis, such as context-sensitivity and the IDE and IFDS frameworks of Reps et al. [21], along with how notions such as bidirectional analysis fit into the "properties as grades of a graded monad" model.

Section 4.4 discussed how the graded monadic embedding is likely to be most useful in the setting of verifying optimising compilers (rather than, say, general Haskell programming). Exploring our approach in this context is future work.

References

1 Torben Amtoft, Hanne Riis Nielson, and Flemming Nielson. *Type and effect systems – behaviours for concurrency.* Imperial College Press, 1999.

2 Nick Benton, Andrew Kennedy, Martin Hofmann, and Vivek Nigam. Counting successes: Effects and transformations for non-deterministic programs. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 56–72. Springer International Publishing, 2016. doi:10.1007/978-3-319-30936-1_3.

3 Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992. doi:10.1093/logcom/2.4.511.

4 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.

5 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 1986, August 4-6, 1986, Cambridge, Massachusetts, USA.*, pages 28–38. ACM, 1986. URL: https://dl.acm.org/citation.cfm?id=319838.

6 Colin S. Gordon. A Generic Approach to Flow-Sensitive Polymorphic Effects. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:31, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2017.13.

7 Pierre Jouvelot and David K Gifford. *Communication effects for message-based concurrency.* Laboratory for Computer Science, Massachusetts Institute of Technology, 1989.

8 John B. Kam and Jeffrey D. Ullman. Monotone Data Flow Analysis Frameworks. *Acta Inf.*, 7:305–317, 1977. doi:10.1007/BF00290339.

9 Shin-ya Katsumata. Parametric Effect Monads and Semantics of Effect Systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 633–645. ACM, 2014. doi:10.1145/2535838.2535846.

10 Uday P. Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data Flow Analysis – Theory and Practice.* CRC Press, 2009. URL: http://www.crcpress.com/product/isbn/9780849328800.

11 Peeter Laud, Tarmo Uustalu, and Varmo Vene. Type systems equivalent to data-flow analyses for imperative languages. *Theoretical Computer Science*, 364(3):292–310, 2006. Applied Semantics. doi:10.1016/j.tcs.2006.08.013.

12 John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960. doi:10.1145/367177.367199.

13 Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991. doi:10.1016/0890-5401(91)90052-4.

14 Steven Muchnick and Neil Jones. *Program Flow Analysis: Theory and Applications.* New York University, Courant Institute of Mathematical Sciences, Computer Science Department, January 1981.

15 Alan Mycroft, Dominic A. Orchard, and Tomas Petricek. Effect Systems Revisited - Control-Flow Algebra and Semantics. In Christian W. Probst, Chris Hankin, and René Rydhof Hansen, editors, *Semantics, Logics, and Calculi – Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays*, volume 9560 of *Lecture Notes in Computer Science*, pages 1–32. Springer, 2016. doi:10.1007/978-3-319-27810-0_1.

16 Flemming Nielson, Patrick Cousot, Mads Dam, Pierpaolo Degano, Pierre Jouvelot, Alan Mycroft, and Bent Thomsen. Logical and operational methods in the analysis of programs and systems. In *LOMAPS workshop on Analysis and Verification of Multiple-Agent Languages*, pages 1–21. Springer, 1996. doi:10.1007/3-540-62503-8_1.

17 Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010.

18 Peter W. O'Hearn. A Primer on Separation Logic (and Automatic Program Verification and Analysis). In Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann, editors, *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 286–318. IOS Press, 2012. `doi:10.3233/978-1-61499-028-4-286`.

19 Dominic Orchard and Tomas Petricek. Embedding Effect Systems in Haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 13–24, New York, NY, USA, 2014. ACM. `doi:10.1145/2633357.2633368`.

20 Dominic A. Orchard, Tomas Petricek, and Alan Mycroft. The semantic marriage of monads and effects. *CoRR*, abs/1401.5391, 2014. `arXiv:1401.5391`.

21 Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In Ron K. Cytron and Peter Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23–25, 1995*, pages 49–61. ACM Press, 1995. `doi:10.1145/199448.199462`.

22 Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven Muchnick and Neil Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7. New York University, Courant Institute of Mathematical Sciences, Computer Science Department, January 1981.

23 Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the Dijkstra monad. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 387–398. ACM, 2013. `doi:10.1145/2491956.2491978`.

24 Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and computation*, 111(2):245–296, 1994.

25 Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. A new verified compiler backend for CakeMl. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 60–73, 2016. `doi:10.1145/2951913.2951924`.

26 Ross Tate. The Sequential Semantics of Producer Effect Systems. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 15–26, New York, NY, USA, 2013. ACM. `doi:10.1145/2429069.2429074`.

27 Philip Wadler. The essence of functional programming. In Ravi Sethi, editor, *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, pages 1–14. ACM Press, 1992. `doi:10.1145/143165.143169`.

28 Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Log.*, 4(1):1–32, 2003. `doi:10.1145/601775.601776`.

A Fixed points and transfer functions

Computing transfer functions allows us to compute data-flow values at program points in imperative programs, as we now show.

Let $(N, E, \mathcal{L})$ be a CFG. We consider backwards analyses (forwards ones are similar). For all $n \in N$ there are two associated data-flow values: $DF_{In}(n)$ and $DF_{Out}(n)$, which we here abbreviate to d_n^{In} and d_n^{Out}. Writing ϕ_n for the transfer function for node n, then d_n^{In} satisfies $d_n^{In} = \phi_n(d_n^{Out})$. We also have an equation on d_n^{Out}: if n is an exit node (i.e. no successors), then $d_n^{Out} = \mathrm{BI}$, where BI represents the boundary information ($\emptyset$ both in the cases of liveness and VBE). If n has a non-empty set of successors, then: $d_n^{Out} = \bigsqcup_{s \in succ(n)} d_s^{In}$

Thus for every node in N we have two data-flow equations, say k in total (where $k = 2|N|$). Let $\vec{d}$ be the k-length vector of all data-flow values associated with the nodes, where d_i is its i^{th} component. The data-flow equations give rise to a k-length vector $\vec{h}$ of monotonic functions in $\mathit{DFValues}^k \to \mathit{DFValues}$ such that $d_i = h_i(\vec{d})$, for $1 \leq i \leq k$. The solution of the data-flow equations is the least vector $\vec{d}$ that satisfies these equations. Starting from a k-length vector $(\bot, \dots, \bot)$, we eventually get to the least solution, given by:

$$\vec{d} = \mathit{fix}\ (\lambda \vec{d'}.\ (h_1(\vec{d'}), \dots, h_k(\vec{d'})))$$

where $\mathit{fix}(f)$ is the least fixed point of a function f.

By contrast, our approach in this paper is to find monotonic (transfer) functions $H_i : \mathit{DFValues} \to \mathit{DFValues}$ (i.e., on single data-flow values) per node that satisfy $d_i = H_i(\mathrm{BI})$. In this case, we have a function $\hat{H} = \lambda d.\ \langle H_1(d), \dots, H_k(d)\rangle$ that maps to a tuple of data-flow values, given by the least fixed point:

$$\hat{H} = \mathit{fix}(\lambda \hat{H}'.\ \lambda d.\ \langle h_1(H_1'(d), \dots, H_k'(d)), \dots, h_k(H_1'(d), \dots, H_k'(d))\rangle)$$

In this paper, the effects correspond to a subset of the H_i functions (either just the incoming ones or the outgoing ones), which end up being transfer functions of the *continuations* (or "from this point on in the CFG") – we can refer to them as *cumulative* transfer functions. The least fixed point $\hat{H}$ in this expression satisfies $d_i = H_i(\mathrm{BI})$ by definition of fixed points. As $d_i = h_i(\vec{d})$ as well, the two approaches give the same result by least fixed point uniqueness.

The main application of these facts is in the proof of Theorem 1. By the conventional definition of live variables, the data-flow equations for the CFG language in this paper are:

$$\begin{array}{lcl} \mathit{Live}(\ell : \mathsf{halt}\ v) & = & \mathit{fv}(v) \\ \mathit{Live}(\ell : \mathsf{if}\ v \geq 0\ \mathsf{then}\ \mathsf{goto}\ \ell_1\ \mathsf{else}\ \mathsf{goto}\ \ell_2) & = & \mathit{fv}(v) \cup \mathit{Live}(\ell_1 : \mathcal{L}(\ell_1)) \cup \mathit{Live}(\ell_2 : \mathcal{L}(\ell_2)) \\ \mathit{Live}(\ell : X := e;\ \mathsf{goto}\ \ell') & = & \langle\!\langle \ell : X := e \rangle\!\rangle_{\mathrm{TF}}(\mathit{Live}(\ell' : \mathcal{L}(\ell'))) \end{array}$$

where $\mathit{Live}(\ell : S)$ is the set of live variables at label ℓ corresponding to statement S and $\mathit{fv}(v)$ is the set of free variables in v. This definition is recursive; the set $\widehat{\mathit{Live}}$ corresponds to its least solution in the partial order of sets (that is, subsets of the set of all variables in a CFG).

Proof of Theorem 1. Let there be n instructions labelled $\ell_1, \dots, \ell_n$ in the CFG. We want to show that $\hat{\Phi}(\ell)(\emptyset) = \widehat{\mathit{Live}}(\ell : \mathcal{L}(\ell))$ hold for all ℓ. For any label ℓ, the exact link between $\hat{\Phi}(\ell)$ and all the other $\hat{\Phi}(\ell')$ depends on what exactly the instruction at ℓ is.

- For $\mathcal{L}(\ell) = \mathsf{halt}\ v$, we have $\hat{\Phi}(\ell) = \langle\!\langle v \rangle\!\rangle_{\mathrm{TF}} = \lambda d.\ d \cup \mathit{fv}(v)$.
- For $\mathcal{L}(\ell) = \mathsf{if}\ v \geq 0\ \mathsf{then}\ \mathsf{goto}\ \ell'\ \mathsf{else}\ \mathsf{goto}\ \ell''$, we have $\hat{\Phi}(\ell) = \langle\!\langle v \rangle\!\rangle_{\mathrm{TF}} \rhd (\hat{\Phi}(\ell') \sqcup \hat{\Phi}(\ell''))$, that is, $\hat{\Phi}(\ell) = \lambda d.\ \mathit{fv}(d) \cup \hat{\Phi}(\ell')(d) \cup \hat{\Phi}(\ell'')(d)$.
- For $\mathcal{L}(\ell) = X := e;\ \mathsf{goto}\ \ell'$, we get $\hat{\Phi}(\ell) = \langle\!\langle \ell : X := e \rangle\!\rangle_{\mathrm{TF}} \rhd \hat{\Phi}(\ell')$, that is, we have $\hat{\Phi}(\ell) = \lambda d.\ \langle\!\langle \ell : X := e \rangle\!\rangle_{\mathrm{TF}}(\hat{\Phi}(\ell')(d))$.

Looking at the previous discussion, $\mathcal{L}(\ell)$ corresponds exactly to $\hat{H}$ as it is also a least solution. Similarly, the expressions for $\widehat{\mathit{Live}}$ are analogous to $\vec{d}$ when looking at the vector given by $\widehat{\mathit{Live}}(\ell_1 : \mathcal{L}(\ell_1)), \dots, \widehat{\mathit{Live}}(\ell_n : \mathcal{L}(\ell_n))$. The empty set is the bottom element of the data-flow lattice for liveness, so $\mathrm{BI} = \emptyset$. Thus $\widehat{\mathit{Live}}(\ell_i : \mathcal{L}(\ell_i))(\emptyset) = \hat{\Phi}(\ell_i)$ for all $1 \leq i \leq n$. ◀

B Haskell embedding

Modern Haskell as provided by the Glasgow Haskell Compiler (GHC) (from at least version 8.2 onwards) can embed our graded monads with transfer-function effect algebras in its types, leveraging our graded monadic McCarthy transformation. Graded monads can be captured via the following type class which uses type families to provide the grading pomonoid (based on the `effect-monad` package[6] by Orchard et al. [19]):

```
import Prelude hiding (Monad(..)) -- hide regular monads and then...
import qualified Prelude as M     -- ...import as qualified to wrap monads later

class GradedMonad (m :: d → * → *) where     -- Pomonoid graded monads
   type Unit m :: d                           -- {Type-level monoid providing the
   type Seq m (r :: d) (s :: d) :: d          -- effect algebra over domain ‘d‘}
   type Sub m (r :: d) (s :: d) :: Constraint -- Type-level partial order
   -- Graded monad operations
   return :: a → m (Unit m) a
   (>>=) :: m r a → (a → m s b) → m (Seq m r s) b
   sub :: Sub m r s ⇒ m r a → m s a
```

We show the encoding of the compositional live-variable analysis, which is graded by the effect algebra of transfer functions. It is a commonly held belief that type-level functions in Haskell cannot be partially applied, mainly because a type-family based encoding is considered. We show an alternate approach that is much more flexible and suits out purposes well.

To capture transfer functions at the type level, we use a class-based encoding with `d` = `[Symbol] → [Symbol] → Constraint` meaning that transfer functions are functional relations between two type-level lists of symbols (which are used to represent sets of variables). These type-level lists later get normalised to form sets by removing duplicates and giving an arbitrary consistent ordering, leveraging the `type-level-sets` package.[7]

The gen_v and $kill_v$ functions for variable `v` are defined at the type-level as:

```
class Gen (v :: Symbol) (dIn :: [Symbol]) (dOut :: [Symbol]) | v dIn → dOut
instance Gen v dIn (v ’: dIn)                   -- add ‘v‘ to the incoming set ‘dIn‘
class Kill (v :: Symbol) (dIn :: [Symbol]) (dOut :: [Symbol]) | v dIn → dOut
instance Remove dIn v dOut ⇒ Kill v dIn dOut -- rem ‘v‘ from ‘dIn‘ to get ‘dOut‘
```

Classes are types of kind `Constraint` so `Gen v :: [Symbol] → [Symbol] → Constraint`. The syntax `v dIn → dOut` is a *functional dependency* telling the type checker that `v` and `dIn` uniquely determine `dOut`, i.e., these class-based relations are really functions. The single instances of each class are then equivalent to the usual λ-based definitions of gen_v and $kill_v$.

The definition of `Kill` uses a recursive type-level function for removing an element from a list, again encoded as a functional relation (for brevity, we skip its recursive definition):

```
class Remove (xs :: [Symbol]) (x :: Symbol) (ys :: [Symbol]) | xs x → ys
```

We can capture type-level identity and function composition (which we write as `:|>` due to its later use for the effect algebra) as:

```
class Id dIn dOut | dIn → dOut     -- Identity function
instance Id d d
```

[6] https://hackage.haskell.org/package/effect-monad

[7] https://hackage.haskell.org/package/type-level-sets

```
class (:|>) f g dIn dOut              -- Function composition
instance (f dIn dMid, g dMid dOut) ⇒ (:|>) g f dIn dOut
```

We then define a data type for a Haskell implementation of the graded monad MultiState$_{\text{triv}}$ by wrapping a monad transformer stack of state monad transformers capturing enough variables for our program. Here we capture a maximum of four mutable variables as:

```
data MultiState (r :: [Symbol] → [Symbol] → Constraint) (a :: Type) =
  MultiState {unMS :: StateT Int (StateT Int (StateT Int (StateT Int Identity))) x}
```

We give `MultiState` a graded monad instance which uses the above type-level identity and function composition:

```
instance GradedMonad MultiState where
   type Unit MultiState     = Id
   type Seq  MultiState r s = r :|> s
   type Sub  MultiState r s = PointwiseSub r s

   return x = MultiState $ M.return x
   (MultiState x) >>= k = MultiState ((M.>>=) x (unMS ∘ k))
   sub (MultiState x) = MultiState x
```

The operations wrap the underlying monad, packing and unpacking the wrapper data type via its constructor and deconstructor. We then define `get` and `put` operations for each of the variables we need, e.g. for X we have `"x"` as its type-level symbol representation:

```
getX :: MultiState (Gen "x") Int
getX = MultiState get
```

```
putX :: Int → MultiState (Kill "x") ()
putX x = MultiState (put x)
```

Example 3 showed the translation of $z := x + y$ as a prefix for a program labelled g. In our Haskell implementation, we can write exactly the same code:

```
exm3 g = do { x ← getX; y ← getY; putZ (x + y); g }
```

This leverages GHC's `RebindableSyntax` extension which allows do {} to be desugared into graded monad operations instead of monad operations. We can then query GHC's type inference which yields the type:

```
exm3 :: MultiState s b → MultiState (Gen "x" :|> (Gen "y" :|> (Kill "z" :|> s))) b
```

To get the data-flow at the current program point, we apply the transfer function grade to the empty set (Section 4.3.1) via the following function:

```
atProgramPoint :: r '[] dOut ⇒ MultiState r x → Set (AsSet dOut)
atProgramPoint (MultiState _) = Set
```

where `AsSet` normalises the type-level list into a set representation and `Set` is a data type with a *phantom type* parameter (not used in any data constructor).

Thus `atProgramPoint` captures the resulting data-flow value `dOut` as a type-level set by forcing the data-flow value input to unify with the boundary value (empty set `'[]`). Applied to `exm3`, GHC calculates the following type representing the set $\{x, y\}$ as expected:

```
atProgramPoint (exm3 (return ())) :: Set '["x", "y"]
```

C Details and proofs for the graded monad of liveness

The state-management operations used in the graded monad definition of Section 4.3.2 (which were omitted for brevity) are defined in turn as follows:

$$\downarrow_{\phi,\phi'} : \mathsf{Store}(\mathit{reads}(\phi \rhd \phi')) \rightarrow \mathsf{Store}(\mathit{reads}(\phi)) = \lambda s.s_{|\phi(\emptyset)}$$

i.e., we restrict the domain of the incoming store s to the set $\mathit{reads}(\phi)$ (hence the name of *restriction* for this operator). This relies on the property that $x \in \phi(\emptyset) \implies x \in (\phi \rhd \phi')(\emptyset)$ which is proved by induction on the generating set of transfer functions (see supplement).

$$\blacktriangleleft_{\phi,\phi'} : \mathsf{Store}(\mathit{reads}(\phi \rhd \phi')) \times \mathsf{Store}(\mathit{footprint}(\phi)) \rightarrow \mathsf{Store}(\mathit{reads}(\phi'))$$
$$= \lambda(s,s').\{\begin{cases} x \mapsto s'(x) & x \in \mathit{footprint}(\phi) \\ x \mapsto s(x) & x \in \mathit{reads}(\phi \rhd \phi') \wedge x \notin \mathit{footprint}(\phi) \end{cases}\}$$

where $x \in \mathit{reads}(\phi')$ i.e., choose from the right state s' if x is in its domain, otherwise chose from s if x is in its domain but not in the domain of s'. We have that $x \notin \mathit{footprint}(\phi) \wedge x \notin \mathit{reads}(\phi \rhd \phi') \implies x \notin \mathit{reads}(\phi')$ (by induction on generating set of transfer functions) which implies that the resulting map is well-defined (a total function).

$$\lhd_{\phi,\phi'} : \mathsf{Store}(\mathit{footprint}(\phi)) \times \mathsf{Store}(\mathit{footprint}(\phi')) \rightarrow \mathsf{Store}(\mathit{footprint}(\phi \rhd \phi'))$$
$$= \lambda(s,s').\{\begin{cases} x \mapsto s'(x) & x \in \mathit{footprint}(\phi') \\ x \mapsto s(x) & x \in \mathit{footprint}(\phi) \wedge x \notin \mathit{footprint}(\phi') \end{cases}\}$$

where $x \in \mathit{footprint}(\phi \rhd \phi')$. This merging operator resembles $\blacktriangleleft$, where an additional lemma $x \notin \mathit{footprint}(\phi) \wedge x \notin \mathit{footprint}(\phi') \implies x \notin \mathit{footprint}(\phi \rhd \phi')$ (by induction on generating set of transfer functions) implies that the resulting map is well-defined (a total function).

▶ **Proposition 5** (Restriction right unit). *$\forall \phi$ and $s \in \mathsf{Store}(\mathit{reads}(\phi))$ then $\downarrow_{\phi,id} s \equiv s$*

▶ **Proposition 6** (Merge $\lhd$ right unit). *$\forall \phi$ and $s \in \mathsf{Store}(\mathit{footprint}(\phi))$ then $s \lhd_{\phi,id} \hat{\emptyset} \equiv s$*

▶ **Proposition 7** (Merge $\blacktriangleleft$ right unit). *$\forall \phi'$ and $s \in \mathsf{Store}(\mathit{reads}(\phi'))$ then $s \blacktriangleleft_{id,\phi'} \hat{\emptyset} \equiv s$*

▶ **Proposition 8** (Merge $\lhd$ left unit). *$\forall \phi'$ and $s \in \mathsf{Store}(\mathit{footprint}(\phi'))$ then $\hat{\emptyset} \lhd_{id,\phi'} s {\equiv} s$*

▶ **Proposition 9** (Restriction closure). *$\forall \phi, \phi', \phi''$ and $s \in \mathsf{Store}(\mathit{reads}((\phi \rhd \phi') \rhd \phi''))$ then:*
$\downarrow_{\phi,\phi'} (\downarrow_{\phi \rhd \phi',\phi''} s) \equiv \downarrow_{\phi,\phi' \rhd \phi''} s$

▶ **Proposition 10** (Merge $\lhd$ associativity). *$\forall \phi, \phi', \phi''$ and $s \in \mathsf{Store}(\mathit{footprint}(\phi))$, $s' \in \mathsf{Store}(\mathit{footprint}(\phi'))$, and $s'' \in \mathsf{Store}(\mathit{footprint}(\phi''))$ then:*
$(s \lhd_{\phi,\phi'} s') \lhd_{(\phi \rhd \phi'),\phi''} s'' \equiv s \lhd_{\phi,\phi' \rhd \phi''} (s' \lhd_{\phi',\phi''} s'')$.

▶ **Proposition 11** (Merge $\lhd/\blacktriangleleft$ associativity). *$\forall \phi, \phi', \phi''$ and $s \in \mathsf{Store}(\mathit{reads}((\phi \rhd \phi') \rhd \phi''))$ and $s' \in \mathsf{Store}(\mathit{footprint}(\phi))$ and $s'' \in \mathsf{Store}(\mathit{footprint}(\phi'))$ then:*
$s \blacktriangleleft_{(\phi \rhd \phi'),\phi''} (s' \lhd_{\phi,\phi'} s'') \equiv (s \blacktriangleleft_{\phi,\phi' \rhd \phi''} s') \blacktriangleleft_{\phi',\phi''} s''$

▶ **Proposition 12** (Merge $\blacktriangleleft$/restriction commutativity). *$\forall \phi, \phi', \phi''$ and $s \in \mathsf{Store}(\mathit{reads}((\phi \rhd \phi') \rhd \phi''))$ and $s' \in \mathsf{Store}(\mathit{footprint}(\phi))$ then: $(\downarrow_{\phi \rhd \phi',\phi''} s) \blacktriangleleft_{\phi,\phi'} s' \equiv \downarrow_{\phi',\phi''} (s \blacktriangleleft_{\phi,\phi' \rhd \phi''} s')$*

The supplementary material (`https://doi.org/10.5281/zenodo.3784967`) provides the proofs. We now prove the identity and associativity axioms for the graded monad. We refer to the monoid axioms as *idL* ($id \rhd \phi = \phi$) and *idR* ($\phi \rhd id = \phi$) and *assoc* ($(\phi \rhd \phi') \rhd \phi'' = \phi \rhd (\phi' \rhd \phi'')$).

(right identity). $\forall m : M^{\phi}\alpha$ then: $m \gg\!=_{\phi,id} \mathsf{return} \equiv m : M^{\phi}\alpha$ which follows by:

$$\begin{aligned}
& m \gg\!=_{\phi,id} \mathsf{return} \\
\{\mathit{defs.}{+}\beta\} \equiv\ & \lambda s.\mathsf{let}(y,s') = m(\downarrow_{\phi,id}\ s)\ \mathsf{in}\ \mathsf{let}(z,s'') = ((\lambda x.\lambda s.(x,\hat{\emptyset}))\ y)(s \blacktriangleleft_{\phi,id} s')\ \mathsf{in}\ (z, s' \lhd_{\phi,id} s'') \\
\{\beta\} \equiv\ & \lambda s.\mathsf{let}(y,s') = m(\downarrow_{\phi,id}\ s)\ \mathsf{in}\ \mathsf{let}(z,s'') = ((\lambda s.(y,\hat{\emptyset}))(s \blacktriangleleft_{\phi,id} s')\ \mathsf{in}\ (z, s' \lhd_{\phi,id} s'') \\
\{\beta\} \equiv\ & \lambda s.\mathsf{let}(y,s') = m(\downarrow_{\phi,id}\ s)\ \mathsf{in}\ \mathsf{let}(z,s'') = (y,\hat{\emptyset})\ \mathsf{in}\ (z, s' \lhd_{\phi,id} s'') \\
\{\beta\} \equiv\ & \lambda s.\mathsf{let}(y,s') = m(\downarrow_{\phi,id}\ s)\ \mathsf{in}\ (y, s' \lhd_{\phi,id} \hat{\emptyset}) \\
\{\mathit{idR}{+}P.5\} \equiv\ & \lambda s.\mathsf{let}(y,s') = m\ s\ \mathsf{in}\ (y, s' \lhd_{\phi,id} \hat{\emptyset}) \\
\{\mathit{idR}{+}P.6\} \equiv\ & \lambda s.\mathsf{let}(y,s') = m\ s\ \mathsf{in}\ (y, s')) \\
\{\beta + \eta\} \equiv\ & m
\end{aligned}$$

(left identity). $\forall x : \alpha,\ f : \alpha \to M^{\phi'}\beta$ then $\mathsf{return}\ x \gg\!=_{id,\phi'} f = f\ x : M^{\phi'}\beta$ follows by:

$$\begin{aligned}
& \mathsf{return}\ x \gg\!=_{id,\phi'} f \\
\{\mathit{defs.}{+}\beta\} \equiv\ & \lambda s.\mathsf{let}(y,s') = ((\lambda x.\lambda s.(x,\hat{\emptyset}))x)(\downarrow_{id,\phi'}\ s)\ \mathsf{in}\ \mathsf{let}(z,s'') = (f\ y)(s \blacktriangleleft_{id,\phi'} s')\ \mathsf{in}\ (z, s' \lhd_{id,\phi'} s'') \\
\{\beta\} \equiv\ & \lambda s.\mathsf{let}(y,s') = (x,\hat{\emptyset})\ \mathsf{in}\ \mathsf{let}(z,s'') = (f\ y)(s \blacktriangleleft_{id,\phi'} s')\ \mathsf{in}\ (z, s' \lhd_{id,\phi'} s'') \\
\{\beta\} \equiv\ & \lambda s.\mathsf{let}(z,s'') = (f\ x)(s \blacktriangleleft_{id,\phi'} \hat{\emptyset})\ \mathsf{in}\ (z, \hat{\emptyset} \lhd_{id,\phi'} s'') \\
\{\mathit{idL}{+}P.7\} \equiv\ & \lambda s.\mathsf{let}(z,s'') = (f\ x)s\ \mathsf{in}\ (z, \hat{\emptyset} \lhd_{id,\phi'} s'') \\
\{\mathit{idL}{+}P.8\} \equiv\ & \lambda s.\mathsf{let}(z,s'') = (f\ x)s\ \mathsf{in}\ (z, s'') \\
\{\beta{+}\eta\} \equiv\ & f\ x
\end{aligned}$$

(associativity). $\forall m : M^{\phi}\alpha, f : \alpha \to M^{\phi'}\beta,\ g : \beta \to M^{\phi''}\gamma$ then
$(m \gg\!=_{\phi,\phi'} f) \gg\!=_{\phi \rhd \phi',\phi''} g = m \gg\!=_{\phi,\phi' \rhd \phi''} (\lambda x.\ f\ x \gg\!=_{\phi',\phi''} g)$
follows by:

$$\begin{aligned}
& (m \gg\!=_{\phi,\phi'} f) \gg\!=_{\phi \rhd \phi',\phi''} g \\
\{\mathit{defs}{+}\beta\} \equiv\ & \lambda s.\mathsf{let}(y,s') = \begin{pmatrix} (\lambda s.\mathsf{let}(y,s') = m(\downarrow_{\phi,\phi'}\ s)\ \mathsf{in} \\ \mathsf{let}(z,s'') = (f\ y)(s \blacktriangleleft_{\phi,\phi'} s')\ \mathsf{in}\ (z, s' \lhd_{\phi,\phi'} s'')) \end{pmatrix} (\downarrow_{\phi \rhd \phi',\phi''}\ s) \\
& \quad \mathsf{in}\ \mathsf{let}(z,s'') = (g\ y)(s \blacktriangleleft_{\phi \rhd \phi',\phi''} s')\ \mathsf{in}\ (z, s' \lhd_{\phi \rhd \phi',\phi''} s'') \\
\{\beta\} \equiv\ & \lambda s.\mathsf{let}(y,s') = \begin{pmatrix} \mathsf{let}(y,s') = m(\downarrow_{\phi,\phi'}\ (\downarrow_{\phi \rhd \phi',\phi''}\ s))\ \mathsf{in} \\ \mathsf{let}(z,s'') = (f\ y)((\downarrow_{\phi \rhd \phi',\phi''}\ s) \blacktriangleleft_{\phi,\phi'} s')\ \mathsf{in}\ (z, s' \lhd_{\phi,\phi'} s'') \end{pmatrix} \\
& \quad \mathsf{in}\ \mathsf{let}(z,s'') = (g\ y)(s \blacktriangleleft_{\phi \rhd \phi',\phi''} s')\ \mathsf{in}\ (z, s' \lhd_{\phi \rhd \phi',\phi''} s'') \\
\{\textsf{let}\text{-}\mathit{assoc}\} \equiv\ & \lambda s.\mathsf{let}(y,s') = m(\downarrow_{\phi,\phi'}\ (\downarrow_{\phi \rhd \phi',\phi''}\ s))\ \mathsf{in} \\
& \quad \mathsf{let}(z,s'') = (f\ y)((\downarrow_{\phi \rhd \phi',\phi''}\ s) \blacktriangleleft_{\phi,\phi'} s') \\
& \quad \mathsf{let}(z',s''') = (g\ z)(s \blacktriangleleft_{(\phi \rhd \phi'),\phi''} (s' \lhd_{\phi,\phi'} s''))\ \mathsf{in}\ (z', (s' \lhd_{\phi,\phi'} s'') \lhd_{(\phi \rhd \phi'),\phi''} s''') \\
\{\mathit{assoc}{+}P.9\text{-}12\} \equiv\ & \lambda s.\mathsf{let}(y,s') = m(\downarrow_{\phi,\phi' \rhd \phi''}\ s)\ \mathsf{in} \\
& \quad \mathsf{let}(z,s'') = (f\ y)(\downarrow_{\phi',\phi''}\ (s \blacktriangleleft_{\phi,\phi' \rhd \phi''} s'))\ \mathsf{in} \\
& \quad \mathsf{let}(z',s''') = (g\ z)((s \blacktriangleleft_{\phi,\phi' \rhd \phi''} s') \blacktriangleleft_{\phi',\phi''} s'')\ \mathsf{in}(z', s' \lhd_{\phi,\phi' \rhd \phi''} (s'' \lhd_{\phi',\phi''} s''')) \\
\{\textsf{let}\text{-}\mathit{assoc}\} \equiv\ & \lambda s.\mathsf{let}(y,s') = m(\downarrow_{\phi,\phi' \rhd \phi''}\ s)\ \mathsf{in} \\
& \quad \mathsf{let}(z,s'') = \begin{pmatrix} \mathsf{let}(y,s'') = (f\ y)(\downarrow_{\phi',\phi''}\ (s \blacktriangleleft_{\phi,\phi' \rhd \phi''} s'))\ \mathsf{in} \\ \mathsf{let}(z,s''') = (g\ y)((s \blacktriangleleft_{\phi,\phi' \rhd \phi''} s') \blacktriangleleft_{\phi',\phi''} s'')\ \mathsf{in}\ (z, s'' \lhd_{\phi',\phi''} s''') \end{pmatrix} \\
& \quad \mathsf{in}\ (z, s' \lhd_{\phi,\phi' \rhd \phi''} s'') \\
\{\beta\} \equiv\ & \lambda s.\mathsf{let}(y,s') = m(\downarrow_{\phi,\phi' \rhd \phi''}\ s)\ \mathsf{in} \\
& \quad \mathsf{let}(z,s'') = (\begin{pmatrix} \lambda x.\lambda s.\mathsf{let}(y,s') = (f\ x)(\downarrow_{\phi',\phi''}\ s)\ \mathsf{in} \\ \mathsf{let}(z,s'') = (g\ y)(s \blacktriangleleft_{\phi',\phi''} s')\ \mathsf{in}\ (z, s' \lhd_{\phi',\phi''} s'') \end{pmatrix}\ y)(s \blacktriangleleft_{\phi,\phi' \rhd \phi''} s') \\
& \quad \mathsf{in}\ (z, s' \lhd_{\phi,\phi' \rhd \phi''} s'') \\
\{\mathit{defs}{+}\beta\} \equiv\ & m \gg\!=_{\phi,\phi' \rhd \phi''} (\lambda x.\ f\ x \gg\!=_{\phi',\phi''} g)
\end{aligned}$$

A Profunctorial Scott Semantics

Zeinab Galal
Université de Paris, IRIF, CNRS, Paris, France
zgalal@irif.fr

Abstract

In this paper, we study the bicategory of profunctors with the free finite coproduct pseudo-comonad and show that it constitutes a model of linear logic that generalizes the Scott model. We formalize the connection between the two models as a change of base for enriched categories which induces a pseudo-functor that preserves all the linear logic structure. We prove that morphisms in the co-Kleisli bicategory correspond to the concept of strongly finitary functors (sifted colimits preserving functors) between presheaf categories. We further show that this model provides solutions of recursive type equations which provides 2-dimensional models of the pure lambda calculus and we also exhibit a fixed point operator on terms.

2012 ACM Subject Classification Theory of computation → Linear logic; Theory of computation → Categorical semantics

Keywords and phrases Linear Logic, Scott Semantics, Profunctors

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.16

Acknowledgements I thank Thomas Ehrhard, Marcelo Fiore, Chaitanya Leena Subramaniam and Christine Tasson for helpful discussions on this article and the referees for their valuable feedback.

1 Introduction

1.1 Scott semantics and linear logic

Domain theory provides a mathematical structure to study computability with a notion of approximation of information. The elements of a domain represent partial stages of computation and the order relation represents increasing computational information. Among the desired properties of the interpretation of a program are monotonicity and continuity, i.e. the more a function has information on its input, the more it will provide information on its output and any finite part of the output can be attained through a finite computation. These features form the basis of Scott semantics of λ-calculus whose framework is Scott-continuous functions (monotonous maps preserving directed suprema) between domains. A fundamental property of Scott-continuous functions is that they admit a least fixed point which allows for the study of recursively defined programs.

Linear logic (**LL**) arose from the analysis by Girard of denotational models of system F (second order λ-calculus). It allows the study of how programs or proofs manage their resources by using exponential modalities that distinguish linear arguments that can be used exactly once and non-linear ones that can be used an arbitrary number of times [13]. One of the most basic models of linear logic is the category of sets and relations **Rel** which provides a quantitative semantics of **LL** as it allows to recover the number of times a program or a proof uses its argument to compute a given output. In quantitative models of **LL**, non-linear programs are thought of as analytic maps that are infinitely differentiable and represented by power series which can be approximated by polynomials. Viewing programs as series, a natural question was to understand the logical counterpart of differentiation, which led Ehrhard and Regnier to introduce differential linear logic and the syntactic notion of Taylor expansion which associates a formal sum of resource λ-terms to a given λ-term [7, 8].

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 16; pp. 16:1–16:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Huth showed that the Scott model of λ-calculus can be extended to a model of **LL** where the objects are prime algebraic complete lattices, the linear maps are functions preserving all suprema and the co-Kleisli maps are Scott-continuous functions [14, 15]. Independently, Winskel gave a simpler presentation based on preorders and ideal relations [22, 23]. In both cases, the co-Kleisli category is equivalent to the category of prime algebraic complete lattices and Scott-continuous functions between them. The obtained linear logic model is qualitative in that it only provides information about which arguments were used to compute a given output but not how many times. The qualitative Scott model is connected to the quantitative differential relational model through an extensional collapse construction discovered by Ehrhard [6]. This construction has been used in the context of intersection types which characterize normalization properties of λ-calculus. The quantitative relational model corresponds to a non-idempotent intersection type system whereas the qualitative Scott model corresponds to an idempotent type system. The extensional collapse construction provides a connection between the two type systems that allows to translate non-idempotent normalization to the indempotent one [5].

1.2 Categorifying Scott semantics

When taking a categorical approach to domain theory, preorders are generalized to categories and a morphism $f : x \to y$ is now an explicit name to represent the fact that y contains more computational information than x. This approach was extensively studied by Winskel among others and has proved in many ways fruitful in the theory of concurrent computation [3, 24]. This analogy can be formalized in the setting of enriched categories. A preorder $A = (|A|, \leq_A)$ corresponds to a category enriched over the two element lattice $\mathbb{2} = (\{\emptyset \leq \mathbb{1}\}, \wedge, \mathbb{1})$ where for every $a, a' \in |A|$, the homset $A(a, a')$ is equal to $\mathbb{1}$ if $a \leq_A a'$ and is empty otherwise. A $\mathbb{2}$-functor between preorders A and B is simply an order-preserving function $f : |A| \to |B|$ and the presheaf category of a preorder $[A^{op}, \mathbb{2}]$ corresponds to the set of down-closed subsets of A ordered with by inclusion. An ideal relation between preorders A and B (a relation up-closed in A and down-closed in B) corresponds to a monotone function $A \to [B^{op}, \mathbb{2}]$. Using the cartesian closed structure, it can be identified with a monotone map $A \times B^{op} \to \mathbb{2}$ which gives the direct correspondence with $\mathbb{2}$-profunctors.

Following this analogy, Cattani and Winskel showed that the bicategory of profunctors with the finite colimit completion pseudo-comonad $\mathcal{F}$ forms a model of linear logic that generalizes intuitions from the Scott model [3]. In their model, filtered colimits generalize directed suprema and Scott-continuous functions correspond to finitary functors. More recently, Fiore, Gambino, Hyland and Winskel used profunctors with the free symmetric monoidal completion pseudo-comonad $\mathcal{S}$ and showed that it forms a differential model of linear logic that generalizes the theory of combinatorial species of structures [10]. The monoidal structure of the exponential modality $\mathcal{S}$ encodes linear substitution and $\mathcal{S}$-species can be considered as a categorified version of the differential relational model.

In this paper, we study the free coproduct completion pseudo-comonad $\mathcal{C}$ (which corresponds to the finite Fam-construction) which models non-linear operations such as duplication and erasure. In the setting of algebraic theories and operads, symmetric operads are monads in the category of combinatorial species $[\mathcal{S}\,\mathbb{1}, \mathbf{Set}]$ with the Day convolution product and a Lawvere theory is a monad in the category $[\mathbf{FinSet}, \mathbf{Set}] \simeq [\mathcal{C}\,\mathbb{1}, \mathbf{Set}]$ with the substitution product. This analogy extends to the many-sorted case where symmetric many-sorted operads correspond to monads in the bicategory of $\mathcal{S}$-species [10]. Similarly, monads for $\mathcal{C}$-species correspond to many-sorted Lawvere theories. $\mathcal{C}$-species are also related to the cartesian closed bicategory of cartesian profunctors studied by Fiore and Joyal [12] where $\mathcal{C}$-species can be obtained by restricting to free cartesian categories.

Our motivation is two-fold: firstly, when we take $\mathcal{C}$ as a pseudo-comonad to interpret the exponential modality, we obtain a model of linear logic that generalizes the Scott model. There is indeed a monoidal functor from **Set** to the two-element lattice $\mathbb{2}$ that induces a change of base pseudo-functor from $\mathcal{C}$-species to the Scott model which commutes with all the constructions of linear logic. The obtained model of $\mathcal{C}$-species gives a different perspective on how to categorify Scott-continuity: directed suprema now correspond to sifted colimits and Scott-continuous functions correspond to strongly finitary functors. These correspondences are summarized in the table below:

a preorder $A = (\lvert A\rvert, \leq_A)$	a small category $\mathbf{A}$
a monotonous function $f : A \to B$	a functor $F : \mathbf{A} \to \mathbf{B}$
a down-closed subset $x \subseteq \lvert A\rvert$	a presheaf $X : \mathbf{A}^{op} \to \mathbf{Set}$
an ideal relation $R \subseteq A \times B$	a profunctor $F : \mathbf{A} \to\!\!\!\!\!\to \mathbf{B}$
inclusion of relations	a natural transformation
a directed supremum	a sifted colimit
a Scott-continuous function	a strongly finitary functor

Secondly, since $\mathcal{S}$-species categorify the relational model and $\mathcal{C}$-species categorify the Scott-model, our future goal is to connect them using a construction in the spirit of the extensional collapse mentionned above and to explore the intersection type counterpart of this construction in the profunctorial setting.

Contributions

- In Section 3, we show that the model of profunctors with the finite coproduct pseudo-comonad $\mathcal{C}$ is a model of linear logic which is a generalization of the qualitative Scott model with **Rel**.
- The connection is formalized by exhibiting a change of base pseudo-functor that commutes with the linear logic structure (Section 5).
- We prove in Section 4 that morphisms in the associated co-Kleisli bicategory correspond to the notion of functors preserving sifted colimits by providing a biequivalence between the two structures.
- Lastly, we show in Section 6 that every recursive type equation built from linear logic connectives has a least fixed point solution, and we exhibit a fixed point operator on terms which allows for the study of recursively defined terms.

Notation

- For an integer $n \in \mathbb{N}$, we write $\underline{n}$ for the set $\{1, \ldots, n\}$.
- The length of a finite sequence of elements $u = \langle a_1, \ldots, a_n \rangle$ is denoted by $|u|$.
- Categories will be denoted in boldface whereas simple text will be used for sets. For a small category $\mathbf{A}$, we denote by $\widehat{\mathbf{A}}$ the presheaf category $[\mathbf{A}^{op}, \mathbf{Set}]$ and write $\mathrm{y}_{\mathbf{A}} : \mathbf{A} \to \widehat{\mathbf{A}}$ for the Yoneda embedding.
- We use $\cong$ for natural isomorphisms between functors or category isomorphisms and $\simeq$ for equivalences.

2 The Qualitative Scott Model of Linear Logic

The category of prime algebraic lattices and maps preserving all suprema gives rise to a model of linear logic whose associated co-Kleisli category is equivalent to the Scott model of prime algebraic lattices and Scott-continuous functions between them [14, 15]. It is however more convenient to manipulate linear logic constructions on preorders rather than on lattices and since any prime algebraic lattice can be obtained as the set of downward closed subsets of a preorder, we adopt the viewpoint of taking our objects to be preorders. The Kleisli category of this model is then equivalent to the Scott model of prime algebraic lattices [22, 23].

Define **ScottL** to be the category whose objects are preordered sets $A = (|A|, \leq_A)$ and a morphism from A to B is a relation $R \subseteq |A| \times |B|$ that is up-closed in $(|A|, \leq_A)$ and down-closed in $(|B|, \leq_B)$. Explicitly, it verifies that for all $a, a' \in |A|$ and $b, b' \in |B|$:

$$(a \leq_A a' \ \wedge \ (a,b) \in R \ \wedge \ b' \leq_B b) \ \Rightarrow \ (a', b') \in R$$

The identity is given by $\mathrm{id}_A := \{(a, a') \mid a' \leq_A a\}$ and composition is the usual composition of relations. The dual of a preordered set A is defined to be $A^\perp := (|A|, \geq_A)$. Every preordered set A induces a a domain $\mathcal{I}(A)$ of ideals (downward closed subsets of A) ordered by inclusion. Morphisms in the linear category $\mathbf{ScottL}(A, B)$ can then be seen as elements of $\mathcal{I}(A^\perp \times B)$; they are also equivalent to functions from $\mathcal{I}(A)$ to $\mathcal{I}(B)$ that commute with all unions.

ScottL is a compact closed category where the tensor product $A \otimes B$ is given by $(|A| \times |B|, \leq_A \times \leq_B)$ and has the singleton preordered set $\mathbb{1}$ as a unit. The additive structure is given by the disjoint union of preorders $A \& B := (|A| + |B|, \leq_A + \leq_B)$ with the empty preordered set $\mathbb{0}$ as zero object.

The exponential modality $! : \mathbf{ScottL} \to \mathbf{ScottL}$ takes a preordered set A to the preordered set whose web $|!A|$ is the set of finite sequences of elements in $|A|$ i.e $|!A| := \{\langle a_1, \ldots, a_n \rangle \mid a_i \in |A|, n \in \mathbb{N}\}$ and the preorder relation is defined as follows:

$$\langle a_1, \ldots, a_n \rangle \leq_{!A} \langle b_1, \ldots, b_m \rangle \quad :\Leftrightarrow \quad \forall i \in \underline{n}, \exists j \in \underline{m}, a_i \leq_A b_j$$

On morphisms, a relation $R \in \mathbf{ScottL}(A, B)$ is mapped to

$$!R := \{(\langle a_1, \ldots a_n \rangle, \langle b_1, \ldots, b_m \rangle) \mid \forall j \in \underline{m}, \exists i \in \underline{n}, (a_i, b_j) \in R\}.$$

The obtained co-Kleisli category $\mathbf{ScottL}_!$ is then equivalent to the category of prime algebraic lattices and Scott-continuous functions between them as every relation in $\mathbf{ScottL}_!(A, B)$ corresponds to a Scott-continuous function $\mathcal{I}(A) \to \mathcal{I}(B)$.

▶ Remark 1. We chose this presentation of the comonad instead of finite subsets [14, 22] or finite multisets [5, 6] since it is more convenient for the profunctorial generalization with the free coproduct pseudo-comonad. Note that for the three presentations, the associated lattices of downward closed subsets are all isomorphic and the associated co-Kleisli categories are all equivalent to the Scott model.

3 The Model of Profunctors

3.1 The bicategory of profunctors

The notion of *profunctor* (or *distributor*) has become increasingly important in theoretical computer science as a tool to model a wide range of bidimensional computational structures. For small categories **A** and **B**, a profunctor $F : \mathbf{A} \nrightarrow \mathbf{B}$ is a functor $F : \mathbf{A} \times \mathbf{B}^{op} \to \mathbf{Set}$ or equivalently a functor $F : \mathbf{A} \to \widehat{\mathbf{B}}$ [2]. Profunctors can be seen as a generalization of **Rel** as a relation $R \subseteq A \times B$ corresponds to a profunctor between discrete categories such that each component is either the empty set or a singleton.

The composite of two profunctors $F : \mathbf{A} \nrightarrow \mathbf{B}$ and $G : \mathbf{B} \nrightarrow \mathbf{C}$ is the profunctor $G \circ F : \mathbf{A} \times \mathbf{C}^{op} \to \mathbf{Set}$ given by the coend formula:

$$(a, c) \mapsto \int^{b \in \mathbf{B}} F(a, b) \times G(b, c).$$

and the identity $\mathrm{id}_{\mathbf{A}} : \mathbf{A} \nrightarrow \mathbf{A}$ is given by the yoneda embedding $\mathrm{y}_{\mathbf{A}} : \mathbf{A} \to \widehat{\mathbf{A}}$. Composition of profunctors is however associative only up to natural isomorphisms which puts us in the setting of a bicategory [17].

▶ **Definition 2.** *The bicategory of profunctors* **Prof** *consists of*

- 0**-cells:** *small categories* $\mathbf{A}, \mathbf{B}$*,*
- 1**-cells:** *profunctors* $F : \mathbf{A} \nrightarrow \mathbf{B}$*,*
- 2**-cells:** *natural transformations between profunctors.*

In [10], Fiore et al. showed that **Prof** is a bicategorical model of **LL** that constitutes a generalization of Joyal's species of structures. **Prof** can be equipped with a symmetric monoidal structure where the unit $\mathbb{1}$ is the category with a unique object and a unique arrow and the tensor product $\otimes : (\mathbf{A}, \mathbf{B}) \mapsto \mathbf{A} \times \mathbf{B}$ is the cartesian product of categories in **Cat**. The dualizer $-^{\perp}$ which takes a small category $\mathbf{A}$ to $\mathbf{A}^{op}$ provides **Prof** with a compact closed structure. The additive structure $\& : (\mathbf{A}, \mathbf{B}) \mapsto \mathbf{A} + \mathbf{B}$ is given by the coproduct in **Cat** which makes **Prof** a cartesian bicategory whose zero object is the empty category $\mathbb{0}$. The exponential modality in their model relies on the free symmetric monoidal completion $\mathcal{S}\mathbf{A}$ for a small category $\mathbf{A}$.

3.2 The free finite coproduct pseudo-comonad

Cattani and Winskel showed that by taking the free finite colimit completion pseudo-comonad $\mathcal{F}$, we obtain a model of **LL** that generalizes the Scott model [3]. The maps obtained in the co-Kleisli bicategory do not however preserve bisimulation which led them to consider the pseudo-comonad of indexed families instead. Among the examples given is the restriction to finite families which corresponds to the free finite coproduct completion $\mathcal{C}$. In this section, we expand this example and exhibit that **Prof** together with the pseudo-comonad $\mathcal{C}$ forms a model of **LL** that gives a different perspective on how to categorify the Scott model. While 1-categorical semantics of linear logic has been extensively studied (see [18] for a complete review of **LL**-models and [7] for differential linear logic), no complete account of what is a bicategorical model of differential linear logic has been given yet. In this section, we take the same compact closed structure for the linear bicategory described in the previous paragraph (see [3] and [10] for more details). The remaining ingredients to obtain a model of **LL** are a pseudo-comonad structure and Seely equivalences satisfying the coherence conditions for a linear exponential pseudo-comonad.

▶ **Definition 3.** *For a small category* $\mathbf{A}$*, define* $\mathcal{C}\mathbf{A}$ *to be the category whose objects are finite sequences* $\langle a_1, \ldots, a_n \rangle$ *of objects of* $\mathbf{A}$ *and a morphism between two sequences* $\langle a_1, \ldots, a_n \rangle$ *and* $\langle b_1, \ldots, b_m \rangle$ *consists of a pair* $(\sigma, (f_i)_{i \in \underline{n}})$ *of a function* $\sigma : \underline{n} \to \underline{m}$ *and a family of morphisms* $f_i : a_i \to b_{\sigma(i)}$ *in* $\mathbf{A}$ *for* $i \in \underline{n}$*. Equivalently, the hom-sets can be described by:*

$$\mathcal{C}\mathbf{A}(\langle a_1, \ldots, a_n \rangle, \langle b_1, \ldots, b_m \rangle) = \prod_{i \in \underline{n}} \sum_{j \in \underline{m}} \mathbf{A}(a_i, b_j).$$

We recall below a classical result:

▶ **Lemma 4.** *For two finite sequences u and v in $\mathcal{C}\mathbf{A}$, the concatenation (denoted by $u \oplus v$) provides a coproduct structure for $\mathcal{C}\mathbf{A}$ and the empty sequence $\langle\rangle$ is initial. $\mathcal{C}\mathbf{A}$ is the free finite coproduct completion of $\mathbf{A}$, i.e. for any functor $F : \mathbf{A} \to \mathbf{B}$ where $\mathbf{B}$ is a category with finite coproducts, there exists a unique (up to natural isomorphism) functor $\overline{F} : \mathcal{C}\mathbf{A} \to \mathbf{B}$ that preserves finite coproducts and makes the following diagram commute:*

$$\begin{tikzcd}
\mathbf{A} \arrow[rr, "\eta_{\mathbf{A}}"] \arrow[dr, "F"'] & & \mathcal{C}\mathbf{A} \arrow[dl, dotted, "\overline{F}"] \\
& \mathbf{B} &
\end{tikzcd}$$

▶ Note 5. To obtain the free symmetric monoidal completion $\mathcal{S}\mathbf{A}$, it suffices to take the subcategory of $\mathcal{C}\mathbf{A}$ where we restrict σ in Definition 3 to be a bijection.

The endofunctor $\mathcal{C} : \mathbf{Cat} \to \mathbf{Cat}$ can be equipped with a 2-monad structure. In order to obtain a pseudo-comonad on **Prof**, one needs to start with the dual construction of the free finite product 2-monad $\mathcal{P} : \mathbf{Cat} \to \mathbf{Cat}$ which takes a small category $\mathbf{A}$ to $\mathcal{P}(\mathbf{A}) = (\mathcal{C}(\mathbf{A}^{op}))^{op}$. In [11], Fiore et al. show that the 2-monad $\mathcal{P}$ lifts to a pseudo-monad on **Prof**. Taking its dual, one obtains the pseudo-comonad of finite coproducts on **Prof** which we briefly describe below.

For a profunctor $F : \mathbf{A} \nrightarrow \mathbf{B}$ between small categories $\mathbf{A}$ and $\mathbf{B}$, $\mathcal{C}F : \mathcal{C}\mathbf{A} \nrightarrow \mathcal{C}\mathbf{B}$ is given by:

$$\mathcal{C}F : (u, v) \mapsto \prod_{j \in |v|} \int^{a_j \in \mathbf{A}} F(a_j, v_j) \times \mathcal{C}\mathbf{A}(\langle a_j \rangle, u)$$

The counit and comultiplication pseudo-natural transformations have the following components:

$$\begin{aligned} &\varepsilon_{\mathbf{A}} : \mathcal{C}\mathbf{A} \nrightarrow \mathbf{A} && \delta_{\mathbf{A}} : \mathcal{C}\mathbf{A} \nrightarrow \mathcal{C}^2\mathbf{A} \\ &(u, a) \mapsto \mathcal{C}\mathbf{A}(\langle a \rangle, u) && (u, \langle u_1, \ldots, u_n \rangle) \mapsto \mathcal{C}\mathbf{A}(u_1 \oplus \cdots \oplus u_n, u) \end{aligned}$$

A morphism $F : \mathcal{C}\mathbf{A} \nrightarrow \mathbf{B}$ in the co-Kleisli bicategory $\mathbf{Prof}_{\mathcal{C}}$ is called a *$\mathcal{C}$-species* and its lifting or promotion $F^{\mathcal{C}} : \mathcal{C}\mathbf{A} \nrightarrow \mathcal{C}\mathbf{B}$ is given by:

$$F^{\mathcal{C}}(u, v) = \mathcal{C}F \circ \delta_{\mathbf{A}}(u, v) = \prod_{j \in |v|} F(u, v_j)$$

The composite in $\mathbf{Prof}_{\mathcal{C}}$ of two $\mathcal{C}$-species $F : \mathcal{C}\mathbf{A} \nrightarrow \mathbf{B}$ and $G : \mathcal{C}\mathbf{B} \nrightarrow \mathbf{C}$ is then given by the profunctorial composition $G \circ F^{\mathcal{C}} : \mathcal{C}\mathbf{A} \nrightarrow \mathbf{C}$.

▶ **Lemma 6.** *There is a Seely adjoint equivalence of categories $\mathcal{C}(\mathbf{A} \,\&\, \mathbf{B}) \simeq \mathcal{C}\mathbf{A} \otimes \mathcal{C}\mathbf{B}$.*

Proof. Define $I_{\mathbf{A},\mathbf{B}} : \mathcal{C}\mathbf{A} \otimes \mathcal{C}\mathbf{B} \to \mathcal{C}(\mathbf{A} \,\&\, \mathbf{B})$ as follows:

$$I_{\mathbf{A},\mathbf{B}} : (u, v) \mapsto \mathcal{C}(i_1)(u) \oplus \mathcal{C}(i_2)(v) \in \mathcal{C}(\mathbf{A} \,\&\, \mathbf{B})$$

where $i_1 : \mathbf{A} \to \mathbf{A} \,\&\, \mathbf{B}$ and $i_2 : \mathbf{B} \to \mathbf{A} \,\&\, \mathbf{B}$ are the coprojections maps. Consider now the functor $p_1 : \mathbf{A} \,\&\, \mathbf{B} \to \mathcal{C}\mathbf{A}$ defined by $p_1(1, a) := \langle a \rangle$ and $p_1(2, b) := \langle\rangle$. This functor induces a functor $\overline{p_1} : \mathcal{C}(\mathbf{A} \,\&\, \mathbf{B}) \to \mathcal{C}\mathbf{A}$ (using the universal property of the free finite coproduct completion) that is a retract of $\mathcal{C}(i_1) : \mathcal{C}\mathbf{A} \to \mathcal{C}(\mathbf{A} \,\&\, \mathbf{B})$. We define similarly a functor $\overline{p_2} : \mathcal{C}(\mathbf{A} \& \mathbf{B}) \to \mathcal{C}\mathbf{B}$ that is a retract of $\mathcal{C}(i_2) : \mathcal{C}\mathbf{B} \to \mathcal{C}(\mathbf{A} \& \mathbf{B})$. For $w \in \mathcal{C}(\mathbf{A} \& \mathbf{B})$, we denote by $w.1 \in \mathcal{C}\mathbf{A}$ its image by $\overline{p_1}$ and by $w.2 \in \mathcal{C}\mathbf{B}$ its image by $\overline{p_2}$. $S_{\mathbf{A},\mathbf{B}} : \mathcal{C}(\mathbf{A} \,\&\, \mathbf{B}) \to \mathcal{C}\mathbf{A} \otimes \mathcal{C}\mathbf{B}$ is then defined to be the functor $w \mapsto (w.1, w.2) \in \mathcal{C}\mathbf{A} \otimes \mathcal{C}\mathbf{B}$.

$$\begin{tikzcd}
\mathcal{C}(\mathbf{A} \mathbin{\&} \mathbf{B}) \arrow[rr, bend left, "S_{\mathbf{A},\mathbf{B}}"] \arrow[rr, phantom, "\top"] & & \mathcal{C}\mathbf{A} \otimes \mathcal{C}\mathbf{B} \arrow[ll, bend left, "I_{\mathbf{A},\mathbf{B}}"]
\end{tikzcd}$$

We now exhibit two natural isomorphisms $\eta : \mathrm{Id}_{\mathcal{C}\mathbf{A}\otimes\mathcal{C}\mathbf{B}} \Rightarrow S_{\mathbf{A},\mathbf{B}} \circ I_{\mathbf{A},\mathbf{B}}$ and $\varepsilon : I_{\mathbf{A},\mathbf{B}} \circ S_{\mathbf{A},\mathbf{B}} \Rightarrow \mathrm{Id}_{\mathcal{C}(\mathbf{A}\&\mathbf{B})}$. For $(u, v) \in \mathcal{C}\mathbf{A} \otimes \mathcal{C}\mathbf{B}$, we have that

$$((\mathcal{C}(i_1)(u) \oplus \mathcal{C}(i_2)(v)).1, (\mathcal{C}(i_1)(u) \oplus \mathcal{C}(i_2)(v)).2) = (u, v)$$

so η is just the identity. Let $w \in \mathcal{C}(\mathbf{A} \mathbin{\&} \mathbf{B})$, ε_w is the reshuffling isomorphism from $\mathcal{C}(i_1)(w.1) \oplus \mathcal{C}(i_2)(w.2)$ to w. The adjunction is obtained by seeing that for $(u, v) \in \mathcal{C}\mathbf{A} \otimes \mathcal{C}\mathbf{B}$ and $w \in \mathcal{C}(\mathbf{A} \mathbin{\&} \mathbf{B})$ there is a natural isomorphism:

$$\mathcal{C}(\mathbf{A} \mathbin{\&} \mathbf{B})(\mathcal{C}(i_1)(u) \oplus \mathcal{C}(i_2)(v), w) \cong \mathcal{C}\mathbf{A}(u, w.1) \times \mathcal{C}\mathbf{B}(v, w.2).$$ ◀

In [10], Fiore et al. show that **Prof** together with the free symmetric monoidal pseudo-comonad $\mathcal{S}$ is a model of differential linear logic which can be seen as a categorification of the differential relational model. We show below that similarly to the Scott model with preorders, $\mathbf{Prof}_{\mathcal{C}}$ is not a model of differential linear logic.

▶ **Lemma 7.** *$\mathbf{Prof}_{\mathcal{C}}$ is not a model of differential linear logic.*

Proof. If $\mathbf{Prof}_{\mathcal{C}}$ were a model of differential linear logic, there would exist a pseudo-natural transformation $\overline{\varepsilon} : \mathrm{Id}_{\mathbf{Prof}} \to \mathcal{C}$ interpreting the codereliction rule. One of the required coherence axioms for the codereliction is $\varepsilon \circ \overline{\varepsilon} = \mathrm{Id}_{\mathbf{Prof}}$. For all $\mathbf{A} \in \mathbf{Cat}$ and $a, a' \in \mathbf{A}$, we then have:

$$\int^{u \in \mathcal{C}\mathbf{A}} \overline{\epsilon}_{\mathbf{A}}(a, u) \times \mathcal{C}\mathbf{A}(\langle a' \rangle, u) \cong \mathbf{A}(a', a)$$

which implies $\overline{\epsilon}(a, \langle a' \rangle) \cong A(a', a)$. Another required coherence diagrams for the codereliction map is that for any object $\mathbf{A}$, $w_{\mathbf{A}} \circ \overline{\varepsilon}_{\mathbf{A}} = \mathbb{0}_{\mathbf{A}}$ where $w_{\mathbf{A}} : \mathcal{C}\mathbf{A} \nrightarrow \mathbb{1}$ is the weakening map given by $u \mapsto \mathcal{C}\mathbf{A}(\langle\rangle, u)$ and $\mathbb{0}_{\mathbf{A}} : \mathbf{A} \nrightarrow \mathbb{1}$ is the empty profunctor. For $a \in \mathbf{A}$, we have:

$$w_{\mathbf{A}} \circ \overline{\varepsilon}_{\mathbf{A}}(a) = \int^{u \in \mathcal{C}\mathbf{A}} \mathcal{C}\mathbf{A}(\langle\rangle, u) \times \overline{\epsilon}_{\mathbf{A}}(a, u) \cong \overline{\epsilon}_{\mathbf{A}}(a, \langle\rangle)$$

Since there is a map $\langle\rangle \to \langle a \rangle$ in $\mathcal{C}\mathbf{A}$, it induces a function from $\overline{\epsilon}_{\mathbf{A}}(a, \langle a \rangle)$ to $\overline{\epsilon}_{\mathbf{A}}(a, \langle\rangle)$. The set $\overline{\epsilon}_{\mathbf{A}}(a, \langle a \rangle) \cong \mathbf{A}(a, a)$ is not empty as it contains id_a so the set $\overline{\epsilon}_{\mathbf{A}}(a, \langle\rangle)$ cannot be empty which contradicts our hypothesis. ◀

The extensional collapse construction between the relational model and the Scott model gives a connection between $\mathbf{Rel}_!$ which is not well-pointed to the well-pointed category $\mathbf{ScottL}_!$. In the categorified setting, the situation is however more subtle. In the case of $\mathcal{S}$-species, Fiore introduced the notion of generalized analytic functor as the Taylor series counterpart of species that generalizes Joyal's original definition for combinatorial species [9]. For small categories $\mathbf{A}$ and $\mathbf{B}$, a functor $P : \widehat{\mathbf{A}} \to \widehat{\mathbf{B}}$ is said to be *analytic* if there exists a generalized species $F : \mathcal{S}\mathbf{A} \nrightarrow \mathbf{B}$ such that P is isomorphic to $\mathbf{Lan}_{s_{\mathbf{A}}} F$ (the left Kan extension of F along $s_{\mathbf{A}}$)

$$\begin{tikzcd}
\mathcal{S}\mathbf{A} \arrow[rr, "F"] \arrow[dr, "s_{\mathbf{A}}"'] & \Downarrow & \widehat{\mathbf{B}} \\
& \widehat{\mathbf{A}} \arrow[ur, dotted, "\mathbf{Lan}_{s_{\mathbf{A}}}(F)"'] &
\end{tikzcd}$$

where $s_{\mathbf{A}} : \mathcal{S}\mathbf{A} \to \widehat{\mathbf{A}}$ is the functor that takes a sequence $\langle a_1, \dots a_n\rangle$ in $\mathcal{S}\mathbf{A}$ to the presheaf $\sum\limits_{i=1}^{n} \mathbb{y}_{\mathbf{A}}(a_i)$ in $\widehat{\mathbf{A}}$. The functor $s_{\mathbf{A}} : \mathcal{S}\mathbf{A} \to \widehat{\mathbf{A}}$ is not fully faithful which entails that the functor giving the correspondence between $\mathcal{S}$-species and analytic functors:

$$\mathbf{Lan}_{s_{\mathbf{A}}} : \mathbf{Prof}_{\mathcal{S}}(\mathbf{A}, \mathbf{B}) \to [\widehat{\mathbf{A}}, \widehat{\mathbf{B}}]$$

is not fully faithful. Fiore however showed that it is possible to reconstruct an $\mathcal{S}$-species from its analytic functor if we restrict the objects to be groupoids [9]. Formally, he showed that there is a biequivalence between the bicategory of $\mathcal{S}$-species restricted to groupoids and the 2-category of analytic functors (whose 0-cells are small groupoids, 1-cells are analytic functors and 2-cells are weak cartesian natural transformations). If we extend the functor $s_{\mathbf{A}}$ to the category $\mathcal{C}\mathbf{A}$, we obtain a fully faithful functor which entails that $\mathbf{Lan}_{s_{\mathbf{A}}} : \mathbf{Prof}_{\mathcal{C}}(\mathbf{A}, \mathbf{B}) \to [\widehat{\mathbf{A}}, \widehat{\mathbf{B}}]$ is now fully faithful as a corollary of a classical result on Kan extension that we recall below.

▶ **Proposition 8** ([16]). *Let $S : \mathbf{A} \to \mathbf{B}$ be a fully faithful functor from a small category $\mathbf{A}$. Then, for every functor $F : \mathbf{A} \to \mathbf{D}$ into a cocomplete category $\mathbf{D}$, the natural transformation $F \Rightarrow \mathbf{Lan}_S(F) \circ S$ is an isomorphism and the functor $\mathbf{Lan}_S : [\mathbf{A}, \mathbf{D}] \to [\mathbf{B}, \mathbf{D}]$ is fully faithful.*

3.3 The cartesian closed structure

▶ **Definition 9.** *A cartesian bicategory $\mathcal{B}$ is* closed *if for every pair of objects $A, B \in \mathcal{B}$, we have:*

1. *an exponential object $A \Rightarrow B$ together with an evaluation map $Ev_{A,B} \in \mathcal{B}((A \Rightarrow B)\&A, B)$ and*
2. *for every $X \in \mathcal{B}$, an adjoint equivalence*

$$\begin{tikzcd} \mathcal{B}(X, B^A) \arrow[r, bend left, "Ev_{A,B} \circ ((-) \,\&\, A)"] \arrow[r, phantom, "\perp"] & \mathcal{B}(X \,\&\, A, B) \arrow[l, bend left, "\lambda"] \end{tikzcd}$$

▶ **Proposition 10.** *$\mathbf{Prof}_{\mathcal{C}}$ is cartesian closed.*

Proof.

1. For small categories $\mathbf{A}$ and $\mathbf{B}$, the exponential object $\mathbf{A} \Rightarrow \mathbf{B}$ is defined as $(\mathcal{C}\mathbf{A})^{op} \times \mathbf{B}$ and the evaluation map $\mathrm{Ev}_{\mathbf{A},\mathbf{B}} : \mathcal{C}\left((\mathbf{A} \Rightarrow \mathbf{B}) \,\&\, \mathbf{A}\right) \mathrel{\ooalign{$\to$\cr\hfil$\mapstochar$\hfil}} \mathbf{B}$ takes $(W, b) \in \mathcal{C}\left((\mathbf{A} \Rightarrow \mathbf{B}) \,\&\, \mathbf{A}\right) \times \mathbf{B}^{op}$ to the set:

$$\int^{u_1 \in \mathcal{C}(\mathbf{A} \Rightarrow \mathbf{B}), u_2 \in \mathcal{C}\mathbf{A}} \mathcal{C}(\mathbf{A} \Rightarrow \mathbf{B})(u_1, W.1) \times \mathcal{C}\mathbf{A}(u_2, W.2) \times \mathcal{C}(\mathbf{A} \Rightarrow \mathbf{B})(\langle (u_2, b)\rangle, u_1)$$
$$\cong \mathcal{C}(\mathbf{A} \Rightarrow \mathbf{B})(\langle (W.2, b)\rangle, W.1)$$

2. For $G : \mathcal{C}(\mathbf{X} \,\&\, \mathbf{A}) \mathrel{\to\!\!\!\!\mapstochar\,} \mathbf{B}$, $\lambda(G) : \mathcal{C}\mathbf{X} \mathrel{\to\!\!\!\!\mapstochar\,} (\mathcal{C}\mathbf{A} \multimap \mathbf{B})$ is defined by

$$\lambda(G) : (z, (u, b)) \mapsto F(\mathcal{C}(i_1)(z) \oplus \mathcal{C}(i_2)(u), b).$$

Let $F : \mathcal{C}\mathbf{X} \mathrel{\to\!\!\!\!\mapstochar\,} (\mathbf{A} \Rightarrow \mathbf{B})$, $F \,\&\, \mathbf{A} : \mathcal{C}(\mathbf{X} \,\&\, \mathbf{A}) \mathrel{\to\!\!\!\!\mapstochar\,} (\mathbf{A} \Rightarrow \mathbf{B}) \,\&\, \mathbf{A}$ is the profunctor that takes $(w, (1, (u, b)))$ in $\mathcal{C}(\mathbf{X} \,\&\, \mathbf{A}) \times ((\mathbf{A} \Rightarrow \mathbf{B}) \,\&\, \mathbf{A})^{op}$ to:

$$F \circ \Pi_1(w, (u, b)) = \int^{z \in \mathcal{C}\mathbf{X}} F(z, (u, b)) \times \mathcal{C}(\mathbf{X} \,\&\, \mathbf{A})(\mathcal{C}(i_1)z, w)$$
$$\cong \int^{z \in \mathcal{C}\mathbf{X}} F(z, (u, b)) \times \mathcal{C}\mathbf{X}(z, w.1) \times \mathcal{C}\mathbf{A}(\langle\rangle, w.2) \cong F(w.1, (u, b))$$

and the image of an element $(w,(2,a)) \in \mathcal{C}(\mathbf{X} \mathbin{\&} \mathbf{A}) \times ((\mathbf{A} \Rightarrow \mathbf{B}) \mathbin{\&} \mathbf{A})^{op}$ is given by

$$\Pi_2(w,a) = \mathcal{C}(\mathbf{X} \mathbin{\&} \mathbf{A})(\langle(2,a)\rangle, w) \cong \mathcal{C}\mathbf{A}(\langle a\rangle, w.2).$$

Hence, its lifting $(F \mathbin{\&} \mathbf{A})^{\mathcal{C}} : \mathcal{C}(\mathbf{X} \mathbin{\&} \mathbf{A}) \mathrel{\to\!\!\!\!\!+} \mathcal{C}((\mathbf{A} \Rightarrow \mathbf{B}) \mathbin{\&} \mathbf{A})$ is given by:

$$(w,W) \mapsto \cong F^{\mathcal{C}}(w.1, W.1) \times \mathcal{C}\mathbf{A}(W.2, w.2)$$

We can now compute $\mathrm{Ev}_{\mathbf{A},\mathbf{B}} \circ (F \mathbin{\&} \mathbf{A}) : \mathcal{C}(\mathbf{X} \mathbin{\&} \mathbf{A}) \mathrel{\to\!\!\!\!\!+} \mathbf{B}$:

$$\begin{aligned}(w,b) &\mapsto \int^{W \in \mathcal{C}(\mathbf{A} \Rightarrow \mathbf{B}) \& \mathbf{A}} \mathrm{Ev}_{\mathbf{A},\mathbf{B}}(W,b) \times (F \mathbin{\&} \mathbf{A})^{\mathcal{C}}(w,W) \\ &\cong \int^{W} \mathcal{C}(\mathbf{A} \Rightarrow \mathbf{B})(\langle(W.2,b)\rangle, W.1) \times F^{\mathcal{C}}(w.1, W.1) \times \mathcal{C}\mathbf{A}(W.2, w.2) \\ &\cong F^{\mathcal{C}}(w.1, \langle(w.1,b)\rangle) \cong F(w.1,(w.2,b)\end{aligned}$$

Consider now two profunctors $F : \mathcal{C}\mathbf{X} \mathrel{\to\!\!\!\!\!+} (\mathbf{A} \Rightarrow \mathbf{B})$ and $G : \mathcal{C}(\mathbf{X} \mathbin{\&} \mathbf{A}) \mathrel{\to\!\!\!\!\!+} \mathbf{B}$, we exhibit the following natural ismorphisms:

$$\eta_F : F \overset{\sim}{\Longrightarrow} \lambda(\mathrm{Ev}_{\mathbf{A},\mathbf{B}} \circ (F \mathbin{\&} \mathbf{A})) \qquad \beta_G : \mathrm{Ev}_{\mathbf{A},\mathbf{B}} \circ (\lambda(G) \mathbin{\&} \mathbf{A}) \overset{\sim}{\Longrightarrow} G$$

For $(z,(u,b)) \in \mathcal{C}\mathbf{X} \times (\mathbf{A} \Rightarrow \mathbf{B})^{op}$, we have:

$$\begin{aligned}\lambda(\mathrm{Ev}_{\mathbf{A},\mathbf{B}} \circ (F \mathbin{\&} \mathbf{A}))(z,(u,b)) &\cong (\mathrm{Ev}_{\mathbf{A},\mathbf{B}} \circ (F \mathbin{\&} \mathbf{A}))(\mathcal{C}i_1 z \oplus \mathcal{C}i_2 u, b) \\ &\cong F((\mathcal{C}i_1 z \oplus \mathcal{C}i_2 u).1, (\mathcal{C}i_1 z \oplus \mathcal{C}i_2 u).2, b)) \cong F(z,(u,b))\end{aligned}$$

and for $(w,b) \in \mathcal{C}(\mathbf{X} \mathbin{\&} \mathbf{A}) \times \mathbf{B}^{op}$, we obtain:

$$\begin{aligned}&\mathrm{Ev}_{\mathbf{A},\mathbf{B}}(\lambda(G) \mathbin{\&} \mathbf{A})(w,b) = \lambda(G)(w.1,(w.2,b)) \\ &\cong G((\mathcal{C}(i_1)(w.1) \oplus \mathcal{C}(i_2)(w.2)), b) \cong G(w,b)\end{aligned}$$

◀

4 Strongly finitary functors

In the case of analytic functors for $\mathcal{S}$-species (restricted to groupoids), one can characterize them as functors preserving filtered colimits and weak wide pullbacks [9]. Cattani and Winskel showed that $\mathcal{F}$-species correspond to the notion of finitary functors, i.e. functors preserving filtered colimits [3]. Filtered colimits are the classical way of generalizing directed suprema in Scott's topology, and they are characterized as colimits which commute with finite limits in **Set**. In this section, we focus on a larger class of colimits, called *sifted colimits* which are colimits which commute with finite products in **Set**. A large part of the theory of locally finitely presentable categories and finitely presentable objects has analogues for sifted colimits. An object a in a category $\mathbf{A}$ is said to be *strongly finitely presentable* if $\mathbf{A}(a,-) : \mathbf{A} \to \mathbf{Set}$ preserves sifted colimits. The full subcategory of these objects in $\mathbf{A}$ is denoted by $\mathbf{A}_{\mathrm{sfp}}$. For a preorder, finitely and strongly presentable objects coincide with the compact elements and in the category **Set**, the two notions coincide with finite sets [1]. A category $\mathbf{A}$ is *strongly locally finitely presentable* if it is cocomplete, $\mathbf{A}_{\mathrm{sfp}}$ is a small category and every object of $\mathbf{A}$ is a sifted colimit of a diagram in $\mathbf{A}_{\mathrm{sfp}}$.

▶ **Lemma 11.** *For a small category $\mathbf{A}$, the presheaf category $\widehat{\mathbf{A}}$ is strongly finitely presentable and every presheaf is a sifted colimit of finite coproducts of representables.*

Proof. Let $\mathbf{A}$ be a small category, then $\widehat{\mathbf{A}}$ is strongly finitely presentable and the strongly finitely presentable objects are the regular projective presheaves [1]. In presheaf categories, the full subcategory of coproducts of representables is a regular projective cover [20]. Hence every presheaf is a sifted colimit of coproducts of representables. Since every coproduct is a filtered colimit of finite coproducts, we obtain the desired result. ◀

Functors preserving sifted colimits are called *strongly finitary functors*. On **Set**, finitary and strongly finitary functors coincide [1].

▶ **Definition 12.** *The* 2*-category* **Sift** *has small categories as objects and a morphism between two categories* $\mathbf{A}$ *and* $\mathbf{B}$ *is a strongly finitary functor* $\mathcal{P} : \widehat{\mathbf{A}} \to \widehat{\mathbf{B}}$*. The* 2*-cells between two such functors are natural transformations.*

The main result of this section is to show that there is a biequivalence between the bicategory $\mathbf{Prof}_{\mathcal{C}}$ and the 2-category **Sift**.

▶ **Lemma 13.** *For a* $\mathcal{C}$*-species* $F : \mathcal{C}\mathbf{A} \nrightarrow \mathbf{B}$*,* $\mathbf{Lan}_{s_\mathbf{A}}(F) : \widehat{\mathbf{A}} \to \widehat{\mathbf{B}}$ *preserves sifted colimits.*

Proof. Let $\mathcal{D} : I \to \widehat{\mathbf{A}}$ be a sifted diagram, we have:

$$\begin{aligned}
&\mathbf{Lan}_{s_\mathbf{A}} F(\varinjlim_{i\in I} \mathcal{D}(i))(b) = \int^{u=\langle a_1,\dots,a_n\rangle} F(u,b) \times \widehat{\mathbf{A}}(s_\mathbf{A}(u), \varinjlim_{i\in I} \mathcal{D}(i))\\
&\cong \int^{u} F(u,b) \times \prod_{j=1}^{n} \widehat{\mathbf{A}}(\mathsf{y}(a_j), \varinjlim_{i\in I} \mathcal{D}(i)) \cong \int^{u} F(u,b) \times \prod_{i=j}^{n} \varinjlim_{i\in I} \mathcal{D}(i)(a_j)\\
&\cong \int^{u} F(u,b) \times \varinjlim_{i\in I} \prod_{j=1}^{n} \mathcal{D}(i)(a_j) \cong \int^{u} F(u,b) \times \varinjlim_{i\in I} \Big(\widehat{\mathbf{A}}(s_\mathbf{A}(u), \mathcal{D}(i))\Big)\\
&\cong \int^{u} \varinjlim_{i\in I} \Big(F(u,b) \times \widehat{\mathbf{A}}(s_\mathbf{A}(u), \mathcal{D}(i))\Big) = \varinjlim_{i\in I} \Big(\int^{u} F(u,b) \times \widehat{\mathbf{A}}(s_\mathbf{A}(u), \mathcal{D}(i))\Big)
\end{aligned}$$

Since sifted colimits commute with finite products, it allows us to obtain the third isomorphism. We then make use of the facts that $(F(u,b) \times -)$ is a left adjoint, and hence colimit-preserving, and that the coend is a colimit and hence commutes with colimits. ◀

▶ **Lemma 14.** *For small categories* $\mathbf{A}$ *and* $\mathbf{B}$*, there is an adjoint equivalence between the categories:*

$$\begin{tikzcd}
\mathbf{Prof}_{\mathcal{C}}(\mathbf{A},\mathbf{B}) \arrow[rr, bend left, "\mathbf{Lan}_{s_\mathbf{A}}(-)"] \arrow[rr, phantom, "\perp"] & & \mathbf{Sift}(\mathbf{A},\mathbf{B}) \arrow[ll, bend left, "- \circ s_\mathbf{A}"]
\end{tikzcd}$$

Proof. Since $s_\mathbf{A}$ is fully faithful, for any $\mathcal{C}$-species F in $\mathbf{Prof}_{\mathcal{C}}(\mathbf{A},\mathbf{B})$ there is a natural isomorphism $\alpha_F : F \Rightarrow (\mathbf{Lan}_{s_\mathbf{A}}(F)) \circ s_\mathbf{A}$. Hence, for a natural transformation $\beta : F_1 \Rightarrow F_2$ in $\mathbf{Prof}_{\mathcal{C}}(\mathbf{A},\mathbf{B})$, its image by $\mathbf{Lan}_{s_\mathbf{A}}(-)$ is the unique natural transformation $\gamma : \mathbf{Lan}_{s_\mathbf{A}}(F_1) \Rightarrow \mathbf{Lan}_{s_\mathbf{A}}(F_2)$ such that $\gamma s_\mathbf{A} \alpha_{F_1} = \beta \alpha_{F_2}$ which provides us with a natural isomorphism $\eta : \mathrm{Id}_{\mathbf{Prof}_{\mathcal{C}}(\mathbf{A},\mathbf{B})} \Rightarrow (\mathbf{Lan}_{s_\mathbf{A}}(-)) \circ s_\mathbf{A}$ by Proposition 8.

Let $P : \widehat{\mathbf{A}} \to \widehat{\mathbf{B}}$ be a functor that preserves sifted colimits. We want to exhibit a natural isomorphism

$$\mathbf{Lan}_{s_\mathbf{A}}(P \circ s_\mathbf{A})(X) \cong P(X)$$

By Lemma 11, X is a sifted colimit of finite coproducts of representables, i.e. there exists a sifted diagram $D : I \to \mathcal{C}\mathbf{A}$ such that $X \cong \varinjlim_{i \in I} s_{\mathbf{A}}(D(i))$:

$$\begin{aligned}
&\mathbf{Lan}_{s_{\mathbf{A}}}(P \circ s_{\mathbf{A}})(X) = \int^{u=\langle a_1,\dots,a_n\rangle} P(s_{\mathbf{A}}(u)) \times \widehat{\mathbf{A}}(s_{\mathbf{A}}(u), X) \\
&\cong \int^{u} P(s_{\mathbf{A}}(u)) \times \prod_{j=1}^{n} \widehat{\mathbf{A}}(\text{よ}(a_j), \varinjlim_{i\in I} s_{\mathbf{A}} D(i)) \cong \int^{u} P(s_{\mathbf{A}}(u)) \times \varinjlim_{i\in I} \prod_{j=1}^{n} \widehat{\mathbf{A}}(\text{よ}(a_j), s_{\mathbf{A}} D(i)) \\
&\cong \int^{u} P(s_{\mathbf{A}}(u)) \times \varinjlim_{i\in I} \widehat{\mathbf{A}}(s_{\mathbf{A}}(u), s_{\mathbf{A}} D(i)) \cong \varinjlim_{i\in I} \int^{u} P(s_{\mathbf{A}}(u)) \times \mathcal{C}\mathbf{A}(s_{\mathbf{A}}(u), s_{\mathbf{A}} D(i)) \\
&\cong \varinjlim_{i\in I} P(s_{\mathbf{A}}(D(i))) \cong P(X)
\end{aligned}$$

which entails the existence of a natural isomorphism $\varepsilon : \mathbf{Lan}_{s_{\mathbf{A}}}(- \circ s_{\mathbf{A}}) \Rightarrow \mathrm{Id}_{\mathbf{Sift}(\mathbf{A},\mathbf{B})}$ as desired. The adjunction

$$[\mathcal{C}\mathbf{A}, \widehat{\mathbf{B}}](F, P \circ s_{\mathbf{A}}) \cong [\widehat{\mathbf{A}}, \widehat{\mathbf{B}}](\mathbf{Lan}_{s_{\mathbf{A}}} F, P).$$

is a direct consequence of the universal property of left Kan extensions (see Theorem 4.38 in [16] for example). ◀

▶ **Proposition 15.** *The bicategory* $\mathbf{Prof}_{\mathcal{C}}$ *is biequivalent to the* 2*-category* $\mathbf{Sift}$.

Proof. We prove that the pseudofunctor $\mathcal{F} : \mathbf{Prof}_{\mathcal{C}} \to \mathbf{Sift}$ defined below is a biequivalence. For $\mathbf{A}$ and $\mathbf{B}$ small categories, we define $\mathcal{F}(\mathbf{A}) := \mathbf{A}$ and

$$\begin{aligned}
\mathcal{F}_{\mathbf{A},\mathbf{B}} &: \mathbf{Prof}_{\mathcal{C}}(\mathbf{A}, \mathbf{B}) \to \mathbf{Sift}(\mathbf{A}, \mathbf{B}) \\
&\quad F : \mathcal{C}\mathbf{A} \nrightarrow \mathbf{B} \mapsto \mathbf{Lan}_{s_{\mathbf{A}}}(F) : \widehat{\mathbf{A}} \to \widehat{\mathbf{B}}
\end{aligned}$$

Since $\mathbf{Prof}_{\mathcal{C}}$ and $\mathbf{Sift}$ have the same objects, it follows immediately that $\mathcal{F} : \mathbf{Prof}_{\mathcal{C}} \to \mathbf{Sift}$ is essentially surjective. Lemma 14 entails that $\mathcal{F}_{\mathbf{A},\mathbf{B}}$ is an adjoint equivalence of categories. ◀

5 From Prof to ScottL

In this section, we formalize the connection between the categorical approach and the preorder model as a change of base for enriched categories. A category enriched over $\mathbb{2} = (\{\emptyset \le \mathbb{1}\}, \wedge, \mathbb{1})$ is a preorder and a $\mathbb{2}$-profunctor between two preorders $A = (|A|, \le_A)$ and $B = (|B|, \le_B)$ corresponds to a relation in $\mathbf{ScottL}(A, B)$. The functor $M : \mathbf{Set} \to \mathbb{2}$ defined by

$$X \mapsto \begin{cases} \emptyset & \text{if } X = \emptyset \\ \mathbb{1} & \text{otherwise} \end{cases}$$

is monoidal and therefore induces a lax pseudo-functor Ψ from $\mathbf{Prof}_{\mathbf{Set}}$ (just denoted by $\mathbf{Prof}$) to $\mathbf{Prof}_{\mathbb{2}} = \mathbf{ScottL}$ [4]. In this section, we give an explicit description of this change of base pseudo-functor $\Psi : \mathbf{Prof} \to \mathbf{ScottL}$ and show that it is in fact a strong pseudo-functor that preserves all the structure of linear logic. The viewpoint of enriched categories enables us to work in a unified setting where both models coexist and the change of base becomes a pseudo-functor that connects the preorder world and the categorified world in a way that preserves the structure of linear logic.

On objects, Ψ sends a small category $\mathbf{A}$ to the following preorder:

$$(\mathrm{Ob}(\mathbf{A}), \le_A) \quad \text{where} \quad a \le_A a' \quad :\Leftrightarrow \quad \mathbf{Hom}_{\mathbf{A}}(a, a') \neq \emptyset$$

For a profunctor $F : \mathbf{A} \nrightarrow \mathbf{B}$, $\Psi_{\mathbf{A},\mathbf{B}}(F)$ is given by $\Psi_{\mathbf{A},\mathbf{B}}(F) := \{(a, b) \mid F(a, b) \neq \emptyset\}$.

▶ **Lemma 16.** *For every* $\mathbf{A}, \mathbf{B}$*,* $\Psi_{\mathbf{A},\mathbf{B}} : \mathbf{Prof}(\mathbf{A}, \mathbf{B}) \to \mathbf{ScottL}(\Psi(\mathbf{A}), \Psi(\mathbf{B}))$ *is functorial.*

Proof. We first need to check that $\Psi_{\mathbf{A},\mathbf{B}}(F)$ is indeed an element of $\mathbf{ScottL}(\Psi(\mathbf{A}), \Psi(\mathbf{B}))$, i.e. that for all $(a, b) \in \Psi_{\mathbf{A},\mathbf{B}}(F)$, $(a', b') \leq_{\mathbf{A}^{op}\times\mathbf{B}} (a, b)$ implies $(a', b') \in \Psi_{\mathbf{A},\mathbf{B}}(F)$. If $(a, b) \in \Psi_{\mathbf{A},\mathbf{B}}(F)$, then $F(a, b) \neq \emptyset$ so there exists an element $s \in F(a, b)$. The inequality $(a', b') \leq_{\mathbf{A}^{op}\times\mathbf{B}} (a, b)$ implies that there exist morphisms $f : a \to a'$ in $\mathbf{A}$ and $g : b' \to b$ in $\mathbf{B}$. Hence, $F(f, g)(s) \in F(a', b')$ which is not empty as desired. When we consider $\mathbf{ScottL}$ as a bicategory, morphisms in $\mathbf{ScottL}(\Psi(\mathbf{A}), \Psi(\mathbf{B}))$ are just inclusions of relations so we only need to show that if there exists a natural transformation $\alpha : F \Rightarrow G$ in $\mathbf{Prof}(\mathbf{A}, \mathbf{B})$, then $\Psi_{\mathbf{A},\mathbf{B}}(F) \subseteq \Psi_{\mathbf{A},\mathbf{B}}(G)$. For $(a, b) \in \Psi_{\mathbf{A},\mathbf{B}}(F)$, if there exists an element $s \in F(a, b)$ then $\alpha_{(a,b)}(s) \in G(a, b)$ which implies that $(a, b) \in \Psi_{\mathbf{A},\mathbf{B}}(G)$ as desired. ◀

▶ **Proposition 17.** Ψ *is a strong pseudo-functor that preserves the linear logic structure.*

Proof.

- For profunctors $F : \mathbf{A} \nrightarrow \mathbf{B}$ and $G : \mathbf{B} \nrightarrow \mathbf{C}$, the following equalities hold:

$$\begin{aligned}
\Psi_{\mathbf{A},\mathbf{C}}(G \circ_{\mathbf{Prof}} F) &= \{(a, c) \mid \int^{b\in\mathbf{B}} F(a, b) \times G(b, c) \neq \emptyset\} \\
&= \{(a, c) \mid \exists b \in \mathrm{Ob}(\mathbf{B}), F(a, b) \neq \emptyset \text{ and } G(b, c) \neq \emptyset\} \\
&= \{(a, c) \mid \exists b \in \mathrm{Ob}(\mathbf{B}), (a, b) \in \Psi_{\mathbf{A},\mathbf{B}}(F) \text{ and } (b, c) \in \Psi_{\mathbf{B},\mathbf{C}}(G)\} \\
&= \Psi_{\mathbf{B},\mathbf{C}}(G) \circ_{\mathbf{ScottL}} \Psi_{\mathbf{A},\mathbf{B}}(F)
\end{aligned}$$

- We only show that Ψ commutes with the pseudo-comonad structure, the other cases being similar. For a small category $\mathbf{A}$, $!\Psi(\mathbf{A})$ is the preorder whose underlying set is equal to the object set of $\mathcal{C}\mathbf{A}$ so $!\Psi$ and $\Psi\mathcal{C}$ coincide on objects. For a profunctor $F : \mathbf{A} \nrightarrow \mathbf{B}$, we have:

$$\begin{aligned}
!\Psi_{\mathbf{A},\mathbf{B}}(F) &= \{(\langle a_1, \dots, a_n\rangle, \langle b_1, \dots, b_m\rangle) \mid \forall j \in \underline{m}, \exists i \in \underline{n}, (a_i, b_j) \in \Psi(F)\} \\
&= \{(\langle a_1, \dots, a_n\rangle, \langle b_1, \dots, b_m\rangle) \mid \forall j \in \underline{m}, \exists i \in \underline{n}, F(a_i, b_j) \neq \emptyset\} \\
&= \{(\langle a_1, \dots, a_n\rangle, \langle b_1, \dots, b_m\rangle) \mid \prod_{j\in\underline{m}} \sum_{i\in\underline{n}} F(a_i, b_j) \neq \emptyset\} \\
&= \{(\langle a_1, \dots, a_n\rangle, \langle b_1, \dots, b_m\rangle) \mid \mathcal{C}F(\langle a_1, \dots, a_n\rangle, \langle b_1, \dots, b_m\rangle) \neq \emptyset\} = \Psi(\mathcal{C}F)
\end{aligned}$$

  The following equalities also hold for the dereliction and the digging pseudo-natural transformations:

$$\begin{aligned}
\Psi(\varepsilon_{\mathbf{A}}) &= \{(u, a) \mid \varepsilon_{\mathbf{A}}(u, a) \neq \emptyset\} = \{(u, a) \mid \sum_{i\in|u|} \mathbf{A}(a, u_i) \neq \emptyset\} \\
&= \{(u, a) \mid \forall i \in |u|, a \leq_{\Psi(\mathbf{A})} u_i\} = \varepsilon_{\Psi(\mathbf{A})} \\
\Psi(\delta_{\mathbf{A}}) &= \{(u, \langle u_1, \dots, u_n\rangle) \mid \mathcal{C}\mathbf{A}(u_1 \oplus \dots \oplus u_n, u) \neq \emptyset\} \\
&= \{(u, \langle u_1, \dots, u_n\rangle) \mid u_1 \oplus \dots \oplus u_n \leq_{\Psi(\mathcal{C}\mathbf{A})} u\} = \delta_{\Psi\mathcal{C}\mathbf{A}}
\end{aligned}$$

◀

6 Recursive Type and Term Equations

6.1 Fixed points of Types

Recursive domain equations play a central role in denotational semantics. A classical example is Scott's D_∞ construction providing an extensional model of the untyped λ-calculus. In $\mathbf{Prof}_{\mathcal{C}}$, we show that full subcategory inclusion is a partial order relation on objects such that all linear logic constructions define Scott-continuous maps on this partially ordered class. It entails that we can give solutions to any recursive type equation constituted of linear logic operators and we exhibit in this section an example of a 2-dimensional model of pure λ-calculus in $\mathbf{Prof}_{\mathcal{C}}$.

▶ **Definition 18.** *For small categories* $\mathbf{A}$ *and* $\mathbf{B}$*, we write* $\mathbf{A} \sqsubseteq \mathbf{B}$ *if* $\mathbf{A}$ *is a full subcategory of* $\mathbf{B}$*, i.e.* $Ob(\mathbf{A}) \subseteq Ob(\mathbf{B})$ *and for all* a *and* a' *in* $Ob(\mathbf{A})$*,* $\mathbf{A}(a, a') = \mathbf{B}(a, a')$*.*

One can easily check that $\sqsubseteq$ defines a partial order relation on the class of small categories. We denote by $\mathbf{Cat}_{\sqsubseteq}$ the obtained partially ordered class and show the following lemma:

▶ **Lemma 19.** $\mathbf{Cat}_{\sqsubseteq}$ *is closed under directed colimits.*

Proof. Let $D : I \to \mathbf{Cat}_{\sqsubseteq}$ be a directed diagram. We denote by $\bigvee_{i\in I} D_i$ the category whose set of objects is $\bigcup_{i\in I} \mathrm{Ob}(D_i)$ so that for any $a, b \in \mathrm{Ob}(\bigvee_{i\in I} D_i)$, there exist $i, j \in I$ such that $a \in \mathrm{Ob}(D_i)$ and $b \in \mathrm{Ob}(D_j)$. Since I is directed, there exists $k \in I$ such that $a, b \in \mathrm{Ob}(D_k)$ so we define $\bigvee_{i\in I} D_i(a, b)$ to be $D_k(a, b)$. ◀

▶ **Lemma 20.** *All the linear logic constructions are Scott-continuous with respect to the order* $\sqsubseteq$*.*

Proof. The proof is routine, we only exhibit the dual and exponential cases:

- **Dual:** It is noteworthy to observe that the dual is monotonous with respect to this order. For $\mathbf{A} \sqsubseteq \mathbf{B}$, we have that $\mathrm{Ob}(\mathbf{A}^{op}) = \mathrm{Ob}(\mathbf{A}) \subseteq \mathrm{Ob}(\mathbf{B}) = \mathrm{Ob}(\mathbf{B}^{op})$ and for any $a, a' \in \mathbf{A}^{op}$, $\mathbf{A}^{op}(a, a') = \mathbf{A}(a', a) = \mathbf{B}(a', a) = \mathbf{B}^{op}(a, a')$ which entails that $\mathbf{A}^{op} \sqsubseteq \mathbf{B}^{op}$. Let $D : I \to \mathbf{Cat}_{\sqsubseteq}$ be a directed diagram, we want to show that $\left(\bigvee_{i\in I} D_i\right)^{op} = \bigvee_{i\in I} D_i^{op}$. It is immediate to show that these two categories have the same objects and for $a, a' \in \bigvee_{i\in I} D_i^{op}$, there exists $k \in I$ such that $a, a' \in \mathrm{Ob}(D_k)$ so that:
$$\bigvee_{i\in I} D_i^{op}(a, a') = D_k^{op}(a, a') = D_k(a', a) = (\bigvee_{i\in I} D_i)(a', a) = (\bigvee_{i\in I} D_i)^{op}(a, a').$$
- **Exponential:** For $\mathbf{A} \sqsubseteq \mathbf{B}$, $\mathrm{Ob}(\mathcal{C}\mathbf{A}) = \{\langle a_1, \ldots, a_n\rangle \mid a_i \in \mathrm{Ob}(\mathbf{A})\} \subseteq \{\langle b_1, \ldots, b_n\rangle \mid b_i \in \mathrm{Ob}(\mathbf{B})\} = \mathrm{Ob}(\mathcal{C}\mathbf{B})$ and for u, v in $\mathrm{Ob}(\mathcal{C}\mathbf{A})$:
$$\mathcal{C}\mathbf{A}(u, v) = \prod_{i\in|u|} \sum_{j\in|v|} \mathbf{A}(u_i, v_j) = \prod_{i\in|u|} \sum_{j\in|v|} \mathbf{B}(u_i, v_j) = \mathcal{C}\mathbf{B}(u, v)$$
which entails that $\mathcal{C}\mathbf{A} \sqsubseteq \mathcal{C}\mathbf{B}$ as desired. Let $D : I \to \mathbf{Cat}_{\sqsubseteq}$ be a directed diagram, we want to show that $\mathcal{C}(\bigvee_{i\in I} D_i) = \bigvee_{i\in I} \mathcal{C}D_i$. For the object sets, we haveL
$$\mathrm{Ob}(\mathcal{C}(\bigvee_{i\in I} D(i))) = \bigcup_{n\in\mathbb{N}} \mathrm{Ob}(\bigvee_{i\in I} D(i))^n = \bigcup_{n\in\mathbb{N}} (\bigcup_{i\in I} \mathrm{Ob}(D_i))^n = \bigcup_{n\in\mathbb{N}} \bigcup_{i\in I} (\mathrm{Ob}(D(i)))^n$$
$$= \bigcup_{i\in I} \bigcup_{n\in\mathbb{N}} (\mathrm{Ob}(D_i))^n = \mathrm{Ob}\left(\bigvee_{i\in I} \mathcal{C}D_i\right)$$
The third equality follows from the fact that directed unions commute with finite products. Consider now two elements $u := \langle x_1, \ldots, x_n\rangle$ and $v := \langle y_1, \ldots, y_m\rangle$ in $\bigvee_{i\in I} \mathcal{C}(D_i)$. Since I is directed, there exists $k \in I$ such that $u, v \in \mathrm{Ob}(\mathcal{C}(D_k))$, we therefore obtain:
$$(\bigvee_{i\in I} \mathcal{C}(D_i)(u, v) = \mathcal{C}(D_k)(u, v) = \prod_{l\in\underline{n}} \sum_{r\in\underline{m}} D_k(x_l, y_r) = \prod_{l\in\underline{n}} \sum_{r\in\underline{m}} \bigvee_{i\in I} D_k(x_l, y_r) = \mathcal{C}(\bigvee_{i\in I} D(i))(u, v)$$
The last equality follows from the fact that $D_k \sqsubseteq \bigvee_{i\in I} D_i$. ◀

▶ **Example 21.** By the previous lemma, any recursive type equation on $\mathbf{Cat}_{\sqsubseteq}$ built from linear logic connectives has a least fixed point. Let $\mathbf{N}$ be the least fixed point solution of $\mathbf{N} = \mathbb{1} \oplus \mathbf{N}$, it can be explicitly described as the category $\mathbf{N} = \bigoplus_{i \in \mathbb{N}} \mathbb{1}$. Consider now $\mathbf{D}$ to be the least fixed point solution of $\mathbf{D} = (\mathcal{C}(\mathbf{N} \multimap \mathbf{D}))^{op}$. Using the Seely equivalence in Lemma 6, we can first note that $\mathbf{D}$ verifies the following equivalence:

$$\begin{aligned} \mathbf{D} &= (\mathcal{C}(\mathbf{N} \multimap \mathbf{D}))^{op} \simeq (\mathcal{C}((\mathbb{1} \oplus \mathbf{N}) \multimap \mathbf{D}))^{op} \simeq (\mathcal{C}((\mathbb{1} \multimap \mathbf{D}) \mathbin{\&} (\mathbf{N} \multimap \mathbf{D})))^{op} \\ &\simeq ((\mathcal{C}(\mathbf{D})) \otimes \mathcal{C}(\mathbf{N} \multimap \mathbf{D}))^{op} = (\mathcal{C}\mathbf{D})^{op} \parr \mathbf{D} = (\mathbf{D} \Rightarrow \mathbf{D}) \end{aligned}$$

The category $\mathbf{D}$ provides an extensional reflexive object for the pure λ-calculus in the cartesian closed bicategory $\mathbf{Prof}_{\mathcal{C}}$. We make explicit its structure below by first giving the application and lambda profunctors:

$$Ap : \mathcal{C}(\mathbf{D} \Rightarrow \mathbf{D}) \nrightarrow \mathbf{D} \qquad \lambda : \mathcal{C}\mathbf{D} \nrightarrow (\mathbf{D} \Rightarrow \mathbf{D})$$

as follows: for $W \in \mathcal{C}(\mathbf{D} \Rightarrow \mathbf{D})$ and $d \in \mathbf{D}^{op}$, let $k \in \mathbb{N}$ be the smallest index such that $W \in \mathcal{C}(\mathbf{D}_k \Rightarrow \mathbf{D}_k)$ and $d \in \mathbf{D}_k^{op}$. Since $\mathbf{D}_k^{op} = (\mathcal{C}((\mathbb{1} \oplus \mathbf{N}) \multimap \mathbf{D}_{k-1})) \cong (\mathcal{C}(\mathbf{D}_{k-1}) \mathbin{\&} (\mathbf{N} \multimap \mathbf{D}_{k-1})))$, we use the Seely equivalence and obtain $d.1 \in \mathcal{C}(\mathbf{D}_{k-1}) \sqsubseteq \mathcal{C}(\mathbf{D}_k)$ and $d.2 \in \mathcal{C}(\mathbf{N} \multimap \mathbf{D}_{k-1}) = \mathbf{D}_k^{op}$. We now define Ap as the profunctor taking (W, d) to $\mathcal{C}(\mathbf{D}_k \Rightarrow \mathbf{D}_k)(\langle (d.1, d.2) \rangle, W)$.

To define $\lambda(u, (v, d))$ for $u \in \mathcal{C}\mathbf{D}$ and $(v, d) \in (\mathbf{D} \Rightarrow \mathbf{D})^{op}$, we first let l to be the smallest index such that $u \in \mathcal{C}(\mathbf{D})_l$, $v \in \mathcal{C}(\mathbf{D}_l)$ and $d \in \mathbf{D}_l^{op} \sqsubseteq \mathbf{D}_{l+1}^{op} = \mathcal{C}((\mathbb{1} \oplus \mathbf{N}) \multimap \mathbf{D}_l) \cong \mathcal{C}(\mathbf{D}_l \mathbin{\&} (\mathbf{N} \multimap \mathbf{D}_l))$. Considering the diagram below,

$$\begin{tikzcd} \mathcal{C}(\mathbf{D})_l \arrow[r, "\mathcal{C}(i_1)"'] & \mathcal{C}(\mathbf{D}_l \mathbin{\&} (\mathbf{N} \multimap \mathbf{D}_l)) \arrow[d, equal] & \mathcal{C}(\mathbf{N} \multimap \mathbf{D}_l) \arrow[l, "\mathcal{C}(i_2)"] \\ & \mathbf{D}_{l+1}^{op} & \end{tikzcd}$$

we obtain that $\mathcal{C}(i_1)(u) \oplus \mathcal{C}(i_2)(d)$ is an element of $\mathbf{D}_{l+1}^{op}$, so we define $\lambda(u, (v, d))$ to be $\mathcal{C}(\mathbf{D}_{l+1})(\mathcal{C}(i_1)(v) \oplus \mathcal{C}(i_2)(d), u)$. We then obtain:

$$\begin{aligned} \lambda \circ Ap(W, (v, d)) &= \int^{u \in \mathcal{C}\mathbf{D}} \lambda(u, (v, d)) \times Ap^{\mathcal{C}}(W, u) = \int^{u} \mathcal{C}\mathbf{D}(\mathcal{C}(i_1)(v) \oplus \mathcal{C}(i_2)(d), u) \times Ap^{\mathcal{C}}(W, u) \\ &\cong Ap(W, \mathcal{C}(i_1)(v) \oplus \mathcal{C}(i_2)(d)) = \mathcal{C}(\mathbf{D} \Rightarrow \mathbf{D})(\langle (v, d) \rangle, W) = \mathrm{Id}_{\mathbf{D} \Rightarrow \mathbf{D}}(W, (v, d)) \end{aligned}$$

The second to last equality follows from the fact that $(\mathcal{C}(i_1)(v) \oplus \mathcal{C}(i_2)(d)).1 = v$ and $(\mathcal{C}(i_1)(v) \oplus \mathcal{C}(i_2)(d)).2 = d$. We also obtain the following isomorphism:

$$\begin{aligned} Ap \circ \lambda(u, d) &= \int^{W \in \mathcal{C}(\mathbf{D} \Rightarrow \mathbf{D})} Ap(W, d) \times \lambda^{\mathcal{C}}(u, W) \\ &= \int^{W} \mathcal{C}(\mathbf{D} \Rightarrow \mathbf{D})(\langle (d.1, d.2) \rangle, W) \times \lambda^{\mathcal{C}}(u, W) \cong \lambda(u, (d.1, d.2)) \\ &= \mathcal{C}\mathbf{D}(\mathcal{C}(i_1)(d.1) \oplus \mathcal{C}(i_2)(d.2), u) \cong \mathcal{C}\mathbf{D}(\langle d \rangle, u) = \mathrm{Id}_{\mathbf{D}}(u, d) \end{aligned}$$

The second to last equality follows from the fact that d is isomorphic to $\mathcal{C}(i_1)(d.1) \oplus \mathcal{C}(i_2)(d.2)$ in $\mathcal{C}(\mathbf{D})$.

6.2 Fixed point operator for terms

▶ **Theorem 22** (e.g. [21]). *Let $\mathbf{C}$ be a category with ω-colimits together with an initial object 0 and let $F : \mathbf{C} \to \mathbf{C}$ be an endofunctor that preserves ω-chains. Then F has an initial algebra obtained by taking the colimit of the following diagram:*

$$\begin{tikzcd} 0 \arrow[r, "i"'] & F(0) \arrow[r, "F(i)"'] & F^2(0) \arrow[r, "F^2(i)"'] & \cdots \end{tikzcd}$$

where i is the unique map from the initial object to $F(0)$.

▶ **Lemma 23** (e.g. [21]). *Let $F : \mathbf{C} \to \mathbf{C}$ be an endofunctor and $a : F(c) \to c$ an initial algebra. Then a is an isomorphism.*

▶ **Definition 24.** *Let $\mathcal{B}$ be a cartesian closed bicategory and A an object of $\mathcal{B}$. A* fixpoint operator for an object *A in $\mathcal{B}$ is a 1-cell $\mathbf{fix}_A \in \mathcal{B}(A \Rightarrow A, A)$ together with an invertible 2-cell α:*

$$\begin{tikzcd} A \Rightarrow A \arrow[d, "\langle Id_{A \Rightarrow A}, \mathbf{fix}_A \rangle"'] \arrow[rd, "\mathbf{fix}_A"] \arrow[d, phantom, shift left=8ex, "\overset{\sim}{\Longrightarrow}_{\alpha}"] & \\ (A \Rightarrow A) \mathbin{\&} A \arrow[r, "Ev_{A,A}"'] & A \end{tikzcd}$$

For $f \in A \Rightarrow A$, we obtain that $Ev_{A,A}\langle f, \mathbf{fix}_A(f) \rangle \overset{\sim}{\Longrightarrow} \mathbf{fix}_A(f)$.

For a small category $\mathbf{A}$, $\mathbf{fix}_{\mathbf{A}} \in \mathbf{Prof}_{\mathcal{C}}(\mathbf{A} \Rightarrow \mathbf{A}, \mathbf{A})$ is obtained as the initial algebra of the following functor:

$$\begin{aligned} \mathcal{Y}_{\mathbf{A}} : \mathbf{Prof}_{\mathcal{C}}(\mathbf{A} \Rightarrow \mathbf{A}, \mathbf{A}) &\to \mathbf{Prof}_{\mathcal{C}}(\mathbf{A} \Rightarrow \mathbf{A}, \mathbf{A}) \\ F &\mapsto \mathrm{Ev} \circ \langle Id, F \rangle \end{aligned}$$

We identify $\mathbf{Prof}_{\mathcal{C}}(\mathbf{A} \Rightarrow \mathbf{A}, \mathbf{A})$ with the presheaf category of $(\mathbf{A} \Rightarrow \mathbf{A}) \Rightarrow \mathbf{A}$ whose initial object is the empty presheaf. Since for any morphism $H : \mathcal{C}\mathbf{X} \nrightarrow \mathbf{Y}$ in $\mathbf{Prof}_{\mathcal{C}}$, $\mathbf{Lan}_{s_{\mathbf{X}}}(H) : \widehat{X} \to \widehat{Y}$ preserves ω-colimits (as a particular case of sifted colimits), we show that $\mathcal{Y}_{\mathbf{A}}$ can be obtained as the left Kan extension of a $\mathcal{C}$-species in $\mathbf{Prof}_{\mathcal{C}}((\mathbf{A} \Rightarrow \mathbf{A}) \Rightarrow \mathbf{A}, (\mathbf{A} \Rightarrow \mathbf{A}) \Rightarrow \mathbf{A})$ which entails the existence of $\mathbf{fix}_{\mathbf{A}}$ by Theorem 22.

Consider the profunctor $\mathcal{Z}_{\mathbf{A}} \in \mathbf{Prof}_{\mathcal{C}}(((\mathbf{A} \Rightarrow \mathbf{A}) \Rightarrow \mathbf{A}) \mathbin{\&} (\mathbf{A} \Rightarrow \mathbf{A}), \mathbf{A})$ defined by the following composition:

$$\begin{tikzcd} ((\mathbf{A} \Rightarrow \mathbf{A}) \Rightarrow \mathbf{A}) \mathbin{\&} (\mathbf{A} \Rightarrow \mathbf{A}) \arrow[d, "Id \mathbin{\&} \langle Id, Id \rangle"'] & & \\ ((\mathbf{A} \Rightarrow \mathbf{A}) \Rightarrow \mathbf{A}) \mathbin{\&} (\mathbf{A} \Rightarrow \mathbf{A}) \mathbin{\&} (\mathbf{A} \Rightarrow \mathbf{A}) \arrow[d, "\mathrm{Ev}_{\mathbf{A} \Rightarrow \mathbf{A}, \mathbf{A}} \mathbin{\&} Id"'] & & \\ \mathbf{A} \mathbin{\&} (\mathbf{A} \Rightarrow \mathbf{A}) \arrow[r, "\langle \pi_2, \pi_1 \rangle"] & (\mathbf{A} \Rightarrow \mathbf{A}) \mathbin{\&} \mathbf{A} \arrow[r, "\mathrm{Ev}_{\mathbf{A},\mathbf{A}}"] & \mathbf{A} \end{tikzcd}$$

By currying, we obtain a profunctor $\lambda(\mathcal{Z}_{\mathbf{A}})$ in $\mathbf{Prof}_{\mathcal{C}}((\mathbf{A} \Rightarrow \mathbf{A}) \Rightarrow \mathbf{A}, (\mathbf{A} \Rightarrow \mathbf{A}) \Rightarrow \mathbf{A})$ whose left Kan extension along $s_{(\mathbf{A} \Rightarrow \mathbf{A}) \Rightarrow \mathbf{A}}$ is isomorphic to $\mathcal{Y}_{\mathbf{A}}$ as desired. Explicitely, $\mathcal{Y}_{\mathbf{A}}$ is given by:

$$\mathcal{Y}_{\mathbf{A}} : (F, (U, a)) = \int^{u \in \mathcal{C}\mathbf{A}} F^{\mathcal{C}}(U, u) \times \mathcal{C}(\mathbf{A} \Rightarrow \mathbf{A})(\langle (u, a) \rangle, U)$$

We can now obtain $\mathbf{fix}_{\mathbf{A}} : \mathcal{C}(\mathbf{A} \Rightarrow \mathbf{A}) \nrightarrow \mathbf{A}$ by computing $\varinjlim_{n \in \omega} \mathcal{Y}^n_{\mathbf{A}}(\mathbb{0})$.

▶ **Example 25.**

- In the theory of combinatorial species, the species of lists is a solution of the equation $L = 1 + X \cdot L$ where 1 is the species whose analytic functor $\mathbf{Set} \to \mathbf{Set}$ is given by $S \mapsto \{\star\}$ and X is the singleton species whose analytic functor is the identity endofunctor on $\mathbf{Set}$. It follows the intuition that a list is either empty or an element followed by a list. In the case of $\mathbf{Prof}_{\mathcal{C}}$, we can define for every small category $\mathbf{A}$ a $\mathcal{C}$-species of lists $L_{\mathbf{A}} : \mathcal{C}\mathbf{A} \nrightarrow \mathbf{A}$. $L_{\mathbf{A}}$ is obtained as the least fixpoint of the operator:

$$\begin{aligned} E_{\mathbf{A}} &: \mathbf{Prof}_{\mathcal{C}}(\mathbf{A}, \mathbf{A}) \to \mathbf{Prof}_{\mathcal{C}}(\mathbf{A}, \mathbf{A}) \\ &\qquad (F, (u,a)) \mapsto \mathbf{1}_{\mathbf{A}}(u,a) + \mathbb{X}_{\mathbf{A}}(u,a) \times F(u,a) = \mathcal{C}\mathbf{A}(\langle\rangle, u) + \mathcal{C}\mathbf{A}(\langle a\rangle, u) \times F(u,a) \end{aligned}$$

  where $\mathbf{1}_{\mathbf{A}}(u,a)$ is the constant species $(u,a) \mapsto \mathcal{C}\mathbf{A}(\langle\rangle, u) \simeq \{\star\}$ and $\mathbb{X}_{\mathbf{A}}$ is the singleton species $(u,a) \mapsto \mathcal{C}\mathbf{A}(\langle a\rangle, u)$. Note that if we take $\mathbf{A}$ to be the category $\mathbb{1}$, we obtain the species 1 and X mentionned above. Explicitly, the $\mathcal{C}$-species of lists $L_{\mathbf{A}} : \mathcal{C}\mathbf{A} \nrightarrow \mathbf{A}$ maps (u,a) to $\sum_{n\in\mathbb{N}} \mathcal{C}\mathbf{A}(\langle a\rangle, u)^n$ which entails that $\mathbf{Lan}_{s_{\mathbf{A}}}(L_{\mathbf{A}}) : \widehat{\mathbf{A}} \to \widehat{\mathbf{A}}$ is given by

$$(X, a) \mapsto \sum_{n\in\mathbb{N}} (X(a))^n.$$

- Using a similar reasoning, we can obtain a $\mathcal{C}$-species of binary trees, which is a solution of the equation $B = 1 + X \cdot B^2$. For a small category $\mathbf{A}$, if we compute the least fixpoint of the operator:

$$\begin{aligned} H_{\mathbf{A}} &: \mathbf{Prof}_{\mathcal{C}}(\mathbf{A}, \mathbf{A}) \to \mathbf{Prof}_{\mathcal{C}}(\mathbf{A}, \mathbf{A}) \\ &\qquad (F, (u,a)) \mapsto \mathcal{C}\mathbf{A}(\langle\rangle, u) + \mathcal{C}\mathbf{A}(\langle a\rangle, u) \times F(u,a) \times F(u,a) \end{aligned}$$

  we obtain the $\mathcal{C}$-species $B_{\mathbf{A}} : \mathcal{C}\mathbf{A} \nrightarrow \mathbf{A}$ that maps (u,a) to $\sum_{n\in\mathbb{N}} C_n \times \mathcal{C}\mathbf{A}(\langle a\rangle, u)^n$, where C_n is the nth Catalan number.

Conclusion and Perspectives

We have seen that the bicategory of profunctors with the free finite coproduct pseudo-comonad $\mathcal{C}$ provides a different perspective on how to categorify Scott continuity. This construction enables us to work in the unified framework of enriched profunctors where the change of base allows us to go from the categorified model to the preorder model while preserving the linear logic structure. An important construction in domain theory is the ideal completion which associates an algebraic domain to a preorder by completing with all directed joins. In the preorder model, the morphisms in the Eilenberg-Moore category can characterized as Scott-continuous functions between ideal completions of preorders. We aim to obtain in future work a 2-categorical analogue of this result with strongly finitary functors between sifted colimit completions of small categories. Another future direction is to connect the differential model of $\mathcal{S}$-species with the Scott model of $\mathcal{C}$-species by using a categorified version of the extensional collapse established by Ehrhard. The relationship between profunctors and intersection types has also recently been explored by Olimpieri where the non-idempotent intersection type system corresponds to the free symmetric monoidal pseudo-monad and the idempotent case corresponds to the cartesian pseudo-monad [19]. Our future goal is to connect the two type systems with the categorified extensional collapse construction.

References

1 Jirí Adámek, Jirí Rosický, and Enrico. M. Vitale. *Algebraic theories: a categorical introduction to general algebra*. Cambridge University Press, 2011.

2 Jean Bénabou. Distributors at work, 2000. Lecture notes written by Thomas Streicher. URL: https://www2.mathematik.tu-darmstadt.de/~streicher/FIBR/DiWo.pdf.

3 Gian Luca Cattani and Glynn Winskel. Profunctors, open maps and bisimulation. *Mathematical Structures in Computer Science*, 15:553–614, June 2005. doi:10.1017/S0960129505004718.

4 Geoff S. H. Cruttwell. *Normed Spaces and the Change of Base for Enriched Categories*. PhD dissertation, Dalhousie University, 2008.

5 Thomas Ehrhard. Collapsing non-idempotent intersection types. *Leibniz International Proceedings in Informatics, LIPIcs*, 16, September 2012. doi:10.4230/LIPIcs.CSL.2012.259.

6 Thomas Ehrhard. The Scott model of Linear Logic is the extensional collapse of its relational model. *Theoretical Computer Science*, 424:20–45, 2012. 26 pages. doi:10.1016/j.tcs.2011.11.027.

7 Thomas Ehrhard. An introduction to differential linear logic: proof-nets, models and antiderivatives. *Math. Struct. Comput. Sci.*, 28(7):995–1060, 2018. doi:10.1017/S0960129516000372.

8 Thomas Ehrhard and Laurent Regnier. The differential lambda-calculus. *Theor. Comput. Sci.*, 309(1):1–41, December 2003. doi:10.1016/S0304-3975(03)00392-X.

9 Marcelo Fiore. Analytic functors between presheaf categories over groupoids. *Theor. Comput. Sci.*, 546:120–131, 2014. doi:10.1016/j.tcs.2014.03.004.

10 Marcelo Fiore, Nicola Gambino, Martin Hyland, and Glynn Winskel. The cartesian closed bicategory of generalised species of structures. *J. Lond. Math. Soc. (2)*, 77(1):203–220, 2008. doi:10.1112/jlms/jdm096.

11 Marcelo Fiore, Nicola Gambino, Martin Hyland, and Glynn Winskel. Relative pseudomonads, kleisli bicategories, and substitution monoidal structures. *Selecta Mathematica*, 24:2791–2830, November 2017. doi:10.1007/s00029-017-0361-3.

12 Marcelo Fiore and André Joyal. Theory of para-toposes. Talk at the Category Theory 2015 Conference, Departamento de Matematica, Universidade de Aveiro, Portugal, 2015.

13 Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. doi:10.1016/0304-3975(87)90045-4.

14 Michael Huth. Linear domains and linear maps. In *Proceedings of the 9th International Conference on Mathematical Foundations of Programming Semantics*, page 438–453, Berlin, Heidelberg, 1993. Springer-Verlag.

15 Michael Huth, Achim Jung, and Klaus Keimel. Linear types and approximation. *Mathematical Structures in Computer Science*, 10, May 1994. doi:10.1017/S0960129500003200.

16 Gregory M. Kelly. *Basic Concepts of Enriched Category Theory*. Lecture note series / London mathematical society. Cambridge University Press, 1982.

17 Tom Leinster. Basic bicategories. In *E-print math.CT/9810017*, 1998.

18 Paul-André Melliès. Categorical semantics of linear logic. In *In: Interactive Models of Computation and Program Behaviour, Panoramas et Synthèses 27, Société Mathématique de France 1?196*, 2009.

19 Federico Olimpieri. Intersection type distributors, 2020. arXiv:2002.01287.

20 Maria Cristina Pedicchio and Walter Tholen. *Categorical foundations : special topics in order, topology, algebra, and Sheaf theory*. Cambridge University Press, 2003. doi:10.1017/CBO9781107340985.

21 Michael B. Smyth and Gordon D. Plotkin. The category-theoretic solution of recursive domain equations. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, page 13–17, USA, 1977. IEEE Computer Society. doi:10.1109/SFCS.1977.30.

22 Glynn Winskel. A linear metalanguage for concurrency. In *Algebraic Methodology and Software Technology, 7th International Conference, AMAST '98*, pages 42–58. Springer, June 1998. doi:10.1007/3-540-49253-4_6.

23 Glynn Winskel. Linearity and nonlinearity in distributed computation. In *Linear Logic in Computer Science*. Cambridge University Press, 2004. doi:10.1017/CBO9780511550850.005.

24 Glynn Winskel. Strategies as profunctors. In Frank Pfenning, editor, *Foundations of Software Science and Computation Structures*, pages 418–433, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

String Diagrams for Optics

Guillaume Boisseau
University of Oxford, UK

Abstract

Optics are a data representation for compositional data access, with lenses as a popular special case. Hedges has presented a diagrammatic calculus for lenses, but in a way that does not generalize to other classes of optic. We present a calculus that works for all optics, not just lenses; this is done by embedding optics into their presheaf category, which naturally features string diagrams. We apply our calculus to the common case of lenses, extend it to effectful lenses, and explore how the laws of optics manifest in this setting.

2012 ACM Subject Classification Software and its engineering → Visual languages; Software and its engineering → Data types and structures; Software and its engineering → Specialized application languages; Software and its engineering → Functional languages

Keywords and phrases Optic, string diagram, lens, category theory, Yoneda lemma

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.17

Related Version A preprint of this paper is available at [2], `https://arxiv.org/abs/2002.11480`. That version may be extended in the future.

Funding *Guillaume Boisseau*: Work funded by the EPSRC.

Acknowledgements I want to thank the reviewers for the SYCO6 workshop for their thorough reviews, and Jeremy Gibbons for his ever helpful comments.

1 Introduction

Optics are a versatile categorical structure. Their best-known special case, *lenses*, have found uses in a variety of contexts, from machine learning to game theory [5]. Their more general instantiations have been studied in the context of bidirectional data transformations [14]. In all cases, their main feature of interest is their composability and their peculiar bidirectional information flow.

In the interest of making them easier to represent and manipulate, authors often spontaneously use diagrams to construct instances of optics [13, 14]. These diagrams are usually informal, with one notable exception in the work of Hedges [4] on diagrams for lenses. Hedges' diagrammatic calculus however assumes a lot of structure on the underlying categories, in a way that doesn't extend to more general optics.

Here we propose instead a different approach that embeds optics into a larger space (namely its presheaf category) that naturally has string diagrams. Not only does this work for the most general optics, but all the diagrammatic gadgets follow naturally from the embedding, and it even allows for useful diagrams that would not be expressible in the category *Optic* alone.

2 Background

We fix a monoidal category $(M, \otimes, I, \lambda, \mu, a)$ throughout the paper.

We assume readers are familiar with *coends*. For an introduction to the material relevant to the study of optics, see [15, Chapter 2].

▶ Note. We will prefer diagrammatic order for composition, using the symbol $\fatsemi$.

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 17; pp. 17:1–17:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2.1 Actegories

▶ **Definition 1** ([10]). *An M-actegory (contraction of "action" and "category") is a category C equipped with a functor $\odot_C : M \times C \to C$ (the "action") and two natural structure isomorphisms $\lambda_x : I \odot_C x \xrightarrow{\sim} x$ and $a_{m,n,x} : (m \otimes n) \odot_C x \xrightarrow{\sim} m \odot_C (n \odot_C x)$ that satisfy compatibility axioms with the monoidal structure of M.*

We will drop the subscripts when the relevant actegory is clear from context. The naming of the structure morphisms clashes with those of M on purpose:

▶ **Proposition 2.** *M has canonically the structure of an M-actegory, with $\odot_M = \otimes$, and λ and a as the actegory structure morphisms.*

In what follows, when we use M as an M-actegory, we assume this canonical structure.

2.2 Optics

▶ **Definition 3** ([15, Proposition 3.1.1]). *Given two M-actegories C and D, we construct the category $Optic_{C,D}$ as follows: objects are pairs $\left(\begin{smallmatrix} x \\ u \end{smallmatrix}\right)$ where $x : C$ and $u : D$, and arrows are elements of the set*

$$Optic_{C,D}(\left(\begin{smallmatrix} x \\ u \end{smallmatrix}\right), \left(\begin{smallmatrix} y \\ v \end{smallmatrix}\right)) := \int^{m:M} C(x, m \odot_C y) \times D(m \odot_D v, u)$$

Given $\alpha : C(x, m \odot_C y)$ and $\beta : D(m \odot_D v, u)$, we will denote the corresponding arrow by $\langle \alpha \,|\, \beta \rangle_m$. Composition and identities are defined componentwise in the expected way; see [15] for more details.

▶ Note. Expanding the definition of coends in *Set*, we get that the coend above denotes the set of pairs $\langle \alpha \,|\, \beta \rangle_m$ with $\alpha : C(x, m \odot_C y)$ and $\beta : D(m \odot_D v, u)$, quotiented by the equation $\langle \alpha \fatsemi (f \odot_C y) \,|\, \beta \rangle_m = \langle \alpha \,|(f \odot_D v) \fatsemi \beta \rangle_n$ for $f : M(n, m)$.

Except in special cases, this category is not monoidal. This prevents us from having string diagrams in the usual way. We will see how to work around this limitation in the rest of the paper.

▶ **Example 4.** The canonical example of optics are lenses. They arise when $C = D = M$ and the monoidal structure of C is cartesian. We get:

$$Lens_C(\left(\begin{smallmatrix} x \\ u \end{smallmatrix}\right), \left(\begin{smallmatrix} y \\ v \end{smallmatrix}\right)) := \int^{c:C} C(x, c \times y) \times C(c \times v, u)$$

While this presentation is pleasantly symmetrical, lenses are usually described as a pair of functions without this unfamiliar coend. We can in fact calculate that both presentations are equivalent:

$$\begin{aligned} Lens_C(\left(\begin{smallmatrix} x \\ u \end{smallmatrix}\right), \left(\begin{smallmatrix} y \\ v \end{smallmatrix}\right)) &= \int^{c:C} C(x, c \times y) \times C(c \times v, u) \\ &\cong \int^{c:C} C(x, y) \times C(x, c) \times C(c \times v, u) \\ &\cong C(x, y) \times \int^{c:C} C(x, c) \times C(c \times v, u) \\ &\cong C(x, y) \times C(x \times v, u) \end{aligned}$$

We recover the usual formulation: a lens from $\left(\begin{smallmatrix} x \\ u \end{smallmatrix}\right)$ to $\left(\begin{smallmatrix} y \\ v \end{smallmatrix}\right)$ is a pair of functions $get : x \to y$ and $put : x \times v \to u$. The intuition is that *get* extracts some y from a datum x, and *put* allows replacing that y by a new v, yielding an updated datum u. It is often the case that $x = u$ and $y = v$, making this intuition clearer, but having distinct types allows for more flexibility.

A concrete example of a lens that gives access to a field of a record can be written in Haskell:

```
data Lens x u y v = L (x -> y) (x -> v -> u)

data Person = P { name :: String, address :: String }
personName :: Lens Person Person String String
personName = L get put
   where get (P name _) = name
         put (P _ address) name = P name address
```

The case for distinct types is well illustrated on tuples:

```
tupleSnd :: Lens (a, b) (a, c) b c
tupleSnd = L get put
   where get (_, b) = b
         put (a, _) c = (a, c)
```

⌟

2.3 Tambara Modules

▶ **Definition 5** ([15, Proposition 5.1.1]). *Given two M-actegories C and D, we construct the category $Tamb_{C,D}$ as follows: objects are (pro)functors $P : C^{op} \times D \to Set$ equipped with a natural transformation $strength : \int_{m:M} P(a,b) \to P(m \odot_C a, m \odot_D b)$ compatible with the actegory structures; arrows are strength-preserving natural transformations.*

This generalizes the usual notion of strength for a profunctor.

▶ **Definition 6.** *We construct the bicategory $Tamb$ as follows: objects are M-actegories; Hom-categories are the categories $Tamb_{C,D}$.*

It inherits its bicategorical structure from the bicategory $Prof$ of profunctors: the identities are the hom-profunctors $C(-,=)$, and the tensor (horizontal composition) is profunctor composition, defined as usual as follows:

$$(P \otimes Q)(a,c) = \int^{b} P(a,b) \times Q(b,c)$$

▶ Note. *Prof* and *Tamb* share in fact a lot of structure. In a sense *Tamb* is the analogue of *Prof* for M-actegories, and we will see that like *Prof* it supports a rich diagrammatic calculus.

Our interest in Tambara modules comes from the following strong relationship with optics:

▶ **Theorem 7** ([15, Proposition 5.5.2]). $[Optic^{op}_{C,D}, Set] \cong Tamb_{C,D}$

Proof. The proof can be found in [15, Proposition 5.5.2], but initially comes from [12, Proposition 6.1] in the special case where $M = C = D$, along with more results on the structure of both of those categories. ◀

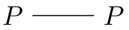
3 Diagrams for Tambara Modules

3.1 Basics

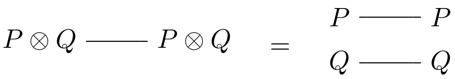
As in any bicategory, cells in *Tamb* can be represented as diagrams, as follows:

A 0-cell (an M-actegory) is represented as a planar region delimited by the other types of cells. For technical reasons we will not represent them in what follows, but it should be kept in mind that 1-cells can only be composed if their types match.

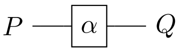
A 1-cell $P : Tamb_{C,D}$ is represented as a wire, with C above and D below:



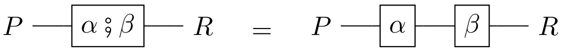
Tensoring (1-cell composition) is vertical juxtaposition (for $P : Tamb_{C,D}$ and $Q : Tamb_{D,E}$):



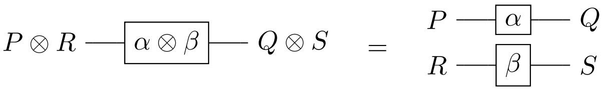
A 2-cell $\alpha : P \to Q$ (for $P, Q : Tamb_{C,D}$) is represented as:



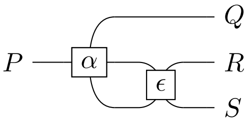
Composition is horizontal juxtaposition:



and tensoring is vertical juxtaposition:



For example, one could represent the following complex composition of cells diagrammatically:



The axioms of bicategories ensure that we can interchange boxes like we do in string diagrams for monoidal categories.

3.2 Oriented Wires

So far, this was common to any bicategory. We can now investigate gadgets specific to *Tamb*.

Let us fix an M-actegory C.

▶ **Definition 8.** *Given $x : C$, let us define two profunctors $R_x := C(-, = \odot_C x)$ and $L_x := C(- \odot_C x, =)$.*

▶ **Proposition 9.** *R_x is in $Tamb_{C,M}$ and L_x is in $Tamb_{M,C}$, where M is taken with its canonical M-actegory structure.*

Proof. R_x is a profunctor $C^{op} \times M \to Set$. The action of the $(m \odot_C -)$ functor provides it with a strength. The same works for L_x. ◀

▶ **Proposition 10.** *R_x extends to a functor $R : C \to Tamb_{C,M}$, and L_x extends to a functor $L : C^{op} \to Tamb_{M,C}$*

Proof. Straightforward from their definitions. ◀

▶ **Proposition 11.** *R and L respect the actegory structures: $R_I \cong L_I \cong M(-,=)$, $R_x \otimes R_m \cong R_{m \odot x}$, and $L_m \otimes L_x \cong L_{m \odot x}$.*

Proof. See appendix A.1. ◀

This justifies the following notation:

$$x \;\rightarrow\!\!\!\!-\!\!\!-\; x \quad := \quad R_x \;\text{———}\; R_x \tag{1}$$

and

$$x \;\rightarrow\!\!\!-\boxed{f}\rightarrow\!\!\!-\; y \quad := \quad R_x \;\text{—}\boxed{R_f}\text{—}\; R_y \tag{2}$$

similarly

$$y \;-\!\!\!\!\leftarrow\!\!\!-\; y \quad := \quad L_y \;\text{———}\; L_y \tag{3}$$

and

$$y \;-\!\!\!\leftarrow\boxed{f}-\!\!\!\leftarrow\; x \quad := \quad L_y \;\text{—}\boxed{L_f}\text{—}\; L_x \tag{4}$$

▶ Note. This choice of notation could create confusion as to whether a box on an oriented wire is meant to be seen as in the image of R/L or not. However we will see later that R and L are fully faithful, and thus this confusion fades away: all boxes on an oriented wire are arrows in C.

From the propositions above, we see that this notation respects composition in C as well as the M-actegory structures (note the inversion that happens when tensoring on a right-oriented wire):

$$\begin{matrix} x \;\rightarrow\!\!\!\!-\!\!\!-\; x \\ m \;\rightarrow\!\!\!\!-\!\!\!-\; m \end{matrix} \quad = \quad m \odot x \;\rightarrow\!\!\!\!-\!\!\!-\; m \odot x$$

$$\begin{matrix} m \;-\!\!\!\!\leftarrow\!\!\!-\; m \\ x \;-\!\!\!\!\leftarrow\!\!\!-\; x \end{matrix} \quad = \quad m \odot x \;-\!\!\!\!\leftarrow\!\!\!-\; m \odot x$$

$$I \;\rightarrow\!\!\!\!-\!\!\!-\; I \quad = \quad \textit{empty diagram}$$

$$I \;-\!\!\!\!\leftarrow\!\!\!-\; I \quad = \quad \textit{empty diagram}$$

▶ Note. Note that because of the types of the 1-cells (that are not shown in the diagrams), not all tensorings of the oriented wires are allowed. For example, it could be tempting to think that $R_x \otimes R_y \cong R_{y \otimes x}$ for $x, y : C$, but not only is C not monoidal in general, the tensoring doesn't even type-check since both R_x and R_y are objects of $\mathit{Tamb}_{C,M}$.

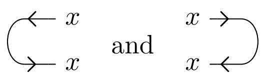
▶ Note. When C is chosen to be M, both R and L provide a monoidal embedding of M into $\mathit{Tamb}_{M,M}$; we will see later that it is also fully faithful. This means that the string diagrams in M have two full and faithful embeddings into the string diagrams of *Tamb*, using the oriented wires.

3.3 Bending Wires

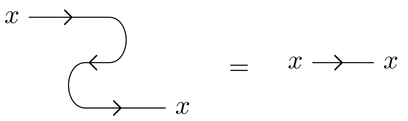
▶ **Proposition 12.** *For a given $x : C$, the modules R_x and L_x are adjoint. Moreover, the structure maps of the adjunction are dinatural in x.*

Proof. $R_x = C(-, = \odot x)$ and $L_x = C(- \odot x, =)$ are clearly adjoint in *Prof*. The adjunction lifts to *Tamb*; see appendix A.2. Dinaturality in x is straightforward from the definition of the unit and counit. ◀

This means that there exist two 2-cells, that we will draw as:

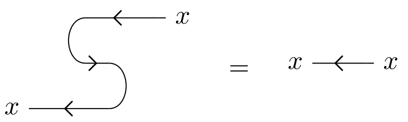


that satisfy the so-called "snake equations":



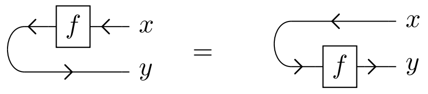
(5)

and



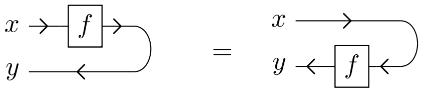
(6)

Those maps are additionally dinatural in x, which means we can also slide C-arrows around them:



(7)

and



(8)

We have discovered an additional property of the diagrammatic language: oriented arrows can be bent downwards. Note that bending upwards is not in general possible.

▶ Note. In the case of set-based lenses (i.e. $C = D = M = \mathit{Set}$ with the cartesian product), the second of those maps (the "cap") was featured in the calculus of [4]. The first map (the "cup") however cannot be expressed in that calculus.

4 Embedding Optics

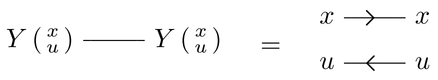
4.1 A Representation Theorem

We will now use this calculus to express optics. Recall from Theorem 7 that presheaves on optics are equivalent to Tambara modules. Consequently, the Yoneda embedding $Y : \mathit{Optic}_{C,D} \to [\mathit{Optic}^{op}_{C,D}, \mathit{Set}] \cong \mathit{Tamb}_{C,D}$ provides a fully faithful embedding of optics into *Tamb*. This is the crucial property that enables our calculus.

▶ **Lemma 13.** $Y\left(\begin{smallmatrix}x\\u\end{smallmatrix}\right) = R_x \otimes L_u$

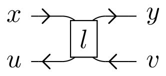
Proof. By definition of Y, R and L, modulo the equivalence of Theorem 7. ◀

Thus $Y\left(\begin{smallmatrix}x\\u\end{smallmatrix}\right)$ has the following nice diagrammatic notation:


$Y\left(\begin{smallmatrix}x\\u\end{smallmatrix}\right)$
$Y\left(\begin{smallmatrix}x\\u\end{smallmatrix}\right)$
=
x
x
u
u
(9)

From this we deduce the main theorem of this paper:

▶ **Theorem 14** (Representation theorem). *Optics $l : \mathit{Optic}_{C,D}(\left(\begin{smallmatrix}x\\u\end{smallmatrix}\right), \left(\begin{smallmatrix}y\\v\end{smallmatrix}\right))$ are in bijection with arrows in $\mathit{Tamb}_{C,D}$ of type:*

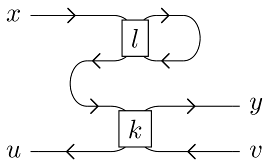


and moreover this bijection is functorial, i.e. composition of optics becomes horizontal composition of diagrams and the identity optic is the identity diagram.

Proof. By full-faithfulness and functoriality of the Yoneda embedding. ◀

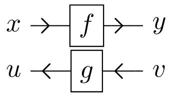
The consequences of this property need stressing: any diagram of this type represents an optic, *even if it is made of subcomponents that are not themselves optics.* A parallel can be drawn with complex numbers: a complex number with no imaginary part represents a real number, regardless of whether it was constructed (using complex operations like rotation) from complex numbers that were not themselves real numbers. In both cases, we can work in this more general space (complex numbers/Tambara modules) to reason more flexibly about the simpler objects (reals/optics).

For example, the following diagram is a valid optic, even though several of its subcomponents are not optics.



4.2 Simple Arrows

The simplest optic we can construct is made out of two simple arrows (i.e. arrows in the base M-actegories). This is sometimes called an *adapter*. Given $f : C(x, y)$ and $g : D(v, u)$, we can see from its type that $R_f \otimes L_g$ is an optic:

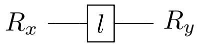


▶ **Lemma 15.** *The optic corresponding to this diagram is* $\langle f \fatsemi \lambda_y^{-1} \mid \lambda_v \fatsemi g\rangle_I$.

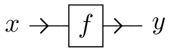
Proof. By a straightforward calculation; see appendix A.3. ◀

The special case of a single simple arrow is particularly interesting:

▶ **Theorem 16.** *All morphisms of type*



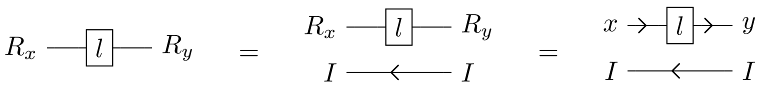
are of the form



for some unique $f : C(x, y)$.

Similarly for L *and wires going to the left.*

Proof. Since $L_I \cong M(-, =)$, we have (using a potentially confusing notation):



Thus by the representation theorem, l can be seen as an optic in $\mathit{Optic}_{C,M}((\begin{smallmatrix}x\\I\end{smallmatrix}), (\begin{smallmatrix}y\\I\end{smallmatrix}))$. We then calculate (see appendix A.4) that $\mathit{Optic}_{C,M}((\begin{smallmatrix}x\\I\end{smallmatrix}), (\begin{smallmatrix}y\\I\end{smallmatrix})) \cong C(x, y)$, with the reverse direction given by the action of R. The proof for L is identical. ◀

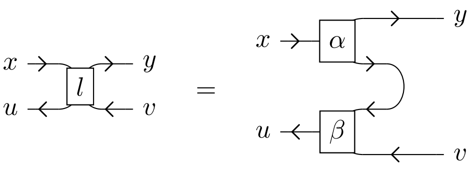
▶ **Corollary 17.** *R and L are fully faithful.*

▶ Note. As pointed out earlier, in the particular case where we choose $C = D = M$ (as in the case of lenses), then R and L both provide a fully-faithful *and monoidal* embedding of the arrows in M into diagrams.

4.3 Refining the Representation Theorem

Together, simple arrows and the cap are enough to represent any optic as a string diagram.

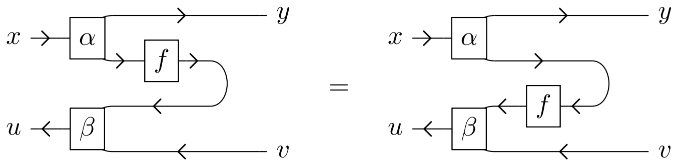
▶ **Theorem 18.** *Given* $\alpha : C(x, m \odot y)$ *and* $\beta : D(m \odot v, u)$*, the optic* $l := \langle \alpha \mid \beta\rangle_m$ *can be represented as follows:*



(10)

Proof. By calculating the composition of the pair of simple arrows with the cap; see appendix A.5. ◀

▶ Note. Recall that the pairs $\langle \alpha \mid \beta \rangle_m$ are defined modulo an equivalence relation. How is this compatible with the diagrammatic notation? The equivalence says that $\langle \alpha \mathbin{;} (f \odot y) \mid \beta \rangle_m = \langle \alpha \mid (f \odot v) \mathbin{;} \beta \rangle_n$; diagrammatically, this becomes:



(11)

Which we already know holds, by sliding f along the bent wire!

5 Applications

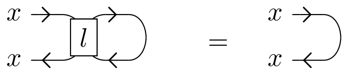
We present two examples of applications of the calculus that illustrate its expressivity.

5.1 Lawful Optics

One of the most striking consequences of this calculus (and the question that led to its discovery) is the neatness with which it can express optic laws.

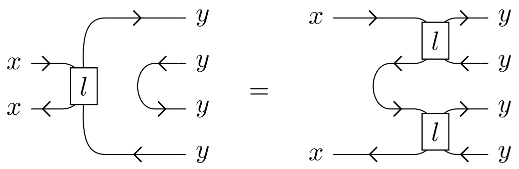
As originally constructed by the Haskell community [9], optics were required to abide by certain round-trip laws that ensure coherence of their operations. Those laws in particular coincide with very-well-behavedness [3] in the case of lenses, which we investigate in more detail in the next section. Riley formalized those laws in a general form [14, Section 3], but the result is rather hard to manipulate. The string calculus enables an alternative (and equivalent) description that is purely diagrammatic:

▶ **Definition 19.** *An optic $l : \binom{x}{x} \to \binom{y}{y}$ is said to be lawful when*



(12)

and



(13)

▶ Note. We can see that lawful optics are exactly the homomorphisms for the "pair-of-pants" comonoid made from pairs of oriented wires. Interestingly, if we view this comonoid as a procomonad on C, then lawful optics are in bijection with its coalgebras on the carrier R_x. This is a significant generalization of the result by O'Connor [11] that lawful lenses are the coalgebras for the store comonad: here the "pair-of-pants" procomonad precisely generalizes the store comonad.

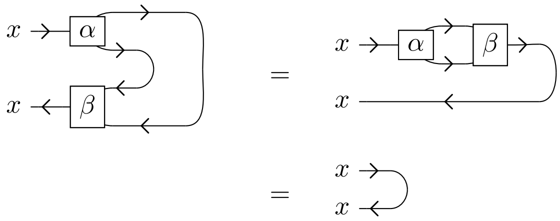
▶ **Theorem 20.** *This notion of lawfulness is equivalent to the one defined by Riley in [14, Section 3].*

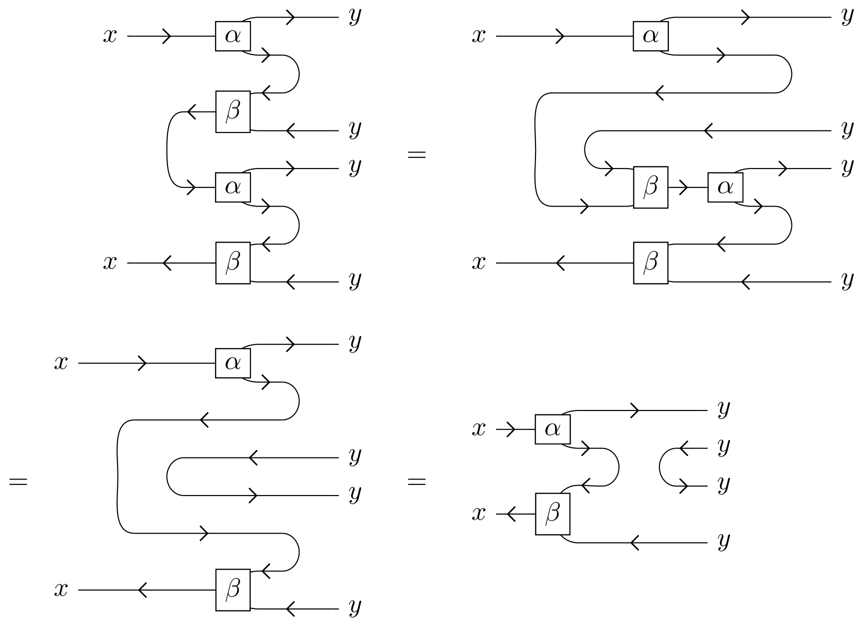
Proof. See appendix A.6. ◀

Thus this diagrammatic definition captures properly the useful and very general notion of lawfulness for optics. Using this theorem, many properties of lawfulness can be derived purely diagrammatically. As an example, let us reprove [14, Proposition 3.0.4]:

▶ **Proposition 21** ([14, Proposition 3.0.4]). *If α and β are mutual inverses, then the optic $\langle \alpha \,|\, \beta \rangle_m$ is lawful.*

Proof.



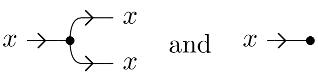


◀

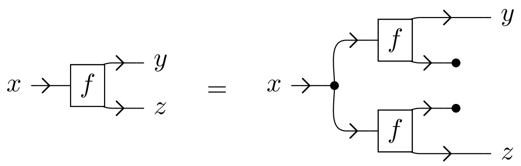
5.2 Cartesian Lenses

The canonical special case of optics, that we mentioned in Example 4, is cartesian lenses. They arise when we restrict ourselves to $C = D = M$ and the monoidal product of C is cartesian.

In this setting, we have two important gadgets in C: duplication and deletion, corresponding respectively to the diagonal map $C(x, x \times x)$ and the terminal map $C(x, I)$. Diagrammatically, we represent them as follows:



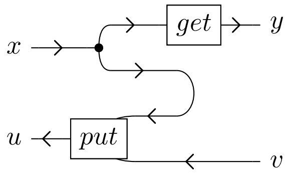
▶ **Lemma 22.** *Given $f : C(X, Y \times Z)$, we have*



(14)

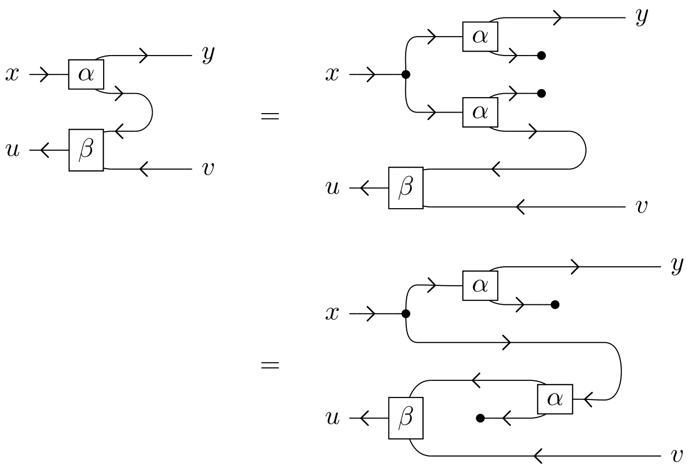
Proof. This corresponds to the standard fact that $f = \langle fst \circ f, snd \circ f \rangle$. ◀

▶ **Theorem 23.** *A lens $l : \left(\begin{smallmatrix}x\\u\end{smallmatrix}\right) \to \left(\begin{smallmatrix}y\\v\end{smallmatrix}\right)$ can be expressed as:*

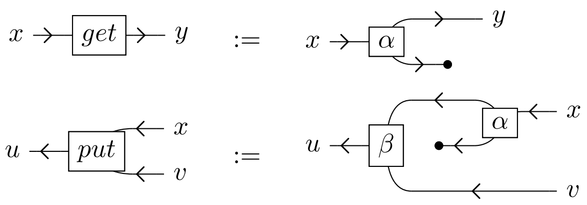


for some $get : C(x, y)$ and $put : C(x \times v, u)$.

Proof.



which has the required shape. We have:

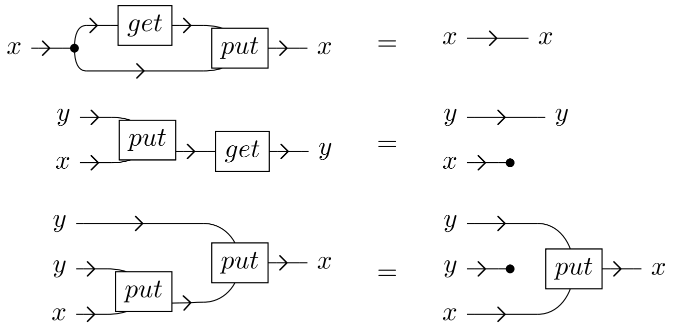


◀

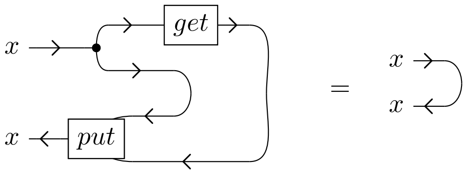
▶ Note. Observe that it is diagrammatically clear that the definition of *put* and *get* in terms of $\langle \alpha \,|\, \beta \rangle_m$ respects the equivalence relation induced by the coend.

We recovered purely diagrammatically the usual formulation of lenses in terms of *get* and *put*, that we had derived in Example 4. In this setting, various properties of lenses can be investigated purely diagrammatically. As an example, let us revisit [14, Proposition 3.0.3], which captures the fact that the general notion of lawfulness for optics coincides with the familiar PutGet, GetPut and PutPut laws [3] (together called "very-well-behavedness") in the case of lenses.

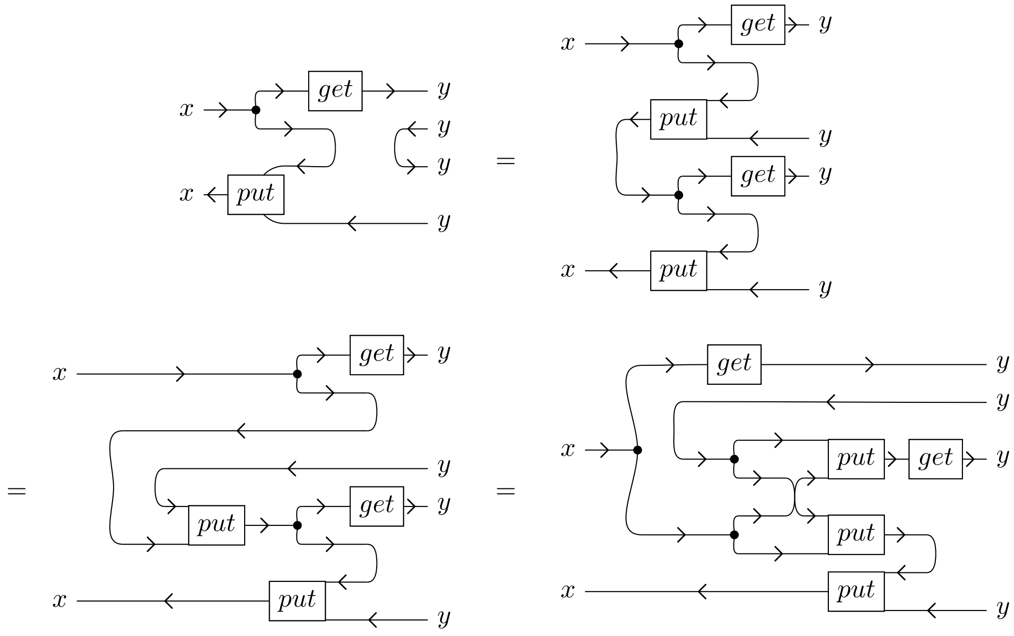
▶ **Proposition 24** ([14, Proposition 3.0.3])**.** *A lens* $l : \left(\begin{smallmatrix}x\\x\end{smallmatrix}\right) \to \left(\begin{smallmatrix}y\\y\end{smallmatrix}\right)$ *is lawful iff the following three laws (respectively called PutGet, GetPut and PutPut) hold in C:*



Proof. Diagrammatically, the fact that a lens is lawful reads:



which is exactly the PutGet law, and:



It is straightforward to see that the PutPut and the GetPut laws together entail this equality. By applying the deletion map successively to the outputs, one can also show that this equation entails those two laws, when y is inhabited. ◀

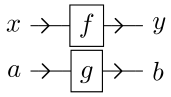
5.3 Effectful Lenses

We now turn to a less common example: effectful lenses. They stem from the desire to allow lenses to perform effects while retrieving or updating data. Various approaches have been proposed; see Abou-Saleh et al. [1] for an overview.

Let C be a cartesian category and T a monad on C. We would like an optic that resembles cartesian lenses from the previous section, but with effectful arrows. This means that we would like our arrows to live in the Kleisli category C_T. This category however is rarely monoidal, let alone cartesian: for it to be monoidal, the monad T would need to be commutative, which rules out large classes of effects that we might want to use. Thus we cannot reuse the results from the previous section. Here we can instead make good use of the generality of monoidal actions: C_T may not be monoidal, but when T is strong (which is rather common), the product of C extends to an action of C on C_T [14, Proposition 4.9.3]. This is enough to define an optic for monadic lenses:

$$MLens_T(\left(\begin{smallmatrix} x \\ u \end{smallmatrix}\right), \left(\begin{smallmatrix} y \\ v \end{smallmatrix}\right)) := \int^{c:C} C_T(x, c \times y) \times C_T(c \times v, u)$$

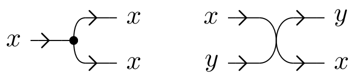
Let us now investigate the diagrams for such an optic. Recall the details of how oriented wires are typed. Here the acting category is C, which means that in a diagram like the following, the typing rules enforce that x, y and f can live in C_T, but a, b and g can only live in C.



This is why we don't need C_T to be monoidal: this calculus only allows an arrow in C_T to be tensored with arrows in C. This gives us a string diagram calculus where otherwise none would have been possible.

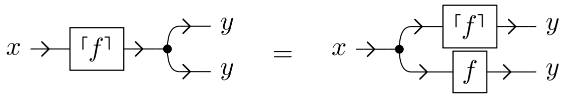
The distinction between effectful maps (in C_T) and pure maps (in C) is an important aspect of this calculus. Note that every pure map f can be lifted to an effectful map written $\ulcorner f \urcorner$, via a canonical functor. This functor also respects the actegory structures, and therefore allows us to embed the pure lenses from the previous section as monadic lenses.

This calculus even inherits some of the diagrammatic features of the previous section: the duplication map and the swap still exist and are represented as before.



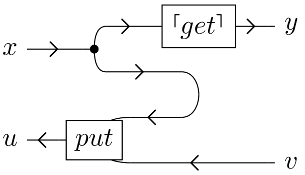
(15)

The difference is that the bottom wire can only carry maps living in C. Whereas before, all maps could be "slid through" the duplication map and swap, now only C-maps (aka pure maps) can:

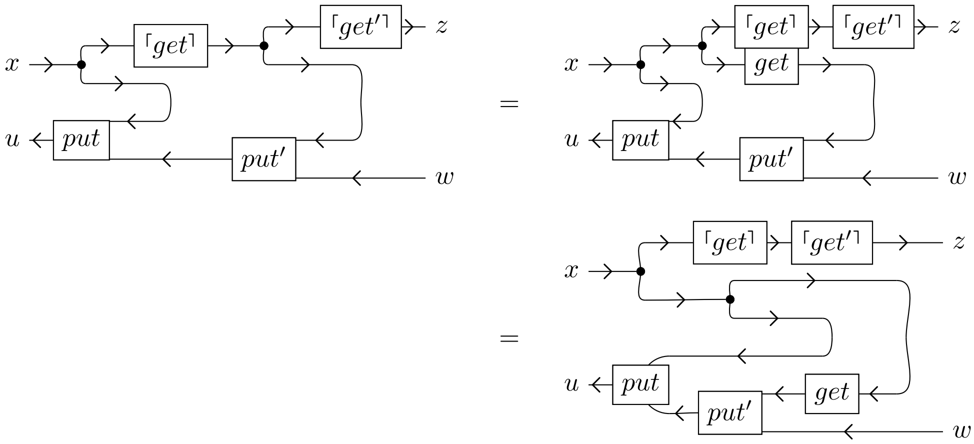


(16)

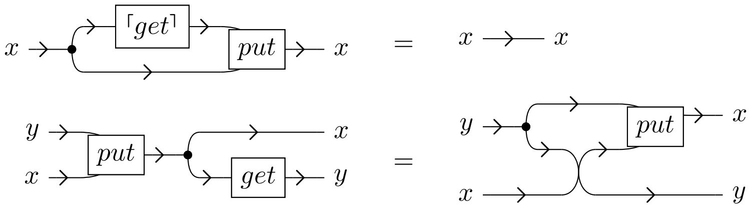
We now restrict ourselves to the monadic lenses proposed by Abou-Saleh et al. [1]. Those lenses are modeled closer to ordinary lenses, in particular their definition does not involve a coend. They are composed of $get : C(x, y)$ and $put : C_T(x \times v, u)$. Diagrammatically they look quite like cartesian lenses:



Note that *get* is required to be pure. This is important to ensure that composing two such lenses stays of that simplified shape. This non-trivial fact can be seen diagrammatically in what follows: if *get* was not pure, it couldn't be slid across the duplication map.


x
$\ulcorner get \urcorner$
$\ulcorner get' \urcorner$
z
u
put
put'
w
=
x
$\ulcorner get \urcorner$
$\ulcorner get' \urcorner$
z
get
u
put
put'
w
=
x
$\ulcorner get \urcorner$
$\ulcorner get' \urcorner$
z
u
put
put'
get
w

Finally, this new calculus can express the laws proposed by Abou-Saleh et al. [1], making them much easier to reason about:



6 Conclusion and Future Work

We have presented a calculus that flowed naturally from the Yoneda embedding of optics into Tambara modules. We have shown that it was well-suited for expressing common properties of optics and proving useful theorems generally, some of which would otherwise be painful to prove. This work however is only the start: it provides the basis of a calculus, whose expressive power hasn't yet been explored in the plethora of topics where optics have found a use. In particular, we expect new specific diagrammatic properties like those of lenses to arise for other kinds of optics like prisms or traversals.

Then, the calculus could be linked with related constructions, like the calculus for teleological categories from [4], or the Int construction from [7].

Properties of *Tamb* as a bicategory also seem worth exploring, in particular its strong similarity with *Prof*, and the link between the properties of M and those of *Tamb*.

Finally, diagrams in *Tamb* with multiple ingoing and outgoing legs seem to relate to combs as in [8] and dialogues in the style of [6]; there is potential for using *Tamb* to provide a basis for general diagrammatic descriptions of those objects.

References

1 Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. Reflections on Monadic Lenses. *A List of Successes That Can Change the World*, 9600:1–31, April 2016. doi:10.1007/978-3-319-30936-1_1.

2 Guillaume Boisseau. String Diagrams for Optics. *arXiv:2002.11480 [math]*, February 2020. arXiv:2002.11480.

3 J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 233–246. ACM, 2005. doi:10.1145/1040305.1040325.

4 Jules Hedges. Coherence for lenses and open games. *arXiv:1704.02230 [cs, math]*, September 2017. arXiv:1704.02230.

5 Jules Hedges. Lenses for Philosophers, August 2018. URL: https://julesh.com/2018/08/16/lenses-for-philosophers/.

6 Jules Hedges. The game semantics of game theory. *arXiv:1904.11287 [cs]*, April 2019. arXiv:1904.11287.

7 André Joyal, Ross Street, and Dominic Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3):447–468, April 1996. doi:10.1017/S0305004100074338.

8 Aleks Kissinger and Sander Uijlen. A categorical semantics for causal structure. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12. IEEE Computer Society, 2017. doi:10.1109/LICS.2017.8005095.

9 Edward Kmett. Haskell lens library, February 2020. URL: https://hackage.haskell.org/package/lens.

10 nLab authors. Actegory. URL: http://ncatlab.org/nlab/show/actegory.

11 Russell O'Connor. Lenses Are Exactly the Coalgebras for the Store Comonad, November 2010. URL: https://r6research.livejournal.com/23705.html.

12 Craig Pastro and Ross Street. Doubles for monoidal categories. *Theory and Applications of Categories*, 21, November 2007. arXiv:0711.1859.

13 Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. Profunctor optics: Modular data accessors. *Programming Journal*, 1(2):7, March 2017. doi:10.22152/programming-journal.org/2017/1/7.

14 Mitchell Riley. Categories of Optics. *arXiv:1809.00738 [math]*, September 2018. arXiv:1809.00738.

15 Mario Román. *Profunctor Optics and Traversals*. Msc Thesis, University of Oxford, January 2020. arXiv:2001.08045.

A Proofs

A.1 R Respects the Actegory Structure (Proposition 11)

Proof (Proposition 11).

$$\begin{aligned}
R_I &= M(-, = \odot_M I) \\
&= M(-, = \otimes I) \\
&\cong M(-, =)
\end{aligned}$$

$$\begin{aligned}
R_x \otimes R_m &= \int^{n:M} C(-, n \odot_C x) \times M(n, = \odot_M m) \\
&= \int^{n:M} C(-, n \odot_C x) \times M(n, = \otimes m) \\
&\cong C(-, (= \otimes m) \odot_C x) \\
&\cong C(-, = \odot_C (m \odot_C x)) \\
&= R_{m \odot_C x}
\end{aligned}$$

$$\begin{aligned}
L_I &= M(- \odot_M I, =) \\
&= M(- \otimes I, =) \\
&\cong M(-, =)
\end{aligned}$$

$$\begin{aligned}
L_m \otimes L_x &= \int^{n:M} M(- \odot_M m, n) \times C(n \odot_C x, =) \\
&= \int^{n:M} M(- \otimes m, n) \times C(n \odot_C x, =) \\
&\cong C((- \otimes m) \odot_C x, =) \\
&\cong C(- \odot_C (m \odot_C x), =) \\
&= L_{m \odot_C x}
\end{aligned}$$

It is easy to check that the corresponding strengths coincide as well. ◀

A.2 R and L Are Adjoint (Proposition 12)

Proof (Proposition 12). The counit $\varepsilon : R_x \otimes L_x \to C(-, =)$ of the adjunction in *Prof* is given by composition in C. We need it to commute with strength:

$$\begin{tikzcd}
\int^{b} C(a, b \odot x) \otimes C(b \odot x, c) \arrow[r, "\fatsemi"] \arrow[d, "strength"'] & C(a, c) \arrow[d, "strength"] \\
\int^{b'} C(m \odot a, b' \odot x) \otimes C(b' \odot x, m \odot c) \arrow[r, "\fatsemi"] & C(m \odot a, m \odot c)
\end{tikzcd}$$

We inline the definition of *strength*, and move the coends out by continuity, to get an equivalent square:

$$\begin{tikzcd}
C(a, b \odot x) \otimes C(b \odot x, c) \arrow[r, "\fatsemi"] \arrow[d, "(m\odot -)\otimes(m\odot -)"'] & C(a,c) \arrow[d, "(m\odot -)"] \\
C(m \odot a, m \odot (b \odot x)) \otimes C(m \odot (b \odot x), m \odot c) \arrow[r, "\fatsemi"] \arrow[d, "C(id, a^{-1})\otimes C(a, id)"'] & C(m \odot a, m \odot c) \arrow[d, "id"] \\
C(m \odot a, (m \otimes b) \odot x) \otimes C((m \otimes b) \odot x, m \odot c) \arrow[r, "\fatsemi"] & C(m \odot a, m \odot c)
\end{tikzcd}$$

The top square commutes by functoriality of $(m \odot -)$; the bottom one by the fact that $a^{-1} \fatsemi a = id$.

Similarly, the unit also lives in *Tamb*. This is enough for the adjunction to lift from *Prof* to *Tamb*. ◀

A.3 Diagram for Simple Arrows (Lemma 15)

Proof (Lemma 15). The diagram corresponds to the 2-cell $R_f \otimes L_g$.

It has type

$$\begin{aligned}
R_f \otimes L_g &: R_x \otimes L_u \to R_y \otimes L_v \\
&= \int_{ab} \Big(\int^m R_x(a,m) \times L_u(m,b)\Big) \to \Big(\int^m R_y(a,m) \times L_v(m,b)\Big)
\end{aligned}$$

And value

$$\begin{aligned}
(R_f \otimes L_g)(\langle p \,|\, q\rangle_m) &= \langle R_f(p) \,|\, L_g(q)\rangle_m \\
&= \langle p \fatsemi (m \odot f) \,|(m \odot g) \fatsemi q\rangle_m
\end{aligned}$$

To get the preimage through Y, we apply this map to the identity optic.

$$\begin{aligned}
(R_f \otimes L_g)(id_{\left(\begin{smallmatrix}x\\u\end{smallmatrix}\right)}) &= (R_f \otimes L_g)(\langle \lambda_x^{-1} \,|\, \lambda_u\rangle_I) \\
&= \langle \lambda_x^{-1} \fatsemi (I \odot f) \,|(I \odot g) \fatsemi \lambda_u\rangle_I \\
&= \langle f \fatsemi \lambda_y^{-1} \,|\, \lambda_v \fatsemi g\rangle_I
\end{aligned}$$

◀

A.4 Simple Arrows Embed Fully-Faithfully (Theorem 16)

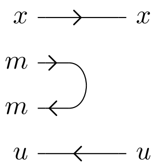
Proof (Theorem 16). We calculate:

$$\begin{aligned}
&Optic_{C,M}(\left(\begin{smallmatrix}x\\I\end{smallmatrix}\right), \left(\begin{smallmatrix}y\\I\end{smallmatrix}\right)) \\
&= \int^m C(x, m \odot_C y) \times M(m \odot_M I, I) \\
&= \int^m C(x, m \odot_C y) \times M(m \otimes I, I) \\
&\cong \int^m C(x, m \odot_C y) \times M(m, I) \\
&\cong C(x, I \odot_C y) \\
&\cong C(x, y)
\end{aligned}$$

By following the isomorphisms, we get that the reverse direction is the function $f : C(x,y) \mapsto \langle f \fatsemi \lambda_y^{-1} \,|\, \lambda_I\rangle_I$, which as we saw previously corresponds to $f \mapsto \iota(f, id_I) = R_f \otimes L_{id_I} = R_f$. ◀

A.5 Representation Theorem (Theorem 18)

▶ **Lemma 25.** *The optic corresponding to this diagram is* $\langle id_{m\odot x} \mid id_{m\odot u}\rangle_m$.



Proof. Let us name the map corresponding to this diagram $\Lambda_{x,m,u}$.

Knowing the action of the cap ε, we obtain by a tedious calculation that we will omit here:

$$\begin{aligned}\Lambda_{x,m,u} &: Y\binom{m\odot x}{m\odot u} \to Y\binom{x}{u}\\ &= \langle \alpha \mid \beta\rangle_n \mapsto \langle \alpha \fatsemi a^{-1}_{n,m,x} \mid a_{n,m,u} \fatsemi \beta\rangle_{n\otimes m}\end{aligned}$$

Thus the corresponding optic is:

$$\begin{aligned}\Lambda_{x,m,u}(id_{\binom{m\odot x}{m\odot u}}) &= \Lambda_{x,m,u}(\langle \lambda^{-1}_{m\odot x} \mid \lambda_{m\odot u}\rangle_I)\\ &= \langle \lambda^{-1}_{m\odot x} \fatsemi a^{-1}_{I,m,x} \mid a_{I,m,u} \fatsemi \lambda_{m\odot u}\rangle_{I\otimes m}\\ &= \langle \lambda^{-1}_m \odot x \mid \lambda_m \odot u\rangle_{I\otimes m}\\ &= \langle (\lambda^{-1}_m \fatsemi \lambda_m) \odot x \mid id_{m\odot u}\rangle_m\\ &= \langle id_{m\odot x} \mid id_{m\odot u}\rangle_m\end{aligned}$$ ◀

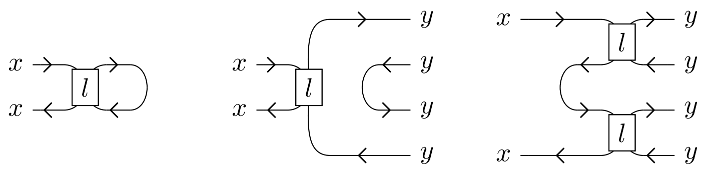
Proof (Theorem 18). The right-hand-side diagram is the composition of two optics of which we know the value: the first is $\langle \alpha \fatsemi \lambda^{-1}_{m\odot y} \mid \lambda_{m\odot v} \fatsemi \beta\rangle_I$; the second is $\langle id_{m\odot y} \mid id_{m\odot v}\rangle_m$.

The resulting optic is thus their composition:

$$\begin{aligned}&\langle \alpha \fatsemi \lambda^{-1}_{m\odot y} \mid \lambda_{m\odot v} \fatsemi \beta\rangle_I \fatsemi \langle id_{m\odot y} \mid id_{m\odot v}\rangle_m\\ &= \langle \alpha \fatsemi \lambda^{-1}_{m\odot y} \fatsemi (I \odot id_{m\odot y}) \fatsemi a^{-1}_{m,I} \mid a_{m,I} \fatsemi (I \odot id_{m\odot v}) \fatsemi \lambda_{m\odot v} \fatsemi \beta\rangle_{I\otimes m}\\ &= \langle \alpha \fatsemi \lambda^{-1}_{m\odot y} \fatsemi a^{-1}_{m,I} \mid a_{m,I} \fatsemi \lambda_{m\odot v} \fatsemi \beta\rangle_{I\otimes m}\\ &= \langle \alpha \fatsemi (\lambda^{-1}_I \odot y) \mid (\lambda_I \odot v) \fatsemi \beta\rangle_{I\otimes m}\\ &= \langle \alpha \fatsemi (\lambda^{-1}_I \odot y) \fatsemi (\lambda_I \odot y) \mid \beta\rangle_m\\ &= \langle \alpha \mid \beta\rangle_m\end{aligned}$$ ◀

A.6 Lawfulness in Diagrams (Theorem 20)

Proof (Theorem 20). Lawfulness in [14, Section 3] is based on three maps named *outside*, *once*, and *twice*. Unpacking the definitions, those three maps applied to an optic l correspond respectively to the three diagrams:



The interesting insight is that the complicated $Optic^2_M$ coend from Riley's paper can be easily constructed diagrammatically by tensoring oriented wires as above. The theorem then follows directly from Riley's definition of lawfulness. ◀

A Reflection on Continuation-Composing Style

Dariusz Biernacki
Insitute of Computer Science, University of Wrocław, Poland
`http://ii.uni.wroc.pl/~dabi/`
dabi@cs.uni.wroc.pl

Mateusz Pyzik
Institute of Computer Science, University Wrocław, Poland
matp@cs.uni.wroc.pl

Filip Sieczkowski
Institute of Computer Science, University Wrocław, Poland
efes@cs.uni.wroc.pl

Abstract

We present a study of the continuation-composing style (CCS) that describes the image of the CPS translation of Danvy and Filinski's `shift` and `reset` delimited-control operators. In CCS continuations are composable rather than abortive as in the traditional CPS, and, therefore, the structure of terms is considerably more complex. We show that the CPS translation from Moggi's computational lambda calculus extended with `shift` and `reset` has a right inverse and that the two translations form a reflection i.e., a Galois connection in which the target is isomorphic to a subset of the source (the orders are given by the reduction relations). Furthermore, we use this result to show that Plotkin's call-by-value lambda calculus extended with `shift` and `reset` is isomorphic to the image of the CPS translation. This result, in particular, provides a first direct-style transformation for delimited continuations that is an inverse of the CPS transformation up to syntactic identity.

2012 ACM Subject Classification Theory of computation → Control primitives; Theory of computation → Lambda calculus

Keywords and phrases delimited control, continuation-passing style, reflection, call-by-value lambda calculus, computational lambda calculus

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.18

Acknowledgements We thank the anonymous reviewers and participants of NWPT'19 as well as the anonymous reviewers of FSCD 2020 for helpful comments on the presentation of this work.

1 Introduction

In higher-order programming languages based on the λ-calculus, continuation-passing style (CPS) is a program format in which functions accept an additional parameter – a continuation – that represents the entire rest of the computation [17]. In CPS, computations are explicitly sequentialised according to a given evaluation strategy, the intermediate results are named, and there are no nested function calls, i.e., all function calls are tail calls. A notion associated with CPS is the notion of a CPS translation that transforms a term in direct style, i.e., where continuations are not passed around, to the corresponding term in CPS [8, 16]. Such translations have been routinely used both to define continuation semantics of higher-order programs, where object-level constructs are CPS-translated to the meta-level λ-calculus [21, 18], and as a compilation step bridging the gap between higher-order and low-level languages [23, 1].

In the context of compilation, a critical concern is correctness of the CPS translation. In his seminal work [16], Plotkin introduced the call-by-value lambda calculus λ_v, equipped with a reduction $\rightarrow$ and equality $=$ theories, along with a CPS translation $*$ to the call-by-name lambda calculus λ_n, for which he proved equational soundness, i.e., $M = N$ in

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 18; pp. 18:1–18:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the source implies $M^* = N^*$ in the target, and he showed that the converse does not hold, i.e., the translation is not complete. Completeness has been obtained by Moggi who devised a monad translation, including a CPS translation as a special case, from the computational lambda calculus λ_c (an extension of λ_v) to the monadic metalanguage λ_{ml} (equipped with an equational theory only) [15]. This result was strengthened by Hatcliff and Danvy [11], who showed that Moggi's monad translation $*$ has an inverse translation $\#$ such that $M = M^{*\#}$ in the source and $N^{\#*} = N$ in the target, i.e., it is an equational correspondence. Finally, Sabry and Felleisen further improved on Plotkin's result by devising a CPS translation that forms an equational correspondence between λ_c and λ_n [19].

Whereas all these results concern equality theories, Sabry and Wadler obtained stronger results in which equality is replaced with reduction (viewed as directed code optimisation) [20]. In particular, they presented a CPS translation $*$ from λ_c to λ_n along with its inverse $\#$ that form a Galois connection satisfying:

- soundness: if $M \twoheadrightarrow N^\#$ in the source then $M^* \twoheadrightarrow N$ in the target;
- completeness: if $M^* \twoheadrightarrow N$ in the target then $M \twoheadrightarrow N^\#$ in the source;

This means that evaluation in the source language is equivalent to compiling, evaluating in the target and decompiling. Moreover, this Galois connection is a reflection by satisfying an additional condition: compiling is a left inverse to decompiling, $M^{\#*} \equiv M$, where $\equiv$ is syntactic identity. Interestingly, Danvy developed a direct-style transformation from λ_n to λ_v that is a left inverse to Plotkin's CPS translation [2], but he did not consider reduction or equality theories.

A particularly interesting application of CPS is in defining the semantics of control operators, i.e., constructs that access and manipulate the continuation [17, 24, 5]. For abortive control operators such as `call/cc`, which model jumps, the image of the CPS translation is more challenging to characterise than in the pure case. The main reason is that in the pure case one continuation identifier suffices, whereas the abortive control operators may use any of the lexically visible continuation identifiers – continuations can be used out of turn. Sabry and Felleisen [19] considered an extension of Felleisen et al.'s λ_v-$\mathcal{C}$-calculus (including `call/cc` and the `abort` operator) [6], and they showed a CPS-translation to λ_n that forms an equational correspondence. A direct-style translation for `call/cc` was also developed by Danvy and Lawall [4]. Their transformation is related to the CPS translation via a Galois connection, induced from the translations and based on the the syntactic structure of terms rather than on reduction relations.

In this work, we study the continuation-composing style (CCS), which arises as the image of the CPS translation of Danvy and Filinski's delimited-control operators `shift` and `reset` [3]. In CCS continuations are composable rather than abortive, which means that not all calls are tail calls and the conditions imposed on where continuation identifiers occur in terms are further relaxed. Continuation composability is central to the expressibility of arbitrary computational effects with continuations [25, 7]. There exist some work devoted to the image of the double CPS translation of `shift` and `reset` in which a meta-continuation is introduced to eliminate nested computations of CCS [3]. Most notably, Kameyama and Hasegawa introduced a direct-style translation which led to a direct-style equational characterisation of the image of the double CPS translation with $\beta\eta$-equality (a subset of λ_n) [12]. However, we are not aware of any published study of the reduction theory of CCS, and the goal of the present work is to fill this vacuum.

To that end, we follow the programme of Sabry and Wadler [20], and we construct a reflection of CCS in two calculi with `shift`, considered as a combinator, and `reset`. We first focus on the $\lambda_{c\mathcal{S}}$-calculus, which is Moggi's computational lambda calculus λ_c, extended with `shift` and `reset` for which we give a CPS translation that eliminates administrative redexes. The image of the translation is $\lambda^{*}_{c\mathcal{S}}$, a call-by-value lambda calculus equipped with a set of dedicated reduction rules. We then define a direct-style translation from this calculus back to a subset of $\lambda_{c\mathcal{S}}$ that we call $\lambda^{\triangleright}_{c\mathcal{S}}$ (the kernel of $\lambda_{c\mathcal{S}}$) and we prove that it is a right inverse to the CPS translation. We then show that the two translations form a reflection with respect to orders given by the respective reduction relations and that the reflection decomposes into an inclusion of $\lambda_{c\mathcal{S}}$ in the kernel $\lambda^{\triangleright}_{c\mathcal{S}}$ and an order isomorphism of $\lambda^{\triangleright}_{c\mathcal{S}}$ and $\lambda^{*}_{c\mathcal{S}}$.

Second, we consider $\lambda_{\mathcal{S}}$, a subcalculus of $\lambda_{c\mathcal{S}}$ that is a more traditional calculus of delimited control and that coincides with λ_v extended with `shift` and `reset`. Building on the results for $\lambda_{c\mathcal{S}}$ and restricting the CPS translation to $\lambda_{\mathcal{S}}$, we show that $\lambda_{\mathcal{S}}$ is isomorphic to $\lambda^{*}_{\mathcal{S}}$, its image through the CPS translation and a subcalculus of $\lambda^{*}_{c\mathcal{S}}$. A byproduct of this development is a one-pass direct-style translation for delimited-control operators, a long missing continuation of the work by Danvy for pure call-by-value lambda calculus [2], and by Danvy and Lawall for abortive control operators [4]. Such transformations make it possible to automatically map continuation-passing programs to their more concise, but at the same time more challenging to design, direct-style counterparts.

The remainder of this article is structured as follows. In Section 2, we briefly introduce the basic notions related to Galois connections and reflections. In Section 3, we introduce the calculi $\lambda_{c\mathcal{S}}$ and $\lambda^{*}_{c\mathcal{S}}$ along with the CPS translation from $\lambda_{c\mathcal{S}}$ to $\lambda^{*}_{c\mathcal{S}}$. In Section 4, we characterise the image of the CPS translation and we define its right inverse – the direct style translation. In Section 5, we prove that the two translations form a reflection and we identify the kernel of the reflection in $\lambda_{c\mathcal{S}}$. In Section 6, we show that when restricted to $\lambda_{\mathcal{S}}$, the CPS translation has an inverse such that the two transformations form an isomorphism. We conclude in Section 7.

2 Galois Connections and Reflections

Below, we recall the essential facts about Galois connections that we use throughout the article. We refer the reader to Sabry and Wadler's work [20] for more detailed background. We treat each set A as equipped with a preorder (i.e., reflexive and transitive) relation $\twoheadrightarrow_A$. In our development, we define these in two ways: either by applying reflexive-transitive closure on a reduction relation $\to_A$, or by truncating a preorder (multi-step reduction) relation $\twoheadrightarrow_X$ of a superset $X \supseteq A$ ($\twoheadrightarrow_A$ is then an induced preorder). In the following, $\equiv_A$ denotes the syntactic identity on A .

In the following, it may be helpful to think about A and B as a source and target calculi, respectively, whereas f and g can be thought of us compiling and decompiling, respectively. We start with the standard notion of monotonicity, i.e, preservation of reduction by the compiling map.

▶ **Definition 1** (Monotone function). *A function $f : A \to B$ is monotone if, and only if $\forall x_1, x_2 \in A \,.\, x_1 \twoheadrightarrow_A x_2 \implies f(x_1) \twoheadrightarrow_B f(x_2)$.*

A Galois connection expresses a form of harmony of compiling and decompiling with respect to reduction relations.

▶ **Definition 2** (Galois connection). *Monotone functions $f : A \to B$ and $g : B \to A$ form a Galois connection if, and only if $a \twoheadrightarrow_A g(b) \iff f(a) \twoheadrightarrow_B b$.*

There is an alternative characterisation of a Galois connection.

$$
\begin{array}{llcl}
\text{terms} & L, M, N & ::= & V \mid P \\
\text{values} & V, W & ::= & x \mid \lambda x\,.\,M \mid \mathcal{S} \\
\text{nonvalues} & P, Q & ::= & M\;N \mid \text{let}\, x = M \,\text{in}\, N \mid \langle M \rangle \\
\text{pure contexts} & J, K & ::= & [\,] \mid K\;M \mid V\;K \mid \text{let}\, x = K \,\text{in}\, M
\end{array}
$$

$$
\begin{array}{llcl}
(\beta.v) & (\lambda x\,.\,M)\,V & \to & M[x := V] \\
(\eta.v) & \lambda x\,.\,V\,x & \to & V \\
(\beta.let) & \text{let}\, x = V \,\text{in}\, M & \to & M[x := V] \\
(\eta.let) & \text{let}\, x = M \,\text{in}\, x & \to & M \\
(assoc) & \text{let}\, x = \text{let}\, y = L \,\text{in}\, M \,\text{in}\, N & \to & \text{let}\, y = L \,\text{in}\, \text{let}\, x = M \,\text{in}\, N \\
(let.1) & P\,N & \to & \text{let}\, x = P \,\text{in}\, x\,N \\
(let.2) & V\,Q & \to & \text{let}\, y = Q \,\text{in}\, V\,y \\
(\beta.\mathcal{S}) & \langle J[\mathcal{S}\;N] \rangle & \to & \langle N\,(\lambda y\,.\langle J[y] \rangle) \rangle \\
(\beta.\mathcal{R}) & \langle V \rangle & \to & V
\end{array}
$$

Figure 1 Direct style calculus $\lambda_{c\mathcal{S}}$.

▶ **Theorem 3** (Equivalent definition of Galois connection). *Monotone functions $f : A \to B$ and $g : B \to A$ form a Galois connection if, and only if*

- $a \twoheadrightarrow_A g(f(a))$ *and*
- $f(g(b)) \twoheadrightarrow_B b$.

When compiling is a left inverse to decompiling, then we have a reflection.

▶ **Definition 4** (Reflection). *A Galois connection $(f : A \to B, g : B \to A)$ is a reflection if, and only if $f(g(b)) \equiv_B b$.*

In case compiling is also a right inverse to decompiling, we have an isomorphism.

▶ **Definition 5** (Order isomorphism). *A reflection $(f : A \to B, g : B \to A)$ is an order isomorphism if, and only if $a \equiv_A g(f(a))$.*

Every reflection factors into an inclusion and an order isomorphism.

▶ **Theorem 6** (Reflection decomposition). *Every reflection $(f : A \to B, g : B \to A)$ decomposes into a reflection (called* inclusion*) $(g \circ f : A \to g[B], id_{g[B]} : g[B] \to A)$ and an order isomorphism $(f : g[B] \to B, g : B \to g[B])$, where $g[B] \subseteq A$ has an induced preorder.*

It follows from Theorem 6 that given a reflection $(f : A \to B, g : B \to A)$, the source calculus has a kernel $g[B]$ (or equivalently, $g[f[A]]$) that is isomorphic with B, or that reflects B. The goal of this work is to identify such a reflection of CCS in call-by-value lambda calculi with delimited continuations.

3 Delimited-Control Operators Shift and Reset

We begin with Moggi's calculus of computations, λ_c, extended with shift and reset delimited control operators, which we dub $\lambda_{c\mathcal{S}}$. The syntax and semantics are presented in Figure 1. The terms of the calculus are divided into values, which include variables, lambda abstractions and the shift combinator $\mathcal{S}$, and computations, which include applications, let-bindings and the reset operator, which serves to delimit the scope of the continuation. Moreover, we introduce the syntactic domain of pure evaluation contexts, which encode a left-to-right call-by-value evaluation strategy and, crucially do not contain the reset operators.

$$
\begin{aligned}
&* : \lambda_{c\mathcal{S}} \to \lambda \\
&M^* && = \lambda\, k\, .(M : k) \\
&V : K && = K\, V^\dagger \\
&(P\, Q) : K && = P : (\lambda\, x\, .(Q : (\lambda\, y\, .\, x\, y\, K))) \\
&(P\, W) : K && = P : (\lambda\, x\, .\, x\, W^\dagger\, K) \\
&(V\, Q) : K && = Q : (\lambda\, y\, .\, V^\dagger\, y\, K) \\
&(V\, W) : K && = V^\dagger W^\dagger K \\
&(\mathrm{let}\, x = M\, \mathrm{in}\, N) : K && = M : (\lambda\, x\, .(N : K)) \\
&\langle M \rangle : K && = K\, (M : (\lambda\, x\, .\, x)) \\
&x^\dagger && = x \\
&(\lambda\, x\, .\, M)^\dagger && = \lambda\, x\, .\, M^* \\
&\mathcal{S}^\dagger && = \lambda\, w\, j\, .\, w\, (\lambda\, y\, k\, .\, k\, (j\, y))\, (\lambda\, x\, .\, x)
\end{aligned}
$$

Figure 2 Conversion from $\lambda_{c\mathcal{S}}$ to Continuation-Composing Style.

The operational semantics of the calculus is given by a contraction relation, which may be performed within any context, as we consider general reduction rather than evaluation. Nonetheless, we still require the pure fragment of evaluation contexts: these are used to match the shift operator with the enclosing reset by the $(\beta.\mathcal{S})$ rule. This rule matches a $\mathcal{S}$ operator applied to some term N in a pure context J closed by a reset operator, *captures* the latter context (together with the reset), reifies it as a function and passes it as an argument to N. Note the duplication of the reset operator in the contractum, which is an important characteristic of shift/reset [22].

Except for $(\beta.\mathcal{S})$ and the simple $(\beta.\mathcal{R})$ rule, the rules are those of λ_c, including β and η rules for applications and the let-bindings, as well as a rule for association, or hoisting, of let bindings. Note, however, that we do not include any η rules for the control operators, restricting ourselves to the appropriate β-reductions, which leads to a minimal extension of λ_c with delimited control. It can be shown that the resulting calculus is confluent. While most presentations treat $\mathcal{S}$ as an operator with a binder (for a continuation variable) rather than as a combinator, the latter approach is hardly non-standard: in particular, most implementations provide shift as a combinator.

We now turn to the CPS transformation for $\lambda_{c\mathcal{S}}$, which is presented in Figure 2. Since the CCS calculus is rather complex, the transformation targets syntactic lambda-terms, and we establish the fact that it only produces terms in CCS a posteriori. This translation extends Sabry and Wadler's CPS translation for λ_c [20], which eliminates unnecessary administrative redexes, to handle the shift and reset delimited control operators. Note that, in contrast to some of the classic one-pass CPS translations for shift and reset, including Danvy and Filinski's [3], the translation does not reduce matching shift-reset pairs at transformation time. Note that without this more conservative approach to source-language redexes we could not hope for establishing the desired reflection.

4 Back to Direct Style

Having defined the CPS translation for the extended computational calculus, we now turn to precisely identifying its image. To this end, we introduce a new calculus, $\lambda^*_{c\mathcal{S}}$, presented in Figure 3. The syntax is given as a mildly context-sensitive grammar in the style of *literal movement grammars* [10]. In this case, context-sensitivity amounts to annotating both term

$$
\begin{array}{llll}
\text{roots} & R & ::= & \lambda\, k\, .\, M_k \\
\text{terms}_\Delta & M, N & ::= & K_\Delta\, V \,|\, V\, W\, K_\Delta \,|\, K_\Delta\, M_\bullet \\
\text{values} & V, W & ::= & x \,|\, \lambda\, x\, .\, R \,|\, S \\
\text{shift} & S & ::= & \lambda\, w\, j\, .\, w\, (\lambda\, y\, k\, .\, k\, (j\, y))\, (\lambda\, x\, .\, x) \\
\text{continuations}_\Delta & J, K & ::= & {}_{(\Delta=k)}\, k \,|\, {}_{(\Delta=\bullet)}\, \lambda\, x\, .\, x \,|\, \lambda\, x\, .\, M_\Delta
\end{array}
$$

$$
\begin{array}{llll}
(\beta.v) & (\lambda\, x\, k\, .\, M_k)\, V\, K_\Delta & \to & M_k[x := V][k := K_\Delta] \\
(\eta.v) & \lambda\, x\, k\, .\, V\, x\, k & \to & V \\
(\beta.let) & (\lambda\, x\, .\, M_\Delta)\, V & \to & M_\Delta[x := V] \\
(\eta.let) & \lambda\, x\, .\, K_\Delta\, x & \to & K_\Delta \\
(\beta.\mathcal{S}) & S\, W\, J_\bullet & \to & W\, (\lambda\, y\, k\, .\, k\, (J_\bullet\, y))\, (\lambda\, x\, .\, x) \\
(\beta.\mathcal{R}) & (\lambda\, x\, .\, x) V & \to & V
\end{array}
$$

Figure 3 Continuation-composing style calculus $\lambda^*_{c\mathcal{S}}$.

and continuation nonterminals with Δ, which ranges over the set of variables extended with $\bullet$, and limiting certain productions to particular annotations. This serves to distinguish the "tail-recursive" parts of the term, where there is a current continuation that needs to be used, from the "returning" calls, where there is no access to the current continuation (and thus the only trivial continuation is the identity). Throughout the following, we use $\Delta \vdash^N_C M$ to mean that a term M is derived as a member of syntactic class N of calculus C under assumptions Δ of the shape appropriate for the given calculus and non-terminal combination; in the case of standard context-free grammars, this assumption context is always empty.

For the semantics of our calculus, we follow the methodology of Sabry and Wadler [20], extending their λ_{cps} calculus with reductions that notionally match our control operators. Note that while this calculus can be considered a subsystem of the lambda-calculus, its reductions take much larger steps, and thus the system is not closed under general λ_v reductions. We begin by establishing that the image of the CPS translation defined in previous section is indeed contained within $\lambda^*_{c\mathcal{S}}$.

▶ **Lemma 7** (Characterisation of CCS). *For all $M \in \lambda_{c\mathcal{S}}$, $M^* \in \lambda^*_{c\mathcal{S}}$.*

Proof. We prove the following propositions by mutual structural induction on the term:

- $\vdash^M_{\lambda_{c\mathcal{S}}} M \implies \vdash^R_{\lambda^*_{c\mathcal{S}}} M^*$,
- $\vdash^M_{\lambda_{c\mathcal{S}}} M \wedge \Delta \vdash^K_{\lambda^*_{c\mathcal{S}}} K \implies \Delta \vdash^M_{\lambda^*_{c\mathcal{S}}} (M : K)$,
- $\vdash^V_{\lambda_{c\mathcal{S}}} V \implies \Delta \vdash^V_{\lambda^*_{c\mathcal{S}}} V^\dagger$. ◀

4.1 Direct-Style Transformation

With the calculus $\lambda^*_{c\mathcal{S}}$ defined, we now turn to a translation to direct style. The target of such translation is $\lambda_{c\mathcal{S}}$, the computational calculus with shift and reset, and the translation is defined in Figure 4, with the definition proceeding inductively on the structure of terms of $\lambda^*_{c\mathcal{S}}$. We can now show that the CPS and DS translations form a retraction pair with respect to syntactic equality (as usual, up to implicit α-equivalence) in $\lambda^*_{c\mathcal{S}}$. We take I_Δ to denote the trivial continuation for Δ, i.e., $I_k = k$ and $I_\bullet = \lambda\, x\, .\, x$.

▶ **Theorem 8** (Right inverse of $*$). *For all $R \in \lambda^*_{c\mathcal{S}}$, $R^{\#*} \equiv R$. Also, the following equalities hold: $M^\sharp_\Delta : I_\Delta \equiv M_\Delta$, $V^{\natural\dagger} \equiv V$, $K^\flat_\Delta[M] : I_\Delta \equiv M : K_\Delta$.*

Proof. By mutual structural induction. ◀

$$
\begin{array}{ll}
\# : \lambda^{*}_{c\mathcal{S}} \to \lambda_{c\mathcal{S}} & \\
(\lambda\, k\, .\, M_k)^{\#} & = M_k^{\sharp} \\
(K_\Delta\, V)^{\sharp} & = K_\Delta^{\flat}[V^{\natural}] \\
(V\, W\, K_\Delta)^{\sharp} & = K_\Delta^{\flat}[V^{\natural}\, W^{\natural}] \\
(K_\Delta\, M_\bullet)^{\sharp} & = K_\Delta^{\flat}[\langle M_\bullet^{\sharp}\rangle] \\
x^{\natural} & = x \\
(\lambda\, x\, .\, R)^{\natural} & = \lambda\, x\, .\, R^{\#} \\
(\lambda\, w\, j\, .\, w\, (\lambda\, y\, k\, .\, k\, (j\, y))\, (\lambda\, x\, .\, x))^{\natural} & = \mathcal{S} \\
k^{\flat} & = [\,] \\
(\lambda\, x\, .\, x)^{\flat} & = [\,] \\
(\lambda\, x\, .\, N_\Delta)^{\flat} & = \mathrm{let}\, x = [\,]\, \mathrm{in}\, N_\Delta^{\sharp}
\end{array}
$$

Figure 4 Back to Direct Style from $\lambda^{*}_{c\mathcal{S}}$.

$$
\begin{array}{llll}
\text{terms} & M, N & ::= & K[V] \mid K[P] \\
\text{values} & V, W & ::= & x \mid \lambda\, x\, .\, M \mid \mathcal{S} \\
\text{nonvalues} & P, Q & ::= & V\, W \mid \langle M \rangle \\
\text{pure contexts} & J, K & ::= & [\,] \mid \mathrm{let}\, x = [\,]\, \mathrm{in}\, M
\end{array}
$$

$$
\begin{array}{llcll}
(\beta.v) & K[(\lambda\, x\, .\, M)\, V] & \to & M[x := V] : K & K \text{ maximal} \\
(\eta.v) & \lambda\, x\, .\, V\, x & \to & V & \\
(\beta.let) & \mathrm{let}\, x = V\, \mathrm{in}\, M & \to & M[x := V] & \\
(\eta.let) & \mathrm{let}\, x = [\,]\, \mathrm{in}\, K[x] & \to & K & \\
(\beta.\mathcal{S}) & \langle J[\mathcal{S}\, W] \rangle & \to & \langle W\, (\lambda\, y\, .\langle J[y] \rangle)\rangle & \\
(\beta.\mathcal{R}) & \langle V \rangle & \to & V &
\end{array}
$$

$$
\begin{array}{ll}
V : K & = K[V] \\
P : K & = K[P] \\
(\mathrm{let}\, x = V\, \mathrm{in}\, M) : K & = \mathrm{let}\, x = V\, \mathrm{in} (M : K) \\
(\mathrm{let}\, x = P\, \mathrm{in}\, M) : K & = \mathrm{let}\, x = P\, \mathrm{in} (M : K)
\end{array}
$$

Figure 5 The kernel direct style calculus $\lambda^{\triangleright}_{c\mathcal{S}}$.

This result establishes that $\lambda^{*}_{c\mathcal{S}}$ does not overestimate the set of valid CCS terms, as any root in $\lambda^{*}_{c\mathcal{S}}$ can be obtained by the CPS transformation from its own translation to $\lambda_{c\mathcal{S}}$. However, not all terms of $\lambda_{c\mathcal{S}}$ can be obtained as the result of the direct style translation. Thus, we define yet another calculus, $\lambda^{\triangleright}_{c\mathcal{S}}$, which characterises the *kernel* of $\lambda_{c\mathcal{S}}$. The definition presented in Figure 5 again follows and extends Sabry and Wadler's take on a refined calculus (this time extending their λ_{c**}); the major difference with respect to $\lambda_{c\mathcal{S}}$ is the fact that all the let-bindings are hoisted, i.e., normalised with respect to associativity rule. The reduction rules need to preserve this fact, which again leads to larger reduction steps: this time, when reducing an application, we may need to reassociate arbitrarily many let-bindings. We finish this section by establishing that the image of our direct style translation falls within $\lambda^{\triangleright}_{c\mathcal{S}}$.

▶ **Lemma 9** (Characterisation of kernel DS). *For all $M \in \lambda^{*}_{c\mathcal{S}}$, $M^{\#} \in \lambda^{\triangleright}_{c\mathcal{S}}$.*

Proof. We prove the following propositions by mutual structural induction on the term:

- $\vdash^{R}_{\lambda^{*}_{c\mathcal{S}}} R \implies \vdash^{M}_{\lambda^{\triangleright}_{c\mathcal{S}}} R^{\#}$,
- $\Delta \vdash^{M}_{\lambda^{*}_{c\mathcal{S}}} M \implies \vdash^{M}_{\lambda^{\triangleright}_{c\mathcal{S}}} M^{\sharp}$,

- $\vdash^{V}_{\lambda^*_{c\mathcal{S}}} V \implies \vdash^{V}_{\lambda^{\triangleright}_{c\mathcal{S}}} V^{\natural}$,
- $\Delta \vdash^{K}_{\lambda^*_{c\mathcal{S}}} K \implies \vdash^{K}_{\lambda^{\triangleright}_{c\mathcal{S}}} K^{\flat}$. ◀

5 Reflection: Computational λ-Calculus with Shift and Reset

Having introduced the three main calculi involved in the reflection and established a syntactic inverse in one direction (in Theorem 8), we now turn to establishing a Galois connection between $\lambda_{c\mathcal{S}}$ and $\lambda^*_{c\mathcal{S}}$. By Theorem 6, such a connection will decompose into an isomorphism and a reflection: in the following we establish that $\lambda^{\triangleright}_{c\mathcal{S}}$ is such a factorisation.

5.1 Monotonicity

In order for our CPS and DS transformations to form a Galois connection, we must first establish that they are monotone maps, i.e., that they preserve the *order* given by reflexive-transitive closure of the reduction relation of, respectively, $\lambda_{c\mathcal{S}}$ and $\lambda^*_{c\mathcal{S}}$. Since for some intermediate results we require zero or one reduction steps, we also introduce $\rightarrow^?$ as a reflexive closure of the relation $\rightarrow$. We begin by establishing that any pure evaluation context J of $\lambda_{c\mathcal{S}}$ can be matched by a continuation of $\lambda^*_{c\mathcal{S}}$.

▶ **Lemma 10** (Existence of a continuation for each context). *For all J, Δ and K_Δ, exists $\hat{J}_\Delta$ such that for all M, $J[M] : K_\Delta \equiv M : \hat{J}_\Delta$.*

Proof. By structural induction on J. ◀

Next, we show that any reduction of $\lambda^*_{c\mathcal{S}}$ continuations extends to the colon translations of a common $\lambda_{c\mathcal{S}}$ term.

▶ **Lemma 11** (Single-step reduction preservation by : in the second argument). *For any $\lambda_{c\mathcal{S}}$ term M and $\lambda^*_{c\mathcal{S}}$ continuations J_Δ and K_Δ such that $J_\Delta \rightarrow K_\Delta$ we have $M : J_\Delta \rightarrow^? M : K_\Delta$.*

Proof. By structural induction on M. ◀

Finally, we can show that the CPS translation preserves single-step reductions, possibly without making a transition in $\lambda^*_{c\mathcal{S}}$. Monotonicity of CPS follows as a simple corollary.

▶ **Lemma 12** (Single-step reduction preservation by :, $*$ and †). *The following implications hold:*

- $M \rightarrow N \implies \forall K \,.\, M : K_\Delta \rightarrow^? N : K_\Delta$,
- $M \rightarrow N \implies M^* \rightarrow^? N^*$,
- $V \rightarrow W \implies V^\dagger \rightarrow^? W^\dagger$

Proof. We prove the statements by mutual induction on the structure of the term, and invert the reduction relation as necessary. Preservation in the second argument is used for some congruences and $(\eta.let)$. Base cases $(let.1), (let.2), (assoc)$ follow by definition. The existence of a continuation is used for $(\beta.\mathcal{S})$.

We show the case for the $(\beta.\mathcal{S})$ reduction as an interesting example. We have the following reduction:

$$\langle J[\mathcal{S}\; W]\rangle \rightarrow \langle W\,(\lambda\, y\,.\langle J[y]\rangle)\rangle,$$

and need to prove that $\langle J[\mathcal{S}\; W]\rangle : K_\Delta \rightarrow^? \langle W\,(\lambda\, y\,.\langle J[y]\rangle)\rangle : K_\Delta$.

We proceed as follows:

$$\begin{aligned}
& & \langle J[\mathcal{S}\ W]\rangle : K_\Delta \\
\equiv_{\text{def.}} & & K_\Delta\,(J[\mathcal{S}\ W] : (\lambda\,x\,.\,x)) \\
\equiv_{\text{existence of cont.}} & & K_\Delta\,(\mathcal{S}\ W : J_\bullet) \\
\equiv_{\text{def.}} & & K_\Delta\,(\mathcal{S}\,W^\dagger\,J_\bullet) \\
\rightarrow_{(\beta.\mathcal{S})} & & K_\Delta\,(W^\dagger\,(\lambda\,y\,k\,.\,k\,(J_\bullet\,y))\,(\lambda\,x\,.\,x)) \\
\equiv_{\text{def.}} & & K_\Delta\,(W^\dagger\,(\lambda\,y\,k\,.\,k\,(y : J_\bullet))\,(\lambda\,x\,.\,x)) \\
\equiv_{\text{existence of cont.}} & & K_\Delta\,(W^\dagger\,(\lambda\,y\,k\,.\,k\,(J[y] : (\lambda\,x\,.\,x)))\,(\lambda\,x\,.\,x)) \\
\equiv_{\text{def.}} & & K_\Delta\,((W\,(\lambda\,y\,.\langle J[y]\rangle)) : (\lambda\,x\,.\,x)) \\
\equiv_{\text{def.}} & & \langle W\,(\lambda\,y\,.\langle J[y]\rangle)\rangle : K_\Delta
\end{aligned}$$

As another examples, consider the sample congruence case, where we have:

$$\frac{V \rightarrow W}{P\,V \rightarrow P\,W},$$

and need to show that $P\,V : K_\Delta \rightarrow^? P\,W : K_\Delta$.

We proceed as follows:

$$\begin{aligned}
& & V \rightarrow W \\
\Longrightarrow\ \text{ind. hyp.} & & V^\dagger \rightarrow W^\dagger \\
\Longrightarrow\ \text{congruence} & & \lambda\,x\,.\,x\,V^\dagger\,K_\Delta \rightarrow \lambda\,x\,.\,x\,W^\dagger\,K_\Delta \\
\Longrightarrow\ \text{second arg. preservation} & & P : (\lambda\,x\,.\,x\,V^\dagger\,K_\Delta) \rightarrow P : (\lambda\,x\,.\,x\,W^\dagger\,K_\Delta) \\
\equiv_{\text{def.}} & & (P\,V) : K_\Delta \rightarrow (P\,W) : K_\Delta
\end{aligned}$$

◀

▶ **Corollary 13** (Monotonicity of $*$). *For all $M_0, M_1 \in \lambda_{c\mathcal{S}}$, $M_0 \twoheadrightarrow M_1$ implies $M_0^* \twoheadrightarrow M_1^*$.*

Monotonicity of the direct-style transformation is simpler to prove: we show that all component parts preserve single-step reductions in $\lambda^{\triangleright}_{c\mathcal{S}}$ (which themselves are possibly multi-step reduction sequences in $\lambda_{c\mathcal{S}}$), and obtain monotonicity as a simple corollary.

▶ **Lemma 14** (Single-step reduction preservation by $\#$, $\sharp$, $\natural$ and $\flat$). *The following implications hold:*

- $R_0 \rightarrow R_1' \implies R_0^\# \rightarrow R_1^\#$,
- $M_k \rightarrow N_k \implies M_k^\sharp \rightarrow N_k^\sharp$,
- $M_\bullet \rightarrow N_\bullet \implies \langle M_\bullet^\sharp\rangle \rightarrow \langle N_\bullet^\sharp\rangle$,
- $V \rightarrow W \implies V^\natural \rightarrow W^\natural$,
- $J_\Delta \rightarrow K_\Delta \implies \forall M\,.\,J_\Delta^\flat[M] \rightarrow K_\Delta^\flat[M]$.

Proof. Mutual structural induction on the first term and then each case by inversion on single-step reduction. ◀

▶ **Corollary 15** (Monotonicity of $\#$). *For all $R_0, R_1 \in \lambda^*_{c\mathcal{S}}$, $R_0 \twoheadrightarrow R_1$ implies $R_0^\# \twoheadrightarrow R_1^\#$.*

5.2 Reflection theorem

Recall from Section 2 that in order to establish that $*$ and $\#$ form a Galois connection, it is enough to show that both compositions are extensive with respect to the appropriate reduction orderings, i.e., $M \twoheadrightarrow M^{*\#}$ and $R \twoheadrightarrow R^{\#*}$, respectively. Since in Theorem 8 we have shown that $R \equiv R^{\#*}$, the latter ordering holds trivially. In this section we establish the remaining property.

▶ **Lemma 16** (Generalised associativity). *The following reduction holds:*

- $K^{\flat}_{\Delta}[\mathrm{let}\, x = L \,\mathrm{in}\, N] \rightarrow^{?} \mathrm{let}\, x = L \,\mathrm{in}\, K^{\flat}_{\Delta}[N]$.

Proof. By cases on K_{Δ}. ◀

▶ **Lemma 17** (Left near inverse of : and †). *The following reductions hold:*

- $K^{\flat}_{\Delta}[M] \twoheadrightarrow (M : K_{\Delta})^{\sharp}$,
- $V \twoheadrightarrow V^{\dagger\natural}$.

Proof. By mutual structural induction. The generalised associativity is used in several cases, it conveniently wraps potential uses of (*let.assoc*) rule, as presented in the following example cases for let-binders and `reset`.

$$\begin{aligned}
& & K^{\flat}_{\Delta}[\mathrm{let}\, x = L \,\mathrm{in}\, N] \\
&\rightarrow^{?}_{\text{gen. assoc.}} & \mathrm{let}\, x = L \,\mathrm{in}\, K^{\flat}_{\Delta}[N] \\
&\twoheadrightarrow_{\text{ind. hyp.}} & \mathrm{let}\, x = L \,\mathrm{in} (N : K_{\Delta})^{\sharp} \\
&\equiv_{\text{def.}} & (\lambda\, x\, .(N : K_{\Delta}))^{\flat}[L] \\
&\twoheadrightarrow_{\text{ind. hyp.}} & ((\mathrm{let}\, x = L \,\mathrm{in}\, N) : K_{\Delta})^{\sharp}
\end{aligned}$$

$$\begin{aligned}
& & K^{\flat}_{\Delta}[\langle M \rangle] \\
&\equiv_{\text{def.}} & K^{\flat}_{\Delta}[\langle I^{\flat}_{\bullet}[M] \rangle] \\
&\twoheadrightarrow_{\text{ind. hyp.}} & K^{\flat}_{\Delta}[\langle (M : I_{\bullet})^{\sharp} \rangle] \\
&\equiv_{\text{def.}} & (K_{\Delta}(M : I_{\bullet}))^{\sharp} \\
&\equiv_{\text{def.}} & (\langle M \rangle : K_{\Delta})^{\sharp}
\end{aligned}$$

◀

▶ **Theorem 18** (Left near inverse of $*$). *For all $M \in \lambda_{c\mathcal{S}}$, $M \twoheadrightarrow M^{*\#}$.*

Proof. Follows from the left near inverse for the : transformation. ◀

▶ **Corollary 19** (Reflection). *Transformations $*$ and $\#$ form a reflection.*

5.3 Reflection decomposition

By Theorem 6, any reflection decomposes into an order isomorphism and an inclusion. In our case, this means that the reflection $(*, \#)$ has a kernel that is isomorphic to $\lambda^{*}_{c\mathcal{S}}$. This is our calculus $\lambda^{\triangleright}_{c\mathcal{S}}$ – although we still need to establish that it is in fact isomorphic to $\lambda^{*}_{c\mathcal{S}}$. To this end, we first calculate the CPS translation as specialised to $\lambda^{\triangleright}_{c\mathcal{S}}$ (i.e., as a composition of inclusion of $\lambda^{\triangleright}_{c\mathcal{S}}$ in $\lambda_{c\mathcal{S}}$ and $*$), which we dub $\star$; this transformation is presented in Figure 6. We can then establish that $\star$ and $\#$ compose to identity, which, together with one-to-one matching of reductions established in Lemma 14 establishes the isomorphism.

$$
\begin{array}{ll}
\star : \lambda^{\triangleright}_{c\mathcal{S}} \to \lambda^{*}_{c\mathcal{S}} & \\
M^{\star} & = \lambda\, k\, .\, M^{\circ}_{k} \\
(K[V])^{\circ}_{\Delta} & = K^{\ddagger}_{\Delta}\, V^{\dagger} \\
(K[V\, W])^{\circ}_{\Delta} & = V^{\dagger}\, W^{\dagger}\, K^{\ddagger}_{\Delta} \\
(K[\langle M\rangle])^{\circ}_{\Delta} & = K^{\ddagger}_{\Delta}\, M^{\circ}_{\bullet} \\
x^{\dagger} & = x \\
(\lambda\, x\, .\, M)^{\dagger} & = \lambda\, x\, .\, M^{\star} \\
\mathcal{S}^{\dagger} & = \lambda\, w\, j\, .\, w\, (\lambda\, y\, k\, .\, k\, (j\, y))\, (\lambda\, x\, .\, x) \\
[\,]^{\ddagger}_{k} & = k \\
[\,]^{\ddagger}_{\bullet} & = \lambda\, x\, .\, x \\
(\mathrm{let}\, x = [\,]\, \mathrm{in}\, N)^{\ddagger}_{\Delta} & = \lambda\, x\, .\, N^{\circ}_{\Delta}
\end{array}
$$

Figure 6 Order isomorphism from $\lambda^{\triangleright}_{c\mathcal{S}}$ to $\lambda^{*}_{c\mathcal{S}}$.

$$
\begin{array}{ll}
\triangleright : \lambda_{c\mathcal{S}} \to \lambda^{\triangleright}_{c\mathcal{S}} & \\
M^{\triangleright} & = M : [\,] \\
V : K & = K[V^{\dagger}] \\
(P\, Q) : K & = P : (\mathrm{let}\, x = [\,]\, \mathrm{in} (Q : (\mathrm{let}\, y = [\,]\, \mathrm{in}\, K[x\, y]))) \\
(P\, W) : K & = P : (\mathrm{let}\, x = [\,]\, \mathrm{in}\, K[x\, W^{\dagger}]) \\
(V\, Q) : K & = Q : (\mathrm{let}\, y = [\,]\, \mathrm{in}\, K[V^{\dagger}\, y]) \\
(V\, W) : K & = K[V^{\dagger}\, W^{\dagger}] \\
(\mathrm{let}\, x = M\, \mathrm{in}\, N) : K & = M : (\mathrm{let}\, x = [\,]\, \mathrm{in} (N : K)) \\
\langle M\rangle : K & = K[\langle M^{\triangleright}\rangle] \\
x^{\dagger} & = x \\
(\lambda\, x\, .\, M)^{\dagger} & = \lambda\, x\, .\, M^{\triangleright} \\
\mathcal{S}^{\dagger} & = \mathcal{S}
\end{array}
$$

Figure 7 Inclusion in $\lambda_{c\mathcal{S}}$ of $\lambda^{\triangleright}_{c\mathcal{S}}$.

▶ **Lemma 20** (Left inverse of $\star$). *For all $M \in \lambda^{\triangleright}_{c\mathcal{S}}$ we have $M \equiv M^{\star\#}$.*

Proof. By mutual induction on the structure of terms, including analogous statements for the auxiliary transformations. ◀

Having established that $\lambda^{\triangleright}_{c\mathcal{S}}$ is isomorphic to $\lambda^{*}_{c\mathcal{S}}$, we obtain an inclusion between $\lambda^{\triangleright}_{c\mathcal{S}}$ and the main calculus, $\lambda_{c\mathcal{S}}$. Thus, to conclude this section we present the one-pass transformation $\triangleright : \lambda_{c\mathcal{S}} \to \lambda^{\triangleright}_{c\mathcal{S}}$, which is computed from the composition of $*$ and $\#$. This transformation, presented in Figure 7, forms a final reflection for these calculi, together with identity: $(\triangleright, \mathrm{id}_{\lambda^{\triangleright}_{c\mathcal{S}}})$ is a reflection between $\lambda_{c\mathcal{S}}$ and $\lambda^{\triangleright}_{c\mathcal{S}}$.

6 Isomorphism: λ_v-Calculus with Shift and Reset

While the results we have obtained thus far provide some fundamental insight into the structure and reductions of computations in continuation composing style, the computational calculus $\lambda_{c\mathcal{S}}$ we took as our source language differs somewhat from calculi with shift and reset that are most commonly studied. Thus, in this section we study $\lambda_{\mathcal{S}}$, the call-by-value lambda calculus extended with delimited control operators, and apply the results we have obtained thus far to this restricted setting.

$$
\begin{array}{llll}
\text{terms} & M, N & ::= & V \mid P \\
\text{values} & V, W & ::= & x \mid \lambda\, x \,.\, M \mid \mathcal{S} \\
\text{nonvalues} & P, Q & ::= & M\; N \mid \langle M \rangle \\
\text{pure contexts} & J, K & ::= & [\,] \mid K\; M \mid V\; K
\end{array}
$$

$$
\begin{array}{llll}
(\beta.v) & (\lambda\, x \,.\, M)\; V & \to & M[x := V] \\
(\eta.v) & \lambda\, x \,.\, V\; x & \to & V \\
(\beta.\mathcal{S}) & \langle J[\mathcal{S}\; N] \rangle & \to & \langle N\,(\lambda\, y \,.\langle J[y] \rangle) \rangle \\
(\beta.\mathcal{R}) & \langle V \rangle & \to & V
\end{array}
$$

Figure 8 Subcalculus $\lambda_{\mathcal{S}}$: $\lambda_{c\mathcal{S}}$ without let.

$$
\begin{array}{llll}
\text{roots} & R & ::= & V_\varepsilon \mid M_\varepsilon \\
\text{terms}_\Sigma & M, N & ::= & {\scriptstyle(\Sigma=\Sigma_1\Sigma_2)}\; K_{\Sigma_1}[P_{\Sigma_2}] \\
\text{values}_\Sigma & V, W & ::= & {\scriptstyle(\Sigma=x)}x \mid {\scriptstyle(\Sigma=\varepsilon)}\; x \mid {\scriptstyle(\Sigma=\varepsilon)}\; \lambda\, x \,.\, R \mid {\scriptstyle(\Sigma=\varepsilon)}\; \mathcal{S} \\
\text{nonvalues}_\Sigma & P, Q & ::= & {\scriptstyle(\Sigma=\Sigma_1\Sigma_2)}\; V_{\Sigma_1}\; W_{\Sigma_2} \mid {\scriptstyle(\Sigma=\varepsilon)}\; \langle R \rangle \\
\text{pure contexts}_\Sigma & J, K & ::= & {\scriptstyle(\Sigma=\varepsilon)}\; [\,] \mid \text{let}\, x = [\,]\, \text{in}\, M_{\Sigma x}
\end{array}
$$

$$
\begin{array}{llll}
(\beta.v) & K_\Sigma[(\lambda\, x \,.\, R)\; V_\varepsilon] & \to & R[x := V_\varepsilon] : K_\Sigma \quad K_\Sigma \text{ maximal} \\
(\eta.v) & \lambda\, x \,.\, V_\varepsilon\; x & \to & V_\varepsilon \\
(\beta.\mathcal{S}) & \langle J_\varepsilon[\mathcal{S}\; W_\varepsilon] \rangle & \to & \langle W_\varepsilon\,(\lambda\, y \,.\langle J_\varepsilon; y \rangle) \rangle \\
(\beta.\mathcal{R}) & K_\Sigma[\langle V_\varepsilon \rangle] & \to & K_\Sigma; V_\varepsilon \quad K_\Sigma \text{ maximal}
\end{array}
$$

$$
\begin{array}{ll}
[\,]; V_\varepsilon & = V_\varepsilon \\
(\text{let}\, x = [\,]\, \text{in}\, M_{\Sigma x}); V_\varepsilon & = M_{\Sigma x}[x := V_\varepsilon] \\
V_\varepsilon : K_\Sigma & = K_\Sigma; V_\varepsilon \\
P_{\Sigma_2} : K_{\Sigma_1} & = K_{\Sigma_1}[P_{\Sigma_2}] \\
(\text{let}\, x = P_{\Sigma_2}\, \text{in}\, M_{\Sigma x}) : K_{\Sigma_1} & = \text{let}\, x = P_{\Sigma_2}\, \text{in}(M_{\Sigma x} : K_{\Sigma_1})
\end{array}
$$

Figure 9 Subcalculus $\lambda^{\triangleright}_{\mathcal{S}}$: the image of $\lambda_{\mathcal{S}}$ via $\triangleright$.

The syntax and semantics of $\lambda_{\mathcal{S}}$ are presented in Figure 8. It is clear that this calculus embeds in $\lambda_{c\mathcal{S}}$. The reduction relation is defined via contraction relation and is a strict subrelation of the induced reduction relation (e.g., $(\lambda\, x \,.\, x)M \twoheadrightarrow M$ in $\lambda_{c\mathcal{S}}$, but not in $\lambda_{\mathcal{S}}$). The reduction relations for the CCS and kernel calculi, presented later on in this section, are images of this restricted reduction relation. It is worth noting that it would also be sound to use the induced relations instead: all transformations, including all-new $\triangleleft$, are designed to be monotone with respect to the original, wider reduction relations. However, we do not pursue the task of characterising the induced relations in this work, and therefore stick to the restricted relations.

We begin by considering the image of $\lambda_{\mathcal{S}}$ under the reflection defined in previous section: this is the calculus $\lambda^{\triangleright}_{\mathcal{S}}$, defined in Figure 9. We establish that it contains the image of $\lambda_{\mathcal{S}}$ under $\triangleright$, while deferring the other inclusion till later.

▶ **Lemma 21** (Restriction of $\triangleright$). *For all $M \in \lambda_{\mathcal{S}}$, $M^{\triangleright} \in \lambda^{\triangleright}_{\mathcal{S}}$.*

With $\lambda^{\triangleright}_{c\mathcal{S}}$ defined, we need to define an inverse transformation, $\triangleleft$. However, the reduction relation on $\lambda_{c\mathcal{S}}$ – induced by its super-calculi – is cumbersome to work with. Thus, we define this transformation for the entire calculus $\lambda_{c\mathcal{S}}$, and establish its properties on its particular sub-calculi post hoc. First, notice that the shape of $\lambda^{\triangleright}_{\mathcal{S}}$ is more constrained than

$$\begin{array}{ll}
\triangleleft : \lambda_{c\mathcal{S}} \to \lambda_{c\mathcal{S}} & \\
(M\,N)^{\triangleleft} & = M^{\triangleleft}\,N^{\triangleleft} \\
(\mathrm{let}\,x = M\,\mathrm{in}\,N)^{\triangleleft} & = N^{\triangleleft}[x := M^{\triangleleft}] \quad (\text{if } N \equiv K[x] \text{ for some } K \text{ and } x \notin \mathrm{FV}(K)) \\
(\mathrm{let}\,x = M\,\mathrm{in}\,N)^{\triangleleft} & = \mathrm{let}\,x = M^{\triangleleft}\,\mathrm{in}\,N^{\triangleleft} \quad (\text{otherwise}) \\
\langle M\rangle^{\triangleleft} & = \langle M^{\triangleleft}\rangle \\
x^{\triangleleft} & = x \\
(\lambda\,x\,.\,M)^{\triangleleft} & = \lambda\,x\,.\,M^{\triangleleft} \\
\mathcal{S}^{\triangleleft} & = \mathcal{S}
\end{array}$$

Figure 10 Deletion of inessential let-expressions.

the general calculus $\lambda_{\mathcal{S}}$, which is mostly due to lifting some subcomputations as pure, linear let-expressions. Thus, the idea behind $\triangleleft$ is to *inline* these (and only these) let-expressions. The transformation is presented in Figure 10.

Additionally, we lift $\triangleleft$ to pure contexts of $\lambda_{c\mathcal{S}}$ as an auxiliary construct in the following. It is easy to check that this makes $\triangleleft$ distribute over plugging of terms into contexts:

$$K^{\triangleleft}[M^{\triangleleft}] \equiv (K[M])^{\triangleleft}.$$

We can now show that the transformation is monotone with respect to the $\lambda_{c\mathcal{S}}$ reductions.

▶ **Theorem 22** (Monotonicity of $\triangleleft$). *For all $M, N \in \lambda_{c\mathcal{S}}$, $M \twoheadrightarrow N$ implies $N^{\triangleleft} \twoheadrightarrow N^{\triangleleft}$.*

Proof. Induction on reflexive-transitive closure of contraction relation. To prove single-step version, apply structural induction on the left-hand-side term and then inversion on contraction relation. The only interesting case is the contraction of `shift`: given $\langle J[\mathcal{S}\ M]\rangle \to \langle M(\lambda\,y\,.\langle J[y]\rangle)\rangle$, show $\langle J[\mathcal{S}\ M]\rangle^{\triangleleft} \twoheadrightarrow \langle M(\lambda\,y\,.\langle J[y]\rangle)\rangle^{\triangleleft}$. Notice that we have $\langle J[\mathcal{S}\ M]\rangle^{\triangleleft} \equiv \langle J^{\triangleleft}[\mathcal{S}\ M^{\triangleleft}]\rangle \to \langle M^{\triangleleft}\,(\lambda\,y\,.\langle J^{\triangleleft}[y]\rangle)\rangle \equiv \langle M\,(\lambda\,y\,.\langle J[y]\rangle)\rangle^{\triangleleft}$. Crucially, transformation preserves purity of contexts. ◀

Now we can ensure that inlining the administrative let-expressions in $\lambda_{\mathcal{S}}^{\triangleright}$ produces terms of $\lambda_{\mathcal{S}}$, and thus that $\triangleright$ restricted to $\lambda_{\mathcal{S}}$ is a section.

▶ **Lemma 23** (Restriction of $\triangleleft$). *For all $R \in \lambda_{\mathcal{S}}^{\triangleright}$, $R^{\triangleleft} \in \lambda_{\mathcal{S}}$.*

▶ **Theorem 24** (Left inverse of $\triangleright$). *For all $M \in \lambda_{\mathcal{S}}$, $M \equiv M^{\triangleright\triangleleft}$. Also, the following identities hold: $V^{\dagger\triangleleft} \equiv V$, $(M : K)^{\triangleleft} \equiv K^{\triangleleft}[M]$.*

Proof. Mutual structural induction on M, V and M, respectively. ◀

To show that $\triangleright$ is also a retraction, we need a tool that can apply a substitution of variables for $\lambda_{\mathcal{S}}$ terms to a $\lambda_{\mathcal{S}}^{\triangleright}$ term.

▶ **Definition 25** (Iterated flattened let-expressions). *Let P, Q refer to the grammar of $\lambda_{\mathcal{S}}$, all other names refer to the grammar of $\lambda_{\mathcal{S}}^{\triangleright}$.*

$$\begin{array}{ll}
\varepsilon \triangleright R & = R \\
[x := P] \triangleright x_x & = P^{\triangleright} \\
[\overline{x := P}, y := Q] \triangleright M_{\overline{x}y} & = [\overline{x := P}] \triangleright (Q : (\mathrm{let}\,y = [\,]\,\mathrm{in}\,M_{\overline{x}y}))
\end{array}$$

We can now establish that $\triangleright$ is a retraction and, as a consequence, that $\lambda_{\mathcal{S}}$ and $\lambda_{\mathcal{S}}^{\triangleright}$ are isomorphic.

$$\begin{array}{llcl}
\text{roots} & R & ::= & \lambda\, k\,.\, T_k \\
\text{trunks}_\Delta & T & ::= & I_\Delta\, V_\varepsilon \mid M_{\Delta,\varepsilon} \\
\text{terms}_{\Delta,\Sigma} & M, N & ::= & {}_{(\Sigma=\Sigma_1\Sigma_2\Sigma_3)}\, V_{\Sigma_2}\, W_{\Sigma_3}\, K_{\Delta,\Sigma_1} \mid K_{\Delta,\Sigma}\, T_\varepsilon \\
\text{values}_\Sigma & V, W & ::= & {}_{(\Sigma=x)} x \mid {}_{(\Sigma=\varepsilon)}\, x \mid {}_{(\Sigma=\varepsilon)}\, \lambda\, x\,.\, R \mid {}_{(\Sigma=\varepsilon)}\, S \\
\text{shift} & S & ::= & \lambda\, w\, j\,.\, w\,(\lambda\, y\, k\,.\, k\,(j\, y))\,(\lambda\, x\,.\, x) \\
\text{trivial continuations}_\Delta & I & ::= & {}_{(\Delta=k)}\, k \mid {}_{(\Delta=\bullet)}\, \lambda\, x\,.\, x \\
\text{continuations}_{\Delta,\Sigma} & J, K & ::= & {}_{(\Sigma=\varepsilon)}\, I_\Delta \mid \lambda\, x\,.\, M_{\Delta,\Sigma x}
\end{array}$$

$$\begin{array}{llcl}
(\beta.v) & (\lambda\, x\, k\,.\, T_k)\, V_\varepsilon\, K_{\Delta,\Sigma} & \to & T_k[x := V_\varepsilon] : K_{\Delta,\Sigma} \\
(\eta.v) & \lambda\, x\, k\,.\, V_\varepsilon\, x\, k & \to & V_\varepsilon \\
(\beta.\mathcal{S}) & S\, W_\varepsilon\, J_{\bullet,\varepsilon} & \to & W_\varepsilon\,(\lambda\, y\, k\,.\, k\,(J_{\bullet,\varepsilon}; y))\,(\lambda\, x\,.\, x) \\
(\beta.\mathcal{R}) & K_{\Delta,\Sigma}\,((\lambda\, x\,.\, x)V) & \to & K_{\Delta,\Sigma}; V
\end{array}$$

$$\begin{array}{ll}
I_\Delta; V_\varepsilon & = I_\Delta\, V_\varepsilon \\
(\lambda\, x\,.\, M_{\Delta,\Sigma x}); V_\varepsilon & = M_{\Delta,\Sigma x}[x := V_\varepsilon] \\
(k\, V_\varepsilon) : K_{\Delta,\Sigma} & = K_{\Delta,\Sigma}; V_\varepsilon \\
M_{k,\varepsilon} : K_{\Delta,\Sigma} & = M_{k,\varepsilon}[k := K_{\Delta,\Sigma}]
\end{array}$$

Figure 11 Subcalculus $\lambda^*_\mathcal{S}$: the CPS image of $\lambda_\mathcal{S}$.

▶ **Theorem 26** (Right inverse of ▷). *The following identities hold:*

- $R^{\triangleleft\triangleright} \equiv R$,
- $(V^{\triangleleft}_{\overline{x}}[\overline{x := P}])^{\triangleright} \equiv [\overline{x := P}] \triangleright V_{\overline{x}}$,
- $(M^{\triangleleft}_{\overline{x}}[\overline{x := P}])^{\triangleright} \equiv [\overline{x := P}] \triangleright M_{\overline{x}}$.

Proof. The proof proceeds by mutual structural induction on R, V and M, respectively. ◀

▶ **Corollary 27** (Isomorphism of $\lambda_\mathcal{S}$ and $\lambda^\triangleright_\mathcal{S}$). *Transformations* $\triangleright : \lambda_\mathcal{S} \to \lambda^\triangleright_\mathcal{S}$ *and* $\triangleleft : \lambda^\triangleright_\mathcal{S} \to \lambda_\mathcal{S}$ *form an isomorphism.*

Although we have shown that $\lambda_\mathcal{S}$ is isomorphic to its image under ▷, both these calculi are in direct style. However, as any image of ▷ (and thus of its latter component, #), $\lambda^\triangleright_\mathcal{S}$ is a sub-calculus of $\lambda^\triangleright_{c\mathcal{S}}$. Therefore, we investigate the final calculus: the image of $\lambda_\mathcal{S}$ under the CPS transformation, $\lambda^*_\mathcal{S}$. The syntax of the calculus is presented in Figure 11.

Note that, as an image of a subcalculus of $\lambda_{c\mathcal{S}}$ under the CPS transformation, $\lambda^*_\mathcal{S}$ is clearly a sub-calculus of $\lambda^*_{c\mathcal{S}}$. Therefore, we are able to narrow down an isomorphism of $\lambda^*_{c\mathcal{S}}$ and $\lambda^\triangleright_{c\mathcal{S}}$ to the appropriate subcalculi arising from $\lambda_\mathcal{S}$.

▶ **Lemma 28** (Isomorphism of $\lambda^\triangleright_\mathcal{S}$ and $\lambda^*_\mathcal{S}$). *Transformations* $\star : \lambda^\triangleright_\mathcal{S} \to \lambda^*_\mathcal{S}$ *and* $\# : \lambda^*_\mathcal{S} \to \lambda^\triangleright_\mathcal{S}$ *form an isomorphism.*

Proof. Isomorphism of wider $\lambda^\triangleright_{c\mathcal{S}}$ and $\lambda^*_{c\mathcal{S}}$ calculi can be narrowed. To complete the proof, check that $\#[\lambda^*_\mathcal{S}] \subseteq \lambda^\triangleright_\mathcal{S}$ and $\star[\lambda^\triangleright_\mathcal{S}] \subseteq \lambda^*_\mathcal{S}$. ◀

By now we have established that $\lambda^\triangleright_\mathcal{S}$ is isomorphic to both $\lambda_\mathcal{S}$ and $\lambda^*_\mathcal{S}$. Therefore, by composition of these isomorphisms, we obtain an isomorphism between $\lambda_\mathcal{S}$ and $\lambda^*_\mathcal{S}$. This establishes a formal connection between a calculus for delimited control in the familiar style and its counterpart in the continuation composing style. In order for the connection to be made more explicit, we compute the composition of # and ◁, which forms the single-pass direct-style transformation from $\lambda^*_\mathcal{S}$ to $\lambda_\mathcal{S}$. This transformation is dubbed ⋄ and presented in Figure 12. We conclude with the following isomorphism theorem.

$$\diamond : \lambda_{\mathcal{S}}^{*} \to \lambda_{\mathcal{S}}$$

$$\begin{aligned}
&(\lambda\, k \,.\, k\, V_\varepsilon)^{\diamond} &&= V_\varepsilon^{\natural} \\
&(\lambda\, k \,.\, M_{k,\varepsilon})^{\diamond} &&= M_{k,\varepsilon} \sharp\, \varepsilon \\
&x_x\, y_y\, K_{\Delta,\Sigma} \sharp\, \sigma \cdot V \cdot W &&= K_{\Delta,\Sigma} \flat\, (\sigma, V\, W) \\
&x_x\, W_\varepsilon\, K_{\Delta,\Sigma} \sharp\, \sigma \cdot V &&= K_{\Delta,\Sigma} \flat\, (\sigma, V\, W_\varepsilon^{\natural}) \\
&V_\varepsilon\, y_y\, K_{\Delta,\Sigma} \sharp\, \sigma \cdot W &&= K_{\Delta,\Sigma} \flat\, (\sigma, V_\varepsilon^{\natural}\, W) \\
&V_\varepsilon\, W_\varepsilon\, K_{\Delta,\Sigma} \sharp\, \sigma &&= K_{\Delta,\Sigma} \flat\, (\sigma, V_\varepsilon^{\natural}\, W_\varepsilon^{\natural}) \\
&K_{\Delta,\Sigma}\, ((\lambda\, x \,.\, x)\, V_\varepsilon) \sharp\, \sigma &&= K_{\Delta,\Sigma} \flat\, (\sigma, \langle V_\varepsilon^{\natural} \rangle) \\
&K_{\Delta,\Sigma}\, M_{\bullet,\varepsilon} \sharp\, \sigma &&= K \flat\, (\sigma, \langle M_{\bullet,\varepsilon} \sharp\, \varepsilon \rangle) \\
&x^{\natural} &&= x \\
&(\lambda\, x \,.\, R)^{\natural} &&= \lambda\, x \,.\, R^{\diamond} \\
&(\lambda\, w\, j \,.\, w\, (\lambda\, y\, k \,.\, k\, (j\, y))\, (\lambda\, x \,.\, x))^{\natural} &&= \mathcal{S} \\
&k \flat\, (\varepsilon, M) &&= M \\
&(\lambda\, x \,.\, x) \flat\, (\varepsilon, M) &&= M \\
&(\lambda\, x \,.\, N_{\Delta,\Sigma x}) \flat\, (\sigma, M) &&= N_{\Delta,\Sigma x} \sharp\, \sigma \cdot M
\end{aligned}$$

Figure 12 Back to Direct Style from $\lambda_{\mathcal{S}}^{*}$. We assume $|\sigma| = |\Sigma|$ for the $\sharp$ and $\flat$ translations. The $\natural$ translation only works on V_ε – the other value cases are handled explicitly, hence variables annotated with themselves.

$$\begin{tikzcd}
 & \lambda_{c\mathcal{S}} \arrow[r, "*"] & \lambda_{c\mathcal{S}}^{*} \arrow[dl, "\#"'] \\
\lambda_{\mathcal{S}} \arrow[ur, hook] \arrow[rr, bend left=20, "*"] & \lambda_{c\mathcal{S}}^{\triangleright} \arrow[u, hook] & \lambda_{\mathcal{S}}^{*} \arrow[u, hook] \arrow[dll, "\#"] \\
\lambda_{\mathcal{S}}^{\triangleright} \arrow[u, "\triangleleft"'] \arrow[ur, hook] \arrow[urr, "*"] & &
\end{tikzcd}$$

Figure 13 Summary of the relationships between the calculi. Hooked arrows denote sub-calculi, $*$ denotes the CPS transformation and # the DS transformation; both can be retracted along some of the inclusions.

▶ **Theorem 29** (Isomorphism of $\lambda_{\mathcal{S}}$ and $\lambda_{\mathcal{S}}^{*}$). *Transformations* $* : \lambda_{\mathcal{S}} \to \lambda_{\mathcal{S}}^{*}$ *and* $\diamond : \lambda_{\mathcal{S}}^{*} \to \lambda_{\mathcal{S}}$ *form an isomorphism.*

7 Conclusion and Future Work

In this work we established a reflection of the image of the CPS translation for the computational lambda calculus λ_c extended with `shift` and `reset`. We also showed that when restricted to an extension of the call-by-value lambda calculus λ_v, the reflection actually forms an isomorphism. To the best of our knowledge, this is the first study that formally establishes such a tight relationship of the direct-style and CCS reduction theories. In particular, the direct-style translation from CCS to $\lambda_{\mathcal{S}}$ that is an inverse of the CPS translation, appears to be a first such translation for `shift` and `reset` in the literature. It can be seen as a continuation of the works by Danvy [2], and by Danvy and Lawall [4]. The connections between the various calculi we studied are summarised in Figure 13.

Besides the theoretical aspects of the presented results, one can view them as a source of sound code optimisations that can be performed both at the level of the source and the target of the CPS translation, which is a standard translation step in compilers. Moreover, in the light of Filinski's seminal result [7], our theory makes it possible to reason about any monadic effect, since the direct-style monad operations `reflect` and `reify` are expressible in terms of `shift` and `reset`.

Several possible directions for future work are on the horizon. First of all, the `shift` operator as considered in this work is a combinator. It seems that if, instead, `shift` was introduced as a special form or as a binder, characterising the image of the CPS translation would require more machinery. Especially in the latter case, we would need to pay special attention to the continuation identifiers bound by `shift`. Introducing a construct `throw` for applying a captured continuation could turn out useful in that scenario.

A delimited-control operator that has been lately gaining currency is `shift0`, a seemingly mild variation on `shift` [3]. This operator is intimately related to the mechanism of algebraic effects and deep handlers [9], a fairly recent and much celebrated approach to computational effects. Establishing a reflection for `shift0`, based on the existing CPS translations [14, 13] would be interesting in its own right, but it could also pave a way to a similar theory for algebraic effects.

Finally, it is quite plausible that following the lines of the present work, one could obtain a similar set of reflections and isomorphisms for a calculus with `call/cc`.

References

1 Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, New York, 1992.

2 Olivier Danvy. Back to direct style. *Sci. Comput. Program.*, 22(3):183–195, 1994. doi:10.1016/0167-6423(94)00003-4.

3 Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*, pages 151–160. ACM, 1990. doi:10.1145/91556.91622.

4 Olivier Danvy and Julia L. Lawall. Back to direct style II: first-class continuations. In *Proceedings of the Conference on Lisp and Functional Programming, LFP 1992, San Francisco, California, USA, 22-24 June 1992*, pages 299–310. ACM, 1992. doi:10.1145/141471.141564.

5 Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*, pages 193–222. North-Holland, 1987.

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

7 Andrzej Filinski. Representing monads. In Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin, editors, *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pages 446–457. ACM Press, 1994. doi:10.1145/174675.178047.

8 Michael J. Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6(3-4):259–288, 1993.

9 Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control. *PACMPL*, 1(ICFP):13:1–13:29, 2017. doi:10.1145/3110257.

10 Annius Groenink. Literal movement grammars. In Steven P. Abney and Erhard W. Hinrichs, editors, *EACL 1995, 7th Conference of the European Chapter of the Association for Computational Linguistics, March 27-31, 1995, University College Dublin, Belfield, Dublin, Ireland*, pages 90–97. The Association for Computer Linguistics, 1995. URL: `https://www.aclweb.org/anthology/E95-1013/`.

11 John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin, editors, *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pages 458–471. ACM Press, 1994. `doi:10.1145/174675.178053`.

12 Yukiyoshi Kameyama and Masahito Hasegawa. A sound and complete axiomatization of delimited continuations. In Olin Shivers, editor, *Proceedings of the 2003 ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, SIGPLAN Notices, Vol. 38, No. 9, pages 177–188, Uppsala, Sweden, August 2003. ACM Press.

13 Marek Materzok. Axiomatizing subtyped delimited continuations. In Simona Ronchi Della Rocca, editor, *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy*, volume 23 of *LIPIcs*, pages 521–539. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013. `doi:10.4230/LIPIcs.CSL.2013.521`.

14 Marek Materzok and Dariusz Biernacki. Subtyping delimited continuations. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 81–93. ACM, 2011. `doi:10.1145/2034773.2034786`.

15 Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 14–23. IEEE Computer Society, 1989. `doi:10.1109/LICS.1989.39155`.

16 Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975. `doi:10.1016/0304-3975(75)90017-1`.

17 John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. `doi:10.1023/A:1010027404223`.

18 John C. Reynolds. *Theories of programming languages*. Cambridge University Press, 1998.

19 Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993.

20 Amr Sabry and Philip Wadler. A reflection on call-by-value. *ACM Trans. Program. Lang. Syst.*, 19(6):916–941, 1997. `doi:10.1145/267959.269968`.

21 David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

22 Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 20(4):371–401, 2007. `doi:10.1007/s10990-007-9010-4`.

23 Guy L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474.

24 Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1/2):135–152, 2000. `doi:10.1023/A:1010026413531`.

25 Philip Wadler. The essence of functional programming. In Ravi Sethi, editor, *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, pages 1–14. ACM Press, 1992. `doi:10.1145/143165.143169`.

A Probabilistic Higher-Order Fixpoint Logic

Yo Mitani
The University of Tokyo, Japan
mitaniyo@kb.is.s.u-tokyo.ac.jp

Naoki Kobayashi
The University of Tokyo, Japan
koba@kb.is.s.u-tokyo.ac.jp

Takeshi Tsukada
The University of Tokyo, Japan
tsukada@kb.is.s.u-toyko.ac.jp

Abstract

We introduce PHFL, a probabilistic extension of higher-order fixpoint logic, which can also be regarded as a higher-order extension of probabilistic temporal logics such as PCTL and the μ^p-calculus. We show that PHFL is strictly more expressive than the μ^p-calculus, and that the PHFL model-checking problem for finite Markov chains is undecidable even for the μ-only, order-1 fragment of PHFL. Furthermore the full PHFL is far more expressive: we give a translation from Lubarsky's μ-arithmetic to PHFL, which implies that PHFL model checking is Π^1_1-hard and Σ^1_1-hard. As a positive result, we characterize a decidable fragment of the PHFL model-checking problems using a novel type system.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Probabilistic logics, higher-order fixpoint logic, model checking

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.19

Funding This work was supported by JSPS KAKENHI Grant Number JP15H05706 and 20H00577.

Acknowledgements We would like to thank anonymous referees for useful comments.

1 Introduction

Temporal logics such as CTL and CTL* have been playing important roles, for example, in system verification. Among the most expressive temporal logics is the *higher-order fixpoint logic* (*HFL* for short) proposed by Viswanathan and Viswanathan [22], which is a higher-order extension of the *modal μ-calculus* [13]. HFL is known to be strictly more expressive than the modal μ-calculus but the model-checking problem against finite models is still decidable.

In view of the increasing importance of probabilistic systems, temporal logics for probabilistic systems (such as PCTL [7]) and their model-checking problems have been studied and applied to verification and analysis of probabilistic systems and randomized distributed algorithms [14]. Recently Castro et al. [2] have proposed a probabilistic extension of the modal μ-calculus, called the *μ^p-calculus*. They showed that the μ^p-calculus is strictly more expressive than PCTL and that the model-checking problem for the μ^p-calculus belongs to NP ∩ co-NP.

In the present paper, we introduce *PHFL*, a probabilistic higher-order fixpoint logic, which can be regarded as a probabilistic extension of HFL and as a higher-order extension of the μ^p-calculus. PHFL strictly subsumes the μ^p-calculus [2], which coincides with order-0 PHFL.

We prove that PHFL model checking for finite Markov chains is undecidable even for the order-1 fragment of PHFL without fixpoint alternations, by giving a reduction of the *value problem* of probabilistic automata [21, 20]. In the presence of fixpoint alternations (i.e.,

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 19; pp. 19:1–19:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

with both least and greatest fixpoint operators), PHFL model checking is even harder: the order-1 PHFL model-checking problem is Π^1_1-hard and Σ^1_1-hard. The proof is by a reduction from the validity checking problem for μ-arithmetic [16] to PHFL model checking. This may be surprising, because both order-0 PHFL model checking (i.e. μ^p-calculus model checking) for finite Markov chains [2] and HFL model checking for finite state systems [22] are decidable. The combination of probabilities and higher-order predicates suddenly makes the model-checking problem highly undecidable.

As a positive result, we identify a decidable subclass of PHFL model-checking problems. To characterize the subclass, we introduce a type system for PHFL formulas, which is parameterized by Markov chains M. We show that the model-checking problem $M \models \varphi$ is decidable provided that φ is typable by the type system for M, by giving a decision procedure using the decidability of existential theories of reals. The decidable subclass is reasonably expressive: the problem of computing termination probabilities of *recursive Markov chains* [3] can be reduced to the subclass.

The rest of this paper is organized as follows. Section 2 introduces PHFL and shows that it is strictly more expressive than the μ^p-calculus. Section 3 proves undecidability of the model-checking problem for μ-only and order-1 PHFL. Section 4 proves that the PHFL model-checking problem is both Π^1_1-hard and Σ^1_1-hard. Section 5 introduces a decidable subclass of PHFL model-checking problems, and shows that the subclass is reasonably large. Section 6 discusses related work, and Section 7 concludes the paper. Proofs omitted in the paper are found in a longer version of this paper [18].

2 PHFL: Probabilistic Higher-order Fixpoint Logic

This section introduces PHFL, a probabilistic extension of HFL [22]. It is a logic used for describing properties of Markov chains. We define its syntax and semantics and show that it is more expressive than the μ^p-calculus [2].

2.1 Markov Chains

We first recall the standard notion of Markov chains. Our definitions follow those in [2].

▶ **Definition 1.** *A* Markov chain *over a set AP of atomic propositions is a tuple $(S, P, \rho_{AP}, s_{in})$ where*

- *S is a finite set of states,*
- *$P : S \times S \to [0, 1]$ satisfying $\forall s. \sum_{s' \in S} P(s, s') = 1$ describes transition probabilities,*
- *$\rho_{AP} : AP \to 2^S$ is a labeling function, and*
- *$s_{in} \in S$ is an initial state.*

For a Markov chain $M = (S, P, \rho_{AP}, s_{in})$, its embedded Kripke structure *is $K = (S, R, \rho_{AP}, s_{in})$ where $R \subseteq S \times S$ is a relation such that $R = \{(s, s') | P(s, s') > 0\}$.*

Intuitively, $P(s, s')$ denotes the probability that the state s transits to the state s', and $\rho_{AP}(p)$ gives the set of states where p is true. Throughout the paper, we assume that the set AP of atomic propositions is closed under negations, in the sense that for any $p \in AP$, there exists $\overline{p} \in AP$ such that $\rho_{AP}(\overline{p}) = S \setminus \rho_{AP}(p)$.

Given a Markov chain M, we often write $S_M, P_M, \rho_{AP,M}, s_{\mathrm{in},M}$ for its components; we omit the subscript M when it is clear from the context.

2.2 Syntax of PHFL Formulas

As in HFL [22, 11], we need the notion of types to define the syntax of PHFL formulas.

The set of types, ranged over by τ, is given by:

$$\tau ::= \mathit{Prop}_{\{0,1\}} \mid \mathit{Prop}_{[0,1]} \mid \tau_1 \to \tau_2.$$

The type $\mathit{Prop}_{\{0,1\}}$ is for *qualitative* propositions, which take truth values (0 for false, and 1 for true). In contrast, $\mathit{Prop}_{[0,1]}$ is the type of *quantitative* propositions, whose values range over $[0, 1]$. Intuitively, the value of a quantitative proposition represents the *probability* that the proposition holds. The type $\tau_1 \to \tau_2$ is for functions from τ_1 to τ_2. For example, $(\mathit{Prop}_{\{0,1\}} \to \mathit{Prop}_{\{0,1\}}) \to \mathit{Prop}_{[0,1]}$ represents the type of (higher-order) quantitative predicates on a qualitative predicate.

We assume a countably infinite set *Var* of variables, ranged over by $X_1, X_2, \dots$. The set of PHFL (pre-)formulas, ranged over by ϕ, is given by:

$$\phi ::= p \mid X \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid [\phi]_J \mid \{\phi\} \mid \Box\phi \mid \Diamond\phi \mid \bigcirc\,\phi \mid \mu X.\phi \mid \nu X.\phi \mid \lambda X.\phi \mid \phi_1\,\phi_2.$$

Here, p ranges over the set AP of atomic propositions (of the underlying Markov chains; we thus assume that AP is closed under negations). The subscript J of $[\phi]_J$ is either "$> r$" or "$\geq r$" for some rational number $r \in [0, 1]$. We often identify J with an interval: for example, "$> r$" is regarded as $(r, 1] = \{\, x \mid r < x \leq 1 \,\}$. Given a quantitative proposition ϕ, the formula $[\phi]_{>r}$ (resp. $[\phi]_{\geq r}$) is a qualitative formula, which is true just if the probability that ϕ holds is greater than r (resp. no less than r). The formulas $\Box\phi$, $\Diamond\phi$, and $\bigcirc\phi$ respectively mean the minimum, maximum, and average probabilities that ϕ holds after a one-step transition. The formulas $\mu X.\phi$ and $\nu X.\phi$ respectively denote the least and greatest fixpoints of $\lambda X.\phi$. Note that ϕ may denote higher-order predicates (unlike in the modal μ-calculus and its probabilistic variants [2, 17, 19], where fixpoints are restricted to propositions). We have also λ-abstractions and applications, to manipulate higher-order predicates. The prefixes μX, νX and λX bind the variable X. As usual, we identify formulas up to the renaming of bound variables and implicitly allow α-conversions.

In order to exclude out ill-formed formulas like $(p_1 \vee p_2)(\phi)$, we restrict the shape of formulas through a type system. A *type environment* is a map from a finite set of variables to the set of types. A *type judgment* is of the form $\Gamma \vdash \phi : \tau$. The typing rules are shown in Figure 1. In the figure, $\mathtt{P}$ is a meta-variable ranging over the set $\{\mathit{Prop}_{\{0,1\}}, \mathit{Prop}_{[0,1]}\}$ of proposition types. For example, the rule for $\phi_1 \wedge \phi_2$ means that $\Gamma \vdash \phi_i \colon \mathit{Prop}_{\{0,1\}}$ for each $i \in \{1, 2\}$ implies $\Gamma \vdash \phi_1 \wedge \phi_2 \colon \mathit{Prop}_{\{0,1\}}$ and that $\Gamma \vdash \phi_i \colon \mathit{Prop}_{[0,1]}$ for each $i \in \{1, 2\}$ implies $\Gamma \vdash \phi_1 \wedge \phi_2 \colon \mathit{Prop}_{[0,1]}$. A formula ϕ is *well-typed* if $\Gamma \vdash \phi : \tau$ is derivable for some Γ and τ. Henceforth, we consider only well-typed formulas.

▶ **Example 2.** For a proposition $p \in AP$, the formula $\phi = (\mu F.\lambda X.X \vee F(\bigcirc X))\,\{p\}$ is a well-typed formula of type $\mathit{Prop}_{[0,1]}$. By unfolding the fixpoint formula, we obtain:

$$\begin{aligned}
\phi &\equiv (\lambda X.X \vee (\mu F.\lambda X.X \vee F(\bigcirc X))(\bigcirc X))\{p\} \\
&\equiv \{p\} \vee (\mu F.\lambda X.X \vee F(\bigcirc X))(\bigcirc\{p\}) \\
&\equiv \{p\} \vee \bigcirc\{p\} \vee (\mu F.\lambda X.X \vee F(\bigcirc X))(\bigcirc\bigcirc\{p\}) \\
&\equiv \{p\} \vee \bigcirc\{p\} \vee \bigcirc\bigcirc\{p\} \vee \cdots
\end{aligned}$$

Thus, intuitively, the formula represents the function that maps each state s to the value $\sup_{k \geq 0} q_k$ where q_k is the probability that a k-step transition sequence starting from the state s ends in a state satisfying p. ◀

$$\frac{}{\Gamma \vdash p : \mathit{Prop}_{\{0,1\}}} \qquad \frac{}{\Gamma, X : \tau \vdash X : \tau} \qquad \frac{\Gamma \vdash \phi : \mathit{Prop}_{[0,1]}}{\Gamma \vdash [\phi]_J : \mathit{Prop}_{\{0,1\}}} \qquad \frac{\Gamma \vdash \phi : \mathit{Prop}_{\{0,1\}}}{\Gamma \vdash \{\phi\} : \mathit{Prop}_{[0,1]}}$$

$$\frac{\Gamma \vdash \phi_1, \phi_2 : \mathtt{P}}{\Gamma \vdash \phi_1 \wedge \phi_2 : \mathtt{P}} \qquad \frac{\Gamma \vdash \phi_1, \phi_2 : \mathtt{P}}{\Gamma \vdash \phi_1 \vee \phi_2 : \mathtt{P}} \qquad \frac{\Gamma \vdash \phi : \mathtt{P}}{\Gamma \vdash \Box\phi : \mathtt{P}}$$

$$\frac{\Gamma \vdash \phi : \mathtt{P}}{\Gamma \vdash \Diamond\phi : \mathtt{P}} \qquad \frac{\Gamma \vdash \phi : \mathit{Prop}_{[0,1]}}{\Gamma \vdash \bigcirc\phi : \mathit{Prop}_{[0,1]}} \qquad \frac{\Gamma, X : \tau \vdash \phi : \tau}{\Gamma \vdash \mu X.\phi : \tau}$$

$$\frac{\Gamma, X : \tau \vdash \phi : \tau}{\Gamma \vdash \nu X.\phi : \tau} \qquad \frac{\Gamma, X : \tau_1 \vdash \phi : \tau_2}{\Gamma \vdash \lambda X.\phi : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash \phi : \tau_1 \to \tau_2 \qquad \Gamma \vdash \psi : \tau_1}{\Gamma \vdash \phi\,\psi : \tau_2}$$

Figure 1 Type Derivation Rules for PHFL.

▶ Remark 3. Following [11], we have excluded out negations. By a transformation similar to that in [15] and our assumption that the set of atomic propositions is closed under negations, any closed formula of PHFL extended with negations can be transformed to an equivalent negation-free formula. ◀

We define the *order* of a type τ by:

$$\mathit{order}(\mathit{Prop}_{\{0,1\}}) = \mathit{order}(\mathit{Prop}_{[0,1]}) = 0 \qquad \mathit{order}(\tau_1 \to \tau_2) = \max(\mathit{order}(\tau_1){+}1, \mathit{order}(\tau_2)).$$

The order of a formula ϕ such that $\Gamma \vdash \phi : \tau$ is the largest order of types used in the derivation of $\Gamma \vdash \phi : \tau$. The *order-$k$ PHFL* is the fragment of PHFL consisting of formulas of order up to k. Order-0 PHFL coincides with the μ^p-calculus [2].

2.3 Semantics

We first give the semantics of types. We write $\leq_{\mathbb{R}}$ for the natural order over the set $\mathbb{R}$ of real numbers, and often omit the subscript when there is no danger of confusion. For a map f, we write $\mathit{dom}(f)$ for the domain of f.

▶ **Definition 4** (Semantics of Types). *For each τ, we define a partially ordered set $[\![\tau]\!] = (D_\tau, \leq_\tau)$ inductively by:*

$$\begin{aligned}
&D_{\mathit{Prop}_{\{0,1\}}} = S \to \{0,1\} \qquad f \leq_{\mathit{Prop}_{\{0,1\}}} g \overset{def}{\iff} \forall s \in S. f(s) \leq g(s)\\
&D_{\mathit{Prop}_{[0,1]}} = S \to [0,1] \qquad f \leq_{\mathit{Prop}_{[0,1]}} g \overset{def}{\iff} \forall s \in S. f(s) \leq g(s)\\
&D_{\tau_1 \to \tau_2} = \{ f \in D_{\tau_1} \to D_{\tau_2} \mid \forall x, y \in D_{\tau_1}. x \leq_{\tau_1} y \implies f(x) \leq_{\tau_2} f(y) \}\\
&f \leq_{\tau_1 \to \tau_2} g \overset{def}{\iff} \forall x \in D_{\tau_1}. f(x) \leq_{\tau_2} g(x).
\end{aligned}$$

For a type environment Γ, we write $[\![\Gamma]\!]$ for the set of maps f such that $\mathit{dom}(f) = \mathit{dom}(\Gamma)$ and $f(x) \in D_{\Gamma(x)}$ for every $x \in \mathit{dom}(\Gamma)$.

Note that $[\![\tau]\!]$ forms a complete lattice for each τ. We write $\bot_\tau$ for the least element of $[\![\tau]\!]$, and for a set $V \subseteq D_\tau$, we write $\bigvee_\tau V$ for the least upper bound of S with respect to $\leq_\tau$; we often omit the subscript τ if it is clear from the context. Note also that for every functional type $\tau_1 \to \tau_2$, every element of $D_{\tau_1 \to \tau_2}$ is monotonic. Thus, for every type τ and every function $f \in D_{\tau \to \tau}$, the least and greatest fixed points of f exist.

We now define the semantics of formulas. Since the meaning of a formula depends on its type environment, we actually define the semantics $[\![\Gamma \vdash \phi : \tau]\!]_M$ for each type judgment $\Gamma \vdash \phi : \tau$. Here, M is the underlying Markov chain, which is often omitted.

▶ **Definition 5** (Semantics of Type Judgement). *Let M be a Markov chain and assume $\Gamma \vdash \phi : \tau$ is derivable. Then its semantics $[\![\Gamma \vdash \phi : \tau]\!]_M \in [\![\Gamma]\!] \to [\![\tau]\!]$ is defined by induction on the (unique) derivation of $\Gamma \vdash \phi : \tau$ by:*

$$
\begin{aligned}
[\![\Gamma \vdash p : \mathit{Prop}_{\{0,1\}}]\!]_M(\rho) &= \lambda s \in S_M.\mathit{if}\ s \in \rho_{AP,M}(p)\ \mathit{then}\ 1\ \mathit{else}\ 0\\
[\![\Gamma \vdash X : \tau]\!]_M(\rho) &= \rho(X)\\
[\![\Gamma \vdash \phi_1 \wedge \phi_2 : \mathtt{P}]\!]_M(\rho) &= \lambda s \in S_M.\ \min_{i \in \{1,2\}} [\![\Gamma \vdash \phi_i : \mathtt{P}]\!]_M(\rho)(s)\\
[\![\Gamma \vdash \phi_1 \vee \phi_2 : \mathtt{P}]\!]_M(\rho) &= \lambda s \in S_M.\ \max_{i \in \{1,2\}} [\![\Gamma \vdash \phi_i : \mathtt{P}]\!]_M(\rho)(s)\\
[\![\Gamma \vdash [\phi]_J : \mathit{Prop}_{\{0,1\}}]\!]_M(\rho) &= \lambda s \in S_M.\mathit{if}\ [\![\Gamma \vdash \phi : \mathit{Prop}_{[0,1]}]\!]_M(\rho)(s) \in J\ \mathit{then}\ 1\ \mathit{else}\ 0\\
[\![\Gamma \vdash \{\phi\} : \mathit{Prop}_{[0,1]}]\!]_M(\rho) &= [\![\Gamma \vdash \phi : \mathit{Prop}_{\{0,1\}}]\!]_M(\rho)\\
[\![\Gamma \vdash \Box\phi : \mathtt{P}]\!]_M(\rho) &= \lambda s \in S_M.\ \min_{s' : P_M(s,s') > 0} [\![\Gamma \vdash \phi : \mathtt{P}]\!]_M(\rho)(s')\\
[\![\Gamma \vdash \Diamond\phi : \mathtt{P}]\!]_M(\rho) &= \lambda s \in S_M.\ \max_{s' : P_M(s,s') > 0} [\![\Gamma \vdash \phi : \mathtt{P}]\!]_M(\rho)(s')\\
[\![\Gamma \vdash \bigcirc\phi : \mathit{Prop}_{[0,1]}]\!]_M(\rho) &= \lambda s \in S_M.\ \sum_{s' \in S_M} P_M(s,s')[\![\Gamma \vdash \phi : \mathit{Prop}_{[0,1]}]\!]_M(\rho)(s')\\
[\![\Gamma \vdash \mu X.\phi : \tau]\!]_M(\rho) &= \mathit{LFP}(\lambda v \in D_\tau.[\![\Gamma, X : \tau \vdash \phi : \tau]\!]_M(\rho[X \mapsto v]))\\
[\![\Gamma \vdash \nu X.\phi : \tau]\!]_M(\rho) &= \mathit{GFP}(\lambda v \in D_\tau.[\![\Gamma, X : \tau \vdash \phi : \tau]\!]_M(\rho[X \mapsto v]))\\
[\![\Gamma \vdash \lambda X.\phi : \tau_1 \to \tau_2]\!]_M(\rho) &= \lambda v \in D_{\tau_1}.[\![\Gamma, X : \tau_1 \vdash \phi : \tau_2]\!]_M(\rho[X \mapsto v])\\
[\![\Gamma \vdash \phi_1\, \phi_2]\!]_M(\rho) &= [\![\Gamma \vdash \phi_1]\!]_M(\rho)\,([\![\Gamma \vdash \phi_2]\!]_M(\rho))
\end{aligned}
$$

Here $\mathtt{P} \in \{\, \mathit{Prop}_{\{0,1\}}, \mathit{Prop}_{[0,1]} \,\}$.

In the definitions of the semantics of $\Box\phi$ and $\Diamond\phi$, the set $S' = \{s' \in S | P(s,s') > 0\}$ is non-empty and finite, because $\sum_{s' \in S} P(s,s') = 1$ and S is finite by the definition of Markov chains. Thus the max/min operations are well-defined. We also note that $[\![\Gamma \vdash \phi : \tau]\!]$ is a monotone function from $[\![\Gamma]\!]$ to $[\![\tau]\!]$ (here $[\![\Gamma]\!]$ is ordered by the component-wise ordering; note also Remark 6 below). This ensures the well-definedness of the semantics of abstractions.

▶ Remark 6. Recall that in a formula $[\phi]_J$, we allow the predicate J to be "$> r$" or "$\geq r$" (where $r \in [0,1]$), but neither "$< r$" nor "$\leq r$". Allowing "$< r$" would break the monotonicity of the semantics of a formula. For example, $[\![\emptyset \vdash \lambda X.[X]_{<1} : \mathit{Prop}_{[0,1]} \to \mathit{Prop}_{\{0,1\}}]\!] = \lambda v \in D_{\mathit{Prop}_{[0,1]}}.\lambda s \in S.(\text{if } v(s) < 1 \text{ then } 1 \text{ else } 0)$ is not monotonic. ◀

We often omit M, the type of the formula, and the type environment in the notation of semantics when there is no confusion and just write $[\![\phi]\!]$ or $[\![\Gamma \vdash \phi]\!]$ for $[\![\Gamma \vdash \phi : \tau]\!]_M$. For a Markov chain $M = (S, P, \rho_{AP}, s_{\text{in}})$ and a closed PHFL formula ϕ of type $\mathit{Prop}_{\{0,1\}}$, we write $M \models \phi$ if $[\![\phi]\!](s_{\text{in}}) = 1$.

▶ **Example 7.** Recall the PHFL formula $\phi = \psi\{p\}$ where $\psi = \mu F.\lambda X.X \vee F(\bigcirc X)$ in Example 2. We have

$$[\![\psi]\!] = \text{LFP}\Big(\lambda v \in D_{\mathit{Prop}_{[0,1]} \to \mathit{Prop}_{[0,1]}}.\lambda x \in D_{\mathit{Prop}_{[0,1]}}.\lambda s \in S.$$

$$\begin{aligned}
&\max\Big(x\,s, v\,(\lambda s' \in S. \sum_{s''} P(s', s'') \cdot (xs''))\,s\Big)\Big) \\
&\geq \Big(\lambda v.\lambda x.\lambda s.\max\Big(x\,s, v\,(\lambda s' \in S. \sum_{s''} P(s', s'') \cdot (xs''))\,s\Big)\Big)^{n+1}(\bot_{Prop_{[0,1]}\to Prop_{[0,1]}}) \\
&= \lambda x.\lambda s.\max_{0\leq k\leq n} \sum_{s_0 s_1 \dots s_k \in S^{k+1}, s_0 = s} \big(x(s_k) \cdot \prod_{0\leq j\leq k-1} P(s_j, s_{j+1})\big)
\end{aligned}$$

for every $n \geq 0$. Thus, we have:

$$[\![\psi]\!] \geq \lambda x.\lambda s \in S.\sup_{k\geq 0} \sum_{s_0 s_1 \dots s_k \in S^{k+1}, s_0 = s} \big(x(s_k) \cdot \prod_{0\leq j\leq k-1} P(s_j, s_{j+1})\big).$$

Actually, the equality holds, because the righthand side is a fixpoint of

$$\lambda v \in D_{Prop_{[0,1]}\to Prop_{[0,1]}}.\lambda x \in D_{Prop_{[0,1]}}.\max(x, v(\lambda s \in S. \sum_{s'} P(s, s') \cdot (xs'))).$$

The semantics of ϕ is, therefore, given by

$$[\![\phi]\!] = \lambda s \in S.\sup_{k\geq 0} \sum_{s_0 s_1 \dots s_k \in S^{k+1}, s_0 = s} \big(\rho_{AP}(p)(s_k) \cdot \prod_{0\leq j\leq k-1} P(s_j, s_{j+1})\big). \blacktriangleleft$$

2.4 Expressive Power

PHFL obviously subsumes the μ^p-calculus [2], which coincides with order-0 PHFL. Hence PHFL also subsumes PCTL [7], since the μ^p-calculus subsumes PCTL [2].

PHFL is *strictly* more expressive than the μ^p-calculus.

▶ **Theorem 8.** *Order-1 PHFL is strictly more expressive than the μ^p-calculus, i.e., there exists an order-1 PHFL proposition ϕ such that ϕ is not equivalent to any μ^p-formula.*

Proof. Let $\mathcal{M}$ be the set of Markov chains $M = (S, P, \rho_{AP}, s_{\text{in}})$ that satisfy the following conditions.

- $S = \{s_0, s_1, \dots, s_n\}$ for a positive integer n,
- $P(s_i, s_{i+1}) = 1$ $(0 \leq i \leq n-1)$, $P(s_n, s_n) = 1$ and $P(s_i, s_j) = 0$ otherwise.
- There are three atomic propositions a, b, c with $\rho_{AP}(a) \cup \rho_{AP}(b) = \{s_0, s_1, \dots, s_{n-1}\}$, $\rho_{AP}(a) \cap \rho_{AP}(b) = \emptyset$ and $\rho_{AP}(c) = \{s_n\}$.
- The initial state is $s_{\text{in}} = s_0$

Let ϕ be the order-1 PHFL formula of type $Prop_{\{0,1\}}$:

$$(\mu F.\lambda X.a \wedge \Diamond(X \vee F(b \wedge \Diamond X)))(b \wedge \Diamond c).$$

Note that, for $M \in \mathcal{M}$, $M \models \phi$ holds just if n is even, and ρ_{AP} satisfies $\rho_{AP}(a) = \{s_0, s_1, \dots, s_{\frac{n}{2}-1}\}$ and $\rho_{AP}(b) = \{s_{\frac{n}{2}}, s_{\frac{n}{2}+1}, \dots, s_{n-1}\}$.

We show that there is no μ^p-formula equivalent to ϕ. Suppose that a μ^p-formula ϕ' *were* equivalent to ϕ, which would imply that $M \models \phi$ if and only if $M \models \phi'$ for any $M \in \mathcal{M}$. For $M \in \mathcal{M}$, let us write K_M for the embedded Kripke structure of M. Since all the transitions in $\mathcal{M}$ are deterministic, there exists a modal μ-calculus formula ϕ'' such that $M \models \phi'$ if and only if $K_M \models \phi''$ (note that ϕ'' is obtained by replacing $\bigcirc$ with $\Diamond$, and replacing $[\phi_1]_J$ with true if J is "≥ 0" and with ϕ_1 otherwise). That would imply that $K_M \models \phi''$ for $M \in \mathcal{M}$, just if n is even and ρ_{AP} satisfies $\rho_{AP}(a) = \{s_0, s_1, \dots, s_{\frac{n}{2}-1}\}$ and $\rho_{AP}(b) = \{s_{\frac{n}{2}}, s_{\frac{n}{2}+1}, \dots, s_{n-1}\}$. But then ϕ'' corresponds to the non-regular language $\{a^m b^m \mid m \geq 1\}$, which contradicts the fact that the modal μ-calculus can express only regular properties. ◀

3 Undecidability of PHFL Model Checking

In this section we prove the undecidability of the following problem.

▶ **Definition 9** (PHFL Model Checking). *The* PHFL model-checking problem *for finite Markov chains is the problem of deciding whether $M \models \phi$, given a (finite) Markov chain M and a closed PHFL formula ϕ of type $\mathit{Prop}_{\{0,1\}}$ as input.*

We prove that the problem is undecidable even for the order-1 fragment of PHFL without fixpoint alternations, by a reduction from the undecidability of the value-1 problem [6] for probabilistic automata [21]. In contrast to the undecidability of PHFL model checking, the corresponding model-checking problems are *decidable* for the full fragments of the μ^p-calculus [2] and (non-probabilistic) HFL [22], with fixpoint alternations. Thus, the combination of probabilities and higher-order predicates introduces a new difficulty.

In Section 3.1, we review the definition of probabilistic automata and the value-1 problem. Section 3.2 shows the reduction from the value-1 problem to the PHFL model-checking problem.

3.1 Probabilistic Automata

We review probabilistic automata [21] and the undecidability of the value-1 problem. Our definition follows [4].

▶ **Definition 10** (Probabilistic Automata). *A* probabilistic automaton *A is a tuple $(Q, \Sigma, q_I, \Delta, F)$ where*

- *Q is a finite set of states,*
- *Σ is a finite set of input symbols,*
- *$q_I \in Q$ is an initial state,*
- *$\Delta : Q \times \Sigma \to D(Q)$, where $D(Q) := \{ f : Q \to [0,1] \mid \sum_{q \in Q} f(q) = 1 \}$ is the set of probabilistic distributions over the set Q, represents transition probabilities, and*
- *$F \subseteq Q$ is a set of accepting states.*

For a word $w = w_1 \cdots w_n \in \Sigma^n$, the probability that w is accepted by $A = (Q, \Sigma, q_I, \Delta, F)$, written $A(w)$, is defined by:

$$A(w) := \sum_{\substack{q_0,\dots,q_{n-1} \in Q, q_n \in F \\ s.t.\ q_0 = q_I}} \prod_{1 \le i \le n} \Delta(q_{i-1}, w_i)(q_i).$$

The *value* of a probabilistic automaton A, denoted by $\mathrm{val}(A)$, is defined by

$$\mathrm{val}(A) := \sup_{w \in \Sigma^*} A(w).$$

The problem of deciding whether $\mathrm{val}(A) = 1$, called the *value-1 problem*, is known to be undecidable.

▶ **Theorem 11** (Undecidability of The Value-1 Problem [6]). *Given a probabilistic automaton A, whether $\mathit{val}(A) = 1$ is undecidable.*

3.2 The Undecidability Result

Let $A = (Q, \Sigma, q_I, \Delta, F)$ be a probabilistic automaton, where $\Sigma = \{c_1, \dots, c_{|\Sigma|}\}$ with $|\Sigma| > 0$. We shall construct a Markov chain M_A and a PHFL formula ϕ_A, so that $\mathrm{val}(A) = 1$ if and only if $M_A \models \phi_A$. The undecidability of PHFL model checking then follows immediately from Theorem 11.

We first construct a Markov chain. The set AP of atomic propositions is $\{\, p_c \mid c \in \Sigma \,\} \uplus \{\, p_F \,\}$. The Markov chain $M_A = (S, P, \rho_{AP}, s_{\mathrm{in}})$ is defined as follows.

- The set S of states is $Q \uplus (Q \times \Sigma)$.
- The transition probability P is given by:

$$\begin{aligned} P((q,c),q') &= \Delta(q,c)(q') && (c \in \Sigma \text{ and } q,q' \in Q)\\ P(q,(q,c)) &= \frac{1}{|\Sigma|} && (c \in \Sigma \text{ and } q \in Q)\\ P(s,s') &= 0 && (\text{otherwise}) \end{aligned}$$

  The first transition (from (q,c) to q') is used to simulate the transition of A from q to q' for the input symbol c. The second transition (from q to (q,c)) is used to choose the next input symbol to be supplied to the automaton; the probability is not important, and replacing $1/|\Sigma|$ with any non-zero probability does not affect the following arguments.
- ρ_{AP} is defined by:

$$\rho_{AP}(p_c) = \{\, (q,c) \mid q \in Q \,\} \qquad \rho_{AP}(p_F) = \{\, q \mid q \in F \,\}.$$

- The initial state is $s_{\mathrm{in}} = q_I$.

Intuitively, the Markov chain M_A simulates the behavior of A. The atomic proposition p_c means that A is currently reading the symbol c, and p_F means that A is in a final state.

Based on this intuition, we now construct the PHFL formula ϕ_A. For each $c \in \Sigma$, we define a formula f_c of type $\mathit{Prop}_{[0,1]} \to \mathit{Prop}_{[0,1]}$ by:

$$f_c := \lambda X.\Diamond(\{p_c\} \wedge \bigcirc X).$$

Intuitively $f_c(\phi)$ denotes the probability that the automaton transits to a state satisfying ϕ given c as the next input. Given a word $w = w_1 w_2 \dots w_n \in \Sigma^*$, we define the formula g_w by

$$g_w := f_{w_1}(f_{w_2}(\dots(f_{w_n}\{p_F\})\dots)).$$

We write A_q for the automaton obtained from A by replacing the initial state with q.

► **Lemma 12.** *$A_q(w) = [\![g_w]\!]_{M_A}(q)$ for every $q \in Q$.*

Proof. By induction on the length of w. ◄

Using Lemma 12, we obtain $\mathrm{val}(A) = \sup_{n \in \omega} [\![\bigvee_{w \in \Sigma^{\le n}} g_w]\!]_{M_A}(q_I)$, where $\Sigma^{\le n}$ is the set of words of length up to n. This can be expressed by using the least fixpoint operator.

► **Theorem 13.** *Let θ_A be the formula of type $\mathit{Prop}_{[0,1]} \to \mathit{Prop}_{[0,1]}$ defined by:*

$$\theta_A := \mu F.\big(\lambda X.X \vee \bigvee_{c \in \Sigma} F\,(f_c\,X)\big).$$

and let $\phi_A := [\theta_A\,\{p_F\}]_{\ge 1}$. Then $\mathit{val}(A) = [\![\theta_A\,\{p_F\}]\!]_{M_A}(q_I)$. Therefore $M_A \models \phi_A$ if and only if $\mathit{val}(A) = 1$.

Proof. Let

$$\xi := \lambda F.\lambda X.X \vee \bigvee_{c\in\Sigma} F\,(f_c\,X).$$

Then, it is easy to verify:

$$[\![\theta_A]\!]_M \quad = \quad [\![\mu F.\xi\,F]\!]_M \quad = \quad \bigvee\nolimits_{\mathit{Prop}_{[0,1]}\to\mathit{Prop}_{[0,1]}}\{[\![\xi^n(\bot)]\!] \mid n\in\omega\}$$

where $\bot := \lambda Z.\mu U.U$ is the formula of type $\mathit{Prop}_{[0,1]} \to \mathit{Prop}_{[0,1]}$, and $\xi^n(x)$ denotes n-times applications of ξ to x.

We have also: $[\![\xi^n(\bot)\,\{p_F\}]\!]_M = [\![\bigvee_{w\in\Sigma^{\leq n}} g_w]\!]_M$. Therefore, we obtain:

$$\mathrm{val}(A) \quad = \quad \sup_n([\![\bigvee_{w\in\Sigma^{\leq n}} g_w]\!]_{M_A}(q_I)) \quad = \quad \sup_n([\![\xi^n(\bot)\{p_F\}]\!](q_I)) \quad = \quad [\![\theta_A\,\{p_F\}]\!]_{M_A}(q_I),$$

which implies the required result. ◀

The following is an immediate corollary of Theorems 11 and 13.

▶ **Corollary 14** (Undecidability of PHFL Model-Checking Problem)**.** *There is no algorithm that, given a Markov chain M and a closed order-1 formula ϕ of type $\mathit{Prop}_{\{0,1\}}$, decides whether $M \models \phi$.*

We close this section with some remarks.[1]

▶ Remark 15. Note that the value val(A) of a probabilistic automaton cannot even be approximately computable [4]: there is no algorithm that outputs "Yes" if val(A) = 1 and "No" if val(A) $\leq \frac{1}{2}$. Thus, the proof of Theorem 13 (in particular, the result val(A) = $[\![\theta_A\,\{p_F\}]\!]_{M_A}(q_I)$) also implies that for a qualitative formula of PHFL ψ, $[\![\psi]\!]$ is not approximately computable in general.

▶ Remark 16. It would be interesting to study a converse encoding, i.e., to find an encoding of some fragment of the PHFL model checking problem into the value-1 problem. Such an encoding may help us find a decidable class of the PHFL model checking problem, based on decidable subclasses for the value-1 problem, such as the one studied in [5].

4 Hardness of the PHFL Model-Checking Problem

In the previous section, we have seen that PHFL model checking is undecidable even for the fragment of PHFL without fixpoint alternations. In this section, we give a lower bound of the hardness of the PHFL model-checking problem in the presence of fixpoint alternations. The following theorem states the main result of this section.

▶ **Theorem 17.** *The order-1 PHFL model-checking problem is Π^1_1-hard and Σ^1_1-hard.*

Note that Π^1_1 and Σ^1_1, defined in terms of the second-order arithmetic, contain very hard problems. For example, the problem of deciding whether a given first-order Peano arithmetic formula is true is in those classes.

We prove this theorem by reducing the validity checking problem of the μ-arithmetic [16] to the PHFL model-checking problem. It is even possible to reduce the validity checking problem of a higher-order extension of the μ-arithmetic to the PHFL model-checking problem. The key in the proof is a representation of natural numbers as quantitative propositions such that all the operations on natural numbers in the μ-arithmetic are expressible in PHFL.

This section is structured as follows. Section 4.1 reviews the basic notions of the μ-arithmetic. Section 4.2 describes the reduction and proves the theorem above.

[1] We would like to thank an anonymous reviewer for pointing them out.

$$\frac{}{\Gamma, X : A \vdash_\mu X : A} \qquad \frac{}{\Gamma \vdash_\mu Z : N} \qquad \frac{\Gamma \vdash_\mu s : N}{\Gamma \vdash_\mu S\, s : N} \qquad \frac{\Gamma \vdash_\mu s, t : N}{\Gamma \vdash_\mu s \leq t : \Omega}$$

$$\frac{\Gamma \vdash_\mu \phi, \psi : \Omega}{\Gamma \vdash_\mu \phi \wedge \psi : \Omega} \qquad \frac{\Gamma \vdash_\mu \phi, \psi : \Omega}{\Gamma \vdash_\mu \phi \vee \psi : \Omega} \qquad \frac{\Gamma, X : A \vdash_\mu \phi : T}{\Gamma \vdash_\mu \lambda X.\phi : A \to T}$$

$$\frac{\Gamma \vdash_\mu \phi : A \to T \qquad \Gamma \vdash_\mu \psi : A}{\Gamma \vdash_\mu \phi\, \psi : T} \qquad \frac{\Gamma, X : T \vdash_\mu \phi : T}{\Gamma \vdash_\mu \mu X.\phi : T} \qquad \frac{\Gamma, X : T \vdash_\mu \phi : T}{\Gamma \vdash_\mu \nu x.\phi : T}$$

Figure 2 Typing Rules for the Higher-order Fixpoint Arithmetic.

4.1 Higher-Order Fixpoint Arithmetic

The μ-arithmetic [16] is a first-order arithmetic with fixpoint operators. This section briefly reviews its higher-order extension, studied by Kobayashi et al. [12].

As in PHFL, we first define the types of μ-arithmetic formulas. The set of *types*, ranged over by A, is given by:

$$A ::= N \mid T \qquad\qquad T ::= \Omega \mid A \to T.$$

The type N is for natural numbers, Ω for (qualitative) propositions, and $A \to T$ for functions. We do not allow functions to return values of type N. We define the order of types of the μ-arithmetic similarly to the PHFL types, by: $\mathit{order}(N) = \mathit{order}(\Omega) = 0$ and $\mathit{order}(A \to T) = \max(\mathit{order}(A) + 1, \mathit{order}(T))$.

Assume a countably infinite set Var of variables ranged over by X. The set of formulas is given by the following grammar.

$$s ::= X \mid Z \mid Ss \qquad \phi ::= X \mid s_1 \leq s_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \lambda X.\phi \mid \phi_1\, \phi_2 \mid \mu X.\phi \mid \nu X.\phi.$$

Here, Z and S respectively denote the constant 0 and the successor function on natural numbers.

The typing rules are shown in Fig. 2. We shall consider only well-typed formulas. We define the *order* of a formula as the largest order of the types of its subformulas.

▶ **Definition 18** (Semantics of Types). *The semantics of a type A is a partially ordered set $[\![A]\!]_\mu = (D_A, \sqsubseteq_A)$ defined inductively on the structure of A as follows.*

1. *The semantics of types N and Ω are defined as follows.*

$$D_N = \mathbb{N} \qquad\qquad n \sqsubseteq_N m \overset{def}{\iff} n = m$$

$$D_\Omega = \{0, 1\} \qquad\qquad p \sqsubseteq_\Omega q \overset{def}{\iff} p \leq q$$

2. *The semantics of the type $A \to T$ is defined as follows.*

$$D_{A\to T} = \{\, f : D_A \to D_T \mid \forall u, v \in D_A. u \sqsubseteq_A v \implies f(u) \sqsubseteq_T f(v)\,\}$$

$$f \sqsubseteq_{A\to T} g \overset{def}{\iff} \forall v \in D_A. f(v) \sqsubseteq_T g(v)$$

The semantics $[\![T]\!]_\mu$ of a type T forms a complete lattice; we write $\bigvee_T$ for the least upper bound operation, and $\bot_T$ for the least element.

The interpretation $[\![\Gamma]\!]_\mu$ of a type environment Γ is the set of functions θ such that $\mathit{dom}(\theta) = \mathit{dom}(\Gamma)$ and that $\theta(X) \in [\![\Gamma(X)]\!]_\mu$ for every $X \in \mathit{dom}(\Gamma)$. It is ordered by the point-wise ordering.

▶ **Definition 19** (Semantics of Formulas). *The semantics of a formula ϕ with judgment $\Gamma \vdash_\mu \phi : A$ is a monotone map from $[\![\Gamma]\!]_\mu$ to $[\![A]\!]_\mu$, defined as follows.*

$$
\begin{aligned}
[\![\Gamma \vdash_\mu X : A]\!]_\mu(\theta) &:= \theta(X)\\
[\![\Gamma \vdash_\mu Z : N]\!]_\mu(\theta) &:= 0\\
[\![\Gamma \vdash_\mu Ss : N]\!]_\mu(\theta) &:= [\![\Gamma \vdash_\mu s : N]\!]_\mu(\theta) + 1\\
[\![\Gamma \vdash_\mu s \le t : \Omega]\!]_\mu(\theta) &:= \begin{cases} 1 & (\textit{if } [\![\Gamma \vdash_\mu s : N]\!]_\mu(\theta) \le [\![\Gamma \vdash_\mu t : N]\!]_\mu(\theta))\\ 0 & (\textit{if } [\![\Gamma \vdash_\mu s : N]\!]_\mu(\theta) > [\![\Gamma \vdash_\mu t : N]\!]_\mu(\theta))\end{cases}\\
[\![\Gamma \vdash_\mu \phi \wedge \psi : \Omega]\!]_\mu(\theta) &:= [\![\Gamma \vdash_\mu \phi : \Omega]\!]_\mu(\theta) \wedge [\![\Gamma \vdash_\mu \phi : \Omega]\!]_\mu(\theta)\\
[\![\Gamma \vdash_\mu \phi \vee \psi : \Omega]\!]_\mu(\theta) &:= [\![\Gamma \vdash_\mu \phi : \Omega]\!]_\mu(\theta) \vee [\![\Gamma \vdash_\mu \phi : \Omega]\!]_\mu(\theta)\\
[\![\Gamma \vdash_\mu \lambda X.\phi : A \to T]\!]_\mu(\theta) &:= \lambda v \in [\![A]\!]_\mu.[\![\Gamma, X : A \vdash_\mu \phi : T]\!]_\mu(\theta[X \mapsto v])\\
[\![\Gamma \vdash_\mu \phi\,\psi : T]\!]_\mu(\theta) &:= [\![\Gamma \vdash_\mu \phi : A \to T]\!]_\mu(\theta)\,([\![\Gamma \vdash_\mu \psi : A]\!]_\mu(\theta))\\
[\![\Gamma \vdash_\mu \mu X.\phi : T]\!]_\mu(\theta) &:= \mathit{LFP}(\lambda v \in D_T.[\![\Gamma, v : T \vdash_\mu \phi : T]\!](\theta[X \mapsto v]))\\
[\![\Gamma \vdash_\mu \nu X.\phi : T]\!]_\mu(\theta) &:= \mathit{GFP}(\lambda v \in D_T.[\![\Gamma, v : T \vdash_\mu \phi : T]\!](\theta[X \mapsto v]))
\end{aligned}
$$

As in the case of PHFL, we write $[\![\phi]\!]_\mu(\theta)$ for $[\![\Gamma \vdash_\mu \phi : A]\!]_\mu(\theta)$ and just $[\![\phi]\!]_\mu$ for $[\![\phi]\!]_\mu(\emptyset)$ when there is no confusion.

▶ **Example 20.** Let $\phi = \mu F.\lambda X.(X = 100 \vee F(S(S\,X)))$ where 100 is an abbreviation of the term $\underbrace{S(S(\dots S}_{100}\ Z)\dots)$. The semantics $[\![\phi]\!]_\mu$ is a function $f : \mathbb{N} \to \{0, 1\}$ where $f(n) = 1$ if and only if n is an even number no greater than 100.

The *validity checking problem* of the higher-order fixpoint arithmetic is the problem of, given a closed formula ϕ of type Ω, deciding whether $[\![\phi]\!]_\mu = 1$. The following result is probably folklore, which follows from the well-known fact that the *fair termination problem for programs* is Π^1_1-complete (see, e.g., Harel [8]), and the fact that the fair termination of a program can be reduced to the validity of a first-order fixpoint arithmetic formula (see, e.g., [12] for the reduction).

▶ **Theorem 21.** *The validity checking problem of the first-order fixpoint arithmetic is Π^1_1-hard and Σ^1_1-hard.*

▶ Remark 22. As for an upper bound, Lubarsky [16] has shown that predicates on natural numbers definable by μ-arithmetic formulas belong to Δ^1_2. One can prove that the validity problem for the μ-arithmetic is Δ^1_2 as well.

4.2 Hardness of PHFL Model Checking

We give a reduction of the validity checking problem of the higher-order fixpoint arithmetic to the PHFL model-checking problem. The main theorem of this section (Theorem 17) is an immediate consequence of this reduction and Theorem 21.

Given a formula ϕ of the higher-order fixpoint arithmetic, we need to effectively construct a pair (ψ, M) of a formula of PHFL and a Markov chain such that ϕ is true if and only if $M \models \psi$. The Markov chain M is independent of the formula ϕ. We first define the Markov chain and then explain the intuition of the translation of formulas.

The Markov chain $M = (S, P, \rho_{AP}, s_{\text{in}})$ is shown in Figure 3. It is defined as follows.

- The set of states is $S = \{s_0, s'_0, s_1, s'_1\}$.

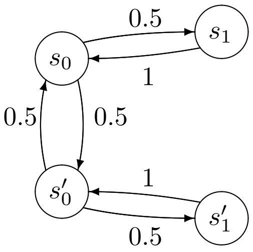


Figure 3 The Markov Chain for Reduction from Higher-order Fixpoint Arithmetic to PHFL.

- The transition probability satisfies $P(s_0, s_1) = P(s_0, s_0') = P(s_0', s_0) = P(s_0', s_1') = \frac{1}{2}$, $P(s_1, s_0) = P(s_1', s_0') = 1$ and $P(s_i, s_j) = 0$ for all other pairs of states.
- There are four atomic propositions p_0, p_0', p_1, and p_1', representing each state (e.g. $\rho_{AP}(p_0) = \{s_0\}$).
- The initial state s_{in} is s_0.

For notational convenience, we write $v \in [\![Prop_{[0,1]}]\!]_M$ as a tuple $(v(s_0), v(s_0'), v(s_1), v(s_1'))$.

As mentioned at the beginning of this section, the key of the reduction is the representation of natural numbers, as well as operations on natural numbers. We represent a natural number n by a quantitative propositional formula ψ such that $[\![\psi]\!]_M = (\frac{1}{2^n}, 1 - \frac{1}{2^n}, \_, \_)$. Here, $\_$ denotes a "don't care" value. We implement primitives on natural numbers Z, S and $\leq$, as follows.

The constant Z can be represented by $\{p_0\}$: then $[\![\{p_0\}]\!]_M = (1, 0, 0, 0) = (1/2^0, 1 - (1/2^0), 0, 0)$ as expected.

Assuming that ψ represents n (i.e. $[\![\psi]\!]_M = (1/2^n, 1 - (1/2^n), \_, \_)$), the successor $n + 1$ can be given by

$$\psi' \quad := \quad \bigcirc((\bigcirc\psi \wedge (p_1 \vee p_1')) \vee p_0).$$

Indeed, we have:

$$\begin{aligned}
[\![\bigcirc\psi]\!]_M &= (\_, \_, \frac{1}{2^n}, 1 - \frac{1}{2^n}) \\
[\![\bigcirc\psi \wedge (p_1 \vee p_1')]\!]_M &= (0, 0, \frac{1}{2^n}, 1 - \frac{1}{2^n}) \\
[\![(\bigcirc\psi \wedge (p_1 \vee p_1')) \vee p_0]\!]_M &= (1, 0, \frac{1}{2^n}, 1 - \frac{1}{2^n}) \\
[\![\bigcirc((\bigcirc\psi \wedge (p_1 \vee p_1')) \vee p_0)]\!]_M &= (\frac{1}{2} \times \frac{1}{2^n}, \frac{1}{2} + \frac{1}{2} \times (1 - \frac{1}{2^n}), \_, \_) \\
&= (\frac{1}{2^{n+1}}, 1 - \frac{1}{2^{n+1}}, \_, \_).
\end{aligned}$$

It remains to encode $\leq$. We use the fact that, for any natural numbers n and m,

$$n \leq m \quad \Leftrightarrow \quad \frac{1}{2^n} \geq \frac{1}{2^m} \quad \Leftrightarrow \quad \frac{1}{2^n} + (1 - \frac{1}{2^m}) \geq 1.$$

The s_0'-component of the representation of a natural number plays an important role below. Assume that ψ and χ represent n and m respectively. Then we have

$$[\![\bigcirc\psi \wedge p_1]\!]_M = (0, 0, \frac{1}{2^n}, 0) \qquad [\![\chi \wedge p_0']\!]_M = (0, 1 - \frac{1}{2^m}, 0, 0)$$

and thus

$$[\![(\bigcirc\psi \wedge p_1) \vee (\chi \wedge p'_0)]\!]_M = (0, 1 - \frac{1}{2^m}, \frac{1}{2^n}, 0).$$

Therefore

$$[\![\bigcirc((\bigcirc\psi \wedge p_1) \vee (\chi \wedge p'_0))]\!]_M = (\frac{1}{2} \times (\frac{1}{2^n} + (1 - \frac{1}{2^m})), \_, \_, \_).$$

Therefore, $n \leq m$ if and only if the s_0-component of the above formula is $\geq \frac{1}{2}$.

Let us formalize the above argument. We first give the translation of types:

$$tr(N) = Prop_{[0,1]} \qquad tr(\Omega) = Prop_{\{0,1\}} \qquad tr(A \to T) = tr(A) \to tr(T).$$

The translation can be naturally extended to type environments. Following the above discussion, the translation of formulas of type N is given by

$$tr(Z) = \{p_0\} \qquad \text{and} \qquad tr(S\,s) = \bigcirc((\bigcirc tr(s) \wedge (p_1 \vee p'_1)) \vee p_0).$$

The comparison operator can be translated as follows:

$$tr(s \leq t) = [(\bigcirc((\bigcirc tr(s) \wedge p_1) \vee (tr(t) \wedge p'_0)))]_{\geq \frac{1}{2}}.$$

The translation of other connectives is straightforward:

$$tr(\phi \wedge \psi) = tr(\phi) \wedge tr(\psi) \qquad tr(\phi \vee \psi) = tr(\phi) \vee tr(\psi) \qquad tr(\lambda X.\phi) = \lambda X.tr(\phi)$$
$$tr(X) = X \qquad tr(\phi\,\psi) = tr(\phi)\,tr(\psi) \qquad tr(\mu X.\phi) = \mu X.tr(\phi) \qquad tr(\nu X.\phi) = \nu X.tr(\phi).$$

The following lemma states that the translation preserves types.

▶ **Lemma 23.** *If $\Gamma \vdash_\mu \phi : A$, then $tr(\Gamma) \vdash tr(\phi) : tr(A)$.*

We prove the correctness of the translation. For each type A of the higher-order fixpoint arithmetic, we define a relation $(\sim_A) \subseteq [\![A]\!]_\mu \times [\![tr(A)]\!]_M$ by induction on A as follows:

$$\begin{aligned}
n \sim_N (r_0, r'_0, r_1, r'_1) \quad &\overset{\text{def}}{\iff} \quad r_0 = \frac{1}{2^n} \text{ and } r'_0 = 1 - \frac{1}{2^n} \\
b \sim_\Omega (r_0, r'_0, r_1, r'_1) \quad &\overset{\text{def}}{\iff} \quad b = r_0 \\
f \sim_{A \to T} g \quad &\overset{\text{def}}{\iff} \quad \forall x \in [\![A]\!]_\mu. \forall y \in [\![tr(A)]\!]_M.\ x \sim_A y \implies f\,x \sim_T g\,y.
\end{aligned}$$

This relation can be naturally extended to the interpretations of type environments: given a type environment Γ of the μ-arithmetic, the relation $(\sim_\Gamma) \subseteq [\![\Gamma]\!]_\mu \times [\![tr(\Gamma)]\!]_M$ is defined by

$$\theta \sim_\Gamma \rho \quad \overset{\text{def}}{\iff} \quad \forall X \in dom(\Gamma).\ \theta(X) \sim_{\Gamma(X)} \rho(X).$$

▶ **Theorem 24.** *Let $\Gamma \vdash_\mu \phi : A$ be a formula of the higher-order fixpoint arithmetic. Assume $\theta \in [\![\Gamma]\!]_\mu$ and $\rho \in [\![tr(\Gamma)]\!]$. If $\theta \sim_\Gamma \rho$, then $[\![\Gamma \vdash_\mu \phi : A]\!]_\mu(\theta) \sim_A [\![tr(\Gamma) \vdash tr(\phi) : tr(A)]\!]_M(\rho)$.*

Proof. See Appendix A. ◀

▶ **Corollary 25.** *The validity problem of the order-k fixpoint arithmetic (where $k > 0$) is reducible to the order-k PHFL model-checking problem.*

Proof. Assume $\emptyset \vdash_\mu \phi : \Omega$. By Theorem 24, $[\![\phi]\!]_\mu \sim_\Omega [\![tr(\phi)]\!]_M$. Therefore, $[\![\phi]\!]_\mu = 1$ if and only if $[\![tr(\phi)]\!]_M(s_0) = 1$, i.e. $M \models tr(\phi)$. The mapping $\phi \mapsto (tr(\phi), M)$ is obviously effective, and preserves the order. ◀

Theorem 17 is an immediate consequence of Theorem 21 and Corollary 25.

5 Decidable Subclass of Order-1 PHFL Model Checking

As we have seen in the last section, PHFL model checking is undecidable in general, even for order 1. In this section, we identify a decidable subclass of the order-1 PHFL model-checking problems (i.e., a set of pairs (ϕ, M) such that whether $M \models \phi$ is decidable). We identify the subclass by using a type system: we define a type system $\mathcal{T}_M$ for PHFL formulas, parameterized by M, such that if ϕ is a proposition well-typed in $\mathcal{T}_M$, then $M \models \phi$ is decidable.

We first explain the idea of the restriction imposed by the type system. By definition, the semantics of a (closed) order-1 PHFL formula ϕ of type $\mathit{Prop}_{[0,1]} \to \mathit{Prop}_{[0,1]}$ with respect to the Markov chain M is a map f_ϕ from the set of functions $S \to [0,1]$ to the same set, where S is the set of states of M. Thus, if $S = \{s_1, s_2, \ldots, s_n\}$ is fixed, f_ϕ can be regarded as a function from $[0,1]^n$ to $[0,1]^n$. Now, if the function f_ϕ were affine, i.e., if there are functions $f_1, f_2, \ldots, f_n$ such that $f_\phi(r_1, r_2, \ldots, r_k) = (f_1(r_1, r_2, \ldots, r_k), \ldots, f_n(r_1, r_2, \ldots, r_k))$, where $f_i(r_1, r_2, \ldots, r_k) = c_{i,0} + c_{i,1} r_1 + \cdots + c_{i,k} r_k$ for some real numbers $c_{i,j}$, then the function f_ϕ would be representable by a finite number of reals $c_{i,j}$. The semantics of an (alternation-free) fixpoint formula would then be given as a solution of a fixpoint equation on the coefficients, which is solvable by appealing to the existential theories of reals.

Based on the observation above, we use a type system to restrict the formulas so that the semantics of every order-1 formula is affine. The conjunction $\phi_1 \wedge \phi_2$ is one of the problematic logical connectives that may make the semantics of an order-1 formula non-affine: recall that the min operator was used to define the semantics of conjunction. We require that for every subformula of the form $\phi_1 \wedge \phi_2$ and for each state $s \in S$, one of the values $[\![\phi_1]\!](s)$ and $[\![\phi_2]\!](s)$ is the constant 0 or 1. We can then remove the min operator, since we have $\min(0, x) = 0$ and $\min(1, x) = x$ for every $x \in [0,1]$.

The discussion above motivates us to refine the type $\mathit{Prop}_{[0,1]}$ of quantitative propositions to $\mathit{Prop}^{T,U}$ where $T, U \subseteq S$ and $T \cap U = \emptyset$. Intuitively, the type $\mathit{Prop}^{T,U}$ is a type for values $v \in \mathit{Prop}_{[0,1]}$ such that $v(s) = 0$ for all $s \in T$ and $v(s) = 1$ for all $s \in U$; there is no guarantee on the value of $v(s)$ for $s \in S \setminus (T \cup U)$. The syntax of refined types is given by:

$$\sigma ::= \kappa \mid \mathit{Prop}_{\{0,1\}} \qquad\qquad \kappa ::= \mathit{Prop}^{T,U} \mid \mathit{Prop}^{T,U} \to \kappa$$

where T and U range over the subsets of S satisfying $T \cap U = \emptyset$. Note that each type $\kappa \neq \mathit{Prop}_{\{0,1\}}$ can be expressed as $\mathit{Prop}^{T_1,U_1} \to \mathit{Prop}^{T_2,U_2} \to \cdots \to \mathit{Prop}^{T_k,U_k} \to \mathit{Prop}^{T,U}$ where $k \geq 0$. The formal definition of the semantics of types is given later.

We restrict PHFL formulas to those given by:

$$\psi ::= [\phi]_J \qquad\qquad \phi ::= \{p\} \mid x \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \bigcirc \phi \mid \mu x.\phi \mid \lambda x.\phi \mid \phi_1\, \phi_2$$

and further restrict them by using the typing rules in Figure 4. In the figure, the type environment $\mathcal{K}$ maps each variable to a type in the set ranged over by κ. The operator $[\cdot]$ has been restricted to the top-level, and the operators $\Diamond, \Box$ and ν have been removed. Note that ψ is a qualitative formula and ϕ is a quantitative formula.

A key rule is for conjunctions. Note that $[\![\phi_1 \wedge \phi_2]\!](s) = 0$ if either $[\![\phi_1]\!](s) = 0$ or $[\![\phi_2]\!](s) = 0$ holds; hence $s \in T_1 \cup T_2$ implies $[\![\phi_1 \wedge \phi_2]\!](s) = 0$. Note also that $[\![\phi_1 \wedge \phi_2]\!](s) = 1$ if both $[\![\phi_1]\!](s) = 1$ *and* $[\![\phi_2]\!](s) = 1$ hold. Thus, $s \in U_1 \cap U_2$ implies $[\![\phi_1 \wedge \phi_2]\!](s) = 1$. This is why $\phi_1 \wedge \phi_2$ has type $\mathit{Prop}^{T_1 \cup T_2, U_1 \cap U_2}$. The extra condition $T_1 \cup U_1 \cup T_2 \cup U_2 = S$ requires that, for each state s, either $[\![\phi_1]\!](s)$ or $[\![\phi_2]\!](s)$ is the constant 0 or 1; recall the earlier discussion on a sufficient condition for the semantics of an order-1 formula to be affine. The rule for disjunctions is analogous.

The following lemma states that a formula that is well-typed in $\mathcal{T}_M$ is also well-typed in the original PHFL type system.

$$\frac{}{\mathcal{K} \vdash_M \{p\} : \mathit{Prop}^{\overline{\rho_{AP}(p)},\rho_{AP}(p)}} \qquad \frac{\mathcal{K} \vdash_M \phi : \mathit{Prop}^{T,U} \qquad T' \subseteq T \quad U' \subseteq U}{\mathcal{K} \vdash_M \phi : \mathit{Prop}^{T',U'}}$$

$$\frac{\mathcal{K} \vdash_M \phi_1 : \mathit{Prop}^{T_1,U_1} \qquad \mathcal{K} \vdash_M \phi_2 : \mathit{Prop}^{T_2,U_2} \qquad T_1 \cup U_1 \cup T_2 \cup U_2 = S}{\mathcal{K} \vdash_M \phi_1 \wedge \phi_2 : \mathit{Prop}^{(T_1 \cup T_2),(U_1 \cap U_2)}}$$

$$\frac{\mathcal{K} \vdash_M \phi_1 : \mathit{Prop}^{T_1,U_1} \qquad \mathcal{K} \vdash_M \phi_2 : \mathit{Prop}^{T_2,U_2} \qquad T_1 \cup U_1 \cup T_2 \cup U_2 = S}{\mathcal{K} \vdash_M \phi_1 \vee \phi_2 : \mathit{Prop}^{(T_1 \cap T_2),(U_1 \cup U_2)}}$$

$$\frac{}{\mathcal{K}, X : \kappa \vdash_M X : \kappa} \qquad \frac{\mathcal{K} \vdash_M \phi : \mathit{Prop}^{T,U}}{\mathcal{K} \vdash_M [\phi]_J : \mathit{Prop}_{\{0,1\}}} \qquad \frac{\mathcal{K} \vdash_M \phi : \mathit{Prop}^{T,U}}{\mathcal{K} \vdash_M \bigcirc\phi : \mathit{Prop}^{\emptyset,\emptyset}}$$

$$\frac{\mathcal{K}, X : \kappa \vdash_M \phi : \kappa}{\mathcal{K} \vdash_M \mu X.\phi : \kappa} \qquad \frac{\mathcal{K}, X : \kappa_1 \vdash_M \phi : \kappa_2}{\mathcal{K} \vdash_M \lambda X.\phi : \kappa_1 \to \kappa_2}$$

$$\frac{\mathcal{K} \vdash_M \phi_0 : \mathit{Prop}^{T_1,U_1} \to \cdots \to \mathit{Prop}^{T_k,U_k} \to \mathit{Prop}^{T,U} \qquad \mathcal{K} \vdash_M \phi_i : \mathit{Prop}^{T_i,U_i}\ (1 \le i \le k)}{\mathcal{K} \vdash_M \phi_0\, \phi_1\, \ldots\, \phi_k : \mathit{Prop}^{T,U}}$$

Figure 4 Type Derivation Rules for the PHFL Subclass. Here $\overline{X}$ means the complement $S \setminus X$.

▶ **Lemma 26.** *Let ϕ be a PHFL formula such that $\mathcal{K} \vdash_M \phi : \kappa$ in $\mathcal{T}_M$. Define the translation from the set of types in $\mathcal{T}_M$ to the set of types in PHFL by*

$$\mathit{tr}(\mathit{Prop}_{\{0,1\}}) = \mathit{Prop}_{\{0,1\}} \qquad \mathit{tr}(\mathit{Prop}^{T,U}) = \mathit{Prop}_{[0,1]} \qquad \mathit{tr}(\kappa_1 \to \kappa_2) = \mathit{tr}(\kappa_1) \to \mathit{tr}(\kappa_2)$$

and the translation of type environment $\mathcal{K}$ by $(\mathit{tr}(\mathcal{K}))(x) = \mathit{tr}(\mathcal{K}(x))$. Then we have $\mathit{tr}(\mathcal{K}) \vdash \phi : \mathit{tr}(\kappa)$.

The lemma above can be proved by induction on the structure of ϕ. Using the lemma, we can define the semantics of a type judgment of the type system $\mathcal{T}_M$ by $[\![\mathcal{K} \vdash_M \phi : \kappa]\!]_M = [\![\mathit{tr}(\mathcal{K}) \vdash \phi : \mathit{tr}(\kappa)]\!]_M$. As before, we often omit the type environment, the derived type and the subscript of the Markov chain in the notation of the semantics.

▶ **Example 27.** Let $p_1, p_2, p_3 \in AP$ be atomic propositions satisfying $\rho_{AP}(p_2) \cap \rho_{AP}(p_3) = \emptyset$. Consider the formula $\phi = \bigcirc((\{p_2\} \wedge \bigcirc\{p_1\}) \vee (\{p_3\} \wedge \bigcirc\{p_1\}))$. For each $s \in S$, the value $[\![\phi]\!](s)$ represents the probability that a two-step transition starting from s reaches a state satisfying p_1 through a state satisfying p_2 or p_3. We can derive $\emptyset \vdash_M \phi : \mathit{Prop}^{\emptyset,\emptyset}$ as follows. First, $\{p_1\}$, $\{p_2\}$, and $\{p_3\}$ have types $\mathit{Prop}^{\overline{\rho_{AP}(p_1)},\rho_{AP}(p_1)}$, $\mathit{Prop}^{\overline{\rho_{AP}(p_2)},\rho_{AP}(p_2)}$, and $\mathit{Prop}^{\overline{\rho_{AP}(p_3)},\rho_{AP}(p_3)}$. It follows that $\{p_2\} \wedge \bigcirc\{p_1\}$ and $\{p_3\} \wedge \bigcirc\{p_1\}$ have types $\mathit{Prop}^{\overline{\rho_{AP}(p_2)},\emptyset)}$ and $\mathit{Prop}^{\overline{\rho_{AP}(p_3)},\emptyset}$. Since $\overline{\rho_{AP}(p_2)} \cup \overline{\rho_{AP}(p_3)} = \overline{\rho_{AP}(p_2) \cap \rho_{AP}(p_3)} = \overline{\emptyset} = S$, the formula $(\{p_2\} \wedge \bigcirc\{p_1\}) \vee (\{p_3\} \wedge \bigcirc\{p_1\})$ has type $\mathit{Prop}^{\overline{\rho_{AP}(p_2)} \cap \overline{\rho_{AP}(p_3)},\emptyset}$, from which we obtain $\emptyset \vdash_M \phi : \mathit{Prop}^{\emptyset,\emptyset}$. Note that the condition $L(p_2) \cap L(p_3) = \emptyset$ was crucial in the type derivation above. ◀

We have the following two theorems. The former one states the decidability result, and the latter one states that the restricted subclass of the PHFL model-checking problems is reasonably expressive. Proofs are found in Appendix B.

▶ **Theorem 28.** *Let M be a Markov chain, and ψ be a PHFL formula satisfying $\vdash_M \psi : \mathit{Prop}_{\{0,1\}}$. Then it is decidable whether $M \models \psi$.*

▶ **Theorem 29.** *There exists an algorithm that takes a recursive Markov chain R and a rational number r as input, and outputs an order-1 PHFL formula ϕ_R and a Markov chain M_R such that $\vdash_{M_R} [\phi_R]_{\geq r} : Prop_{\{0,1\}}$, and the termination probability of R is no less than r if and only if $M_R \models [\phi_R]_{\geq r}$.*

6 Related Work

As mentioned in Section 1, PHFL can be regarded as a probabilistic extension of the higher-order fixpoint logic, and as a higher-order extension of the μ^p-calculus. We thus compare our work with previous studies on (non-probabilistic) higher-order fixpoint logic and those on (non-higher-order) probabilistic logics. As already mentioned, for (non-probabilistic) HFL, model checking of finite-state systems is known to be decidable [22], and k-EXPTIME complete [1]. This is in a sharp contrast with our result that PHFL model checking is highly undecidable (both Π^1_1-hard and Σ^1_1-hard) even at order 1. As for studies on probabilistic logics, besides the μ^p-calculus, there are other probabilistic extensions of the modal μ-calculus [19, 9, 17]. To our knowledge, however, ours is the first *higher-order* and probabilistic extension of the modal μ-calculus.

Recently, Kobayashi et al. [10] introduced PHORS, a probabilistic extension of higher-order recursion schemes (HORS), which can also be viewed as a higher-order extension of recursive Markov chains (or probabilistic pushdown systems), and proved that the almost sure termination problem is undecidable. Although the problem setting is quite different (in our work, the *logic* is higher-order whereas the *system* to be verified is higher-order in their work), our encoding of the μ-arithmetic has been partially inspired by their undecidability proof; they also represented a natural number n as the probability $\frac{1}{2^n}$.

7 Conclusion

We have introduced PHFL, a probabilistic logic which can be regarded as both a probabilistic extension of HFL and a higher-order extension of the probabilistic logic μ^p-calculus. We have shown that the model-checking problem for PHFL for a finite Markov chain is undecidable for the μ-only and order-1 fragment. We have also shown that the model-checking problem for the full order-1 fragment of PHFL is Π^1_1-hard and Σ^1_1-hard. As positive results, we have introduced a decidable subclass of the PHFL model-checking problems, and showed that the termination problem of Recursive Markov Chains can be encoded in the subclass.

Finding an upper bound of the hardness of the PHFL model-checking problem is left for future work. It is also left for future work to find a larger, more natural decidable class of PHFL model-checking problems.

References

1 Roland Axelsson, Martin Lange, and Rafal Somla. The complexity of model checking higher-order fixpoint logic. *Logical Methods in Computer Science*, 3(2), 2007. `doi:10.2168/LMCS-3(2:7)2007`.

2 Pablo F. Castro, Cecilia Kilmurray, and Nir Piterman. Tractable probabilistic mu-calculus that expresses probabilistic temporal logics. In Ernst W. Mayr and Nicolas Ollinger, editors, *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany*, volume 30 of *LIPIcs*, pages 211–223. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. `doi:10.4230/LIPIcs.STACS.2015.211`.

3 Kousha Etessami and Mihalis Yannakakis. Recursive markov chains, stochastic grammars, and monotone systems of nonlinear equations. *J. ACM*, 56(1):1:1–1:66, 2009. doi:10.1145/1462153.1462154.

4 Nathanaël Fijalkow. Undecidability results for probabilistic automata. *SIGLOG News*, 4(4):10–17, 2017. URL: https://dl.acm.org/citation.cfm?id=3157833.

5 Nathanaël Fijalkow, Hugo Gimbert, Edon Kelmendi, and Youssouf Oualhadj. Deciding the value 1 problem for probabilistic leaktight automata. *Logical Methods in Computer Science*, 11(2), 2015. doi:10.2168/LMCS-11(2:12)2015.

6 Hugo Gimbert and Youssouf Oualhadj. Probabilistic automata on finite words: Decidable and undecidable problems. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part II*, volume 6199 of *Lecture Notes in Computer Science*, pages 527–538. Springer, 2010. doi:10.1007/978-3-642-14162-1_44.

7 Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Asp. Comput.*, 6(5):512–535, 1994. doi:10.1007/BF01211866.

8 David Harel. Effective transformations on infinite trees, with applications to high undecidability, dominoes, and fairness. *J. ACM*, 33(1):224–248, 1986. doi:10.1145/4904.4993.

9 Michael Huth and Marta Z. Kwiatkowska. Quantitative analysis and model checking. In *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 - July 2, 1997*, pages 111–122. IEEE Computer Society, 1997. doi:10.1109/LICS.1997.614940.

10 Naoki Kobayashi, Ugo Dal Lago, and Charles Grellois. On the termination problem for probabilistic higher-order recursive programs. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, pages 1–14. IEEE, 2019. doi:10.1109/LICS.2019.8785679.

11 Naoki Kobayashi, Étienne Lozes, and Florian Bruse. On the relationship between higher-order recursion schemes and higher-order fixpoint 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 246–259. ACM, 2017. URL: http://dl.acm.org/citation.cfm?id=3009854.

12 Naoki Kobayashi, Takeshi Tsukada, and Keiichi Watanabe. Higher-order program verification via HFL model checking. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 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 10801 of *Lecture Notes in Computer Science*, pages 711–738. Springer, 2018. doi:10.1007/978-3-319-89884-1_25.

13 Dexter Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983. doi:10.1016/0304-3975(82)90125-6.

14 Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011. doi:10.1007/978-3-642-22110-1_47.

15 Étienne Lozes. A type-directed negation elimination. In Ralph Matthes and Matteo Mio, editors, *Proceedings Tenth International Workshop on Fixed Points in Computer Science, FICS 2015, Berlin, Germany, September 11-12, 2015*, volume 191 of *EPTCS*, pages 132–142, 2015. doi:10.4204/EPTCS.191.12.

16 Robert S. Lubarsky. mu-definable sets of integers. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 343–352. IEEE Computer Society, 1989. doi:10.1109/LICS.1989.39189.

17 Matteo Mio and Alex Simpson. Łukasiewicz mu-calculus. In David Baelde and Arnaud Carayol, editors, *Proceedings Workshop on Fixed Points in Computer Science, FICS 2013, Turino, Italy, September 1st, 2013*, volume 126 of *EPTCS*, pages 87–104, 2013. doi:10.4204/EPTCS.126.7.

18 Yo Mitani, Naoki Kobayashi, and Takeshi Tsukada. A probabilistic higher-order fixpoint logic, 2020. A full version, available from http://www.kb.is.s.u-tokyo.ac.jp/~mitaniyo/papers/fscd2020-long.pdf.

19 Carroll Morgan and Annabelle McIver. A probabilistic temporal calculus based on expectations. In *Proc. Formal Methods Pacific*, pages 4–22. Springer, 1997.

20 Azaria Paz. *Introduction to probabilistic automata*. Academic Press, 1971.

21 Michael O Rabin. Probabilistic automata. *Information and control*, 6(3):230–245, 1963.

22 Mahesh Viswanathan and Ramesh Viswanathan. A higher order modal fixed point logic. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, volume 3170 of *Lecture Notes in Computer Science*, pages 512–528. Springer, 2004. doi:10.1007/978-3-540-28644-8_33.

Appendix

A Proof of Theorem 24

We prove the theorem by induction on the structure of ϕ. In this proof, we omit the subscript M of $[\![-]\!]_M$ for simplicity. We discuss only the main cases; see [18] for more details.

- Case $\phi = X$.
  We have $tr(\phi) = X$ and $[\![\phi]\!]_\mu(\theta) = \theta(X) \sim_{\Gamma(X)} \rho(X) = [\![tr(\phi)]\!](\rho)$.
- Case $\phi = Z$.
  Then $tr(\phi) = \{p'_0\}$ and $A = N$. We have $[\![\phi]\!]_\mu(\theta) = 0 \sim_N (1,0,0,0) = [\![tr(\phi)]\!](\rho)$.
- Case $\phi = S\,t$.
  Let $n = [\![t]\!]_\mu(\theta)$. By the induction hypothesis, we have

$$[\![tr(t)]\!](\rho) = \left(\frac{1}{2^n},\ 1 - \frac{1}{2^n},\ \_,\ \_\right).$$

  By the definition of $tr(\phi)$ and calculation, we have

$$[\![tr(\phi)]\!](\rho) = \left(\frac{1}{2^{n+1}},\ 1 - \frac{1}{2^{n+1}},\ \_,\ \_\right),$$

  which implies $[\![S\,t]\!]_\mu(\theta) = n+1 \sim_N [\![tr(\phi)]\!](\rho)$.
- Case $\phi = (s \le t)$.
  Let $n = [\![s]\!]_\mu(\theta)$ and $m = [\![t]\!]_\mu(\theta)$. By the induction hypothesis, we have

$$[\![tr(s)]\!](\rho) = (\frac{1}{2^n},\ ,1 - \frac{1}{2^n},\ \_,\ \_)$$
$$[\![tr(t)]\!](\rho) = (\frac{1}{2^m},\ ,1 - \frac{1}{2^m},\ \_,\ \_).$$

  By the definition of $tr(s \le t)$ and calculation, we have

$$[\![tr(s \le t)]\!](\rho) = \begin{cases} (1,\_,\_,\_) & (\text{if } \frac{1}{2} \times \left(\frac{1}{2^n} + (1 - \frac{1}{2^m})\right) \ge \frac{1}{2}, \text{ i.e. if } n \le m) \\ (0,\_,\_,\_) & (\text{if } \frac{1}{2} \times \left(\frac{1}{2^n} + (1 - \frac{1}{2^m})\right) < \frac{1}{2}, \text{ i.e., if } n > m). \end{cases}$$

  Thus, we have $[\![s \le t]\!]_\mu(\theta) \sim_\Omega [\![tr(s \le t)]\!](\rho)$ as required.

- Case $\phi = \lambda X.\psi$. In this case, A is of the form $B \to T$, with $\Gamma, X : B \vdash_\mu \psi : T$. By the induction hypothesis, ψ satisfies

  $$[\![\Gamma, X : B \vdash_\mu \psi : T]\!]_\mu(\theta[X \mapsto v]) \sim_{B\to T} [\![\mathit{tr}(\Gamma, X : B) \vdash \mathit{tr}(\psi) : \mathit{tr}(T)]\!](\rho[X \mapsto u])$$

  for any $v \in [\![B]\!]$ and $u \in [\![\mathit{tr}(B)]\!]$ such that $v \sim_B u$.
  Therefore, by the definition of $\sim_{B\to T}$, we have

  $$[\![\Gamma \vdash_\mu \phi : B \to T]\!]_\mu(\theta) \sim_{B\to T} [\![\mathit{tr}(\Gamma) \vdash \mathit{tr}(\phi) : \mathit{tr}(B \to T)]\!](\rho)$$

  as required.
- Case $\phi = \psi_1\, \psi_2$. We have $A = T$, with $\Gamma \vdash_\mu \psi_1 : B \to T$ and $\Gamma \vdash_\mu \psi_2 : B$.
  By the induction hypothesis, we have $[\![\psi_1]\!]_\mu(\theta) \sim_{B\to T} [\![\mathit{tr}(\psi_1)]\!](\rho)$ and $[\![\psi_2]\!]_\mu(\theta) \sim_B [\![\mathit{tr}(\psi_2)]\!](\rho)$. Therefore by the definition of $\sim_{B\to T}$, we have

  $$\begin{aligned}[\![\psi_1\, \psi_2]\!]_\mu(\theta) &= [\![\psi_1]\!]_\mu(\theta)\,([\![\psi_2]\!]_\mu(\theta))\\ &\sim_A [\![\mathit{tr}(\psi_1)]\!](\rho)\,([\![\mathit{tr}(\psi_2)]\!](\rho))\\ &= [\![\mathit{tr}(\psi_1\, \psi_2)]\!](\rho)\end{aligned}$$

  as desired.
- Case $\phi = \mu X.\psi$.
  In this case, $A = T$, with $\Gamma, X{:}T \vdash_\mu \psi : T$. By the induction hypothesis, for any $v \in [\![T]\!]_\mu$ and $u \in [\![\mathit{tr}(T)]\!]$ such that $v \sim_T u$, we have

  $$[\![\psi]\!]_\mu(\theta[X \mapsto v]) \sim_T [\![\mathit{tr}(\psi)]\!](\rho[X \mapsto u]).$$

  Since $\mathit{tr}(\mu X.\psi) = \mu X.\mathit{tr}(\psi)$, it suffices to show:

  $$[\![\mu X.\psi]\!]_\mu(\theta) \sim_T [\![\mu X.\mathit{tr}(\psi)]\!](\rho).$$

  Let $\mathcal{F} : [\![T]\!]_\mu \to [\![T]\!]_\mu$ and $\mathcal{G} : [\![\mathit{tr}(T)]\!] \to [\![\mathit{tr}(T)]\!]$ be the functions defined by:

  $$\mathcal{F}(v) := [\![\psi]\!]_\mu(\theta[X \mapsto v]) \qquad \mathcal{G}(u) := [\![\mathit{tr}(\psi)]\!](\rho[X \mapsto u]).$$

  By the reasoning above, we have $\mathcal{F} \sim_{T\to T} \mathcal{G}$. By the definitions of the semantics, we have $[\![\mu X.\psi]\!]_\mu(\theta) = \mathrm{LFP}(\mathcal{F})$ and $[\![\mu X.\psi]\!](\rho) = \mathrm{LFP}(\mathcal{G})$. Then there exists an ordinal α such that

  $$\mathrm{LFP}(\mathcal{F}) = \mathcal{F}^\alpha(\bot_T) \qquad \text{and} \qquad \mathrm{LFP}(\mathcal{G}) = \mathcal{G}^\alpha(\bot_{\mathit{tr}(T)}),$$

  where $f^\beta(x)$ is defined by $f^0(x) = x$, $f^{\beta+1} = f(f^\beta(x))$, and $f^\beta = \bigvee_{\gamma<\beta} f^\gamma(x)$ if β is a limit ordinal. We shall prove by (transfinite) induction on β that $\mathcal{F}^\beta(\bot_T) \sim_T \mathcal{G}^\beta(\bot_{\mathit{tr}(T)})$, which would imply

  $$\mathrm{LFP}(\mathcal{F}) = \mathcal{F}^\alpha(\bot_T) \sim_T \mathcal{G}^\alpha(\bot_{\mathit{tr}(T)}) = \mathrm{LFP}(\mathcal{G})$$

  as required.
  The base case $\mathcal{F}^0(\bot_T) = \bot_T \sim_T \bot_{\mathit{tr}(T)} = \mathcal{G}^0(\bot_{\mathit{tr}(T)})$ follows by a straightforward induction on the structure of T. The case where β is a successor ordinal follows immediately from the induction hypothesis and $\mathcal{F} \sim_{T\to T} \mathcal{G}$. If β is a limit ordinal, then

  $$\mathcal{F}^\beta(\bot_T) = \bigvee\nolimits_{\gamma<\beta} \mathcal{F}^\gamma(\bot_T) \qquad \text{and} \qquad \mathcal{G}^\beta(\bot_T) = \bigvee\nolimits_{\gamma<\beta} \mathcal{G}^\gamma(\bot_T).$$

By the induction hypothesis (of the transfinite induction),

$$\mathcal{F}^{\gamma}(\bot_T) \sim_T \mathcal{G}^{\gamma}(\bot_T)$$

for every $\gamma < \beta$. Since $\sim_T$ is preserved by the least upper-bound operation (which can be proved by an easy induction on T), we have

$$\mathcal{F}^{\beta}(\bot_T) \sim_T \mathcal{G}^{\beta}(\bot_T)$$

as required.

- Case $\phi = \nu X.\psi$. Similar to the case for $\phi = \mu X.\psi$ above.

B Proofs for Section 5

B.1 Proof of Theorem 28

We first give a matrix representation for each value of the semantics of types of $\mathcal{T}_M$. As mentioned before, we fix the underlying Markov chain M with the set of states $S = \{s_1, s_2, \dots, s_n\}$. Henceforth, we identify the set of functions $S \to [0,1]$ with the set $[0,1]^n$.

We first give the formal definition of the semantics of types in $\mathcal{T}_M$. As explained in Section 5, the values of function types are restricted to affine functions.

▶ **Definition 30.** *For each type $\kappa \neq Prop_{\{0,1\}}$ in the type system $\mathcal{T}_M$, we define its semantics $[\![\kappa]\!] = (D_\kappa, \sqsubseteq_\kappa)$ by induction on κ as follows.*

1. *For $\kappa = Prop^{T,U}$, D_κ is the set $\{v \in [\![Prop_{[0,1]}]\!] \mid \forall s \in T.v(s) = 0, \forall s \in U.v(s) = 1\}$ and $f_1 \sqsubseteq_\kappa f_2$ if and only if $\forall s \in S.f_1(s) \le f_2(s)$.*
2. *For $\kappa = Prop^{T_1,U_1} \to Prop^{T_2,U_2} \to \dots Prop^{T_k,U_k} \to Prop^{T,U}$ $(k \ge 1)$, D_κ is the set of affine functions $f : ([0,1]^n)^k \to [0,1]^n$ which belong to $[\![tr(\kappa)]\!]$ (with the identification between $[0,1]^S$ and $[0,1]^n$), and $f_1 \sqsubseteq_\kappa f_2$ if and only if for every tuple $(v_1, v_2, \dots, v_k)$ in $[\![Prop^{T_1,U_1}]\!] \times \cdots \times [\![Prop^{T_k,U_k}]\!]$, the relation $f_1\, v_1\, v_2\, \dots\, v_k \sqsubseteq f_2\, v_1\, v_2\, \dots\, v_k$ holds.*

We now give a matrix representation $Mat_\kappa(f)$ for each type $\kappa \neq Prop_{\{0,1\}}$ of $\mathcal{T}_M$ and $f \in [\![\kappa]\!]$. For $v \in [\![Prop^{T,U}]\!]$, we write $Vec(v)$ for the $1 \times n$ matrix $(v(s_1)\, v(s_2)\, \dots\, v(s_n))$.

▶ **Definition 31** (Matrix Representation). *For an element $f \in [\![\kappa]\!]$ where $\kappa = Prop^{T_1,U_1} \to Prop^{T_2,U_2} \to \dots Prop^{T_k,U_k} \to Prop^{T,U}$ $(k \ge 0)$, its* matrix representation *$Mat_\kappa(f)$ is the (unique) matrix $M = (m_{ij})_{ij}$ of size $(n+1) \times (kn+1)$ satisfying the following conditions.*

1. *For every tuple $(v_1, v_2, \dots, v_k)$ where $v_i \in [\![Prop^{T_i,U_i}]\!]$ $(1 \le i \le k)$, the following equality holds.*

$$M \begin{pmatrix} 1 & Vec(v_1) & Vec(v_2) & \dots & Vec(v_k) \end{pmatrix}^\top = \begin{pmatrix} 1 & Vec(f\, v_1\, v_2\, \dots\, v_k) \end{pmatrix}^\top$$

2. *For each i $(1 \le i \le k)$, $s_j \in T_i \cup U_i$, and ℓ $(1 \le \ell \le n+1)$, the equality $m_{\ell,(i-1)n+j+1} = 0$ holds.*
3. *For each j $(1 \le j \le kn+1)$ and $s_i \in T$, the equality $m_{i+1,j} = 0$ holds. Also, for each j $(2 \le j \le kn+1)$ and $s_i \in U$, the equaities $m_{i+1,1} = 1$ and $m_{i+1,j} = 0$ hold.*
4. *For each j $(2 \le j \le kn+1)$, the equalities $m_{11} = 1$ and $m_{1j} = 0$ hold.*

The existence of M satisfying the first condition is obvious from the assumption that f is affine. The other conditions are imposed to ensure the uniqueness of M. We often omit the type annotation and just write Mat for Mat_κ.

When $k = 0$, the matrix representation $Mat(v)$ for $v \in [\![Prop^{T,U}]\!]$ is given by $Mat(v) = \begin{pmatrix} 1 & Vec(v) \end{pmatrix}^\top$.

Given a 2-dimensional matrix M, we write M_{ij} for the (i,j)-entry of M. The order $\leq$ between two matrices M and M' of the same size $(n+1)\times(kn+1)$ is defined as the pointwise order, i.e., $M \leq M' \overset{\mathrm{def}}{\iff} \forall 1 \leq i \leq n+1, 1 \leq j \leq kn+1. M_{ij} \leq M'_{ij}$.

We define the matrix semantics of a type κ by $[\![\kappa]\!]_{\mathit{Mat}} = \mathit{Mat}([\![\kappa]\!]) = \{\mathit{Mat}(f) \mid f \in [\![\kappa]\!]\}$. For a type environment $\mathcal{K}$, its matrix semantics $[\![\mathcal{K}]\!]_{\mathit{Mat}}$ is the set of maps η_{Mat} satisfying $\mathit{dom}(\eta_{\mathit{Mat}}) = \mathit{dom}(\mathcal{K})$ and $\eta_{\mathit{Mat}}(X) \in [\![\mathcal{K}(X)]\!]_{\mathit{Mat}}$ for all $X \in \mathit{dom}(\mathcal{K})$. For a type derivation $\mathcal{K} \vdash_M \phi : \kappa$, we write $[\![\mathcal{K} \vdash_M \phi : \kappa]\!]_{\mathit{Mat}}$ for the map from $[\![\mathcal{K}]\!]_{\mathit{Mat}}$ to $\mathit{Mat}([\![\kappa]\!])$ defined by:

$$[\![\mathcal{K} \vdash_M \phi : \kappa]\!]_{\mathit{Mat}}(\eta_{\mathit{Mat}}) = \mathit{Mat}([\![\mathcal{K} \vdash_M \phi : \kappa]\!](\eta))$$

Here, η satisfies $\eta(X) = \mathit{Mat}^{-1}(\eta_{\mathit{Mat}}(X))$ for each $X \in \mathit{dom}(\mathcal{K})$. For the well-definedness of $[\![\mathcal{K} \vdash_M \phi : \kappa]\!]_{\mathit{Mat}}$ above, it must be the case that $[\![\mathcal{K} \vdash_M \phi : \kappa]\!](\eta) \in [\![\kappa]\!]$, which can be easily checked.

▶ **Example 32.** Let $M = (S, P, \rho_{AP}, s_{\mathrm{in}})$ be a Markov chain such that

- $S = \{s_1, s_2, s_3\}$,
- P satisfies $P(s_1, s_2) = 0.4$, $P(s_1, s_3) = 0.6$, $P(s_2, s_1) = P(s_3, s_1) = 1$ and $P(s_i, s_j) = 0$ for all the other pairs $(s_i, s_j) \in S \times S$,
- there exist $p_1, p_2, p_3 \in AP$ such that $\rho_{AP}(p_i) = \{s_i\}$ for each $i \in \{1, 2, 3\}$, and
- $s_{\mathrm{in}} = s_1$

Let us consider the the formula $\phi = \lambda X. \bigcirc (((\{p_1\} \vee \{p_2\}) \wedge \bigcirc X) \vee (\{p_3\} \wedge \bigcirc X))$. The matrix representation of the semantics of ϕ is

$$\left[\!\!\left[\phi : \mathit{Prop}^{\{s_3\},\emptyset} \to \mathit{Prop}^{\emptyset,\emptyset}\right]\!\!\right]_{\mathit{Mat}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.4 & 0 \\ 0 & 0 & 0.4 & 0 \end{pmatrix}$$

$$\left[\!\!\left[\phi : \mathit{Prop}^{\emptyset,\{s_3\}} \to \mathit{Prop}^{\emptyset,\emptyset}\right]\!\!\right]_{\mathit{Mat}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0.6 & 0 & 0.4 & 0 \\ 0.6 & 0 & 0.4 & 0 \end{pmatrix}$$

Note that the matrix representation $[\![\phi : \kappa]\!]_{\mathit{Mat}}$ depends on the type κ. ◀

Henceforth, we assume that a formula is given in the form of a *hierarchical equation system* (HES) [11]: $\mathcal{E} = (X_1 =_\mu \phi_1; X_2 =_\mu \phi_2; \dots; X_k =_\mu \phi_k)$, where ϕ_i does not contain fixpoint operators. The corresponding PHFL formula $\mathit{toPHFL}(\mathcal{E})$ is given by:

$$\mathit{toPHFL}(X =_\mu \phi) = \mu X.\phi \qquad \mathit{toPHFL}(\mathcal{E}; X =_\mu \phi) = \mathit{toPHFL}([\mu X.\phi/X]\mathcal{E}).$$

For an HES $\mathcal{E} = (X_1 =_\mu \phi_1; X_2 =_\mu \phi_2; \dots; X_k =_\mu \phi_k)$, we define the fixpoint equation $\mathit{toFP}(\mathcal{E})$ by:

$$\mathit{toFP}(\mathcal{E}) := (M_1 = [\![\phi_1]\!]_{\mathit{Mat}}(\eta_{\mathit{Mat}}); M_2 = [\![\phi_2]\!]_{\mathit{Mat}}(\eta_{\mathit{Mat}}); \dots; M_k = [\![\phi_k]\!]_{\mathit{Mat}}(\eta_{\mathit{Mat}})).$$

Here, η_{Mat} maps each variable X_i to the matrix M_i that contains variables that represent unknown values. The following theorem guarantees that the semantics of the formula can be effectively computed by using the matrix representation (see [18] for details).

▶ **Theorem 33.** *Let ϕ be a formula whose HES form is $(X_1 =_\mu \phi_1; X_2 =_\mu \phi_2; \dots; X_k =_\mu \phi_k)$. Suppose $\emptyset \vdash_M \phi : \mathit{Prop}^{T,U}$. Let $(M_1 = m_1; M_2 = m_2; \dots; M_k = m_k)$ be the least solution of the fixpoint equation $\mathit{toFP}(\mathcal{E})$, and v be the entry of the matrix m_1 which corresponds to the initial state s_{in} of the Markov chain. Then we have $[\![\phi]\!](s_{in}) = v$.*

Since a fixpoint equation on reals can be solved in PSPACE [3], we have the following result as a corollary of this theorem, which subsumes Theorem 28.

▶ **Corollary 34.** *Let M be a Markov chain. If $\emptyset \vdash_M \psi : Prop_{\{0,1\}}$, then whether $M \models \psi$ is decidable in space polynomial in $n(d+s)$, where n is the number of the states of M, d is the size of ψ and s is the sum of the arities of the order-1 variables bound by fixpoint operators.*

B.2 Proof of Theorem 29

Recursive Markov chains can be encoded as order-1 probabilistic HORS (PHORS) [10]. Thus, in this section, we show how the termination problem for PHORS can be encoded into a PHFL model-checking problem in the restricted class.

We transform an order-1 PHORS $\mathcal{G}$ to a pair of a Markov chain M and a PHFL formula ϕ typable in $\mathcal{T}_M$ where, for any $0 \leq r \leq 1$, the value $[\![[\phi]_{\geq r}]\!](s_{\text{in}})$ over the Markov chain M equals 1 if and only if the termination probability of $\mathcal{G}$ is no less than r.

In the rest of this section we follow the notational conventions and definitions about PHORS and higher-order fixpoint equations from [10].

We first fix an order-1 PHORS $\mathcal{G} = (\mathcal{N}, \mathcal{R}, S)$ where $dom(\mathcal{N}) = \{S, F_1, F_2, \ldots, F_m\}$, $\mathcal{N}(F_i) = \underbrace{\mathsf{o} \to \mathsf{o} \to \cdots \to \mathsf{o}}_{k_i} \to \mathsf{o}$ (which is denoted by $\mathsf{o}^{k_i} \to \mathsf{o}$) and $\mathcal{R}$ is such that $F_i\, X_1\, X_2\, \ldots\, X_{k_i} = t_{i,L} \oplus_{p_i} t_{i,R}$ for each $1 \leq i \leq m$ and $S = t_S \oplus_1 \Omega$. Without loss of generality, we assume $p_1 \leq p_2 \leq \cdots \leq p_m$. We write $\mathcal{P}(\mathcal{G})$ for the termination probability of the PHORS $\mathcal{G}$.

We define the Markov chain $M = (S, P, \rho_{AP}, s_{\text{in}})$ as follows.

- $S = \{s_0, s_1, \ldots, s_{m+1}\}$,
- P satisfies $P(s_0, s_1) = p_1$, $P(s_0, s_i) = p_i - p_{i-1}$ for $2 \leq i \leq m$, $P(s_0, s_{m+1}) = 1 - p_m$, $P(s_i, s_0) = 1$ for $1 \leq i \leq m+1$ and $P(s_i, s_j) = 0$ otherwise,
- $\rho_{AP}(P_i) = \{s_i\}$ for each $0 \leq i \leq m+1$, and
- $s_{\text{in}} = s_0$.

Before defining the formula ϕ, we define, for each applicative term t of PHORS, the PHFL formula $\langle t \rangle$ by induction on the structure of t as follows.

$$\begin{aligned} \langle \mathit{halt} \rangle &= \{P_0\} & \langle \Omega \rangle &= \{\text{false}\} \\ \langle X \rangle &= X & \langle F_i \rangle &= F_i \\ \langle f\, u_1\, u_2\, \ldots\, u_m \rangle &= \langle f \rangle\ \langle u_1 \rangle\ \langle u_2 \rangle\ \ldots\ \langle u_k \rangle\,. \end{aligned}$$

We also define the formula $br(\phi_L, \phi_R, i)$ for formulas ϕ_L, ϕ_R and an index $1 \leq i \leq m$ by:

$$br(\phi_L, \phi_R, i) = \{P_0\} \wedge \bigcirc \left(\left((\bigcirc \phi_L) \wedge \left(\bigvee_{1 \leq j \leq i} \{P_j\} \right) \right) \vee \left((\bigcirc \phi_R) \wedge \left(\bigvee_{i+1 \leq j \leq m+1} \{P_j\} \right) \right) \right).$$

Then the desired formula ϕ is given by $\phi = \mathit{toPHFL}(\mathcal{E})$ where $\mathcal{E} = (S =_\mu \langle t_S \rangle\,; F_1 =_\mu \lambda X_1.\lambda X_2.\ldots\lambda X_{k_1}.br(\langle t_{1,L} \rangle, \langle t_{1,R} \rangle, 1);\ldots;F_m =_\mu \lambda X_1.\lambda X_2.\ldots\lambda X_{k_m}.br(\langle t_{m,L} \rangle, \langle t_{m,R} \rangle, m))$. Then ϕ has type $Prop^{\{s_1, s_2, \ldots, s_{m+1}\}, \emptyset}$ (thus belongs to the decidable subclass), and $\mathcal{P}(\mathcal{G}) = [\![\phi]\!](s_{\text{in}})$ holds; see [18] for details.

Adaptive Non-Linear Pattern Matching Automata

Rick Erkens
Eindhoven University of Technology, The Netherlands
r.j.a.erkens@tue.nl

Maurice Laveaux
Eindhoven University of Technology, The Netherlands
m.laveaux@tue.nl

Abstract

Efficient pattern matching is fundamental for practical term rewrite engines. By preprocessing the given patterns into a finite deterministic automaton the matching patterns can be decided in a single traversal of the relevant parts of the input term. Most automaton-based techniques are restricted to linear patterns, where each variable occurs at most once, and require an additional post-processing step to check so-called variable consistency. However, we can show that interleaving the variable consistency and pattern matching phases can reduce the number of required steps to find all matches. Therefore, we take the existing adaptive pattern matching automata as introduced by Sekar et al and extend these with consistency checks. We prove that the resulting deterministic pattern matching automaton is correct, and show that its evaluation depth can be shorter than two-phase approaches.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Pattern matching, Term indexing, Tree automata

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.20

Acknowledgements We would like to thank Jan Friso Groote, Bas Luttik and Tim Willemse for their feedback and discussion. We would also like to thank the anonymous reviewers for their effort and constructive remarks and questions.

1 Introduction

Term rewriting is a universal model of computation that is used in various applications, for example to evaluate equalities or simplify expressions in model checking and theorem proving. In its simplest form, a binary relation on *terms*, which is described by the *term rewrite system*, defines the available reduction steps. Term rewriting is then the process of repeatedly applying these reduction steps when applicable. The fundamental step in finding which reduction steps are applicable is *pattern matching*.

There are two variants for the pattern matching problem. *Root pattern matching* can be described as follows: given a term t and a set of patterns, determine the subset of patterns such that these are (syntactically) equal to t under a suitable substitution for their variables. The other variant, called *complete pattern matching*, determines the matching patterns for all subterms of t. Root pattern matching is often sufficient for term rewriting, because reduction steps invalidate other matches. A root pattern matching algorithm can be used to naively solve the complete pattern matching problem by applying it to every subterm.

As the matching patterns need to be decided at each reduction step, various *term indexing* techniques [7] have been proposed to determine matching patterns efficiently. Adaptive pattern matching automata [8] (APMA) are tree-like data structures that are constructed from a set of patterns. By using such an automaton one can decide the matching patterns by only examining each function symbol of the input term at most once. Moreover it allows for *adaptive* strategies, i.e., matching strategies that are not restricted to a fixed traversal

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 20; pp. 20:1–20:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

such as a left-to-right traversal in [4]. The size of an APMA is worst-case exponential in the size of the pattern set, but in practice its size is typically smaller and this preprocessing step is beneficial when many terms have to be matched against a fixed pattern set.

The APMA approach works for sets of linear patterns, that is, in every pattern every variable occurs at most once. As mentioned in other literature [4, 8] the non-linear matching problem can be solved by first preprocessing the patterns, then solving the linear matching problem and lastly checking so-called *variable consistency.* Performing matching and consistency checking separately does not yield the optimal matching time. Therefore we extend the existing APMA with consistency checking on the fly. Our extension preserves the adaptive traversal of [8] and allows information about the matching step to influence the consistency checking, and the other way around.

We introduce *consistency automata* (CA) to perform the variable consistency check efficiently for a set of patterns. The practical use of this automaton is based on similar observations as the pattern matching automata: there may be overlapping consistency constraints for multiple patterns in a set. We prove the correctness for these consistency automata and provide an analysis of its time and space complexity. We prove that the consistency automaton approach yields a correct consistency checking algorithm for non-linear patterns. Then we introduce *adaptive non-linear pattern matching automata* (ANPMA), a combination of adaptive pattern matching automata and consistency automata. ANPMAs use information from both match and consistency checks to allow the removal of redundant steps. We show that ANPMA yield a correct matching algorithm for non-linear patterns. To this end we also give a correctness proof for the APMA approach from [8], which was not given in the original work.

We compare this work with other term indexing techniques. Most techniques use tree-like data structures with deterministic [1, 4, 8, 9] or non-deterministic [3, 2, 6, 10, 5] evaluation. In this setting a deterministic evaluation guarantees that all positions in the input term are inspected at most once. Non-deterministic approaches typically have smaller automata, but the same position might be inspected multiple times for input terms as a result of backtracking.

Not all techniques support matching non-linear patterns. *Discrimination trees* [6], *substitution trees* [5] and *match trees* [9] can be extended with on-the-fly consistency checks for matching non-linear patterns. Their evaluation strategy however is restricted to preorder evaluation and variable consistency must be checked whenever a variable which has already been bound occurs at the current evaluated position of the pattern. We have also considered *code trees* [10], which also have preorder evaluation with backtracking. These allow consistency checks to occur at different places. All three approaches might inspect the same position multiple times due to backtracking. The ANPMAs introduced in this paper mitigate these issues: consistency checks are allowed to occur at any point in the automaton, the evaluation strategy is not limited to a fixed strategy and there are no redundant checks.

2 Preliminaries

In this section the preliminaries of first-order terms and the pattern matching problem are defined. We denote the *disjoint union* of two sets A and B by $A \uplus B$. Given two sets A and B we use $A \to B$, $A \rightharpoonup B$ and $A \hookrightarrow B$ to denote the sets of total, partial and total injective functions from A to B respectively. We assume that a partial function yields a special symbol $\bot$ for elements in its domain for which it is undefined. Furthermore, we assume the existence of an index set $\mathcal{I}$ and use $A \times \mathcal{I}$ to denote the indexed family with elements denoted by $i : a$ for $a \in A$ and $i \in \mathcal{I}$.

Let $\mathbb{F} = \biguplus_{i \in \mathbb{N}} \mathbb{F}_i$ be a *ranked* alphabet. We say that $f \in \mathbb{F}_i$ is a *function symbol* with arity, written $\mathsf{ar}(f)$, equal to i. Let $\Sigma = \mathbb{V} \uplus \mathbb{F}$ be a *signature* where $\mathbb{V}$ is a set of variables. The set of terms over Σ, denoted by $\mathbb{T}_\Sigma$, is defined as the smallest set such that $\mathbb{V} \subseteq \mathbb{T}_\Sigma$ and whenever $t_1, \ldots, t_n \in \mathbb{T}_\Sigma$ and $f \in \mathbb{F}_n$, then also $f(t_1, \ldots, t_n) \in \mathbb{T}_\Sigma$. We typically use the symbols x, y for variables, symbols a, b for function symbols of arity zero (constants), f, g, h for function symbols of other arities and t, u for terms. The *head* of a term, written as head, is defined as $\mathsf{head}(x) = x$ for a variable x and $\mathsf{head}(f(t_1, \ldots, t_n)) = f$ for a term $f(t_1, \ldots, t_n)$. We use $\mathsf{vars}(t)$ to denote the set of variables that occur in term t. A term for which $\mathsf{vars}(t) = \emptyset$ is called a *ground term.* A *pattern* is a term of the form $f(t_1, \ldots, t_n)$. A pattern is *linear* iff every variable occurs at most once in it.

We define the (syntactical) equality relation $= \subseteq \mathbb{T}^2$ as the smallest relation such that $x = x$ for all $x \in \mathbb{V}$, and $f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)$ if and only if $t_i = t'_i$ for all $1 \leq i \leq n$. Furthermore, the equality relation modulo variables $=_\omega \subseteq \mathbb{T}^2$ is the smallest relation such that $x =_\omega y$ for all $x, y \in \mathbb{V}$, and $f(t_1, \ldots, t_n) =_\omega f(t'_1, \ldots, t'_n)$ if and only if $t_i =_\omega t'_i$ for all $1 \leq i \leq n$. Both $=$ and $=_\omega$ satisfy reflexivity, symmetry and transitivity and thus are equivalence relations, and we can observe that $= \subseteq =_\omega$.

A *substitution* σ is a total function from variables to terms. The application of a substitution σ to a term t, denoted by t^σ, is the term where variables of t have been replaced by the term assigned by the substitution. This can be inductively defined as $x^\sigma = \sigma(x)$ and $f(t_1, \ldots, t_n)^\sigma = f(t_1^\sigma, \ldots, t_n^\sigma)$. We say that term u *matches* t, denoted by $t \leq u$, iff there is a substitution σ such that $t^\sigma = u$. Terms t and u *unify* iff there is a substitution σ such that $t^\sigma = u^\sigma$.

We define the set of *positions* $\mathbb{P}$ as the set of finite sequences over natural numbers where the *root* position, denoted by ϵ, is the identity element and concatenation, denoted by dot, is an associative operator. Given a term t we define $t[\epsilon] = t$ and if $t[p] = f(t_1, \ldots, t_n)$ then $t[p.i]$ for $1 \leq i \leq n$ is equal to t_i. Note that $t[p]$ may not be defined, e.g., $f(x, y)[3]$ and $f(x, y)[1.1]$. A position p is *higher* than q, denoted by $p \sqsubseteq q$, iff there is position $r \in \mathbb{N}^*$ such that $p.r = q$. Position p is *strictly higher* than q, denoted by $p \sqsubset q$, whenever $p \sqsubseteq q$ and $p \neq q$. We say that a term $t[q]$ is a *subterm* of $t[p]$ if $p \sqsubset q$ and $t[q]$ is defined. The replacement of the subterm at position p by term u in term t is denoted by $t[p/u]$, which is defined as $t[\epsilon/u] = u$ and $f(t_1, \ldots, t_n)[(i.p)/u] = f(t_1, \ldots, t_i[p/u], \ldots, t_n)$. The *fringe* of a term t, denoted by $\mathcal{F}(t)$, is the set of all positions at which a variable occurs, given by $\mathcal{F}(t) = \{p \in \mathbb{P} \mid t[p] \in \mathbb{V}\}$.

We also define a restricted signature for terms with a one-to-one correspondence between variables and positions. First, we define $\mathbb{V}_\mathbb{P}$ as the set of *position variables* $\{\omega_p \mid p \in \mathbb{P}\}$. Consider the signature $\Sigma_\mathbb{P} = \mathbb{F} \uplus \mathbb{V}_\mathbb{P}$. We say that a term $t \in \mathbb{T}_{\Sigma_\mathbb{P}}$ is *position annotated* iff for all $p \in \mathcal{F}(t)$ we have that $t[p] = \omega_p$. For example, the terms ω_ϵ and $f(\omega_1, g(\omega_{2.1}))$ are position annotated whereas the term $f(\omega_{1.1})$ is not. Position annotated patterns are linear as each variable can occur at most once.

A *matching function* decides for a given term and a set of patterns the exact subset of these patterns that match the given term.

▶ **Definition 1.** *Let $\mathcal{L} \subseteq \mathbb{T}_\Sigma$ be a set of patterns. A function $match_\mathcal{L} : \mathbb{T}_\Sigma \to 2^{\mathbb{T}_\Sigma}$ is a* matching function *for $\mathcal{L}$ iff for all terms t we have $match_\mathcal{L}(t) = \{\ell \in \mathcal{L} \mid \exists \sigma : \ell^\sigma = t\}$. If $\mathcal{L}$ is a set of linear patterns then $match_\mathcal{L}$ is a* linear matching function*.*

3 Adaptive Pattern Matching Automata

For a single linear pattern to match a given term it is necessary that every function symbol of the pattern occurs at the same position in the given term.

▶ **Proposition 2.** *Let ℓ and ℓ' be linear patterns. We have that $\ell \leq \ell'$ if and only if for all positions p: if $\mathsf{head}(\ell[p]) \in \mathbb{F}$ then $\mathsf{head}(\ell[p]) = \mathsf{head}(\ell'[p])$.*

A naive matching algorithm for linear patterns follows directly from this proposition: to find all matches for term t one can check the proposition for every pattern separately. However, for a *set* of patterns we can observe that whenever a specific position of the given term is inspected a decision can be made for all patterns at the same time. This is the purpose of so-called *term indexing techniques* [7]. Sekar et al. [8] describe the construction of a so-called *adaptive pattern matching automaton*, abbreviated as APMA. Given a set of *linear* patterns $\mathcal{L}$ an APMA can be constructed that can be used to decide for every term $t \in \mathbb{T}_\Sigma$ which patterns of $\mathcal{L}$ are matches for t. The advantage of using an APMA over the naive approach is that for every input term, every position is inspected at most once.

We present the evaluation and construction procedures of APMAs slightly differently compared to the presentation by Sekar et al. APMAs are state machines in which every state is a matching state, which is labelled with a position, or final state, which is labelled with a set of patterns. Matching states indicate that the term under evaluation is being inspected at the labelled position. Final states indicate that a set of matching patterns is found. The transitions are labelled by function symbols or an additional *fresh* symbol $\boxtimes \notin \mathbb{F}$; let $\mathbb{F}_\boxtimes = \mathbb{F} \uplus \{\boxtimes\}$.

▶ **Definition 3.** *An APMA is a tuple (S, δ, L, s_0) where:*

- *$S = S_M \uplus S_F$ is a finite set of states consisting of a set of* match states *S_M and a set of* final states *S_F;*
- *$\delta : S_M \times \mathbb{F}_\boxtimes \rightharpoonup S$ is a partial transition function;*
- *$L = L_M \uplus L_F$ is a state labelling function with $L_M : S_M \to \mathbb{P}$ and $L_F : S_F \to 2^{\mathbb{T}_\Sigma}$*
- *$s_0 \in S_M$ is the initial state.*

We only consider APMAs that have a tree structure that is rooted in s_0. That is, δ is an injective partial mapping and there is no pair (s, f) with $\delta(s, f) = s_0$.

Consider the patterns $f(a, b, x), f(c, b, x)$ and $f(c, b, c)$ with $a, b, c \in \mathbb{F}_0$, $f \in \mathbb{F}_3$ and $x \in \mathbb{V}$. Figure 1 shows an APMA that can be used to decide which of these patterns match. In addition to the position label on every matching state, it also displays the term that represents what has been matched so far. That is, in the state labelled with position 2, only the function symbol f has been inspected. The term $f(\omega_1, \omega_2, \omega_3)$ represents that f has been inspected and the variables at positions 1, 2 and 3 represent that these positions have not been inspected. We refer to this term as a *prefix.* Prefixes are not a part of the APMA; they are included for comprehensiveness only. Later they will aid in the construction algorithm and the correctness proof.

The function Match below defines the evaluation of an APMA on a term. Upon reaching a final state $s \in S_F$ the evaluation yields the set of terms $L_F(s)$. In a matching state $s \in S_M$ the head symbol $\mathsf{head}(t[L_M(s)])$ is examined. If there is an outgoing transition labelled with this head symbol then evaluation continues in the resulting state; otherwise the $\boxtimes$-transition is taken. Whenever there is no outgoing $\boxtimes$-transition then there is no match and the evaluation returns the empty set as a result.

$$\textsc{Match}(M, t, s) = \begin{cases} L_F(s) & \text{if } s \in S_F \\ \textsc{Match}(M, t, \delta(s, f)) & \text{if } s \in S_M \wedge \delta(s, f) \neq \bot \\ \textsc{Match}(M, t, \delta(s, \boxtimes)) & \text{if } s \in S_M \wedge \delta(s, \boxtimes) \neq \bot \wedge \delta(s, f) = \bot \\ \emptyset & \text{if } s \in S_M \wedge \delta(s, \boxtimes) = \delta(s, f) = \bot \end{cases}$$

$$\text{where } f = \mathsf{head}(t[L_M(s)])$$

If we consider the APMA M of Figure 1 and let initial state s_0 be the topmost state in the figure. We have $\textsc{Match}(M, f(a, b, a), s_0) = \{f(a, b, x)\}$ and $\textsc{Match}(M, f(b, b, b), s_0) = \emptyset$. The term $f(c, b, b)$ will yield the pattern set $\{f(c, b, x)\}$.

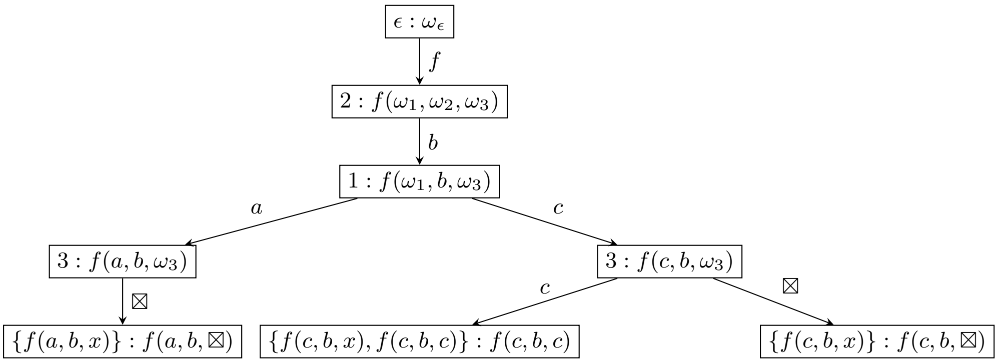

$\epsilon : \omega_\epsilon$
f
$2 : f(\omega_1, \omega_2, \omega_3)$
b
$1 : f(\omega_1, b, \omega_3)$
a
c
$3 : f(a, b, \omega_3)$
$3 : f(c, b, \omega_3)$
$\boxtimes$
c
$\boxtimes$
$\{f(a, b, x)\} : f(a, b, \boxtimes)$
$\{f(c, b, x), f(c, b, c)\} : f(c, b, c)$
$\{f(c, b, x)\} : f(c, b, \boxtimes)$

Figure 1 An APMA constructed from the patterns given above.

In Algorithm 1 the APMA construction is defined. Intuitively Construct creates the APMA from root to leaf based on the pattern set $\mathcal{L}$ and the selection function Select. For convenience we also assume that all patterns in $\mathcal{L}$ are position-annotated. In later sections we drop this assumption in order to treat non-linear patterns. The algorithm is initially called with the initial state s_0, after which every recursive call corresponds to a state deeper in the tree. The parameter Select is a function that determines in each recursive call which position from work becomes the label for the current state. Based on the selected position, the current state and the pattern set, outgoing transitions are created to fresh states where the construction continues recursively.

The prefix associated with each state plays an important role during construction. The function symbols in pref represent which function symbols have been matched so far and the variables in pref represent which positions have not been inspected yet. Each recursive call starts by removing all the patterns from $\mathcal{L}$ that do not unify with pref. Any match for the removed patterns cannot reach the state of the subautomaton that is currently being constructed. Therefore, the removed patterns do not have to be considered for the remainder of the construction. If there are no variables in pref then there is nothing to be inspected anymore. This is the termination condition for the construction; the current state s will be labelled with the patterns that unify with pref. Otherwise, the work that still has to be done, i.e., the set of positions that still have to be inspected, is the fringe of pref, denoted by $\mathcal{F}(\mathsf{pref})$. If pref has the symbol $\boxtimes$ at position p then none of the patterns in $\mathcal{L}$ that have a non-variable subterm at position p can unify with the prefix any more.

3.1 Proof of Correctness

We prove that this construction yields an APMA that is suitable to solve the matching problem for non-empty finite sets of linear patterns. We make use of the following auxilliary definitions. A *path* to s_n is a sequence of state and function symbol pairs $(s_0, f_0), \ldots, (s_{n-1}, f_{n-1}) \in S_M \times \mathbb{F}_\boxtimes$ such that $\delta(s_i, f_i) = s_{i+1}$ for all $i < n$. Because δ is required to be an injective partial mapping there is a unique path to s for every state s, which we denote by $\mathsf{path}(s)$. A matching state s is *top-down* iff $L(s) = \epsilon$ or there is a pair (s_i, f_i) in $\mathsf{path}(s)$ with $L(s_i).j = L(s)$ for some $1 \leq j \leq \mathsf{ar}(f_i)$. State s is *canonical* iff there are no two states in $\mathsf{path}(s)$ that are labelled with the same position. Finally we say that an APMA is *well-formed* iff all matching states are top-down and canonical.

Well-formed APMAs allow us to inductively reconstruct the prefix of a state s as it was created in the construction algorithm. We allow slight overloading of the notation and denote the prefix of state s by $\mathsf{pref}(s)$. It is constructed inductively for well-formed APMAs by $\mathsf{pref}(s_0) =$

Algorithm 1 Given a finite set of patterns $\mathcal{L}$, this algorithm constructs an APMA for $\mathcal{L}$. Initially, it is called with $M = (\emptyset, \emptyset, \emptyset, s_0)$, the initial state $s = s_0$ and the prefix $\mathsf{pref} = \omega_\epsilon$.

```
**procedure** $\textsc{Construct}(\mathcal{L}, \textsc{Select}, M, s, \mathsf{pref})$
    $\mathcal{L}' := \{\ell \in \mathcal{L} \mid \ell$ unifies with $\mathsf{pref}\}$
    $\mathsf{work} := \mathcal{F}(\mathsf{pref})$
    **if** $\mathsf{work} = \emptyset$ **then**
        $M := M[S_F := (S_F \cup \{s\}), L_F := L_F[s \mapsto \mathcal{L}']]$
    **else**
        $\mathsf{pos} := \textsc{Select}(\mathsf{work})$
        $M := M[S_M := (S_M \cup \{s\}), L_M := L_M[s \mapsto \mathsf{pos}]]$
        $F := \{f \in \mathbb{F} \mid \exists \ell \in \mathcal{L}' : \mathsf{head}(\ell[\mathsf{pos}]) = f\}$
        **for** $f \in F$ **do**
            $M := M[\delta := \delta[(s, f) \mapsto s']]$ where $s'$ is a fresh unbranded state w.r.t. $M$
            $M := \textsc{Construct}(\mathcal{L}, \textsc{Select}, M, s', \mathsf{pref}[\mathsf{pos} := f(\omega_{\mathsf{pos}.1}, \ldots, \omega_{\mathsf{pos}.\mathsf{ar}(f)})])$
        **if** $\exists \ell \in \mathcal{L}' : \exists \mathsf{pos}' \sqsubseteq \mathsf{pos} : \mathsf{head}(\ell[\mathsf{pos}']) \in \mathbb{V}$ **then**
            $M := M[\delta := \delta[(s, \boxtimes) \mapsto s']]$ where $s'$ is a fresh unbranded state w.r.t. $M$
            $M := \textsc{Construct}(\mathcal{L}, \textsc{Select}, M, s', \mathsf{pref}[\mathsf{pos} := \boxtimes])$
    **return** $M$
```

ω_ϵ and if $\delta(s_i, f) = s_{i+1}$ then $\mathsf{pref}(s_{i+1}) = \mathsf{pref}(s_i)[L(s_i)/f(\omega_{L(s_i).1}, \ldots, \omega_{L(s_i).\mathsf{ar}(f)})]$. Similarly, we denote the patterns of state s for all states by $\mathcal{L}(s) = \{\ell \in \mathcal{L} \mid \ell$ unifies with $\mathsf{pref}(s)\}$. Lastly we use an arbitrary function $\textsc{Select} : 2^{\mathbb{P}} \to \mathbb{P}$ such that for all sets of positions work we have $\textsc{Select}(\mathsf{work}) \in \mathsf{work}$.

▶ **Lemma 4.** *For all finite, non-empty sets of patterns $\mathcal{L}$ we have that the procedure $\textsc{Construct}(\mathcal{L}, \textsc{Select}, (\emptyset, \emptyset, \emptyset, s_0), s_0, \omega_\epsilon)$ terminates and yields a well-formed APMA $M = (S, \delta, L, s_0)$.*

For the remainder of the correctness proof assume an arbitrary finite, non-empty set of position annotated patterns $\mathcal{L}$ and let $M = (S, \delta, L, s_0)$ be the APMA for $\mathcal{L}$ that results from $\textsc{Construct}(\mathcal{L}, \textsc{Select}, (\emptyset, \emptyset, \emptyset, s_0), s_0, \omega_\epsilon)$. Furthermore, let t be an arbitrary term and let $\mathcal{L}_t = \{\ell \in \mathcal{L} \mid \ell \leq t\}$.

The following lemmas state some claims and invariants about $\textsc{Construct}$ and its relation to $\textsc{Match}$. The proofs are rather tedious and are attached in the appendix.

▶ **Lemma 5.** *For every every final state s: (a) the set $L(s)$ is non-empty, (b) $\mathsf{pref}(s)$ is a ground term, and (c) for all $\ell \in L(s)$ we have $\ell \leq \mathsf{pref}(s)$. Moreover (d) for every pattern $\ell \in \mathcal{L}$ there is at least one final state s with $\ell \in L(s)$.*

▶ **Lemma 6.** *For all states s such that $\textsc{Match}(M, t, s_0) = \textsc{Match}(M, t, s)$ it holds that $\mathcal{L}_t \subseteq \mathcal{L}(s)$.*

▶ **Lemma 7.** *It holds that:*

a) *If $\mathcal{L}_t = \emptyset$ then $\textsc{Match}(M, t, s_0) = \emptyset$;*

b) *If $\mathcal{L}_t \neq \emptyset$ then $\textsc{Match}(M, t, s_0) = \textsc{Match}(M, t, s_f)$ for some final state s_f.*

▶ **Lemma 8.** *If $\textsc{Match}(M, t, s_0) = \textsc{Match}(M, t, s_f)$ for some final state s_f then $L(s_f) = \mathcal{L}_t$.*

▶ **Theorem 9.** *Then $\lambda t.\textsc{Match}(M, t, s_0)$ is a linear matching function for pattern set $\mathcal{L}$.*

Proof. Let t be an arbitrary term and let $\mathcal{L}_t = \{\ell \in \mathcal{L} \mid \ell \leq t\}$. If $\mathcal{L}_t = \emptyset$ then by Lemma 7 we get that $\textsc{Match}(M, t, s_0) = \emptyset = \mathcal{L}_t$ as required. If $\mathcal{L}_t$ is non-empty then by Lemma 7 we have that $\textsc{Match}(M, t, s_0) = \textsc{Match}(M, t, s_f)$ for some final state s_f. Then by definition of $\textsc{Match}$ we get $\textsc{Match}(M, t, s_f) = L(s_f)$. From Lemma 8 it follows that $L(s_f) = \mathcal{L}_t$, by which we can conclude $\textsc{Match}(M, t, s_0) = \mathcal{L}_t$. Hence $\lambda t.\textsc{Match}(M, t, s_0)$ is a linear matching function for $\mathcal{L}$. ◀

4 Consistency Checking

As mentioned in other literature [4, 7, 8] a linear matching algorithm can be used to solve the non-linear matching problem by transforming the patterns and checking so-called *variable consistency* after the matching phase. This is required because a variable which occurs at multiple positions can only be assigned a single value in the matching substitution. First, a transformation step ensures that all input terms are changed into linear patterns by *renaming* different occurrences of the variables in the non-linear patterns. The linear matching algorithm can then be used to solve part of the non-linear matching problem. Finally, a consistency check is performed to remove the linear patterns for which the substitution that witnesses the match is not valid for the original patterns. We first focus on efficiently deciding this variable consistency step.

4.1 Pattern Renaming

A straightforward way to achieve the renaming would be to introduce new variables for each position in the fringe of each pattern. However, for patterns $f(x, a)$ and $f(x', y')$ the variables x and x' could be identical such that the assignment for x (or equally x') yields a substitution for both patterns. We can use position annotated variables, which are identical for the same position in different patterns, to obtain these overlapping assignments.

For the consistency check it is necessary to keep track of equality constraints that are forgotten when a non-linear pattern is renamed. For this purpose we introduce *consistency classes* [7]. This is a set of positions with the following notion.

▶ **Definition 10.** *Given a term t and a consistency class $C \subseteq \mathbb{P}$ we say that t is* consistent *w.r.t. C if and only if $t[p] = t[q]$ for all $p, q \in C$.*

A pattern can give rise to multiple consistency classes. For instance, consider the pattern $f(x, x, y, y, y, z)$. Based on the occurrences of variables x, y and z we derive the three classes $\{1, 2\}$, $\{3, 4, 5\}$ and $\{6\}$. This means that for the input term $t = f(t_1, \ldots, t_6)$ that both $t[1] = t[2]$ and $t[3] = t[4] = t[5]$ must hold; and finally $t[i] = t[i]$ holds trivially for all $1 \leq i \leq 6$, for this term to be consistent w.r.t. these classes. A set of consistency classes is referred to as a *consistency partition*. The notion of term consistency w.r.t. a consistency class is extended as follows. A term t is consistent w.r.t. a consistency partition P iff t is consistent w.r.t. C for every $C \in P$.

First, we illustrate the renaming procedure by means of an example. For the purpose of renaming, partitions of the fringe of a pattern are sufficient. Consider three patterns $f(x, x, z)$, $f(x, y, x)$ and $f(x, x, x)$. After renaming we obtain the following pairs of a linear pattern and the corresponding consistency partition: $(f(\omega_1, \omega_2, \omega_3), P_1)$, $(f(\omega_1, \omega_2, \omega_3), P_2)$ and $(f(\omega_1, \omega_2, \omega_3), P_3)$; with the consistency partitions $P_1 = \{\{1, 2\}, \{3\}\}$, $P_2 = \{\{1, 3\}, \{2\}\}$ and $P_3 = \{\{1, 2, 3\}\}$. The term $f(a, a, b)$ matches $f(\omega_1, \omega_2, \omega_3)$ as witnessed by the substitution $\mathsf{id}[\omega_1 \mapsto a, \omega_2 \mapsto a, \omega_3 \mapsto b]$, but $f(a, a, b)$ is only consistent w.r.t. partition P_1. Therefore, the given term only matches pattern $f(x, x, z)$.

We define a rename function that yields a position annotated term and a consistency partition over $\mathcal{F}(t)$ for any given term.

▶ **Definition 11.** *The term rename function $\mathsf{rename} : \mathbb{T}_\Sigma \to (\mathbb{T}_{\Sigma_\mathbb{P}} \times 2^{2^\mathbb{P}})$ is defined as*

$$\mathsf{rename}(t) = (\mathsf{rename}_1(t, \epsilon), \{\{p \in \mathbb{P} \mid t[p] = x\} \mid x \in \mathsf{vars}(t)\})$$

where $\mathsf{rename}_1(t,\epsilon) : (\mathbb{T}_\Sigma \times \mathbb{P}) \to \mathbb{T}_{\Sigma_\mathbb{P}}$ *renames the variables of the given term to position variables, which is defined below.*

$$\begin{aligned} \mathsf{rename}_1(x,p) &= \omega_p && \textit{if } x \in \mathbb{V} \\ \mathsf{rename}_1(f(t_1,\ldots,t_n),p) &= f(\mathsf{rename}_1(t_1,p.1),\ldots,\mathsf{rename}_1(t_n,p.n)) \end{aligned}$$

Note that for linear patterns the result is a position annotated term with trivial consistency classes. We show a number of characteristic properties of the rename function which are essential for the non-linear matching algorithm.

▶ **Lemma 12.** *For all terms* $t \in \mathbb{T}_\Sigma$ *if* $(t',P) = \mathsf{rename}(t)$ *then:*

- $t =_\omega t'$;
- *for all* $p \in \mathcal{F}(t)$: $t'[p] = \omega_p$;
- *for all* $u \in \mathbb{T}_\Sigma$ *it holds that* u *matches* t *if and only if* u *matches* t' *and* u *is consistent w.r.t.* P.

For the variable consistency phase a straightforward implementation follows directly from Definition 10. Let $P = \{C_1,\ldots,C_n\}$ be a partition. For each consistency class C_i, for $1 \leq i \leq n$, there are $|C_i| - 1$ comparisons to perform, after which the consistency of a term w.r.t. C_i is determined. This can be extended to partitions by performing such a check for every consistency class in the given partition. We use the function $\textsc{is-consistent}(t,P)$ to denote this naive algorithm. For a set of partitions $\{P_1,\ldots,P_m\}$ the (naive) consistency check requires exactly $\sum_{1\leq j\leq m}\sum_{C\in P_j}|C|$ comparisons if t is consistent w.r.t. P.

For the renaming procedure we must consider that the patterns $f(x,x)$ and $f(x,y)$ are both renamed to the linear pattern $f(\omega_1,\omega_2)$. However, then it is no longer possible to identify the corresponding original pattern. This can be solved by considering an indexed family of patterns, indexed by elements from $\mathcal{I}$, and adapting the rename function to preserve the corresponding indices. Now, when given an indexed linear pattern that resulted from renaming we can identify the corresponding original pattern by its index. The following lemma follows directly from the third property of Lemma 12.

▶ **Lemma 13.** *Let* $\mathcal{L} \subseteq \mathbb{T}_\Sigma \times \mathcal{I}$ *be a set of patterns and let* $\mathcal{L}_r \subseteq \mathbb{T}_{\Sigma_\mathbb{P}} \times 2^{2^\mathbb{P}} \times \mathcal{I}$ *be the set of linear patterns and corresponding consistency partitions resulting from renaming; i.e.,* $\mathcal{L}_r = \{\mathsf{rename}(l) \mid l \in \mathcal{L}\}$. *Let* $\mathsf{match\text{-}linear} : \mathbb{T}_\Sigma \times 2^{\mathbb{T}_\Sigma} \times \mathcal{I} \to 2^{\mathbb{T}_\Sigma} \times \mathcal{I}$ *be a linear matching function that preserves indices. For any term* $t \in \mathcal{L}$ *we define* $\mathsf{match} : (\mathbb{T}_\Sigma \times (\mathbb{T}_{\Sigma_\mathbb{P}} \times 2^{2^\mathbb{P}} \times \mathcal{I})) \to \mathbb{T}_\Sigma$ *as:*

$$\mathsf{match}(t,\mathcal{L}_r) = \{\ell \mid i : \ell' \in \mathcal{L}' \wedge i : (\ell',P) \in \mathcal{L}_r \wedge \textsc{is-consistent}(t,P)\}$$

where $\mathcal{L}'$ *is equal to* $\mathsf{match\text{-}linear}(t,\{i : \ell' \mid i : (\ell',P) \in \mathcal{L}_r\})$. *The function* match *is a matching function.*

4.2 Consistency Automata

In this section, we are only going to focus on solving the consistency checking efficiently and later on we show that the matching time can be further improved by interleaving the choices. Consider the consistency partitions $P_1 = \{\{1,2\},\{3\}\}$, $P_2 = \{\{1,3\},\{2\}\}$ and $P_3 = \{\{1,2,3\}\}$ again. We would expect that similarly to an APMA we can use the fact that comparisons of overlapping partitions can be used to determine the subset of all consistent partitions directly. This means that, at most three comparisons $t[1] = t[2]$, $t[2] = t[3]$ and $t[1] = t[3]$ would have to be performed to determine the consistent partitions. For this reason,

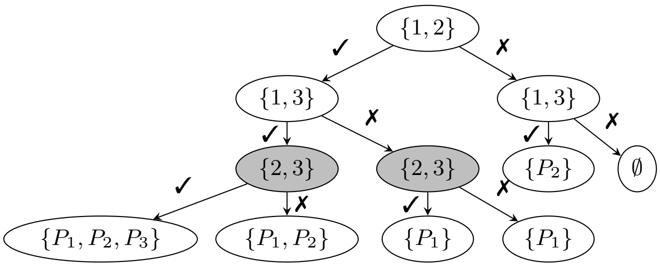


Figure 2 The CA for the partitions $P_1 = \{\{1,3\},\{2\}\}$, $P_2 = \{\{1,2\},\{3\}\}$ and $P_3 = \{\{1,2,3\}\}$ where positions 1 and 2 are compared first, followed by 1 and 3 and finally 2 and 3. The grey states are redundant and can be removed as shown in later steps.

we define *consistency automata* which are constructed from a set of consistency partitions. Each state of this automaton is labelled with a pair of positions that should be compared. Similar labelling is also present in other matching algorithms [10], but not presented as a separate automaton. Afterwards, we show that redundant comparisons can be removed such that this example requires at most two comparisons.

A consistency automaton, abbreviated CA, is a state machine where every state is a consistency state, which is labelled with a pair of positions, or a final state, which is labelled with set of partitions. The transitions are labelled with either ✓ or ✗ to indicate that the compared positions are equal or unequal respectively. The evaluation of a CA determines the consistency of a term w.r.t. a given set of partitions.

▶ **Definition 14.** *A* consistency automaton *is a tuple* (S, δ, L, s_0) *where:*

- $S = S_C \uplus S_F$ *is a set of states consisting of a set of consistency states* S_C *and a set of final states* S_F*;*
- $\delta : (S_C \times \{✓, ✗\}) \to S$ *is a transition function;*
- $L = L_C \uplus L_F$ *is a state labelling function with* $L_C : S_C \to \mathbb{P}^2$ *and* $L_F : S_F \to 2^{\mathcal{I}}$*;*
- $s_0 \in S$ *is the initial state.*

We show an example to illustrate the intuition behind the evaluation function of a CA. Consider the consistency partitions $P_1 = \{\{1,2\},\{3\}\}$, $P_2 = \{\{1,3\},\{2\}\}$ and $P_3 = \{\{1,2,3\}\}$ again. Figure 2 shows a CA that can be used to decide the consistency of a given term t w.r.t. any of these partitions. If the consider the state labelled with $\{1,2\}$ the subterms $t[1]$ and $t[2]$ are compared. Whenever these are equal the evaluation continues with the ✓-branch and it continues with the ✗-branch otherwise. If a final state (labelled with partitions) is reached then t is consistent w.r.t. these partitions by construction.

The evaluation function of a CA for the input term and a given state, starting with the initial state, is defined below. First, it checks whether the current state is final, in which case the label $L(s)$ indicates the set of indices such that t is consistent w.r.t. the partitions P_i for $i \in L(s)$. Otherwise, evaluation proceeds by considering the pair of positions given by $S_C(s)$. The positions given by $S_C(s)$ are unordered pairs of positions (or 2-sets), denoted by $\mathbb{P}^2$, with elements $\{p, q\}$ such that $p \neq q$. These unordered pairs avoid unnecessary comparisons by the reflexivity and symmetry of term equality. If the comparison yields true the evaluation proceeds with the state of the outgoing ✓-transition; otherwise it proceeds with the state of the outgoing ✗-transition.

$$\textsc{eval-ca}(M,t,s) = \begin{cases} L_F(s) & \text{if } s \in S_F \\ \textsc{eval-ca}(M,t,\delta(s,\checkmark)) & \text{if } s \in S_C \wedge t[p] = t[q] \text{ where } \{p,q\} = L_C(s) \\ \textsc{eval-ca}(M,t,\delta(s,\text{✗})) & \text{if } s \in S_C \wedge t[p] \neq t[q] \text{ where } \{p,q\} = L_C(s) \end{cases}$$

The construction procedure of a CA is defined in Algorithm 2. Its parameters are the automaton M that has been constructed so far, the set of partitions P and the current state s. Additionally, parameter E contains the pairs of positions where the subterms are known to be equal, and similarly N is the set of pairs that are known to be different. Lastly, a selection function SELECT is used to define the strategy for choosing the next positions to compare.

The partitions in P for which a pair $\{p,q\}$ of positions is known to be different are removed as these can not be consistent. The remaining partitions form the set P'. To denote the remaining work concisely we introduce the notation $\subseteq\in$ for the composition of $\subseteq$ and $\in$; formally $A \subseteq\in B$ iff $\exists C \in B : A \subseteq C$. Each pair of E that has already been compared is removed from work. The condition on line 4 checks whether there are no choices left to be made. If this is the case then all partitions in P' are consistent by construction and the labelling function is set to yield the partitions P'.

Otherwise, a pair $\{p,q\}$ of positions in work is chosen by the SELECT function and two outgoing transitions are created. A ✓-transition is created that is taken during evaluation whenever the subterms at positions p and q are equal and this information is recorded in E. Otherwise, the fact that these are not equal is recorded in N and a corresponding ✗-transition is created.

Algorithm 2 Given a set of partitions $P = \{P_1, \ldots, P_n\}$ then CONSTRUCT-CA(P, SELECT) computes a CA using CONSTRUCT-CA(P, SELECT, $(\emptyset, \emptyset, \emptyset, s_0), s_0, \emptyset, \emptyset$) that can be used to evaluate the consistent partitions using EVAL-CA.

```
 1: procedure CONSTRUCT-CA($P$, SELECT, $M$, $s$, $E$, $N$)
 2:     $P' := \{P_i \in P \mid \neg\exists C \in P_i : \exists\{p,q\} \in N : p,q \in C\}$
 3:     work $:= \{\{p,q\} \in \mathbb{P}^2 \mid \{p,q\} \subseteq\in P_i \wedge P_i \in P'\} \setminus E$
 4:     if work $= \emptyset$ then
 5:         $M := M[S_F := (S_F \cup \{s\}), L_F := L_F[s \mapsto P']]$
 6:     else
 7:         $\{p,q\} :=$ SELECT(work)
 8:         $M := M[S_C := (S_C \cup \{s\}), L_C := L_C[s \mapsto \{p,q\}]]$
 9:         $M :=$ CONSTRUCT-CA($P$, SELECT, $M[\delta := \delta[(s,\checkmark) \mapsto s']], s', E \cup \{\{p,q\}\}, N$)    where $s'$ is a fresh unbranded state w.r.t. $M$.
10:         $M :=$ CONSTRUCT-CA($P$, SELECT, $M[\delta := \delta[(s,\text{✗}) \mapsto s']], s', E, N \cup \{\{p,q\}\}$)    where $s'$ is a fresh unbranded state w.r.t. $M$.
11:     return $M$
```

The consistency automata obtained from this construction are not optimal, but later on we show how some redundancies can be removed.

4.3 Proof of Correctness

We show the correctness of the construction and evaluation of a CA as defined in Theorem 17. In the following statements let $P = \{P_1, \ldots, P_n\}$ be a set of partitions where each partition is a finite set of finite consistency classes and let SELECT : $2^{\mathbb{P}^2} \to \mathbb{P}^2$ be any selection function such that SELECT(work) $\in$ work for all non-empty work $\subseteq 2^{\mathbb{P}^2}$. For the termination of the construction procedure we can show that the number of choices in work strictly decreases at each recursive call. Again, the complete proofs are present in the appendix.

▶ **Lemma 15.** *The procedure* CONSTRUCT-CA$(P, \textsc{Select})$ *terminates.*

For the construction procedure we can show that for parameter s it holds that $s \notin S$ as a precondition. Therefore, we can use $\mathsf{work}(s) : S \to 2^{\mathbb{P}}$, $E(s) : S \to 2^{\mathbb{P}^2}$ and $N(s) : S \to 2^{\mathbb{P}^2}$ to denote the values of work, E and N respectively during the recursive call of CONSTRUCT-CA$(P, \textsc{Select}, M, s, E, N)$. For the termination of the evaluation procedure we can show that $\mathsf{work}(s)$ strictly decreases for the visited states s.

For the proof of partial correctness we show a relation between the pairs in $E(s)$ and $N(s)$ and the comparisons performed in the evaluation function. First, we define for a term $t \in \mathbb{T}_\Sigma$ and parameters $E, N \subseteq 2^{\mathbb{P}^2}$ the notion of *consistency* where t is consistent w.r.t. E and N, denoted by $(E, N) \models t$, iff:

- $\forall \{p, q\} \in E : t[p] = t[q]$, and
- $\forall \{p, q\} \in N : t[p] \neq t[q]$

A consistency automaton $M = (S, \delta, L, s_0)$ is *well-formed* iff for all terms $t \in \mathbb{T}_\Sigma$ and all recursive calls EVAL-CA(M, t, s_0) = EVAL-CA(M, t, s_n) it holds that that $(E(s_n), N(s_n)) \models t$.

▶ **Lemma 16.** *Let $M = (S, \delta, L, s_0)$ be the result of* CONSTRUCT-CA$(P, \textsc{Select})$*. Then M is well-formed.*

Finally, we can show the correctness of using consistency automata to evaluate the consistency of a given term w.r.t. partitions in P.

▶ **Theorem 17.** *Let $M = (S, \delta, L, s_0)$ be the result of* CONSTRUCT-CA$(P, \textsc{Select})$ *then*

- *for all terms $t \in \mathbb{T}_\Sigma$ we have $P' =$* EVAL-CA(M, s_0, t) *for some $P' \subseteq 2^{2^{\mathbb{P}}}$, and*
- *for all $P_j \in P$ it holds that $P_j \in P'$ iff the term t is consistent w.r.t. P_j.*

Proof. We have already shown termination of the construction procedure in Lemma 15. Let P' be the set of partitions returned by EVAL-CA(M, t, s_0), let $P_i \in P$ be any partition and EVAL-CA(M, t, s_0) = EVAL-CA(M, t, s_n) for some final state $s_n \in S_F$. By Lemma 16 it holds for all $\{p, q\} \in E(s_n)$ that $t[p] = t[q]$ and for all $\{p, q\} \in N(s_n)$ that $t[p] \neq t[q]$.

$\Longrightarrow$ **)** Assume that $P_j \in P'$. For all p, q such that $\{p, q\} \subseteq\in P_j$ it holds that $\{p, q\} \in E(s_n)$ as $\mathsf{work}(s_n)$ is equal to $\emptyset$ for s_n to become a final state in the construction. Therefore, for all $p, q \in C$ for consistency class $C \in P_j$ it holds that $t[p] = t[q]$ and as such t is consistent w.r.t. P_j.

$\Longleftarrow$ **)** Assume that term t is consistent w.r.t. P_j. Proof by contradiction, assume that $P_j \notin P'$. As such, there is a position pair $\{p, q\} \subseteq\in P_j$ such that $\{p, q\} \in N(s_n)$. However, then it follows that $t[p] \neq t[q]$, from which we conclude that t can not be consistent w.r.t. P_j. ◀

4.4 Efficiency

Given a CA M and a term t we define the *evaluation depth*, denoted by $\mathrm{ED}(M, t)$, as the number of recursive EVAL-CA calls made to reach the final state. The size, denoted by $|M|$, is given by the number of states of M. The number of transitions is omitted as each non-final state has exactly two outgoing transitions. We define a notion of relative efficiency that compares the evaluation depth of two automata for all input terms.

▶ **Definition 18.** *Given two consistency automata $M = (S, \delta, L, s_0)$ and $M' = (S', \delta', L', s'_0)$ for a set of consistency partition P. We say that $M \preceq M'$ iff for all terms $t \in \mathbb{T}_\Sigma$ it holds that $ED(M, t) \leq ED(M', t)$.*

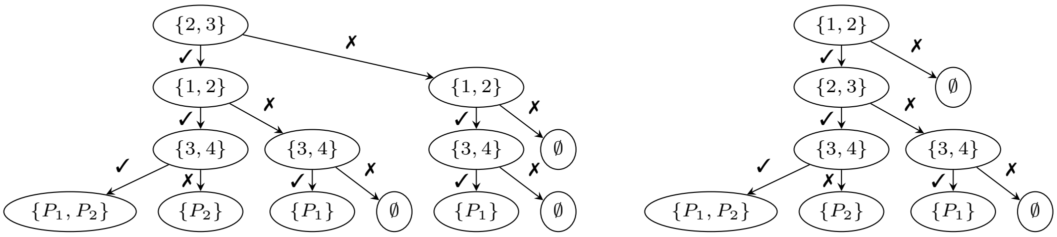


Figure 3 Two CA for the partitions $P_1 = \{\{1,2\},\{3,4\}\}$ and $P_2 = \{\{1,2,3\}\}$. The CA on the left chooses $\{2,3\}$ first. However, as shown on the right selecting $\{1,2\}$ first removes both partitions, and leads to a smaller CA.

We present two ways to improve the (time and space) efficiency of consistency automata. First of all, the selection function used for construction influences the relative efficiency and size of the resulting CA as shown in Figure 3.

Changing the selection function does not necessarily result in the most efficient (equivalent) CA. If we consider Figure 2 again, we can observe that the resulting automaton is not optimal, despite being the smallest w.r.t. the selection function, because some of the (final) states are not reached during evaluation of any given term. For example, the final state labelled with $\{P_1, P_2\}$ is not reachable, because any term $t \in \mathbb{T}_\Sigma$ that satisfies $t[1] = t[2]$ and $t[1] = t[3]$ can not have that $t[2] \neq t[3]$ by the transitivity of term equality. Removing the redundant states reduces the number of states and yields a relatively more efficient CA.

Given a CA $M = (S, \delta, L, s_0)$ and a non-final state $s \in S_C$ we give the following conditions for its redundancy. Namely, whenever for all terms $t \in \mathbb{T}_\Sigma$ that satisfy $(E(s), N(s)) \models t$ it holds that $t[p] = t[q]$, for $\{p, q\} := L_C(s)$, s is said to be ✓-redundant. Similarly, whenever we can show that all consistent terms satisfy $t[p] \neq t[q]$ then s is ✗-redundant. Redundant states can be removed from the automata without affecting the correctness of its evaluation in the following way.

A state s that is ✓-redundant can be *removed* by updating δ such that the incoming transition $\delta(r, a) = s$, for some $r \in S$ and $a \in \{✓, ✗\}$, is updated to $\delta(s, ✓)$. A similar transformation of δ can be applied for states that are ✗-redundant using $\delta(s, ✗)$. We can observe that such a removal results in a relatively more efficient CA and that the size of the CA is reduced by the number of states in the ✓-branch (or ✗-branch) respectively if states unreachable by the transition relation are removed. Next, we prove that removal does not influence the correctness of evaluation.

▶ **Lemma 19.** *Let $M = (S, \delta, L, s_0)$ be any CA that is well-formed. Then the resulting CA M' where a ✓-redundant or ✗-redundant state $v \in S$ is removed remains well-formed.*

Using Lemma 19 and the fact that removing redundant states does not change the labelling of any state we have shown that $\textsc{eval-ca}(M, s_0, t) = \textsc{eval-ca}(M', s_0, t)$ for all t.

If we consider Figure 2 again it follows from transitivity that the left indicated state is ✓-redundant and the right indicated state ✗-redundant. If the indicated states are removed then all states of the resulting CA are reachable, which could be argued for as a form of local optimum. For transitivity it is relatively straightforward to construct a procedure to identify and remove these states. However, it would be more interesting to devise a method that determines all redundant states. Additional redundancies follow from the ordering of positions. For example, a term can never be equal to any of its subterms. Defining this complete procedure is left as future work.

4.5 Worst-case Complexity

We establish several upper and lower bounds on the space and time complexity for consistency automata. The *maximum evaluation depth*, given by $\max_{t \in \mathbb{T}_\Sigma}(\mathrm{ED}(M,t))$, is the measurement for time complexity, where only the number of comparisons is counted. Finally, we only consider the time-optimal automaton M for the complexity analysis, which is the CA where the maximal evaluation depth is minimal from all possible selection functions.

For the time complexity of consistency automata we can show that each pair of positions is compared at most once. Let n be the number of unique position pairs in the given partitions, where each pair of positions is counted at most once. It can be shown that the worst-case time complexity of the consistency automata evaluation is tightly bounded by $\mathcal{O}(n)$ and its corresponding size is $\mathcal{O}(2^n)$. This follows essentially from the size of work for the first call to the construction procedure, which reduces in each recursive call. The given bounds are also tight as we can construct an example where the maximum evaluation depth requires exactly n comparisons.

5 Adaptive Non-linear Pattern Matching Automata

We have shown in Lemma 13 that a naive matching algorithm for non-linear patterns can be obtained by using a linear matching function followed by a consistency check. In that case we have to check the consistency of all partitions returned by the linear matching function. However, as shown in the following example overlapping patterns can unify with the same prefix, but no term can match both patterns at the same time.

Consider the patterns: $\ell_1 : f(x,x)$ and $\ell_2 : f(a,b)$. After renaming we obtain the following pairs $(f(\omega_1,\omega_2),\{\{1,2\}\})$ and $(f(a,b)),\{\emptyset\})$. Now, the resulting APMA has a final state labelled with both patterns as shown in Figure 4a. We can observe that the consistency check of positions one and two always yields false whenever the evaluation of a term ends up in the final state labelled with $\{\ell_1,\ell_2\}$, because terms a and b are not equal. Therefore, this comparison would be unnecessary.

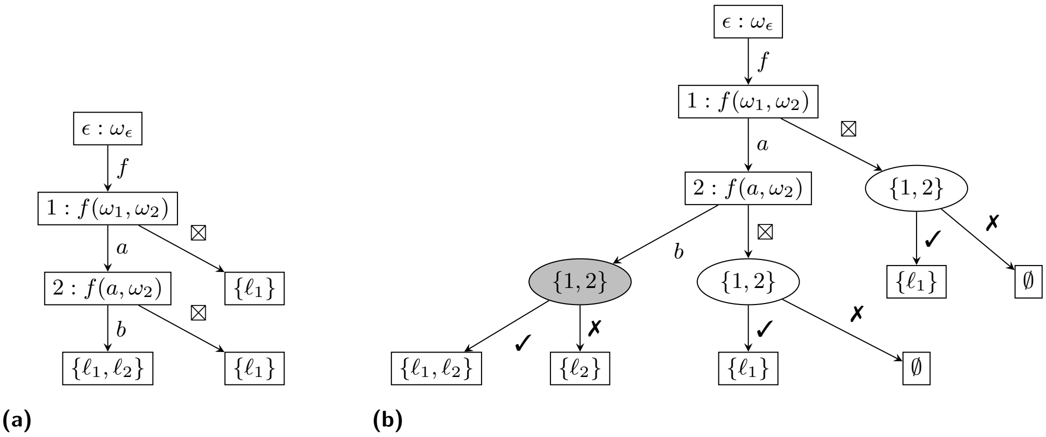


Figure 4 The resulting APMA shown on the left and the corresponding ANPMA with a grey ✗-redundant state on the right.

We could also consider an alternative where the consistency phase is performed first, but then we have the problem that whenever the given term is consistent w.r.t. partition $\{1,2\}$ that matching on $f(a,b)$ is avoided. To enable these kind of efficiency improvements, we propose a combination of APMAs and CAs to obtain a matching automaton for non-linear patterns called *adaptive non-linear pattern matching automata*, abbreviated as ANPMAs. The result is an automaton that has three kinds of states; matching states of APMAs, consistency states of CAs and final states, and two transition functions; one for matching states and one for consistency states.

▶ **Definition 20.** *An adaptive non-linear pattern matching automaton (ANPMA) is a tuple* (S, δ, L, s_0) *with*

- $S = S_M \uplus S_C \uplus S_F$ *is a set of states where* S_M *is a set of matching states,* S_C *is a set of consistency states and* S_F *is a set of final states;*
- $\delta = \delta_F \uplus \delta_C$ *is a partial transition function with* $\delta_F : S_M \times \mathbb{F} \rightharpoonup S$ *and* $\delta_C : S_C \times \{\checkmark, \text{✗}\} \to S$*;*
- $L = L_M \uplus L_C \uplus L_F$ *is a state labelling function with* $L_M : S_M \to \mathbb{P}$*,* $L_C : S_C \to \mathbb{P}^2$ *and* $L_F : S_F \to 2^{\mathbb{T}}$*;*
- $s_0 \in S_M$ *is the initial state.*

We only consider ANPMAs that have a tree structure rooted in s_0. Given an ANPMA $M = (S, \delta, L, s_0)$ and a term t the procedure $\textsc{Match}(M, s_0, t)$ below defines the evaluation of the ANPMA. It is essentially the combination of the evaluation functions for the APMA and CA depending on the current state.

$$\textsc{Match}\text{ANPMA}(M, t, s) = \begin{cases} L_M(s) & \text{if } s \in S_F \\ \textsc{Match}\text{ANPMA}(M, t, \delta_F(s, f)) & \text{if } s \in S_M \wedge \delta(s, f) \neq \bot \\ \textsc{Match}\text{ANPMA}(M, t, \delta_F(s, \boxtimes)) & \text{if } s \in S_M \wedge \delta(s, \boxtimes) \neq \bot \wedge \delta(s, f) = \bot \\ \emptyset & \text{if } s \in S_M \wedge \delta(s, \boxtimes) = \delta(s, f) = \bot \\ \textsc{Match}\text{ANPMA}(M, t, \delta_C(s, \checkmark)) & \text{if } s \in S_C \wedge t[p] = t[q] \\ \textsc{Match}\text{ANPMA}(M, t, \delta_C(s, \text{✗})) & \text{if } s \in S_C \wedge t[p] \neq t[q] \end{cases}$$

$$\text{where } f = \mathsf{head}(t[L_M(s)]) \text{ and } \{p, q\} = L_C(s)$$

The construction algorithm of the ANPMA is defined in Algorithm 3. It combines the construction algorithm of APMAs (Algorithm 1) and the construction algorithm for CAs (Algorithm 2). The parameters that remain the same value during the recursion are the original set $\mathcal{L}$, the result of renaming $\mathcal{L}_r$ and the selection function $\textsc{Select}$. Next, we have the ANPMA M, a state s and finally the current prefix pref similar to the APMA construction and the sets of position pairs E and N as in the consistency automata construction.

First we remove the terms that do not have to be considered anymore. These are the elements $i : (\ell, P)$ from $\mathcal{L}_r$ such that P is inconsistent due to the pairs in N and pref does not unify with ℓ. Obtaining work for both types of choices is almost the same as before. However, for workC we have added the condition that the positions must be defined in the prefix to ensure that these positions are indeed defined when evaluating a term. The termination condition is that both workF and workC are empty, or that the set of patterns $\mathcal{L}_r'$ has become empty. The latter can happen when the inconsistency of two positions removes a pattern, which could still have other positions to be matched.

The function $\textsc{Select}$ is a function that chooses a position from workF or a pair of positions from workC. Its result determines the kind of state that s becomes and as such also the outgoing transitions. If a position is selected then s will become a matching state and the construction continues as in Algorithm 1. Otherwise, similar to Algorithm 2 two fresh states and two outgoing transition labelled with ✓ and ✗ are created, after which the parameters E and N are updated.

Algorithm 3 Given a set of patterns $\mathcal{L}$ and a renamed set of patterns $\mathcal{L}_r$, this algorithm computes an ANPMA for $\mathcal{L}$. Initially it is called with $M = (\emptyset, \emptyset, \emptyset, s_0)$, the initial state $s = s_0$, the prefix $\mathsf{pref} = \omega_\epsilon$, and $E = N = \emptyset$.

```
procedure ConstructANPMA($\mathcal{L}, \mathcal{L}_r, \textsc{Select}, M, s, \mathsf{pref}, E, N$)
    $\mathcal{L}'_r := \{i : (\ell, P) \in \mathcal{L}_r \mid \ell$ unifies with $\mathsf{pref} \wedge \neg\exists C \in P : \exists\{p,q\} \in N : p, q \in C\}$
    $\mathsf{workF} := \mathcal{F}(\mathsf{pref})$
    $\mathsf{workC} := \{\{p,q\} \in \mathbb{P}^2 \mid \{p,q\} \subseteq\in P_i \wedge (i : \ell, i : P_i) \in \mathcal{L}'_r \wedge$ $\mathsf{pref}[p]$ and $\mathsf{pref}[q]$ are defined$\} \setminus E$
    if ($\mathsf{workF} = \emptyset$ and $\mathsf{workC} = \emptyset$) or ($\mathcal{L}'_r = \emptyset$) then
        $M := M[S_F := S_F \cup \{s\}, L := L[s \mapsto \{i : \ell \in \mathcal{L} \mid i : \ell' \in \mathcal{L}'\}]]$
    else
        $\mathsf{next} := \textsc{Select}(\mathsf{workF}, \mathsf{workC})$
        if $\mathsf{next} = \mathsf{pos}$ for some position $\mathsf{pos}$ then
            $M := M[S_M := (S_M \cup \{s\}), L_M := L_M[s \mapsto \mathsf{pos}]]$
            $F := \{f \in \mathbb{F} \mid \exists(i : (\ell, P)) \in \mathcal{L}'_r : \mathsf{head}(\ell[\mathsf{pos}]) = f\}$
            for $f \in F$ do
                $M := M[\delta := \delta[(s, f) \mapsto s']]$ where $s'$ is a fresh unbranded state w.r.t. $M$
                $M := \textsc{ConstructANPMA}(\mathcal{L}, \mathcal{L}_r, \textsc{Select}, M, s', \mathsf{pref}[\mathsf{pos}/f(\omega_{\mathsf{pos}.1}, \ldots, \omega_{\mathsf{pos}.\mathsf{ar}(f)})], E, N)$
            if $\exists(i : (\ell, P)) \in \mathcal{L}'_r : \exists \mathsf{pos}' \leq \mathsf{pos} : \mathsf{head}(\ell[\mathsf{pos}']) \in \mathbb{V}$ then
                $M := M[\delta := \delta[(s, \boxtimes) \mapsto s']]$ where $s'$ is a fresh unbranded state w.r.t. $M$
                $M := \textsc{ConstructANPMA}(\mathcal{L}, \mathcal{L}_r, \textsc{Select}, M, s', \mathsf{pref}[\mathsf{pos}/\boxtimes], E, N)$
        else if $\mathsf{next} = \{p,q\}$ for some pair $\{p,q\} \in \mathbb{P}^2$ then
            $M := M[S_C := (S_C \cup \{s\}), L_C := L_C[s \mapsto \{p,q\}]]$
            $M := \textsc{ConstructANPMA}(\mathcal{L}, \mathcal{L}_r, \textsc{Select}, M[\delta_C := \delta_C[(s, ✓) \mapsto s']], s', \mathsf{pref}, E \cup \{\{p,q\}\}, N)$
          where $s'$ is an unbranded state w.r.t. $M$.
            $M := \textsc{ConstructANPMA}(\mathcal{L}, \mathcal{L}_r, \textsc{Select}, M[\delta_C := \delta_C[(s, ✗) \mapsto s']], s', \mathsf{pref}, E, N \cup \{\{p,q\}\})$
          where $s'$ is an unbranded state w.r.t. $M$.
    return $M$
```

5.1 Correctness

The ANPMA construction algorithm yields an ANPMA that is suitable to solve the matching problem for non-empty finite sets of (non-linear) patterns. This can be shown by combining the efforts of Theorem 9 and Theorem 17 and the proofs can be found in the appendix.

Let $\mathcal{L}$ be a finite non-empty indexed family of (non-linear) patterns and let $(\mathcal{L}_r, P) = \mathsf{rename}(\mathcal{L})$. Suppose that $\textsc{Select} : 2^{\mathbb{P}} \times 2^{\mathbb{P}^2} \to \mathbb{P} \uplus \mathbb{P}^2$ is any function such that for all sets of positions workF and position pairs workC we have that $\textsc{Select}(\mathsf{workF}, \mathsf{workC}) \in \mathsf{workF} \uplus \mathsf{workC}$.

We extend the auxiliary definitions for APMA as follows. A *path* to s_n is a sequence with both types of labels $(s_0, a_0), \ldots, (s_{n-1}, a_{n-1}) \in S \times (\mathbb{F}_\boxtimes \uplus \{✓, ✗\})$ such that $\delta(s_i, a_i) = s_{i+1}$ for all $i < n$. A position p is called *visible* for state s iff there is a pair (s_i, a_i) in $\mathsf{path}(s)$ such that $L(s_i).i = p$ for some $1 \leq i \leq \mathsf{ar}(f_i)$ or $L(s) = \epsilon$. A state s is *top-down* iff $s \in S_M$ and $L_M(s)$ is visible or $s \in S_C$ and both positions in $L_C(s)$ are visible. State s is *canonical* iff there are no two matching states in $\mathsf{path}(s)$ that are labelled with the same position. Finally we say that an ANPMA is *well-formed* iff $L(s_0) = \epsilon$, and all states are top-down and canonical.

▶ **Lemma 21.** *The procedure* $\textsc{ConstructANPMA}(\mathcal{L}_r, P, \textsc{Select}, (\emptyset, \emptyset, \emptyset, s_0), s_0, \omega_\epsilon, \emptyset, \emptyset)$ *terminates and yields a well-formed ANPMA.*

Let $M = (S, \delta, L, s_0)$ be the ANPMA resulting from $\textsc{ConstructANPMA}(\mathcal{L}, \textsc{Select})$. Let $t \in \mathbb{T}_\Sigma$ be a term and $\mathcal{L}_t$ be equal to $\{i : \ell \in \mathcal{L} \mid \ell \leq t\}$. For every state $s \in S$ we define $\mathcal{L}(s)$ to be equal to $\{i : \ell \in \mathcal{L} \mid i : (\ell', P) \in \mathcal{L}'_r(s_i)\}$. We show that the evaluation algorithm on M satisfies a number of invariants.

▶ **Lemma 22.** *For all* $s \in S$ *such that* $\textsc{MatchANPMA}(M, t, s_0) = \textsc{MatchANPMA}(M, t, s)$ *it holds that: (a)* $(E(s_i), N(s_i)) \models t$*, (b)* $\mathcal{L}_t \subseteq \mathcal{L}(s)$ *and (c) if* $s \in S_F$ *then* $L(s_f) = \mathcal{L}_t$*.*

▶ **Lemma 23.** *If $\mathcal{L}_t = \emptyset$ then* $\textsc{MatchANPMA}(M, t, s_0) = \emptyset$.

▶ **Theorem 24.** *Then* $\lambda t.\textsc{MatchANPMA}(M, t, s_0)$ *is a matching function for* $\mathcal{L}$.

Proof. If $\mathcal{L}_t$ is empty then by Lemma 23 we get that $\textsc{MatchANPMA}(M, t, s_0) = \emptyset = \mathcal{L}_t$ as required. Otherwise, we have that $\textsc{MatchANPMA}(M, t, s_0) = \textsc{MatchANPMA}(M, t, s_f)$ for some final state s_f. Then by the definition of MatchANPMA and Lemma 22 we conclude $\textsc{MatchANPMA}(M, t, s_0) = \textsc{MatchANPMA}(M, t, s_f) = L(s_f) = \mathcal{L}_t$. ◀

5.2 Strategy

The notion of ✓-redundancy (and ✗-redundancy) that we defined for CA can be easily extended to ANPMA. However, we can even identify more redundant states by considering the prefix for a given state s. Namely, for a state s labelled with a pair of positions $\{p, q\}$, given by $L_C(s)$, we can observe that s is ✗-redundant whenever $\mathsf{pref}[p]$ does not unify with $\mathsf{pref}[q]$, because if they do not unify then they can not be equal. Consider the patterns $\ell_1 : f(x, x)$ and $\ell_2 : f(a, b)$ again, we show the resulting ANPMA in Figure 4b.

6 Conclusion and Future Work

In this paper, we presented a formal proof for the correctness of APMAs. Furthermore, we introduced CAs as a deterministic automaton to perform the consistency checking, from which some redundant states could be removed by taking the previous choices into account. These two automata are then combined to obtain an ANPMA which could be evaluated by only performing comparisons and taking the corresponding outgoing edge.

ANPMAs offer a formal platform to study the relations between linear pattern matching and consistency checking. There are still some questions that have arisen from this work. As mentioned in the previous section, the current ANPMA construction algorithm can contain redundant states. For the moment it is still unclear how to detect which states are redundant. An interesting direction for future research is to optimise the ANPMA construction algorithm that creates an optimal ANPMA on the fly.

Secondly we did not study selection functions in this work. All three automaton construction algorithms in this paper are parametrised in a selection function that decides for each node what will happen next. We have shown that all constructions yield correct automata for any selection function, with the side note that the selection indeed yields an element from its input set. The size of all three kinds of automata depends heavily on the selection function that is used. For APMAs some selection functions have already been studied in [8].

Thirdly it would be interesting to implement this approach. This work is a theoretical approach to ultimately obtain micro-optimisations in for example term rewriting. Many formalisms do not support non-linear patterns and as discussed in the introduction, many solutions to the pattern matching problem do not support it. It would be interesting to find out in practise whether exploiting $\mathcal{O}(1)$ term equality checking is worth the extra overhead that the ANPMA approach carries with it.

References

1 L. Cardelli. Compiling a functional language. In *LISP and Functional Programming*, pages 208–217. ACM, 1984.

2 J. Christian. Flatterms, discrimination nets, and fast term rewriting. *Journal of Automated Reasoning*, 10(1):95–113, 1993. doi:10.1007/BF00881866.

3 Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *ICFP*, pages 26–37. ACM, 2001.

4 Albert Gräf. Left-to-right tree pattern matching. In *RTA*, volume 488 of *LNCS*, pages 323–334. Springer, 1991.

5 P. Graf. Substitution tree indexing. In J. Hsiang, editor, *Rewriting Techniques and Applications*, pages 117–131, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

6 W. McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9(2):147–167, October 1992. doi:10.1007/BF00245458.

7 R. Sekar, I.V. Ramakrishnan, and A. Voronkov. Chapter 26 - term indexing. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, pages 1853–1964. North-Holland, Amsterdam, 2001. doi:10.1016/B978-044450813-3/50028-X.

8 R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. *SIAM Journal of Computing*, 24(6):1207–1234, 1995. doi:10.1137/S0097539793246252.

9 M. van Weerdenburg. An account of implementing applicative term rewriting. *Electronic Notes in Theoretical Computer Science*, 174(10):139–155, 2007.

10 A. Voronkov. The anatomy of vampire implementing bottom-up procedures with code trees. *Journal of Automated Reasoning*, 15(2):237–265, 1995.

A Proof of Lemma 4

Proof. The set $\mathcal{L}$ is finite, so all non recursive statements terminate. The for loop in particular treats finitely many function symbols from F. Finally, we show that the prefixes of the recursive calls are ordered by the matching ordering $<$. The algorithm Construct realises that the prefix is only defined for defined positions of patterns in $\mathcal{L}$. Hence this ordering is well-founded on the recursive calls and the construction terminates.

Upon termination the result M is indeed an APMA. For every function symbol in F exactly one transition is created and at most one $\boxtimes$-transition is created, so δ is a partial mapping. Since the target states of these transitions are fresh we have that δ is injective. Moreover there is no transition to s_0 since the algorithm is initially called with s_0. Hence M is an APMA.

We check that M is well-formed. By construction we have $L(s_0) = \epsilon$ since the construction procedure is called with the prefix ω_ϵ. Let s be an arbitrary non-final state and consider the stage of the construction algorithm Construct($\mathcal{L}$, Select, M, s, pref). A position label $p.i$ is only chosen if it occurs in the fringe of pref. Therefore there must have been a state labelled with p where the variable $\omega_{p.i}$ was put in the prefix, so s must be top-down. Lastly s is canonical because once a position p is chosen, it cannot be chosen again since the variable ω_p is replaced by an element of $\mathbb{F}_\boxtimes$ in the prefix. Hence M meets all requirements for well-formedness. ◀

B Proof of Lemma 5

Proof. First observe that $\mathcal{L}(s)$ is non-empty for all states s. Let s be a final state.

a) Since $L(s) = \mathcal{L}(s)$ and $\mathcal{L}(s)$ is non-empty the claim holds.

b) The prefix $\mathsf{pref}(s)$ is ground for final states s because the construction only creates final states if $\mathsf{pref}(s)$ has no variables.

c) By construction we have $L(s) = \{\ell \in \mathcal{L} \mid \ell \text{ unifies with } \mathsf{pref}(s)\}$. Since $\mathsf{pref}(s)$ is ground we have that for all $\ell \in L(s)$ that $\ell \leq \mathsf{pref}(s)$.

d) Let $\ell \in \mathcal{L}$. The following invariant holds for the construction algorithm: for all matching states s', if $\ell \in \mathcal{L}(s')$ then there is a pair (s'', f) such that $\delta(s', f) = s''$ and $\ell \in \mathcal{L}(s'')$. ◀

C Proof of Lemma 6

Proof. By induction on the length of $\mathsf{path}(s)$. If there are no pairs in $\mathsf{path}(s)$ then it must be that $s = s_0$. For the initial state we have $\mathcal{L}_t \subseteq \mathcal{L} = \mathcal{L}(s_0) = \mathcal{L}(s)$, so the base case holds.

Let s be an arbitrary state and suppose that $\textsc{Match}(M, t, s_0) = \textsc{Match}(M, t, s)$ and assume the induction hypothesis $\mathcal{L}_t \subseteq \mathcal{L}(s)$. Now suppose $\textsc{Match}(M, t, s) = \textsc{Match}(M, t, s')$ where $s' = \delta(s, f)$ for some $f \in \mathbb{F}_{\boxtimes}$ and let $L(s) = p$.

- If $f \in \mathbb{F}$ then $\mathsf{pref}(s') = \mathsf{pref}(s)[p/f(\omega_{p.1}, \ldots, \omega_{p.\mathsf{ar}(f)})]$. By definition of $\textsc{Match}$ we know that $\mathsf{head}(t[p]) = f$.

  Let $\ell \in \mathcal{L}_t$. We show that ℓ unifies with $\mathsf{pref}(s')$. We know that $\ell \leq t$ by assumption. From the induction hypothesis it follows that ℓ unifies with $\mathsf{pref}(s)$. So there is a term u such that $\ell \leq u$ and $\mathsf{pref}(s) \leq u$. Then we distinguish two cases.
  - If $\ell[p']$ is a variable for some $p' \sqsubseteq p$ then ℓ unifies with $\mathsf{pref}(s')$.
  - If $\mathsf{head}(\ell[p])$ is a function symbol then by $\ell \leq t$ it must be that $\mathsf{head}(\ell[p]) = f$, so ℓ unifies with $\mathsf{pref}(s')$.
- If $f = \boxtimes$ then $\mathsf{pref}(s') = \mathsf{pref}(s)[p/\boxtimes]$. By definition of $\textsc{Match}$ we know that $\delta(s, \mathsf{head}(t[p]))$ is undefined.

  From the construction algorithm we then know that there is no pattern $\ell \in \mathcal{L}(s)$ such that $\mathsf{head}(\ell[p]) \in \mathbb{F}$ and there is at least one pattern $\ell \in \mathcal{L}(s)$ such that $\ell[p']$ is a variable for some position $p' \sqsubseteq p$.

  Let $\ell \in \mathcal{L}_t$. By induction hypothesis we know that ℓ unifies with $\mathsf{pref}(s)$. We show that ℓ unifies with $\mathsf{pref}(s')$ by showing that $\ell[p'] = \omega_{p'}$ for some position $p' \sqsubseteq p$.
  - Suppose that $\ell[p]$ exists. Since $\ell \leq t$ and $\mathsf{head}(t[p]) \neq \mathsf{head}(\ell[p])$ it must be that $\ell[p] = \omega_p$.
  - Suppose that $\ell[p]$ does not exist. Pick the lowest position p' such that $p' \sqsubset p$ and $\ell[p']$ exists and assume for a contradiction that $\mathsf{head}(\ell[p']) = f$ for some function symbol f. Then it must be that $\mathsf{head}(\mathsf{pref}(s)[p']) = f$ by the induction hypothesis. However, $\mathsf{pref}(s)[p]$ exists and from $p' \sqsubset p$ it follows that $\ell[p]$ has subterms of the function symbol f, which contradicts the assumption that p' is the lowest position strictly higher than p. So $\ell[p'] = \omega_{p'}$. ◀

D Proof of Lemma 7

Proof.

a) We show that $\textsc{Match}(M, t, s_0) \neq L(s)$ for all final states s. Let s_f be an arbitrary final state and pick some pattern $\ell \in L(s_f)$. By assumption $\ell \not\leq t$ and by Proposition 2 it follows that there is a position p and a function symbol $f \in \mathbb{F}$ such that $\mathsf{head}(\ell[p]) = f$ and $\mathsf{head}(t[p]) \neq f$. By Lemma 5 it must be that $\mathsf{head}(\mathsf{pref}(s)[p]) = f$, by which there must be a pair $(s_i, f) \in \mathsf{path}(s)$. Since $\textsc{Match}$ is a function we have $\textsc{Match}(M, t, s_0) = \textsc{Match}(M, t, s_i) = \textsc{Match}(M, t, s_f)$. However, by definition of $\textsc{Match}$ we know that $\mathsf{head}(t[p]) = f$, which contradicts the assumption that $l \in L(s_f)$.

b) Let $\ell \in \mathcal{L}_t$. We prove that for all s such that $\textsc{Match}(M, s_0, t) = \textsc{Match}(M, s, t)$, we have that $\delta(s, \mathsf{head}(t[L(s)]))$ or $\delta(s, \boxtimes)$ is defined.

Suppose that $\textsc{Match}(M, s_0, t) = \textsc{Match}(M, s, t)$. From Lemma 6 it follows that $\ell \in \mathcal{L}(s)$. If $\mathsf{head}(\ell[L(s)]) = f$ for some function symbol f then the construction algorithm created an f-transition to a new state, by which $\delta(s, f)$ exists. Otherwise if $\mathsf{head}(\ell[L(s)])$ does not exist then by $\ell \leq t$ there must be a position $p \sqsubset L(s)$ such that $\ell[p] = \omega_p$. In that case a $\boxtimes$-transition is created and hence $\delta(s, \boxtimes)$ exists.

By definition of $\textsc{Match}$ we then have that $\textsc{Match}(M, s_0, t)$ cannot yield the empty set, so it must terminate in a final state. ◀

E Proof of Lemma 8

Proof. Since $L(s_f) = \mathcal{L}(s_f)$ we know that $\mathcal{L}_t \subseteq L(s_f)$ by Lemma 6. It only remains show that $L(s_f) \subseteq \mathcal{L}_t$. Since s_f is a final state we have that $\mathsf{pref}(s_f)$ is ground; therefore $L(s_f) = \{\ell \in \mathcal{L} \mid \ell \leq \mathsf{pref}(s_f)\}$. Suppose for a contradiction that there is some $\ell \leq \mathsf{pref}(s_f)$ such that $\ell \not\leq t$. Then there is a position p such that $\mathsf{head}(\ell[p]) \in \mathbb{F}$ and $\mathsf{head}(t[p]) \neq \mathsf{head}(\ell[p])$. We have $\mathsf{head}(\ell[p]) = \mathsf{head}(\mathsf{pref}(s_f)[p])$ by assumption. So, there is a pair (s_i, f_i) in $\mathsf{path}(s_f)$ such that $L(s_i) = p$. By definition of MATCH we then have $\mathsf{head}(t[p]) = f_i = \mathsf{head}(\ell[p])$, a contradiction. ◀

F Proof of Lemma 12

Proof. We can show by induction on positions that $t =_\omega \mathsf{rename}_1(t, \epsilon)$ to prove the first statement. For the second statement let $p \in \mathcal{F}(t)$. First, we can show that $t'[p] = \mathsf{rename}_1(t[p], p)$ by induction on position p. From $t[p] \in \mathbb{V}$ it follows that $t'[p]$ is equal to ω_p.

For the last property let P be equal to $\{\{p \in \mathbb{P} \mid t[p] = x\} \mid x \in \mathsf{vars}(t)\}$ and let u be an arbitrary term. Assume that u is consistent w.r.t. P and u matches t'. The latter means that there is a substitution σ such that $t'^\sigma = u$. It follows that for all positions $p \in \mathcal{F}(t')$ that $\sigma(t'[p]) = u[p]$. As u is consistent w.r.t. P it means that for all $x \in \mathbb{V}$ and $p, q \in \mathbb{P}$ that if $t[p] = t[q] = x$ then $u[p] = u[q]$. Therefore, we can construct the substitution ρ such that for all $p \in \mathcal{F}(t)$ we assign $u[p]$ to $t[p]$, where the latter is some variable in $\mathsf{vars}(t)$. The observation of consistency above lets us conclude that there is only one such substitution ρ. From $t =_\omega t'$ it follows that $t^\rho = t'^\sigma$ and as such $t^\rho = u$, which means that u matches t.

Otherwise, if u matches t then there is a substitution σ such that $t^\sigma = u$. Let ρ be the substitution such that for all positions $p \in \mathcal{F}(t)$ we assign $\sigma(t[p])$ (which is equal to $u[p]$) to ω_p. As t' is linear it follows that each ω_p is assigned once and thus $\rho(\omega_p) = \sigma(t[p])$ by definition. Again, from $t =_\omega t'$ it follows that $t'^\rho = t^\sigma$ and as such u matches t'. Finally, for all positions p and q such that $t[p] = t[q] = x$ for variable $x \in \mathbb{V}$ it follows that $u[p] = u[q] = \sigma(x)$. We can thus conclude that u is consistent w.r.t. P. ◀

G Proof of Lemma 15

Proof. Consider the pair of positions $\{p, q\}$ that is taken from work at line 7. It is easy to see that $\{p, q\} \notin E$, and $\{p, q\} \notin N$ follows directly from the fact that P' only consists of partitions of which the consistency classes do not contain positions together in a pair of N. Therefore, it follows that in subsequent recursive calls $\{p, q\}$ cannot be in work again as either E or N is extended with $\{p, q\}$ and no elements are ever removed from E or N. Furthermore, the execution of all other statements terminates as $\#(P)$ is finite, which also means that $|E|$ and $|N|$ are finite as inserted pairs satisfy $\{p, q\} \subseteq\in P'$. Finally, the selection function terminates by assumption. ◀

H Proof of Lemma 16

Proof. The recursive calls form an *evaluation series* $(s_0, a_0), \ldots, (s_n, a_n)$ for $s_i \in S$ and $a_i \in \{✓, ✗\}$ for $0 \leq i < n$ such that $\textsc{eval-ca}(M, s_i, t) = \textsc{eval-ca}(M, s_{i+1}, t)$ and $\delta(s_i, a_i) = s_{i+1}$. Let $t \in \mathbb{T}_\Sigma$ be any term. We prove the statement by induction on the length of the evaluation series.

Base case. We have $E(s_0) = N(s_0) = \emptyset$ and as such the statement holds vacuously.

Inductive step. Suppose that the statement holds for $\textsc{eval-ca}(M,t,s_0) = \textsc{eval-ca}(M,t,s)$. Suppose that $\textsc{eval-ca}(M,t,s) = \textsc{eval-ca}(M,t,s')$ where $s' = \delta(s,a)$ for $a \in \{✓,✗\}$ and let $L_C(s) = \{p,q\}$. There are two cases to consider:

- $t[p] = t[q]$ in which case $E(s')$, where s' is equal to $\delta(s,✓)$, is $E(s)$ extended with $\{p,q\}$ and $N(s') = N(s)$.
- Otherwise, $t[p] \neq t[q]$ in which case $N(s')$ is equal to $N(s)$ extended with $\{p,q\}$ and $E(s') = E(s)$.

In both cases $(E(s'), N(s')) \models t$ holds by definition. ◀

I Proof of Lemma 19

Proof. The recursive calls form an *evaluation series* $(s_0,a_0),\ldots,(s_n,a_n)$ for $s_i \in S$ and $a_i \in \{✓,✗\}$ for $0 \leq i < n$ such that $\textsc{eval-ca}(M,s_i,t) = \textsc{eval-ca}(M,s_{i+1},t)$ and $\delta(s_i,a_i) = s_{i+1}$. By well-formedness of M we know, for all terms $t \in \mathbb{T}_\Sigma$ and all evaluation series $(s_0,a_0),\ldots,(s_k,a_k) \in (S \times \{✓,✗\})$ of $\textsc{eval-ca}(M,s_0,t)$, that for all states s_i, with $0 \leq i \leq k$, it holds that $(E(s_i),N(s_i)) \models t$. Now, we only have to consider sequences that contain the state v as the other evaluation sequences remain the same. Consider any such sequence and let u be the state in that sequence such that $\delta(u,a) = v$, for some $a \in \{✓,✗\}$, and let t be an arbitrary term. Note that the initial state can not be removed by this procedure. Let $\{p,q\}$ be the value of $L_C(v)$ then there are two cases to consider:

- v is ✓-redundant. It follows that $t[p] = t[q]$ for $\{p,q\} := L_C(s)$. All sequences such that v occurs in it must contain exactly the pair $(v,✓)$ by definition of ✓-redundancy. We conclude that $(E(u) \cup \{\{p,q\}\}, N(u)) \models t$ holds and the term remains consistent with all extensions to E and N for the remaining states in the sequence.
- s is ✗-redundant. Similarly, with the observation that $(E(v), N(v) \cup \{\{p,q\}\}) \models t$. ◀

J Proof for Lemma 21

Proof. We only show that the recursion terminates. The rest is similar to the proof for Lemma 4, with the additional observation that positions in P are only chosen when they are defined in the prefix. Given the parameters $\mathsf{pref}_1, E_1, N_1$ and $\mathsf{pref}_2, E_2, N_2$ we can fix the ordering:

$$(\mathsf{pref}_1 < \mathsf{pref}_2 \wedge E_1 = E_2 \wedge N_1 = N_2) \vee$$
$$(\mathsf{pref}_1 = \mathsf{pref}_2 \wedge E_1 \subset E_2 \wedge N_1 = N_2) \vee$$
$$(\mathsf{pref}_1 = \mathsf{pref}_2 \wedge E_1 = E_2 \wedge N_1 \subset N_2)\,.$$

The prefixes are again only defined on positions that are defined in patterns of $\mathcal{L}$ and the sets E and N are bounded by a finite product of positions, hence the ordering is well-founded. The recursive calls conform to to this ordering; therefore the recursion terminates. ◀

K Proof for Lemma 22

Proof. Take an arbitrary term t. We prove the first two invariants by induction on the length of $\mathsf{path}(s)$.

Base case, the empty path and as such $s = s_0$. $E(s_0) = N(s_0) = \emptyset$ and $\mathcal{L}_t \subseteq \mathcal{L}$, and $\mathcal{L} = \mathcal{L}(s_0) = \mathcal{L}(s)$, as such the statements hold vacuously.

Inductive step. Let s be an arbitrary state and suppose that the statements hold for $\textsc{Match}\text{ANPMA}(A, t, s_0) = \textsc{Match}\text{ANPMA}(A, t, s)$. Suppose $\textsc{Match}\text{ANPMA}(A, t, s) = \textsc{Match}\text{ANPMA}(A, t, s')$ for some $s' = \delta(s, x)$ such that $x \in (\mathbb{F}_{\boxtimes} \uplus \{✓, ✗\})$. Now, there are two cases to consider:

- $s \in S_C$. Let $\{p, q\}$ be the value of $L_C(s_k)$. Again, there are two cases to consider:
  - $t[p] = t[q]$ in which case $E(s')$ is $E(s) \cup \{p, q\}$ and $N(s') = N(s)$. Therefore, $(E(s'), N(s')) \models t$ holds. Furthermore, $\mathcal{L}(s') = \mathcal{L}(s)$ because also $\mathsf{pref}(s') = \mathsf{pref}(s)$.
  - Otherwise, $t[p] \neq t[q]$ in which case $N(s')$ is equal to $N(s) \cup \{p, q\}$ and $E(s') = E(s)$. Therefore, $(E(s'), N(s')) \models t$ holds. Consider any $i : l \in \mathcal{L}(s)$ such that $i : l \notin \mathcal{L}(s')$. From $\mathsf{pref}(s') = \mathsf{pref}(s)$ it follows that for $i : (l', P) \in \mathcal{L}_r$ it holds that P is not consistent w.r.t. t by observation that positions $\{p, q\} \subseteq\in P$ are included in N and $t[p] \neq t[q]$. Therefore, by Lemma 12 it holds that $i : l \notin \mathcal{L}_t$.
- $s \in S_M$. It hold that $E(s') = E(s)$ and $N(s') = N(s)$. Therefore, $(E(s'), N(s')) \models t$ remains true. Now, we can use the same argument as before to argue that any pattern removed must not unify with $\mathsf{pref}(s')$. Then the same arguments as given in Lemma 6 can be used to show that $\mathcal{L}_t \subseteq \mathcal{L}(s')$ holds.

Finally, if $s \in S_F$ from the fact that $L(s) = \mathcal{L}(s)$ we know that $\mathcal{L}_t \subseteq L(s)$. It only remains show that $L(s_f) \subseteq \mathcal{L}_t$. There are two cases for this state to become a final state during construction:

- Both $\mathsf{workC} = \emptyset$ and $\mathsf{workF} = \emptyset$. Suppose for a contradiction that there is some $i : l \in L(s_f)$ such that $i : l \notin \mathcal{L}_t$. It follows that $l \not\leq t$, which means that for $i : (l', P) \in \mathcal{L}_r$ that $l' \not\leq t$ or t is not consistent w.r.t. P by Lemma 12. We show that both cases lead to a contradiction:
  - Case $l' \not\leq t$. This follows essentially from the same observations as Lemma 8.
  - Case t is not consistent w.r.t. P. From the fact that $\mathsf{pref}(s_f)$ unifies with t and that it is a ground term due to $\mathsf{workF} = \emptyset$ it follows that for all p, q such that $\{p, q\} \subseteq\in P_i$ they are defined in $\mathsf{pref}(s)$ and therefore it holds that $\{p, q\} \in E(s)$. Therefore, for all $p, q \in C$ for consistency class $C \in P_i$ it holds that $t[p] = t[q]$ and as such t is consistent w.r.t. P_i. As such i is not an element of $L(s_f)$, contradicting our assumption.
- The set $\mathcal{L}(s)$ is empty. In this case $L(s_f)$ is empty and $L(s_f) \subseteq \mathcal{L}_t$ by definition. ◀

L Proof for Lemma 23

Proof. We show that $\textsc{Match}(M, t, s_0) \neq L(s)$ for all final states s for which for $L(s) \neq \emptyset$. Let s_f be an arbitrary final state such that $L(s) \neq \emptyset$ and pick some pattern $i : \ell \in L(s_f)$. By assumption $\ell \not\leq t$ and by Lemma 12 it holds for the pair $i : (\ell', P) \in \mathcal{L}_r$ that $\ell' \not\leq t$ or t is not consistent w.r.t. P.

- If $\ell' \not\leq t$ then by Proposition 2 it follows that there is a position p and a function symbol $f \in \mathbb{F}$ such that $\mathsf{head}(\ell[p]) = f$ and $\mathsf{head}(t[p]) \neq f$. By Lemma 5 it must be that $\mathsf{head}(\mathsf{pref}(s)[p]) = f$, by which there must be a pair $(s_i, f) \in \mathsf{path}(s)$. Since $\textsc{Match}\text{ANPMA}$ is a function we again have that $\textsc{Match}\text{ANPMA}(M, t, s_0) = \textsc{Match}\text{ANPMA}(M, t, s_i) = \textsc{Match}\text{ANPMA}(M, t, s_f)$. However, by its definition we know that $\mathsf{head}(t[p]) = f$, which contradicts the assumption that $i : l \in L(s_f)$.
- If t is not consistent w.r.t. P. By Lemma 22 we know that $(E(s_f), N(s_f)) \models t$ and for all pairs $\{p, q\} \subseteq\in P$ it holds that $\{p, q\} \in E$ for workC to become empty, because all positions of pattern l are defined in the prefix $\mathsf{pref}(s_f)$. As such t must be consistent w.r.t. P, which contradicts the assumption that $i : l \in L(s_f)$. ◀

On Average-Case Hardness of Higher-Order Model Checking

Yoshiki Nakamura
Tokyo Institute of Technology, Japan
nakamura.yoshiki.ny@gmail.com

Kazuyuki Asada
Tohoku University, Sendai, Japan
asada@riec.tohoku.ac.jp

Naoki Kobayashi
The University of Tokyo, Japan
koba@is.s.u-tokyo.ac.jp

Ryoma Sin'ya
Akita University, Japan
ryoma@math.akita-u.ac.jp

Takeshi Tsukada
The University of Tokyo, Japan
tsukada@is.s.u-tokyo.ac.jp

Abstract

We study a mixture between the *average* case and worst case complexities of higher-order model checking, the problem of deciding whether the tree generated by a given λY-term (or equivalently, a higher-order recursion scheme) satisfies the property expressed by a given tree automaton. Higher-order model checking has recently been studied extensively in the context of higher-order program verification. Although the worst-case complexity of the problem is k-EXPTIME complete for order-k terms, various higher-order model checkers have been developed that run efficiently for typical inputs, and program verification tools have been constructed on top of them. One may, therefore, hope that higher-order model checking can be solved efficiently in the *average* case, despite the worst-case complexity. We provide a negative result, by showing that, under certain assumptions, for almost every term, the higher-order model checking problem specialized for the term is k-EXPTIME hard with respect to the size of automata. The proof is based on a novel intersection type system that characterizes terms that do not contain any useless subterms.

2012 ACM Subject Classification Theory of computation → Program verification

Keywords and phrases Higher-order model checking, average-case complexity, intersection type system

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.21

Related Version A full version of the paper is available at `https://www.kb.is.s.u-tokyo.ac.jp/~koba/papers/OnAverageCaseHOMC.pdf` [20].

Funding This work was supported by JSPS Kakenhi 15H05706 and JSPS Kakenhi 20H00577.

Acknowledgements We would like to thank anonymous referees for useful comments.

1 Introduction

Higher-order model checking [12, 21, 24] asks whether the (possibly infinite) tree generated by a given λY-term (or equivalently, a higher-order recursion scheme) is accepted by a given tree automaton. The problem was shown to be decidable by Ong in 2006 [21], and has been applied to higher-order program verification [15, 16, 22, 19]. Although the worst-case complexity of

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 21; pp. 21:1–21:23
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

higher-order model checking is k-EXPTIME complete (where k is the type-theoretic order of the given λY-term), practical higher-order model checkers have been developed that run fast for many typical inputs. They lead to the development of various automated verification tools for higher-order functional programs.

In view of the situation above, we are interested in the following question: why do higher-order model checkers run efficiently, despite the extremely high worst case complexity? There are a couple of known reasons. First, the worst-case time complexity of higher-order model checking is actually polynomial in the size of a given term, provided that the other parameters (the largest order and arity of functions, and the size of an automaton) are fixed [17]. Second, linear functions do not blow up the complexity [5]. These reasons alone, however, do not fully explain why higher-order model checking works in practice. For example, for the first point above, the constant factor determined by the other parameters is huge.

In the present paper, we consider another possibility: higher-order model checking may actually be easy in the *average* case; in other words, it may be the case that hard instances that cost k-EXPTIME are sparse and many of the instances of higher-order model checking can be solved more efficiently. We give a somewhat negative result on that possibility. For each term t of the λY-calculus, we consider the following higher-order model checking problem specialized to t:

HOMC$(t,\cdot)$: Given a tree automaton $\mathcal{A}$, decide whether the tree generated by t is accepted by $\mathcal{A}$.

Our main result is that for *almost every* term t of order-k that is sufficiently large, HOMC$(t,\cdot)$ is k-EXPTIME hard. A little more precisely, we prove that, for the set $\mathtt{Terms}_{n,k}$ of terms of size n and order k (modulo certain additional conditions that we explain later), the ratio of "hard" terms:

$$\frac{\#\{t \in \mathtt{Terms}_{n,k} \mid \mathrm{HOMC}(t,\cdot) \text{ is } k\text{-EXPTIME hard}\}}{\#\mathtt{Terms}_{n,k}}$$

tends to 1 if $n \to \infty$ (where $\#S$ denotes the cardinality of a set S). In other words, if we pick up a term randomly according to the uniform distribution over $\mathtt{Terms}_{n,k}$, it is likely that there exists a bad automaton $\mathcal{A}$ such that HOMC$(t,\mathcal{A})$ is very hard. Note that this is a mixture between the average case and worst-case analysis: the result above says that in the *average case* on the choice of a term t, the complexity of HOMC$(t,\cdot)$ is k-EXPTIME hard in the *worst-case* on the choice of an automaton.

In order to make the above analysis meaningful, we have to carefully define the set $\mathtt{Terms}_{n,k}$ of terms. To see why, consider a term of the form $(\lambda x.\mathtt{c})t$, where $\mathtt{c}$ is a nullary tree constructor. The term generates the singleton tree $\mathtt{c}$; so, no matter how large t is, the problem HOMC$((\lambda x.\mathtt{c})t,\cdot)$ is easy. Thus, if we include such terms in $\mathtt{Terms}_{n,k}$, the ratio of hard instances above would not be 1 for the trivial reason. In the context of applications of higher-order model checking to program verification, however, such instances are unlikely to appear: a λY-term corresponds to a program, and it is unlikely that one writes a program that contains such a huge useless term t. (It might be the case for machine-generated programs, but even in that case, one can apply simple preprocessing to remove such useless terms before invoking a costly higher-order model checking algorithm.) We, therefore, exclude out, from $\mathtt{Terms}_{n,k}$, terms that contain any useless subterms. Here, a subterm t_1 of t is useless if replacing t_1 with another term never changes the tree generated by t. (We will impose further conditions such as the number of variables, which will be explained in Section 2.)

Once the set $\mathtt{Terms}_{n,k}$ is properly chosen as explained above, our main result can be proved as follows. First, according to Kobayashi and Ong's work on the complexity of higher-order model checking [18], there exists an order-k "hard" term $t_{\textsc{Hard},k}$ such that

$\mathrm{HOMC}(t_{\mathrm{HARD},k}, \cdot)$ is k-EXPTIME complete. Second, according to Asada et al.'s work on quantitative analysis on λ-terms [1], any sufficiently large term t can be decomposed into the form $E[C_1, \dots, C_m]$ for sufficiently many contexts $C_1, \dots, C_m$, where each C_i is large enough to be replaced by a context, say C_i', that contains the hard term $t_{\mathrm{HARD},k}$, without changing the term size. Thus, by using their argument (which originates from the so called "infinite monkey theorem" stating that almost every word contains any given word), we can deduce that almost every sufficiently large term contains the hard term $t_{\mathrm{HARD},k}$, *if we ignore the condition that useless terms should be excluded.* Finally (and most importantly), we can choose a context C_i' that contains the hard term, so that if $E[C_1, \dots, C_i, \dots, C_m]$ belongs to $\mathtt{Terms}_{n,k}$ (and therefore does not contain any useless subterms), then so does $E[C_1, \dots, C_i', \dots, C_m]$.

To obtain the last part of the result, we develop a novel intersection type system that completely characterizes the set of terms that do not contain useless terms, in the sense that a closed term t is typable if and only if t does not contain any useless term. This type system is one of the main contributions of the present paper, and may be of independent interest. Type systems for useless code elimination have been studied before [6, 7, 13] (in particular, Damiani [7] used intersection types), but the complete characterization was not known, to our knowledge.

The rest of this paper is structured as follows. Section 2 provides formal definitions of λY-terms and the higher-oder model checking. Section 3 states our main result and gives a proof outline. Sections 4–6 prove the theorem. Section 7 discusses related work, and Section 8 concludes this article.

2 Preliminaries

For a map f, we write $\mathrm{dom}(f)$ for the domain of f and $\mathrm{rng}(f)$ for the range of f. We denote by $\mathbb{N}$ the set of non-negative integers and by $\mathbb{N}_+$ the set of positive integers. For $m, n \in \mathbb{N}$, we write $[m, n]$ for the set $\{i \in \mathbb{N} \mid m \leq i \leq n\}$, and $[n]$ for $[1, n]$; note that $[0] = \emptyset$. The cardinality of a set A is denoted by $\#(A)$. We use $A \uplus B$ instead of $A \cup B$ if sets A and B are disjoint. For a set A, we write A^* for the set of finite sequences consisting of elements of A. An L*-labeled tree* is a partial map T from $\mathbb{N}_+^*$ to L such that, for every $\langle \alpha, i\rangle \in \mathbb{N}_+^* \times \mathbb{N}_+$, if $\alpha \cdot i \in \mathrm{dom}(T)$, then $\{\alpha, \alpha \cdot 1, \dots, \alpha \cdot (i-1)\} \subseteq \mathrm{dom}(T)$. An L-labeled tree T is called *finite* if $\mathrm{dom}(T)$ is finite. We write $\mathtt{r}_T(\alpha)$ for the number of children of a node α in T, i.e., $\mathtt{r}_T(\alpha) = \#\{i \in \mathbb{N}_+ \mid \alpha \cdot i \in \mathrm{dom}(T)\}$. A *ranked alphabet* Σ is a map from a finite set of symbols to $\mathbb{N}$. We call $\Sigma(a)$ the *rank* of a. A $\mathrm{dom}(\Sigma)$-labeled tree T is called a Σ*-ranked tree* (Σ*-tree*, for short) if, for every $\alpha \in \mathrm{dom}(T)$, $\mathtt{r}_T(\alpha) = \Sigma(T(\alpha))$.

2.1 λY-Terms as Tree Generators

In this subsection, we introduce (simply-typed) λY-terms [27] as generators of (possibly infinite) Σ-trees. In the context of higher-order model checking, higher-order recursion schemes have originally been used as generators of trees [12, 21], but the λY-terms (with constants of order up to 1 as tree constructors), which are equi-expressive with higher-order recursion schemes, (see, e.g., [25]), have also been used in later studies on higher-order model checking [24]. For the purpose of the present paper, we find it more convenient to use λY-terms.

Let Σ be a ranked alphabet. Each $a \in \mathrm{dom}(\Sigma)$ is called a *tree constructor*. We use meta-variables a, b, c for tree constructors (and $\mathtt{a}, \mathtt{b}, \mathtt{c}, \dots$ for concrete symbols). The set of *simple types* is defined by: $\kappa ::= \mathtt{o} \mid \kappa_1 \to \kappa_2$. The *ground type* $\mathtt{o}$ is the type of trees. The *order* and *arity* of a simple type κ, written $\mathtt{ord}(\kappa)$ and $\mathtt{ar}(\kappa)$ respectively, are defined by: $\mathtt{ord}(\kappa_1 \to \cdots \to \kappa_n \to \mathtt{o}) \triangleq \max(\{0\} \cup \{\mathtt{ord}(\kappa_i) + 1 \mid 1 \leq i \leq n\})$ and $\mathtt{ar}(\kappa_1 \to \cdots \to \kappa_n \to \mathtt{o}) \triangleq n$, where $n \geq 0$. Let $\mathcal{V}$ be a countably infinite set, which is ranged over by x, y, z.

▶ **Definition 1** (λY-terms). *The set of* (λY-)terms *(over* Σ*) is defined by:*

$$t ::= x^\kappa \mid \lambda x^\kappa.t \mid \lambda\_^\kappa.t \mid t_1\, t_2 \mid \mathbf{Y}^\kappa t \mid a(t_1, \dots, t_{\Sigma(a)}) \mid \bot^\kappa.$$

We call elements of $\mathcal{V} \uplus \{\_\}$ *variables* and use meta-variables $\bar{x}, \bar{y}, \bar{z}$ for them. As in the standard λY-calculus, the constructor $\mathbf{Y}^\kappa$ may be considered a fixpoint operator of type $(\kappa \to \kappa) \to \kappa$. The special variable '_' denotes an unused variable (hence can occur only in a binder, not in the body of a function). For each type κ, we have a special term $\bot^\kappa$, which intuitively represents an unused term and will play an important role in the definition of minimal terms. We often omit type annotations (for example, $\lambda x^\kappa.x^\kappa$ is just written $\lambda x.x$). For a term t, we write $\mathbf{FV}(t)$ for the set of all the free variables of t.

A *simple type environment* Γ is a finite partial map from $\mathcal{V}$ (recall that the special variable _ does not belong to $\mathcal{V}$) to the set of simple types. We simply write $\Gamma, x : \kappa$ for $\Gamma \uplus \{x \mapsto \kappa\}$. The type judgment relation $\Gamma \vdash_{\mathrm{ST}} t : \kappa$ is inductively defined by the following rules:

$$\frac{}{x : \kappa \vdash_{\mathrm{ST}} x^\kappa : \kappa}\,(\mathrm{Var}) \quad \frac{\Gamma, x : \kappa \vdash_{\mathrm{ST}} t : \kappa'}{\Gamma \vdash_{\mathrm{ST}} \lambda x^\kappa.t : \kappa \to \kappa'}\,(\mathrm{Abs1}) \quad \frac{\Gamma \vdash_{\mathrm{ST}} t : \kappa'}{\Gamma \vdash_{\mathrm{ST}} \lambda \bar{x}^\kappa.t : \kappa \to \kappa'}\,(\mathrm{Abs2}) \quad \frac{}{\emptyset \vdash_{\mathrm{ST}} \bot^\kappa : \kappa}\,(\bot)$$

$$\frac{\Gamma_1 \vdash_{\mathrm{ST}} t : \kappa \to \kappa' \quad \Gamma_2 \vdash_{\mathrm{ST}} s : \kappa}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{ST}} t\, s : \kappa'}\,(\mathrm{App}) \quad \frac{\Gamma_1 \vdash_{\mathrm{ST}} t_1 : \mathtt{o} \ \dots \ \Gamma_n \vdash_{\mathrm{ST}} t_n : \mathtt{o}}{\bigcup_{i \in [n]} \Gamma_i \vdash_{\mathrm{ST}} a(t_1, \dots, t_n) : \mathtt{o}}\,(a) \quad \frac{\Gamma \vdash_{\mathrm{ST}} t : \kappa \to \kappa}{\Gamma \vdash_{\mathrm{ST}} \mathbf{Y}^\kappa\, t : \kappa}\,(\mathbf{Y})$$

Henceforth, we only consider *well-typed* terms (i.e., terms t such that $\Gamma \vdash_{\mathrm{ST}} t : \kappa$ for some $\langle \Gamma, \kappa \rangle$). Note that for every well-typed term t, there is a unique pair $\langle \Gamma, \kappa \rangle$ such that $\Gamma \vdash_{\mathrm{ST}} t : \kappa$; and moreover, its derivation tree is also uniquely determined. We sometimes annotate a term with its type, like t^κ, when t has type κ (under a certain type environment). We say that t is *closed* if $\Gamma = \emptyset$; and that t is *ground-typed* if $\kappa = \mathtt{o}$.

▶ **Definition 2.** *The* (call-by-name) reduction relation $\longrightarrow$ *is defined as the least binary relation on well-typed terms (up to α-equivalence) closed under the following rules, where we write $t\{s/x\}$ for the term obtained from t by substituting s for all the free occurrences of x in a capture-avoiding manner:*

$$(\beta) \quad (\lambda \bar{x}.t)\, s \longrightarrow t\{s/\bar{x}\}; \quad (\mathbf{Y}) \quad \mathbf{Y}t \longrightarrow t\,(\mathbf{Y}t); \quad (\bot) \quad \bot^{\kappa_1 \to \kappa_2} t \longrightarrow \bot^{\kappa_2};$$
$$(\mathrm{App}) \quad tu \longrightarrow t'u \ \textit{if}\ t \longrightarrow t'; \ (a) \quad a(t_1, \dots, t_n) \longrightarrow a(t_1, \dots, t_{i-1}, t'_i, t_{i+1}, \dots, t_n) \ \textit{if}\ t_i \longrightarrow t'_i.$$

We write $\longrightarrow^*$ for the reflexive transitive closure of $\longrightarrow$.

The *tree generated by a closed and ground λY-term* t is the one obtained from t by (possibly) infinite rewriting with respect to the above reduction relation. The precise definition is given below.

We write $\Sigma^\bot$ for the ranked alphabet $\Sigma \uplus \{\bot \mapsto 0\}$. We define *the binary relation $\sqsubseteq$ on $\Sigma^\bot$-trees* by: $T_1 \sqsubseteq T_2$ if and only if (i) $\mathrm{dom}(T_1) \subseteq \mathrm{dom}(T_2)$ and (ii) for every $\alpha \in \mathrm{dom}(T_1)$, $T_1(\alpha) = \bot$ or $T_1(\alpha) = T_2(\alpha)$. We write $T_1 \sqsubset T_2$ if $T_1 \sqsubseteq T_2$ and $T_1 \neq T_2$. We denote the join of $\{T_i\}_{i \in I}$ with respect to $\sqsubseteq$ by $\bigsqcup_{i \in I} T_i$ if defined.

A term consisting of only tree constructors and $\bot^\mathtt{o}$ can naturally be regarded as a $\Sigma^\bot$-tree. For example, $\mathtt{b}(\mathtt{c}, \mathtt{a}(\bot^\mathtt{o}))$ can be regarded as the $\Sigma^\bot$-tree: $\{\epsilon \mapsto \mathtt{b}, 1 \mapsto \mathtt{c}, 2 \mapsto \mathtt{a}, 2 \cdot 1 \mapsto \bot\}$; hence we identify finite trees and terms consisting of tree constructors and $\bot^\mathtt{o}$ below. For each closed and ground-typed term t, the $\Sigma^\perp$-tree $t^\perp$ is defined by: $t^\perp \triangleq a(t_1^\perp, \dots, t_{\Sigma(a)}^\perp)$ if $t = a(t_1, \dots, t_{\Sigma(a)})$; and $t^\perp \triangleq \perp$ otherwise. The *value tree* of a closed and ground-typed term t, written $T(t)$, is defined by: $T(t) \triangleq \bigsqcup\{s^\perp \mid t \longrightarrow^* s\}$. For example, consider the value tree of $(\mathbf{Y}t_1)\mathtt{c}$ where $t_1 = \lambda f^{\mathtt{o}\to\mathtt{o}}.\lambda x^{\mathtt{o}}.\mathtt{b}(x, f(\mathtt{a}(x)))$. By applying the reduction rules $(\mathbf{Y})$ and (β), we can obtain the following reduction sequence

$$(\mathbf{Y}t_1)\mathtt{c} \longrightarrow t_1(\mathbf{Y}t_1)\mathtt{c} \longrightarrow^* \mathtt{b}(\mathtt{c}, (\mathbf{Y}t_1)(\mathtt{a}(\mathtt{c}))) \longrightarrow^* \mathtt{b}(\mathtt{c}, \mathtt{b}(\mathtt{a}(\mathtt{c}), (\mathbf{Y}t_1)(\mathtt{a}(\mathtt{a}(\mathtt{c})))))$$

and observe that $T(t)$ is the infinite tree of the form $\mathtt{b}(\mathtt{c}, \mathtt{b}(\mathtt{a}(\mathtt{c}), \mathtt{b}(\mathtt{a}(\mathtt{a}(\mathtt{c})), \mathtt{b}(\cdots))))$.

We also define the size and order of a term, which will be used in the complexity analysis.

▶ **Definition 3** (size, order). *The* size *of a term t is defined by: $|x| = |\perp| \triangleq 1$, $|\lambda\bar{x}.t| = |\mathbf{Y}t| \triangleq 1 + |t|$, $|t_1\, t_2| \triangleq 1 + |t_1| + |t_2|$, and $|a(t_1, \dots, t_{\Sigma(a)})| \triangleq 1 + \sum_{i\in[\Sigma(a)]} |t_i|$. The* order *of a term t, written $\mathtt{ord}\,(t)$, is defined by:*

$$\mathtt{ord}\,(t) \triangleq \max(\{0\} \cup \{\mathtt{ord}\,(\kappa) \mid \lambda x^\kappa.s \text{ or } \mathbf{Y}^\kappa s \text{ is a subterm of } t\}).$$

Note that the size of a variable is a constant; this is appropriate in our context, as we fix the number of variables in the main theorem (Theorem 7).

▶ Remark 4. Our definition of the order of a λY-term given above deviates from the standard definition of the order of a λY-term (where the order of a term is defined as the largest order of the types of subterms) [25]. For example, the order of $Y^{\mathtt{o}}\lambda x^{\mathtt{o}}.\mathtt{a}(x)$ (which generates a unary infinite tree consisting of only $\mathtt{a}$) is 0 in our definition, but it is 1 in the standard definition, because $\lambda x^{\mathtt{o}}.\mathtt{a}(x)$ has type $\mathtt{o} \to \mathtt{o}$, which has order 1. Our definition is motivated to make the order of λY-term equivalent to that of the corresponding higher-order recursion scheme (where the order is defined as the largest order of the types of recursive functions); for example, the above term corresponds to the higher-order recursion scheme consisting of a single rule $S \longrightarrow \mathtt{a}(S)$, whose order is 0. The translation from higher-order recursion schemes to λY-terms given in [25] is order-preserving in our definition, but increases the order by 1 in the definition of [25]. There is also a translation from λY-terms to higher-order recursion schemes that preserves the order in our definition (given an order-k λY-term, reduce all the β-redexes of the form $(\lambda x^\kappa.s)t$ with $\mathtt{ord}\,(\kappa) = k$ first, and then apply the translation suggested in [25]; the first phase of β-reductions may incur an exponential blow-up, which can be avoided by appropriately introducing non-terminals to avoid duplications of terms).

2.2 Higher-Order Model Checking

We assume the notion of *alternating parity tree automaton* (*APT* for short): see, e.g., [10]. The precise definition of APT is unnecessary for understanding our technical development in later sections, once you admit the results in this subsection. We recall the definition of higher-order model checking below.

▶ **Definition 5** (higher-order model checking problem). *The* higher-order model checking problem, *written* $\mathrm{HOMC}\,(\cdot,\cdot)$*, is the problem of, given a closed and ground-typed λY-term t over Σ and an APT $\mathcal{A}$ over Σ as input, deciding whether $\mathcal{A}$ accepts $T(t)$. We write* $\mathrm{HOMC}_k(\cdot,\cdot)$ *when the first input is restricted to a term of order-k. We denote by* $\mathrm{HOMC}\,(t,\cdot)$ *the problem obtained by fixing the first input to t, i.e., the problem of, given an APT $\mathcal{A}$ as input, deciding whether $\mathcal{A}$ accepts $T(t)$.*

Ong [21] has shown that the $\mathrm{HOMC}_k(\cdot,\cdot)$ is k-EXPTIME complete (combined complexity) for each $k \geq 0$. The following theorem states the complexity of $\mathrm{HOMC}\,(t,\cdot)$, which serves as a basis of the present work.

▶ **Theorem 6** ([21] for (1) and [18, Theorem 3.8] for (2))**.** *For each* $k \geq 1$*,*

(1) *for every order-*k λY*-term* t*,* $\mathrm{HOMC}\,(t,\cdot)$ *is decidable in* k-EXPTIME*; and*

(2) *there exists an order-*k λY*-term* $t_{\mathrm{HARD},k}$ *such that* $\mathrm{HOMC}\,(t_{\mathrm{HARD},k},\cdot)$ *is* k-EXPTIME *hard.*

3 Main Theorem

This section formally states the main result of the paper: for almost every order-k λY-term, the higher-order model checking problem $\mathrm{HOMC}(t,\cdot)$ is k-EXPTIME hard, under a certain assumption, and sketches an overall structure of the proof. We first prepare some auxiliary notations. We denote by $[t]_\alpha$ the α-equivalence class of t. In our quantitative analysis, we count α-equivalent terms at most once (e.g., we do not distinguish $(\lambda x.\lambda y.x)z$ and $(\lambda z.\lambda \_.z)z$). We define $\#\texttt{vars}\,(t) \triangleq \min\{\#(\mathbf{V}(t')) \mid t' \in [t]_\alpha\}$, where $\mathbf{V}(t)$ denotes the set of all the variables (except _) occurring in t. Namely, $\#\texttt{vars}\,(t)$ is the *minimum number of variables* occurring in the term t, up to α-equivalence. For example, $\#\texttt{vars}\,((\lambda x.\lambda y.x)z) = 1$ since the term is α-equivalent to $(\lambda z.\lambda \_.z)z$. The *internal arity* of a term t, written $\texttt{iar}\,(t)$, is defined by: $\texttt{iar}\,(t) \triangleq \max(\{\texttt{ar}\,(\kappa) \mid s^\kappa \text{ is a subterm of } t\})$.

Let $\hat{\Lambda}_n(k,\iota,\xi)$ be the set of all (α-equivalence classes of) closed and ground-typed λY-terms such that[1]

(i) the size is n (i.e., $|t| = n$);
(ii) the order is up to k (i.e., $\texttt{ord}\,(t) \leq k$);
(iii) the internal arity is up to ι (i.e., $\texttt{iar}\,(t) \leq \iota$);
(iv) the number of variable names is up to ξ (i.e., $\#\texttt{vars}\,(t) \leq \xi$); and
(v) the terms are *minimal* (see Section 3.1 below for the definition).

The main theorem is stated as follows.

▶ **Theorem 7** (main theorem)**.** *For each* $k \geq 1$*, let* ι *and* ξ *be sufficiently large natural numbers. Then,*

$$\lim_{n\to\infty} \frac{\#\Big(\{t \in \hat{\Lambda}_n(k,\iota,\xi) \mid \mathrm{HOMC}(t,\cdot) \textit{ is } k\text{-EXPTIME } \textit{hard}\}\Big)}{\#\Big(\hat{\Lambda}_n(k,\iota,\xi)\Big)} = 1.$$

Below we first define the minimality in Section 3.1 and give a proof outline in Section 3.2.

3.1 Minimal Terms

Intuitively, a term is *minimal* if it has no useless subterm. For the formal definition, we first define the relation $\sqsubseteq$ on *terms*, which is analogous to the corresponding relation ($\sqsubseteq$) on trees.

▶ **Definition 8.** *The* approximate relation $\sqsubseteq$ *is the least precongruence (i.e., the relation closed under all the term constructors) such that* $\bot^\kappa \sqsubseteq t^\kappa$*.*

[1] The set $\hat{\Lambda}_n(k,\iota,\xi)$ implicitly depends on the choice of ranked alphabet Σ. The main theorem holds independently of the choice of Σ unless Σ is unreasonably small.

In other words, $s \sqsubseteq t$ means that s is obtained from t by replacing subterms $t_1^{\kappa_1}, \dots, t_n^{\kappa_n}$ with $\perp^{\kappa_1}, \dots, \perp^{\kappa_n}$. We write $s \sqsubset t$ if $s \sqsubseteq t$ and $s \neq t$. We denote the join of $\{t_i\}_{i \in I}$ with respect to $\sqsubseteq$ (i.e., the least upper bound of $\{t_i\}_{i \in I}$ with respect to $\sqsubseteq$) by $\bigsqcup_{i \in I} t_i$ if defined, and we sometimes write $t_1 \sqcup \dots \sqcup t_n$ for $\bigsqcup_{i \in [n]} t_i$. For example, $(\lambda x.\mathtt{b}(x, \perp)) \sqcup (\lambda x.\mathtt{b}(\perp, x)) = \lambda x.\mathtt{b}(x, x)$. Note that, with respect to $\Sigma^{\perp}$-tree terms, the relation $\sqsubseteq$ on terms is equivalent to the relation $\sqsubseteq$ on $\Sigma^{\perp}$-trees.

▶ **Definition 9.** *A closed and ground-typed term t is* minimal *if for every $s \sqsubset t$, $T(s) \neq T(t)$.*

In other words, a term t is *not minimal* if there exists s obtained by replacing a non-$\perp$ subterm u of t with $\perp$ such that $T(s) = T(t)$.

▶ **Example 10.** Let $t = (\lambda x.\lambda y.x)\ \mathtt{a}\ u$, with $u \neq \perp$. Then the value tree $T(t) = \mathtt{a}$ (since $(\lambda x.\lambda y.x)\ \mathtt{a}\ u \longrightarrow (\lambda y.\mathtt{a})\ u \longrightarrow \mathtt{a}$). Note that the subterm u is "useless"; indeed the term $s = (\lambda x.\lambda y.x)\ \mathtt{a}\ \perp$, obtained from t by replacing u with $\perp$, also generates $\mathtt{a}$. Thus, t is not minimal. In contrast, s is minimal. In fact, any term obtained by replacing a non-$\perp$ subterm of s with $\perp$ (such as $(\lambda x.\lambda y.\perp)\ \mathtt{a}\ \perp$) fails to generate $\mathtt{a}$.

The following proposition gives an important property of minimal terms. We write $t' \preceq t$ when t' is a subterm of a term t.

▶ **Proposition 11.** *Let t be a closed and ground-typed term. If t is minimal, then for every non-$\perp$, closed and ground-typed subterm $s \preceq t$, its value tree $T(s)$ is a subtree of $T(t)$.*

This property is intuitively obvious. Since t is minimal, the subterm s assumed to be non-$\perp$ must be used in the computation of the value tree $T(t)$. As s is closed and ground-typed, the only way to use s is to place its value tree $T(s)$ somewhere in $T(t)$; hence the proposition. For a formal proof, see the full version [20].

3.2 Proof Outline

For each k, let $t_{\mathrm{HARD},k}$ be an order-k closed, ground-typed term such that the problem $\mathrm{HOMC}(t, \cdot)$ is k-EXPTIME hard. The existence of $t_{\mathrm{HARD},k}$ is guaranteed by Theorem 6 (2). We can assume, without loss of generality, that $t_{\mathrm{HARD},k}$ is minimal; otherwise take a minimal element $t'_{\mathrm{HARD},k}$ of $\{s \mid T(s) = T(t_{\mathrm{HARD},k})\}$. Theorem 7 follows immediately from Lemmas 12 and 13 below, which respectively state: (a) for each order k, every order-k minimal term containing the "hard" term $t_{\mathrm{HARD},k}$ as a subterm yields k-EXPTIME-hardness for the higher-order model checking problem; and (b) almost every minimal term of order k contains the "hard" term $t_{\mathrm{HARD},k}$ as a subterm.

▶ **Lemma 12.** *Let $k \geq 1$. For every* minimal *λY-term $t \succeq t_{\mathrm{HARD},k}$,* $\mathrm{HOMC}\,(t, \cdot)$ *is k-EXPTIME hard.*

Proof. Assume that $t \succeq t_{\mathrm{HARD},k}$. Then $T(t) \succeq T(t_{\mathrm{HARD},k})$ by Proposition 11, i.e. $T(t_{\mathrm{HARD},k}) = (T(t){\restriction_\alpha})$ for some $\alpha \in \mathrm{dom}(T(t))$ where $(T{\restriction_\alpha})$ denotes the subtree of T induced by the node α. Let c be the length of α. For any APT $\mathcal{A}$, we can construct an automaton $\mathcal{A}{\restriction_\alpha}$ by adding c states to $\mathcal{A}$ and replacing the initial state so that $\mathcal{A}{\restriction_\alpha}$ accepts T if and only if $\mathcal{A}$ accepts $T{\restriction_\alpha}$ (intuitively, $\mathcal{A}{\restriction_\alpha}$ first moves to the node α then behaves like $\mathcal{A}$). Then the polynomial-time function $\mathcal{A} \mapsto (\mathcal{A}{\restriction_\alpha})$ gives a polynomial-time reduction from $\mathrm{HOMC}\,(t_{\mathrm{HARD},k}, \cdot)$ to $\mathrm{HOMC}\,(t, \cdot)$ The lemma follows from k-EXPTIME-hardness of $\mathrm{HOMC}\,(t_{\mathrm{HARD},k}, \cdot)$. ◀

▶ **Lemma 13.** *For each $k \geq 1$, let ι and ξ be sufficiently large natural numbers. Then,*

$$\lim_{n\to\infty} \frac{\#\Big(\{t \in \hat{\Lambda}_n(k,\iota,\xi) \mid t \succeq t_{\textsc{Hard},k}\}\Big)}{\#\Big(\hat{\Lambda}_n(k,\iota,\xi)\Big)} = 1.$$

It remains to show Lemma 13. To this end, we introduce the following lemma (where the precise definition of *second-order contexts* will be given in Section 4).

▶ **Lemma 14.** *For each $k \geq 1$, let ι and ξ be sufficiently large natural numbers. There exists m such that the following holds: Let $n \geq m$, E be any second-order linear context, and C be any affine context such that $|C| \geq m$ and $E[C] \in \hat{\Lambda}_n(k,\iota,\xi)$. Then there exists an affine context $D \succeq t_{\textsc{Hard},k}$ such that $E[D] \in \hat{\Lambda}_n(k,\iota,\xi)$.*

We show how Lemma 13 follows from Lemma 14 in Section 4. We then introduce a new intersection type system that characterizes the minimality in Section 5, and use it to prove Lemma 14 in Section 6.

4 Infinite Monkey Theorem for Minimal Terms

We sketch a proof of Lemma 13 (modulo Lemma 14) in this section; see [20] for the full proof. The proof is analogous to that of the following, so-called *infinite monkey theorem* (a.k.a. "Borges's theorem" [9, p.61, Note I.35]) for words:

▶ **Theorem 15.** *Let Σ be a finite alphabet. For any word $x \in \Sigma^*$, almost all words contain x as a subword, i.e.*

$$\lim_{n\to\infty} \frac{\#(\{w \in \Sigma^n \mid w = uxv \textit{ for some } u,v \in \Sigma^*\})}{\#(\Sigma^n)} = 1.$$

The theorem above follows from the following reasoning: Any word w can be decomposed into the form $w_1 w_2 \cdots w_p w'$ where $|w_i| = |x|$ and $|w'| < |x|$. If we pick w randomly, the probability that w_i coincides with x is $(\frac{1}{|\Sigma|})^{|x|}$; hence the probability that w contains x is at least $1 - (1 - (\frac{1}{|\Sigma|})^{|x|})^p$, which tends to 1 when n tends to infinity. For the purpose of proving Lemma 13, we analogously decompose each term t to the form $E[C_1,\ldots,C_p]$ (where E and C_i respectively correspond to w' and w_i above), by using the tree decomposition in [1]. We can then use Lemma 14 to prove Lemma 13. The hardest part is actually to prove Lemma 14, which is deferred to Section 6.

We first need to prepare some definitions. In order to make use of the tree decomposition function Φ_m in [1], below we regard a λY-term over Σ as a $\Sigma_{\hat{\Lambda}(k,\iota,\xi)}$-tree where $\Sigma_{\hat{\Lambda}(k,\iota,\xi)}$ is an extension of Σ defined by:

$$\begin{aligned}\Sigma_{\hat{\Lambda}(k,\iota,\xi)} \triangleq \Sigma &\uplus \{x \mapsto 0 \mid x \in \mathcal{V}_\xi\}\\ &\uplus \{\lambda \bar{x}^{\kappa} \mapsto 1 \mid \bar{x} \in \mathcal{V}_\xi \uplus \{\_\}, \mathtt{ord}\,(\kappa) \leq k, \mathtt{iar}\,(\kappa) \leq \iota\}\\ &\uplus \{@ \mapsto 2\} \uplus \{\mathbf{Y}^{\kappa} \mapsto 1, \bot^{\kappa} \mapsto 0 \mid \mathtt{ord}\,(\kappa) \leq k, \mathtt{iar}\,(\kappa) \leq \iota\}\end{aligned}$$

Here, $\mathcal{V}_\xi = \{x_1, \cdots, x_\xi\}$ is a finite subset of $\mathcal{V}$ and the symbol @ represents the application operation.

Next, we recall the notion of contexts and second-order contexts used in the decomposition. A *context* is a tree with special leaves [ ] called *holes*. Formally, the set of contexts over Σ is given by

$$C ::= [\,] \mid a(C_1,\ldots,C_{\Sigma(a)}),$$

where a ranges over $\mathrm{dom}(\Sigma)$. We call a context with k holes a *k-context*, and call a context *affine* if it is a 0- or 1-context. The *size* of a context C, denoted by $|C|$, is inductively defined by: $|[\,]| \triangleq 0$ and $|a(C_1, \ldots, C_{\Sigma(a)})| \triangleq 1 + |C_1| + \cdots + |C_{\Sigma(a)}|$. For a k-context C and contexts $\overrightarrow{C} = C_1 \cdots C_k$, we write $C[\overrightarrow{C}]$ or $C[C_1, \ldots, C_k]$ for the context obtained by replacing each occurrence of $[\,]$ in C with C_i in the left-to-right order.

A *second-order context* is an expression having holes of the form $\llbracket\rrbracket^n_k$ (called *second-order holes*), which should be filled with a k-context of size n. Formally, the set of second-order contexts over Σ, ranged over by E, is defined by:

$$E ::= \llbracket\rrbracket^n_k[E_1, \ldots, E_k] \ \mid\ a(E_1, \ldots, E_{\Sigma(a)}) \quad (a \in \mathrm{dom}(\Sigma)).$$

We write $\mathtt{shn}\,(E)$ for the number of the second-order holes in E, and $E.i$ for the i-th leftmost second-order hole in E.

▶ **Definition 16** (substitution for second-order contexts)**.** *For a context C and a second-order hole $\llbracket\rrbracket^n_k$, we write $C : \llbracket\rrbracket^n_k$ if C is a k-context of size n. For a second-order context E and a sequence of contexts $\overrightarrow{C} = C_1 \cdots C_{\mathtt{shn}(E)}$ such that $C_i : E.i$ for each $i \in [\mathtt{shn}\,(E)]$, we write $E[\overrightarrow{C}]$ or $E[C_1, \ldots, C_{\mathtt{shn}(E)}]$ for the tree which can be obtained by replacing each occurrence of $\llbracket\rrbracket$ in E with C_i in the left-to-right manner (and by interpreting the syntactical bracket $[-]$ as the substitution operation for usual contexts), where $\#\left(\overrightarrow{C_i}\right) = \mathtt{shn}\,(E_i)$ for each i:*

$$(\llbracket\rrbracket^n_k[E_1, \ldots, E_k])\,[C \cdot \overrightarrow{C_1} \cdots \overrightarrow{C_k}] \triangleq C[E_1[\overrightarrow{C_1}], \ldots, E_k[\overrightarrow{C_k}]]$$
$$(a(E_1, \ldots, E_{\Sigma(a)}))[\overrightarrow{C_1} \cdots \overrightarrow{C}_{\Sigma(a)}] \triangleq a(E_1[\overrightarrow{C_1}], \ldots, E_{\Sigma(a)}[\overrightarrow{C}_{\Sigma(a)}]).$$

We can use the decomposition function Φ_m (where $m > 0$ is an integer parameter) introduced in [1] to uniquely decompose (the tree representation of) a λY-term t to $(E, C_1, \ldots, C_k)$ such that (i) E is a second-order context, (ii) C_i's are affine contexts such that $m \leq |C_i| \leq rm$ (where r is the largest arity of the symbols in $\Sigma_{\hat{\Lambda}(k,\iota,\xi)}$), (iii) $t = E[C_1, \ldots, C_k]$, (iv) $k \geq \frac{|t|}{2rm}$, (v) $\Phi_m(E[C_1, \ldots, C_{i-1}, D_i, C_{i+1}, \ldots, C_k]) = (E, C_1, \ldots, C_{i-1}, D_i, C_{i+1}, \ldots, C_k)$ for any "good" context D_i (see [1, 20] for the definition of "goodness").

▶ **Example 17.** The term $\mathtt{a}((\lambda\_.\mathtt{a}((\lambda x.\mathtt{a}(x))(\mathbf{Y}\,\lambda y.\mathtt{a}(y))))\bot)$ on the left hand side of Figure 1 can be decomposed into the second-order context $\mathtt{a}(\llbracket\rrbracket^4_1[(\llbracket\rrbracket^3_0[])(\mathbf{Y}\,(\llbracket\rrbracket^3_0[]))])$ and affine contexts, as shown on the right hand side.

We are now ready to provide a proof sketch of Lemma 13. Let us define $S^{n,E}_i$ and $\mathcal{E}^n_m$ by:

$$S^{n,E}_i \triangleq \{t \in \hat{\Lambda}_n(k, \iota, \xi) \mid \Phi_m(t) = (E, C_1, \ldots, C_\ell),\ \text{and}\ t_{\text{HARD},k} \not\preceq C_j \text{ for each } j \in [i]\}$$
$$\mathcal{E}^n_m \triangleq \left\{E \mid \Phi_m(t) = (E, \cdots) \text{ for some } t \in \hat{\Lambda}_n(k, \iota, \xi)\right\}.$$

Below we write $E \star (C_1, \ldots, C_\ell) \in S$ to mean $E[C_1, \ldots, C_\ell] \in S$ and $\Phi_m(E[C_1, \ldots, C_\ell]) = (E, C_1, \ldots, C_\ell)$. If n and m (with $n > m$) are sufficiently large, we can estimate the ratio $\frac{\#(S^{n,E}_i)}{\#(S^{n,E}_{i-1})}$ for $i \in [\mathtt{shn}\,(E)]$ by:

$$\frac{\#\left(S^{n,E}_i\right)}{\#\left(S^{n,E}_{i-1}\right)} = \frac{\sum_{C_1,\ldots,C_{i-1},C_{i+1},\ldots,C_{\mathtt{shn}(E)}} \#\left(\left\{C_i \mid E \star (C_1, \ldots, C_{\mathtt{shn}(E)}) \in S^{n,E}_{i-1}, t_{\text{HARD},k} \not\preceq C_i\right\}\right)}{\sum_{C_1,\ldots,C_{i-1},C_{i+1},\ldots,C_{\mathtt{shn}(E)}} \#\left(C_i \mid E \star (C_1, \ldots, C_{\mathtt{shn}(E)}) \in S^{n,E}_{i-1}\right)}$$
$$\leq \max_{C_1,\ldots,C_{i-1},C_{i+1},\ldots,C_{\mathtt{shn}(E)}} \frac{\#\left(\left\{C_i \mid E \star (C_1, \ldots, C_{\mathtt{shn}(E)}) \in S^{n,E}_{i-1}, t_{\text{HARD},k} \not\preceq C_i\right\}\right)}{\#\left(\left\{C_i \mid E \star (C_1, \ldots, C_{\mathtt{shn}(E)}) \in S^{n,E}_{i-1}\right\}\right)}$$
$$\left(\text{by } \frac{\Sigma_j s_j}{\Sigma_j r_j} \leq \max_j \frac{s_j}{r_j} \text{ if } r_j > 0, s_j \geq 0 \text{ for every } j\right)$$

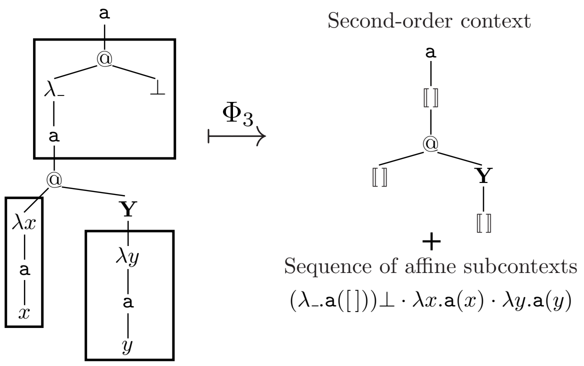

a
@
λ_
⊥
a
@
λx
a
x
Y
λy
a
y
Φ3
Second-order context
a
[]
@
[]
Y
[]
+
Sequence of affine subcontexts
(λ_.a([]))⊥ · λx.a(x) · λy.a(y)

Figure 1 An example of term decomposition. The parts surrounded by rectangles on the left hand side show the extracted affine subcontexts, and the remaining part of the tree is the second-order tree context.

$$
\begin{aligned}
&\le \max_{C_1,\dots,C_{i-1},C_{i+1},\dots,C_{\mathtt{shn}(E)}} \frac{\#\left(\left\{C_i \mid E \star (C_1,\dots,C_{\mathtt{shn}(E)}) \in S^{n,E}_{i-1}\right\}\right) - 1}{\#\left(\left\{C_i \mid E \star (C_1,\dots,C_{\mathtt{shn}(E)}) \in S^{n,E}_{i-1}\right\}\right)}\\
&\qquad \text{(by Lemma 14)}\\
&= \max_{C_1,\dots,C_{i-1},C_{i+1},\dots,C_{\mathtt{shn}(E)}} 1 - \frac{1}{\#\left(\left\{C_i \mid E \star (C_1,\dots,C_{\mathtt{shn}(E)}) \in S^{n,E}_{i-1}\right\}\right)}\\
&\le 1 - \frac{1}{\gamma^{rm}}
\end{aligned}
$$

for some $\gamma > 1$. Here, $C_1,\dots,C_{i-1},C_{i+1},\dots,C_{\mathtt{shn}(E)}$ in the subscript of max range over the set of contexts for which the denominator $\#\left(\left\{C_i \mid E \star (C_1,\dots,C_{\mathtt{shn}(E)}) \in S^{n,E}_{i}\right\}\right)$ is non-zero. The last inequality follows from Property (ii) of Φ_m (that the size of C_i is bounded by rm), and the fact that the number of contexts of a given size ℓ can be bounded by γ^ℓ for some γ.

Thus, we have:

$$
\begin{aligned}
\frac{\#\left(\{t \in \hat{\Lambda}_n(k,\iota,\xi) \mid t_{\textsc{Hard},k} \not\precsim t\}\right)}{\#\left(\hat{\Lambda}_n(k,\iota,\xi)\right)} &\le \frac{\sum_{E\in\mathcal{E}^n_m} \#\left(S^{n,E}_{\mathtt{shn}(E)}\right)}{\sum_{E\in\mathcal{E}^n_m} \#\left(S^{n,E}_{0}\right)} \quad \text{(by the properties of } \Phi_m)\\
&\le \max_{E\in\mathcal{E}^n_m} \frac{\#\left(S^{n,E}_{\mathtt{shn}(E)}\right)}{\#\left(S^{n,E}_{0}\right)} \quad \left(\text{by } \frac{\sum_i s_i}{\sum_i r_i} \le \max_i \frac{s_i}{r_i}\right)\\
&= \max_{E\in\mathcal{E}^n_m} \frac{\#\left(S^{n,E}_{1}\right)}{\#\left(S^{n,E}_{0}\right)} \cdot \frac{\#\left(S^{n,E}_{2}\right)}{\#\left(S^{n,E}_{1}\right)} \cdot \cdots \cdot \frac{\#\left(S^{n,E}_{\mathtt{shn}(E)}\right)}{\#\left(S^{n,E}_{\mathtt{shn}(E)-1}\right)}.\\
&\le \max_{E\in\mathcal{E}^n_m} (1 - \frac{1}{\gamma^{rm}})^{\mathtt{shn}(E)}\\
&\le (1 - \frac{1}{\gamma^{rm}})^{\frac{n}{2rm}} \quad \text{(by Property (iv) of } \Phi_m)\\
&\to 0 \text{ (as } n \to \infty)
\end{aligned}
$$

Therefore, we obtain

$$\lim_{n\to\infty} \frac{\#\Big(\{t \in \hat{\Lambda}_n(k,\iota,\xi) \mid t \succeq t_{\mathrm{HARD},k}\}\Big)}{\#\Big(\hat{\Lambda}_n(k,\iota,\xi)\Big)} = 1$$

as required.

5 Intersection Types for Minimal Terms

This section introduces an intersection type system for characterizing minimal terms, so that a closed, ground-typed term is typable just if it is minimal. For the terms in Example 10, $(\lambda x.\lambda y.x)$ $\mathtt{a}$ $\bot$ is typable in the intersection type system but $(\lambda x.\lambda y.x)$ $\mathtt{a}$ $\mathtt{a}$ is not. This intersection type system will serve as a key tool for proving Lemma 14 in Section 6.

The sets of *prime intersection types* and *intersection types* are defined by:

$$\tau,\sigma \text{ (prime intersection types) } ::= \mathsf{o} \mid \theta \to \tau \qquad \theta,\delta \text{ (intersection types) } ::= \textstyle\bigwedge^{\kappa}\{\tau_1,\dots,\tau_n\}.$$

Here, $n \geq 0$: Intuitively, o is the type of terms that generate non-$\bot$ trees that will occur in the value tree. The intersection type $\bigwedge^{\kappa}\{\tau_1,\dots,\tau_n\}$ describes terms that are used according to types $\tau_1,\dots,\tau_n$. In particular, $\bigwedge^{\kappa}\emptyset$ is the type of terms that are not used. Note that $\{\tau_1,\dots,\tau_n\}$ in $\bigwedge^{\kappa}\{\tau_1,\dots,\tau_n\}$ is a *set* rather than a multiset; thus we consider here idempotent intersection types. The superscript κ (which ranges over the set of simple types) is used for distinguishing between, for example, $\emptyset^{\mathsf{o}}$ and $\emptyset^{\mathsf{o}\to\mathsf{o}}$; we, however, often omit the superscript.

We often write $\bigwedge^{\kappa}_{i\in[n]}\tau_i$ or $\tau_1 \wedge \cdots \wedge \tau_n$ for $\bigwedge^{\kappa}\{\tau_1,\dots,\tau_n\}$, and $\top^{\kappa}$ (or just $\top$) for $\bigwedge^{\kappa}\emptyset$. For each intersection types $\theta = \bigwedge^{\kappa} S$ and $\delta = \bigwedge^{\kappa} T$, We denote by $\theta \wedge \delta$ the intersection type $\bigwedge^{\kappa}(S \cup T)$. We use $\bar{\theta}$, $\bar{\delta}$ to denote a prime intersection type or an intersection type. An *intersection type environment*, written as Θ or Δ, is a finite partial mapping from $\mathcal{V}$ to the set of intersection types. For each Θ, $x \in \mathcal{V} \setminus \mathrm{dom}(\Theta)$, and θ, we write $(\Theta, x : \theta)$ for $\Theta \uplus \{x \mapsto \theta\}$. The *refinement relation* $\bar{\theta} :: \kappa$ (resp. $\Theta :: \Gamma$) is the least relation closed under the following rules, where $n \geq 0$:

$$\frac{}{\mathsf{o} :: \mathsf{o}} \qquad \frac{\tau_1 :: \kappa \quad \dots \quad \tau_n :: \kappa}{\bigwedge^{\kappa}_{i\in[n]}\tau_i :: \kappa} \qquad \frac{\theta :: \kappa \quad \tau :: \kappa'}{(\theta \to \tau) :: (\kappa \to \kappa')} \qquad \frac{}{\emptyset :: \emptyset} \qquad \frac{\Theta :: \Gamma \quad \theta :: \kappa}{(\Theta, x : \theta) :: (\Gamma, x : \kappa)}.$$

Note that, for each $\bar{\theta}$ (and similarly for Θ), there exists at most one simple type κ such that $\bar{\theta} :: \kappa$. Henceforth we only consider intersection types occurring in this refinement relation (so, we always make the assumption that for each $\bar{\theta}$, $\bar{\theta} :: \kappa$ holds for some κ).

We write $\Theta \wedge \Delta$ for the intersection type environment $\{x \mapsto \Theta(x) \wedge \Delta(x) \mid x \in \mathrm{dom}(\Theta) \cup \mathrm{dom}(\Delta)\}$, where $\Theta(x) = \top^{\kappa}$ (where κ is uniquely determined by $\Delta(x)$) if $x \notin \mathrm{dom}(\Theta)$, and similarly for the case $x \notin \mathrm{dom}(\Delta)$.

The *intersection type judgment relation* $\Theta \vdash t : \bar{\theta}$ is inductively defined by the typing rules in Figure 2. We implicitly assume that, whenever $\Theta \vdash t : \bar{\theta}$ occurs in a rule, $\Gamma \vdash_{\mathrm{ST}} t : \kappa$, $\Theta :: \Gamma$, and $\bar{\theta} :: \kappa$ hold for some Γ and κ; for example, in (Var), it must be the case that $\tau :: \kappa$.

Many of the rules are the same as those of standard intersection type systems, but peculiar to our type system is the use of $\sqcup$ in the rules ($\wedge$) and (**Y**1). In ($\wedge$), the premises say that each t_i is used according to τ_i; think of t_i as a "used" part of some term t such that $t_i \sqsubseteq t$. In the conclusion, those used parts $t_1,\dots,t_n$ are "merged" to obtain $\bigsqcup_{i\in[n]} t_i$ as the used part of t when it is accessed according to types $\tau_1,\dots,\tau_n$. For example, consider the term $\lambda x^{\mathsf{o}\to\mathsf{o}\to\mathsf{o}}.\lambda y^{\mathsf{o}}.\lambda z^{\mathsf{o}}.x\,y\,z$. Then we have $\emptyset \vdash \lambda x^{\mathsf{o}\to\mathsf{o}\to\mathsf{o}}.\lambda y^{\mathsf{o}}.\lambda z^{\mathsf{o}}.x\,y\,\bot : (\mathsf{o} \to \top \to \mathsf{o}) \to$

$$\frac{}{x : \bigwedge\{\tau\} \vdash x^{\kappa} : \tau}\,(\text{Var}) \qquad \frac{\Theta, x : \theta \vdash t : \tau}{\Theta \vdash \lambda x.t : \theta \to \tau}\,(\text{Abs1}) \qquad \frac{\Theta \vdash t : \tau}{\Theta \vdash \lambda \bar{x}.t : \top \to \tau}\,(\text{Abs2})$$

$$\frac{\Theta \vdash t : \theta \to \tau \quad \Delta \vdash s : \theta}{\Theta \wedge \Delta \vdash t\, s : \tau}\,(\text{App}) \qquad \frac{\Theta \vdash t_1\,(\mathbf{Y} t_2) : \tau}{\Theta \vdash \mathbf{Y}(t_1 \sqcup t_2) : \tau}\,(\mathbf{Y}1) \qquad \frac{\Theta \vdash t\,\bot : \tau}{\Theta \vdash \mathbf{Y} t : \tau}\,(\mathbf{Y}2)$$

$$\frac{\Theta_1 \vdash t_1 : \theta_1 \ \ \ldots \ \ \Theta_n \vdash t_n : \theta_n}{\bigwedge_{i \in [n]} \Theta_i \vdash a(t_1, \ldots, t_n) : \mathtt{o}}\,(a) \qquad \frac{\Theta_1 \vdash t_1 : \tau_1 \ \ \ldots \ \ \Theta_n \vdash t_n : \tau_n}{\bigwedge_{i \in [n]} \Theta_i \vdash \bigsqcup_{i \in [n]} t_i : \bigwedge_{i \in [n]} \tau_i}\,(\wedge) \qquad \frac{\Theta \vdash t : \bar{\theta}}{\Theta, x : \top \vdash t : \bar{\theta}}\,(\top)$$

Figure 2 The intersection type system for the minimality (see Section 3.1 for the operator $\sqcup$).

$\mathtt{o} \to \top \to \mathtt{o}$ and $\emptyset \vdash \lambda x^{\mathtt{o} \to \mathtt{o} \to \mathtt{o}}.\lambda y^{\mathtt{o}}.\lambda z^{\mathtt{o}}.x \perp z : (\top \to \mathtt{o} \to \mathtt{o}) \to \top \to \mathtt{o} \to \mathtt{o}$. From those judgments, we obtain

$$\emptyset \vdash \lambda x^{\mathtt{o} \to \mathtt{o} \to \mathtt{o}}.\lambda y^{\mathtt{o}}.\lambda z^{\mathtt{o}}.x\, y\, z : ((\mathtt{o} \to \top \to \mathtt{o}) \to \mathtt{o} \to \top \to \mathtt{o}) \wedge ((\top \to \mathtt{o} \to \mathtt{o}) \to \top \to \mathtt{o} \to \mathtt{o})$$

by using $(\wedge)$. Note that when $n = 0$, the rule $(\wedge)$ allows us to derive $\emptyset \vdash \bot : \top$.

There are two typing rules for $\mathbf{Y}\, t$. The rule $(\mathbf{Y}1)$ covers the case where $\mathbf{Y}\, t$ is reduced to $t(\mathbf{Y}\, t)$ and the argument $\mathbf{Y}\, t$ is used again; t_1 in the premise represents the used part of the head occurrence of t, whereas t_2 represents the used part of the occurrence of t in the argument $\mathbf{Y}\, t$. In the conclusion, both parts are merged to obtain $t_1 \sqcup t_2$ as the used part of t. For example, consider $\mathbf{Y}\, t$ where $t = \lambda f.\lambda x.\lambda y.\mathtt{b}\, x\, (f \perp y)$ and $\tau = \mathtt{o} \to \bot \to \mathtt{o}$. Then we have $\emptyset \vdash t_1(\mathbf{Y}\, t_2) : \tau$ for $t_1 = \lambda f.\lambda x.\lambda y.\mathtt{b}\, x\, (f \perp \perp)$ and $t_2 = \lambda f.\lambda x.\lambda y.\mathtt{b} \perp (f \perp \perp)$. By using $(\mathbf{Y}1)$, we obtain $\emptyset \vdash \mathbf{Y}(\lambda f.\lambda x.\lambda y.\mathtt{b}\, x\, (f \perp \perp)) : \tau$, which correctly models the used part of $\mathbf{Y}\, t$. The rule $(\mathbf{Y}2)$ is for the case where recursive calls do not contribute to the result. For example, consider the term $t = \mathbf{Y}(\lambda x.\mathtt{a}(\bot))$. Then from $\emptyset \vdash (\lambda x.\mathtt{a}(\bot))\bot : \mathtt{o}$, we obtain $\emptyset \vdash t : \mathtt{o}$.

The theorem below states that the minimality is correctly characterized by our intersection type system. See Appendix A for an outline of a proof; the full proof is found in [20].

▶ **Theorem 18** (soundness and completeness). *For every closed and ground-typed term t, t is minimal if and only if $\emptyset \vdash t : \bar{\theta}$ for some $\bar{\theta}$.*

We give examples of type derivations below.

▶ **Example 19** (cf. Example 10). Let $t = (\lambda x^{\mathtt{o}}.\lambda y^{\mathtt{o}}.x^{\mathtt{o}})\ \mathtt{a} \perp^{\mathtt{o}}$ and $s = (\lambda x^{\mathtt{o}}.\lambda y^{\mathtt{o}}.x^{\mathtt{o}})\ \mathtt{a}\ \mathtt{a}$. Then we can show that t is minimal by giving the derivation tree of $\emptyset \vdash t : \mathtt{o}$ as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : \bigwedge\{\mathtt{o}\} \vdash x^{\mathtt{o}} : \mathtt{o}}\ (\text{Var})}{x : \bigwedge\{\mathtt{o}\} \vdash \lambda y^{\mathtt{o}}.x^{\mathtt{o}} : \top \to \mathtt{o}}\ (\text{Abs2})}{\emptyset \vdash \lambda x^{\mathtt{o}}.\lambda y^{\mathtt{o}}.x^{\mathtt{o}} : \bigwedge\{\mathtt{o}\} \to \top \to \mathtt{o}}\ (\text{Abs1}) \qquad \dfrac{\dfrac{}{\emptyset \vdash \mathtt{a} : \mathtt{o}}\ (\mathtt{a})}{\emptyset \vdash \mathtt{a} : \bigwedge\{\mathtt{o}\}}\ (\wedge)}{\emptyset \vdash (\lambda x^{\mathtt{o}}.\lambda y^{\mathtt{o}}.x^{\mathtt{o}})\ \mathtt{a} : \top \to \mathtt{o}}\ (\text{App}) \qquad \dfrac{}{\emptyset \vdash \perp^{\mathtt{o}} : \top}\ (\wedge)}{\emptyset \vdash (\lambda x^{\mathtt{o}}.\lambda y^{\mathtt{o}}.x^{\mathtt{o}})\ \mathtt{a} \perp^{\mathtt{o}} : \mathtt{o}}\ (\text{App})$$

In contrast, $\emptyset \nvdash s : \mathtt{o}$, because $x : \bigwedge\{\mathtt{o}\}, y : \bigwedge\{\mathtt{o}\} \nvdash x^{\mathtt{o}} : \mathtt{o}$.

The following is a more complex example, where intersection types play an important role.

▶ **Example 20.** Let $s = (\lambda f^{(\mathsf{o}\to\mathsf{o}\to\mathsf{o})\to\mathsf{o}}.\mathsf{a}(f\,\mathsf{fst}, f\,\mathsf{snd}))$, $u = (\lambda g^{\mathsf{o}\to\mathsf{o}\to\mathsf{o}}.g\,\mathtt{b}\,\mathtt{c})$, and $t = s\,u$, where $\mathsf{fst} = \lambda x^{\mathsf{o}}.\lambda y^{\mathsf{o}}.x^{\mathsf{o}}$ and $\mathsf{snd} = \lambda x^{\mathsf{o}}.\lambda y^{\mathsf{o}}.y^{\mathsf{o}}$. Then $\emptyset \vdash t : \mathsf{o}$ is derived from the following two derivations by applying (App), where $\tau_1 = \wedge\{\mathsf{o}\} \to \top \to \mathsf{o}$ and $\tau_2 = \top \to \wedge\{\mathsf{o}\} \to \mathsf{o}$. Hence this t is minimal.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : \wedge\{\wedge\{\tau_1\} \to \mathsf{o}\} \vdash f : \wedge\{\tau_1\} \to \mathsf{o}}\,(\mathrm{Var}) \qquad \dfrac{\dfrac{\text{(similarly to Example 19)}}{\emptyset \vdash \mathsf{fst} : \tau_1}}{\emptyset \vdash \mathsf{fst} : \wedge\{\tau_1\}}\,(\wedge)}{f : \wedge\{\wedge\{\tau_1\} \to \mathsf{o}\} \vdash f\,\mathsf{fst} : \mathsf{o}}\,(\mathrm{App})}{f : \wedge\{\wedge\{\tau_1\} \to \mathsf{o}\} \vdash f\,\mathsf{fst} : \wedge\{\mathsf{o}\}}\,(\wedge) \qquad \dfrac{\text{(similarly to the left)}}{f : \wedge\{\wedge\{\tau_2\} \to \mathsf{o}\} \vdash f\,\mathsf{snd} : \wedge\{\mathsf{o}\}}\,(\wedge)}{\dfrac{f : \wedge\{\wedge\{\tau_1\} \to \mathsf{o}, \wedge\{\tau_2\} \to \mathsf{o}\} \vdash \mathsf{a}(f\,\mathsf{fst}, f\,\mathsf{snd}) : \mathsf{o}}{\emptyset \vdash \lambda f.\mathsf{a}(f\,\mathsf{fst}, f\,\mathsf{snd}) : \bigwedge_{l\in[2]}\{\wedge\{\tau_l\} \to \mathsf{o}\} \to \mathsf{o}}\,(\mathrm{Abs1})}\,(\mathsf{a})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{g : \wedge\{\tau_1\} \vdash g : \wedge\{\mathsf{o}\} \to \top \to \mathsf{o}}\,(\mathrm{Var}) \qquad \dfrac{\dfrac{}{\emptyset \vdash \mathtt{b} : \mathsf{o}}\,(\mathtt{b})}{\emptyset \vdash \mathtt{b} : \wedge\{\mathsf{o}\}}\,(\wedge)}{g : \wedge\{\tau_1\} \vdash g\,\mathtt{b} : \top \to \mathsf{o}}\,(\mathrm{App}) \qquad \dfrac{}{\emptyset \vdash \bot : \top}\,(\wedge)}{g : \wedge\{\tau_1\} \vdash g\,\mathtt{b}\,\bot : \mathsf{o}}\,(\mathrm{App})}{\emptyset \vdash \lambda g.g\,\mathtt{b}\,\bot : \wedge\{\tau_1\} \to \mathsf{o}}\,(\mathrm{Abs1}) \qquad \dfrac{\dfrac{\text{(similarly to the left)}}{g : \wedge\{\tau_2\} \vdash g\,\bot\,\mathtt{c} : \mathsf{o}}\,(\mathrm{App})}{\emptyset \vdash \lambda g.g\,\bot\,\mathtt{c} : \wedge\{\tau_2\} \to \mathsf{o}}\,(\mathrm{Abs1})}{\emptyset \vdash \lambda g.g\,\mathtt{b}\,\mathtt{c} : \bigwedge_{l\in[2]}\{\wedge\{\tau_l\} \to \mathsf{o}\}}\,(\wedge)$$

Note that the term u is "used" in two different ways in t: in $f\,\mathsf{fst}$, the subterm $\mathtt{b}$ is used, whereas in $f\,\mathsf{snd}$, the subterm $\mathtt{c}$ is used.

6 Proof of the Main Lemma (Lemma 14)

In this section, we prove Lemma 14 by using the intersection type system from the previous section. Recall that we need to prove that if $E[C] \in \hat{\Lambda}_n(k, \iota, \xi)$, then there is a context $D \succeq t_{\mathrm{HARD},k}$ such that $E[D] \in \hat{\Lambda}_n(k, \iota, \xi)$. Thanks to the result of the previous section, $E[C] \in \hat{\Lambda}_n(k, \iota, \xi)$ implies that $E[C]$ is typable in the intersection type system. Thus, it suffices to construct D of the same size such that (i) C has "the same typing properties" as D, and (ii) D contains $t_{\mathrm{HARD},k}$. To this end, we first extend the notion of types to those of contexts (called *context-types*) in Section 6.1. We then show in Section 6.2 that we can indeed construct a context D that has the same context types as C, and prove Lemma 14.

6.1 Context-Types

For each affine-context C, we write $C \lhd_{\mathrm{ST}} \{\langle\Gamma'_1, \kappa'_1\rangle, \dots, \langle\Gamma'_n, \kappa'_n\rangle\} \Rrightarrow \langle\Gamma, \kappa\rangle$ if there is a derivation tree of $\Gamma \vdash_{\mathrm{ST}} C[\mathtt{x}] : \kappa$ with the assumptions $\{\Gamma'_1 \vdash_{\mathrm{ST}} \mathtt{x} : \kappa'_1, \dots, \Gamma'_n \vdash_{\mathrm{ST}} \mathtt{x} : \kappa'_n\}$, where n is at most one and $\mathtt{x}$ is a variable not occurring in C. Intuitively, it means that there is a derivation tree of $\Gamma' \vdash_{\mathrm{ST}} C : \kappa'$ with the assumptions $\{\Gamma'_1 \vdash_{\mathrm{ST}} [\,] : \kappa'_1, \dots, \Gamma'_n \vdash_{\mathrm{ST}} [\,] : \kappa'_n\}$ (see also Example 21). We often write $t \lhd_{\mathrm{ST}} \tilde{\theta}$ for $t \lhd_{\mathrm{ST}} \emptyset \Rrightarrow \tilde{\theta}$. We use $\tilde{\kappa}$ to denote a pair $\langle\Gamma, \kappa\rangle$ and use $\tilde{\nu}$ to denote a $\{\langle\Gamma'_1, \kappa'_1\rangle, \dots, \langle\Gamma'_n, \kappa'_n\rangle\} \Rrightarrow \langle\Gamma, \kappa\rangle$. Note that C is a term (resp. a linear-context) if $C \lhd_{\mathrm{ST}} \{\langle\Gamma'_1, \kappa'_1\rangle, \dots, \langle\Gamma'_n, \kappa'_n\rangle\} \Rrightarrow \langle\Gamma, \kappa\rangle$ holds for $n = 0$ (resp. $n = 1$). Below we extend the notion of $\lhd_{\mathrm{ST}}$ to the intersection type system. The set of *(affine-)context-types*, ranged over by $\tilde{\mu}$, is defined as follows, where $n \geq 0$ and we may write $\tilde{\theta}^+$ for $\tilde{\theta}$ if $\tilde{\theta} \neq \emptyset$:

$$\tilde{\tau} ::= \langle\Theta, \tau\rangle \qquad \tilde{\theta} ::= \{\tilde{\tau}_1, \dots, \tilde{\tau}_n\} \qquad \tilde{\pi} ::= \tilde{\tau} \mid \tilde{\theta}^+ \qquad \tilde{\mu} ::= \tilde{\theta} \Rrightarrow \tilde{\pi}.$$

For $\tilde{\mu}$, intuitively, $\tilde{\theta} \Rrightarrow \tilde{\tau}$ denotes the pair of the assumptions ($\tilde{\theta}$) and the conclusion ($\tilde{\tau}$) of a derivation tree, and $\tilde{\theta} \Rrightarrow \tilde{\theta}^+$ denotes the pair of the assumptions ($\tilde{\theta}$) and the conclusions

($\tilde{\theta}^+$) of one or more derivation trees. The *refinement relation* is defined as the least relation closed under the following rules, where $n \geq 0$:

$$\frac{\Theta :: \Gamma \qquad \tau :: \kappa}{\langle \Theta, \tau \rangle :: \langle \Gamma, \kappa \rangle} \quad \frac{\tilde{\tau}_1 :: \langle \Gamma, \kappa \rangle \qquad \dots \qquad \tilde{\tau}_n :: \langle \Gamma, \kappa \rangle}{\{\tilde{\tau}_1, \dots, \tilde{\tau}_n\} :: \langle \Gamma, \kappa \rangle} \quad \frac{\tilde{\theta}' :: \langle \Gamma', \kappa' \rangle \qquad \tilde{\pi} :: \langle \Gamma, \kappa \rangle}{\tilde{\theta}' \Rrightarrow \tilde{\pi} :: \langle \Gamma', \kappa' \rangle \Rrightarrow \langle \Gamma, \kappa \rangle}.$$

Henceforth we only consider context-types occurring in this refinement relation (so, we always make the assumptions that for each $\tilde{\theta}' \Rrightarrow \tilde{\theta}$, for some $\langle \Gamma, \kappa \rangle$ and $\langle \Gamma', \kappa' \rangle$, $\tilde{\theta} :: \langle \Gamma, \kappa \rangle$ and $\tilde{\theta}' :: \langle \Gamma', \kappa' \rangle$). For each affine-context C, we write $C \lhd \{\langle \Theta'_1, \tau'_1 \rangle, \dots, \langle \Theta'_n, \tau'_n \rangle\} \Rrightarrow \langle \Theta, \tau \rangle$ if there is a derivation tree of $\Theta \vdash C[\mathtt{x}] : \tau$ with the assumptions $\{\Theta'_1 \vdash \mathtt{x} : \tau'_1, \dots, \Theta'_n \vdash \mathtt{x} : \tau'_n\}$, where $\mathtt{x}$ is a variable not occurring in C. For $n \geq 1$, we write $(\bigsqcup_{i \in [n]} C_i) \lhd (\bigcup_{i \in [n]} \tilde{\theta}'_i) \Rrightarrow \{\tilde{\tau}_1, \dots, \tilde{\tau}_n\}$ if $C_i \lhd \tilde{\theta}'_i \Rrightarrow \tilde{\tau}_i$ for each $i \in [n]$. We often write $t \lhd \tilde{\theta}$ for $t \lhd \emptyset \Rrightarrow \tilde{\theta}$.

▶ **Example 21.** Let $C = (\lambda g^{\kappa_0}.[\,]\, \mathtt{b}\, \mathtt{c})$, where $\kappa_0 = \mathtt{o} \to \mathtt{o} \to \mathtt{o}$. Note that $C[g]$ is the term u in Example 20. Then, we have $C \lhd_{\mathrm{ST}} \{\langle (g : \kappa_0), \kappa_0 \rangle\} \Rrightarrow \langle \emptyset, \kappa_0 \to \mathtt{o} \rangle$ by $g : \kappa_0 \vdash_{\mathrm{ST}} g : \kappa_0$ and $\emptyset \vdash_{\mathrm{ST}} C[g] : \kappa_0 \to \mathtt{o}$. Also, we have $C \lhd \{\langle (g : \tau_1), \tau_1 \rangle, \langle (g : \tau_2), \tau_2 \rangle\} \Rrightarrow \langle \emptyset, \bigwedge_{l \in [2]} \{\wedge\{\tau_l\} \to \mathtt{o}\} \rangle$ by using the derivation tree in Example 20 with regarding g as a hole, where $\tau_1 = \wedge\{\mathtt{o}\} \to \top \to \mathtt{o}$ and $\tau_2 = \top \to \wedge\{\mathtt{o}\} \to \mathtt{o}$. Furthermore, we also have $C \lhd \{\langle (g : \tau_1), \tau_1 \rangle, \langle (g : \tau_2), \tau_2 \rangle\} \Rrightarrow \{\langle \emptyset, \wedge\{\tau_1\} \to \mathtt{o} \rangle, \langle \emptyset, \wedge\{\tau_2\} \to \mathtt{o} \rangle\}$. It is because C is the join of $C_1 = (\lambda g^{\kappa_0}.[\,]\, \mathtt{b} \perp^{\mathtt{o}})$ and $C_2 = (\lambda g^{\kappa_0}.[\,] \perp^{\mathtt{o}} \mathtt{c})$. Here, note that $C[g] = C_1[g] \sqcup C_2[g]$, $C_1 \lhd \{\langle (g : \tau_1), \tau_1 \rangle\} \Rrightarrow \langle \emptyset, \wedge\{\tau_1\} \to \mathtt{o} \rangle$, and $C_2 \lhd \{\langle (g : \tau_2), \tau_2 \rangle\} \Rrightarrow \langle \emptyset, \wedge\{\tau_2\} \to \mathtt{o} \rangle$.

Below we list a few properties (see Appendix B for the proofs).

▶ **Proposition 22** (substitution). *Suppose that C is a linear-context. If $C \lhd \tilde{\theta}' \Rrightarrow \tilde{\theta}$ and $C' \lhd \tilde{\theta}'' \Rrightarrow \tilde{\theta}'$, then $C[C'] \lhd \tilde{\theta}'' \Rrightarrow \tilde{\theta}$.*

▶ **Proposition 23** (inverse substitution). *Suppose that C is a linear-context. If $C[C'] \lhd \tilde{\theta}'' \Rrightarrow \tilde{\theta}$, then $C \lhd \tilde{\theta}' \Rrightarrow \tilde{\theta}$ and $C' \lhd \tilde{\theta}'' \Rrightarrow \tilde{\theta}'$ for some $\tilde{\theta}'$.*

These properties enable us to replace contexts preserving the minimality. For example, given $\emptyset \vdash C[D[t]] : \mathtt{o}$ (i.e., $C[D[t]]$ is minimal); then by Proposition 23, $C \lhd \tilde{\theta} \Rrightarrow \{\langle \emptyset, \mathtt{o} \rangle\}$, $D \lhd \tilde{\theta}' \Rrightarrow \tilde{\theta}$, and $t \lhd \tilde{\theta}'$ for some $\tilde{\theta}$ and $\tilde{\theta}'$; then by Proposition 22, $C[D'[t]] \lhd \{\langle \emptyset, \mathtt{o} \rangle\}$ (hence, $C[D'[t]]$ is also minimal) for each linear context $D' \lhd \tilde{\theta}' \Rrightarrow \tilde{\theta}$.

In the following subsection, we will show in Lemma 25 that for the term $C[D[t]]$ in the above, if the size $|D|$ is sufficiently large, then one can choose D' as a term satisfying (i) $D' \succeq t_{\mathrm{HARD},k}$, and (ii) $|D'| = |D|$. Thus, from a term $C[D[t]] \in \hat{\Lambda}_n(k, \iota, \xi)$ such that $|D|$ is sufficiently large, one can construct a term $C[D'[t]]$ satisfying (i) $C[D'[t]] \succeq t_{\mathrm{HARD},k}$, and (ii) $|C[D'[t]]| = |C[D[t]]|$ and $C[D'[t]]$ is minimal (hence, $C[D'[t]] \in \hat{\Lambda}_n(k, \iota, \xi)$). This transformation method will help us to show Lemma 14.

6.2 Proof of Lemma 14

Here, we fix parameters k, ι, and ξ. W.l.o.g., in the following, we only consider terms, contexts, and environments having only variables in a fixed set $\mathcal{V}_\xi \triangleq \{z_1, \dots, z_\xi\}$ (of size ξ). We say that $\langle \Gamma, \kappa \rangle$ is *($\langle k, \iota, \xi \rangle$-)bounded* if $\max\{\mathtt{ord}(\kappa') \mid \kappa' \in \{\kappa\} \cup \mathrm{rng}(\Gamma)\} \leq k$ and $\max\{\mathtt{iar}(\kappa') \mid \kappa' \in \{\kappa\} \cup \mathrm{rng}(\Gamma)\} \leq \iota$; and that $\langle \Gamma', \kappa' \rangle \Rrightarrow \langle \Gamma, \kappa \rangle$ is *bounded* if both $\langle \Gamma', \kappa' \rangle$ and $\langle \Gamma, \kappa \rangle$ are; and that a context-type $\tilde{\mu}$ is *bounded* if the $\tilde{\nu}$ such that $\tilde{\mu} :: \tilde{\nu}$ is bounded. We also say that t is *bounded* if $\mathtt{ord}(t) \leq k$ and $\mathtt{iar}(t) \leq \iota$; and that a linear-context C is *bounded* if $C[\perp]$ is. Also, we use $\mathtt{a}$ (resp. $\mathtt{b}$, $\mathtt{c}$) to denote a tree constructor of arity 0 (resp. 2, 1).

The following technical lemma allows conversion between a ground-typed term and a term of a required typing property: see Appendix C for a proof.

▶ **Lemma 24.**

(1) *Suppose that* $\tilde{\theta}^+ :: \langle \Gamma, \kappa \rangle$ *is bounded. If* $\#(\mathrm{dom}(\Gamma)) < \xi$ *or* $\mathtt{ar}\,(\kappa) < \iota$, *then there exists a bounded linear-context* $C_{\tilde{\theta}^+}$ *such that* $C_{\tilde{\theta}^+} \lhd \{\langle \emptyset, \mathtt{o} \rangle\} \Rrightarrow \tilde{\theta}^+$.

(2) *Suppose that* $\tilde{\theta}$ *is bounded. Then, there exists a bounded affine-context* $D_{\tilde{\theta}}$ *such that* $D_{\tilde{\theta}} \lhd \tilde{\theta} \Rrightarrow \{\langle \emptyset, \mathtt{o} \rangle\}$.

By Lemma 24, from a given bounded context-type $\tilde{\theta}' \Rrightarrow \tilde{\theta}^+$, one can construct a bounded affine-context having this context-type as $C_{\tilde{\theta}^+}[D_{\tilde{\theta}'}]$, except the case of that $\#(\mathrm{dom}(\Gamma)) = \xi$ and $\mathtt{ar}\,(\kappa) = \iota$. See Appendix C.1 for the boundary case; actually, terms having such context-type are of a special form.

The following is the key lemma, which shows that for any bounded context-type, one can construct a context D that has the context-type and contains the hard term $t_{\textsc{Hard},k}$.

▶ **Lemma 25.** *Suppose that* $C \lhd \tilde{\theta}' \Rrightarrow \tilde{\theta}^+$ *for some bounded affine-context* C. *Then for some* m_0, *for every* $m \geq m_0$, *there exists a bounded affine-context* D *of size* m *such that* $D \lhd \tilde{\theta}' \Rrightarrow \tilde{\theta}^+$ *and* $D \succeq t_{\textsc{Hard},k}$.

Proof. Let $\langle \Gamma, \kappa \rangle$ be such that $\tilde{\theta}^+ \lhd \langle \Gamma, \kappa \rangle$. Note that $\tilde{\theta}'$ and $\tilde{\theta}^+$ are also bounded.

(a) $\#(\mathrm{dom}(\Gamma)) < \xi$ or $\mathtt{ar}\,(\kappa) < \iota$: For each $l \geq 0$, let D_l be as follows, where $\mathtt{c}^l(\mathtt{a})$ is the term $\mathtt{c}(\dots \mathtt{c}(\mathtt{a}) \dots)$ that $\mathtt{c}$ occurs l times and $D_{\tilde{\theta}'}$ and $C_{\tilde{\theta}^+}$ are the ones in Lemma 24:

$$D_l \triangleq C_{\tilde{\theta}^+}[\mathtt{b}(t_{\textsc{Hard},k}, \mathtt{b}(\mathtt{c}^l(\mathtt{a}), []))][D_{\tilde{\theta}'}].$$

Then $D_l \succeq t_{\textsc{Hard},k}$ is obvious, and $D_l \lhd \tilde{\theta}' \Rrightarrow \tilde{\theta}^+$ by Proposition 22 (since $\mathtt{b}(t_{\textsc{Hard},k}, \mathtt{b}(\mathtt{c}^l, [])) \lhd \{\langle \emptyset, \mathtt{o} \rangle\} \Rrightarrow \{\langle \emptyset, \mathtt{o} \rangle\}$). Therefore, the claim has been proved by using these $D_1, D_2, \cdots$.

(b) Otherwise: Then, $\Gamma \vdash_{\mathrm{ST}} C[\bot] : \kappa$, $C[\bot]$ is bounded, and $\#(\mathrm{dom}(\Gamma)) = \xi$ and $\mathtt{ar}\,(\kappa) = \iota$, so C should be of the form $\lambda\_.C_0$ (see Appendix C.1). By Proposition 23, $C_0 \lhd \tilde{\theta}' \Rrightarrow \tilde{\theta}_0$ and $\lambda\_.[] \lhd \tilde{\theta}_0 \Rrightarrow \tilde{\theta}$ for some $\tilde{\theta}_0$. Then $\mathtt{ar}\,(C_0) < \mathtt{ar}\,(C) \leq \iota$ and $\tilde{\theta}_0 \neq \emptyset$ by $C_0 \neq \bot$ (since $\xi > 0$). Therefore by (a), for some m_0', there is $\{D_l'\}_{l \geq m_0'}$ such that $D_l' \lhd \tilde{\theta}' \Rrightarrow \tilde{\theta}_0$, $D_l' \succeq t_{\textsc{Hard},k}$, and $|D_l'| = l$ for each $l \geq m_0'$. Let $D_l = \lambda\_.D_l'$. Then $D_l \succeq t_{\textsc{Hard},k}$ is obvious, and $D_l \lhd \tilde{\theta}' \Rrightarrow \tilde{\theta}^+$ by Proposition 22. Therefore, the claim has been proved by using these $D_{m_0'}, D_{m_0'+1}, \cdots$. ◀

We are now ready to prove the main lemma.

Proof (of Lemma 14). Let $m \triangleq \max\{m_{\tilde{\theta}' \Rrightarrow \tilde{\theta}^+} \mid C \lhd \tilde{\theta}' \Rrightarrow \tilde{\theta}^+ \text{ for some bounded } C\}$, where each $m_{\tilde{\theta}' \Rrightarrow \tilde{\theta}^+}$ is the m_0 in Lemma 25. Indeed such m exists, since the number of bounded context-types is finite. Recall $E[C] \in \hat{\Lambda}_n(k, \iota, \xi)$. Let $\tilde{E}$ be a linear-context such that $E[C] = \tilde{E}[C[t]]$ for some t or $E[C] = \tilde{E}[C]$. For the sake of brevity, we only write the case of that C is linear-context (i.e., $E[C] = \tilde{E}[C[t]]$). Since $\tilde{E}[C[t]]$ is minimal, $\emptyset \vdash \tilde{E}[C[t]] : \tilde{\theta}$ for some $\tilde{\theta} :: \mathtt{o}$ (Theorem 18). Then $\tilde{E}[C[t]] \lhd \emptyset \Rrightarrow \{\langle \emptyset, \mathtt{o} \rangle\}$ (by $\tilde{E}[C[t]] \neq \bot$). By Proposition 23, there exist $\tilde{\theta}$ and $\tilde{\theta}'$ such that $\tilde{E} \lhd \tilde{\theta} \Rrightarrow \{\langle \emptyset, \mathtt{o} \rangle\}$, $C \lhd \tilde{\theta}' \Rrightarrow \tilde{\theta}$, and $t \lhd \emptyset \Rrightarrow \tilde{\theta}'$. By Lemma 25 (and $C \neq \bot$), there exists a bounded linear-context $D \lhd \tilde{\theta}' \Rrightarrow \tilde{\theta}$ such that $D \succeq t_{\textsc{Hard},k}$ and $|D| = |C|$. Therefore $\tilde{E}[D[t]] \lhd \emptyset \Rrightarrow \{\langle \emptyset, \mathtt{o} \rangle\}$ (hence, $\emptyset \vdash \tilde{E}[D[t]] : \wedge\{\mathtt{o}\}$) by Proposition 22, and thus $E[D]$ is minimal (Theorem 18). Hence, $E[D] \in \hat{\Lambda}_n(k, \iota, \xi)$. ◀

7 Related Work

Ong [21] proved the k-EXPTIME completeness of higher-order model checking. There have also been results on parameterized complexity [15, 17, 18] and the complexity of subclasses of the problem [18, 5]. To our knowledge, however, they are all about the worst-case complexity. Despite the extremely high worst-case complexity, practical model checkers have been developed that run quite fast for typical inputs [14, 4, 23, 28], which has led to the motivating question for our work: is higher-order model checking really hard in the average case?

Technically, closest to ours is the work of Asada et al. [26, 1] on a quantitative analysis of the length of β-reduction sequences of simply-typed λ-terms. In fact, our use of the tree-version of infinite monkey theorem (to show that almost every term contains a "hard" term), as well as the tree decomposition [1] has been inspired by their work and other studies on quantitative analysis of the λ-calculus and combinatory logics [8, 2]. The main new difficulty was that, unlike in the case of the length of β-reduction sequences, even if t is a "hard" term to model-check, a term $C[t]$ that contains t as a subterm may not be hard to model-check, because t may not actually be used in $C[t]$ or may be irrelevant for the property to be checked. This has led us to restrict terms to "minimal ones" that do not contain unnecessary subterms. The restriction turned out to be natural also for our goal: we wish to model the *average* case that arises in the actual applications to program verification, and the restriction to minimal terms helps us exclude out unlikely inputs.

We have used an intersection type system to characterize minimal terms. Related type systems have been studied in the context of useless code elimination [6, 7, 13]. In particular, Daminani [7] also used an intersection type system. To our knowledge, however, previous studies do not provide a *complete* characterization of minimal terms (especially in the presence of recursion).

There has been much interest in the average-case complexity in the field of computational complexity: see [3] for a good survey. In their terminology, our ultimate goal is to answer whether $(\mathrm{HOMC}_k(\cdot,\cdot),\mathcal{U})$ belongs to $\mathrm{Avg}_\delta\mathrm{DTIME}(f(n))$ (the class of distributional problems that can be solved in time $f(n)$ for at least $(1-\delta(n))$-fraction of the inputs of size n),[2] where $\mathrm{HOMC}_k(\cdot,\cdot)$ is the higher-order model checking problem of order k, $\mathcal{U}$ is a uniform distribution on inputs of each size n, δ is a function that is asymptotically smaller than $\lambda n.1$, and $f(n)$ is a function asymptotically much smaller than $\mathit{exp}_k(cn)$ (a k-fold exponential function). The result obtained in the present paper (Theorem 7) is not yet of this form, and is rather a mixture of average-case and worst-case analysis, which may be of independent interest from the perspective of complexity theory.

8 Conclusion

We have studied a mixture of average-case and worst-case complexity of higher-order model checking, and shown that for almost every minimal λY-term t of order-k, the higher-order model checking problem specialized for t is k-EXPTIME hard with respect to the size of a tree automaton. To our knowledge, this is the first result on the average-case hardness of higher-order model checking. To obtain the result, we have given a complete type-based characterization of "minimal" terms that contain no useless subterms, which may be of independent interest. Pure average-case analysis of the hardness of higher-order model checking is left for future work.

[2] A similar notion has also been studied under the name "generic-case complexity" [11].

References

1 Kazuyuki Asada, Naoki Kobayashi, Ryoma Sin’ya, and Takeshi Tsukada. Almost Every Simply Typed Lambda-Term Has a Long Beta-Reduction Sequence. *Logical Methods in Computer Science*, 15(1), 2019. doi:10.23638/LMCS-15(1:16)2019.

2 Maciej Bendkowski, Katarzyna Grygiel, and Marek Zaionc. On the likelihood of normalization in combinatory logic. *J. Log. Comput.*, 27(7):2251–2269, 2017. doi:10.1093/logcom/exx005.

3 Andrej Bogdanov and Luca Trevisan. Average-case complexity. *CoRR*, abs/cs/0606037, 2006. arXiv:cs/0606037.

4 Christopher H. Broadbent and Naoki Kobayashi. Saturation-based model checking of higher-order recursion schemes. In *Proceedings of Computer Science Logic (CSL)*, volume 23 of *LIPIcs*, pages 129–148. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013. doi:10.4230/LIPIcs.CSL.2013.129.

5 Pierre Clairambault, Charles Grellois, and Andrzej S. Murawski. Linearity in higher-order recursion schemes. *PACMPL*, 2(POPL):39:1–39:29, 2018. doi:10.1145/3158127.

6 Mario Coppo, Ferruccio Damiani, and Paola Giannini. Refinement types for program analysis. In *Proceedings of International Static Analysis Symposium (SAS)*, volume 1145 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 1996. doi:10.1007/3-540-61739-6_39.

7 Ferruccio Damiani. A conjunctive type system for useless-code elimination. *Mathematical Structures in Computer Science*, 13(1):157–197, 2003. doi:10.1017/S0960129502003869.

8 René David, Katarzyna Grygiel, Jakub Kozik, Christophe Raffalli, Guillaume Theyssier, and Marek Zaionc. Asymptotically almost all λ-terms are strongly normalizing. *Logical Methods in Computer Science*, 9(1), 2013. doi:10.2168/LMCS-9(1:2)2013.

9 Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 1 edition, 2009. doi:10.1017/CBO9780511801655.

10 Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-36387-4.

11 Ilya Kapovich, Alexei G. Myasnikov, Paul Schupp, and Vladimir Shpilrain. Generic-case complexity, decision problems in group theory and random walks. *CoRR*, abs/math/0203239, 2002. URL: https://arxiv.org/abs/math/0203239.

12 Teodor Knapik, Damian Niwinski, and Pawel Urzyczyn. Higher-order pushdown trees are easy. In *Proceedings of International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 2303 of *Lecture Notes in Computer Science*, pages 205–222. Springer, 2002. doi:10.1007/3-540-45931-6_15.

13 Naoki Kobayashi. Type-based useless-variable elimination. *Higher-Order and Symbolic Computation*, 14(2-3):221–260, 2001. doi:10.1023/A:1012944815270.

14 Naoki Kobayashi. Model-checking higher-order functions. In *Proceedings of ACM SIGPLAN conference on Principles and Practice of Declarative Programming (PPDP)*, pages 25–36. ACM Press, 2009. doi:10.1145/1599410.1599415.

15 Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 416–428. ACM Press, 2009. doi:10.1145/1594834.1480933.

16 Naoki Kobayashi. Model checking higher-order programs. *Journal of the ACM*, 60(3), 2013. doi:10.1145/2487241.2487246.

17 Naoki Kobayashi and C.-H. Luke Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *Proceedings of IEEE Symposium on Logic in Computer Science (LICS)*, pages 179–188. IEEE Computer Society Press, 2009. doi:10.1109/LICS.2009.29.

18 Naoki Kobayashi and C.-H. Luke Ong. Complexity of Model Checking Recursion Schemes for Fragments of the Modal Mu-Calculus. *Logical Methods in Computer Science*, 7(4), 2012. doi:10.2168/LMCS-7(4:9)2011.

19 Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. Predicate abstraction and CEGAR for higher-order model checking. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 222–233. ACM Press, 2011. doi:10.1145/1993498.1993525.

20 Yoshiki Nakamura, Kazuyuki Asada, Naoki Kobayashi, Ryoma Sin'ya, and Takeshi Tsukada. On average-case hardness of higher-order model checking. **A full version.** Available from https://www.kb.is.s.u-tokyo.ac.jp/~koba/papers/OnAverageCaseHOMC.pdf.

21 C.-H. Luke Ong. On model-checking trees generated by higher-order recursion schemes. In *Proceedings of IEEE Symposium on Logic in Computer Science (LICS)*, pages 81–90. IEEE Computer Society Press, 2006. doi:10.1109/LICS.2006.38.

22 C.-H. Luke Ong and Steven Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 587–598. ACM Press, 2011. doi:10.1145/1925844.1926453.

23 Steven Ramsay, Robin Neatherway, and C.-H. Luke Ong. A type-directed abstraction refinement approach to higher-order model checking. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 61–72. ACM Press, 2014. doi:10.1145/2535838.2535873.

24 Sylvain Salvati and Igor Walukiewicz. Krivine machines and higher-order schemes. In *Proceedings of International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 6756 of *Lecture Notes in Computer Science*, pages 162–173. Springer, 2011. doi:10.1007/978-3-642-22012-8_12.

25 Sylvain Salvati and Igor Walukiewicz. Recursive schemes, krivine machines, and collapsible pushdown automata. In *Proceedings of International Workshop on Reachability Problems (RP)*, volume 7550 of *Lecture Notes in Computer Science*, pages 6–20. Springer, 2012. doi:10.1007/978-3-642-33512-9_2.

26 Ryoma Sin'ya, Kazuyuki Asada, Naoki Kobayashi, and Takeshi Tsukada. Almost every simply typed λ-term has a long β-reduction sequence. In *Proceedings of International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 10203 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2017. doi:10.1007/978-3-662-54458-7_4.

27 Richard Statman. On the lambda Y calculus. *APAL*, 130(1-3):325–337, 2004. doi:10.1016/j.apal.2004.04.004.

28 Ryota Suzuki, Koichi Fujima, Naoki Kobayashi, and Takeshi Tsukada. Streett automata model checking of higher-order recursion schemes. In *Proceedings of International Conference on Formal Structures for Computation and Deduction (FSCD)*, volume 84 of *LIPIcs*, pages 32:1–32:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.FSCD.2017.32.

A Supplementary Materials for Section 5

This section sketches a proof of Theorem 18 in Section 5, as well as that of Proposition 11. The full proofs can be found in the long version [20].

A.1 A Characterization of the Minimality

We give a technically convenient characterization of the minimality. Let t be a closed and ground-typed term and u be a subterm of t, i.e. $t = C[u]$ for some linear-context C. Recall that t is minimal if it has no useless subterm; so in particular u must be "used". Intuitively

u in $C[u]$ is used if $t = C[u] \longrightarrow^* E[u]$ for some evaluation context E,[3] but this criterion is incorrect. Consider, for example,

$$t \quad = \quad a\big((\lambda x.x)\, u,\ \bot\, \underline{u}\big) \quad \longrightarrow \quad a(u,\ \bot u),$$

where u appears under an evaluation context $E = a([\,], \bot\, u)$ after the reduction but the underlined occurrence of u is indeed useless. This example suggests that we need to distinguish different occurrences of the same term.

Let ℓ be a special tree constructor of arity 1 such that $\ell \notin \Sigma$. We call this symbol a *label* and use it to mark focused occurrences of (sub)terms. The *labelling* operation $(-)^\ell$ is defined, for a term t of type $\kappa_1 \to \ldots \to \kappa_k \to \mathsf{o}$, by $t^\ell := \lambda z_1.\ldots\lambda z_k.\ell(t\, z_1\, \ldots\, z_k)$. For a term t possibly having ℓ, we write $\natural(t)$ for the term obtained by removing ℓ.

A term is *labelled* if it contains ℓ; otherwise it is *unlabelled.* A labelled finite tree V is *well-labelled* if $V = D[\ell(u)]$ and $\natural(u) \neq \bot$ for some D and u.

▶ **Theorem 26** (Characterization of the minimality)**.** *Let t be a closed and ground-typed term over Σ. Then, t is minimal if and only if for every $\langle C, s\rangle$ such that $t = C[s]$ and $s \neq \bot$, there exists a finite well-labelled tree V such that $C[s^\ell] \longrightarrow^* \sqsupseteq V$.*

A.2 Proof Sketch of Proposition 11

Let t be a closed and ground-typed term. Assume that t is minimal, and let s be a non-$\bot$, closed and ground-typed subterm of t. Then $t = C[s]$ for some linear-context C. By Theorem 26, the value tree of $C[s^\ell]$ contains ℓ, say $T(C[s^\ell]) = D[\ell(V)]$. One can show that V is the value tree of s, which implies that $T(C[s^\ell])$ contains $T(s)$ as a subtree. Since $T(t) = T(C[s])$ is obtained by removing ℓ from $T(C[s^\ell])$, it also contains $T(s)$ as a subtree.

A.3 Proof Sketch of Theorem 18

Since we shall study possibly labelled terms, we first introduce a typing rule for $\ell(t)$:

$$\frac{\Theta \vdash t : \mathsf{o}}{\Theta \vdash \ell(t) : \mathsf{o}}(\ell) \quad .$$

Note that the rule for $\ell(t)$ differs from that for tree constructors: the argument of ℓ must be of type o, whereas those of a tree constructor can be of type $\bot$ in addition to o. The above rule ensures that $\emptyset \vdash \ell(t) : \mathsf{o}$ implies $\natural(t) \neq \bot$. So we have the following lemma.

▶ **Lemma 27.** *Let V be a labelled finite tree. If $\emptyset \vdash V : \mathsf{o}$, then V is well-labelled.*

We use Subject Reduction and Subject Expansion in the soundness and completeness proofs of our system, similar to proofs for other intersection type systems. However the standard version of Subject Reduction and Subject Expansion does not hold for our system since minimality is not preserved by reduction nor expansion. For example, consider

$$\Big(\lambda f.a\big(f\,(\lambda x\_.x),\ f\,(\lambda\_y.y)\big)\Big)\,(\lambda g.g\,b\,c) \quad \longrightarrow \quad a\big((\lambda g.g\,b\,c)\,(\lambda x\_.x),\ (\lambda g.g\,b\,c)\,(\lambda\_y.y)\big),$$

where a, b and c are tree constructors; the left-hand-side is minimal but the right-hand-side is not. In order to retain minimality, the right-hand-side has to be approximated:

$$a\big((\lambda g.g\,b\,c)\,(\lambda x\_.x),\ (\lambda g.g\,b\,c)\,(\lambda\_y.y)\big) \quad \sqsupseteq \quad a\big((\lambda g.g\,b\,\bot)\,(\lambda x\_.x),\ (\lambda g.g\,\bot\,c)\,(\lambda\_y.y)\big).$$

[3] Although evaluation contexts are not explicitly defined in Section 2, they are implicitly given in Definition 2 and their concrete definition should be clear.

The next lemma is a correct version, which takes account of the approximation relation.

▶ **Lemma 28** (Subject Reduction / Subject Expansion). *Assume that $s_1 \sqsubseteq t_1$ and $\Theta \vdash s_1 : \bar{\theta}$.*

(1) *If $t_1 \longrightarrow^* t_2$, then there exists $s_2 \sqsubseteq t_2$ with $\Theta \vdash s_2 : \bar{\theta}$ such that $s_1 \longrightarrow^* \sqsupseteq s_2$. Furthermore, if s_1 is labelled, we can choose s_2 so that it is labelled.*

(2) *If $t_0 \longrightarrow^* t_1$, then there exists $s_0 \sqsubseteq t_0$ with $\Theta \vdash s_0 : \bar{\theta}$ such that $s_0 \longrightarrow^* \sqsupseteq s_1$.*

The proof of completeness is rather straightforward. Note that, given a term t and a tree V with $t \longrightarrow^* \sqsupseteq V$, Subject Expansion induces a derivation of $\emptyset \vdash t' : \bar{\theta}$ for some $t' \sqsubseteq t$. The key to the completeness proof is to find sufficiently large V so that $t' = t$.

▶ **Theorem 29** (completeness). *Let t be any closed and ground-typed term over Σ. If t is minimal, then $\emptyset \vdash t : \bar{\theta}$ for some $\bar{\theta}$.*

Proof sketch. Since t is minimal, by Theorem 26, for each $\langle C, s\rangle$ such that $t = C[s]$ and $s \neq \bot$, one can find a finite well-labelled tree $V_C = D_C[\ell(u_C)]$ such that

$$C[s^{\ell}] \quad \longrightarrow^* \sqsupseteq \quad V_C \quad = \quad D_C[\ell(u_C)]. \tag{1}$$

We can assume without loss of generality that ℓ does not occur in D_C. Let $V = \bigsqcup_C D_C[\natural u_C]$, where C ranges over linear contexts such that $t = C[s]$ holds for some $s \neq \bot$. This is well-defined since $D_C[\natural u_C] \sqsubseteq T(t)$ for every C. Since V is an unlabelled tree, $\emptyset \vdash V : \bar{\theta}$ for some $\bar{\theta}$. Then by Subject Expansion (Lemma 28(2)), there exists $t' \sqsubseteq t$ such that $t' \longrightarrow^* \sqsupseteq V$ and $\emptyset \vdash t' : \bar{\theta}$.

It suffices to show that $t' = t$. Assume $t' \sqsubset t$ for contradiction. By the assumption, there exists $\langle C, s\rangle$ such that $t = C[s]$, $s \neq \bot$, and $t' \sqsubseteq C[\bot]$. Then

$$C[s^{\ell}] \quad \sqsupseteq \quad C[\bot] \quad \sqsupseteq \quad t' \quad \longrightarrow^* \sqsupseteq \quad V \quad \sqsupseteq \quad D_C[\natural u_C],$$

and thus $C[s^{\ell}] \longrightarrow^* \sqsupseteq D_C[\natural u_C]$. This together with (1) implies that $D_C[\ell(u_C)] \sqcup D_C[\natural u_C]$ is well-defined. This means $\ell(u_C) \sqcup \natural(u_C)$ is well-defined, which contradicts to the assumption that $\natural(u_C) \neq \bot$. ◀

The soundness proof requires another trick, since Subject Reduction alone does not ensure that a label eventually appears under an evaluation context in the presence of divergence. A term is $\mathbf{Y}$-*free* if it does not have $\mathbf{Y}$. The evaluation of a $\mathbf{Y}$-free term always terminates, and the soundness of the type system for $\mathbf{Y}$-free terms is relatively easy to prove. So we aim to remove $\mathbf{Y}$ in a given term before applying Subject Reduction, preserving its type.

Consider the rewriting rule $C[\mathbf{Y}\, t] \hookrightarrow C[t\,(\mathbf{Y}\, t)]$, which is allowed to be applied to any occurrence of $\mathbf{Y}\, t$ (not restricted to those under evaluation contexts). Then $\succeq_{\mathbf{Y}}$ is defined as $\hookrightarrow^* \sqsupseteq$. The next lemma is a key to the soundness proof, reflecting the inductive nature of the rules for $\mathbf{Y}$ in our type system.

▶ **Lemma 30.** *Assume that $\Theta \vdash t : \bar{\theta}$. Then there exists a $\mathbf{Y}$-free term s such that $t \succeq_{\mathbf{Y}} s$ and $\Theta \vdash s : \bar{\theta}$. Furthermore, if t is labelled, one can choose s so that s is also labelled.*

▶ **Theorem 31** (soundness). *Let t be any closed and ground-typed term over Σ. If $\emptyset \vdash t : \bar{\theta}$ for some $\bar{\theta}$, then t is minimal.*

Proof sketch. If $\bar{\theta} = \top$, then $t = \bot$ and thus t is minimal. Otherwise, we can assume without loss of generality that $\bar{\theta} = \mathtt{o}$. By Theorem 26, it suffices to show that, for every $\langle C, s\rangle$ with $t = C[s]$ and $s \neq \bot$, there exists a finite well-labelled tree V such that $C[s^{\ell}] \longrightarrow^* \sqsupseteq V$.

Assume that $t = C[s]$ and $s \neq \bot$. Since $\emptyset \vdash C[s] : \mathtt{o}$ and $s \neq \bot$, one can show that $\emptyset \vdash C[s^\ell] : \mathtt{o}$. By Lemma 30, there exists a $\mathbf{Y}$-free labelled term $\emptyset \vdash u : \mathtt{o}$ such that $C[s^\ell] \succeq_{\mathbf{Y}} u$. Since u is $\mathbf{Y}$-free, its evaluation terminates, i.e. $u \longrightarrow^* W$ for some tree W. By Subject Reduction (Lemma 28(1)), there exists a labelled term $V \sqsubseteq W$ such that $u \longrightarrow^* \sqsupseteq V$ and $\emptyset \vdash V : \mathtt{o}$.

It suffices to show that $C[s^\ell] \longrightarrow^* \sqsupseteq V$ and that V is a well-labelled tree. The former claim follows from $C[s^\ell] \succeq_{\mathbf{Y}} u \longrightarrow^* \sqsupseteq V$ because $\succeq_{\mathbf{Y}}$ can be seen as a kind of reduction. To prove the latter, observe that V is a tree since every approximation of a tree is a tree. So V as a well-typed and labelled tree is well-labelled by Lemma 27. ◀

B Proof of Proposition 22 and 23

▶ **Lemma 32.** *Suppose that C is a linear-context. If $C \lhd \tilde\theta' \Rrightarrow \tilde\tau$ and $C' \lhd \tilde\theta'' \Rrightarrow \tilde\theta'$, then $C[C'] \lhd \tilde\theta'' \Rrightarrow \{\tilde\tau\}$.*

Proof. Let $\tilde\theta' = \{\tilde\tau'_1, \dots, \tilde\tau'_n\}$. By $C' \lhd \tilde\theta'' \Rrightarrow \tilde\theta'$, there exists $\{\langle \tilde\theta''_{i,j}, C'_{i,j}\rangle\}_{i\in[n],j\in[k_i]}$ such that $C' = \bigsqcup_{i\in[n],j\in[k_i]} C'_{i,j}$, $\tilde\theta'' = \bigcup_{i\in[n],j\in[k_i]} \tilde\theta''_{i,j}$, and $C'_{i,j} \lhd \tilde\theta''_{i,j} \Rrightarrow \tilde\tau'_i$. Here, we can assume that $k_1 = \cdots = k_n$ (so, we denote them by k). Then from the derivation tree of $C \lhd \tilde\theta' \Rrightarrow \tilde\tau$ (see the left-hand side below), we can construct a derivation tree of $C[C'] \lhd \tilde\theta'' \Rrightarrow \tilde\tau$ (see the right-hand side below) by copying the form of the derivation tree of $C \lhd \tilde\theta' \Rrightarrow \tilde\tau$ as follows, where $\tilde\tau = \langle \Theta, \tau\rangle$ and $f\colon [m] \to [n']$ is a surjective map:

$$\dfrac{\mathtt{x} \lhd \tilde\tau'_{f(1)} \quad \cdots \quad \mathtt{x} \lhd \tilde\tau'_{f(m)}}{\begin{array}{c}\vdots\,\mathcal{T}\\ \Theta \vdash C[\mathtt{x}] : \tau\end{array}} \quad \rightsquigarrow_{\text{copy } \mathcal{T}} \quad \dfrac{\dfrac{C'_{f(1),1} \lhd \tilde\theta''_{f(1),1} \Rrightarrow \tilde\tau'_{f(1)} \quad \cdots \quad C'_{f(m),1} \lhd \tilde\theta''_{f(m),1} \Rrightarrow \tilde\tau'_{f(m)}}{\begin{array}{c}\vdots\,\mathcal{T}\\ \Theta \vdash C[\bigsqcup_{i\in[n]} C'_{i,1}] : \tau\end{array}} \quad \cdots \quad \dfrac{\cdots \quad \cdots \quad \cdots}{\begin{array}{c}\vdots\,\mathcal{T}\\ \Theta \vdash C[\bigsqcup_{i\in[n]} C'_{i,k}] : \tau\end{array}}}{\Theta \vdash C[C'] : \tau}\,(\wedge) .$$

◀

Proof of Proposition 22. Let $\tilde\theta' = \{\tilde\tau'_1, \dots, \tilde\tau'_{n'}\}$ and $\tilde\theta = \{\tilde\tau_1, \dots, \tilde\tau_n\}$. By $C \lhd \tilde\theta' \Rrightarrow \tilde\theta$, there exists $\{\langle \tilde\theta'_i, C_i\rangle\}_{i\in[m]}$ such that $C = \bigsqcup_{i\in[m]} C_i$, $\tilde\theta' = \bigcup_{i\in[m]} \tilde\theta'_i$, and $C_i \lhd \tilde\theta'_i \Rrightarrow \tilde\tau_{f(i)}$. By $C' \lhd \tilde\theta'' \Rrightarrow \tilde\theta'$, there exists $\{\langle \tilde\theta''_j, C''_j\rangle\}_{j\in[n']}$ such that $C' = \bigsqcup_{j\in[n']} C''_j$, $\tilde\theta'' = \bigcup_{j\in[n']} \tilde\theta''_j$, and $C''_j \lhd \tilde\theta''_j \Rrightarrow \{\tilde\tau_j\}$. Let $C'_i = \bigsqcup_{j\in[n'];\tilde\tau'_j\in\tilde\theta'_i} C''_j$ and let $\tilde\theta''_i = \bigcup_{j\in[n'];\tilde\tau'_j\in\tilde\theta'_i} \tilde\theta''_j$. Then $C'_i \lhd \tilde\theta''_i \Rrightarrow \tilde\theta'_i$. By Lemma 32, $C_i[C'_i] \lhd \tilde\theta''_i \Rrightarrow \tilde\tau_{f(i)}$. Therefore, $C[C'] \lhd \tilde\theta'' \Rrightarrow \tilde\theta$. ◀

▶ **Lemma 33.** *Suppose that C is a linear-context. If $C[C'] \lhd \tilde\theta'' \Rrightarrow \tilde\tau$, then $C \lhd \tilde\theta' \Rrightarrow \tilde\tau$ and $C' \lhd \tilde\theta'' \Rrightarrow \tilde\theta'$ for some $\tilde\theta'$.*

Proof. Then (the derivation tree of) $C[C'] \lhd \tilde\theta'' \Rrightarrow \tilde\tau$ should be of the form in the right-hand side below, where $\tilde\tau = \langle\Theta, \tau\rangle$, $C' = \bigsqcup_{i\in[m]} C'_i$, and $\tilde\theta'' = \bigcup_{i\in[m]} \tilde\theta''_i$. We let $\tilde\theta' = \{\tilde\tau'_1, \dots, \tilde\tau'_m\}$. Then, $C' \lhd \tilde\theta'' \Rrightarrow \tilde\theta'$ is immediate and $C \lhd \tilde\theta' \Rrightarrow \tilde\tau$ is shown by replacing each subterm arise from t to $\mathtt{x}$ (see the left-hand side below):

$$\dfrac{\mathtt{x} \lhd \tilde\tau'_1 \quad \cdots \quad \mathtt{x} \lhd \tilde\tau'_m}{\begin{array}{c}\vdots\,\mathcal{T}\\ \Theta \vdash C[\mathtt{x}] : \tau\end{array}} \quad \leftsquigarrow \quad \dfrac{C'_1 \lhd \tilde\theta''_1 \Rrightarrow \tilde\tau'_1 \quad \cdots \quad C'_m \lhd \tilde\theta''_m \Rrightarrow \tilde\tau'_m}{\begin{array}{c}\vdots\,\mathcal{T}\\ \Theta \vdash C[C'] : \tau\end{array}} .$$

◀

Proof of Proposition 23. Let $\tilde\theta'' = \{\tilde\tau''_1, \dots, \tilde\tau''_{n''}\}$ and $\tilde\theta = \{\tilde\tau_1, \dots, \tilde\tau_n\}$. By $C[C'] \lhd \tilde\theta'' \Rrightarrow \tilde\theta$, there exist a surjective map $f\colon [m] \to [n]$ and a sequence $\{\langle C_i, C'_i, \tilde\theta''_i\rangle\}_{i\in[m]}$ such that $C_i[C'_i] \lhd \tilde\theta''_i \Rrightarrow \tilde\tau_{f(i)}$, $C = \bigsqcup_{i\in[m]} C_i$, $C' = \bigsqcup_{i\in[m]} C'_i$, and $\tilde\theta'' = \bigcup_{i\in[m]} \tilde\theta''_i$ (see also the full version [20]). By Lemma 33, $C_i \lhd \tilde\theta'_i \Rrightarrow \tilde\tau_{f(i)}$ and $C'_i \lhd \tilde\theta''_i \Rrightarrow \tilde\theta'_i$ for some $\tilde\theta'_i$. We now let $\tilde\theta' = \bigcup_{j\in[m]} \tilde\theta'_i$. Then, both $C' \lhd \tilde\theta'' \Rrightarrow \tilde\theta'$ and $C \lhd \tilde\theta' \Rrightarrow \tilde\theta$ are immediate. ◀

C Proof of Lemma 24

The *size* of a simple type κ and a simple type environment Γ, written $|\kappa|$ and $|\Gamma|$ respectively, is defined by: $|\kappa| \triangleq 1$ if $\kappa = \mathtt{o}$, $|\kappa| \triangleq 1+|\kappa_1|+|\kappa_2|$ if $\kappa = \kappa_1 \to \kappa_2$, and $|\Gamma| \triangleq 1+\sum_{x\in\mathrm{dom}(\Gamma)} |\Gamma(x)|$.

▶ **Definition 34.** *The term $t_{\Gamma,\kappa}$ is inductively defined as follows, where in the second case, $l = \min\{i \in [\xi] \mid z_i \in \mathrm{dom}(\Gamma)\}$; and in the third case, $l = \min\{i \in [\xi] \mid z_i \notin \mathrm{dom}(\Gamma)\}$:*

$$t_{\Gamma,\kappa} \triangleq \begin{cases} \mathtt{a} & (\kappa = \mathtt{o} \textit{ and } \Gamma = \emptyset) \\ \mathtt{b}(z_l t_{\emptyset,\kappa^1} \dots t_{\emptyset,\kappa^m}, t_{\Gamma',\mathtt{o}}) & (\kappa = \mathtt{o} \textit{ and } \Gamma = (\Gamma', z_l : \kappa^1 \to \dots \to \kappa^m \to \mathtt{o})) \\ \lambda z_l.t_{(\Gamma,z_l:\kappa'),\kappa''} & (\kappa = \kappa' \to \kappa'' \textit{ and } \#(\mathrm{dom}(\Gamma)) < \xi) \\ (\lambda z_1.t_{(z_1:\mathtt{o}),\kappa})\ t_{\Gamma,\mathtt{o}} & (\kappa = \kappa' \to \kappa'' \textit{ and } \mathtt{ar}\,(\kappa) < \iota) \\ \textit{undefined} & (\textit{otherwise}) \end{cases}.$$

▶ **Proposition 35.** *Suppose that $\langle\Gamma,\kappa\rangle$ is ($\langle k,\iota,\xi\rangle$-)bounded. If $\#(\mathrm{dom}(\Gamma)) < \xi$ or $\mathtt{ar}\,(\kappa) < \iota$, then* (1) *$t_{\Gamma,\kappa}$ is defined,* (2) *$\Gamma \vdash_{\mathrm{ST}} t_{\Gamma,\kappa} : \kappa$, and* (3) *$t_{\Gamma,\kappa}$ is bounded.*

Proof. By a straightforward induction on the parameter $\langle|\kappa|,|\Gamma|\rangle$. ◀

We now extend the above for intersection types.

▶ **Definition 36.** *The term $t_{\Theta,\bar\theta}$ is inductively defined as follows, where in the second case, $l = \min\{i \in [\xi] \mid z_i \in \mathrm{dom}(\Theta)\}$; and in the third case, $l = \min\{i \in [\xi] \mid z_i \notin \mathrm{dom}(\Theta)\}$:*

$$t_{\Theta,\bar\theta} \triangleq \begin{cases} \mathtt{a} & (\bar\theta = \mathtt{o} \textit{ and } \Theta = \emptyset) \\ \mathtt{b}(\bigsqcup_{i\in[n]} z_l t_{\emptyset,\theta_i^1} \dots t_{\emptyset,\theta_i^m}, t_{\Theta',\mathtt{o}}) & (\bar\theta = \mathtt{o} \textit{ and } \Theta = (\Theta', z_l : \bigwedge_{i\in[n]} \theta_i^1 \to \dots \to \theta_i^m \to \mathtt{o})) \\ \lambda z_l.t_{(\Theta,z_l:\theta'),\tau''} & (\bar\theta = \theta' \to \tau'' \textit{ and } \#(\mathrm{dom}(\Theta)) < \xi) \\ (\lambda z_1.t_{(z_1:\wedge\{\mathtt{o}\}),\bar\theta})\ t_{\Theta,\mathtt{o}} & (\bar\theta = \theta' \to \tau'' \textit{ and } \mathtt{ar}\,(\kappa) < \iota) \\ \bigsqcup_{i\in[n]} t_{\Theta,\tau_i} & (\bar\theta = \bigwedge_{i\in[n]} \tau_i \textit{ and } n \geq 1) \\ \bot^\kappa & (\bar\theta = \top^\kappa \textit{ and } \Theta = \emptyset) \\ \textit{undefined} & (\textit{otherwise}) \end{cases}.$$

▶ **Proposition 37.** *Suppose that $\langle\Theta,\bar\theta\rangle :: \langle\Gamma,\kappa\rangle$ for some bounded $\langle\Gamma,\kappa\rangle$. If $\#(\mathrm{dom}(\Gamma)) < \xi$, $\mathtt{ar}\,(\kappa) < \iota$, or $\langle\Theta,\bar\theta\rangle = \langle\emptyset,\top\rangle$, then* (1) *$t_{\Theta,\bar\theta}$ is defined,* (2) *$t_{\Theta,\bar\theta} \sqsubseteq t_{\Gamma,\kappa}$,* (3) *$\Theta \vdash t_{\Theta,\bar\theta} : \bar\theta$, and* (4) *$t_{\Theta,\bar\theta}$ is bounded.*

Proof. By a straightforward induction on the parameter $\langle|\kappa|,|\Gamma|\rangle$. The existence of the join in each case can be ensured by the assumption (2). ◀

We now extend the above for context-types to prove Lemma 24.

▶ **Definition 38.** *The linear-context $C_{\tilde\tau}$ is inductively defined as follows, where in the second case, $l = \min\{i \in [\xi] \mid z_i \notin \mathrm{dom}(\Theta)\}$:*

$$C_{\langle\Theta,\tau\rangle} \triangleq \begin{cases} \mathtt{b}(t_{\Theta,\mathtt{o}}, [\,]) & (\tau = \mathtt{o}) \\ \lambda z_l.C_{\langle(\Theta,z_l:\theta'),\tau''\rangle} & (\tau = \theta' \to \tau'' \textit{ and } \#(\mathrm{dom}(\Theta)) < \xi) \\ (\lambda z_1.t_{(z_1:\wedge\{\mathtt{o}\}),\tau})\ C_{\langle\Theta,\mathtt{o}\rangle} & (\tau = \theta' \to \tau'' \textit{ and } \mathtt{ar}\,(\tau) < \iota) \\ \textit{undefined} & (\textit{otherwise}) \end{cases}.$$

For each $\tilde\theta^+ = \{\tilde\tau_1,\dots,\tilde\tau_n\}$, let $C_{\tilde\theta^+} \triangleq \bigsqcup_{i\in[n]} C_{\tilde\tau_i}$. This is well-defined thanks to Proposition 37(2).

▶ **Proposition 39.** *Suppose that $\tilde\theta^+ :: \langle\Gamma,\kappa\rangle$ for some bounded $\langle\Gamma,\kappa\rangle$. If $\#(\mathrm{dom}(\Gamma)) < \xi$ or $\mathtt{ar}\,(\kappa) < \iota$, then* (1) *$C_{\tilde\theta^+}$ is defined,* (2) *$C_{\tilde\theta^+} \lhd \{\langle\emptyset,\mathtt{o}\rangle\} \Rrightarrow \tilde\theta$, and* (3) *$C_{\tilde\theta^+}$ is bounded.*

Proof. By a straightforward induction on the parameter $\langle |\kappa|, |\Gamma| \rangle$. ◀

▶ **Definition 40.** *The linear-context $D_{\tilde{\tau}}$ is defined as follows, where in the first case, $l = \min\{i \in [\xi] \mid z_i \in \mathrm{dom}(\Theta)\}$; and in the second case, $\tau = \theta^1 \to \ldots \to \theta^m \to \mathtt{o}$:* $D_{\langle \Theta, \tau \rangle} \triangleq \begin{cases} (\lambda z_l . D_{\langle \Theta', \tau \rangle})\, t_{\emptyset, \theta_l} & (\Theta = (\Theta', z_l : \theta_l)) \\ \mathtt{c}([\,]\, t_{\emptyset, \theta^1} \,\ldots\, t_{\emptyset, \theta^m}) & (\Theta = \emptyset) \end{cases}$. *Let $D_{\tilde{\theta}^+} \triangleq \bigsqcup_{i \in [n]} D_{\tilde{\tau}_i}$ for each $\tilde{\theta}^+ = \{\tilde{\tau}_1, \ldots, \tilde{\tau}_n\}$. This is well-defined thanks to Proposition 37(2). Also, specially, let $D_\emptyset \triangleq \mathtt{a}$.*

▶ **Proposition 41.** *Suppose that $\tilde{\theta} :: \langle \Gamma, \kappa \rangle$ for some bounded $\langle \Gamma, \kappa \rangle$. Then,* (1) *$D_{\tilde{\theta}}$ is defined,* (2) *$D_{\tilde{\theta}} \lhd \tilde{\theta} \Rrightarrow \{\langle \emptyset, \mathtt{o} \rangle\}$, and* (3) *$D_{\tilde{\theta}}$ is bounded.*

Proof. By a straightforward induction on the parameter $\langle |\kappa|, |\Gamma| \rangle$. ◀

As a consequence of Proposition 39 and 41, Lemma 24 has been proved.

C.1 On the Boundary Case of Lemma 24(1)

Here, we consider the boundary case of Lemma 24(1), i.e., $\Gamma \vdash_{\mathrm{ST}} t : \kappa$, t is $\langle k, \iota, \xi \rangle$-bounded, $\#(\mathrm{dom}(\Gamma)) = \xi$, and $\mathtt{ar}\,(\kappa) = \iota$. Actually in this case, t should be of a special form.

▶ **Lemma 42.** *Suppose that*

(1) $\Gamma \vdash_{\mathrm{ST}} t : \kappa$,

(2) *t is $\langle k, \iota, \xi \rangle$-bounded,*

(3) *$\#(\mathrm{dom}(\Gamma)) = \xi$, and*

(4) *$\mathtt{ar}\,(\kappa) = \iota$.*

Then, t is α-equivalent to a term of the form $\lambda\_.t_1$.

Proof. By $\xi > 1$, $t \neq x$ and $t \neq \bot$. By $\iota > 0$, $t \neq a(t_1, \ldots, t_{\Sigma(a)})$. By $\mathtt{ar}\,(\kappa) = \iota$, $t \neq t_1 t_2$ and $t \neq \mathbf{Y} t_1$. Therefore t is of the form $\lambda \bar{x}.t_1$. By that t is bounded and $\#(\mathrm{dom}(\Gamma)) = \xi$, the last rule of $\Gamma \vdash_{\mathrm{ST}} \lambda \bar{x}.t_1 : \kappa$ should be (Abs2), so $\Gamma \vdash_{\mathrm{ST}} t_1 : \kappa''$, where $\kappa = \kappa' \to \kappa''$. Then $\bar{x}$ does not occur in t_1 as a free variable. Therefore t is α-equivalent to the term $\lambda\_.t_1$. ◀

Size-Preserving Translations from Order-$(n+1)$ Word Grammars to Order-n Tree Grammars

Kazuyuki Asada
Tohoku University, Sendai, Japan
http://www.riec.tohoku.ac.jp/~asada/
asada@riec.tohoku.ac.jp

Naoki Kobayashi
The University of Tokyo, Japan
http://www-kb.is.s.u-tokyo.ac.jp/~koba/
koba@kb.is.s.u-tokyo.ac.jp

Abstract

Higher-order grammars have recently been studied actively in the context of automated verification of higher-order programs. Asada and Kobayashi have previously shown that, for any order-$(n+1)$ word grammar, there exists an order-n grammar whose frontier language coincides with the language generated by the word grammar. Their translation, however, blows up the size of the grammar, which inhibited complexity-preserving reductions from decision problems on word grammars to those on tree grammars. In this paper, we present a new translation from order-$(n+1)$ word grammars to order-n tree grammars that is size-preserving in the sense that the size of the output tree grammar is polynomial in the size of an input tree grammar. The new translation and its correctness proof are arguably much simpler than the previous translation and proof.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory

Keywords and phrases higher-order grammar, word language, tree language, complexity

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.22

Related Version A full version of the paper is available at http://www.riec.tohoku.ac.jp/~asada/papers/spWtoT.pdf.

Funding This work was supported by JSPS Kakenhi 15H05706, 18K11156, and 20H00577.

Acknowledgements We would like to thank anonymous referees for useful comments.

1 Introduction

It is well known that there is a close relationship between context-free word languages and regular tree languages: for any regular tree language L, its frontier language, i.e., the word language consisting of the sequence of leaf symbols of each tree in L, is context-free; conversely, for any context-free language with no empty sequence, the set of parse trees is a regular tree language. Damm [6] has shown that this correspondence generalizes to *safe* higher-order languages: for any order-n (safe) language L, there is a corresponding order-$(n+1)$ word language that corresponds to the frontier language of L, and vice versa. Asada and Kobayashi [1] extended the result to *unsafe* higher-order languages. The results allow us to reduce a problem on order-$(n+1)$ word languages (like linear-time property verification of order-$(n+1)$ functional programs) to a problem on order-n (but tree) languages. Such a reduction may be useful, since the cost of various problems on a higher-order language (such as HORS model checking) is usually in the tower of exponentials whose height is the order of the language.

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 22; pp. 22:1–22:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Unfortunately, however, the translation of Asada and Kobayashi [1] from order-$(n+1)$ word grammars to order-n tree grammars (which is the hard direction) blows up the size of the grammar: the output tree grammar is hyper-exponential in the size of an input word grammar, which cancels off the benefit of reducing the order of the grammar. Also, both the translation and correctness proof were very complex. Clemente et al. [5] have shown another translation from order-$(n+1)$ word grammars to order-n tree grammars (as a component of their algorithm for the diagonal problem), but their translation also suffers from the same problem of blowing up the size of the grammar.

In the present paper, we present a much simpler translation from order-$(n+1)$ word grammars to order-n tree grammars, where the size of the output grammar is *polynomial* in the size of an input grammar.[1]

Some of the known results on the complexity of decision problems on higher-order languages follow immediately from our result. For example, Parys [12] has shown that the complexity of the diagonal problem is n-EXPTIME-complete for order-n tree grammars and $(n-1)$-EXPTIME-complete for order-n word grammars. He separately discussed the tree and word cases, but using our result, the upper-bound for the word-case follows immediately from the tree case. (Note that the hardness follows from the translation of [1] in the opposite direction.) For another example, it is known that the inclusion problem between an order-$(n+1)$ word language and a regular language (to which various program verification problems can be reduced) can be decided in n-EXPTIME in the size of the order-$(n+1)$ word grammar; it is, for example, obtained as a corollary of Kobayashi and Ong's result [10] that linear-time property model checking of order-$(n+1)$ higher-order recursion schemes (HORS) is n-EXPTIME complete. Using our result, the upper-bound of the complexity of the inclusion problem can be obtained as an immediate corollary of the n-EXPTIME completeness of the modal μ-calculus model checking of HORS [11] and our result.

Our new translation from word grammars to tree grammars is in a sense more elementary than the previous known translations [1, 5]. While the previous translations used intersection types, our new translation, which has been inspired from [3, 9], uses only simple types. The correctness proof of the new translation is also arguably much simpler than those of the previous translations.

Related Work. As explained above, a translation from order-$(n+1)$ word grammars to order-n tree grammars has first been shown by Damm [6, Theorem 7.17] for safe grammars. His translation does not seem to generalize to unsafe grammars (grammars without the so called "safety restriction" [7, 4]). Asada and Kobayashi [1] and Clemente et al. [5] have shown translations for unsafe grammars. As mentioned already, these previous translations are quite different from ours. Both of the previous translations [1, 5] used intersection types and replicated a term for each intersection type. Since the number of intersection types need for the translation of order-k grammars is k-fold exponential, it was inevitable for the previous translations to blow up the size of the grammar. In contrast, our translation does not use intersection types, and only keeps track of how order-0 variables are used.

As for applications, Asada and Kobayashi's translation [1] has been used to prove a pumping lemma for higher-order grammars *modulo* a certain conjecture [2]. Clemente et al. [5] used (an extension of) their translation to prove the decidability of the diagonal problem.

[1] More precisely, the transformation of Asada and Kobayashi [1] consists of two steps, a main step and an administrative step. We simplify only the main step; for applications discussed below, we do not need the administrative step.

Our translation has been inspired from related transformations developed in [3, 9]. Asada and Kobayashi [3] used a related transformation for order-3 grammars, which was used to prove a variant of the pumping lemma for higher-order grammars. Kobayashi et al. [9] used a similar transformation technique for probabilistic higher-order grammars called PHORS, to provide a fixpoint characterization of the termination probability of PHORS.

The rest of the paper is structured as follows. Section 2 reviews the definition of higher-order grammars, and states the main result. Section 3 gives the definition of the new transformation from order-$(n+1)$ word grammars to order-n tree grammars. Section 4 proves the complexity results, and discusses an application of the main result. Section 5 proves the correctness of the translation. Section 6 concludes the paper.

2 Preliminaries

We first review basic definitions on higher-order grammars [6, 1] in Section 2.1; our definitions given below basically follow those of [1], with slight modifications. We then state our main result in Section 2.2.

We often write $\widetilde{a}$ for a sequence $a_1, \dots, a_n$, and write $|\tilde{a}|$ for the length n of the sequence.

2.1 Higher-order grammars

▶ **Definition 1** (types and terms). *The set of* (simple) types*, ranged over by* κ*, is given by:* $\kappa ::= \mathsf{o} \mid \kappa_1 \to \kappa_2$*. The* order*,* arity*, and* size *of a type* κ*, written* $\mathtt{ord}(\kappa)$*,* $\mathtt{ari}(\kappa)$*, and* $|\kappa|$*, are defined by*

$$\begin{aligned} \mathtt{ord}(\mathsf{o}) &:= 0 & \mathtt{ord}(\kappa_1 \to \kappa_2) &:= \max\big(\mathtt{ord}(\kappa_1)+1, \mathtt{ord}(\kappa_2)\big) \\ \mathtt{ari}(\mathsf{o}) &:= 0 & \mathtt{ari}(\kappa_1 \to \kappa_2) &:= 1+\mathtt{ari}(\kappa_2) \\ |\mathsf{o}| &:= 1 & |\kappa_1 \to \kappa_2| &:= |\kappa_1|+|\kappa_2|+1. \end{aligned}$$

The type o *describes trees, and* $\kappa_1 \to \kappa_2$ *describes functions from* κ_1 *to* κ_2*. The set of* terms*, ranged over by* $s, t, u, \dots$*, is defined by:*

$$t ::= x \mid a\, t_1 \,\cdots\, t_k \mid t_1\, t_2 \mid t_1 \oplus t_2 \mid \Omega.$$

Here, x *ranges over a denumerable set of variables, and* a *over a set of constants (which represent tree constructors). We also use meta-variables* y, z, F, G *for variables. Variables and constants are also called* non-terminals *and* terminals *respectively. A* ranked alphabet Σ *is a map from a finite set of terminals to the set of natural numbers; we call* $\Sigma(a)$ *the* arity *of a terminal* a*. We implicitly assume a ranked alphabet whose domain contains all terminals discussed, unless explicitly described. The term* $t_1 \oplus t_2$ *denotes a non-deterministic choice between* t_1 *and* t_2*, and* Ω *denotes divergence. A term is called an* applicative term *if it contains neither* $\oplus$ *nor* Ω*.*

A (simple) type environment $\mathcal{K}$ *is a sequence of type bindings of the form* $x : \kappa$*, where* $\mathcal{K}$ *may contain at most one binding for each variable. The type judgment relation* $\mathcal{K} \vdash_{\mathtt{ST}} t : \kappa$ *is defined by the following rules.*

$$\frac{}{\mathcal{K},\, x : \kappa,\, \mathcal{K}' \vdash_{\mathtt{ST}} x : \kappa} \qquad \frac{\Sigma(a) = k \qquad \mathcal{K} \vdash_{\mathtt{ST}} t_i : \mathsf{o} \quad (\textit{for each } i \in \{1, \dots, k\})}{\mathcal{K} \vdash_{\mathtt{ST}} a\, t_1 \cdots t_k : \mathsf{o}}$$

$$\frac{\mathcal{K} \vdash_{\mathtt{ST}} t_1 : \kappa_2 \to \kappa \qquad \mathcal{K} \vdash_{\mathtt{ST}} t_2 : \kappa_2}{\mathcal{K} \vdash_{\mathtt{ST}} t_1\, t_2 : \kappa} \qquad \frac{\mathcal{K} \vdash_{\mathtt{ST}} t_1 : \mathsf{o} \qquad \mathcal{K} \vdash_{\mathtt{ST}} t_2 : \mathsf{o}}{\mathcal{K} \vdash_{\mathtt{ST}} t_1 \oplus t_2 : \mathsf{o}} \qquad \frac{}{\mathcal{K} \vdash_{\mathtt{ST}} \Omega : \mathsf{o}}$$

For a technical convenience, in the typing rule for constants, we require that a terminal must be fully applied; this does not restrict the expressive power of higher-order grammars introduced below. Note that, given $\mathcal{K}$ and t, there exists at most one type κ such that $\mathcal{K} \vdash_{\mathsf{ST}} t : \kappa$. We call κ the type of t (with respect to $\mathcal{K}$). Henceforth, we consider only well-typed terms.

A term t is called ground *(with respect to $\mathcal{K}$) if $\mathcal{K} \vdash_{\mathsf{ST}} t : \mathsf{o}$, and t is called a (finite, Σ-ranked)* tree *if t is a closed ground applicative term consisting of only terminals. We write* $\mathbf{Tree}_\Sigma$ *for the set of Σ-ranked trees, and use the meta-variable v for trees.*

We define the size $|t|$ *of a term t by: $|x| := 1$, $|a\,t_1 \cdots, t_k| := 1 + |t_1| + \cdots + |t_k|$, $|s\,t| := |s| + |t| + 1$, $|s \oplus t| := |s| + |t| + 1$, and $|\Omega| := 1$.*

▶ **Definition 2** (higher-order grammar)**.** *A* higher-order grammar *(or* grammar *for short) is a quadruple $(\Sigma, \mathcal{N}, \mathcal{R}, t^\circ)$, where*

- *Σ is a ranked alphabet,*
- *$\mathcal{N}$ is a map from a finite set of non-terminals to their types,*
- *$\mathcal{R}$ is a finite set of* rewriting rules *of the form $F\,x_1 \cdots x_\ell \to t$, where: (i) t is a term, (ii) $\mathcal{N}(F) = \kappa_1 \to \cdots \to \kappa_\ell \to \mathsf{o}$, (iii) $\mathcal{N}, x_1 : \kappa_1, \ldots, x_\ell : \kappa_\ell \vdash_{\mathsf{ST}} t : \mathsf{o}$ holds, and (iv) in $\mathcal{R}$ there is exactly one rule for each nonterminal,*
- *t° is a term called the* start term*, and $\mathcal{N} \vdash_{\mathsf{ST}} t^\circ : \mathsf{o}$.*

The order *of a grammar $\mathcal{G}$ is defined as the largest order of the types of non-terminals (or 0 if dom($\mathcal{N}$) is empty). We define the* size *of $\mathcal{N}$, written as $|\mathcal{N}|$, by $|\mathcal{N}| := \sum_{F \in \mathcal{N}} |\mathcal{N}(F)|$, and also define the* Curry-style *and* Church-style sizes *of a grammar $\mathcal{G}$, written as $|\mathcal{G}|_{\mathrm{cy}}$ and $|\mathcal{G}|_{\mathrm{ch}}$, by: $|\mathcal{G}|_{\mathrm{cy}} := |t^\circ| + \sum_{(F\,x_1 \cdots x_\ell \to t) \in \mathcal{R}} (|t| + \ell)$ and $|\mathcal{G}|_{\mathrm{ch}} := |\mathcal{G}|_{\mathrm{cy}} + |\mathcal{N}|$, respectively. We sometimes omit the subscript of $|\mathcal{G}|_{\mathrm{ch}}$ and write $|\mathcal{G}|$. $|\mathcal{G}|_{\mathrm{ch}}$ (rather than $|\mathcal{G}|_{\mathrm{cy}}$) is essentially the same as the definition of the grammar size in [12] (the results of [12] will be discussed in Section 4.2, as an application of our results).*

The tree language $\mathcal{L}(\mathcal{G})$ generated by *a grammar $\mathcal{G}$ is defined as follows. The set of* evaluation contexts *(ranged over by E) is defined by the grammar:*

$$E ::= [\,] \mid a\,t_1 \ldots t_{i-1}\,E\,t_{i+1} \ldots t_{\Sigma(a)} \ (1 \le i \le \Sigma(a)) \mid E \oplus t \mid t \oplus E.$$

Below we consider only contexts E such that $\mathcal{N}, \mathcal{K}, [\,] : \mathsf{o} \vdash_{\mathsf{ST}} E : \mathsf{o}$. For a grammar $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, t^\circ)$, the rewriting relation $\longrightarrow_{\mathcal{G}}$ is defined as follows:

$$\frac{(F\,\widetilde{x} \to t) \in \mathcal{R}}{E[\,F\,\widetilde{s}\,] \longrightarrow_{\mathcal{G}} E[\,[\widetilde{s}/\widetilde{x}]t\,]} \qquad \frac{}{E[\,a\,\widetilde{s}\,(t_1 \oplus t_2)\,\widetilde{u}\,] \longrightarrow_{\mathcal{G}} E[\,(a\,\widetilde{s}\,t_1\,\widetilde{u}) \oplus (a\,\widetilde{s}\,t_2\,\widetilde{u})\,]}$$

We write $\longrightarrow_{\mathcal{G}}^*$ for the reflexive transitive closure of $\longrightarrow_{\mathcal{G}}$. We may omit the subscript $\mathcal{G}$ and write $\longrightarrow$ and $\longrightarrow^*$, if $\mathcal{G}$ is clear from the context. We define the set of choice contexts *(ranged over by C) by: $C ::= [\,] \mid C \oplus t \mid t \oplus C$. For $\mathcal{N} \vdash_{\mathsf{ST}} t : \mathsf{o}$, we define $\mathcal{L}(\mathcal{G}, t) := \{v \in \mathbf{Tree}_\Sigma \mid t \longrightarrow_{\mathcal{G}}^* C[v]\}$, and $\mathcal{L}(\mathcal{G}) := \mathcal{L}(\mathcal{G}, t^\circ)$.*

▶ Remark 3. In [1], we used a slightly different definition of a higher-order grammar: (i) The definition in the present paper uses a start term, while that in [1] uses a *start symbol* (i.e., a start term is restricted to some non-terminal S). (ii) In the present paper $\mathcal{R}$ is deterministic and total, and $\oplus$ and Ω may occur on the right hand side of each rule, while in [1] $\mathcal{R}$ is not necessarily deterministic nor total, and neither $\oplus$ nor Ω may occur. The two styles of grammars can be mutually translated in an obvious manner.

The grammars defined above may also be viewed as generators of word languages.

▶ **Definition 4** (word alphabet / $\mathtt{br}$-alphabet)**.** *A ranked alphabet Σ is called a* word alphabet *if it has a special nullary terminal* $\mathtt{e}$ *and all the other terminals have arity* 1*. A grammar $\mathcal{G}$ is called a* word grammar *if its alphabet is a word alphabet. For a tree $v = a_1(\cdots(a_n\,\mathtt{e})\cdots)$ of a word grammar, we define* $\mathbf{word}(v) = a_1\cdots a_n$*. The* word language *generated by a word grammar $\mathcal{G}$, written $\mathcal{L}_{\mathtt{w}}(\mathcal{G})$, is* $\mathbf{word}(\mathcal{L}(\mathcal{G}))$*.*

The frontier word *of a tree v, written* $\boldsymbol{leaves}(v)$*, is the sequence of symbols in the leaves of v. It is defined inductively by:* $\boldsymbol{leaves}(a) = a$ *when* $\Sigma(a) = 0$*, and* $\boldsymbol{leaves}(a\,v_1\,\cdots\,v_k) = \boldsymbol{leaves}(v_1)\cdots\boldsymbol{leaves}(v_k)$ *when* $\Sigma(a) = k > 0$*. We write $\mathcal{L}_{\mathtt{leaf}}(\mathcal{G})$ and $\mathcal{L}_{\mathtt{leaf}}(\mathcal{G},t)$ for* $\boldsymbol{leaves}(\mathcal{L}(\mathcal{G}))$ *and* $\boldsymbol{leaves}(\mathcal{L}(\mathcal{G},t))$*, respectively, and call $\mathcal{L}_{\mathtt{leaf}}(\mathcal{G})$ the* frontier language *generated by $\mathcal{G}$.*

A $\mathtt{br}$-alphabet *is a ranked alphabet such that it has a special binary constant* $\mathtt{br}$ *and a special nullary constant* $\mathtt{e}$*, and the other constants are nullary. We call a grammar $\mathcal{G}$ a* $\mathtt{br}$-grammar *if its alphabet is a* $\mathtt{br}$*-alphabet. Intuitively,* $\mathtt{br}$ *and* $\mathtt{e}$ *represent the concatenation of two frontier words and the empty word ε respectively. For a word w, we write $w{\uparrow}_{\mathtt{e}}$ for the word obtained by removing all the occurrences of* $\mathtt{e}$ *in w, and $\mathcal{L}{\uparrow}_{\mathtt{e}}$ for $\{w{\uparrow}_{\mathtt{e}} \mid w \in \mathcal{L}\}$. We write $s \approx t$ if $\mathcal{L}_{\mathtt{leaf}}(\mathcal{G},s){\uparrow}_{\mathtt{e}} = \mathcal{L}_{\mathtt{leaf}}(\mathcal{G},t){\uparrow}_{\mathtt{e}}$.*

For a word alphabet Σ, we define the $\mathtt{br}$-alphabet of Σ*, written $\mathtt{br}(\Sigma)$, by:* $\mathtt{br}(\Sigma) := \{\mathtt{e} \mapsto 0, \mathtt{br} \mapsto 2\} \cup \{a \mapsto 0 \mid \Sigma(a) = 1\}$*.*

We note that the classes of order-0, order-1, and order-2 word languages coincide with those of regular, context-free, and indexed languages, respectively [13].

▶ **Example 5.** Consider the order-2 (word) grammar $\mathcal{G}_1 = (\{\mathtt{a}:1, \mathtt{e}:0\}, \{S:\mathtt{o}, F:(\mathtt{o}\to\mathtt{o})\to(\mathtt{o}\to\mathtt{o}), T:(\mathtt{o}\to\mathtt{o})\to(\mathtt{o}\to\mathtt{o}), A:\mathtt{o}\to\mathtt{o}\}, \mathcal{R}_1, S)$, where $\mathcal{R}_1$ consists of:

$$S \to F\,A\,\mathtt{e} \qquad F\,f\,x \to (f\,x) \oplus (F\,(T\,f)\,x) \qquad T\,f\,x \to f(f\,x) \qquad A\,x \to \mathtt{a}\,x$$

S is reduced, for example, as follows

$$\begin{aligned} S \longrightarrow F\,A\,\mathtt{e} \longrightarrow C_1[F\,(T\,A)\,\mathtt{e}] \longrightarrow C_2[F\,(T\,(T\,A))\,\mathtt{e}] &\longrightarrow C_3[(T\,(T\,A))\,\mathtt{e}] \\ &\longrightarrow C_3[T\,A\,(T\,A\,\mathtt{e})] \longrightarrow^* C_3[\mathtt{a}^4(\mathtt{e})] \end{aligned}$$

where C_1, C_2, and C_3 are some appropriate choice contexts. The word language $\mathcal{L}_{\mathtt{w}}(\mathcal{G}_1)$ is $\{\mathtt{a}^{2^n} \mid n \geq 0\}$.

Consider the order-1 (tree) grammar $\mathcal{G}_2 = (\{\mathtt{br}:2, \mathtt{a}:0, \mathtt{e}:0\}, \{S:\mathtt{o}, F:\mathtt{o}\to\mathtt{o}, T:\mathtt{o}\to\mathtt{o}\}, \mathcal{R}_2, S)$, where $\mathcal{R}_2$ consists of:

$$S \to F\,\mathtt{a} \qquad F\,f \to f \oplus (F(T\,f)) \qquad T\,f \to \mathtt{br}\,f\,f.$$

The frontier language $\mathcal{L}_{\mathtt{leaf}}(\mathcal{G}_2)$ coincides with $\mathcal{L}_{\mathtt{w}}(\mathcal{G}_1)$ above. This (existence of an order-1 tree grammar corresponding to an order-2 word grammar) is not a coincidence, as stated in Theorem 6 below.

2.2 The main result

The following theorem states the main result of the present paper.

▶ **Theorem 6.** *For any $n \geq 0$, there exist an effective translation $\mathcal{T}_n$ from order-$(n+1)$ word grammars to order-n* $\mathtt{br}$*-grammars and a polynomial p_n such that, for any order-$(n+1)$ word grammar $\mathcal{G}$, $\mathcal{L}_{\mathtt{leaf}}(\mathcal{T}_n(\mathcal{G})){\uparrow}_{\mathtt{e}} = \mathcal{L}_{\mathtt{w}}(\mathcal{G})$ and $|\mathcal{T}_n(\mathcal{G})| \leq p_n(|\mathcal{G}|)$.*

The theorem above follows from Theorems 12 and 13, given in Sections 4 and 5 respectively. Theorem 6 without the condition $|\mathcal{T}_n(\mathcal{G})| \leq p_n(|\mathcal{G}|)$ has been proved in [1] (Theorem 7).

▶ Remark 7. Asada and Kobayashi [1] have also presented a post-processing transformation for removing $\mathtt{e}$, which also suffers from a hyper-exponential blow-up of the grammar size. We do not think that there exists a size-preserving transformation that achieves the removal of $\mathtt{e}$. Fortunately, however, the post-processing transformation is unnecessary for the applications discussed in the introduction.

3 The Transformation

A basic idea of our translation ($\mathcal{T}_n$ in Theorem 6) to decrease the order of a grammar is to represent an order-1 word function as a tuple of order-0 terms, each of which represents the set of (the tree representations of) words that may be generated before a certain target term (such as an argument, a constant, or a variable) is encountered. For example, consider a term $t\,u$ of type $\mathtt{o}$ where t has type $\mathtt{o} \to \mathtt{o}$. The set of words generated by $t\,u$ is $T_0 \cup (T_1 \cdot U_0)$, where $T_0 = \{w \mid t\,x \longrightarrow^* w(\mathtt{e})\}$, $T_1 = \{w \mid t\,x \longrightarrow^* w(x)\}$ (for a fresh variable x of type $\mathtt{o}$), and $U_0 = \{w \mid u \longrightarrow^* w(\mathtt{e})\}$. In other words, T_0 is the set of words that are generated by t before $\mathtt{e}$ is encountered (without using the argument), and T_1 is the set of words that are generated before the argument is encountered. If we can convert t and u to t_0, t_1, and u_0 that respectively generate tree representations of T_0, T_1, and U_0, then the whole term $t\,u$ can be converted to $t_0 \oplus (\mathtt{br}\,t_1\,u_0)$ (where recall that the binary tree constructor $\mathtt{br}$ plays the role of word concatenation). In this manner, the order-1 term t has been replaced by order-0 terms t_0 and t_1. As a concrete example, consider an order-1 grammar:

$$S \to T\,U \qquad T\,x \to \mathtt{a}\,\mathtt{e} \oplus \mathtt{b}\,x \qquad U \to \mathtt{c}\,\mathtt{e}.$$

Then it can be converted to the order-0 grammar:

$$S \to T_0 \oplus (\mathtt{br}\;T_1\,U_0) \qquad T_0 \to \mathtt{a} \qquad T_1 \to \mathtt{b} \qquad U_0 \to \mathtt{c}.$$

(In the actual translation below, $\mathtt{e}$ should actually be passed around as a variable.)

For higher-order grammars, we apply the above transformation inductively (although a further twist is required as we explain later). For example, consider the grammar (which is a contrived version of Example 5).

$$S \to T\,A\,\mathtt{e} \qquad T\,f\,x \to f(f\,x) \qquad A\,x \to \mathtt{a}\,x.$$

Since the first argument f of T has type $\mathtt{o} \to \mathtt{o}$, it is replaced by an order-0 variable f_1, which is bound to a term that generates (the tree representation of) words generated by f before the argument is encountered. T is then replaced by an order-1 function T_1 which, given f_1, generates words generated by $T\,f\,x$ before x is encountered. The resulting grammar is thus:

$$S \to T_1\,A_1 \qquad T_1\,f_1 \to \mathtt{br}\,f_1\,f_1 \qquad A_1 \to \mathtt{a}.$$

Note that $f(f\,x)$ generates x only after the outer call of f uses the argument $f\,x$, and then the inner call of f uses the argument, hence the body $\mathtt{br}\,f_1\,f_1$ of T_1. Notice that the type of T_1 is $\mathtt{o} \to \mathtt{o}$, which has order 1. In general, a function of type $\kappa_1 \to \cdots \to \kappa_k \to \mathtt{o}^\ell \to \mathtt{o}$ (where $\mathtt{ord}(\kappa_k) > 0$) is converted to a tuple of functions of type $\kappa_1^\dagger \to \cdots \to \kappa_k^\dagger \to \mathtt{o}$, where $(\cdot)^\dagger$ represents recursive applications of the type conversion.

A further twist is required for higher-order cases. For example, consider the grammar:

$$S \to H\,\mathtt{e}\,(\mathtt{b}\,\mathtt{e}) \qquad H\,x\,y \to F(G\,x\,y) \qquad F\,g \to g\,A \qquad G\,x\,y\,h \to (h\,x) \oplus (h(h\,y))$$

where the types of non-terminals are:

$$S : \mathtt{o} \qquad H : \mathtt{o} \to \mathtt{o} \to \mathtt{o} \qquad F : ((\mathtt{o} \to \mathtt{o}) \to \mathtt{o}) \to \mathtt{o} \qquad G : \mathtt{o} \to \mathtt{o} \to (\mathtt{o} \to \mathtt{o}) \to \mathtt{o}$$

(the rule and type of A are as before, hence omitted). Based on the idea above, the rule for S can be translated to:

$$S \to H_1 \oplus (\mathtt{br}\, H_2\, \mathtt{b}),$$

where H_1 (H_2, resp.) generates the words generated by $H\, x\, y$ before x (y, resp.) is encountered in the original grammar. But how can the rules for H_1 and H_2 be obtained from that of H? The head symbol F of the body of H has no order-0 argument, so the idea explained above does not apply. We translate the rules for H, F, and G to:

$$\begin{array}{l} H_1 \to F_0(G_0\, \mathtt{e}\, \Omega) \qquad H_2 \to F_0(G_0\, \Omega\, \mathtt{e}) \qquad F_0\, g_0 \to g_0\, A_1 \\ G_0\, x_0\, y_0\, h_1 \to (\mathtt{br}\, h_1\, x_0) \oplus (\mathtt{br}\, h_1\, (\mathtt{br}\, h_1\, y_0)). \end{array}$$

Here, $F_0\, g_0$ generates the words generated by $F\, g$ before a certain target symbol (say, z) is encountered, assuming that $g_0\, A_1$ generates the words generated by $g\, A$ before z is encountered. In H_1, the argument g_0 is set to $G_0\, \mathtt{e}\, \Omega$ (so that when x is reached in the original grammar, the empty word is generated, and when y is reached in the original grammar, no word is generated) since the target symbol is x, while in H_2, g_0 is set to $G_0\, \Omega\, \mathtt{e}$. Similarly, G_0 is a function to generate the words generated by $G\, x\, y\, h$ in the original grammar before a certain target symbol is encountered, assuming that the arguments x_0 and y_0 generate the words generated by x and y before the target is encountered.

The following is a variation of the example above:

$$\begin{array}{l} S \to H\, \mathtt{e}\, (\mathtt{b}\, \mathtt{e}) \qquad H\, x\, y \to K(G\, x)\, y \qquad K\, k\, y \to F(k\, y) \qquad F\, g \to g\, A \\ G\, x\, y\, h \to (h\, x) \oplus (h(h\, y)) \end{array}$$

where $K : (\mathtt{o} \to (\mathtt{o} \to \mathtt{o}) \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}$. We have just introduced an auxiliary step to reduce $H\, x\, y$ to $F(G\, x\, y)$, via $K(G\, x)\, y$. It can be translated to:

$$\begin{array}{l} S \to H_1 \oplus (\mathtt{br}\, H_2\, \mathtt{b}) \qquad H_1 \to K_0(G_0\, \mathtt{e}) \qquad H_2 \to K_1(G_0\, \Omega) \qquad K_0\, k_0 \to F_0(k_0\, \Omega) \\ K_1\, k_1 \to F_0(k_1\, \mathtt{e}) \qquad F_0\, g_0 \to g_0\, A_1 \qquad G_0\, x_0\, y_0\, h_1 \to (\mathtt{br}\, h_1\, x_0) \oplus (\mathtt{br}\, h_1\, (\mathtt{br}\, h_1\, y_0)). \end{array}$$

Here, K_0 is a function to generate the words generated by $K\, k\, y$ before a certain target (embedded in k) is encountered (thus, $K_0\, (G_0\, \mathtt{e})$ generates the words generated by $K\, (G\, x)\, y$ before the target x is encountered), whereas K_1 is a function to generate the words generated by $K\, k\, y$ before y is encountered. Different arguments $G_0\, \mathtt{e}$ and $G_0\, \Omega$ are, therefore, passed to K_0 and K_1. In $G_0\, \mathtt{e}$, the target is set to x (hence x has been replaced by $\mathtt{e}$), whereas in $G_0\, \Omega$, the target has not been set yet (and is later set to y when $G_0\, \Omega$ is passed to K_1).

Note that the translations above have been chosen to clarify the essence of our transformation; they do not exactly match the actual translations defined below. Henceforth, we often write $\kappa_1 \to \cdots \to \kappa_k \Rightarrow \mathtt{o}^\ell \to \mathtt{o}$ for $\kappa_1 \to \cdots \to \kappa_k \to \mathtt{o}^\ell \to \mathtt{o}$ when either $\mathtt{ord}(\kappa_k) > 0$ or $k = 0$; note that $\kappa_1, \ldots, \kappa_{k-1}$ (but not κ_k) may be the ground type $\mathtt{o}$. We abbreviate $\kappa_1 \to \cdots \to \kappa_k \Rightarrow \mathtt{o}^\ell \to \mathtt{o}$ to $\widetilde{\kappa} \Rightarrow \mathtt{o}^\ell \to \mathtt{o}$, and define $\mathtt{gar}(\widetilde{\kappa} \Rightarrow \mathtt{o}^\ell \to \mathtt{o}) := \ell$. We assume that a given word grammar is normalized to the form $(\Sigma, \mathcal{N}, \mathcal{R}, S\, \mathtt{e})$, where $\mathcal{R}$ does not contain $\mathtt{e}$.

The discussions above suggest that for each term t of an order-n type $\widetilde{\kappa} \Rightarrow \mathtt{o}^\ell \to \mathtt{o}$, with order-0 variables $x_1, \ldots, x_k$, we need to consider the following tuple of order-$(n-1)$ terms in the target grammar:

$$(t_0, t_1, \ldots, t_\ell, t_{\ell+1}, \ldots, t_{\ell+k}, t_{\ell+k+1}).$$

Here, each element t_i of the tuple is a function used to generate (tree representations of) the words generated by t before a certain target is encountered, where the "target" is:

- the i-th ground-type argument (precisely, the i-th argument of the part $\mathsf{o}^\ell \to \mathsf{o}$ in the type $\widetilde{\kappa} \Rightarrow \mathsf{o}^\ell \to \mathsf{o}$ of t) if $1 \le i \le \ell$,
- $x_{i-\ell}$ if $\ell + 1 \le i \le \ell + k$,
- set inside the term t if $i = 0$, and
- set later by the higher-order arguments of t if $i = \ell + k + 1$.

We formalize the translation for terms as a type-based transformation relation of the form:

$$\mathcal{K}; x_1, \dots, x_k \vdash_{\mathcal{N}} t : \widetilde{\kappa} \Rightarrow \mathsf{o}^\ell \to \mathsf{o} \rightsquigarrow (t_0, t_1, \dots, t_\ell, t_{\ell+1}, \dots, t_{\ell+k}, t_{\ell+k+1}).$$

where $\mathcal{K}$ is a type environment and $x_1, \dots, x_k$ are order-0 variables; we often omit the subscript of $\vdash_{\mathcal{N}}$ and write $\vdash$ if $\mathcal{N}$ is clear from the context. The relation means that the term t of type $\widetilde{\kappa} \Rightarrow \mathsf{o}^\ell \to \mathsf{o}$ is transformed to the tuple $(t_0, t_1, \dots, t_\ell, t_{\ell+1}, \dots, t_{\ell+k}, t_{\ell+k+1})$, where the role of each term is as described above. The relation is defined by the transformation rules given in Fig. 1.

The translation $\mathcal{N}^\ddagger$ of the types of nonterminals used in (Tr-Gram) is defined as follows. We first define the translation of types by:

$$\begin{aligned}
&(\kappa_1 \to \dots \to \kappa_k \Rightarrow \mathsf{o}^\ell \to \mathsf{o})^\dagger := \\
&\qquad ({\kappa_1}^\dagger \to \dots \to {\kappa_k}^\dagger \to \mathsf{o}) \times ({\kappa_1}^{\dagger'} \to \dots \to {\kappa_k}^{\dagger'} \to \mathsf{o})^\ell \times ({\kappa_1}^\dagger \to \dots \to {\kappa_k}^\dagger \to \mathsf{o}), \\
&(\kappa_1 \to \dots \to \kappa_k \Rightarrow \mathsf{o}^\ell \to \mathsf{o})^{\dagger'} := ({\kappa_1}^{\dagger'} \to \dots \to {\kappa_k}^{\dagger'} \to \mathsf{o})^\ell \times ({\kappa_1}^\dagger \to \dots \to {\kappa_k}^\dagger \to \mathsf{o}), \\
&(\kappa_1 \to \dots \to \kappa_k \Rightarrow \mathsf{o}^\ell \to \mathsf{o})^{\dagger+m} := \\
&\qquad ({\kappa_1}^\dagger \to \dots \to {\kappa_k}^\dagger \to \mathsf{o}) \times ({\kappa_1}^{\dagger'} \to \dots \to {\kappa_k}^{\dagger'} \to \mathsf{o})^\ell \times ({\kappa_1}^\dagger \to \dots \to {\kappa_k}^\dagger \to \mathsf{o})^{m+1}
\end{aligned}$$

where $m \ge -1$. The translation of type environments is defined by:

$$(y_1 : \kappa_1, \dots, y_k : \kappa_k)^\dagger := (y_{1,0}, \dots, y_{1,\mathtt{gar}(\kappa_1)+1}) : {\kappa_1}^\dagger, \dots, (y_{k,0}, \dots, y_{k,\mathtt{gar}(\kappa_k)+1}) : {\kappa_k}^\dagger.$$

Finally, we define $\mathcal{N}^\ddagger$ by:

$$(F_1 : \kappa_1, \dots, F_k : \kappa_k)^\ddagger := (F_{1,0}, \dots, F_{1,\mathtt{gar}(\kappa_1)}) : {\kappa_1}^{\dagger-1}, \dots, (F_{k,0}, \dots, F_{k,\mathtt{gar}(\kappa_k)}) : {\kappa_k}^{\dagger-1}.$$

As in (Tr-Rule), we translate each rule $F\, y_1 \cdots y_m\, x_1 \dots x_k \to t$ with $\mathcal{N}(F) = \kappa_1 \to \dots \to \kappa_m \Rightarrow \mathsf{o}^k \to \mathsf{o}$ by the relation:

$$y_1 : \kappa_1, \dots, y_m : \kappa_m; x_1, \dots, x_k : \mathsf{o} \vdash_{\mathcal{N}} t : \mathsf{o} \rightsquigarrow (t_0, t_1, \dots, t_k, t_{k+1}).$$

Note that every t being transformed never contains e since $\mathcal{R}$ does not contain e.

We now explain some of the key rules. There are two rules for variables: (Tr-VarG) for ground type variables $x_1, \dots, x_k$, and (Tr-Var) for variables bound in $\mathcal{K}$ (note that some of them also may have ground type o). In (Tr-VarG), the $(i+1)$-th component of the output should represent the words generated before x_i is encountered; since x_i is immediately encountered, the component is set to e. The other components are set to Ω (which generates no word), since no other variable is encountered by reducing x_i. In (Tr-Var), each type binding $y : \kappa$ in $\mathcal{K}$ gets (implicitly) translated to $(y_0, \dots, y_{\ell+1}) : \kappa^\dagger$. In the output of translation, the last $k + 1$ components (which represent the words generated until $x_1, \dots, x_k$ and an unknown target are encountered) are set to $y_{\ell+1}$, because x_i's are unknown for the environment. In Example 8 below, we explain why we use $y_{\ell+1}$ (rather than

$$\frac{}{\mathcal{K}; x_1, \ldots, x_k : \mathsf{o} \vdash_{\mathcal{N}} \Omega : \mathsf{o} \rightsquigarrow (\Omega^{k+2})} \tag{\textsc{Tr-Omega}}$$

$$\frac{}{\mathcal{K}; x_1, \ldots, x_k : \mathsf{o} \vdash_{\mathcal{N}} x_i : \mathsf{o} \rightsquigarrow (\Omega^{i}, \mathsf{e}, \Omega^{k-i+1})} \tag{\textsc{Tr-VarG}}$$

$$\frac{\mathcal{K}(y) = \widetilde{\kappa} \Rightarrow \mathsf{o}^{\ell} \to \mathsf{o}}{\mathcal{K}; x_1, \ldots, x_k : \mathsf{o} \vdash_{\mathcal{N}} y : \widetilde{\kappa} \Rightarrow \mathsf{o}^{\ell} \to \mathsf{o} \rightsquigarrow (y_0, y_1, \ldots, y_{\ell}, (y_{\ell+1})^{k+1})} \tag{\textsc{Tr-Var}}$$

$$\frac{\mathcal{N}(F) = \widetilde{\kappa} \Rightarrow \mathsf{o}^{\ell} \to \mathsf{o}}{\mathcal{K}; x_1, \ldots, x_k : \mathsf{o} \vdash_{\mathcal{N}} F : \widetilde{\kappa} \Rightarrow \mathsf{o}^{\ell} \to \mathsf{o} \rightsquigarrow (F_0, F_1, \ldots, F_{\ell}, (F_0)^{k+1})} \tag{\textsc{Tr-NT}}$$

$$\frac{\mathcal{K}; x_1, \ldots, x_k : \mathsf{o} \vdash_{\mathcal{N}} t : \mathsf{o} \rightsquigarrow (t_0, \ldots, t_{k+1})}{\mathcal{K}; x_1, \ldots, x_k : \mathsf{o} \vdash_{\mathcal{N}} a(t) : \mathsf{o} \rightsquigarrow (\mathtt{br}\ a\ t_0, \ldots, \mathtt{br}\ a\ t_{k+1})} \tag{\textsc{Tr-Const}}$$

$$\frac{\begin{array}{c}\mathcal{K}; x_1, \ldots, x_k : \mathsf{o} \vdash_{\mathcal{N}} s : \kappa_0 \to \widetilde{\kappa} \Rightarrow \mathsf{o}^{\ell} \to \mathsf{o} \rightsquigarrow (s_0, \ldots, s_{\ell+k+1}) \\ \mathcal{K}; x_1, \ldots, x_k : \mathsf{o} \vdash_{\mathcal{N}} t : \kappa_0 \rightsquigarrow (t_0, \ldots, t_{\ell'+k+1}) \qquad \mathtt{gar}(\kappa_0) = \ell'\end{array}}{\begin{array}{c}\mathcal{K}; x_1, \ldots, x_k : \mathsf{o} \vdash_{\mathcal{N}} s\,t : \widetilde{\kappa} \Rightarrow \mathsf{o}^{\ell} \to \mathsf{o} \\ \rightsquigarrow (s_0(t_0, \ldots, t_{\ell'}, t_{\ell'+k+1}), s_1(t_1, \ldots, t_{\ell'}, t_{\ell'+k+1}), \ldots, s_{\ell}(t_1, \ldots, t_{\ell'}, t_{\ell'+k+1}), \\ \qquad s_{\ell+1}(t_{\ell'+1}, t_1, \ldots, t_{\ell'}, t_{\ell'+k+1}), \ldots, s_{\ell+k+1}(t_{\ell'+k+1}, t_1, \ldots, t_{\ell'}, t_{\ell'+k+1}))\end{array}} \tag{\textsc{Tr-App}}$$

$$\frac{\begin{array}{c}\mathcal{K}; x_1, \ldots, x_k : \mathsf{o} \vdash_{\mathcal{N}} s : \mathsf{o}^{\ell+1} \to \mathsf{o} \rightsquigarrow (s_0, \ldots, s_{\ell+k+2}) \\ \mathcal{K}; x_1, \ldots, x_k : \mathsf{o} \vdash_{\mathcal{N}} t : \mathsf{o} \rightsquigarrow (t_0, \ldots, t_{k+1})\end{array}}{\begin{array}{c}\mathcal{K}; x_1, \ldots, x_k : \mathsf{o} \vdash_{\mathcal{N}} s\,t : \mathsf{o}^{\ell} \to \mathsf{o} \\ \rightsquigarrow (s_0 \oplus (\mathtt{br}\ s_1\ t_0), s_2, \ldots, s_{\ell+1}, s_{\ell+2} \oplus (\mathtt{br}\ s_1\ t_1), \ldots, s_{\ell+k+2} \oplus (\mathtt{br}\ s_1\ t_{k+1}))\end{array}} \tag{\textsc{Tr-AppG}}$$

$$\frac{\begin{array}{c}\mathcal{K}; x_1, \ldots, x_k : \mathsf{o} \vdash_{\mathcal{N}} s : \mathsf{o} \rightsquigarrow (s_0, \ldots, s_{k+1}) \\ \mathcal{K}; x_1, \ldots, x_k : \mathsf{o} \vdash_{\mathcal{N}} t : \mathsf{o} \rightsquigarrow (t_0, \ldots, t_{k+1})\end{array}}{\mathcal{K}; x_1, \ldots, x_k : \mathsf{o} \vdash_{\mathcal{N}} s \oplus t : \mathsf{o} \rightsquigarrow (s_0 \oplus t_0, \ldots, s_{k+1} \oplus t_{k+1})} \tag{\textsc{Tr-Choice}}$$

$$\frac{\begin{array}{c}\mathcal{N}(F) = \kappa_1 \to \cdots \to \kappa_m \Rightarrow \mathsf{o}^{k} \to \mathsf{o} \\ y_1 : \kappa_1, \ldots, y_m : \kappa_m; x_1, \ldots, x_k : \mathsf{o} \vdash_{\mathcal{N}} t : \mathsf{o} \rightsquigarrow (t_0, t_1, \ldots, t_k, t_{k+1}) \\ \widetilde{y_i} = (y_{i,0}, \ldots, y_{i,\mathtt{gar}(\kappa_i)+1}) \qquad \widetilde{y_i}' = (y_{i,1}, \ldots, y_{i,\mathtt{gar}(\kappa_i)+1}) \qquad (1 \leq i \leq m)\end{array}}{\vdash_{\mathcal{N}} (F\, y_1\, \cdots\, y_m\, x_1\, \cdots\, x_k \to t) \rightsquigarrow \left(\begin{array}{l}\left\{F_0\, \widetilde{y_1}\, \cdots\, \widetilde{y_m} \to t_0\right\} \cup \\ \left\{F_i\, \widetilde{y_1}'\, \cdots\, \widetilde{y_m}' \to t_i \,\middle|\, i \in \{1, \ldots, k\}\right\}\end{array}\right)} \tag{\textsc{Tr-Rule}}$$

$$\frac{\begin{array}{c}\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S\,\mathsf{e}) \qquad \mathsf{e} \text{ does not occur in } \mathcal{R} \\ \mathcal{R}' = \bigcup\{\mathcal{R}'' \mid \vdash_{\mathcal{N}} (F\,\widetilde{y}\,\widetilde{x} \to t) \rightsquigarrow \mathcal{R}'',\ (F\,\widetilde{y}\,\widetilde{x} \to t) \in \mathcal{R}\}\end{array}}{\mathcal{G} \rightsquigarrow (\mathtt{br}(\Sigma), \mathcal{N}^{\ddagger}, \mathcal{R}', S_1)} \tag{\textsc{Tr-Gram}}$$

Figure 1 Translation rules from a word grammar to a tree grammar.

y_0) for the last $k+1$ components of (Tr-Var). There are also two rules for applications, (Tr-App) and (Tr-AppG). (Tr-AppG) is used when the function s has an order-1 type. The first component of the output is set to $s_0 \oplus (\mathtt{br}\ s_1\ t_0)$, since $s\,t$ encounters the current target either when s does so without using t (which is covered by s_0), or s uses the argument t and then t encounters the target (which is covered by $\mathtt{br}\ s_1\ t_0$). In (Tr-App), the output is just an application, but we need to choose the function and the argument appropriately for each component of the output tuple. Note that κ_0 in (Tr-App) can be $\mathtt{o}$.

▶ **Example 8.** The following example of the translation shows why we need to use $y_{\ell+1}$ rather than y_0 for the last $k+1$ components in (Tr-Var). Let us consider the following grammar:

$$\begin{array}{lll} S\,x \;\;\to F\,x\,(G\,x) & F\,y\,g \to T\,(A\,g)\,y & T\,h\,y \to h(h\,y) \\ A\,g\,y \to g\,(J\,y) & G\,x\,h \to h\,x & J\,y\,z \to (\mathtt{b}\,y) \oplus (\mathtt{c}\,z) \end{array}$$

$$\begin{array}{lll} S : \mathtt{o} \to \mathtt{o}, & F : \mathtt{o} \to ((\mathtt{o} \to \mathtt{o}) \to \mathtt{o}) \to \mathtt{o}, & T : (\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}, \\ A : ((\mathtt{o} \to \mathtt{o}) \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}, & G : \mathtt{o} \to (\mathtt{o} \to \mathtt{o}) \to \mathtt{o}, & J : \mathtt{o} \to \mathtt{o} \to \mathtt{o}. \end{array}$$

Then we have the following reduction sequence:

$$\begin{aligned} &S\,\mathtt{e} \longrightarrow F\,\mathtt{e}\,(G\,\mathtt{e}) \longrightarrow T(A(G\,\mathtt{e}))\mathtt{e} \longrightarrow A(G\,\mathtt{e})(A(G\,\mathtt{e})\mathtt{e}) \longrightarrow G\,\mathtt{e}\,(J\,(A(G\,\mathtt{e})\mathtt{e})) \\ &\quad \longrightarrow J\,(A(G\,\mathtt{e})\mathtt{e})\,\mathtt{e} \longrightarrow \big(\mathtt{b}\,(A(G\,\mathtt{e})\mathtt{e})\big) \oplus (\mathtt{c}\,\mathtt{e}) \longrightarrow \big(\mathtt{b}\,(G\,\mathtt{e}(J\,\mathtt{e}))\big) \oplus (\mathtt{c}\,\mathtt{e}) \\ &\quad \longrightarrow \big(\mathtt{b}\,(J\,\mathtt{e}\,\mathtt{e})\big) \oplus (\mathtt{c}\,\mathtt{e}) \longrightarrow \big(\mathtt{b}\,((\mathtt{b}\,\mathtt{e}) \oplus (\mathtt{c}\,\mathtt{e}))\big) \oplus (\mathtt{c}\,\mathtt{e}) \longrightarrow \big((\mathtt{b}\,(\mathtt{b}\,\mathtt{e})) \oplus (\mathtt{b}\,(\mathtt{c}\,\mathtt{e}))\big) \oplus (\mathtt{c}\,\mathtt{e}). \end{aligned}$$

Thus, the language generated by the grammar is $\{\mathtt{bb}, \mathtt{bc}, \mathtt{c}\}$. The translation produces:

$$\begin{array}{ll} \multicolumn{2}{l}{S_1 \to F_0\,(\mathtt{e},\,\Omega)\,(G_0(\mathtt{e},\Omega),\,G_0(\Omega,\Omega))} \\ \multicolumn{2}{l}{F_0\,(y_0,y_1)\,(g_0,g_1) \to \big(T_0(A_0(g_0,g_1),\;A_1(g_1),\;A_0(g_1,g_1))\big)} \\ \multicolumn{2}{l}{\qquad\qquad\qquad\quad \oplus\big(\mathtt{br}\;\big(T_1(\qquad\qquad\quad A_1(g_1),\;A_0(g_1,g_1))\big)\,y_0\big)} \\ T_0(h_0,h_1,h_2) \to h_0 \oplus (\mathtt{br}\,h_1\,h_0) & A_0(g_0,g_1) \to g_0(\mathtt{br}\,J_1\,\Omega,\;J_2,\;\mathtt{br}\,J_1\,\Omega) \\ T_1(\quad\; h_1,h_2) \to \mathtt{br}\,h_1\,(\mathtt{br}\,h_1\,\mathtt{e}) & A_1(\quad\; g_1) \to g_1(\mathtt{br}\,J_1\,\mathtt{e}\,,\;J_2,\;\mathtt{br}\,J_1\,\Omega) \\ \multicolumn{2}{l}{G_0\,(x_0,x_1)\,(h_0,h_1,h_2) \to h_0 \oplus (\mathtt{br}\,h_1\,x_0)} \\ \multicolumn{2}{l}{J_0 \to \Omega \qquad J_1 \to \mathtt{b} \qquad J_2 \to \mathtt{c}.} \end{array}$$

The frontier language generated by the output grammar is $\{\mathtt{bb}, \mathtt{bc}, \mathtt{c}\}$. If we changed the definition of (Tr-Var) by replacing y_{k+1} with y_0, then $A_1(g_1)$ in the body of F_0 above would be replaced with $A_1(g_0)$. – But then we would wrongly obtain $\mathtt{cc}$ as a member of the frontier language.

The following theorem guarantees that the output of the translation is a valid grammar; the preservation of the language will be proved later in Section 5.

▶ **Theorem 9.** *Suppose that $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S\,\mathtt{e})$ is an order-$(n+1)$ word grammar, and $\mathcal{G} \rightsquigarrow \mathcal{G}'$. Then $\mathcal{G}'$ is a (well-typed) order-n tree grammar.*

The well-typedness of $\mathcal{G}'$ follows immediately from the following lemma, which can be proved by a straightforward induction.

▶ **Lemma 10.** *If $\mathcal{K}; \widetilde{x} : \mathtt{o} \vdash_{\mathcal{N}} t : \kappa \rightsquigarrow (\widetilde{t})$, then $\mathcal{N}^{\ddagger}, \mathcal{K}^{\dagger} \vdash (\widetilde{t}) : \kappa^{\dagger+|\tilde{x}|}$. Furthermore for $y \in dom(\mathcal{K})$, y_0 does not occur freely in $\widetilde{t}$ except for t_0.*

4 Complexity and Application

Here we give some complexity results of our translation: an upper bound of the size of an output grammar and the time complexity of the translation. Based on the complexity results, we discuss an application of our translation.

For a grammar $\mathcal{G}$, we write $A_{\mathcal{G}}$ (or A if $\mathcal{G}$ is clear from the context) for the largest arity of the terminals, variables, and nonterminals occurring in $\mathcal{G}$. Note that, for an input word grammar $\mathcal{G}$, we have $A_{\mathcal{G}} \leq |\mathcal{N}| \leq |\mathcal{G}|_{\mathrm{ch}}$ and $|\mathcal{G}|_{\mathrm{cy}} \leq |\mathcal{G}|_{\mathrm{ch}}$, but it is not necessary the case that $A_{\mathcal{G}} \leq |\mathcal{G}|_{\mathrm{cy}}$ (for any κ, $\{S \to F(G\,\mathtt{e}),\ \ F\,f^{\kappa\to\mathrm{o}} \to \mathtt{e},\ \ G\,x^{\mathrm{o}}\,g^{\kappa} \to \mathtt{e}\}$ is well-typed and $A_{\mathcal{G}} \geq \mathtt{ari}(\kappa)$). Also note that $A_{\mathcal{G}} \geq 1$ for an input grammar $\mathcal{G}$ (since the start term is $S\,\mathtt{e}$). For $r = (F\,\widetilde{x} \to t) \in \mathcal{R}$, we define $|r| := |t| + \mathtt{ari}(\mathcal{N}(F))$, so that $|\mathcal{G}|_{\mathrm{cy}} = |t^{\circ}| + \sum_{r\in\mathcal{R}} |r|$; also, we write $\mathcal{R}_r$ for $\mathcal{R}'$ such that $\vdash_{\mathcal{N}} r \rightsquigarrow \mathcal{R}'$ by (Tr-Rule).

4.1 Complexity

To suppress the blow-up of the cost of our translation, we pre-process an input grammar to convert it to a certain normal form, as in [8]. A grammar $\mathcal{G}$ is in *normal form* if each rule is of the form

$$F\,\widetilde{z} \to \big(f_{1,0}(f_{1,1}\widetilde{z_{1,1}})\cdots(f_{1,\ell_1}\widetilde{z_{1,\ell_1}})\big) \oplus \cdots \oplus \big(f_{k,0}(f_{k,1}\widetilde{z_{k,1}})\cdots(f_{k,\ell_k}\widetilde{z_{k,\ell_k}})\big),$$

where each $f_{h,i}$ is a non-terminal, terminal, or variable; and each $z_{h,i,j}$ is a variable; note that k and each ℓ_h can be 0. (Rigorously, the expression like $t_1 \oplus \cdots \oplus t_k$ can be represented as $D[t_1]\cdots[t_k]$ with $D ::= [\,] \mid \Omega \mid D \oplus D$.) By the following transformation, any grammar $\mathcal{G}$ can be transformed to a grammar in normal form: First note that any rule is of the form

$$F\,\widetilde{z} \to \big(f_{1,0}\,t_{1,1}\,\ldots\,t_{1,\ell_1}\big) \oplus \cdots \oplus \big(f_{k,0}\,t_{k,1}\,\ldots\,t_{k,\ell_k}\big)$$

where each $f_{h,i}$ is a non-terminal, terminal, or variable, and each $t_{h,i}$ is a term (which may contain $\oplus$ and Ω). If there exist h and i such that $t_{h,i}$ is not of the form $f_{h,i}\widetilde{z_{h,i}}$, then we replace the occurrence $t_{h,i}$ with $G\,\widetilde{z_{h,i}}$ and add the rule $G\,\widetilde{z_{h,i}}\,\widetilde{y} \to t_{h,i}\,\widetilde{y}$, where G is a fresh non-terminal and $\{\widetilde{z_{h,i}}\}$ $(\subseteq \{\widetilde{z}\})$ is all the free variables occurring in $t_{h,i}$. By repeated applications of this transformation, we obtain a grammar in normal form; note that the number of repeated applications is at most $|\mathcal{G}|_{\mathrm{cy}}$. The following lemma guarantees that the preprocessing transformation does not blow up the grammar size; see the full version for the proof.

▶ **Lemma 11.** *Let $\mathcal{G}' = (\Sigma, \mathcal{N}', \mathcal{R}', S\,\mathtt{e})$ be the grammar obtained from $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S\,\mathtt{e})$ by the above transformation. We write A for $A_{\mathcal{G}}$. Then,*

1. $A_{\mathcal{G}'} \leq 2A$*,*
2. $|\mathcal{N}'| \leq O(|\mathcal{N}| \times |\mathcal{G}|_{\mathrm{cy}})$
3. $|\mathcal{G}'|_{\mathrm{cy}} \in O(A \times |\mathcal{G}|_{\mathrm{cy}})$*,*
4. *the time complexity of the normal-form transformation is* $O(A \times |\mathcal{G}|_{\mathrm{cy}})$*,*
5. $\mathtt{ord}(\mathcal{G}') = \mathtt{ord}(\mathcal{G})$*.*

We obtain the following complexity results; see Appendix A for the proof.

▶ **Theorem 12.** *Let $\mathcal{G}'$ be the output of our translation (i.e., the composite of the normal-form transformation and that of Fig. 1) for an input grammar $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S\,\mathtt{e})$. We write A for $A_{\mathcal{G}}$. Then,*

1. $A_{\mathcal{G}'} \in O(A^2)$*,*
2. $|\mathcal{G}'|_{\mathrm{cy}} \in O(A^4 \times |\mathcal{G}|_{\mathrm{cy}})$ *and* $|\mathcal{G}'|_{\mathrm{ch}} \in O(A^4 \times |\mathcal{G}|_{\mathrm{cy}} + (|\mathcal{N}| \times |\mathcal{G}|_{\mathrm{cy}})^n)$ *where* $n = \mathtt{ord}(\mathcal{G})$*,*
3. *the time complexity of our translation is* $O((A^3 + |\mathcal{N}|) \times |\mathcal{G}|_{\mathrm{cy}})$*.*

4.2 Applications

Our translation can be used to bridge results on higher-order word and tree grammars. Besides applications of our previous translation [1], our new *size-preserving* transformation can be applied in contexts where the complexity is critical. For example, Parys [12] showed that the diagonal problem for higher-order grammars (which is called *nondeterministic higher-order recursion schemes* in [12]) is $(n-1)$-EXPTIME-complete for order-n word grammars, and n-EXPTIME-complete for order-n tree grammars. In his paper, the two results, for words and for trees, were proved separately. By using the result in the present paper, however, the $(n-1)$-EXPTIME-completeness of the diagonal problem for word grammars of order n immediately follows from the n-EXPTIME-completeness of the diagonal problem for tree grammars of order n (where, on the hardness part, use the converse transformation given in [1, Theorem 5]).

5 Correctness of the Translation

Here we prove the following theorem, which states the correctness of the translation.

▶ **Theorem 13.** *If $\mathcal{G} \rightsquigarrow \mathcal{G}'$, then $\mathcal{L}_{\mathtt{w}}(\mathcal{G}) = \mathcal{L}_{\mathtt{leaf}}(\mathcal{G}')\uparrow_{\mathtt{e}}$.*

We show this theorem in the following steps:

- First, in Section 5.1, we reduce the proof to the case where $\mathcal{G}$ is *recursion-free* (Lemma 15). This is rather a standard technique, which uses the *finite approximation* $\mathcal{G}^{(m)}$: the reduction of $\mathcal{G}^{(m)}$ imitates that of $\mathcal{G}$ up to m-steps, but diverges after m-steps.
- Then we show the statement of Theorem 13 with the assumption that $\mathcal{G}$ is recursion-free (Lemma 21). This is the main part, proved by using the subject reduction property. In the proof of the subject reduction, we modify the reduction $\longrightarrow_{\mathcal{G}}$ of the source grammar $\mathcal{G}$, by using *explicit substitutions*. We define the modified reduction in Section 5.2, and then we show the subject reduction and the correctness for a recursion-free grammar in Section 5.3.

5.1 Reduction of the correctness to recursion-free grammars

▶ **Definition 14** (recursion-free grammars). *A grammar $\mathcal{G}$ is called* recursion-free *if there is no cyclic dependency on its non-terminals. Precisely, we define a binary relation $\succ_{\mathcal{G}}$ on nonterminals of $\mathcal{G}$ by: $F \succ_{\mathcal{G}} F'$ iff F' occurs on the right-hand side of the rule for F; then $\mathcal{G}$ is recursion free if the transitive closure $\succ_{\mathcal{G}}^*$ of $\succ_{\mathcal{G}}$ is irreflexive (i.e., $F \succ_{\mathcal{G}}^* F$ for no $F \in \mathcal{N}$).*

Here we show the following lemma:

▶ **Lemma 15.** *Suppose that, for any $\mathcal{G}$ and $\mathcal{G}'$, if $\mathcal{G}$ is recursion-free and $\mathcal{G} \rightsquigarrow \mathcal{G}'$, then $\mathcal{L}_{\mathtt{w}}(\mathcal{G}) = \mathcal{L}_{\mathtt{leaf}}(\mathcal{G}')\uparrow_{\mathtt{e}}$. Then, for any $\mathcal{G}$ and $\mathcal{G}'$, if $\mathcal{G} \rightsquigarrow \mathcal{G}'$, then $\mathcal{L}_{\mathtt{w}}(\mathcal{G}) = \mathcal{L}_{\mathtt{leaf}}(\mathcal{G}')\uparrow_{\mathtt{e}}$.*

To show this lemma, we define the finite approximation mentioned above. Given terms t, s_i $(i \in I)$ and variables x_i $(i \in I)$, we write $[s_i/x_i]_{i\in I}t$ for the term obtained by simultaneously substituting s_i for x_i in t. Given $m \geq 0$ and $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, t^\circ)$, we define the *$m$-th approximation* $\mathcal{G}^{(m)}$ of $\mathcal{G}$ as follows.

$$\begin{aligned}
\mathcal{N}^{(m)} &:= \{F^{(i)} \mapsto \mathcal{N}(F) \mid F \in dom(\mathcal{N}), 0 \leq i \leq m\}\\
t_{\mathcal{N}}^{(i)} &:= [F^{(i)}/F]_{F\in\mathcal{N}}t \qquad \text{(for any term } t \text{ and } i \in \{0,\dots,m\})\\
\mathcal{R}_{\mathcal{N}}^{(m)} &:= \{F^{(i)}\,\widetilde{x} \to t_{\mathcal{N}}^{(i-1)} \mid (F\,\widetilde{x} \to t) \in \mathcal{R}, 1 \leq i \leq m\}\\
&\quad\cup \{F^{(0)}\,\widetilde{x} \to \Omega \mid (F\,\widetilde{x} \to t) \in \mathcal{R}\}\\
\mathcal{G}^{(m)} &:= (\Sigma, \mathcal{N}^{(m)}, \mathcal{R}_{\mathcal{N}}^{(m)}, (t^\circ)_{\mathcal{N}}^{(m)}).
\end{aligned}$$

We write $t^{(i)}$ and $\mathcal{R}^{(m)}$ for $t^{(i)}_{\mathcal{N}}$ and $\mathcal{R}^{(m)}_{\mathcal{N}}$ if $\mathcal{N}$ is clear from the context.

The following are basic properties of $\mathcal{G}^{(m)}$:

▶ **Lemma 16.**

1. *If $\mathcal{G}$ is in the domain of the transformation $\rightsquigarrow$ (i.e., $\mathcal{G}$ is of the form $(\Sigma, \mathcal{N}, \mathcal{R}, S\,\mathtt{e})$ where Σ is a word alphabet and $\mathcal{R}$ does not contain $\mathtt{e}$), then so is $\mathcal{G}^{(m)}$.*
2. *$\mathcal{G}^{(m)}$ is a recursion-free grammar.*
3. *$\mathcal{L}(\mathcal{G}) = \cup_m \mathcal{L}(\mathcal{G}^{(m)})$. Hence, $\mathcal{L}_{\mathtt{w}}(\mathcal{G}) = \cup_m \mathcal{L}_{\mathtt{w}}(\mathcal{G}^{(m)})$ for any word grammar $\mathcal{G}$, and $\mathcal{L}_{\mathtt{leaf}}(\mathcal{G})\!\uparrow_{\mathtt{e}} = \cup_m \mathcal{L}_{\mathtt{leaf}}(\mathcal{G}^{(m)})\!\uparrow_{\mathtt{e}}$ for any $\mathtt{br}$-grammar $\mathcal{G}$.*

Proof. Item 1 is clear, and also Item 2 is clear (since $F^{(i)} \succ_{\mathcal{G}^{(m)}} G^{(j)}$ implies $j = i - 1$).

Item 3: Let $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, t^\circ)$. To show $\cup_m \mathcal{L}(\mathcal{G}^{(m)}) \subseteq \mathcal{L}(\mathcal{G})$, let $v \in \mathcal{L}(\mathcal{G}^{(m)})$; i.e., there exist a choice context C and a reduction sequence Π in $\mathcal{G}^{(m)}$ from $(t^\circ)^{(m)}$ to $C[v]$. Let us call a rule in $\{F^{(0)}\,\widetilde{x} \to \Omega \mid (F\,\widetilde{x} \to t) \in \mathcal{R}\}$ a *bottom rule*. We can easily show the following fact: if $t \longrightarrow s$ by some bottom rule r and $s \longrightarrow u$, then there is a term u' such that $t \longrightarrow u'$ and $u' \longrightarrow^* u$ where the latter rewriting uses only r. By using this fact repeatedly, from Π, we obtain a choice context C' and a reduction sequence Π' in $\mathcal{G}^{(m)}$ from $(t^\circ)^{(m)}$ to $C'[v]$ where Π' does not use any bottom rule. Then, by dropping the index i of every nonterminal $F^{(i)}$ in Π', we obtain a reduction sequence in $\mathcal{G}$ from t° to $C''[v]$ for some C''.

Conversely, let $v \in \mathcal{L}(\mathcal{G})$. We have C and a reduction sequence Π in $\mathcal{G}$ from t° to $C[v]$; let m be the length of Π. Then there exist C' and a reduction sequence Π' in $\mathcal{G}^{(m)}$ from $(t^\circ)^{(m)}$ to $C'[v]$ (such that Π is obtained from Π' by dropping the indices of all the nonterminals in Π'). Hence $v \in \mathcal{L}(\mathcal{G}^{(m)})$, and thus $\mathcal{L}(\mathcal{G}) \subseteq \cup_m \mathcal{L}(\mathcal{G}^{(m)})$. ◀

The following lemma is trivial; see the full version for the proof.

▶ **Lemma 17.** *If $\mathcal{G} \rightsquigarrow \mathcal{G}'$ and $\mathcal{G}^{(m)} \rightsquigarrow \mathcal{G}'_m$, then $\mathcal{L}_{\mathtt{leaf}}(\mathcal{G}'^{(m)})\!\uparrow_{\mathtt{e}} = \mathcal{L}_{\mathtt{leaf}}(\mathcal{G}'_m)\!\uparrow_{\mathtt{e}}$.*

Now we can show:

Proof of Lemma 15. Let $\mathcal{G} \rightsquigarrow \mathcal{G}'$ and $\mathcal{G}^{(m)} \rightsquigarrow \mathcal{G}'_m$ for each m. Then, by the assumption,

$$\mathcal{L}_{\mathtt{w}}(\mathcal{G}^{(m)}) \quad = \quad \mathcal{L}_{\mathtt{leaf}}(\mathcal{G}'_m)\!\uparrow_{\mathtt{e}}. \tag{1}$$

Then,

$$\begin{aligned}
\mathcal{L}_{\mathtt{w}}(\mathcal{G}) &= \cup_m \mathcal{L}_{\mathtt{w}}(\mathcal{G}^{(m)}) && \text{(by Lemma 16-3)}\\
&= \cup_m \mathcal{L}_{\mathtt{leaf}}(\mathcal{G}'_m)\!\uparrow_{\mathtt{e}} && \text{(by (1))}\\
&= \cup_m \mathcal{L}_{\mathtt{leaf}}(\mathcal{G}'^{(m)})\!\uparrow_{\mathtt{e}} && \text{(by Lemma 17)}\\
&= \mathcal{L}_{\mathtt{leaf}}(\mathcal{G}')\!\uparrow_{\mathtt{e}}. && \text{(by Lemma 16-3).}
\end{aligned}$$

◀

5.2 The modified reduction of source grammars

As explained at the beginning of this section, we modify the reduction relation of a word grammar $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, t^\circ)$ by using explicit substitutions. We first extend the set of terms as follows, which we call *extended terms*:

$$t ::= x \mid \mathtt{e} \mid a(t) \mid t_1\, t_2 \mid t_1 \oplus t_2 \mid \Omega \mid \{t_1/x_1, \dots, t_\ell/x_\ell\}t_0$$

Here, we write $a(t)$ instead of $a\,t$, to emphasize that a is a unary constructor. Recall that nonterminals are included in variables x. The term $\{t_1/x_1, \dots, t_\ell/x_\ell\}t_0$ is an *explicit substitution* and is limited to the ground type:

$$\frac{\mathcal{K} \vdash_{\mathtt{ST}} t_i : \mathtt{o} \quad (i \in \{1, \dots, \ell\}) \qquad \mathcal{K}, x_1 : \mathtt{o}, \dots, x_\ell : \mathtt{o} \vdash_{\mathtt{ST}} t_0 : \mathtt{o}}{\mathcal{K} \vdash_{\mathtt{ST}} \{t_1/x_1, \dots, t_\ell/x_\ell\}t_0 : \mathtt{o}}$$

We call the original notions of a term, evaluation context, and choice context a *non-extended term*, *non-extended evaluation context*, and *non-extended choice context*, respectively, if we need the clarity. We use the same meta-variables s, t, u for the extended terms, and also use E and C for the extended evaluation contexts and extended choice contexts defined below. We avoid this ambiguity as follows: in this subsection, these meta-variables range over the extended notions unless we declare otherwise; in the next subsection (i.e., in Section 5.3 for the proof of the correctness for recursion-free grammars), the source term of the transformation relation $\rightsquigarrow$ is an extended one and the target term is a non-extended one.

We define the *extended evaluation contexts* by the following grammar:

$$E ::= [\,] \mid a(E) \mid E \oplus t \mid t \oplus E \mid \{t_1/x_1, \dots, t_\ell/x_\ell\}E.$$

We often abbreviate $\{t_1/x_1, \dots, t_\ell/x_\ell\}t_0$ and $\{t_1/x_1, \dots, t_\ell/x_\ell\}E$ to $\{\widetilde{t}/\widetilde{x}\}t_0$ and $\{\widetilde{t}/\widetilde{x}\}E$, respectively. Then the *modified reduction*, written as $\longrightarrow_{\mathtt{es},\mathcal{G}}$ (or $\longrightarrow_{\mathtt{es}}$ if $\mathcal{G}$ is clear), is defined as follows:

$$\dfrac{\begin{array}{c} F\,\widetilde{y}\,\widetilde{z} \to u \in \mathcal{R} \\ \mathcal{N}(F) = \widetilde{\kappa} \Rightarrow \mathtt{o}^\ell \to \mathtt{o} \qquad |\widetilde{\kappa}| = |\widetilde{y}| = |\widetilde{s}| \qquad \ell = |\widetilde{z}| = |\widetilde{t}| \qquad \widetilde{z} \text{ do not occur in } E[F\,\widetilde{s}\,\widetilde{t}] \end{array}}{E[F\,\widetilde{s}\,\widetilde{t}] \longrightarrow_{\mathtt{es}} E[\{\widetilde{t}/\widetilde{z}\}[\widetilde{s}/\widetilde{y}]u]}$$

$$\dfrac{}{E[\{\widetilde{s}/\widetilde{z}\}z_i] \longrightarrow_{\mathtt{es}} E[s_i]} \qquad \dfrac{x \notin \{z_1, \dots, z_{|\widetilde{s}|}\} \cup dom(\mathcal{N})}{E[\{\widetilde{s}/\widetilde{z}\}x] \longrightarrow_{\mathtt{es}} E[x]}$$

$$\dfrac{}{E[\{\widetilde{s}/\widetilde{z}\}\mathsf{e}] \longrightarrow_{\mathtt{es}} E[\mathsf{e}]} \qquad \dfrac{}{E[\{\widetilde{s}/\widetilde{z}\}a(t)] \longrightarrow_{\mathtt{es}} E[a(\{\widetilde{s}/\widetilde{z}\}t)]}$$

$$\dfrac{}{E[\{\widetilde{s}/\widetilde{z}\}(t_1 \oplus t_2)] \longrightarrow_{\mathtt{es}} E[(\{\widetilde{s}/\widetilde{z}\}t_1) \oplus (\{\widetilde{s}/\widetilde{z}\}t_2)]} \qquad \dfrac{}{E[a(t_1 \oplus t_2)] \longrightarrow_{\mathtt{es}} E[a(t_1) \oplus a(t_2)]}$$

We define the *extended choice contexts* by: $C ::= [\,] \mid C \oplus t \mid t \oplus C$. For $\mathcal{N} \vdash_{\mathtt{ST}} t : \mathtt{o}$, we define the languages generated by a term and by a grammar with respect to $\longrightarrow_{\mathtt{es}}$ as follows:

$$\mathcal{L}^{\mathtt{es}}_{\mathtt{w}}(\mathcal{G}, t) = \{\mathbf{word}(v) \mid t \longrightarrow^*_{\mathtt{es}} C[v]\} \qquad\qquad \mathcal{L}^{\mathtt{es}}_{\mathtt{w}}(\mathcal{G}) = \mathcal{L}^{\mathtt{es}}_{\mathtt{w}}(\mathcal{G}, t^\circ).$$

There is an obvious function ψ from the extended terms to the non-extended terms that performs every explicit substitution as the real substitution. The following lemma can be proved in a standard manner; see the full version for the proof.

▶ **Lemma 18.** *Let $\mathcal{G}$ be a word grammar.*

1. *For any extended term $\mathcal{N} \vdash_{\mathtt{ST}} t : \mathtt{o}$, we have $\mathcal{L}_{\mathtt{w}}(\mathcal{G}, \psi(t)) = \mathcal{L}^{\mathtt{es}}_{\mathtt{w}}(\mathcal{G}, t)$. Especially, $\mathcal{L}_{\mathtt{w}}(\mathcal{G}) = \mathcal{L}^{\mathtt{es}}_{\mathtt{w}}(\mathcal{G})$.*
2. *For any extended terms $\mathcal{N} \vdash t, t' : \mathtt{o}$, if $t \longrightarrow_{\mathtt{es}} t'$, then $\mathcal{L}^{\mathtt{es}}_{\mathtt{w}}(\mathcal{G}, t) = \mathcal{L}^{\mathtt{es}}_{\mathtt{w}}(\mathcal{G}, t')$.*

We also extend the transformation relation to handle explicit substitutions:

$$\dfrac{\begin{array}{c} \mathcal{K}; x_1, \dots, x_k : \mathtt{o} \vdash s_i : \mathtt{o} \rightsquigarrow (s_{i,0}, \dots, s_{i,k+1}) \qquad (i \in \{1, \dots, \ell\}) \\ \mathcal{K}; z_1, \dots, z_\ell, x_1, \dots, x_k : \mathtt{o} \vdash t : \mathtt{o} \rightsquigarrow (t_0, \dots, t_{\ell+k+1}) \end{array}}{\begin{array}{l} \mathcal{K}; x_1, \dots, x_k : \mathtt{o} \vdash \{\widetilde{s}/\widetilde{z}\}t : \mathtt{o} \rightsquigarrow \big(t_0 \oplus \bigoplus_{i=1}^{\ell}(\mathtt{br}\ t_i\ s_{i,0}), \\ \qquad t_{\ell+1} \oplus \bigoplus_{i=1}^{\ell}(\mathtt{br}\ t_i\ s_{i,1}),\ \dots,\ t_{\ell+k+1} \oplus \bigoplus_{i=1}^{\ell}(\mathtt{br}\ t_i\ s_{i,k+1})\big) \end{array}} \quad (\textsc{Tr-Sub})$$

5.3 Correctness for recursion-free grammars

To show the correctness for recursion-free grammars, we use the following substitution lemma and the subject reduction property:

▶ **Lemma 19** (substitution lemma)**.** *Suppose that u contains no explicit substitutions (i.e., no subterms of the form $\{\widetilde{t'}/\widetilde{z'}\}s'$) and*

$$\begin{aligned}
&\widetilde{y} : \widetilde{\kappa}; \widetilde{z} : \mathsf{o} \vdash u : \kappa' \rightsquigarrow (u_0, \dots, u_{\mathtt{gar}(\kappa')+|\tilde{z}|}, u_{\mathtt{gar}(\kappa')+|\tilde{z}|+1})\\
&\widetilde{x} : \mathsf{o} \vdash s'_i : \kappa_i \rightsquigarrow (s'_{i,0}, \dots, s'_{i,\ell_i+k+1}) \qquad \ell_i = \mathtt{gar}(\kappa_i) \qquad (i = 1, \dots, |\tilde{\kappa}|)\\
&k = |\tilde{x}| \qquad \{\widetilde{x}\} \cap \{\widetilde{z}\} = \emptyset.
\end{aligned}$$

We define the following substitution functions

$$\begin{aligned}
\theta_j &= \theta_{1,j} \cdots \theta_{|\tilde{\kappa}|,j} \qquad (j = 0, \dots, k)\\
\theta_{i,0} &= [s'_{i,0}/y_{i,0}, s'_{i,1}/y_{i,1}, \dots, s'_{i,\ell_i}/y_{i,\ell_i}, s'_{i,\ell_i+k+1}/y_{i,\ell_i+1}] \qquad (i = 1, \dots, |\tilde{\kappa}|)\\
\theta_{i,j} &= [s'_{i,\ell_i+j}/y_{i,0}, s'_{i,1}/y_{i,1}, \dots, s'_{i,\ell_i}/y_{i,\ell_i}, s'_{i,\ell_i+k+1}/y_{i,\ell_i+1}]\\
&\qquad\qquad\qquad\qquad (i = 1, \dots, |\tilde{\kappa}|,\ j = 1, \dots, k+1).
\end{aligned}$$

Then we have:

1. $\widetilde{z}, \widetilde{x} : \mathsf{o} \vdash [\widetilde{s'}/\widetilde{y}]u : \kappa' \rightsquigarrow (\theta_0 u_0,\ \dots,\ \theta_0 u_{\mathtt{gar}(\kappa')+|\tilde{z}|},\ \theta_1 u_0,\ \dots,\ \theta_k u_0,\ \theta_0 u_{\mathtt{gar}(\kappa')+|\tilde{z}|+1})$
2. $\theta_0 u_{\mathtt{gar}(\kappa')+|\tilde{z}|+1} \ = \ \theta_{k+1} u_0.$

Proof. Both items can be shown by induction on u and case analysis on the last rule used for the derivation $\widetilde{y} : \widetilde{\kappa}; \widetilde{z} : \mathsf{o} \vdash u : \kappa' \rightsquigarrow (u_0, \dots, u_{\mathtt{gar}(\kappa')+|\tilde{z}|+1})$. See Appendix B. ◀

▶ **Lemma 20** (subject reduction)**.** *If $x_1, \dots, x_k : \mathsf{o} \vdash s : \mathsf{o} \rightsquigarrow (s_0, \dots, s_{k+1})$ and $s \longrightarrow_{\mathtt{es}} t$, then there exist $t_0, \dots, t_{k+1}$ such that $x_1, \dots, x_k : \mathsf{o} \vdash t : \mathsf{o} \rightsquigarrow (t_0, \dots, t_{k+1})$ and $s_i \approx t_i$ for each $i \in \{1, \dots, k+1\}$.*

Proof. We use Lemma 19. The proof proceeds by induction on the derivation of $\widetilde{x}{:}\mathsf{o} \vdash s : \mathsf{o} \rightsquigarrow (s_0, \dots, s_{k+1})$ and case analysis on evaluation contexts and redexes. See Appendix B. ◀

Now we show the correctness for recursion-free grammars; as explained already, Theorem 13 follows immediately from this and Lemma 15.

▶ **Lemma 21.** *Suppose that $\mathcal{G}$ is recursion-free. If $\mathcal{G} \rightsquigarrow \mathcal{G}'$, then $\mathcal{L}_{\mathtt{w}}(\mathcal{G}) = \mathcal{L}_{\mathtt{leaf}}(\mathcal{G}')\!\uparrow_{\mathtt{e}}$.*

Proof. Since $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S\,\mathsf{e})$ is recursion-free, every term $x : \mathsf{o} \vdash s : \mathsf{o}$ is strongly normalizing with respect to $\longrightarrow_{\mathtt{es}}$. Since the reduction relation is finitely branching, the length of reduction sequences from s is bounded. Let $\#(s)$ be the length of the longest reduction sequence from s.

Now we show that

$$x : \mathsf{o} \vdash_{\mathcal{N}} s : \mathsf{o} \rightsquigarrow (s_0, s_1, s_2) \qquad \text{implies} \qquad \mathcal{L}^{\mathtt{es}}_{\mathtt{w}}(\mathcal{G}, [\mathsf{e}/x]s) = \mathcal{L}_{\mathtt{leaf}}(\mathcal{G}', s_1)\!\uparrow_{\mathtt{e}} \tag{2}$$

by induction on $\#(s)$. If (2) holds, then we can complete the proof as follows: let the rule of S be $S\,x \to s$; then $\mathcal{G}'$ has the rule $S_1 \to s_1$, and by Lemma 18-1 and (2) we have

$$\mathcal{L}_{\mathtt{w}}(\mathcal{G}) = \mathcal{L}^{\mathtt{es}}_{\mathtt{w}}(\mathcal{G}, S\,\mathsf{e}) = \mathcal{L}^{\mathtt{es}}_{\mathtt{w}}(\mathcal{G}, [\mathsf{e}/x]s) = \mathcal{L}_{\mathtt{leaf}}(\mathcal{G}', s_1)\!\uparrow_{\mathtt{e}} = \mathcal{L}_{\mathtt{leaf}}(\mathcal{G}')\!\uparrow_{\mathtt{e}}.$$

In the base case that $\#(s) = 0$, i.e., in the case where s is a normal form, first note that, given a term t satisfies $x_1 : \mathsf{o}, \dots, x_n : \mathsf{o} \vdash_{\mathtt{ST}} t : \mathsf{o}$ for some $x_1, \dots, x_n$, then t is a normal form with respect to $\longrightarrow_{\mathtt{es}}$ iff t is generated by the following grammar with start symbol w:

$$w ::= r \mid w \oplus w \qquad r ::= \delta \mid x \mid a(r)\ (a \in \Sigma) \qquad \delta ::= \Omega \mid \{t_1/x_1, \dots, t_k/x_k\}\delta.$$

Then we can show (2) by induction on this grammar. (For the case $s = \delta$, show that $x_1, \ldots, x_k : \mathtt{o} \vdash \delta : \mathtt{o} \rightsquigarrow (\widetilde{s})$ implies $s_i \approx \Omega$ for every i, by induction on δ.)

In the case $\#(s) > 0$, suppose $s \longrightarrow_{\mathtt{es}} s'$. Hence $[\mathtt{e}/x]s \longrightarrow_{\mathtt{es}} [\mathtt{e}/x]s'$. By Lemma 20, there exist s'_0, s'_1, and s'_2 such that $x : \mathtt{o} \vdash_{\mathcal{N}} s' : \mathtt{o} \rightsquigarrow (s'_0, s'_1, s'_2)$ and $s_1 \approx s'_1$. By the induction hypothesis, $\mathcal{L}^{\mathtt{es}}_{\mathtt{w}}(\mathcal{G}, [\mathtt{e}/x]s') = \mathcal{L}_{\mathtt{leaf}}(\mathcal{G}', s'_1){\uparrow}_{\mathtt{e}}$. Then, by Lemma 18-2 and $s'_1 \approx s_1$, we have

$$\mathcal{L}^{\mathtt{es}}_{\mathtt{w}}(\mathcal{G}, [\mathtt{e}/x]s) = \mathcal{L}^{\mathtt{es}}_{\mathtt{w}}(\mathcal{G}, [\mathtt{e}/x]s') = \mathcal{L}_{\mathtt{leaf}}(\mathcal{G}', s'_1){\uparrow}_{\mathtt{e}} = \mathcal{L}_{\mathtt{leaf}}(\mathcal{G}', s_1){\uparrow}_{\mathtt{e}}. \qquad \blacktriangleleft$$

6 Conclusion

We have given a new transformation that converts an order-$(n+1)$ word grammar to an order-n tree grammar whose frontier language coincides with the word language of the input grammar, and have proved that, unlike the previous transformations [1, 12], our transformation is size-preserving, in that the size of the output grammar is polynomial in the size of an input grammar. The time complexity is also polynomial in the size of input grammar. These properties allow us to establish a link between algorithms for higher-order word and tree grammars. As a concrete example of this, we have also applied our result to the work of Parys [12] on the complexity of the diagonal problems for word and tree languages..

References

1 Kazuyuki Asada and Naoki Kobayashi. On word and frontier languages of unsafe higher-order grammars. *CoRR*, abs/1604.01595, 2016. A summary has been published in Proceedings of ICALP 2016. URL: `http://arxiv.org/abs/1604.01595`.

2 Kazuyuki Asada and Naoki Kobayashi. Pumping lemma for higher-order languages. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017*, volume 80 of *LIPIcs*, pages 97:1–97:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.ICALP.2017.97`.

3 Kazuyuki Asada and Naoki Kobayashi. Lambda-definable order-3 tree functions are well-quasi-ordered. In *38th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2018*, volume 122 of *LIPIcs*, pages 14:1–14:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. `doi:10.4230/LIPIcs.FSTTCS.2018.14`.

4 William Blum and C.-H. Luke Ong. The safe lambda calculus. *Logical Methods in Computer Science*, 5(1), 2009. `doi:10.2168/LMCS-5(1:3)2009`.

5 Lorenzo Clemente, Paweł Parys, Sylvain Salvati, and Igor Walukiewicz. The diagonal problem for higher-order recursion schemes is decidable. In *31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2016*, pages 96–105. ACM, 2016. `doi:10.1145/2933575.2934527`.

6 Werner Damm. The IO- and OI-hierarchies. *Theor. Comput. Sci.*, 20(2):95–207, 1982. `doi:10.1016/0304-3975(82)90009-3`.

7 Teodor Knapik, Damian Niwiński, Paweł Urzyczyn, and Igor Walukiewicz. Unsafe grammars and panic automata. In *32nd International Colloquium on Automata, Languages and Programming, ICALP 2005*, volume 3580 of *LNCS*, pages 1450–1461. Springer, 2005. `doi:10.1007/11523468_117`.

8 Naoki Kobayashi. Model checking higher-order programs. *Journal of the ACM*, 60(3), 2013. `doi:10.1145/2487241.2487246`.

9 Naoki Kobayashi, Ugo Dal Lago, and Charles Grellois. On the termination problem for probabilistic higher-order recursive programs. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019*, pages 1–14. IEEE, 2019. `doi:10.1109/LICS.2019.8785679`.

10 Naoki Kobayashi and C.-H. Luke Ong. Complexity of model checking recursion schemes for fragments of the modal mu-calculus. *Logical Methods in Computer Science*, 7(4), 2011. `doi:10.2168/LMCS-7(4:9)2011`.

11 C.-H. Luke Ong. On model-checking trees generated by higher-order recursion schemes. In *21th IEEE Symposium on Logic in Computer Science, LICS 2006*, pages 81–90. IEEE Computer Society, 2006. `doi:10.1109/LICS.2006.38`.

12 Paweł Parys. The Complexity of the Diagonal Problem for Recursion Schemes. In *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017*, volume 93 of *LIPIcs*, pages 45:1–45:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018. `doi:10.4230/LIPIcs.FSTTCS.2017.45`.

13 Mitchell Wand. An algebraic formulation of the Chomsky hierarchy. In *Category Theory Applied to Computation and Control*, volume 25 of *LNCS*, pages 209–213. Springer, 1974. `doi:10.1007/3-540-07142-3_84`.

A Proof of Theorem 12

Proof of Theorem 12. On Item 1, see the full version.

Next, we show Item 2: note that, by Lemma 11, we only need to show the following: if $\mathcal{G}$ is already in normal form, then $|\mathcal{G}'|_{\mathrm{cy}} \in O(A^3 \times |\mathcal{G}|_{\mathrm{cy}})$ and $|\mathcal{G}'|_{\mathrm{ch}} \in O(A^3 \times |\mathcal{G}|_{\mathrm{cy}} + |\mathcal{N}|^n)$.

Let: $r = (F\,\widetilde{y}\,\widetilde{x} \to s) \in \mathcal{R}$, $\mathcal{N}(F) = \kappa'_1 \to \cdots \to \kappa'_m \Rightarrow \mathsf{o}^k \to \mathsf{o}$, $\mathcal{K} = y_1 : \kappa'_1, \dots, y_m : \kappa'_m$, and $\mathfrak{D}_0$ be the derivation tree of $\mathcal{K}; \widetilde{x} : \mathsf{o} \vdash_{\mathcal{N}} s : \mathsf{o} \rightsquigarrow (s_0, \dots, s_{k+1})$.

Further, let $\mathcal{K}; \widetilde{x} : \mathsf{o} \vdash_{\mathcal{N}} t : \widetilde{\kappa} \Rightarrow \mathsf{o}^\ell \to \mathsf{o} \rightsquigarrow (\widetilde{t})$ be an arbitrary judgment in $\mathfrak{D}_0$, and $\mathfrak{D}$ be the sub-derivation-tree of $\mathfrak{D}_0$ for this judgment. Then we can show $\ell \leq A$ by induction on the height of $\mathfrak{D}$. Next, let h (h', resp.) be the largest number, in the depth direction, of uses of (Tr-App) ((Tr-AppG), resp.) in $\mathfrak{D}$: precisely, h (h', resp.) is defined as the least number p such that in any path of $\mathfrak{D}$, the number of (Tr-App) ((Tr-AppG), resp.) in the path is no more than p. Then for each $i \in \{0, \dots, \ell + k + 1\}$, we can show

$$|t_i| \leq 2(A+2)^h 2^{h'} |t| \tag{3}$$

by induction on t and case analysis on the last rule of $\mathfrak{D}$.

Now let us consider the case $\mathfrak{D} = \mathfrak{D}_0$ (and hence $t = s$). Since $\mathcal{G}$ is in normal form, $h, h' \leq 2$; hence by (3), $|s_i| \leq O(A^2 \times |s|)$ for each i. Also note that, from Item 1, $\mathtt{ari}(\mathcal{N}^\ddagger(F_i)) \leq O(A^2)$ for each i. Thus, for each $r' \in \mathcal{R}_r$, we have $|r'| \leq O(A^2 \times |r|)$. Since the cardinality of $\mathcal{R}_r$ is no more than $A + 1$, we have:

$$|\mathcal{G}'|_{\mathrm{cy}} = |S_1| + \sum_{r \in \mathcal{R}} \Big(\sum_{r' \in \mathcal{R}_r} |r'| \Big) \leq |S\,\mathsf{e}| + \sum_{r \in \mathcal{R}} O(A^3 \times |r|) \leq O(A^3 \times |\mathcal{G}|_{\mathrm{cy}}).$$

Next we calculate $|\mathcal{G}'|_{\mathrm{ch}}$. We define $|\kappa_1 \times \cdots \times \kappa_k| := |\kappa_1| + \cdots + |\kappa_k| + k - 1$. We can show that $|\kappa^\dagger|$, $|\kappa^{\dagger'}| \leq 3|\kappa|^{\mathtt{ord}(\kappa)}$ simultaneously by induction on κ. Hence, we have

$$|\mathcal{N}^\ddagger| \leq \sum_{F \in \mathcal{N}} |\mathcal{N}(F)^\dagger| \leq \sum_{F \in \mathcal{N}} 3|\mathcal{N}(F)|^{\mathtt{ord}(\mathcal{N}(F))} \leq 3 \sum_{F \in \mathcal{N}} |\mathcal{N}(F)|^n \leq 3|\mathcal{N}|^n$$

and thus $|\mathcal{G}'|_{\mathrm{ch}} \leq O(A^3 \times |\mathcal{G}|_{\mathrm{cy}} + |\mathcal{N}|^n)$.

Lastly, we show Item 3. Note that, by Lemma 11, we only need to show the following: if $\mathcal{G}$ is already in normal form, then the transformation given in Fig. 1 takes $O(A^2|\mathcal{G}|_{\mathrm{cy}} + |\mathcal{N}|)$ time.

We implement terms by directed acyclic graphs. The implementation $\lfloor \kappa \rfloor$ of a type κ is as follows: Let κ be of the form $\kappa_1 \to \cdots \to \kappa_k \Rightarrow \mathsf{o}^\ell \to \mathsf{o}$. If $k = 0$, $\lfloor \kappa \rfloor$ is the 2-length array $[k, \ell]$. If $k > 0$, note that $\kappa = \kappa_1 \to (\kappa_2 \to \cdots \to \kappa_k \Rightarrow \mathsf{o}^\ell \to \mathsf{o})$ and we define $\lfloor \kappa \rfloor$ as the 4-length array $[k, \ell, \lfloor \kappa_1 \rfloor, \lfloor \kappa_2 \to \cdots \to \kappa_k \Rightarrow \mathsf{o}^\ell \to \mathsf{o} \rfloor]$. Then, the two operations that respectively extract k and ℓ from a type $\kappa = \kappa_1 \to \cdots \to \kappa_k \Rightarrow \mathsf{o}^\ell \to \mathsf{o}$ and the type construction $\kappa \to \kappa'$ from κ and κ' can be performed by $O(1)$. The operation that extracts the sequence $\kappa_1, \ldots, \kappa_k$ from a type $\kappa = \kappa_1 \to \cdots \to \kappa_k \Rightarrow \mathsf{o}^\ell \to \mathsf{o}$ can be performed by $O(\mathtt{ari}(\kappa))$. During the computation of $\mathcal{G}'$ from $\mathcal{G}$, we need to extract such a sequence $\kappa_1, \ldots, \kappa_k$ just at the top of the derivation tree of (Tr-Rule) for each rewriting rule; then in the derivation tree, all the typing environments are common and can be shared.

Now let us consider the computation of a derivation tree; look at Fig. 1. To compute the base transformation rules (i.e., (Tr-Omega), (Tr-VarG), (Tr-Var), and (Tr-NT)), it takes $O(A)$ time, as the length of the resulting tuple is bounded by $O(A)$. The computation for one occurrence of (Tr-Choice) in the derivation tree takes $O(k)$ time, i.e., if the computations for s and t take p and q time, respectively, then that for $s \oplus t$ takes $p + q + O(k)$ time. Similarly, the computations for occurrences of (Tr-Const), (Tr-AppG), and (Tr-App) take $O(k)$, $O(\ell + k)$, and $O(\ell'(\ell + k))$ time, respectively. Thus, the running time for every node of the derivation tree is $O(A^2)$.

Hence, for each rule $F\,\widetilde{y}\,\widetilde{x} \to t$ of $\mathcal{G}$, the computation of $y_1{:}\kappa_1, \ldots, y_m{:}\kappa_m; x_1, \ldots, x_k{:}\mathsf{o} \vdash_{\mathcal{N}} t : \mathsf{o} \rightsquigarrow (t_0, t_1, \ldots, t_k, t_{k+1})$ takes $O(A^2|t|)$ time. Also, that of all $\widetilde{y_i}$ and $\widetilde{y_i}'$ $(1 \le i \le m)$ takes $O(A^2)$ time. Therefore, that of (Tr-Rule) runs in $O(A^2|t|)$ time. Thus the computation of $\mathcal{R}'$ takes $O(A^2|\mathcal{G}|_{\mathrm{cy}})$ time.

Also the computation of $\mathcal{N}^\ddagger$ takes $O(|\mathcal{N}|)$ time because, for $\kappa = \kappa_1 \to \cdots \to \kappa_k \Rightarrow \mathsf{o}^\ell \to \mathsf{o}$, that of $\kappa^\dagger$ (and simultaneously of $\kappa^{\dagger'}$) takes $O(|\kappa_1| + \cdots + |\kappa_k| + k + \ell)$ time. Hence, the computation of (Tr-Gram) runs in $O(A^2|\mathcal{G}|_{\mathrm{cy}} + |\mathcal{N}|)$. ◀

B Proofs for Section 5.3

Due to the space limit, we focus only on some important cases when we perform case analysis in proofs. See the full version for detailed proofs.

First we show the substitution lemma (Lemma 19), where we use the following lemma:

▶ **Lemma 22** (Weakening). *If* $\mathcal{K}; x_1, \ldots, x_k : \mathsf{o} \vdash t : \kappa \rightsquigarrow (t_0, \ldots, t_{\ell+k+1})$ *where* $\mathtt{gar}(\kappa) = \ell$*, then* $\mathcal{K}; x_0, x_1, \ldots, x_k : \mathsf{o} \vdash t : \kappa \rightsquigarrow (t_0, \ldots, t_\ell, t_{\ell+k+1}, t_{\ell+1}, \ldots, t_{\ell+k}, t_{\ell+k+1})$.

Proof. Straightforward induction on t. ◀

Proof of Lemma 19. First we show the second item of the lemma. We have:

$$\theta_0 u_{\mathtt{gar}(\kappa')+|\tilde{z}|+1} = \theta_{k+1} u_{\mathtt{gar}(\kappa')+|\tilde{z}|+1} \quad = \quad \theta_{k+1} u_0$$

where the former equation follows from Lemma 10, and the latter one can be shown by straightforward induction on u and case analysis on the last rule used for the derivation $\widetilde{y} : \widetilde{\kappa}; \widetilde{z} : \mathsf{o} \vdash u : \kappa' \rightsquigarrow (u_0, \ldots, u_{\mathtt{gar}(\kappa')+|\tilde{z}|+1})$, where we unfold the definition of θ_{k+1} only in the case of (Tr-Var).

Next we show the first item, again by induction on u and case analysis on the last rule used for the derivation $\widetilde{y} : \widetilde{\kappa}; \widetilde{z} : \mathsf{o} \vdash u : \kappa' \rightsquigarrow (u_0, \ldots, u_{\mathtt{gar}(\kappa')+|\tilde{z}|+1})$.

- Case of (Tr-Var): Let the last rule be the following:

$$\frac{\mathcal{K}(y_{i'}) = \widetilde{\kappa} \Rightarrow \mathsf{o}^{\ell_{i'}} \to \mathsf{o}}{\widetilde{y} : \widetilde{\kappa}; \widetilde{z} : \mathsf{o} \vdash y_{i'} : \widetilde{\kappa} \Rightarrow \mathsf{o}^{\ell_{i'}} \to \mathsf{o} \rightsquigarrow (y_{i',0}, y_{i',1}, \ldots, y_{i',\ell_{i'}}, (y_{i',\ell_{i'}+1})^{|\tilde{z}|+1})}$$

Now $[\widetilde{s'}/\widetilde{y}]u = [\widetilde{s'}/\widetilde{y}]y_{i'} = s'_{i'}$, and by the assumption and the weakening lemma

(Lemma 22),

$$\begin{aligned}
&\widetilde{z}, \widetilde{x} : \mathtt{o} \vdash s'_{i'} : \kappa_{i'} \rightsquigarrow \\
&\qquad (s'_{i',0},\ \dots,\ s'_{i',\ell_{i'}},\ (s'_{i',\ell_{i'}+k+1})^{|\tilde{z}|},\ s'_{i',\ell_{i'}+1},\ \dots,\ s'_{i',\ell_{i'}+k},\ s'_{i',\ell_{i'}+k+1}) \\
&\quad = (\theta_0 y_{i',0},\ \dots,\ \theta_0 y_{i',\ell_{i'}},\ (\theta_0 y_{i',\ell_{i'}+1})^{|\tilde{z}|},\ \theta_1 y_{i',0},\ \dots,\ \theta_k y_{i',0},\ \theta_0 y_{i',\ell_{i'}+1})
\end{aligned}$$

as required.

- Case of (Tr-App): Let the last rule be the following:

$$\frac{\begin{array}{c}\widetilde{y} : \widetilde{\kappa}; \widetilde{z} : \mathtt{o} \vdash s : \kappa'_0 \to \widetilde{\kappa'} \Rightarrow \mathtt{o}^\ell \to \mathtt{o} \rightsquigarrow (s_0, \dots, s_{\ell+|\tilde{z}|+1}) \\ \widetilde{y} : \widetilde{\kappa}; \widetilde{z} : \mathtt{o} \vdash t : \kappa'_0 \rightsquigarrow (t_0, \dots, t_{\ell'+|\tilde{z}|+1}) \qquad \mathtt{gar}(\kappa'_0) = \ell'\end{array}}{\begin{array}{l}\widetilde{y} : \widetilde{\kappa}; \widetilde{z} : \mathtt{o} \vdash s\,t : \widetilde{\kappa'} \Rightarrow \mathtt{o}^\ell \to \mathtt{o} \rightsquigarrow \\ \big(s_0(t_0, \dots, t_{\ell'}, t_{\ell'+|\tilde{z}|+1}),\ s_1(t_1, \dots, t_{\ell'}, t_{\ell'+|\tilde{z}|+1}),\ \dots,\ s_\ell(t_1, \dots, t_{\ell'}, t_{\ell'+|\tilde{z}|+1}), \\ \ s_{\ell+1}(t_{\ell'+1}, t_1, \dots, t_{\ell'}, t_{\ell'+|\tilde{z}|+1}),\ \dots,\ s_{\ell+|\tilde{z}|+1}(t_{\ell'+|\tilde{z}|+1}, t_1, \dots, t_{\ell'}, t_{\ell'+|\tilde{z}|+1})\big)\end{array}}$$

  By induction hypothesis and (Tr-App), we have:

$$\frac{\begin{array}{l}\widetilde{z}, \widetilde{x} : \mathtt{o} \vdash [\widetilde{s'}/\widetilde{y}]s : \kappa'_0 \to \widetilde{\kappa'} \Rightarrow \mathtt{o}^\ell \to \mathtt{o} \rightsquigarrow (\theta_0 s_0, \dots, \theta_0 s_{\ell+|\tilde{z}|}, \theta_1 s_0 \dots, \theta_k s_0, \theta_0 s_{\ell+|\tilde{z}|+1}) \\ \widetilde{z}, \widetilde{x} : \mathtt{o} \vdash [\widetilde{s'}/\widetilde{y}]t : \kappa'_0 \rightsquigarrow (\theta_0 t_0, \dots, \theta_0 t_{\ell'+|\tilde{z}|}, \theta_1 t_0 \dots, \theta_k t_0, \theta_0 t_{\ell'+|\tilde{z}|+1}) \qquad \mathtt{gar}(\kappa'_0) = \ell'\end{array}}{\begin{array}{l}\widetilde{z}, \widetilde{x} : \mathtt{o} \vdash [\widetilde{s'}/\widetilde{y}]s\,[\widetilde{s'}/\widetilde{y}]t : \widetilde{\kappa'} \Rightarrow \mathtt{o}^\ell \to \mathtt{o} \rightsquigarrow \\ \big(\theta_0 s_0(\theta_0 t_0, \dots, \theta_0 t_{\ell'}, \theta_0 t_{\ell'+|\tilde{z}|+1}), \\ \ \theta_0 s_1(\theta_0 t_1, \dots, \theta_0 t_{\ell'}, \theta_0 t_{\ell'+|\tilde{z}|+1}), \dots, \theta_0 s_\ell(\theta_0 t_1, \dots, \theta_0 t_{\ell'}, \theta_0 t_{\ell'+|\tilde{z}|+1}), \\ \ \theta_0 s_{\ell+1}(\theta_0 t_{\ell'+1}, \theta_0 t_1, \dots, \theta_0 t_{\ell'}, \theta_0 t_{\ell'+|\tilde{z}|+1}), \dots, \theta_0 s_{\ell+|\tilde{z}|}(\theta_0 t_{\ell'+|\tilde{z}|}, \theta_0 t_1, \dots, \theta_0 t_{\ell'}, \theta_0 t_{\ell'+|\tilde{z}|+1}) \\ \ \theta_1 s_0(\theta_1 t_0, \theta_0 t_1, \dots, \theta_0 t_{\ell'}, \theta_0 t_{\ell'+|\tilde{z}|+1}), \dots, \theta_k s_0(\theta_k t_0, \theta_0 t_1, \dots, \theta_0 t_{\ell'}, \theta_0 t_{\ell'+|\tilde{z}|+1}), \\ \ \theta_0 s_{\ell+|\tilde{z}|+1}(\theta_0 t_{\ell'+|\tilde{z}|+1}, \theta_0 t_1, \dots, \theta_0 t_{\ell'}, \theta_0 t_{\ell'+|\tilde{z}|+1})\big)\end{array}}$$

  Then it is enough to show

$$\begin{aligned}
&\theta_j s_0(\theta_j t_0, \theta_0 t_1, \dots, \theta_0 t_{\ell'}, \theta_0 t_{\ell'+|\tilde{z}|+1}) = \theta_j \big(s_0(t_0, t_1, \dots, t_{\ell'}, t_{\ell'+|\tilde{z}|+1})\big) \quad (j \in \{1, \dots, k\}) \\
&\text{i.e.,}\quad \theta_0 t_i = \theta_j t_i \quad (i \in \{1, \dots, \ell', \ell' + |\tilde{z}| + 1\},\ j \in \{1, \dots, k\})
\end{aligned}$$

  which follows from Lemma 10.

- Case of (Tr-AppG): Let the last rule be the following:

$$\frac{\begin{array}{c}\widetilde{y} : \widetilde{\kappa}; \widetilde{z} : \mathtt{o} \vdash s : \mathtt{o}^{\ell+1} \to \mathtt{o} \rightsquigarrow (s_0, \dots, s_{\ell+|\tilde{z}|+2}) \\ \widetilde{y} : \widetilde{\kappa}; \widetilde{z} : \mathtt{o} \vdash t : \mathtt{o} \rightsquigarrow (t_0, \dots, t_{|\tilde{z}|+1})\end{array}}{\begin{array}{l}\widetilde{y} : \widetilde{\kappa}; \widetilde{z} : \mathtt{o} \vdash s\,t : \mathtt{o}^\ell \to \mathtt{o} \rightsquigarrow \\ \big(s_0 \oplus (\mathtt{br}\ s_1\ t_0),\ s_2,\ \dots,\ s_{\ell+1},\ s_{\ell+2} \oplus (\mathtt{br}\ s_1\ t_1),\ \dots,\ s_{\ell+|\tilde{z}|+2} \oplus (\mathtt{br}\ s_1\ t_{|\tilde{z}|+1})\big)\end{array}}$$

  By induction hypothesis and (Tr-AppG), we have:

$$\frac{\begin{array}{l}\widetilde{z}, \widetilde{x} : \mathtt{o} \vdash [\widetilde{s'}/\widetilde{y}]s : \mathtt{o}^{\ell+1} \to \mathtt{o} \rightsquigarrow (\theta_0 s_0, \dots, \theta_0 s_{\ell+|\tilde{z}|+1}, \theta_1 s_0 \dots, \theta_k s_0, \theta_0 s_{\ell+|\tilde{z}|+2}) \\ \widetilde{z}, \widetilde{x} : \mathtt{o} \vdash [\widetilde{s'}/\widetilde{y}]t : \mathtt{o} \rightsquigarrow (\theta_0 t_0, \dots, \theta_0 t_{|\tilde{z}|}, \theta_1 t_0, \dots, \theta_k t_0, \theta_0 t_{|\tilde{z}|+1})\end{array}}{\begin{array}{l}\widetilde{z}, \widetilde{x} : \mathtt{o} \vdash [\widetilde{s'}/\widetilde{y}]s\,[\widetilde{s'}/\widetilde{y}]t : \mathtt{o}^\ell \to \mathtt{o} \rightsquigarrow \\ \qquad \big(\theta_0 s_0 \oplus (\mathtt{br}\ \theta_0 s_1\ \theta_0 t_0),\ \theta_0 s_2,\ \dots,\ \theta_0 s_{\ell+1}, \\ \qquad\ \theta_0 s_{\ell+2} \oplus (\mathtt{br}\ \theta_0 s_1\ \theta_0 t_1),\ \dots,\ \theta_0 s_{\ell+|\tilde{z}|+1} \oplus (\mathtt{br}\ \theta_0 s_1\ \theta_0 t_{|\tilde{z}|}), \\ \qquad\ \theta_1 s_0 \oplus (\mathtt{br}\ \theta_0 s_1\ \theta_1 t_0),\ \dots,\ \theta_k s_0 \oplus (\mathtt{br}\ \theta_0 s_1\ \theta_k t_0), \\ \qquad\ \theta_0 s_{\ell+|\tilde{z}|+2} \oplus (\mathtt{br}\ \theta_0 s_1\ \theta_0 t_{|\tilde{z}|+1})\big)\end{array}}$$

  Then it is enough to show $\theta_0 s_1 = \theta_j s_1$ for $j = 1, \dots, k$, which follows from Lemma 10. ◀

Now we prove the subject reduction lemma:

Proof of Lemma 20. We restate Lemma 20 as follows with different meta-variables s', t', s'_i, t'_i for convenience of the proof:

> If $x_1,\dots,x_k:\mathtt{o}\vdash s':\mathtt{o}\rightsquigarrow(s'_0,\dots,s'_{k+1})$ and $s'\longrightarrow_{\mathtt{es}} t'$, then there exist $t'_0,\dots,t'_{k+1}$ such that $x_1,\dots,x_k:\mathtt{o}\vdash t':\mathtt{o}\rightsquigarrow(t'_0,\dots,t'_{k+1})$ and $s'_i\approx t'_i$ for each $i\in\{1,\dots,k+1\}$.

The proof proceeds by induction on the derivation of $x_1,\dots,x_k:\mathtt{o}\vdash s':\mathtt{o}\rightsquigarrow(s'_0,\dots,s'_{k+1})$.

Let s' be of the form $E[s'']$ where s'' is the redex of $\longrightarrow_{\mathtt{es}}$. The case where $E\neq[\,]$ can be proved easily by induction hypothesis (as the transformation rules are compositional). So we consider only the case where $E=[\,]$. We perform case analysis on the redex $s''(=s')$:

- Case where $s'=F\,\widetilde{s}\,\widetilde{t}\longrightarrow_{\mathtt{es}}\{\widetilde{t}/\widetilde{z}\}[\widetilde{s}/\widetilde{y}]u$: In this case,

$$F\,\widetilde{y}\,\widetilde{z}\to u\in\mathcal{R}\qquad \mathcal{N}(F)=\widetilde{\kappa}\Rightarrow\mathtt{o}^{\ell}\to\mathtt{o}\qquad |\tilde{\kappa}|=|\tilde{y}|=|\tilde{s}|\qquad \ell=|\tilde{z}|=|\tilde{t}|,$$

and $\widetilde{z}$ do not occur in $F\,\widetilde{s}\,\widetilde{t}$.
By the derivation of $\widetilde{x}:\mathtt{o}\vdash(s'=)\;F\,\widetilde{s}\,\widetilde{t}:\mathtt{o}\rightsquigarrow(s'_0,\dots,s'_{k+1})$, we have:

$$\begin{aligned}
&\widetilde{x}:\mathtt{o}\vdash F:\widetilde{\kappa}\Rightarrow\mathtt{o}^{\ell}\to\mathtt{o}\rightsquigarrow(F_0,F_1,\dots,F_\ell,(F_0)^{k+1})\\
&\widetilde{x}:\mathtt{o}\vdash s_i:\kappa_i\rightsquigarrow(s_{i,0},\dots,s_{i,\ell_i+k+1})\qquad \mathtt{gar}(\kappa_i)=\ell_i\qquad(i=1,\dots,|\tilde{s}|) &&(4)\\
&\widetilde{x}:\mathtt{o}\vdash t_i:\mathtt{o}\rightsquigarrow(t_{i,0},\dots,t_{i,k+1})\qquad(i=1,\dots,|\tilde{t}|) &&(5)\\
&(s'_0,\dots,s'_{k+1})=(v_{\ell+1,0},\dots,v_{\ell+1,k+1})
\end{aligned}$$

where for $i=1,\dots,|\tilde{s}|$ and $j=1,\dots,|\tilde{t}|(=\ell)$, we define:

$$\begin{aligned}
&(v'_{1,0},\dots,v'_{1,\ell+k+1}):=(F_0,F_1,\dots,F_\ell,(F_0)^{k+1})\\
&(v'_{i+1,0},\dots,v'_{i+1,\ell+k+1}):=\\
&\quad\big(v'_{i,0}(s_{i,0},\dots,s_{i,\ell_i},s_{i,\ell_i+k+1}),\\
&\quad\ v'_{i,1}(s_{i,1},\dots,s_{i,\ell_i},s_{i,\ell_i+k+1}),\dots,v'_{i,\ell}(s_{i,1},\dots,s_{i,\ell_i},s_{i,\ell_i+k+1}),\\
&\quad\ v'_{i,\ell+1}(s_{i,\ell_i+1},s_{i,1},\dots,s_{i,\ell_i},s_{i,\ell_i+k+1}),\dots,v'_{i,\ell+k+1}(s_{i,\ell_i+k+1},s_{i,1},\dots,s_{i,\ell_i},s_{i,\ell_i+k+1})\big)\\
&(v_{1,0},\dots,v_{1,\ell+k+1}):=(v'_{|\tilde{s}|+1,0},\dots,v'_{|\tilde{s}|+1,\ell+k+1})\\
&(v_{j+1,0},\dots,v_{j+1,\ell+k+1-j}):=\\
&\quad\big(v_{j,0}\oplus(\mathtt{br}\;v_{j,1}\;t_{j,0}),\;v_{j,2},\dots,v_{j,\ell+1-j},\\
&\quad\ v_{j,\ell+2-j}\oplus(\mathtt{br}\;v_{j,1}\;t_{j,1}),\dots,v_{j,\ell+k+2-j}\oplus(\mathtt{br}\;v_{j,1}\;t_{j,k+1})\big).
\end{aligned}$$

Then we have

$$\begin{aligned}
&(s'_0,s'_1,\dots,s'_{k+1})=(v_{\ell+1,0},v_{\ell+1,1},\dots,v_{\ell+1,k+1})\\
&=\Big(v_{\ell,0}\oplus(\mathtt{br}\;v_{\ell,1}\;t_{\ell,0}),\;v_{\ell,2}\oplus(\mathtt{br}\;v_{\ell,1}\;t_{\ell,1}),\;\dots,\;v_{\ell,k+2}\oplus(\mathtt{br}\;v_{\ell,1}\;t_{\ell,k+1})\Big)\\
&=\Big(\big(v_{\ell-1,0}\oplus(\mathtt{br}\;v_{\ell-1,1}\;t_{\ell-1,0})\big)\oplus(\mathtt{br}\;v_{\ell-1,2}\;t_{\ell,0}),\\
&\qquad\big(v_{\ell-1,3}\oplus(\mathtt{br}\;v_{\ell-1,1}\;t_{\ell-1,1})\big)\oplus(\mathtt{br}\;v_{\ell-1,2}\;t_{\ell,1}),\;\dots,\\
&\qquad\big(v_{\ell-1,k+3}\oplus(\mathtt{br}\;v_{\ell-1,1}\;t_{\ell-1,k+1})\big)\oplus(\mathtt{br}\;v_{\ell-1,2}\;t_{\ell,k+1})\Big) &&(6)\\
&=\dots\\
&=\Big(v_{1,0}\oplus(\mathtt{br}\;v_{1,1}\;t_{1,0})\oplus\cdots\oplus(\mathtt{br}\;v_{1,\ell}\;t_{\ell,0}),\\
&\qquad v_{1,\ell+1}\oplus(\mathtt{br}\;v_{1,1}\;t_{1,1})\oplus\cdots\oplus(\mathtt{br}\;v_{1,\ell}\;t_{\ell,1}),\;\dots,\\
&\qquad v_{1,\ell+k+1}\oplus(\mathtt{br}\;v_{1,1}\;t_{1,k+1})\oplus\cdots\oplus(\mathtt{br}\;v_{1,\ell}\;t_{\ell,k+1})\Big)
\end{aligned}$$

and

$$\begin{aligned}
v_{1,0} &= F_0(s_{1,0},\dots,s_{1,\ell_1},s_{1,\ell_1+k+1})\cdots(s_{|\tilde{s}|,0},\dots,s_{|\tilde{s}|,\ell_{|\tilde{s}|}},s_{|\tilde{s}|,\ell_{|\tilde{s}|}+k+1})\\
v_{1,j} &= F_j(s_{1,1},\dots,s_{1,\ell_1},s_{1,\ell_1+k+1})\cdots(s_{|\tilde{s}|,1},\dots,s_{|\tilde{s}|,\ell_{|\tilde{s}|}},s_{|\tilde{s}|,\ell_{|\tilde{s}|}+k+1}) \qquad (j=1,\dots,\ell)\\
v_{1,\ell+j} &= F_0(s_{1,\ell_1+j},s_{1,1},\dots,s_{1,\ell_1},s_{1,\ell_1+k+1})\cdots(s_{|\tilde{s}|,\ell_{|\tilde{s}|}+j},s_{|\tilde{s}|,1},\dots,s_{|\tilde{s}|,\ell_{|\tilde{s}|}},s_{|\tilde{s}|,\ell_{|\tilde{s}|}+k+1})\\
&\qquad\qquad (j=1,\dots,k+1).
\end{aligned}$$

Next let us consider $t' = \{\widetilde{t}/\widetilde{z}\}[\widetilde{s}/\widetilde{y}]u$. For some $\widetilde{u}$, we have

$$\widetilde{y} : \widetilde{\kappa}; \widetilde{z} : \mathsf{o} \vdash u : \mathsf{o} \rightsquigarrow (u_0,\dots,u_{\ell+1}). \tag{7}$$

By (7), (4), and the substitution lemma (Lemma 19), we have

$$\widetilde{z},\widetilde{x} : \mathsf{o} \vdash [\widetilde{s}/\widetilde{y}]u : \mathsf{o} \rightsquigarrow (\theta_0 u_0, \theta_0 u_1, \dots, \theta_0 u_\ell, \theta_1 u_0, \dots, \theta_k u_0, \theta_0 u_{\ell+1}) \tag{8}$$

$$\theta_0 u_{\ell+1} \;=\; \theta_{k+1} u_0 \tag{9}$$

where

$$\begin{aligned}
\theta_j &= \theta_{1,j}\cdots\theta_{|\tilde{\kappa}|,j} \qquad (j=0,\dots,k)\\
\theta_{i,0} &= [s_{i,0}/y_{i,0},\dots,s_{i,\ell_i}/y_{i,\ell_i},s_{i,\ell_i+k+1}/y_{i,\ell_i+1}] \qquad (i=1,\dots,|\tilde{\kappa}|)\\
\theta_{i,j} &= [s_{i,\ell_i+j}/y_{i,0},\dots,s_{i,\ell_i}/y_{i,\ell_i},s_{i,\ell_i+k+1}/y_{i,\ell_i+1}] \quad (i=1,\dots,|\tilde{\kappa}|,\ j=1,\dots,k+1).
\end{aligned}$$

Further, by (5), (8), and (Tr-Sub), we have

$$\widetilde{x} : \mathsf{o} \vdash (t' =)\ \{\widetilde{t}/\widetilde{z}\}[\widetilde{s}/\widetilde{y}]u : \mathsf{o} \rightsquigarrow (t'_0,\dots,t'_{k+1})$$

where

$$\begin{aligned}
&(t'_0,t'_1,\dots,t'_k,t'_{k+1}) :=\\
&\quad \big((\theta_0 u_0) \oplus \textstyle\bigoplus_{i=1}^{\ell}(\mathtt{br}\ (\theta_0 u_i)\ t_{i,0}),\\
&\quad\ (\theta_1 u_0) \oplus \textstyle\bigoplus_{i=1}^{\ell}(\mathtt{br}\ (\theta_0 u_i)\ t_{i,1}),\ \dots,\ (\theta_k u_0) \oplus \textstyle\bigoplus_{i=1}^{\ell}(\mathtt{br}\ (\theta_0 u_i)\ t_{i,k}),\\
&\quad\ (\theta_0 u_{\ell+1}) \oplus \textstyle\bigoplus_{i=1}^{\ell}(\mathtt{br}\ (\theta_0 u_i)\ t_{i,k+1})\big).
\end{aligned} \tag{10}$$

Now, by (Tr-Rule), the transformed grammar has the following rules

$$F_0\,\widetilde{y_1}\ \cdots\ \widetilde{y_{|\tilde{y}|}} \to u_0, \qquad F_j\,\widetilde{y_1}'\ \cdots\ \widetilde{y_{|\tilde{y}|}}' \to u_j \quad (j=1,\dots,\ell)$$

where

$$\widetilde{y_i} = (y_{i,0},\dots,y_{i,\mathtt{gar}(\kappa_i)+1}) \qquad \widetilde{y_i}' = (y_{i,1},\dots,y_{i,\mathtt{gar}(\kappa_i)+1}) \qquad (i=1,\dots,|\widetilde{y}|).$$

Then,

$$v_{1,0} \longrightarrow^* \theta_0 u_0 \tag{11}$$

$$v_{1,j} \longrightarrow^* \theta_0 u_j \qquad (j=1,\dots,\ell) \tag{12}$$

$$v_{1,\ell+j} \longrightarrow^* \theta_j u_0 \qquad (j=1,\dots,k) \tag{13}$$

$$v_{1,\ell+k+1} \longrightarrow^* \theta_{k+1} u_0 \;=\; \theta_0 u_{\ell+1} \tag{14}$$

where: the equation in (14) is just (9); and for (12) note that $y_{1,0}, \ldots, y_{|\tilde{\kappa}|,0}$ do not occur in u_j by Lemma 10. Then, we have

$$\begin{aligned}
&(s'_0, s'_1, \ldots, s'_k, s'_{k+1}) \\
&= \big(v_{1,0} \oplus \textstyle\bigoplus_{i=1}^{\ell}(\texttt{br}\ v_{1,i}\ t_{i,0}), && (\because (6)) \\
&\quad v_{1,\ell+1} \oplus \textstyle\bigoplus_{i=1}^{\ell}(\texttt{br}\ v_{1,i}\ t_{i,1}),\ \ldots,\ v_{1,\ell+k} \oplus \textstyle\bigoplus_{i=1}^{\ell}(\texttt{br}\ v_{1,i}\ t_{i,k}) \\
&\quad v_{1,\ell+k+1} \oplus \textstyle\bigoplus_{i=1}^{\ell}(\texttt{br}\ v_{1,i}\ t_{i,k+1})\big) \\
&\approx \big((\theta_0 u_0) \oplus \textstyle\bigoplus_{i=1}^{\ell}(\texttt{br}\ (\theta_0 u_i)\ t_{i,0}), && (\because (11)\text{-}(14)) \\
&\quad (\theta_1 u_0) \oplus \textstyle\bigoplus_{i=1}^{\ell}(\texttt{br}\ (\theta_0 u_i)\ t_{i,1}),\ \ldots,\ (\theta_k u_0) \oplus \textstyle\bigoplus_{i=1}^{\ell}(\texttt{br}\ (\theta_0 u_i)\ t_{i,k}), \\
&\quad (\theta_0 u_{\ell+1}) \oplus \textstyle\bigoplus_{i=1}^{\ell}(\texttt{br}\ (\theta_0 u_i)\ t_{i,k+1})\big) \\
&= (t'_0, t'_1, \ldots, t'_k, t'_{k+1}) && (\because (10))
\end{aligned}$$

as required. ◀

A Syntax for Mutual Inductive Families

Ambrus Kaposi
Eötvös Loránd University, Budapest, Hungary
akaposi@inf.elte.hu

Jakob von Raumer
University of Nottingham, UK
jakob@von-raumer.de

Abstract

Inductive families of types are a feature of most languages based on dependent types. They are usually described either by syntactic schemes or by encodings of strictly positive functors such as combinator languages or containers. The former approaches are informal and give only external signatures, the latter approaches suffer from encoding overheads and do not directly represent mutual types.

In this paper we propose a direct method for describing signatures for mutual inductive families using a domain-specific type theory. A signature is a context (roughly speaking, a list of types) in this small type theory. Algebras, displayed algebras and sections are defined by models of this type theory: the standard model, the logical predicate and a logical relation interpretation, respectively. We reduce the existence of initial algebras for these signatures to the existence of the syntax of our domain-specific type theory. As this theory is very simple, its normal syntax can be encoded using indexed W-types. To the best of our knowledge, this is the first formalisation of the folklore fact that mutual inductive types can be reduced to indexed W-types.

The contents of this paper were formalised in the proof assistant Agda.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases type theory, inductive types, mutual inductive types, W-types, Agda

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.23

Supplementary Material All of the results were formalised in the proof assistant Agda, the source code is available online at `https://bitbucket.org/javra/inductive-families/src/master/agda`.

Funding *Ambrus Kaposi*: I acknowledge the support of the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme funding scheme, Project no. ED_18-1-2019-0030 (Application-specific highly reliable IT solutions), and of the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunication).
Jakob von Raumer: I acknowledge the support of COST Action EUTypes CA15123.

1 Introduction

Programming languages based on type theory rely heavily on easy, flexible and sound ways to define new data types. Usually, type theories allow for the definition of *inductive types*, which are defined by giving a list of constructors which generate the elements of the type. One prime example for such an inductive type is the type of natural numbers $\mathbb{N} : \mathsf{Set}$ which is generated by the zero constructor $0 : \mathbb{N}$ and the successor function $\mathsf{S} : \mathbb{N} \to \mathbb{N}$. Besides these plain inductive types, dependent type theories often make use of *inductive families of types* (also called indexed inductive types) where, instead of just a type we define a type family over a previously defined type. This enables us for example to define the type of vectors of a type A as a family $\mathsf{Vec} : \mathbb{N} \to \mathsf{Set}$, by a constructor for the empty vector $\mathsf{nil} : \mathsf{Vec}\,0$ and $\mathsf{cons} : A \to (n : \mathbb{N}) \to \mathsf{Vec}\,n \to \mathsf{Vec}\,(\mathsf{S}\,n)$. Besides inductive families, another recurring need

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 23; pp. 23:1–23:21
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

is the one for *mutual definitions*: Often, we want to define more than one inductive type simultaneously with constructors referring to any of these types. For example we might want to obtain the predicates of a natural number being even and odd in reference to each other by defining

$$\begin{aligned}&\mathsf{isEven} : \mathbb{N} \to \mathsf{Set},\\ &\mathsf{isOdd}\ \ : \mathbb{N} \to \mathsf{Set}\end{aligned}$$

by constructors

$$\begin{aligned}&\mathsf{even0} : \mathsf{isEven}\, 0,\\ &\mathsf{evenS} : (n : \mathbb{N}) \to \mathsf{isOdd}\, n \to \mathsf{isEven}\, (\mathsf{S}\, n), \text{ and}\\ &\mathsf{oddS}\ \ : (n : \mathbb{N}) \to \mathsf{isEven}\, n \to \mathsf{isOdd}\, (\mathsf{S}\, n).\end{aligned}$$

Syntaxes of programming languages usually also consist of mutually given inductive types, such as expressions (indexed by their types), commands, blocks, etc. We call these types *mutual inductive families.*

There is a folklore trick to reduce mutual inductive families to inductive families. For example, isEven–isOdd can be simulated by a single family indexed over an extra boolean which says which sort is meant: $\mathsf{isEven?} : \mathsf{Bool} \to \mathbb{N} \to \mathsf{Set}$. Now isEven is simulated by $\mathsf{isEven?}\,\mathsf{true}$ and isOdd by $\mathsf{isEven?}\,\mathsf{false}$. To show that this technique works for every mutual inductive family, we first have to provide a general definition for mutual inductive families.

The description of inductive families was Peter Dybjer's external scheme [17]. He extended type theory with new deriviation rules for inductive families and their constructors, elimination principles and computation rules. His approach does not allow internal manipulation of signatures and it can only be formalised as an extension of a pre-existing syntax of type theory, however it covers mutual inductive families as well.

Another popular method is the functorial approach: strictly positive functors are encoded either using a combinator calculus [15] or using indexed containers [4]. An algebra of such a functor F is given by a family X and a morphism $F\,X \to X$, the initial algebra is given by the least fixpoint of the functor. The codes for the functors can be expressed internally allowing generic programming with signatures. A powerful application of this method is the automatic derivation of substitution laws for syntaxes with binders [2]. The drawback of the functorial approach is its encoding overhead – mutual types have to be transformed to indexed types, separate constructors have to be given as single families and in uncurried forms. The indexed container encoding, while being very concise, also relies on function extensionality. E.g. without assuming function extensionality, there are many different, unequal constructors for zero [6, Section 2.1]. These constructors cannot be made definitionally equal even in the presence of function extensionality – they contain definitionally unequal $\bot \to \mathbb{N}$ functions.

In this paper we aim to formalise mutual inductive families in a direct way, in the spirit of the original Dybjer definition. Drawing inspiration from the syntax of signatures for quotient inductive-inductive types (QIITs) and higher inductive-inductive types (HIITs) given by Kaposi, Kovács and Altenkirch [27, 26], we define signatures for mutual inductive families using the syntax of a small type theory tailor made for this purpose. We call this type theory the *theory of signatures*. A signature is a context in the theory of signatures, that is, roughly, a list of types. For example, the signature of natural numbers is given by the context $(N : \mathsf{Set},\ \mathit{0} : N,\ S : N \to N)$, where N, $\mathit{0}$ and S are simply variable names. The rules for the theory of signatures enforce that we can only write strictly positive constructors. This syntax allows us to write down the definition of an inductive family in the same way as it would look like in a theorem prover like Agda [30], Lean [16], or Coq [9].

The syntax for the theory of signatures can be internalised in type theory but it can also be seen as an external type theory in which one can describe signatures. We will present our syntax internally to a type theory, define its semantics and show that all mutual inductive families can be reduced to indexed W-types. All of the results were formalised in the proof assistant Agda, the source code is available online[1].

Contributions and structure

This paper contributes the following to the literature on inductive types.

- A syntax for mutual inductive families in which signatures can be defined in a direct way, simply by listing the types of sorts and constructors (Section 2). This syntax can be encoded by indexed W-types.
- Semantics for each signature: notions of algebras, displayed algebras and sections (Section 3). These explain what it means that an inductive type specified by a signature exists. The computation rules are specified as propositional (rather than definitional) equalities.
- An extension of the theory of signatures to a full substitution calculus (Section 4.1).
- A proof that each mutual inductive type can be constructed from the theory of signatures (Section 4.2), and as a by-product, a proof that mutual inductive families can be reduced to W-types. The reduction only justifies propositional computation rules.

Related work

As mentioned earlier, schemes for inductive types can be categorised into (1) external schemes, (2) internal combinatorial or (3) internal semantic schemes. Our approach is between (1) and (2). It compares to (2) as lambda-calculus compares to combinatory logic. To illustrate the difference, we list the signature for natural numbers in all approaches. (1) Dybjer [17] defines natural numbers by the formation rule $N : set$ and introduction rules $0 : N$ and $s : (u : N)N$. Our syntax will encode the same information by a sort context $(\cdot \rhd \mathsf{U})$ and a point context $\cdot \rhd \mathsf{El}\,(\mathsf{var}\,\mathsf{vz}) \rhd \mathsf{var}\,\mathsf{vz} \Rightarrow_{\mathsf{p}} \mathsf{El}\,(\mathsf{var}\,\mathsf{vz})$. The difference in encoding is that we use de Bruijn indices instead of variable names and El when decoding an index to a type (but not on the left hand side of the arrow $\Rightarrow_{\mathsf{p}}$, see later). (2) In [15, 2], natural numbers are specified by ‘$\sigma\,\mathsf{Bool}\,(\lambda b\,.\,\mathsf{if}\,b\,\mathsf{then}$ ‘$\blacksquare\,\mathsf{tt}\,\mathsf{else}$ ‘$\mathsf{X}\,\mathsf{tt}\,($‘$\blacksquare\,\mathsf{tt}))$. The two constructors are encoded as one constructor with a Bool parameter. When this is true (zero case), there are no more parameters (denoted by ‘$\blacksquare$), when it is false (successor) there is one recursive argument signified by ‘X. The tts are necessary because the type of natural numbers does not have indices. (3) The container representation [4] of natural numbers is given by the type Bool (expressing that there are two constructors) and a family of sets over Bool, $\lambda b\,.\,\mathsf{if}\,b\,\mathsf{then}\,\bot\,\mathsf{else}\,\top$ expressing that the first constructor has zero and the second constructor has one recursive argument. We list the related work categorised as above.

(1) External syntactic schemes similar to the Dybjer's were used to describe mutual inductive families of Coq on paper [32] and inside Coq [8], inductive-recursive types [18], subsets of higher inductive types [11, 19, 14], and inductive and coinductive types [10].

(2) Internal combinatorial schemes are defined by Benke, Dybjer and Jansson [12] for different classes of inductive types for the purpose of generic programming. Their signatures can be seen as uncurried versions of our signatures with some encoding overhead. In addition to our signatures, they separate the cases of parametrised and indexed definitions, while

[1] `https://bitbucket.org/javra/inductive-families/src/master/agda`

we only have indexed ones and they also cover infinitary constructors. They also feature iterated signatures, while we only model these using the function space with metatheoretic domain. [20, 21] use combinator languages to axiomatise inductive-recursive types and indexed inductive-recursive types, respectively. The same technique was used to describe inductive-inductive types [29] and inductive families [15, 2].

(3) Internal semantic schemes: Containers for describing signatures of W-types were introduced in [1] and extended to indexed W-types (potentially infinitary inductive families) in [4] and QW-types (allowing equality constructors) [22]. In fact, indexed W-types were introduced as "tree sets" much earlier, by Petersson and Synek [33]. The more semantic treatments of higher inductive types [28] and quotient inductive-inductive types [3] don't provide schemes for the allowed constructors.

The direct inspiration of our work are the domain-specific type theories for describing higher inductive-inductive [25] and quotient inductive-inductive signatures [27]. The latter also derives all QIITs from a theory of QIIT signatures. Note that the analogous result in our paper is not a consequence of the result of [27]. We use a similar proof, however we have a much weaker assumption: we derive all mutual inductive families from the theory of *mutual inductive family signatures*, instead of the theory of QIIT signatures. Moreover, we also show how to reduce our weaker theory of signatures to indexed W-types. Such a reduction is not done in [27], and is probably not possible for the theory of QIIT signatures.

Notation and metatheory

Throughout the paper, we will assume that we are given a type theory with a hierarchy of universes Set_i (we omit the indices for readability), Π-types, Σ-types, unit type $\mathbf{1}$, propositional equality $- = -$, and indexed W-types [4] (see Appendix A). We write Σ-types as $(x : A) \times B$ and Π-types as $(x : A) \to B$ where B might refer to x. We write implicit arguments in curly braces $\{x : A\} \to B$ or simply omit them. Definitional equality is denoted $- \equiv -$. We presume that the type theory is extensional, that is, given a term $t : u = v$, we have $u \equiv v$. It is expected that all definitions could be translated to intensional type theory with the necessary coercions and transports following Hofmann's translation [23, 31, 35]. In the Agda formalisation we use explicit transports and rewrite rules occasionally as a limited version of equality reflection. We also assume function extensionality, this is necessary to handle the Π-types in our syntax with metatheoretic domain ($\hat{\Pi}_{\mathsf{s}}$, $\hat{\Pi}_{\mathsf{p}}$). In the formalisation we do not use uniqueness of identity proofs and we conjecture that our usages of equality reflection do not imply it.

2 Signatures for Mutual Inductive Families

In this section we define a syntax for a small type theory for describing signatures of mutual inductive families. We call this the *theory of signatures.* The idea is that a signature is a context in this theory, starting with the declaration of the sorts as functions into the universe U, then listing the constructors for the sorts in any order. We call these point constructors following [34]. This theory is much simpler than the full syntax of dependent type theory. For example, there are no interdependencies between sorts, neither between point constructors, and no references from sorts to point constructors. We reflect these properties in our syntax by separating sort contexts $\mathsf{Con}_{\mathsf{s}}$ from point contexts $\mathsf{Con}_{\mathsf{p}}$, and the latter will be indexed over the former. We define an intrinsically typed syntax (in the style of [7, 5]), that is, we don't have preterms or typing relations, only well-scoped, well-typed terms, well-formed contexts and types.

▶ **Definition 1** (The Theory of Signatures). *The syntax is defined inductively by the following 6 sorts and 13 constructors. These 6 types can be encoded as indexed W-types, the detailed construction of which is provided in Appendix A.*

$$\begin{array}{lll@{\qquad}lll}
\mathsf{Ty}_\mathsf{s} & : & \mathsf{Set} & \mathsf{Ty}_\mathsf{p} & : & \mathsf{Con}_\mathsf{s} \to \mathsf{Set} \\
\mathsf{U} & : & \mathsf{Ty}_\mathsf{s} & \mathsf{El} & : & \mathsf{Tm}_\mathsf{s}\,\mathit{\Gamma}_s\,\mathsf{U} \to \mathsf{Ty}_\mathsf{p}\,\mathit{\Gamma}_s \\
\hat{\Pi}_\mathsf{s} & : & (T : \mathsf{Set}) \to (T \to \mathsf{Ty}_\mathsf{s}) \to \mathsf{Ty}_\mathsf{s} & \hat{\Pi}_\mathsf{p} & : & (T : \mathsf{Set}) \to (T \to \mathsf{Ty}_\mathsf{p}\,\mathit{\Gamma}_s) \to \mathsf{Ty}_\mathsf{p}\,\mathit{\Gamma}_s \\
\mathsf{Con}_\mathsf{s} & : & \mathsf{Set} & - \Rightarrow_\mathsf{p} - & : & \mathsf{Tm}_\mathsf{s}\,\mathit{\Gamma}_s\,\mathsf{U} \to \mathsf{Ty}_\mathsf{p}\,\mathit{\Gamma}_s \to \mathsf{Ty}_\mathsf{p}\,\mathit{\Gamma}_s \\
\cdot & : & \mathsf{Con}_\mathsf{s} & \mathsf{Con}_\mathsf{p} & : & \mathsf{Con}_\mathsf{s} \to \mathsf{Set} \\
- \rhd - & : & \mathsf{Con}_\mathsf{s} \to \mathsf{Ty}_\mathsf{s} \to \mathsf{Con}_\mathsf{s} & \cdot & : & \mathsf{Con}_\mathsf{p}\,\mathit{\Gamma}_s \\
\mathsf{Var}_\mathsf{s} & : & \mathsf{Con}_\mathsf{s} \to \mathsf{Ty}_\mathsf{s} \to \mathsf{Set} & - \rhd - & : & \mathsf{Con}_\mathsf{p}\,\mathit{\Gamma}_s \to \mathsf{Ty}_\mathsf{p}\,\mathit{\Gamma}_s \to \mathsf{Con}_\mathsf{p}\,\mathit{\Gamma}_s \\
\mathsf{vz} & : & \mathsf{Var}_\mathsf{s}\,(\mathit{\Gamma}_s \rhd A_s)\,A_s & & & \\
\mathsf{vs} & : & \mathsf{Var}_\mathsf{s}\,\mathit{\Gamma}_s\,A_s \to \mathsf{Var}_\mathsf{s}\,(\mathit{\Gamma}_s \rhd B_s)\,A_s & & & \\
\mathsf{Tm}_\mathsf{s} & : & \mathsf{Con}_\mathsf{s} \to \mathsf{Ty}_\mathsf{s} \to \mathsf{Set} & & & \\
\mathsf{var} & : & \mathsf{Var}_\mathsf{s}\,\mathit{\Gamma}_s\,A_s \to \mathsf{Tm}_\mathsf{s}\,\mathit{\Gamma}_s\,A_s & & & \\
- @ - & : & \mathsf{Tm}_\mathsf{s}\,\mathit{\Gamma}_s\,(\hat{\Pi}_\mathsf{s}\,T\,A_s) \to (\tau : T) \to \mathsf{Tm}_\mathsf{s}\,\mathit{\Gamma}_s\,(A_s\,\tau) & & &
\end{array}$$

A *sort type* Ty_s is either a *universe* U or is given by an indexing type T and a sort type for each element of T. The latter can be seen as a function space where the domain is metatheoretic, hence the notation $\hat{\Pi}_\mathsf{s}$. We use the abbreviation $T \mathrel{\hat{\Rightarrow}_\mathsf{s}} A_s$ for $\hat{\Pi}_\mathsf{s}\,T\,(\lambda\tau.A_s)$ when $A_s : \mathsf{Ty}_\mathsf{s}$. A *sort context* Con_s is simply a snoc-list of sort types (empty context $\cdot$ and context extension $- \rhd -$). In order to refer to sorts we introduce typed de Bruijn variables Var_s with zero vz and successor vs constructors. Just as variables, *sort terms* Tm_s are indexed by a sort context and a sort type. Each variable is a term (var) and we have application $- @ -$ for the function space $\hat{\Pi}_\mathsf{s}$. Note that $t : \mathsf{Tm}_\mathsf{s}\,\mathit{\Gamma}_s\,A_s$ carries similar information to $\mathit{\Gamma}_s \vdash t : A_s$ in a presentation of a syntax with preterms and typing relations, but we do not have preterms, only well-typed terms.

Point constructors are represented by *point types* Ty_p over a given sort context. The type formers are the element type for the universe U, a function type with metatheoretic domain $\hat{\Pi}_\mathsf{p}$ and a non-dependent function type $- \Rightarrow_\mathsf{p} -$ where the domain is in U. The former function type allows adding parameters to constructors, the latter allows adding recursive arguments. We use the abbreviation $T \mathrel{\hat{\Rightarrow}_\mathsf{p}} A_p$ for $\hat{\Pi}_\mathsf{p}\,T\,(\lambda\tau.A_p)$ when $A_p : \mathsf{Ty}_\mathsf{p}$. A *point context* over a given sort context is a snoc-list of point types all in the same sort context.

▶ **Example 2** (Natural Numbers, Vectors, Parity). A common example for inductive types, the natural numbers, with one constructor for zero and one for the successor function, are represented by the following sort and point contexts. On the right hand side, we write the same with an informal notation using variable names.

$$\begin{array}{lll@{\qquad\qquad}l}
N_s & :\equiv & (\cdot \rhd \mathsf{U}) & (N : \mathsf{U}) \\
N & :\equiv & \big(\cdot \rhd \mathsf{El}\,(\mathsf{var}\,\mathsf{vz}) \rhd \mathsf{var}\,\mathsf{vz} \Rightarrow_\mathsf{p} \mathsf{El}\,(\mathsf{var}\,\mathsf{vz})\big) & (\mathit{zero} : N, \mathit{suc} : N \to N)
\end{array}$$

The only sort is referred to by $\mathsf{var}\,\mathsf{vz}$. As shown by the constructor for successor, on the left hand side of the arrow $\Rightarrow_\mathsf{p}$ we have to write a sort term of type U, and not a point type. This makes sure that all constructors are *strictly positive*, as the only ways to form sort terms are variables and applications.

An example of a real indexed type is the type family of vectors of a fixed type $A : \mathsf{Set}$. We also assume that we have natural numbers in our metatheory.

$$\begin{array}{ll}
V_s :\equiv (\cdot \rhd \ \mathbb{N} \mathrel{\hat{\Rightarrow}_{\mathsf{s}}} \mathsf{U}) & (\mathit{Vec} \ : \mathbb{N} \to \mathsf{U}) \\
V \ \ :\equiv \big(\cdot \rhd \mathsf{El}\,(\mathsf{var}\,\mathsf{vz} \mathbin{@} 0) \rhd & (\mathit{nil} \ \ \ : \mathit{Vec}\,0, \\
\qquad\quad A \mathrel{\hat{\Rightarrow}_{\mathsf{p}}} \hat{\Pi}_{\mathsf{p}}\,\mathbb{N}\,\big(\lambda n.\mathsf{var}\,\mathsf{vz} \mathbin{@} n \Rightarrow_{\mathsf{p}} & \ \mathit{cons} : A \to (n : \mathbb{N}) \to \mathit{Vec}\,n \to \\
\qquad\qquad\qquad\qquad \mathsf{El}\,(\mathsf{var}\,\mathsf{vz} \mathbin{@} (n+1))\big)\big) & \qquad\qquad\qquad\qquad \mathit{Vec}\,(n+1))
\end{array}$$

As our sort has a function type, whenever we have to refer to it in constructors, we have to use the application @ to specify the natural number index. In the *cons* constructor, we use both kinds of function types: the first two function types are $\hat{\Pi}_{\mathsf{p}}$ as they refer to the parameters of type A and $\mathbb{N}$. The last function type is $\Rightarrow_{\mathsf{p}}$ as it refers to a recursive argument.

We revisit the parity example from the introduction.

$$\begin{array}{ll}
P_s :\equiv (\cdot \rhd \ \mathbb{N} \mathrel{\hat{\Rightarrow}_{\mathsf{s}}} \mathsf{U} \rhd \mathbb{N} \mathrel{\hat{\Rightarrow}_{\mathsf{s}}} \mathsf{U}) & (\mathit{isEven} : \mathbb{N} \to \mathsf{U}, \mathit{isOdd} : \mathbb{N} \to \mathsf{U}) \\
P \ \ :\equiv \big(\cdot \rhd \mathsf{El}\,(\mathsf{var}\,(\mathsf{vs}\,\mathsf{vz}) \mathbin{@} 0) \rhd & (\mathit{even0} \ \ : \mathit{isEven}\,0, \\
\qquad\quad \hat{\Pi}_{\mathsf{p}}\,\mathbb{N}\,\big(\lambda n.\mathsf{var}\,\mathsf{vz} \mathbin{@} n \Rightarrow_{\mathsf{p}} & \ \mathit{evenS} \ \ : (n : \mathbb{N}) \to \mathit{isOdd}\,n \to \\
\qquad\qquad\quad \mathsf{El}\,(\mathsf{var}\,(\mathsf{vs}\,\mathsf{vz}) \mathbin{@} (n+1))\big) \rhd & \qquad\qquad\qquad\quad \mathit{isEven}\,(n+1), \\
\qquad\quad \hat{\Pi}_{\mathsf{p}}\,\mathbb{N}\,\big(\lambda n.\mathsf{var}\,(\mathsf{vs}\,\mathsf{vz}) \mathbin{@} n \Rightarrow_{\mathsf{p}} & \ \mathit{oddS} \ \ \ : (n : \mathbb{N}) \to \mathit{isEven}\,n \to \\
\qquad\qquad\quad \mathsf{El}\,(\mathsf{var}\,\mathsf{vz} \mathbin{@} (n+1))\big)\big) & \qquad\qquad\qquad\quad \mathit{isOdd}\,(n+1))
\end{array}$$

The sort context P_s has length two, we refer to the isEven sort by $\mathsf{var}\,(\mathsf{vs}\,\mathsf{vz})$, to the isOdd sort by $\mathsf{var}\,\mathsf{vz}$.

3 Algebras, Displayed Algebras, and Sections

In this section we provide semantics for the theory of signatures (Definition 1). A signature is given by a sort context Γ_s and a point context $\Gamma : \mathsf{Con}_{\mathsf{p}}\,\Gamma_s$. For each such signature, we will obtain notions of *algebras*, *displayed algebras* and *sections* of displayed algebras. From the signature for natural numbers given in Example 2 we will derive that a natural number algebra is an element of

$$(N : \mathsf{Set}) \times N \times (N \to N),$$

a displayed natural number algebra over an algebra (N, z, s) is an element of

$$(P : N \to \mathsf{Set}) \times P\,z \times ((n : N) \to P\,n \to P\,(s\,n)),$$

and a section of a displayed algebra (P, w, h) over (N, z, s) is an element of

$$(f : (n : N) \to P\,n) \times (f\,z = w) \times \big((n : N) \to f\,(s\,n) = h\,n\,(f\,n)\big).$$

The constructors of the inductive type will be elements of the algebra, the arguments of the eliminator (sometimes called motives and methods) form a displayed algebra over the constructors, while the eliminator itself is a section. The equalities in the section are the computation rules (β rules) for the eliminator.

More formally, we will define operations $\_^{\mathsf{A}}$, $\_^{\mathsf{D}}$ and $\_^{\mathsf{S}}$ for computing algebras, displayed algebras and sections. As sort and point contexts are separate, we have to define them separately for both.

$$\begin{array}{lll}
{\Gamma_s}^{\mathsf{A}} : \mathsf{Set} & {\Gamma_s}^{\mathsf{D}} : {\Gamma_s}^{\mathsf{A}} \to \mathsf{Set} & {\Gamma_s}^{\mathsf{S}} : (\gamma_s : {\Gamma_s}^{\mathsf{A}}) \to {\Gamma_s}^{\mathsf{D}}\,\gamma_s \to \mathsf{Set} \\
\Gamma^{\mathsf{A}} \;: {\Gamma_s}^{\mathsf{A}} \to \mathsf{Set} & \Gamma^{\mathsf{D}} \;: {\Gamma_s}^{\mathsf{D}}\,\gamma_s \to \Gamma^{\mathsf{A}}\,\gamma_s \to \mathsf{Set} & \Gamma^{\mathsf{S}} \;: {\Gamma_s}^{\mathsf{S}}\,\gamma_s\,\gamma_s^d \to (\gamma : \Gamma^{\mathsf{A}}\,\gamma_s) \to \\
 & & \quad \Gamma^{\mathsf{D}}\,\gamma_s^d\,\gamma \to \mathsf{Set}
\end{array}$$

Putting them together, we get algebras as $(\gamma_s : {\Gamma_s}^{\mathsf{A}}) \times \Gamma^{\mathsf{A}}\,\gamma_s$, displayed algebras over a (γ_s, γ) by $(\gamma_s^d : {\Gamma_s}^{\mathsf{D}}\,\gamma_s) \times \Gamma^{\mathsf{D}}\,\gamma_s^d\,\gamma$, and sections of (γ_s^d, γ^d) by $(\gamma_s^s : {\Gamma_s}^{\mathsf{S}}\,\gamma_s\,\gamma_s^d) \times \Gamma^{\mathsf{S}}\,\gamma_s^s\,\gamma\,\gamma^d$.

The algebra operator corresponds to building the standard model (set model, metacircular interpretation [24, 5]) of the theory of signatures.

▶ **Definition 3** (Algebra Operation). *We map sort types and sort contexts to types, variables and terms are mapped to functions from the interpretation of their context to the interpretation of their types, point types and point contexts are mapped to families over the interpretation of the sort contexts.*

$$\begin{array}{lll}
\_^{\mathsf{A}} : \mathsf{Ty_s} \to \mathsf{Set} & \_^{\mathsf{A}} : \mathsf{Var_s}\,\Gamma_s\,A_s \to {\Gamma_s}^{\mathsf{A}} \to {A_s}^{\mathsf{A}} & \_^{\mathsf{A}} : \mathsf{Ty_p}\,\Gamma_s \quad \to {\Gamma_s}^{\mathsf{A}} \to \mathsf{Set} \\
\_^{\mathsf{A}} : \mathsf{Con_s} \to \mathsf{Set} & \_^{\mathsf{A}} : \mathsf{Tm_s}\,\Gamma_s\,A_s \to {\Gamma_s}^{\mathsf{A}} \to {A_s}^{\mathsf{A}} & \_^{\mathsf{A}} : \mathsf{Con_p}\,\Gamma_s \to {\Gamma_s}^{\mathsf{A}} \to \mathsf{Set}
\end{array}$$

We go through each operation in order. First, sort types are interpreted as functions into the universe (left column), and sort contexts become iterated product types (right column).

$$\begin{array}{llll}
\mathsf{U}^{\mathsf{A}} & :\equiv \mathsf{Set} & \cdot^{\mathsf{A}} & :\equiv \mathbf{1} \\
(\hat{\Pi}_{\mathsf{s}}\,T\,A_s)^{\mathsf{A}} & :\equiv (\tau : T) \to (A\,\tau)^{\mathsf{A}} & (\Gamma_s \rhd A_s)^{\mathsf{A}} & :\equiv {\Gamma_s}^{\mathsf{A}} \times {A_s}^{\mathsf{A}}
\end{array}$$

We use variables and terms to navigate these iterated products via iterated projections, and to apply function sorts to parameters.

$$\begin{array}{llll}
\mathsf{vz}^{\mathsf{A}}\,(\gamma_s, \alpha_s) & :\equiv \alpha_s & (\mathsf{var}\,x)^{\mathsf{A}}\,\gamma_s & :\equiv x^{\mathsf{A}}\,\gamma_s \\
(\mathsf{vs}\,x)^{\mathsf{A}}\,(\gamma_s, \alpha_s) & :\equiv x^{\mathsf{A}}\,\gamma_s & (t \,@\, \tau)^{\mathsf{A}}\,\gamma_s & :\equiv (t^{\mathsf{A}}\,\gamma_s)\,\tau
\end{array}$$

For point types, both function types become metatheoretic functions and we erase the element operator, since it does not have any semantic meaning. Just as sort contexts, point contexts are interdependency-free lists of the interpretations of their constituent types.

$$\begin{array}{llll}
(\mathsf{El}\,a)^{\mathsf{A}}\,\gamma_s & :\equiv a^{\mathsf{A}}\,\gamma_s & \cdot^{\mathsf{A}}\,\gamma_s & :\equiv \mathbf{1} \\
(\hat{\Pi}_{\mathsf{p}}\,T\,A)^{\mathsf{A}}\,\gamma_s & :\equiv (\tau : T) \to (A\,\tau)^{\mathsf{A}}\,\gamma_s & (\Gamma \rhd A)^{\mathsf{A}}\,\gamma_s & :\equiv \Gamma^{\mathsf{A}}\,\gamma_s \times A^{\mathsf{A}}\,\gamma_s \\
(a \Rightarrow_{\mathsf{p}} A)^{\mathsf{A}}\,\gamma_s & :\equiv a^{\mathsf{A}}\,\gamma_s \to A^{\mathsf{A}}\,\gamma_s & &
\end{array}$$

▶ **Example 4** (Revisiting Natural Numbers, Vectors, Parity). Looking at the signatures in Example 2, we observe that the algebra interpretations are the expected left-nested product types starting with $\mathbf{1}$. For natural numbers, we have ${N_s}^{\mathsf{A}} \equiv \mathbf{1} \times \mathsf{Set}$. Given a $(\star, M) : \mathbf{1} \times \mathsf{Set}$, the algebras of its point contexts become $N^{\mathsf{A}}\,(\star, M) \equiv (\mathbf{1} \times M) \times (M \to M)$. For vectors, the sorts in an algebra are elements of ${V_s}^{\mathsf{A}} \equiv \mathbf{1} \times (\mathbb{N} \to \mathsf{Set})$, and given such a $(\star, W)$, the point algebras are given by $V^{\mathsf{A}}\,(\star, W) \equiv \mathbf{1} \times W\,0 \times (A \to (n : \mathbb{N}) \to W\,n \to W\,(n+1))$. For parity, the sorts in an algebra are ${P_s}^{\mathsf{A}} \equiv \mathbf{1} \times (\mathbb{N} \to \mathsf{Set}) \times (\mathbb{N} \to \mathsf{Set})$, and given such a $(\star, E, O)$, point algebras are $P^{\mathsf{A}}\,(\star, E, O) \equiv \mathbf{1} \times E\,0 \times \big((n : \mathbb{N}) \to O\,n \to E\,(n+1)\big) \times \big((n : \mathbb{N}) \to E\,n \to O\,(n+1)\big)$.

Displayed algebras can be seen as the logical predicate interpretation [13] of the syntax.

▶ **Definition 5** (Displayed Algebra Operation). *Sort contexts and types become predicates over their algebra interpretations, while the displayed algebra interpretation of variables and terms says that they respect the predicates (usually called fundamental lemma).*

$$
\begin{aligned}
&\_^{\mathsf{D}} : (A_s : \mathsf{Ty_s}) \to {A_s}^{\mathsf{A}} \to \mathsf{Set} && \_^{\mathsf{D}} : (x : \mathsf{Var_s}\,\Gamma_s\,A_s) \to {\Gamma_s}^{\mathsf{D}}\,\gamma_s \to {A_s}^{\mathsf{D}}\,(x^{\mathsf{A}}\,\gamma_s)\\
&\_^{\mathsf{D}} : (\Gamma_s : \mathsf{Con_s}) \to {\Gamma_s}^{\mathsf{A}} \to \mathsf{Set} && \_^{\mathsf{D}} : (t : \mathsf{Tm_s}\,\Gamma_s\,A_s) \to {\Gamma_s}^{\mathsf{D}}\,\gamma_s \to {A_s}^{\mathsf{D}}\,(t^{\mathsf{A}}\,\gamma_s)
\end{aligned}
$$

Point types and contexts become predicates over their corresponding algebra interpretations, but these predicates also depend on witnesses of the predicates for the sort contexts.

$$
\begin{aligned}
&\_^{\mathsf{D}} : (A : \mathsf{Ty_p}\,\Gamma_s) \to {\Gamma_s}^{\mathsf{D}}\,\gamma_s \to A^{\mathsf{A}}\,\gamma_s \to \mathsf{Set}\\
&\_^{\mathsf{D}} : (\Gamma : \mathsf{Con_p}\,\Gamma_s) \to {\Gamma_s}^{\mathsf{D}}\,\gamma_s \to \Gamma^{\mathsf{A}}\,\gamma_s \to \mathsf{Set}
\end{aligned}
$$

The interpretation of U *is predicate space, interpretations of* $\hat{\Pi}_{\mathsf{p}}$ *and sort contexts are pointwise.*

$$
\begin{aligned}
&\mathsf{U}^{\mathsf{D}}\,T && :\equiv T \to \mathsf{U} && \cdot^{\mathsf{D}}\,\star && :\equiv \mathbf{1}\\
&(\hat{\Pi}_{\mathsf{s}}\,T\,A_s)^{\mathsf{D}}\,f_s && :\equiv (\tau : T) \to (A_s\,\tau)^{\mathsf{D}}\,(f_s\,\tau) && (\Gamma_s \rhd A_s)^{\mathsf{D}}\,(\gamma_s, \alpha_s) && :\equiv \Gamma_s^{\mathsf{D}}\,\gamma_s \times {A_s}^{\mathsf{D}}\,\alpha_s
\end{aligned}
$$

The interpretation of terms follows the same pattern as for algebras, variables are lookups, application is metatheoretic application, we omit listing them. On point types, the interpretation of El *is again non-interesting, the interpretation of* $\hat{\Pi}_{\mathsf{p}}$ *is pointwise, while the interpretation of* $\Rightarrow_{\mathsf{p}}$ *says that if the predicate holds for the input, then it holds for the output.*

$$
\begin{aligned}
&(\mathsf{El}\,a)^{\mathsf{D}}\,\gamma_s^d\,\alpha && :\equiv a^{\mathsf{D}}\,\gamma^d\,\alpha\\
&(\hat{\Pi}_{\mathsf{p}}\,T\,A)^{\mathsf{D}}\,\gamma_s^d\,f && :\equiv (\tau : T) \to (A\,\tau)^{\mathsf{D}}\,\gamma_s^d\,(f\,\tau)\\
&(a \Rightarrow_{\mathsf{p}} A)^{\mathsf{D}}\,\gamma_s^d\,f && :\equiv \{\alpha : a^{\mathsf{A}}\,\gamma_s\} \to a^{\mathsf{D}}\,\gamma_s^d\,\alpha \to A^{\mathsf{D}}\,\gamma_s^d\,(f\,\alpha)
\end{aligned}
$$

Finally, point contexts are interpreted as iterated products again, they contain witnesses that the predicates hold for everything in the algebra.

$$
\begin{aligned}
&\cdot^{\mathsf{D}}\,\gamma_s^d\,\gamma && :\equiv \mathbf{1}\\
&(\Gamma \rhd A)^{\mathsf{D}}\,\gamma_s^d\,(\gamma, \alpha) && :\equiv \Gamma^{\mathsf{D}}\,\gamma_s^d\,\gamma \times A^{\mathsf{D}}\,\gamma_s^d\,\alpha
\end{aligned}
$$

▶ **Example 6** (Revisiting Natural Numbers, Vectors, Parity). Given $(\star, M) : {N_s}^{\mathsf{A}}$ and $(\star, z, s) : N^{\mathsf{A}}\,(\star, M)$, the displayed sort algebra is a predicate on M, concretely ${N_s}^{\mathsf{D}}\,(\star, M) \equiv \mathbf{1} \times (M \to \mathsf{Set})$. This can be seen as the motive of the eliminator if (M, z, s) is the initial algebra. Given such a $(\star, Q)$, the displayed point algebra computes the types of methods of the eliminator, $N^{\mathsf{D}}\,(\star, Q)\,(\star, z, s) \equiv \mathbf{1} \times Q\,z \times \big((n : M) \to Q\,n \to Q\,(s\,n)\big)$ as expected. Given a vector algebra $(\star, W)$, $(\star, \mathit{nil}, \mathit{cons})$, a displayed sort algebra computes to ${V_s}^{\mathsf{D}}\,(\star, W) \equiv \mathbf{1} \times ((n : \mathbb{N}) \to W\,n \to \mathsf{Set})$, and the displayed point algebra is $V^{\mathsf{D}}\,(\star, Q)\,(\star, \mathit{nil}, \mathit{cons}) \equiv \mathbf{1} \times Q\,0\,\mathit{nil} \times \big((a : A)(x : \mathbb{N})(v : W\,n) \to Q\,n\,v \to Q\,(n+1)\,(\mathit{cons}\,a\,x\,v)\big)$. Finally, given a parity algebra $(\star, E, O)$, $(\star, \mathit{e0}, \mathit{eS}, \mathit{oS})$, the displayed sort algebra consists of ${P_s}^{\mathsf{D}}\,(\star, E, O) \equiv \mathbf{1} \times ((n : \mathbb{N}) \to E\,n \to \mathsf{Set}) \times ((n : \mathbb{N}) \to O\,n \to \mathsf{Set})$ and given such a $(\star, Q, R)$, displayed point algebras are $P^{\mathsf{D}}\,(\star, Q, R)\,(\star, \mathit{e0}, \mathit{eS}, \mathit{oS}) \equiv \mathbf{1} \times Q\,0\,\mathit{e0} \times \big((n : \mathbb{N})(o : O\,n) \to R\,n\,o \to Q\,(n+1)\,(\mathit{eS}\,n\,o)\big) \times \big((n : \mathbb{N})(e : E\,n) \to Q\,n\,e \to R\,(n+1)\,(\mathit{oS}\,n\,e)\big)$. Given a family Q over E and a family R over O, these express that $\mathit{e0}$ witnesses Q, while eS turns witnesses of R into witnesses of Q and oS turns witnesses of Q into witnesses of R.

Sections are dependent binary logical relations, where the interpretation of U, El and $\Rightarrow_{\mathsf{p}}$ are non-standard.

▶ **Definition 7** (Section Operation). *For sorts and sort contexts, sections are dependent binary relations over the corresponding algebra and displayed algebra. "Dependent" here means the type of the second argument of the relation depends on the first.*

$$\begin{aligned}
&\_^{\mathsf{S}} : (A_s : \mathsf{Ty_s}) && \to (\alpha_s : {A_s}^{\mathsf{A}}) \to {A_s}^{\mathsf{D}}\,\alpha_s \to \mathsf{Set}\\
&\_^{\mathsf{S}} : (\Gamma_s : \mathsf{Con_s}) && \to (\gamma_s : {\Gamma_s}^{\mathsf{A}}) \to {\Gamma_s}^{\mathsf{D}}\,\gamma_s \to \mathsf{Set}
\end{aligned}$$

The interpretation of variables expresses that if the relation holds at the context, then it also holds at the type for the algebra and displayed algebra interpretation of the variable. We have the same for terms.

$$\begin{aligned}
&\_^{\mathsf{S}} : (x : \mathsf{Var_s}\,\Gamma_s\,A_s) \to {\Gamma_s}^{\mathsf{S}}\,\gamma_s\,\gamma_s^d \to {A_s}^{\mathsf{S}}\,(x^{\mathsf{A}}\,\gamma_s)\,(x^{\mathsf{D}}\,\gamma_s^d)\\
&\_^{\mathsf{S}} : (t : \mathsf{Tm_s}\,\Gamma_s\,A_s) \to {\Gamma_s}^{\mathsf{S}}\,\gamma_s\,\gamma_s^d \to {A_s}^{\mathsf{S}}\,(t^{\mathsf{A}}\,\gamma_s)\,(t^{\mathsf{D}}\,\gamma_s^d)
\end{aligned}$$

The interpretation of point types are dependent binary relations displayed over witnesses of relatedness for the relations for the contexts.

$$\begin{aligned}
&\_^{\mathsf{S}} : (A : \mathsf{Ty_p}\,\Gamma_s) && \to {\Gamma_s}^{\mathsf{S}}\,\gamma_s\,\gamma_s^d \to (\alpha : A^{\mathsf{A}}\,\gamma_s) \to A^{\mathsf{D}}\,\gamma_s^d\,\alpha \to \mathsf{Set}\\
&\_^{\mathsf{S}} : (\Gamma : \mathsf{Con_p}\,\Gamma_s) && \to {\Gamma_s}^{\mathsf{S}}\,\gamma_s\,\gamma_s^d \to (\gamma : \Gamma^{\mathsf{A}}\,\gamma_s) \to \Gamma^{\mathsf{D}}\,\gamma_s^d\,\gamma \to \mathsf{Set}
\end{aligned}$$

Sections of the universe are given as dependent functions (instead of dependent relation space as is usual for logical relations). The interpretation of $\hat{\Pi}_{\mathsf{s}}$ is pointwise, and so is that of sort contexts.

$$\begin{aligned}
&\mathsf{U}^{\mathsf{S}}\,T\,T^d && :\equiv (\tau : T) \to T^d\,\tau\\
&(\hat{\Pi}_{\mathsf{s}}\,T\,A_s)^{\mathsf{S}}\,f_s\,f_s^d && :\equiv (\tau : T) \to (A_s\,\tau)^{\mathsf{S}}\,(f_s\,\tau)\,(f_s^d\,\tau)\\
&\cdot^{\mathsf{S}}\,\star\,\star && :\equiv \mathbf{1}\\
&(\Gamma_s \rhd A_s)^{\mathsf{S}}\,(\gamma_s,\alpha_s)\,(\gamma_s^d,\alpha_s^d) && :\equiv {\Gamma_s}^{\mathsf{S}}\,\gamma_s\,\gamma_s^d \times {A_s}^{\mathsf{S}}\,\alpha_s\,\alpha_s
\end{aligned}$$

Sort terms follow the usual pattern of variables selecting sort interpretations via projections of products and interpreting the application by metatheoretic application:

$$\begin{aligned}
&\mathsf{vz}^{\mathsf{S}}\,(\gamma_s^s,\alpha_s^s) && :\equiv \alpha_s^s && \qquad (\mathsf{var}\,x)^{\mathsf{S}}\,\gamma_s^s && :\equiv x^{\mathsf{S}}\,\gamma_s^s\\
&(\mathsf{vs}\,x)^{\mathsf{S}}\,(\gamma_s^s,\alpha_s^s) && :\equiv x^{\mathsf{S}}\,\gamma_s^s && \qquad (t\,@\,\tau)^{\mathsf{S}}\,\gamma_s^s && :\equiv t^{\mathsf{S}}\,\gamma_s^s\,\tau
\end{aligned}$$

Sections on point types express equalities. Each point type ends with an $\mathsf{El}\,a$, and the section says that the function given by a^{S} returns the witness of the predicate α^d. $\hat{\Pi}_{\mathsf{p}}$ is defined pointwise, while $\Rightarrow_{\mathsf{p}}$ says that for any input, the outputs of f and f^d are related by A^{S}, where we use a^{S} again to produce a witness of the predicate on the right hand side.

$$\begin{aligned}
&(\mathsf{El}\,a)^{\mathsf{S}}\,\gamma_s^s\,\alpha\,\alpha^d && :\equiv (a^{\mathsf{S}}\,\gamma_s^s\,\alpha = \alpha^d)\\
&(\hat{\Pi}_{\mathsf{p}}\,T\,A)^{\mathsf{S}}\,\gamma_s^s\,f\,f^d && :\equiv (\tau : T) \to (A\,\tau)^{\mathsf{S}}\,\gamma_s^s\,(f\,\tau)\,(f^d\,\tau)\\
&(a \Rightarrow_{\mathsf{p}} A)^{\mathsf{S}}\,\gamma_s^s\,f\,f^d && :\equiv (\alpha : a^{\mathsf{A}}\,\gamma_s) \to A^{\mathsf{S}}\,\gamma_s^s\,(f\,\alpha)\,(f^d\,(a^{\mathsf{S}}\,\gamma_s^s\,\alpha))
\end{aligned}$$

The definition of sections of point contexts is, again, just an iteration of products.

$$\begin{aligned}
&\cdot^{\mathsf{S}}\,\gamma_s^s\,\gamma\,\gamma^d && :\equiv \mathbf{1}\\
&(\Gamma \rhd A)^{\mathsf{S}}\,\gamma_s^s\,(\gamma,\alpha)\,(\gamma^d,\alpha^d) && :\equiv \Gamma^{\mathsf{S}}\,\gamma_s^s\,\gamma\,\gamma^d \times A^{\mathsf{S}}\,\gamma_s^s\,\alpha\,\alpha^d
\end{aligned}$$

▶ **Example 8** (Revisiting Natural Numbers, Vectors, Parity)**.** Using the same notation for algebras and displayed algebras as in Example 6, a section of a natural number displayed algebra is a $(\star, f)$ having type ${N_s}^{\mathsf{S}}\,(\star, M)\,(\star, Q) \equiv \mathbf{1} \times ((n : M) \to Q\,n)$ together with a witness of $N^{\mathsf{S}}\,(\star, f)\,(\star, z, s)\,(\star, w, h) \equiv \mathbf{1} \times (f\,z = w) \times ((n : N) \to f\,(s\,n) = h\,n\,(f\,n))$. These equalities are the computation rules of the eliminator. For vectors, the section operation computes to ${V_s}^{\mathsf{S}}\,(\star, W)\,(\star, Q) \equiv \mathbf{1} \times \big((n : \mathbb{N})(v : W\,n) \to Q\,n\,v\big)$ and to $V^{\mathsf{S}}\,(\star, f)\,(\star, \mathit{nil}, \mathit{cons})\,(\star, \mathit{nil}^d, \mathit{cons}^d) \equiv \mathbf{1} \times (f\,0\,\mathit{nil} = \mathit{nil}^d) \times \big((a : A)(n : \mathbb{N})(v : W\,n) \to f\,(n+1)\,(\mathit{cons}\,a\,n\,v) = \mathit{cons}^d\,a\,n\,(f\,n\,v)\big)$. For the parity families, sections are ${P_s}^{\mathsf{S}}\,(\star, E, O)\,(\star, Q, R) \equiv \mathbf{1} \times ((n : \mathbb{N})(e : E\,n) \to Q\,n\,e) \times ((n : \mathbb{N})(o : O\,n) \to R\,n\,o)$ together with $P^{\mathsf{S}}\,(\star, f, g)\,(\star, e0, eS, oS)\,(e0^d, eS^d, oS^d) \equiv \mathbf{1} \times (f\,0\,e0 = e0^d) \times \big((n : \mathbb{N})(o : O\,n) \to f\,(n+1)\,(eS\,n\,o) = eS^d\,n\,(g\,n\,o)\big) \times \big((n : \mathbb{N})(e : E\,n) \to g\,(n+1)\,(oS\,n\,e) = oS^d\,n\,(f\,n\,e)\big)$. A section for parity displayed algebras consists of two functions f, g which map $e0$ to $e0^d$, eS to eS^d and oS to oS^d.

4 Existence of Inductive Families

When does a type theory "support" types of our specification of mutual inductive families and how does this compare to well-established notions of inductive types? The intended meaning of the signatures is clear from the definition of their algebras as seen in Section 3, the types of their eliminators and computation rules are specified in the definitions of displayed algebras and sections. This means that we can formally say what it means for inductive families to exist in a type theory. In this section, we will prove that any metatheory with indexed W-types supports our notion of mutual inductive families or, in other words, mutual inductive families can be *reduced* to indexed W-types:

▶ **Theorem 9** (Existence of Inductive Families)**.** *For every signature of inductive families given by a sort context $\Omega_s : \mathsf{Con_s}$ and point context $\Omega : \mathsf{Con_p}\,\Omega_s$, there are are sort and point* constructors *in the form of*

$$\begin{aligned} \mathsf{con_s}\,\Omega &: {\Omega_s}^{\mathsf{A}} \text{ and} \\ \mathsf{con}\,\Omega &: \Omega^{\mathsf{A}}\,(\mathsf{con_s}\,\Omega) \end{aligned}$$

such that for each displayed algebra given by motives $\omega_s^d : {\Omega_s}^{\mathsf{D}}\,(\mathsf{con_s}\,\Omega)$ and methods $\omega^d : \Omega^{\mathsf{D}}\,\omega_s^d\,(\mathsf{con}\,\Omega)$ we have an eliminator *given by sections*

$$\begin{aligned} \mathsf{elim_s}\,\Omega\,\omega_s^d &: {\Omega_s}^{\mathsf{S}}\,(\mathsf{con_s}\,\Omega)\,\omega_s^d \text{ with} \\ \mathsf{elim}\,\Omega\,\omega^d &: \Omega^{\mathsf{S}}\,(\mathsf{elim_s}\Omega\omega_s^d)\,(\mathsf{con}\,\Omega)\,\omega^d. \end{aligned}$$

Note that this definition of existence only requires the computation rules contained in $\mathsf{elim}\,\Omega\,\omega^d$ to hold *propositionally*. One might also wish for *strict* reduction rules instead to enable better computational behaviour.

Our strategy to prove this theorem is to first extend our syntax to a full substitution calculus including sort and point substitutions and point types (Section 4.1). Then we construct a *term model* using the extended syntax, which we can then show to be the initial algebra (Section 4.2).

4.1 A Substitution Calculus for the Syntax

The syntax for usual type theories includes substitutions. We did not have to mention them in the theory of signatures because of the simplicity of mutual inductive definitions. In other words, our syntax only contains normal forms (there are also no conversion rules in our syntax). However when doing constructions on the syntax, it is sometimes useful to have a full syntax, this includes a category of substitutions. We will make use of them in Section 4.2.

▶ **Definition 10** (Sort Substitutions). *A calculus of substitutions* $\mathsf{Sub_s}$ *of sort contexts is useful to compare sort contexts themselves as well as to relate point contexts over different sort contexts. We define them to be inductively generated by*

$$\begin{aligned}
&\mathsf{Sub_s} : \mathsf{Con_s} \to \mathsf{Con_s} \to \mathsf{Set}\\
&\epsilon \quad\;\; : \mathsf{Sub_s}\,\mathit{\Gamma}_s\,\cdot\\
&\text{–},\text{–} \;\; : \mathsf{Sub_s}\,\mathit{\Gamma}_s\,\mathit{\Delta}_s \to \mathsf{Tm_s}\,\mathit{\Gamma}_s\,A_s \to \mathsf{Sub_s}\,\mathit{\Gamma}_s\,(\mathit{\Delta}_s \rhd A_s)
\end{aligned}$$

Like with the syntax of signatures itself, $\mathsf{Sub_s}$ *can be encoded as an indexed W-type as shown in Appendix A. These substitutions allow us to substitute point types, point contexts, and sort terms via the following "pullback" operations:*

$$\begin{aligned}
&\text{–}[\text{–}] : \mathsf{Ty_p}\,\mathit{\Delta}_s \;\;\to \mathsf{Sub_s}\,\mathit{\Gamma}_s\,\mathit{\Delta}_s \to \mathsf{Ty_p}\,\mathit{\Gamma}_s \qquad &&\text{–}[\text{–}] : \mathsf{Var_s}\,\mathit{\Delta}_s\,A_s \to \mathsf{Sub_s}\,\mathit{\Gamma}_s\,\mathit{\Delta}_s \to \mathsf{Tm_s}\,\mathit{\Gamma}_s\,A_s\\
&\text{–}[\text{–}] : \mathsf{Con_p}\,\mathit{\Delta}_s \to \mathsf{Sub_s}\,\mathit{\Gamma}_s\,\mathit{\Delta}_s \to \mathsf{Con_p}\,\mathit{\Gamma}_s &&\text{–}[\text{–}] : \mathsf{Tm_s}\,\mathit{\Delta}_s\,A_s \to \mathsf{Sub_s}\,\mathit{\Gamma}_s\,\mathit{\Delta}_s \to \mathsf{Tm_s}\,\mathit{\Gamma}_s\,A_s
\end{aligned}$$

given by the defining rules for substitution

$$\begin{aligned}
&\hat{\Pi}_{\mathsf{p}}\,T\,A[\sigma] &&:\equiv \hat{\Pi}_{\mathsf{p}}\,T\,(\lambda\tau.(A\,\tau)[\sigma]) \qquad &&\mathsf{vz}[\sigma,t] &&:\equiv t\\
&\mathsf{El}\,a[\sigma] &&:\equiv \mathsf{El}\,(a[\sigma]) &&(\mathsf{vs}\,x)[\sigma,t] &&:\equiv x[\sigma]\\
&(a \Rightarrow_{\mathsf{p}} A)[\sigma] &&:\equiv a[\sigma] \Rightarrow_{\mathsf{p}} A[\sigma] &&(\mathsf{var}\,x)[\sigma] &&:\equiv x[\sigma]\\
&\cdot\,[\sigma] &&:\equiv \cdot &&(t\,@\,\tau)[\sigma] &&:\equiv t[\sigma]\,@\,\tau\\
&(\mathit{\Gamma} \rhd A)[\sigma] &&:\equiv \mathit{\Gamma}[\sigma] \rhd A[\sigma]
\end{aligned}$$

We can derive from this some useful gadgets of the substitutional calculus: We can define the weakening *of a substitution* $\sigma : \mathsf{Sub_s}\,\mathit{\Gamma}_s\,\mathit{\Delta}_s$ *to the substitution* $\mathsf{wk}_\sigma : \mathsf{Sub_s}\,(\mathit{\Gamma}_s \rhd A_s)\,\mathit{\Delta}_s$ *via recursion on* σ *by* $\mathsf{wk}_\epsilon :\equiv \epsilon$ *and* $\mathsf{wk}_{\sigma,t} :\equiv (\mathsf{wk}_\sigma, \mathsf{vs}\,t)$.

Using wk*, we can then recover the categorical structure of the substitutions by defining the identity* $\mathsf{id}_{\mathit{\Gamma}_s} : \mathsf{Sub_s}\,\mathit{\Gamma}_s\,\mathit{\Gamma}_s$ *by recursion of the context* $\mathit{\Gamma}_s$*:* $\mathsf{id}_{\cdot} :\equiv \epsilon$ *and* $\mathsf{id}_{\mathit{\Gamma}_s \rhd A_s} :\equiv (\mathsf{wk}_{\mathsf{id}_{\mathit{\Gamma}_s}}, \mathsf{var}\,\mathsf{vz})$. *Composition* $\sigma \circ \delta : \mathsf{Sub_s}\,\mathit{\Gamma}_s\,\mathit{\Delta}_s$ *of substitutions* $\sigma : \mathsf{Sub_s}\,\mathit{\Theta}_s\,\mathit{\Delta}_s$ *and* $\delta : \mathsf{Sub_s}\,\mathit{\Gamma}_s\,\mathit{\Delta}_s$ *is defined by recursion on the first substitution:* $\epsilon \circ \delta :\equiv \epsilon$*,* $(\sigma, t) \circ \delta :\equiv (\sigma \circ \delta, t[\delta])$.

The projections $\pi_1\,\sigma : \mathsf{Sub_s}\,\mathit{\Gamma}_s\,\mathit{\Delta}_s$ *and* $\pi_2\,\sigma : \mathsf{Tm_{ss}}\,\mathit{\Gamma}_s\,A_s$ *of a substitution* $\sigma : \mathsf{Sub_s}\,\mathit{\Gamma}_s\,(\mathit{\Delta}_s \rhd A_s)$ *are just projections of* $\times$*-types: Any substitution between* $\mathit{\Gamma}_s$ *and* $\mathit{\Delta}_s \rhd A_s$ *is of the form* $\sigma,\,t$ *and we can just set* $\pi_1\,(\sigma,\,t) :\equiv \sigma$ *and* $\pi_2\,(\sigma,\,t) :\equiv t$.

Obviously, we might also want to consider algebras, displayed algebras, and their sections over these substitutions.

▶ **Definition 11** (Semantics of Sort Substitutions). *We can extend the algebra operator by defining it on substitutions by functions between the interpretations of sort contexts:*

$$\text{–}^{\mathsf{A}} : \mathsf{Sub_s}\,\mathit{\Gamma}_s\,\mathit{\Delta}_s \to {\mathit{\Gamma}_s}^{\mathsf{A}} \to {\mathit{\Delta}_s}^{\mathsf{A}}$$

This is done by setting $\epsilon^{\mathsf{A}} :\equiv \star$ *and* $(\sigma,\,t)^{\mathsf{A}} :\equiv (\sigma^{\mathsf{A}}, t^{\mathsf{A}})$.

The type of displayed algebras over a sort substitution should be the type of function between the displayed algebras of its domain and codomain, where in the latter we have to apply the function which we get from the algebra *over the substitution:*

$$\_^{\mathsf{D}} : (\sigma : \mathsf{Sub_s}\, \mathit{\Gamma}_s\, \mathit{\Delta}_s) \to {\mathit{\Gamma}_s}^{\mathsf{D}}\, \gamma_s \to {\mathit{\Delta}_s}^{\mathsf{D}}\, (\sigma^{\mathsf{A}}\, \gamma_s)$$

These are defined, like in the non-displayed case, by setting $\epsilon^{\mathsf{D}}\, \gamma_s^d :\equiv \star$ *and* $(\sigma,\, t)^{\mathsf{D}}\, \gamma_s^d :\equiv \big(\sigma^{\mathsf{D}}\, \gamma_s^d, t^{\mathsf{D}}\, \gamma_s^d\big)$*.*

A section of a displayed algebra of a sort substitution is supposed to map sections of its domain to sections of its codomain:

$$\_^{\mathsf{S}} : (\sigma : \mathsf{Sub_s}\, \mathit{\Gamma}_s\, \mathit{\Delta}_s) \to {\mathit{\Gamma}_s}^{\mathsf{S}}\, \gamma_s\, \gamma_s^d \to {\mathit{\Delta}_s}^{\mathsf{S}}\, (\sigma^{\mathsf{A}}\, \gamma_s)\, (\sigma^{\mathsf{D}}\, \gamma_s^d)$$

Again, this is happening componentwise by having: $\epsilon^{\mathsf{S}}\, \gamma_s^s :\equiv \star$ *and* $(\sigma,\, t)^{\mathsf{S}}\, \gamma_s^s :\equiv \big(\sigma^{\mathsf{S}}\, \gamma_s^s, t^{\mathsf{S}}\, \gamma_s^s\big)$*.*

▶ **Lemma 12.** *It is easy to check that this definition of algebras of a subtitution respects the substitution calculus given in Definition 10 in the following sense:*

$$\begin{aligned}
(A[\sigma])^{\mathsf{A}}\, \gamma_s &= A^{\mathsf{A}}\, (\sigma^{\mathsf{A}}\, \gamma_s), & \mathsf{wk_\sigma}^{\mathsf{A}}\, (\gamma_s, \alpha_s) &= \sigma^{\mathsf{A}}\, \gamma_s,\\
(t[\sigma])^{\mathsf{A}}\, \gamma_s &= t^{\mathsf{A}}\, (\sigma^{\mathsf{A}}\, \gamma_s), & (\pi_1\, \sigma)^{\mathsf{A}}\, \gamma_s &= \mathsf{pr}_1\, (\sigma^{\mathsf{A}}\, \gamma_s), \textit{ and}\\
\mathsf{id}^{\mathsf{A}}\, \gamma_s &= \gamma_s, & (\pi_2\, \sigma)^{\mathsf{A}}\, \gamma_s &= \mathsf{pr}_2\, (\sigma^{\mathsf{A}}\, \gamma_s).\\
(\sigma \circ \delta)^{\mathsf{A}}\, \gamma_s &= \sigma^{\mathsf{A}}\, (\delta^{\mathsf{A}}\, \gamma_s), & &
\end{aligned}$$

Proof. We can prove the first rule by recursion on the point type $A : \mathsf{Ty_p}\, \mathit{\Gamma}_s$, the second rule by recursing on the term $t : \mathsf{Tm_s}\, \mathit{\Gamma}_s\, A_s$, the third by induction on the context, and all other by induction by the substitution. Analogous rules hold for displayed algebras over substitutions. ◀

The model which is initial in the category of all models is usually called the *term model*. This is because in this model, a type gets interpreted as the set of all of its terms. Since our signatures form – or are at least strongly inspired by – a type theoretic syntax as well, we might hope to deploy the same strategy for inductive families. In the core of this interpretation is the issue of how to find an interpretation for a given sort term a of the universe token U. The interpretation of this ought to be the terms of the *point type* $\mathsf{El}(a)$ associated with this sort term. But our syntax does not mention terms of point types at all, since point constructors are not interdependent! So our solution is to retrofit the theory with terms, as well as substitutions for the point contexts:

▶ **Definition 13** (Point Substitution Calculus). *Let us fix a sort context* $\mathit{\Gamma}_s : \mathsf{Con_s}$*. It turns out that there are three ways to construct reasonable terms of point types in* $\mathit{\Gamma}_s$*: Via variables to navigate point contexts and application constructors for each of the two kinds of* Π*-types present in the syntax.*

$$\begin{aligned}
\mathsf{Var_p} &: \mathsf{Con_p}\, \mathit{\Gamma}_s \to \mathsf{Ty_p}\, \mathit{\Gamma}_s \to \mathsf{Set} & \mathsf{var}\ &: \mathsf{Var_p}\, \mathit{\Gamma}\, A \to \mathsf{Tm_p}\, \mathit{\Gamma}\, A\\
\mathsf{Tm_p} &: \mathsf{Con_p}\, \mathit{\Gamma}_s \to \mathsf{Ty_p}\, \mathit{\Gamma}_s \to \mathsf{Set} & -\hat{@}-&: \mathsf{Tm_p}\, \mathit{\Gamma}\, (\hat{\Pi}_{\mathsf{p}}\, T\, A) \to (\tau : T) \to \mathsf{Tm_p}\, \mathit{\Gamma}\, (A\, \tau)\\
\mathsf{vz} &: \mathsf{Var_p}\, (\mathit{\Gamma} \rhd A)\, A & -@-&: \mathsf{Tm_p}\, \mathit{\Gamma}\, (a \Rightarrow_{\mathsf{p}} A) \to \mathsf{Tm_p}\, \mathit{\Gamma}\, (\mathsf{El}\, a) \to \mathsf{Tm_p}\, \mathit{\Gamma}\, A\\
\mathsf{vs} &: \mathsf{Var_p}\, \mathit{\Gamma}\, A \to \mathsf{Var_p}\, (\mathit{\Gamma} \rhd B)\, A & &
\end{aligned}$$

Like with the sort substitutions defined in Definition 10, we define substitutions between point contexts over a fixed sort context $\mathit{\Gamma}_s : \mathsf{Con_s}$ *to be lists of point terms:*

$$\begin{aligned}
&\mathsf{Sub_p} : \mathsf{Con_p}\,\mathit{\Gamma}_s \to \mathsf{Con_p}\,\mathit{\Gamma}_s \to \mathsf{Set}\\
&\epsilon \quad\;\;\, : \mathsf{Sub_p}\,\mathit{\Gamma}\,\cdot\\
&\_,\_ \;\; : \mathsf{Sub_p}\,\mathit{\Gamma}\,\mathit{\Delta} \to \mathsf{Tm_p}\,\mathit{\Gamma}\,A \to \mathsf{Sub_p}\,\mathit{\Gamma}\,(\mathit{\Delta} \rhd A)
\end{aligned}$$

All of these can again be encoded as indexed W-types (cf. Appendix A). We can again define a pullback operations for variables and terms – this time for point terms – along substitutions in the form of

$$\_[\_] : \mathsf{Var_p}\,\mathit{\Delta}\,A \to \mathsf{Sub_p}\,\mathit{\Gamma}\,\mathit{\Delta} \to \mathsf{Tm_p}\,\mathit{\Gamma}\,A \qquad \_[\_] : \mathsf{Tm_p}\,\mathit{\Delta}\,A \to \mathsf{Sub_p}\,\mathit{\Gamma}\,\mathit{\Delta} \to \mathsf{Tm_p}\,\mathit{\Gamma}\,A$$

which are defined recursively by

$$\begin{aligned}
&\mathsf{vz}[\sigma,t] &&:\equiv t && (\mathsf{var}\,x)[\sigma] &&:\equiv x[\sigma]\\
&(\mathsf{vs}\,x)[\sigma,t] &&:\equiv x[\sigma] && (t\,\hat{@}\,\tau)[\sigma] &&:\equiv t[\sigma]\,\hat{@}\,\tau\\
& && && (t\,@\,u)[\sigma] &&:\equiv t[\sigma]\,@\,u[\sigma]
\end{aligned}$$

Analogously to Definition 10 we can define the weakening $\mathsf{wk}_\sigma : \mathsf{Sub_p}\,(\mathit{\Gamma} \rhd A)\,\mathit{\Delta}$ *of a point substitution* $\sigma : \mathsf{Sub_p}\,\mathit{\Gamma}\,\mathit{\Delta}$ *along a point type* $A : \mathsf{Ty_p}\,\mathit{\Gamma}_s$*, the identity substitution* $\mathsf{id} : \mathsf{Sub_p}\,\mathit{\Gamma}\,\mathit{\Gamma}$ *and the composition* $\sigma \circ \delta : \mathsf{Sub_p}\,\mathit{\Gamma}\,\mathit{\Delta}$ *of substitutions* $\sigma : \mathsf{Sub_p}\,\mathit{\Theta}\,\mathit{\Delta}$ *and* $\delta : \mathsf{Sub_p}\,\mathit{\Gamma}\,\mathit{\Theta}$*.*

As an auxiliary construction for our existence proof we will furthermore need notions of algebras, displayed algebras, and sections for the point terms and point substitutions:

▶ **Definition 14** (Semantics of Point Substitutions & Terms). *Let us fix a sort context* $\mathit{\Gamma}_s : \mathsf{Con_s}$ *and an algebra* $\gamma_s : \mathit{\Gamma}_s^{\mathsf{A}}$ *over it. We can give semantic meaning to point types and point substitution by extending the algebra operator with the following components:*

$$\begin{aligned}
&\_^{\mathsf{A}} : \mathsf{Var_p}\,\mathit{\Gamma}\,A \to \mathit{\Gamma}^{\mathsf{A}}\,\gamma_s \to A^{\mathsf{A}}\,\gamma_s && \_^{\mathsf{A}} : \mathsf{Sub_p}\,\mathit{\Gamma}\,\mathit{\Delta} \to \mathit{\Gamma}^{\mathsf{A}}\,\gamma_s \to \mathit{\Delta}^{\mathsf{A}}\,\gamma_s\\
&\_^{\mathsf{A}} : \mathsf{Tm_p}\,\mathit{\Gamma}\,A \to \mathit{\Gamma}^{\mathsf{A}}\,\gamma_s \to A^{\mathsf{A}}\,\gamma_s
\end{aligned}$$

These components are, in essence, defined the same way as their respective parts on sorts. Of course, apart from these defining equations, this definition of algebras is also well-behaved under the other components of substitutional calculus, analogous to sort substitutions (cf. Lemma 12).

Let us now also fix a displayed algebra $\gamma_s^d : {\mathit{\Gamma}_s}^{\mathsf{D}}\,\gamma_s$*. For the displayed version of these algebras, the interpretation of point terms and of point substitutions needs to depend on these and, additionally, on an algebra and displayed algebra of the underlying point context. This leads to the following interpretations:*

$$\begin{aligned}
&\_^{\mathsf{D}} : (x : \mathsf{Var_p}\,\mathit{\Gamma}\,A) &&\to \mathit{\Gamma}^{\mathsf{D}}\,\gamma_s^d\,\gamma \to A^{\mathsf{D}}\,\gamma_s^d\,(x^{\mathsf{A}}\,\gamma)\\
&\_^{\mathsf{D}} : (t : \mathsf{Tm_p}\,\mathit{\Gamma}\,A) &&\to \mathit{\Gamma}^{\mathsf{D}}\,\gamma_s^d\,\gamma \to A^{\mathsf{D}}\,\gamma_s^d\,(t^{\mathsf{A}}\,\gamma)\\
&\_^{\mathsf{D}} : (\sigma : \mathsf{Sub_p}\,\mathit{\Gamma}\,\mathit{\Delta}) &&\to \mathit{\Gamma}^{\mathsf{D}}\,\gamma_s^d\,\gamma \to \mathit{\Delta}^{\mathsf{D}}\,\gamma_s^d\,(\sigma^{\mathsf{A}}\,\gamma)
\end{aligned}$$

Again, we define them by equations resembling the ones for sort substitutions, and again, substitution rules analogous to the ones in Lemma 12 hold.

4.2 Constructing all Inductive Families from the Syntax

In this section, assuming our type theory supports the theory of signatures (including the extensions of Section 4.1), we show that all mutual inductive families described by signatures exist. To give an intuition for this construction, consider the example of natural numbers: In its initial algebra, we want the interpretation of the U sort to contain exactly the elements z, $s\,z$, $s\,(s\,z)$, $\dots$, where z and s are point terms, pointing to the zero and successor constructor, respectively. But we observe that these are just the point terms of the type $\mathsf{El}\,N$ in the context $\Delta := (N : \mathsf{U}, z : \mathsf{El}\,N, s : N \Rightarrow_{\mathsf{p}} \mathsf{El}\,N)$ (for the sake of the example, we use variable names and don't separate sort and point contexts). So we define the initial algebra as $\big(\mathsf{Tm}\,\Delta\,(\mathsf{El}\,N), z, \lambda t.s \mathbin{@} t\big) : \Delta^{\mathsf{A}}$. Note that given any other algebra $(A, a, f) : \Delta^{\mathsf{A}}$ and natural number $n : \mathsf{Tm}\,\Delta\,(\mathsf{El}\,N)$, we can simply use the algebra interpretation to obtain the result of the non-dependent elimination principle on n: $n^{\mathsf{A}}\,(A, a, f)$ will have type A, moreover $z^{\mathsf{A}}\,(A, a, f) = a$ and $(s \mathbin{@} t)^{\mathsf{A}}\,(A, a, f) = f\,(t^{\mathsf{A}}\,(A, a, f))$ which are the correct computation rules. The same idea works for displayed algebras: we can use the –^{D} operation on a natural number (given as a term) to obtain the result of the dependent elimination principle. In the following we will give the general description of this approach and prove its initiality by giving the dependent eliminator.

For the remainder of this section, let us fix the sort context $\Omega_s : \mathsf{Con}_{\mathsf{s}}$ and the point context $\Omega : \mathsf{Con}_{\mathsf{p}}\,\Omega_s$ which we want to construct by giving $\mathsf{con}_{\mathsf{s}}\,\Omega : {\Omega_s}^{\mathsf{A}}$ and $\mathsf{con}\,\Omega : \Omega^{\mathsf{A}}\,(\mathsf{con}_{\mathsf{s}}\,\Omega)$. Our definition of the constructor uses the trick to index several of the constructions by a second sort or point context together with a sort or point substitution from Ω_s or Ω. We can think of this second context as some sort of a "sub-context" of a fixed context.

▶ **Definition 15** (The Sort Constructor). *The generalised sort constructor consists of:*

$$\mathsf{con}'_{\mathsf{s}} : \mathsf{Sub}_{\mathsf{s}}\,\Omega_s\,\Gamma_s \to {\Gamma_s}^{\mathsf{A}}$$

We can define this recursively via $\mathsf{con}'_{\mathsf{s}}\,\epsilon :\equiv \star$ *and* $\mathsf{con}'_{\mathsf{s}}\,(\sigma, t) :\equiv (\mathsf{con}'_{\mathsf{s}}\,\sigma, \mathsf{con}'_{\mathsf{s}}\,t)$ *where on sort terms we will define a constructor operation yielding an algebra of the respective sort type:*

$$\mathsf{con}'_{\mathsf{s}} : \mathsf{Tm}_{\mathsf{s}}\,\Omega_s\,A_s \to {A_s}^{\mathsf{A}}$$

This operation will on universe terms consist of the type of point terms in the point context Ω, while on metatheoretic sort functions, it will return a function with constructor of the applied term:

$$\begin{aligned} \mathsf{con}'_{\mathsf{s}}\,a &:\equiv \mathsf{Tm}_{\mathsf{p}}\,\Omega\,(\mathsf{El}\,a) && \textit{for } a : \mathsf{Tm}_{\mathsf{s}}\,\Gamma_s\,\mathsf{U} \textit{ and} \\ \mathsf{con}'_{\mathsf{s}}\,t &:\equiv \lambda\tau.\mathsf{con}'_{\mathsf{s}}\,(t \mathbin{\hat{@}} \tau) && \textit{for } t : \mathsf{Tm}_{\mathsf{s}}\,\Omega_s\,(\hat{\Pi}_{\mathsf{s}}\,T\,A_s). \end{aligned}$$

This construction is already enough to give the sort constructor required in Theorem 9 by pinning the substitution to be the identity: $\mathsf{con}_{\mathsf{s}}\,\Omega :\equiv \mathsf{con}'_{\mathsf{s}}\,\mathsf{id}_{\Omega_s} : {\Omega_s}^{\mathsf{A}}$.

It is not immediately clear that the operation on substitutions and the operation on sort terms is well-behaved under the pullback along substitutions. We can, however, show that this is indeed the case:

▶ **Lemma 16** (Coherence of the Sort Constructor). *For all substitutions $\sigma : \mathsf{Sub}_{\mathsf{s}}\,\Omega_s\,\Gamma_s$ and $t : \mathsf{Tm}_{\mathsf{s}}\,\Gamma_s\,A_s$, taking a constructor of t pulled back along σ has the same effect as taking the term algebra over the context algebra generated by the constructor on σ, that is,* $t^{\mathsf{A}}\,(\mathsf{con}'_{\mathsf{s}}\,\sigma) = \mathsf{con}'_{\mathsf{s}}\,(t[\sigma])$.

Proof. Let us first do a case distinction on the substitution. If it is ϵ, then $\Gamma_s = \cdot$, and it is easy to see that there are no terms in the empty sort context. Thus, we can assume the substitution to be of the form $(\sigma,\, s)$. In this case, lets recurse on the term and see that

$$\begin{aligned}
(\mathsf{var}\,\mathsf{vz})^{\mathsf{A}}(\mathsf{con}'_{\mathsf{s}}\,(\sigma,\, s)) &= \mathsf{vz}^{\mathsf{A}}\,(\mathsf{con}'_{\mathsf{s}}\,\sigma, \mathsf{con}'_{\mathsf{s}}\,s) \\
&= \mathsf{con}'_{\mathsf{s}}\,s \\
&= \mathsf{con}'_{\mathsf{s}}\,(\mathsf{var}\,\mathsf{vz}[\sigma,\, s]),
\end{aligned}$$

$$\begin{aligned}
(\mathsf{var}\,(\mathsf{vs}\,x))^{\mathsf{A}}\,(\mathsf{con}'_{\mathsf{s}}\,(\sigma,\, s)) &= \mathsf{var}\,(\mathsf{vs}\,x)^{\mathsf{A}}(\mathsf{con}'_{\mathsf{s}}\,\sigma, \mathsf{con}'_{\mathsf{s}}\,s) \\
&= (\mathsf{var}\,x)^{\mathsf{A}}\,(\mathsf{con}'_{\mathsf{s}}\,\sigma) \\
&= \mathsf{con}'_{\mathsf{s}}\,(\mathsf{var}\,x[\sigma]) && \text{by induction} \\
&= \mathsf{con}'_{\mathsf{s}}\,(\mathsf{var}\,(\mathsf{vs}\,x)[\sigma,\, s]), && \text{and lastly}
\end{aligned}$$

$$\begin{aligned}
(f \mathbin{\hat{@}} \tau)^{\mathsf{A}}\,(\mathsf{con}'_{\mathsf{s}}\,(\sigma,\, s)) &= f^{\mathsf{A}}\,(\mathsf{con}'_{\mathsf{s}}\,(\sigma,\, s))\,\tau \\
&= \mathsf{con}'_{\mathsf{s}}\,(f[\sigma,\, s])\,\tau && \text{by induction} \\
&= \mathsf{con}'_{\mathsf{s}}\,((f\,\tau)[\sigma,\, s]) && \text{for } f : \hat{\Pi}_{\mathsf{s}}\,T\,B.
\end{aligned}$$ ◀

We can now use this lemma to do a trick with $\mathsf{con}\,\Omega$ similar to the trick we did for $\mathsf{con}_{\mathsf{s}}\,\Omega$: Replace the fixed point context with a variable one, together with a substitution from Ω, and define the constructor recursively on point types.

▶ **Definition 17** (The Point Constructor). *We define operations on point contexts and point terms, resulting in algebras, in the following form:*

$$\mathsf{con}' : \mathsf{Sub}_{\mathsf{p}}\,\Omega\,\Gamma \to \Gamma^{\mathsf{A}}\,(\mathsf{con}_{\mathsf{s}}\,\Omega) \qquad \mathsf{con}' : \mathsf{Tm}_{\mathsf{p}}\,\Omega\,A \to A^{\mathsf{A}}\,(\mathsf{con}_{\mathsf{s}}\,\Omega)$$

The operation on point substitutions is defined recursively by $\mathsf{con}'\,\epsilon :\equiv \star$ *and* $\mathsf{con}'\,(\sigma,\, t) :\equiv (\mathsf{con}'\,\sigma, \mathsf{con}'\,t)$*, whereas for point terms, note that if* $t : \mathsf{Tm}_{\mathsf{p}}\,\Omega\,(\mathsf{El}\,a)$*, then by Lemma 16*

$$t : \mathsf{con}'_{\mathsf{s}}\,a \equiv \mathsf{con}'_{\mathsf{s}}(a[\mathsf{id}]) = a^{\mathsf{A}}\,(\mathsf{con}'_{\mathsf{s}}\,\mathsf{id}_{\Omega_s}) \equiv (\mathsf{El}\,a)^{\mathsf{A}}\,(\mathsf{con}_{\mathsf{s}}\,\Omega),$$

which allows us to define the constructor operator by

$$\begin{aligned}
&\mathsf{con}'\,t :\equiv t && \text{for } t : \mathsf{Tm}_{\mathsf{p}}\,\Omega\,(\mathsf{El}\,a), \\
&\mathsf{con}'t \;:\equiv \lambda\tau.\,\mathsf{con}'\,(t \mathbin{\hat{@}} \tau) && \text{for } t : \mathsf{Tm}_{\mathsf{p}}\,\Omega\,(\hat{\Pi}_{\mathsf{p}}\,T\,A), \text{ and} \\
&\mathsf{con}'t \;:\equiv \lambda u.\,\mathsf{con}'\,(t \mathbin{@} u) && \text{for } t : \mathsf{Tm}_{\mathsf{p}}\,\Omega\,(a \Rightarrow_{\mathsf{p}} A).
\end{aligned}$$

This concludes the definition of the constructors, since we can set, like for the sort constructor, $\mathsf{con}\,\Omega :\equiv \mathsf{con}'\,\mathsf{id}_\Omega : \Omega^{\mathsf{A}}\,(\mathsf{con}_{\mathsf{s}}\,\Omega)$*.*

Again, the construction comes with a property that makes it coherent under pulled back point terms. Analogously to Lemma 16, this coherence looks as follows:

▶ **Lemma 18** (Coherence of the Point Constructor). *For all point substitutions* $\sigma : \mathsf{Sub}_{\mathsf{p}}\,\Omega\,\Gamma$ *and point terms* $t : \mathsf{Tm}_{\mathsf{p}}\,\Gamma\,A$*, pulling back has the same effect as the point constructor as in* $t^{\mathsf{A}}\,(\mathsf{con}'\sigma) = \mathsf{con}'t[\sigma]$*.*

The proof is by induction on σ and t, and analogous to the one of Lemma 16, see Appendix B.

With the constructors defined, let us move on to the construction of the eliminator. Let us from now on fix displayed algebras $\omega^d_s : \Omega^{\mathsf{D}}_s\,(\mathsf{con}_{\mathsf{s}}\,\Omega)$ and $\omega^d : \Omega^{\mathsf{D}}\,\omega^d_s\,(\mathsf{con}\,\Omega)$. We will proceed in the same order as for the constructors and start by generalizing $\mathsf{elim}_{\mathsf{s}}\,\Omega\,\omega^d$ to arbitrary subcontexts of Ω by giving constructions on sort substitutions and sort terms.

▶ **Definition 19** (The Eliminator). *The generalized eliminator will take substitutions or sort terms to give sections of sort types or sort contexts, respectively:*

$$\begin{aligned}&\mathsf{elim_s}' : (\sigma : \mathsf{Sub_s}\,\Omega_s\,\Gamma_s) \to {\Gamma_s}^{\mathsf{S}}\,(\sigma^{\mathsf{A}}\,(\mathsf{con_s}\,\Omega))\,(\sigma^{\mathsf{D}}\,\omega_s^d)\\ &\mathsf{elim_s}' : (t : \mathsf{Tm_s}\,\Omega_s\,A_s)\ \to {A_s}^{\mathsf{S}}\,(t^{\mathsf{A}}\,(\mathsf{con_s}\,\Omega))\,(t^{\mathsf{D}}\,\omega_s^d)\end{aligned}$$

The first rule is defined by recursion using the second construction as usual: $\mathsf{elim_s}'\,\epsilon :\equiv \star$ *and* $\mathsf{elim_s}'\,(\sigma, t) :\equiv \big(\mathsf{elim_s}'\,\sigma, \mathsf{elim_s}'\,t\big)$*. For the sort terms, we observe that, by Lemmas 16 and 18, for* $a : \mathsf{Tm_s}\,\Omega_s\,\mathsf{U}$ *and* $t : a^{\mathsf{A}}\,(\mathsf{con_s}\,\Omega)$ *we have* $\mathsf{U}^{\mathsf{S}}(a^{\mathsf{D}}\,\omega_s^d\,(t^{\mathsf{A}}\,(\mathsf{con}\,\Omega)) = a^{\mathsf{D}}\,\omega_s^d\,t$*, and thus we can set, disregarding transports,*

$$\begin{aligned}&\mathsf{elim_s}'\,a :\equiv \lambda t.\, t^{\mathsf{D}}\,\omega^d && \textit{for } a : \mathsf{Tm_s}\,\Omega_s\,\mathsf{U} \textit{ and}\\ &\mathsf{elim_s}'\,t\ :\equiv \lambda\tau.\,\mathsf{elim_s}'\,(t\,\hat{@}\,\tau) && \textit{for } t : \mathsf{Tm_s}\,\Omega_s\,(\hat{\Pi}_{\mathsf{s}}\,T\,A_s).\end{aligned}$$

Now we set $\mathsf{elim_s}\,\Omega\,\omega^d :\equiv \mathsf{elim_s}'\,\mathsf{id}_{\Omega_s}$*.*

Similar to Lemma 16, these definitions are coherent in the following form:

▶ **Lemma 20.** *Given a sort substitution* $\sigma : \mathsf{Sub_s}\,\Omega_s\,\Gamma_s$ *and a sort term* $t : \mathsf{Tm_s}\,\Gamma_s\,A_s$*, the eliminator of a pulled back term is the section of the term, evaluated at the eliminator on a substitution:* $\mathsf{elim_s}'\,(t[\sigma]) = t^{\mathsf{S}}\,(\mathsf{elim_s}'\,\sigma)$*.*

Proof. The proof strategy is exactly the same as for Lemma 16. ◀

As a last step, we still need to prove the computation rules for the eliminator, consisting of a section a displayed algebra over a given point context. Consistent with Definition 15, we generalize them to arbitrary point substitutions and point terms.

▶ **Lemma 21** (Computation Rules). *We prove the computation rules for our eliminator* $\mathsf{elim_s}\,\Omega\,\omega^d$ *to be a section of subcontexts of* Ω *and of point terms in* Ω*:*

$$\begin{aligned}&\mathsf{elim}' : (\sigma : \mathsf{Sub_p}\,\Omega\,\Gamma) \to \Gamma^{\mathsf{S}}\,(\mathsf{elim_s}\,\Omega\,\omega^d)\,(\sigma^{A}\,(\mathsf{con}\,\Omega))\,(\sigma^{\mathsf{D}}\,\omega^d)\\ &\mathsf{elim}' : (t : \mathsf{Tm_p}\,\Omega\,A)\ \to A^{\mathsf{S}}\,(\mathsf{elim_s}\,\Omega\,\omega^d)\,(t^{A}\,(\mathsf{con}\,\Omega))\,(t^{\mathsf{D}}\,\omega^d)\end{aligned}$$

Proof. Using the elim' for terms, the one for substitutions can be implemented in a straightforward way by recursion on the point substitution: $\mathsf{elim}'\,\epsilon \equiv \star$ and $\mathsf{elim}'\,(\sigma, t) \equiv \big(\mathsf{elim}'\,\sigma, \mathsf{elim}' t\big)$.

We implement elim' for a term $t : \mathsf{Tm_p}\,\Omega\,A$ by case distinction on its type A. If $A = \mathsf{El}\,a$, we prove the following equality with the help of Lemmas 18 and 20:

$$a^{\mathsf{S}}\,(\mathsf{elim_s}'\,\mathsf{id}_{\Omega_s})\,(t^{\mathsf{A}}\,(\mathsf{con}\,\Omega)) = a^{\mathsf{S}}\,(\mathsf{elim_s}'\,\mathsf{id}_{\Omega_s})\,t = \mathsf{elim_s}'\,a\,t = t^{\mathsf{D}}\,\omega^d.$$

For the other two cases, we use the induction hypotheses:

$$\begin{aligned}&\mathsf{elim}' t :\equiv \lambda\tau.\,\mathsf{elim}'\,(t\,\hat{@}\,\tau)\ \text{ for } t : \mathsf{Tm_p}\,\Omega\,(\hat{\Pi}_{\mathsf{p}}\,T\,A), \text{ and}\\ &\mathsf{elim}' t :\equiv \lambda u.\,\mathsf{elim}'\,(t\,@\,u) \text{ for } t : \mathsf{Tm_p}\,\Omega\,(a \Rightarrow_{\mathsf{p}} A).\end{aligned}$$ ◀

Proof of Theorem 9. Lemma 21 completes the construction of the eliminator and setting $\mathsf{elim}\,\Omega\,\omega^d :\equiv \mathsf{elim}'\,\mathsf{id}_{\Omega}$ completes the existence proofs for of inductive families. ◀

5 Conclusions and further work

We defined a syntax of signatures for mutual inductive families which is very close to the usual way of specifying such types in proof assistants: by a list of sorts and then a list of constructors. We defined semantics for these signatures and showed how to derive the initial algebra for any signature just by using the syntax of signatures. The syntax of signatures was only given by normal forms, hence we could encode them as indexed W-types. Thus we obtained a formalisation of the reduction of mutual inductive families to indexed W-types. The lack of such a proof in the literature might be due to the absence of direct, convenient descriptions of mutual inductive types.

In the future, we would like to investigate how to integrate the theory of signatures into the core language of a proof assistant and how generic programming can be performed by induction on signatures, e.g. proving injectivity, disjointness of constructors, or decidability of equality. Also, we would like to extend the theory of signatures and its semantics with infinitary constructors. Currently, infinitely branching trees cannot be described as a signature, and as a consequence, the theory of signatures itself cannot be described as a signature.

References

1 Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers — constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.

2 Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type and scope safe universe of syntaxes with binding: Their semantics and proofs. *Proc. ACM Program. Lang.*, 2(ICFP):90:1–90:30, July 2018. doi:10.1145/3236785.

3 Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, pages 293–310, Cham, 2018. Springer International Publishing.

4 Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed containers. *Journal of Functional Programming*, 25, 2015. doi:10.1017/S095679681500009X.

5 Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *ACM SIGPLAN Notices*, volume 51(1), pages 18–29. ACM, 2016.

6 Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*, PLPV '07, pages 57–68, New York, NY, USA, 2007. ACM. doi:10.1145/1292597.1292608.

7 Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic, 13th International Workshop, CSL '99*, pages 453–468, 1999.

8 Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards certified meta-programming with typed Template-Coq. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 20–39. Springer, 2018. doi:10.1007/978-3-319-94821-8_2.

9 Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The Coq proof assistant reference manual: Version 6.1, 1997.

10 Henning Basold and Herman Geuvers. Type Theory based on Dependent Inductive and Coinductive Types. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of LICS '16*, pages 327–336. ACM, 2016. doi:10.1145/2933575.2934514.

11 Henning Basold, Herman Geuvers, and Niels van der Weide. Higher inductive types in programming. *Journal of Universal Computer Science*, 23:63–88, 2017.

12 Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nord. J. Comput.*, 10(4):265–289, 2003.

13 Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free — parametricity for dependent types. *Journal of Functional Programming*, 22(02):107–152, 2012. doi:10.1017/S0956796812000056.

14 Evan Cavallo and Robert Harper. Higher inductive types in cubical computational type theory. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290314.

15 James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 3–14, New York, NY, USA, 2010. ACM. doi:10.1145/1863543.1863547.

16 Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015. doi:10.1007/978-3-319-21401-6_26.

17 Peter Dybjer. Inductive families. *Formal aspects of computing*, 6(4):440–465, 1994.

18 Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65:525–549, 2000.

19 Peter Dybjer and Hugo Moeneclaey. Finitary higher inductive types in the groupoid model. *Electronic Notes in Theoretical Computer Science*, 336:119–134, 2018. The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII). doi:10.1016/j.entcs.2018.03.019.

20 Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In *Typed Lambda Calculi and Applications, volume 1581 of Lecture Notes in Computer Science*, pages 129–146. Springer, 1999.

21 Peter Dybjer and Anton Setzer. Indexed induction-recursion. *J. Log. Algebr. Program.*, 66(1):1–49, 2006. doi:10.1016/j.jlap.2005.07.001.

22 Marcelo Fiore, Andrew M. Pitts, and S. C. Steenkamp. Constructing Infinitary Quotient-Inductive Types. *arXiv e-prints*, page arXiv:1911.06899, November 2019. arXiv:1911.06899.

23 Martin Hofmann. Conservativity of equality reflection over intensional type theory. In *TYPES 95*, pages 153–164, 1995.

24 Martin Hofmann. Syntax and semantics of dependent types. In *Semantics and Logics of Computation*, pages 79–130. Cambridge University Press, 1997.

25 Ambrus Kaposi and András Kovács. A syntax for higher inductive-inductive types. In *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018)*, volume 108 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:18, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.FSCD.2018.20.

26 Ambrus Kaposi and András Kovács. Signatures and induction principles for higher inductive-inductive types. *arXiv preprint arXiv:1902.00297*, 2019.

27 Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. Constructing quotient inductive-inductive types. *Proceedings of the ACM on Programming Languages*, 3(POPL):2, 2019.

28 Peter LeFanu Lumsdaine and Michael Shulman. Semantics of higher inductive types. *Mathematical Proceedings of the Cambridge Philosophical Society*, pages 1–50, 2019. doi:10.1017/S030500411900015X.

29 Fredrik Nordvall Forsberg. *Inductive-inductive definitions*. PhD thesis, Swansea University, 2013.

30 Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009.

31 Nicolas Oury. *Extensionality in the calculus of constructions*, pages 278–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. doi:10.1007/11541868_18.

32 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 (TLCA)*, number 664 in Lecture Notes in Computer Science, 1993.

33 Kent Petersson and Dan Synek. A set constructor for inductive sets in martin-löf's type theory. In David H. Pitt, David E. Rydeheard, Peter Dybjer, Andrew M. Pitts, and Axel Poigné, editors, *Category Theory and Computer Science, Manchester, UK, September 5-8, 1989, Proceedings*, volume 389 of *Lecture Notes in Computer Science*, pages 128–140. Springer, 1989. doi:10.1007/BFb0018349.

34 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

35 Théo Winterhalter, Matthieu Sozeau, and Nicolas Tabareau. Eliminating reflection from type theory. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 91–103. ACM, 2019. doi:10.1145/3293880.3294095.

A Deriving the Syntax from Indexed W-Types

We recall the notion of an indexed W-type.

▶ **Definition 22** (Indexed W-Types, [4]). *The indexed W-type* $\mathsf{IW}^{o,r}_{A,B} : I \to \mathsf{Set}$ *for input data*

$$\begin{aligned}
&I : \mathsf{Set} && \textit{(“index type”)}\\
&A : \mathsf{Set} && \textit{(“shapes”)}\\
&B : A \to \mathsf{Set} && \textit{(“positions”)}\\
&o : A \to I && \textit{(“output indices”) and}\\
&r : (a : A) \to B\,a \to I && \textit{(“recursive indices”)}
\end{aligned}$$

is the inductive type on the constructor of the following form:

$$\frac{a : A \qquad c : (b : B\,a) \to \mathsf{IW}^{o,r}_{A,B}\,(r\,a\,b)}{\mathsf{sup}\,a\,b : \mathsf{IW}^{o,r}_{A,B}\,(o\,a)}$$

admitting a dependent eliminator

$$\frac{C : \{i : I\} \to \mathsf{IW}^{o,r}_{A,B}\,i \to \mathsf{Set} \qquad \begin{array}{l} p : (a : A)\Big(c : (b : B\,a) \to \mathsf{IW}^{o,r}_{A,B}\,(r\,a\,b)\Big) \\ \qquad \to \Big((b : B\,a) \to C\,(c\,b)\Big) \to C\,(\mathsf{sup}\,a\,c)\end{array}}{\mathsf{elim}_{\mathsf{IW}}\,C\,p : (i : I)(w : \mathsf{IW}^{o,r}_{A,B}\,i) \to C\,w}$$

with the reduction rule

$$\mathsf{elim}_{\mathsf{IW}}\,C\,p\,(o\,a)\,(\mathsf{sup}\,a\,c) \equiv p\,a\,c\,(\lambda b.\,\mathsf{elim}_{\mathsf{IW}}\,C\,p\,(r\,a\,b)\,(c\,b)).$$

Using this definition of indexed W-types we now want to represent our extended syntax as such:

Table 1 The input data for the indexed W-types representing the internalized syntax for inductive families.

i	$I_i : \mathsf{Set}$	$A_i : \mathsf{Set}$	$B_i : A_i \to \mathsf{Set}$	$o_i : A_i \to I_i$	$r_i : (a : A_i) \to B_i\, a \to I_i$
$\mathsf{Ty_s}$	$\mathbf{1}$	$\mathbf{1}$ $+\mathsf{Set}$	$\mathsf{inl}\, \star \mapsto \mathbf{0}$ $\mathsf{inr}\, T \mapsto T$	$- \mapsto \star$	$- \mapsto \star$
$\mathsf{Con_s}$	$\mathbf{1}$	$\mathbf{1}$ $+\mathsf{Ty_s}$	$\mathsf{inl}\, \star \mapsto \mathbf{0}$ $\mathsf{inr}\, B \mapsto \mathbf{1}$	$- \mapsto \star$	$- \mapsto \star$
$\mathsf{Var_s}$	$\mathsf{Con_s}$	$\mathsf{Con_s}$ $+\mathsf{Con_s} \times \mathsf{Ty_s}$	$\mathsf{inl}\, \Gamma_s \mapsto \mathbf{0}$ $\mathsf{inr}\, (\Gamma_s, B') \mapsto \mathbf{1}$	$\mathsf{inl}\, \Gamma_s \mapsto (\Gamma_s,\, B)$ $\mathsf{inr}\, (\Gamma_s, B') \mapsto (\Gamma_s,\, B')$	$-$ $\mathsf{inr}\, (\Gamma_s, B')\, \star \mapsto \Gamma_s$
$\mathsf{Tm_s}\, \Gamma_s$	$\mathsf{Ty_s}$	$\mathsf{Ty_s}$ $+(T : \mathsf{Set}) \times (T \to \mathsf{Ty_s}) \times T$	$\mathsf{inl}\, B \mapsto \mathbf{0}$ $\mathsf{inr}\, - \mapsto \mathbf{1}$	$\mathsf{inl}\, B \mapsto \mathbf{0}$ $\mathsf{inr}\, (T, B, \tau) \mapsto B\, \tau$	$-$ $\mathsf{inr}\, (T, B, \tau)\, \star \mapsto \hat{\Pi}_{\mathsf{s}}\, T\, B$
$\mathsf{Sub_s}\, \Gamma_s$	$\mathbf{1}$	$\mathbf{1}$ $+(B : \mathsf{Ty_s}) \times \mathsf{Tm_s}\, \Gamma_s\, B$	$\mathsf{inl}\, \star \mapsto \mathbf{0}$ $\mathsf{inr}\, (B, t) \mapsto \mathbf{1}$	$- \mapsto \star$	$- \mapsto \star$
$\mathsf{Ty_p}\, \Gamma_s$	$\mathbf{1}$	$\mathsf{Tm_s}\, \Gamma\, \mathsf{U}$ $+\mathsf{Set}$ $+\mathsf{Tm_s} \Gamma\, \mathsf{U}$	$\mathsf{inl}\, a \mapsto \mathbf{0}$ $\mathsf{inr}\, (\mathsf{inl}\, \tau) \mapsto T$ $\mathsf{inr}\, (\mathsf{inr}\, a) \mapsto \mathbf{1}$	$- \mapsto \star$	$- \mapsto \star$
$\mathsf{Con_p}\, \Gamma_s$	$\mathbf{1}$	$\mathbf{1}$ $+\mathsf{Ty_s}\, \Gamma_s$	$\mathsf{inl}\, \star \mapsto \mathbf{0}$ $\mathsf{inr}\, A \mapsto \mathbf{1}$	$- \mapsto \star$	$- \mapsto \star$
$\mathsf{Var_p} - A$	$\mathsf{Con_p}\, \Gamma_s$	$\mathsf{Con_p}\, \Gamma_s$ $+\mathsf{Con_p}\, \Gamma_s \times \mathsf{Ty_p}\, \Gamma_s$	$\mathsf{inl}\, \Gamma \mapsto \mathbf{0}$ $\mathsf{inr}\, (\Gamma, A') \mapsto \mathbf{1}$	$\mathsf{inl}\, \Gamma \mapsto (\Gamma,\, A)$ $\mathsf{inr}\, (\Gamma, A') \mapsto (\Gamma,\, A')$	$-$ $\mathsf{inr}\, (\Gamma, A')\, \star \mapsto \Gamma$
$\mathsf{Tm_p}\, \Gamma$	$\mathsf{Ty_p}\, \Gamma_s$	$(A : \mathsf{Ty_p}\, \Gamma_s) \times \mathsf{Var_p}\, \Gamma\, A$	$\mathsf{inl}\, (A, v) \mapsto \mathbf{0}$ $\mathsf{inr}\, (\mathsf{inl}\, -) \mapsto \mathbf{2}$ $\mathsf{inr}\, (\mathsf{inr}\, -) \mapsto \mathbf{1}$	$\mathsf{inl}\, (A, v) \mapsto A$ $\mathsf{inr}\, (\mathsf{inl}\, (A, a)) \mapsto A$ $\mathsf{inr}\, (\mathsf{inr}\, (T, A, \tau)) \mapsto A\, \tau$	$-$ $\mathsf{inr}\, (\mathsf{inl}\, (A, a))\, 0 \mapsto (a \Rightarrow_{\mathsf{p}} A)$ $\mathsf{inr}\, (\mathsf{inl}\, (A, a))\, 1 \mapsto \mathsf{El}\, a$ $\mathsf{inr}\, (\mathsf{inr}\, (T, A, \tau))\, \star \mapsto \hat{\Pi}_{\mathsf{p}}\, T\, A$
$\mathsf{Sub_p}\, \Gamma$	$\mathbf{1}$	$\mathbf{1}$ $+(A : \mathsf{Ty_p}\, \Gamma_s) \times \mathsf{Tm_p}\, \Gamma\, A$	$\mathsf{inl}\, \star \mapsto \mathbf{0}$ $\mathsf{inr}\, (A, t) \mapsto \mathbf{1}$	$- \mapsto \star$	$- \mapsto \star$

▶ **Definition 23** (IF-Syntax as W-Types)**.** *We define the types defined in Definition 1, Definition 10, and Definition 13 as follows:*

$$
\begin{aligned}
\mathsf{Ty_s} &:\equiv \mathsf{IW}^{o_{\mathsf{Ty_s}}, r_{\mathsf{Ty_s}}}_{A_{\mathsf{Ty_s}}, B_{\mathsf{Ty_s}}}\ \star\,, \\
\mathsf{Con_s} &:\equiv \mathsf{IW}^{o_{\mathsf{Con_s}}, r_{\mathsf{Con_s}}}_{A_{\mathsf{Con_s}}, B_{\mathsf{Con_s}}}\ \star\,, \\
\mathsf{Var_s} - B &:\equiv \mathsf{IW}^{o_{\mathsf{Var_s}}\, B, r_{\mathsf{Var_s}}\, B}_{A_{\mathsf{Var_s}}\, B, B_{\mathsf{Var_s}}\, B}, \\
\mathsf{Tm_s}\, \Gamma_s &:\equiv \mathsf{IW}^{o_{\mathsf{Tm_s}}\, \Gamma_s, r_{\mathsf{Tm_s}}\, \Gamma_s}_{A_{\mathsf{Tm_s}}\, \Gamma_s, B_{\mathsf{Tm_s}}\, \Gamma_s}, \\
\mathsf{Sub_s}\, \Gamma_s &:\equiv \mathsf{IW}^{o_{\mathsf{Sub_s}}\, \Gamma_s, r_{\mathsf{Sub_s}}\, \Gamma_s}_{A_{\mathsf{Sub_s}}\, \Gamma_s, B_{\mathsf{Sub_s}}\, \Gamma_s}, \\
\mathsf{Ty_p}\, \Gamma_s &:\equiv \mathsf{IW}^{o_{\mathsf{Ty_p}}, r_{\mathsf{Ty_p}}}_{A_{\mathsf{Ty_p}}, B_{\mathsf{Ty_p}}}\ \star\,, \\
\mathsf{Con_p}\, \Gamma_s &:\equiv \mathsf{IW}^{o_{\mathsf{Con_p}}, r_{\mathsf{Con_p}}}_{A_{\mathsf{Con_p}}, B_{\mathsf{Con_p}}}\ \star\,, \\
\mathsf{Var_p} - A &:\equiv \mathsf{IW}^{o_{\mathsf{Var_p}}\, A, r_{\mathsf{Var_p}}\, A}_{A_{\mathsf{Var_p}}\, A, B_{\mathsf{Var_p}}\, A}, \\
\mathsf{Tm_p}\, \Gamma &:\equiv \mathsf{IW}^{o_{\mathsf{Tm_p}}\, \Gamma, r_{\mathsf{Tm_p}}\, \Gamma}_{A_{\mathsf{Tm_p}}\, \Gamma, B_{\mathsf{Tm_p}}\, \Gamma}, \\
\mathsf{Sub_p}\, \Gamma &:\equiv \mathsf{IW}^{o_{\mathsf{Sub_p}}\, \Gamma, r_{\mathsf{Sub_p}}\, \Gamma}_{A_{\mathsf{Sub_p}}\, \Gamma, B_{\mathsf{Sub_p}}\, \Gamma},
\end{aligned}
$$

where the respective input data for the indexed W-types is given in Table 1.

B Proof of Lemma 18

Proof. Repeating the strategy of the proof of Lemma 16, we again see that we can assume the substitution to be of an extended form $(\sigma,\ s)$, since there are no point terms in the empty point context. Now, by recursion on the term we see that

$$\begin{aligned}
(\mathsf{var}\,\mathsf{vz})^{\mathsf{A}}\,(\mathsf{con}'\,(\sigma,\ s)) &= (\mathsf{var}\,\mathsf{vz})^{\mathsf{A}}\,(\mathsf{con}'\,\sigma, \mathsf{con}'\,s) \\
&= \mathsf{con}'\,s \\
&= \mathsf{con}'\,(\mathsf{var}\,\mathsf{vz}[\sigma,\ s]),
\end{aligned}$$

$$\begin{aligned}
(\mathsf{var}(\mathsf{vs}\,x))^{\mathsf{A}}\,(\mathsf{con}'\,(\sigma,\ s)) &= (\mathsf{var}\,(\mathsf{vs}\,x))^{\mathsf{A}}\,(\mathsf{con}'\,\sigma, \mathsf{con}'\,s) \\
&= (\mathsf{var}\,x)^{\mathsf{A}}\,(\mathsf{con}'\,\sigma) \\
&= \mathsf{con}'\,(\mathsf{var}\,x[\sigma]) && \text{by induction} \\
&= \mathsf{con}'\,(\mathsf{var}\,(\mathsf{vs}\,x)[\sigma,\ s]),
\end{aligned}$$

$$\begin{aligned}
(t\,@\,u)^{\mathsf{A}}\,(\mathsf{con}'\,\sigma) &= t^{\mathsf{A}}\,(\mathsf{con}'\,\sigma)\ \big(u^{\mathsf{A}}\,(\mathsf{con}'\,\sigma)\big) \\
&= \mathsf{con}'\,(t[\sigma])\,(\mathsf{con}'\,(u[\sigma])) && \text{by induction} \\
&= \mathsf{con}'\,((t\,@\,u)[\sigma]), \text{ and}
\end{aligned}$$

$$\begin{aligned}
(t\,\hat{@}\,\tau)^{\mathsf{A}}\,(\mathsf{con}'\,\sigma) &= t^{\mathsf{A}}\,(\mathsf{con}'\,\sigma)\,\tau \\
&= \mathsf{con}'\,(t[\sigma])\,\tau && \text{by induction} \\
&= \mathsf{con}'\,((t\,\hat{@}\,\tau)[\sigma]).
\end{aligned}$$

◀

Towards Constructive Hybrid Semantics

Tim Lukas Diezel
FAU Erlangen-Nürnberg, Germany
Tim.L.Diezel@fau.de

Sergey Goncharov
FAU Erlangen-Nürnberg, Germany
Sergey.Goncharov@fau.de

Abstract

With *hybrid systems* becoming ever more pervasive, the underlying semantic challenges emerge in their entirety. The need for principled semantic foundations has been recognized previously in the case of discrete computation and discrete data, with subsequent implementations in programming languages and proof assistants. Hybrid systems, contrastingly, do not directly fit into the classical semantic paradigms due to the presence of quite specific "non-programmable" features, such as *Zeno behaviour* and the inherent indispensable reliance on a notion of continuous time. Here, we analyze the phenomenon of hybrid semantics from a constructive viewpoint. In doing so, we propose a monad-based semantics, generic over a given ordered monoid representing the time domain, hence abstracting from the monoid of constructive reals. We implement our construction as a *higher inductive-inductive type* in the recent cubical extension of the Agda proof assistant, significantly using state-of-the-art advances of *homotopy type theory*. We show that classically, i.e. under the *axiom of choice*, our construction admits a charaterization in terms of directed sequence completion.

2012 ACM Subject Classification Theory of computation → Categorical semantics; Theory of computation → Axiomatic semantics

Keywords and phrases Hybrid semantics, Elgot iteration, Homotopy type theory, Agda

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.24

Supplementary Material The recent version of our implementation can be found at `https://github.com/sergey-goncharov/hybrid-agda`.

Funding *Sergey Goncharov*: Support by Deutsche Forschungsgemeinschaft (DFG) under project GO 2161/1-2 is gratefully acknowledged.

Acknowledgements We would like to thank anonymous referees for carefully reading the text and making various points which contributed to improving presentation, and also to Stefan Milius for references on conservative completion.

1 Introduction

Hybrid semantics underlies *cyber-physical systems*, which are systems combining discrete communication and control with continuous evolution of physical (chemical, biological, neuromorphic, etc) processes, typically described by systems of (ordinary) differential equations. Semantic theories, rooted in the classical notion of computability, have been explored thoroughly in recent decades [24, 15, 18]. On the one hand, this has led to a better understanding of the corresponding discrete time systems, thus implicitly or explicitly contributing to improving their design. On the other hand, the results were used in developing verification environments and proof assistants. Hybrid semantics however, requires a massive reconsideration of the established approaches due to a number of features not covered standardly such as *Zeno behaviour*, i.e. the phenomenon of switching the discrete control state infinitely many times within a (physically) finite time interval. Moreover, hybrid computation is inherently intertwined with *reasoning*. For example, in order to describe the movement of a

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 24; pp. 24:1–24:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

ball, one has to be able to calculate the moments of collision of the ball with obstacles. From a practical point of view, a clear mathematical formulation of hybrid semantics is needed to be able to deal with verification challenges arising from safety critical systems such as self-driving cars, or aircrafts, or surgeon robots. A natural way to do this is to turn to the *computational*, and more specifically to the *constructive* side of the issue. We thus ask: *"How constructive is hybrid semantics?"* and *"What is the impact of the constructive viewpoint on the verification challenges that may or may not be solved?"*. In order to account for these questions we orient towards principled constructive environments such as *intensional type theory* and the corresponding implementations such as Coq and Agda.

The key concept in the heart of hybrid semantics are *real numbers*. The major existing approaches to hybrid systems rely on classical (non-constructive) real numbers, which are suitable for reasoning but not for computationally feasible operational semantics and not for computer representation of hybrid programs. Observations in a similar vein have been made recently [19, 5]. Here we assume a constructive notion of real numbers and constructive hybrid trajectories, i.e. time indexed sequences, based on them.

In a nutshell, we develop a generalization of the following denotational domain

$$\mathbb{R}_+ \times X \cup \bar{\mathbb{R}}_+ \tag{$\bigstar$}$$

for modelling *durations* of hybrid programs, where $\mathbb{R}_+$ stands for non-negative real numbers and $\bar{\mathbb{R}}_+$ stands for non-negative real numbers extended with infinity. This domain contains pairs (d, x) produced by computations that converge in time d and deliver a final value x, and possibly infinite durations $d\colon \bar{\mathbb{R}}_+$ corresponding to computations that diverge in time d (hence, not delivering any value). We generalize ($\bigstar$) in two directions: by replacing $\mathbb{R}_+$ with a general ordered monoid (to capture various notions of time), and by getting rid of non-constructive principles underlying the use of ($\bigstar$). Roughly, the latter direction is motivated by the fact that ($\bigstar$) does not adequately model iterative computation, unless the *law of excluded middle* is admitted – the fact, that ($\bigstar$) is presented as a disjoint union of denotation domains for convergent and divergent computations, would entail that program termination is *decidable* for this semantics. This phenomenon is known for the *partiality monad* [6, 4], which was developed as a replacement for the *maybe monad* $X \uplus \{\bot\}$, and in exactly the same sense our present construction replaces ($\bigstar$). Even more so, our construction is a direct generalization of the partiality monad construction from [4] and contains the latter as a special case. We dub the obtained monad $\tilde{\mathbf{L}}$ the *(generalized) duration monad* following previous work [9, 10] and keeping in touch with the idea that monoid elements represent time duration, even under a possibly far reaching generalization of the notion of time.

Despite technical similarity, our generalization raises issues which are degenerate and hence ineffective for the partiality monad. For one thing, the idea of characterizations in terms of complete partial orders stems from *domain theory*, where a suitable *information order* for the target denotational domain is assumed, with a bottom element $\bot$ representing divergence. In ($\bigstar$), the notions of divergence continuously range over extended real numbers $\bar{\mathbb{R}}_+$, in particular, we have the least (0) and the greatest (∞) notions of divergence. It turns out that one can select $\bot$ to be 0 and define the information order suitably by combining the standard idea from domain theory, that divergent computations are denotationally smaller than the convergent ones, with the comparison total order on real numbers.

The partiality monad thus becomes the duration monad over a trivial (i.e. single element) ordered monoid, i.e. a computation over it either finishes instantly or diverges. The partial order relation of the trivial ordered monoid is *decidable* in the type-theoretic sense (under the propositions-as-types discipline, the corresponding type satisfies excluded middle). Our paramount example of an ordered monoid though are non-negative reals $\mathbb{R}_+$ whose partial order relation is *not* decidable. The move from the decidable case to possibly undecidable ones has a significant impact on the choice of conditions for countable sequences whose least upper bounds can be computed. Here we stick to *directed sequences* rather than monotone sequences as before [4]. While in the decidable case both approaches are equivalent, in general not only directedness of sequences differs from monotonicity, but also the former comes in two forms: *extensional* and *intensional*. In the latter case, we demand for any two elements of a directed sequence to exist a concrete element greater than both, while in the former one we demand that such an element exists but it is not known which one it is. In type-theoretic terms the difference is expressed by means of *propositional truncation*.

In terms of category theory, we identify each $\widetilde{L}X$ as a suitable *free object* on X, or equivalently as an *initial object* in a certain comma category, in accordance with the previous construction of the partiality monad [4]. An alternative approach to constructing the latter presupposes the axiom of countable choice and essentially amounts to quotienting the space of monotone sequence over $X \uplus \{\bot\}$ by weak bisimilarity [6, 21]. This quotienting procedure can also be viewed as *ω-completion* of $X \uplus \{\bot\}$ regarded as a *flat domain* [17]. Under countable choice, both constructions are known to be equivalent. We establish an analogue of this equivalence but only in classical setting (under the full axiom of choice) and, again, replacing monotone sequences with directed ones. Remarkably, the completion procedure gives some insight into Zeno behaviour: when forming the completion of a partial order, Zeno behaviour contributes via duplication of least upper bounds that already exist in the original set because those cannot be detected via the completion procedure. For example, directed sequence completion (in contrast e.g. to *Cauchy completion*) cannot identify the sequence $1/2, 3/4, 7/8, \ldots$ and $1, 1, 1, \ldots$ To remedy this, we also introduce a coarsened version $\bar{\mathbf{L}}$ of $\widetilde{\mathbf{L}}$ by additionally demanding that all the originally existing least upper bounds must be kept intact. This corresponds to a modified completion procedure [14], which is dubbed *conservative completion* in [23].

We formalized our duration monad via higher inductive-inductive types in the Agda proof assistant (version 2.6.1-96d0dd0) using the version of the cubical library from Feb 6 2020. The recent version of our implementation can be found at `https://github.com/sergey-goncharov/hybrid-agda`.

Paper Organization

After short preliminaries in Section 2, we present our motivation in Section 3. In Section 4 we provide our main construction using complete monoid modules in categorical terms, and subsequenty characterize the obtained object in Section 5 under the assumption of the axiom of choice. In Section 6 we give the main construction of the generalized duration monad coping with Zeno behaviour. We discuss our formalization of both our constructions as higher inductive-inductive types in cubical Agda in Section 7. A conclusion and our plans of further work are given in Section 8.

2 Preliminaries

We work in an ambient theory of sets **Set** throughout, unless stated otherwise not assuming it to validate either excluded middle or any form of choice. For example **Set** can be understood as the type of HoTT types of h-level 2 [20]. Generally, we refer to the cited *HoTT book* as a comprehensive presentation of the underlying foundational realm for our results. An *ordered monoid* is a monoid $(\mathbb{M}, +, 0)$ together with a partial order $\leqslant$ on $\mathbb{M}$ such that 0 is the least element and $+$ is monotone on the right, i.e. $0 \leqslant a$ and $b \leqslant c \Rightarrow a + b \leqslant a + c$ for all $a, b, c \colon \mathbb{M}$.

We do not generally assume the dual monotonicity law (!) $a \leqslant b \Rightarrow a + c \leqslant b + c$. Even though we use the additive notation $+$ for generic monoids, we do not assume commutativity of $+$. By $\mathbb{R}$, $\mathbb{R}_+$ and $\bar{\mathbb{R}}_+$ we respectively denote (constructive) reals, non-negative reals and extended non-negative reals (i.e. $\mathbb{R}_+$ suitably extended with the infinite value ∞). For least upper bounds of families $(s_i)_i$, we interchangeably use the notation $\bigvee(s_i)_i$ and $\bigvee_i s_i$.

We assume basic familiarity with the concepts of category theory, specifically with *universal arrows* [13] in the form of *free objects* [2] (i.e. universal arrows for faithful functors). A monad $\mathbf{T}$ (on $\mathbf{Set}$) is determined by a *Kleisli triple* $(T, \eta, (-)^\star)$, consisting of a map $T\colon \mathbf{Set} \to \mathbf{Set}$, together with a $\mathbf{Set}$-indexed class of morphisms $\eta_X\colon X \to TX$ and *Kleisli lifting* sending each $f\colon X \to TY$ to $f^\star\colon TX \to TY$ and obeying *monad laws*: $\eta^\star = \mathsf{id}$, $f^\star\eta = f$, $(f^\star g)^\star = f^\star g^\star$ (it follows from this definition that T extends to a functor and η to a natural transformation). The map $f, g \mapsto f^\star g$ is called *Kleisli composition*. In program semantics, in order to interpret while-loops, one more specifically needs monads equipped with a notion of iteration. Monads from a suitable class called *(complete) Elgot monads* are required to support an *Elgot iteration* operator $(f\colon X \to T(Y \uplus X)) \mapsto (f^\dagger\colon X \to TY)$ subject to established laws of iteration [11]. We will continue to use bold capitals (e.g. $\mathbf{T}$) for monads over the corresponding endofunctors written as capital Romans (e.g. T).

3 Hybrid Semantics and Beyond

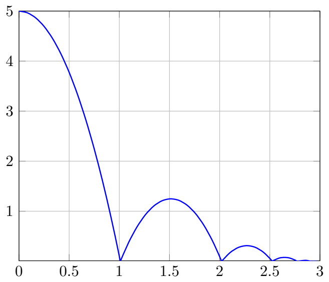
Let us briefly recall the hybrid language HybCore from previous work [10]. The grammar is as follows:

$$
\begin{aligned}
v, w &::= x \mid \star \mid \mathsf{true} \mid \mathsf{false} \mid (v, w) \mid f(v) && (f \in \Sigma)\\
p, q &::= \lceil v \rceil \mid (x, y) := p;\, q \mid x := p;\, q\\
&\quad\mid x := t.\, v \;\&\; w \mid \mathsf{if}\ v\ \mathsf{then}\ p\ \mathsf{else}\ q \mid x := p\ \mathsf{while}\ v\ \{q\}
\end{aligned}
$$

Here, x, y refer to variables, v, w refer to *values* and p, q refer to *programs* (as prescribed by the *fine-grain call-by-value* discipline [12]). Besides the standard Boolean values (**true** and **false**) and the canonical value $\star$ of the unit type, new values can be generated by Cartesian pairs (v, w), and by transforming other values with functions coming from a custom signature Σ. The latter is specifically meant to contain all the necessary parametrized time-dependent functions of type $X \times \mathbb{R}_+ \to Y$ representing continuous dynamics, and are regarded as atomic constructs by HybCore. A program either instantly returns a value ($\lceil v \rceil$), or is obtained by using one of the standard imperative style constructs, or by using the construct $x := t.\, v \;\&\; w$, which simultaneously abstracts v over the time variable t and restricts the domain of definiteness of the obtained function to the largest interval (either of the form $[0, d)$ or $[0, d]$) on which the predicate w holds throughout. A standard example is *bouncing ball*:

$$x := \lceil(5, 0)\rceil\ \mathsf{while\ true}\ \{(h, v) := (x := t.\, \mathit{ball}(x, t) \;\&\; \mathsf{fst}\, x \geqslant 0);\, \lceil(h, -0.5v)\rceil\}$$

Here, the height h and the velocity v are initially set to 5 and 0 correspondingly, and *ball* is a signature symbol representing the continuous dynamics of a flying ball. Each time the ball touches the ground v is reset to $-0.5v$ with -0.5 standing for the *damping factor*. This behaviour is repeated in the loop indefinitely. Graphically, we obtain the following *trajectory*



representing the dependency of the ball's hight in time. Among other things, this example demonstrates *Zeno behaviour* which is inherent to hybrid systems: infinitely many iterations occur in finite time.

We use HybCore as a motivation for the *duration semantics*, which assigns to any program a map $f\colon X \to \mathbb{R}_+ \times Y \cup \bar{\mathbb{R}}_+$ meaning that

- if $f(x) = (d, y)\colon \mathbb{R}_+ \times Y$ then the program yields the final value y in finite time d;
- if $f(x) = d \in \bar{\mathbb{R}}_+$ then the program diverges in finite or infinite time d (e.g. by exposing Zeno behaviour) and hence does not deliver any final value.

This semantics yields a monad $\mathbb{R}_+ \times (-) \cup \bar{\mathbb{R}}_+$, which is further generalized as follows.

▶ **Definition 1** (Monoid Module, Generalized Writer Monad [10])**.** *Given a monoid* $(\mathbb{M}, +, 0)$, *a* monoid module *is a set* $\mathbb{E}$ *equipped with a map* $\triangleright\colon \mathbb{M} \times \mathbb{E} \to \mathbb{E}$, *subject to the laws*

$$0 \triangleright e = e \qquad\qquad (m + n) \triangleright e = m \triangleright (n \triangleright e)$$

Every monoid-module pair $(\mathbb{M}, \mathbb{E})$ *induces the following monad* $\mathbf{T} = (T, \eta, (-)^\star)$ *which we call the* generalized writer monad*:* $T = \mathbb{M} \times (-) \cup \mathbb{E}$, $\eta_X(x) = (0, x)$, *and*

$$\begin{aligned} f^\star(m, x) &= (m + n, y) && \textit{where} \quad m\colon \mathbb{M},\ x\colon X,\ f(x) = (n, y)\colon \mathbb{M} \times Y \\ f^\star(m, x) &= m \triangleright e && \textit{where} \quad m\colon \mathbb{M},\ x\colon X,\ f(x) = e\colon \mathbb{E} \\ f^\star(e) &= e && \textit{where} \quad e\colon \mathbb{E} \end{aligned}$$

This yields a joint generalization of the writer monad ($\mathbb{E} = \varnothing$) *and the exception monad* ($\mathbb{M} = 1$)*.*

In order to interpret while-loops of HybCore w.r.t. the duration semantics, one needs to turn $\mathbb{R}_+ \times (-) \cup \bar{\mathbb{R}}_+$ into an Elgot monad, which is, however, impossible in a constructive setting because this would imply decidability of program divergence. This is analogous to the fact that the maybe-monad, i.e. the generalized writer monad over $\mathbb{M} = 1$, $\mathbb{E} = 1$ cannot generally serve as a model of partiality [4].

We next abstract from a concrete choice of $\mathbb{M}$ and fix the following stock of running examples for further reference.

▶ **Example 2** (Ordered Monoids)**.** Following are some ordered monoids on **Set**.

1. $(1, !, \star, \{(\star, \star)\})$ is a trivial ordered monoid over a one-element carrier $1 = \{\star\}$.
2. $(\mathbb{N}, +, 0, \leqslant)$ is an ordered monoid of natural numbers $\mathbb{N}$.
3. $(\mathbb{Q}_+, +, 0, \leqslant)$ is an ordered monoid of non-negative rational numbers $\mathbb{Q}_+$.
4. $(\mathbb{R}_+, +, 0, \leqslant)$ is an ordered monoid of non-negative real numbers $\mathbb{R}_+$.
5. $(A^\star, \cdot, \epsilon, \leqslant)$ is an ordered monoid of finite strings where ϵ is the empty string and $\leqslant$ is defined as follows: $u \leqslant v$ iff u is a prefix of v, i.e. there exists a w such that $uw = v$.

6. $(A^{[0,\mathbb{R}_+)}, {}^\frown, (0,!), \leqslant)$, the *monoid of (finite) trajectories* over a given set A, is defined as follows: $A^{[0,\mathbb{R}_+)} = \Sigma_{d\colon \mathbb{R}_+}\, [0,d) \to A$ is the set of *finite trajectories* valued on A; elements are pairs (d,e) where $d\colon \mathbb{R}_+$ and $e\colon [0,d) \to A$ is a *trajectory* of duration d; the concatenation operation ${}^\frown$ is defined as follows:

$$(d_1, e_1) {}^\frown (d_2, e_2) = (d_1 + d_2, \lambda t.\, \mathit{if}\ t < d_1\ \mathit{then}\ e_1(t)\ \mathit{else}\ e_2(t - d_1)).$$

The unit is $(0,!)$ where $!\colon [0,0) \to A$ is the *empty trajectory* and the relation $\leqslant$ is defined as follows: $(d_1,e_1) \leqslant (d_2,e_2)$ if $d_1 \leqslant d_2$ and $e_1(t) = e_2(t)$ for every $t\colon [0,d_1)$. Note that $A^\star$ is isomorphic to $\Sigma_{n\colon \mathbb{N}}\, A^n$, hence $A^{[0,\mathbb{R}_+)}$ can be understood as a counterpart of $A^\star$ obtained by changing the underlying notion of time from discrete ($\mathbb{N}$) to continuous ($\mathbb{R}_+$).
7. $(L, \vee, 0, \leqslant)$ is an ordered monoid for any join semilattice L with a bottom element 0.
8. $(\mathbb{M}_1 \times \mathbb{M}_2, +, (0_1, 0_2), \leqslant)$ is an ordered monoid, provided that so are $(\mathbb{M}_1, +_1, 0_1, \leqslant_1)$ and $(\mathbb{M}_2, +_2, 0_2, \leqslant_2)$; under these assumptions $(a_1,a_2) + (b_1,b_2) = (a_1 +_1 b_1, a_2 +_2 b_2)$ and $(a_1,a_2) \leqslant (b_1,b_2)$ if $a_1 \leqslant_1 b_1$ and $a_2 \leqslant_2 b_2$.

We intuitively regard ordered monoids as carriers of various (possibly exotic) notions of time. The examples **4.** and **6.** are of direct use for hybrid semantics: the former (*duration semantics*) corresponds to the semantics capturing only durations of programs; the latter (*evolution semantics*) captures both the durations and the intermediate states arranged in trajectories (cf. [10]). The monoids in **2.** and **3.** capture discrete and rational notions of time. In **5.**, the discrete time instants are labelled by the elements of A, in particular, $1^\star \cong \mathbb{N}$ corresponds to vacuous labelling. Both **5.** and **6.** illustrate our decision to make do without the left monotonicity law $a \leqslant b \Rightarrow a + c \leqslant b + c$, which is not satisfied by these examples.

We will treat an ordered monoid $\mathbb{M}$ as an input parameter to our constructions, while the corresponding monoid module $\mathbb{E}$ will universally arise from $\mathbb{M}$. In contrast to the generalized writer monad, $\mathbb{E}$ will no longer be a disjoint component of TX, however, the equation $T\emptyset = \mathbb{E}$ will have to remain true. In fact, the perspective we take is to consider the whole TX as a monoid module, regarded as an algebraic structure, and generated by X.

4 Complete Monoid Modules, Categorically

To obtain a constructively feasible replacement for the monad in Definition 1, we follow the idea used to define the partiality monad in intensional type theory [4], which can be abstractly summarized as follows:

1. Introduce a suitable category of algebras $\mathbf{Alg}_{\mathbf{T}}$ over $\mathbf{Set}$.
2. Obtain $\mathbf{T}$ from an adjunction $\begin{tikzcd} \mathbf{Set} \arrow[r, bend left] \arrow[r, phantom, "\perp"] & \mathbf{Alg}_{\mathbf{T}} \arrow[l, bend left] \end{tikzcd}$.

This scenario raises three main questions:

- How to specify the category of algebras $\mathbf{Alg}_{\mathbf{T}}$?
- How to construct $\mathbf{T}$ (i.e. prove that it exists)?
- How to ensure that the constructed monad $\mathbf{T}$ does indeed faithfully capture the intended semantics?

In this section we introduce a monad $\tilde{\mathbf{L}}$ parametrized by a generic ordered monoid $\mathbb{M}$. The construction of $\tilde{\mathbf{L}}$ is formulated in abstract category-theoretic terms, thus remaining agnostic about any specific choice of foundations. In Section 5, we then show that classically $\tilde{\mathbf{L}}$ can be characterized in terms of directed sequence completion and in Section 7 we discuss a formalization of $\tilde{\mathbf{L}}$ in the constructive realm of HoTT and cubical Agda.

As the first step, we identify the category of algebras $\mathbf{Alg}_{\tilde{\mathbf{L}}}$, which in our case are called *complete* $\mathbb{M}$*-modules*. Complete $\mathbb{M}$-modules are a proper generalization of *partiality algebras* [4] (corresponding to trivial $\mathbb{M}$).

▶ **Definition 3** (Complete $\mathbb{M}$-Modules). *An* ordered $\mathbb{M}$-module *w.r.t. an ordered monoid* $(\mathbb{M}, +, 0, \leqslant)$, *is an* $\mathbb{M}$*-module* $(\mathbb{E}, \rhd)$ *together with a partial order* $\sqsubseteq$ *and a least element* $\bot$, *such that* $\rhd$ *is monotone on the right and* $(- \rhd \bot)$ *is monotone, i.e.*

$$\frac{}{\bot \sqsubseteq x} \qquad \frac{x \sqsubseteq y}{a \rhd x \sqsubseteq a \rhd y} \qquad \frac{a \leqslant b}{a \rhd \bot \sqsubseteq b \rhd \bot}$$

We call the last property restricted left monotonicity*. An infinite sequence* $s_1, s_2, \ldots$ *is* monotone *if* $s_i \sqsubseteq s_{i+1}$ *for every* i *and* directed *if for every* i *and every* j *there exists* k *such that* $s_i \sqsubseteq s_k$ *and* $s_j \sqsubseteq s_k$*. Clearly, every monotone sequence is directed.*

An ordered $\mathbb{M}$*-module is* (directed ω-)complete *if for every directed sequence* $(s_i)_i$ *on* $\mathbb{E}$ *there is a least upper bound* $\bigsqcup_i s_i$ *and* $\rhd$ *is continuous on the right, i.e.*

$$\frac{}{s_i \sqsubseteq \bigsqcup_i s_i} \qquad \frac{\forall i.\, s_i \sqsubseteq x}{\bigsqcup_i s_i \sqsubseteq x} \qquad \frac{}{\bigsqcup_i a \rhd s_i \sqsubseteq a \rhd \bigsqcup_i s_i}$$

(the law $a \rhd \bigsqcup_i s_i \sqsubseteq \bigsqcup_i a \rhd s_i$ *is derivable).*

Note that any ordered monoid $\mathbb{M}$, which is complete as a partial order is a complete $\mathbb{M}$-module under $\rhd = +$, provided that $+$ is right continuous. Consider further examples of complete $\mathbb{M}$-modules.

▶ **Example 4.** Let us revisit Example **2.** For illustration purposes, let us assume here that **Set** is a classical set theory, e.g. ZFC.

- In **2.**–**4.**, $\bar{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$ is an ordered $\mathbb{N}$-module, $\bar{\mathbb{Q}}_+ = \mathbb{Q}_+ \cup \{\infty\}$ is an ordered $\mathbb{Q}_+$-module and $\bar{\mathbb{R}}_+ = \mathbb{R}_+ \cup \{\infty\}$ is an ordered $\mathbb{R}_+$-module respectively: $x \rhd y = x + y$ and $x \rhd \infty = \infty$, $\bot = 0$, $x \leqslant y \Rightarrow x \sqsubseteq y$ and $x \sqsubseteq \infty$ always. Both $\bar{\mathbb{Q}}_+$ and $\bar{\mathbb{R}}_+$ are also ordered $\mathbb{N}$-modules and $\bar{\mathbb{R}}_+$ is an ordered $\mathbb{Q}_+$-module. Now, $\bar{\mathbb{N}}$ is a complete $\mathbb{N}$-module and $\bar{\mathbb{R}}_+$ is a complete $\mathbb{R}_+$-module: $\bigsqcup_i s_i$ is the least upper bound of $(s_i)_i$ if s is bounded and ∞ otherwise. Also $\bar{\mathbb{R}}_+$ is a complete $\mathbb{N}$-module and a complete $\mathbb{Q}_+$-module, but $\bar{\mathbb{Q}}_+$ is not complete w.r.t. any monoid because it is not complete as a partial order.
- In **5.**, the set A^ω of infinite strings and the set $A^{\leqslant\omega} = A^\star \cup A^\omega$ of finite and infinite strings are both $A^\star$-modules under prefixing a string with a finite string. Moreover, $A^{\leqslant\omega}$ is a complete $A^\star$-module with the empty word as $\bot$ and least upper bounds calculated in the obvious way.
- In **6.** we defined the set $A^{[0,\mathbb{R}_+)} = \Sigma_{d\colon \mathbb{R}_+} [0, d) \to A$ of finite trajectories, which is of course an $A^{[0,\mathbb{R}_+)}$-module on itself. Analogously, let $A^{[0,\bar{\mathbb{R}}_+)} = \Sigma_{d\colon \bar{\mathbb{R}}_+} [0, d) \to A$ be the set of *finite or infinite trajectories* under the same operations, partially ordered in the same way and with $\rhd$ extended as follows: $(d_1, e_1) \rhd (d_2, e_2) = (d_1, e_1) \frown (d_2, e_2)$ if $d_1\colon \mathbb{R}_+$ and $(\infty, e_1) \rhd (d_2, e_2) = (\infty, e_1)$. Now, to calculate the least upper bound of a directed sequence $(d_1, e_1), (d_2, e_2), \ldots$, we first calculate $d = \bigvee_i d_i$ and then calculate $e\colon [0, d) \to A$ at each point $x\colon [0, d)$ by setting $e(x) = e_i(x)$ for sufficiently large i, which is guaranteed to exist by definition. Thus $A^{[0,\bar{\mathbb{R}}_+)}$ is a complete $A^{[0,\mathbb{R}_+)}$-module.

We emphasize the non-constructive flavor of the above examples: when forming a disjoint union such as $A^\star \cup A^\omega$, we make an explicit distinction between finite and infinite data, which cannot be realized constructively. For example, it is not possible to decide whether a string is finite or infinite on the basis of a given finite prefix whatever long.

Complete $\mathbb{M}$-modules form a category $\mathbf{Alg}_{\tilde{\mathbb{L}}}$ together with complete $\mathbb{M}$-module morphisms, which we define as follows.

▶ **Definition 5** (Complete $\mathbb{M}$-Module Morphisms). *Given two complete $\mathbb{M}$-modules $\mathbb{E}$ and $\mathbb{F}$, a* complete $\mathbb{M}$-module morphism *from $\mathbb{E}$ to $\mathbb{F}$ is a map $f\colon \mathbb{E} \to \mathbb{F}$ such that*

$$\frac{}{f(a \rhd x) = a \rhd f(x)} \qquad \frac{}{f(\bot) = \bot} \qquad \frac{x \sqsubseteq y}{f(x) \sqsubseteq f(y)} \qquad \frac{}{f\left(\bigsqcup_i s_i\right) = \bigsqcup_i f(s_i)}$$

We next instantiate the general categorical notion of *free object* [2] to complete $\mathbb{M}$-modules as follows.

▶ **Definition 6** (Free Complete $\mathbb{M}$-Modules). *A* free complete $\mathbb{M}$-module *on a set X consists of a complete $\mathbb{M}$-module $\widetilde{L}X$ and a map $\eta_X\colon X \to \widetilde{L}X$, such that for every map $f\colon X \to \mathbb{E}$ there exists a unique complete $\mathbb{M}$-module morphism $f^\star\colon \widetilde{L}X \to \mathbb{E}$ and the following diagram commutes*

$$\begin{tikzcd}
\widetilde{L}X \arrow[r, dotted, "f^\star"] & \mathbb{E} \\
X \arrow[u, "\eta_X"] \arrow[ur, "f"']
\end{tikzcd} \tag{1}$$

In different terms, $(\widetilde{L}X, \eta_X)$ is an initial object in the comma category $X \downarrow \eta_X$.

We proceed under the assumption that every $\widetilde{L}X$ exists and defer further details related to this issue until Section 7 where we discuss our construction of $\widetilde{L}X$ as a quotient inductive-inductive type. For now, we concentrate on checking if the object $\widetilde{L}X$ does indeed correctly capture the intended semantics. First, observe the following.

▶ **Theorem 7.**

1. *The forgetful functor $U\colon \mathbf{Alg}_{\widetilde{\mathbf{L}}} \to \mathbf{Set}$ has a left adjoint $F\colon \mathbf{Set} \to \mathbf{Alg}_{\widetilde{\mathbf{L}}}$ inducing a monad over $\widetilde{L} = UF$ with unit η and Kleisli lifting $(\text{–})^\star$ agreeing with* (1).
2. *$\widetilde{\mathbf{L}}$ is enriched over directed complete partial orders, and moreover, Kleisli composition is strict on both sides.*
3. *$\widetilde{\mathbf{L}}$ is an Elgot monad with the iteration operator $(f\colon X \to \widetilde{\mathbf{L}}(Y \uplus X))^\dagger$ calculated as a least fixed point of the map $[\eta, \text{–}]^\star f\colon (X \to \widetilde{L}Y) \to (X \to \widetilde{L}Y)$.*

Proof.

1. It is a standard category-theoretic fact [2] that existence of all free objects $\widetilde{L}X$ implies existence of the left adjoint F to the forgetful functor U. The arising monad $\widetilde{\mathbf{L}}$ is then as described.

2. Every $\widetilde{L}X$ carries a complete partial order by definition. The laws $f^\star(\bot) = \bot$, $p \sqsubseteq q \Rightarrow f^\star(p) \sqsubseteq f^\star(q)$ and $f^\star\left(\bigsqcup_i s_i\right) = \bigsqcup_i f^\star(s_i)$ follow from the fact that $f^\star$ is a complete $\mathbb{M}$-module morphism by definition. The dual properties of Kleisli composition amount to $f \sqsubseteq g \Rightarrow f^\star \sqsubseteq g^\star$ and $\bigsqcup_i f_i^\star = \left(\bigsqcup_i f_i\right)^\star$ assuming pointwise extension of the order on the function spaces. Assuming the former, the latter easily follows from the universal property (1) since $\left(\bigsqcup_i f_i^\star\right)\eta = \bigsqcup_i f_i^\star\,\eta = \bigsqcup_i f_i = \left(\bigsqcup_i f_i\right)^\star \eta$.

The fact that $f \sqsubseteq g$ implies $f^\star \sqsubseteq g^\star$ (for $f, g\colon X \to \widetilde{L}Y$) is by no means entailed by a generic category-theoretic argument. We show it as follows in slightly more generality for any $f, g\colon X \to \mathbb{E}$. For any complete $\mathbb{M}$-module $\mathbb{E}$, let $\mathbb{E}^\downarrow$ be the set of down-closed $\mathbb{E}$-submodules of $\mathbb{E}$. Then $\mathbb{E}^\downarrow$ is itself an $\mathbb{E}$-module under

$$a \rhd S = \{a \rhd x \mid x\colon S\}, \qquad \bot = \varnothing, \qquad S \sqsubseteq R \iff S \subseteq R, \qquad \bigsqcup_i S_i = \bigcup_i S_i.$$

Now, given $f, g \colon X \to \mathbb{E}$, let $f{\downarrow}$ and $g{\downarrow}$ be the pointwise principal ideals induced by f and g, e.g. $f{\downarrow}(x) = \{y \mid y \sqsubseteq f(x)\}$. Now, if $f \sqsubseteq g$ then $(f^\star{\downarrow} \cup g^\star{\downarrow})\,\eta = f{\downarrow} \cup g{\downarrow} = g{\downarrow} = g^\star{\downarrow}\,\eta$, hence $f^\star{\downarrow} \cup g^\star{\downarrow} = g^\star{\downarrow}$, which entails the desired inequation $f^\star \sqsubseteq g^\star$.

3. Finally, the fact that the above defined iteration operator turns $\widetilde{\mathbf{L}}$ into an Elgot monad was shown in previous work [11]. ◀

▶ Remark 8. Observe that $\mathbf{Alg}_{\widetilde{\mathbf{L}}}$ need not be the Eilenberg-Moore category of $\widetilde{\mathbf{L}}$. Already with $\mathbb{M} = 1$ we obtain as $\mathbf{Alg}_{\widetilde{\mathbf{L}}}$ the category of directed complete partial orders, which is known not to be monadic over $\mathbf{Set}$.

Using Theorem 7, we can immediately equip HybCore with a denotational semantics. The key clause, not entailed by the standard monad-based paradigm [16, 12], is while-loops, for which we put

$$\begin{aligned}&[\![x := p \ \mathbf{while}\ b\ \{q\}]\!](\sigma)\\ &\qquad = \big((\lambda\sigma, x.\ \mathit{if}\ [\![b]\!](\sigma,x)\ \mathit{then}\ (\widetilde{L}\,\mathsf{inr})[\![q]\!](\sigma,x)\ \mathit{else}\ \eta(\mathsf{inl}\,x))^\dagger\big)^\star(\sigma, [\![p]\!]\sigma)\end{aligned}$$

where $\sigma \colon [\![\Gamma]\!]$ interprets free variables from the variable context Γ, $[\![p]\!] \colon [\![\Gamma]\!] \to \widetilde{L}[\![X]\!]$ and $[\![q]\!] \colon [\![\Gamma]\!] \times [\![X]\!] \to \widetilde{L}[\![X]\!]$ are semantics of the involved programs, and $[\![b]\!] \colon [\![\Gamma]\!] \times [\![X]\!] \to 2$ is the semantics of the Boolean test b. By applying this to Example 2 (4) and to Example 2 (6), we obtain duration semantics and evolution semantics correspondingly (cf. [10]).

5 Complete Monoid Modules, Classically

Throughout this section we assume that $\mathbf{Set}$ is a classical set theory satisfying the axiom of choice (and hence also the law of excluded middle). Some portion of the presented developments can in fact be interpreted constructively or using more conservative principles such as the axiom of countable choice. However, we currently do not know how to fully rebase the following material on such more relaxed assumptions.

Our purpose here is to obtain a concrete description of the monad $\widetilde{\mathbf{L}}$ and make sure that the result is in agreement with the expectations. To that end we develop an alternative characterization of $\widetilde{L}X$ in terms of directed sequences.

▶ **Definition 9** (Directed Sequences). *A directed sequence is a countable infinite sequence $s_1, s_2, \ldots$ with the property that for every i and every j there is k such that $s_i \sqsubseteq s_k$ and $s_j \sqsubseteq s_k$. An (ω-)directed complete set is a partially ordered set in which every (ω-)directed sequence has a least upper bound.*

We start off by equipping the set $\mathbb{M}_X = \mathbb{M} \times (X \uplus \{\bot\})$ with the structure of an ordered $\mathbb{M}$-module as follows:

$$\frac{}{a \rhd_X (b, x) = (a + b, x)} \qquad \frac{}{(a, \mathsf{inl}\,x) \sqsubseteq_X (a, \mathsf{inl}\,x)} \qquad \frac{a \leqslant b}{(a, \mathsf{inr}\,\bot) \sqsubseteq_X (b, x)}$$

The idea is to use the elements of directed sequences $s \colon \mathbb{N} \to \mathbb{M}_X$ as progressively improving pieces of information about the final outcome of the underlying computational process. The elements of the form $(a, \mathsf{inl}\,x)$ are the maximal elements of this order, indicating that x is a final output of the process at time instant a. The elements of the form $(a, \mathsf{inr}\,\bot)$ represent *potential divergence* the time instant a, or later. The constraint $(a, \mathsf{inr}\,\bot) \sqsubseteq_X (b, x)$ with $a \leqslant b$ indicates that this potential divergence can still be resolved into successful termination over time. However, this need not happen. In particular, we allow for Zeno behaviour in the form of monotone sequences:

$$(a_1, \mathsf{inr}\,\bot) \sqsubseteq_X (a_2, \mathsf{inr}\,\bot) \sqsubseteq_X \ldots$$

with $a_1 < a_2 < \ldots$ where the set $\{a_1, a_2, \ldots\}$ has an upper bound in $\mathbb{M}$. Existence of such sequences, of course, depends on the properties of $\mathbb{M}$. For example, they do not exist for $\mathbb{M} = \mathbb{N}$, but they do exist for $\mathbb{R}_+$, e.g. $a_1 = 1/2$, $a_2 = 1/2 + 1/4$, $\ldots$

The following property is easy to verify.

▶ **Proposition 10.** *For any set X, $(\mathbb{M}_X, \rhd_X, (0, \mathsf{inr}\,\bot), \sqsubseteq_X)$ is an ordered $\mathbb{M}$-module.*

We next need to quotient the space of directed sequences $\mathbb{N} \to \mathbb{M}_X$ suitably to ensure that two sequences which tend to the same value are indistinguishable. To this end, we adapt the standard idea of chain completion from domain theory [17] by moving from monotone sequences to directed sequences.

▶ **Definition 11** (Directed Sequence Completion). *For any poset $(A, \leqslant)$, we define a preorder $\lesssim$ on directed sequences over A as follows:*

$$(s_i)_i \lesssim (t_i)_i \iff \forall i\colon \mathbb{N}.\, \exists j\colon \mathbb{N}.\, s_i \leqslant t_j.$$

This induces the equivalence $\sim$ on directed sequences as follows:

$$s \sim t \iff s \lesssim t \wedge t \lesssim s.$$

The directed sequence completion *$\widetilde{A}$ of A is the poset $(\widetilde{A}, \lessapprox)$, defined as follows: $\widetilde{A}$ is the quotient of the space of directed sequences over A by $\sim$, and $[s]_\sim \lessapprox [t]_\sim$ whenever $s \lesssim t$, where, as usual, we denote by $[s]_\sim \colon \widetilde{A}$ the equivalence class of $s\colon \mathbb{N} \to A$ in $\widetilde{A}$. Let us also agree to use the notation $[s_i]_i$ instead of $[(s_i)_i]_\sim$.*

▶ Remark 12 (Ideal Completion). An apparently more common, and classically equivalent, way (see e.g. [23, 1]) to introduce $\widetilde{A}$ is to use *ideal completion.* An ideal in A is a nonempty subset of A that is downward closed and directed. We could thus alternatively view $\widetilde{A}$ not as a set of equivalence classes but as a set of ideals of A generated by directed sequences. This switch of perspective is based on a representation of quotients by means of equivalence classes, which is uncomplicated for set theory but, of course, not for type theory.

The directed sequence completion of $\mathbb{M}_X$ yields a complete $\mathbb{M}$-module $\widetilde{\mathbb{M}}_X$. To show this we use the *Cantor pairing function* $\pi\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ with

$$\pi(x, y) = \frac{1}{2}(x + y)(x + y + 1) + x,$$

which witnesses an isomorphism between $\mathbb{N}$ and $\mathbb{N} \times \mathbb{N}$ whose inverse we denote $\pi^{-1} = (\pi_1^{-1}, \pi_2^{-1})\colon \mathbb{N} \to \mathbb{N} \times \mathbb{N}$.

▶ **Proposition 13.** *The quotient $(\widetilde{\mathbb{M}}_X, \rhd, \bot, \lessapprox, \widetilde{\bigvee})$ is a complete $\mathbb{M}$-module under*

$$a \rhd [s_i]_i = [a \rhd_X s_i]_i, \qquad \bot = [(0, \mathsf{inr}\,\bot)]_i, \qquad \overset{\approx}{\bigvee}_i\, [s_{i,j}]_j = [s_{\pi_1^{-1}(i), \pi_2^{-1}(i)}]_i.$$

Proof Sketch. If $(s_i)_i \sim (t_i)_i$ for directed sequences $(s_i)_i$ and $(t_i)_i$ then it is easy to see that both $(a \rhd_X s_i)_i$ and $(a \rhd_X t_i)_i$ are directed and $(a \rhd_X s_i)_i \sim (a \rhd_X t_i)_i$, hence $\rhd$ is correctly defined. Let us check that $\overset{\approx}{\bigvee}_i\, [s_{i,j}]_j$ is correctly defined. First, if $(s_{i,j})_j \sim (t_{i,j})_j$ for every i, then for all j there is j' such that $s_{i,j} \leqslant t_{i,j'}$ for every i, and conversely for all j there is j' such that $t_{i,j} \leqslant s_{i,j'}$ for every i, hence $(s_{\pi_1^{-1}(i),\pi_2^{-1}(i)})_i \sim (t_{\pi_1^{-1}(i),\pi_2^{-1}(i)})_i$. Therefore, the expression for $\tilde{\tilde{\bigvee}}_i\, [s_{i,j}]_j$ does not depend on the representatives of the equivalence classes $[s_{i,j}]_i$. To show correctness of the definition of $\tilde{\tilde{\bigvee}}_i\, [s_{i,j}]_j$, we are left to verify that $(s_{\pi_1^{-1}(i),\pi_2^{-1}(i)})_i$ is directed. Let $n, m \colon \mathbb{N}$ and construct a suitable k such that $s_{\pi_1^{-1}(n),\pi_2^{-1}(n)} \leqslant s_{\pi_1^{-1}(k),\pi_2^{-1}(k)}$ and $s_{\pi_1^{-1}(m),\pi_2^{-1}(m)} \leqslant s_{\pi_1^{-1}(k),\pi_2^{-1}(k)}$. Let $(n_1, n_2) = \pi^{-1}(n)$ and $(m_1, m_2) = \pi^{-1}(m)$. By assumptions, there is k_1 such that $(s_{n_1,j})_j \lesssim (s_{k_1,j})_j$ and $(s_{m_1,j})_j \lesssim (s_{k_1,j})_j$. Thus we obtain $n_2', m_2' \colon \mathbb{N}$ such that $s_{n_1,n_2} \leqslant s_{k_1,n_2'}$ and $s_{m_1,m_2} \leqslant s_{k_1,m_2'}$ and $k_2 \colon \mathbb{N}$ such that $s_{n_1,n_2} \leqslant s_{k_1,n_2'} \leqslant s_{k_1,k_2}$ and $s_{m_1,m_2} \leqslant s_{k_1,m_2'} \leqslant s_{k_1,k_2}$. Hence, we can take $k = \pi(k_1, k_2)$.

The axioms of complete $\mathbb{M}$-modules then transfer from $\mathbb{M}_X$ to $\widetilde{\mathbb{M}}_X$. ◀

The defined $\mathbb{M}$-module is in fact the free one on X.

▶ **Theorem 14.** *$((\widetilde{\mathbb{M}}_X, \triangleright, \bot, \tilde{\lesssim}, \tilde{\tilde{\bigvee}}), \eta)$ is the free complete $\mathbb{M}$-module on X with $\eta(x) = [(0, \mathsf{inl}\, x)]_i$.*

Proof Sketch. We fix a complete $\mathbb{M}$-module $(\mathbb{E}, \triangleright, \bot, \sqsubseteq, \bigsqcup)$ together with a map $f \colon X \to \mathbb{E}$. Our goal is to construct a unique complete $\mathbb{M}$-module morphism $f^\star \colon \widetilde{\mathbb{M}}_X \to \mathbb{E}$ satisfying $f^\star(\eta(x)) = f(x)$. To this purpose we define an auxiliary function $\ell \colon X \uplus \{\bot\} \to \mathbb{E}$ as follows:

$$\ell(\mathsf{inl}\, x) = f(x), \qquad\qquad \ell(\mathsf{inr}\, \bot) = \bot.$$

We then define $f^\star \colon \widetilde{\mathbb{M}}_X \to \mathbb{E}$ by putting $f^\star[(a_i, x_i)]_i = \bigsqcup_i a_i \triangleright \ell(x_i)$. The remaining technical work amounts to showing that the above definition of $f^\star$ is valid and that $f^\star$ is the unique morphism making (1) commute. ◀

Theorem 14 can be understood as a soundness and completeness property: it shows that the axioms of complete $\mathbb{M}$-modules are sound and complete over the models arising from $\mathbb{M}$ by directed sequence completion. Hence, we also obtain a more explicit description of the object $\tilde{L}X$.

▶ **Corollary 15.** *$\tilde{L}X$ and $\widetilde{\mathbb{M}}_X$ are isomorphic in the category of complete $\mathbb{M}$-modules.*

Let us write $\widetilde{\mathbb{M}}_\emptyset$ simply as $\widetilde{\mathbb{M}}$.

▶ **Proposition 16.** *$\tilde{L}X \cong \mathbb{M} \times X \cup \widetilde{\mathbb{M}}$.*

Proof Sketch. The argument relies on the law of excluded middle: there are precisely two kinds of directed sequences over $\mathbb{M}_X$:

1. those, which contain an element of the from $(a, \mathsf{inl}\, x)$;
2. those, which only contain elements of the form $(a, \mathsf{inr}\, \bot)$.

In the first case, the limit is $(a, \mathsf{inl}\, x)$ (a is uniquely determined, because $(a, \mathsf{inl}\, x)$ and $(a', \mathsf{inl}\, x')$ are incompatible, unless $a = a'$ and $x = x'$). In the latter case, the limit is $(c, \mathsf{inr}\, \bot)$ where where c is the least upper bound over all such a that $(a, \mathsf{inr}\, \bot)$ is in the sequence. This produces the dichotomy of $\tilde{L}X$ as $\mathbb{M} \times X \cup \widetilde{\mathbb{M}}$. ◀

We thus obtain agreement with Example 1. Let us again revisit Examples 2 and 4.

▶ **Example 17.** By Proposition 16, $\tilde{L}X$ is completely determined by the initial complete $\mathbb{M}$-module $\widetilde{\mathbb{M}}$. Hence we stick to the latter. We use the fact that classically directed sequence completion and monotone sequence completion coincide. Hence w.l.o.g. we view $\widetilde{\mathbb{M}}$ as a quotient of the space of monotone sequences.

1. Completion of natural numbers yields natural numbers extended with infinity: $\widetilde{\mathbb{N}} \cong \mathbb{N} \cup \{\infty\} = \bar{\mathbb{N}}$.
2. For non-negative rational numbers we obtain $\widetilde{\mathbb{Q}}_+ \cong \mathbb{Q}_+ \uplus (\mathbb{R}_+ \smallsetminus \{0\}) \uplus \{\infty\} \cong \bar{\mathbb{Q}}_+ \uplus (\mathbb{R}_+ \smallsetminus \{0\})$. That is, a monotone sequence $s_1 \leqslant s_2 \leqslant \ldots$ is either eventually constant, i.e. has a rational number as a least upper bound, or unbounded and hence its least upper bound is infinity ∞, or, finally, the sequence is Zeno, i.e. its supremum which can be rational or irrational is not reached. For example, the sequences

   $$1/2 \leqslant 3/4 \leqslant 7/8 \leqslant \ldots \qquad \text{and} \qquad 1 \leqslant 1 \leqslant 1 \leqslant \ldots$$

   both tend to 1, but neither directed sequence completion nor the free object construction (which agree, as we have seen) identify them.
3. Similarly, for real numbers: $\widetilde{\mathbb{R}}^+ \cong \mathbb{R}_+ \uplus (\mathbb{R}_+ \smallsetminus \{0\}) \uplus \{\infty\} \cong \bar{\mathbb{R}}_+ \uplus (\mathbb{R}_+ \smallsetminus \{0\})$. That is, except for 0 and ∞, every real number in $\widetilde{\mathbb{R}}^+$ is counted twice: as a Zeno value and as a non-Zeno value.
4. By completing finite strings we expectedly obtain finite and infinite strings: $\widetilde{A}^\star = A^\star \cup A^\omega = A^{\leqslant\omega}$.
5. Recall the monoid $A^{[0,\mathbb{R}_+)}$ of finite trajectories **6.** from Example 2 and the monoid $A^{[0,\bar{\mathbb{R}}_+)}$ of finite and infinite trajectories **4.** from Example 4. Then, analogously to the case of real numbers, we obtain $\widetilde{A^{[0,\mathbb{R}_+)}} \cong A^{[0,\bar{\mathbb{R}}_+)} \uplus (A^{[0,\mathbb{R}_+)} \smallsetminus \{(0,!)\})$.

6 Conservatively Complete Monoid Modules

The effect of duplicating values caused by Zeno behaviour makes definite computational sense, however it might also be undesirable. The way we defined the monad $\widetilde{\mathbf{L}}$ as a result of a universal construction allows us to remedy it easily.

▶ **Definition 18** (Conservatively Complete Monoid Modules). *A complete monoid* $\mathbb{M}$*-module is* conservatively complete *if it satisfies the following additional axiom: for every directed sequence* $(a_i)_i$ *in* $\mathbb{M}$*, such that the least upper bound* $\bigvee_i a_i$ *exists, the directed sequence* $(a_i \triangleright \bot)_i$ *has the least upper bound* $(\bigvee_i a_i) \triangleright \bot$*.*

Again, conservatively complete monoid $\mathbb{M}$-modules form a category $\mathbf{Alg}_{\bar{\mathbf{L}}}$ under the same morphisms as complete monoid modules (i.e. $\mathbf{Alg}_{\bar{\mathbf{L}}}$ is a full subcategory of $\mathbf{Alg}_{\widetilde{\mathbf{L}}}$). By replicating the previous construction of a free object $\bar{L}X$, we obtain a monad $\bar{\mathbf{L}}$, for which a complete analogue of Theorem 7 holds.

▶ **Theorem 19.**

1. *The forgetful functor* $U\colon \mathbf{Alg}_{\bar{\mathbf{L}}} \to \mathbf{Set}$ *has a left adjoint* $F\colon \mathbf{Set} \to \mathbf{Alg}_{\bar{\mathbf{L}}}$ *inducing a monad over* $\bar{L} = UF$ *with unit* η *and Kleisli lifting* $(-)^\star$*.*
2. $\bar{\mathbf{L}}$ *is enriched over directed complete partial orders, and moreover Kleisli composition is strict on both sides.*
3. $\bar{\mathbf{L}}$ *is an Elgot monad with the iteration operator* $(f\colon X \to \bar{L}(Y \uplus X))^\dagger$ *calculated as a least fixed point of the map* $[\eta, -]^\star f\colon (X \to \bar{L}Y) \to (X \to \bar{L}Y)$*.*

Again, we write $\bar{\mathbb{M}}$ instead of $\bar{\mathbb{M}}_\emptyset$.

Next, we would like to establish an analogue of Theorem 14 in order to be able to explicitly calculate $\bar{L}X$. To that end, we need to rebase our approach the construction on *conservative completion* of partial orders. In order to facilitate the corresponding construction [14], until the end of this section we impose the following further assumption which is satisfied by all our examples.

▶ **Assumption 20.** *Let* $(a_i)_i$ *and* $(b_i)_i$ *be directed sequences over* $\mathbb{M}$*.*

1. *If* $\bigvee_i b_i$ *exists then* $\bigvee_i a + b_i = a + \bigvee_i b_i$*.*
2. *If* $\bigvee_i a_i$ *exists and for every* i *there exists* j *such that* $a_i \leqslant b_j$ *then either* $\bigvee_i b_i$ *exists or* $\bigvee_i a_i \leqslant b_j$ *for some* j*.*

Recall the definitions of $\mathbb{M}_X$ and $\tilde{\mathbb{M}}_X$ from Section 5 and let $\bar{\mathbb{M}}_X$ be constructed in the same way as $\tilde{\mathbb{M}}_X$, but with the following additional clause added to the equivalence relation $\sim$:

$$(a_i, \mathsf{inr}\,\bot)_i \sim (a, \mathsf{inr}\,\bot)_i \qquad \text{whenever} \qquad a = \bigvee\nolimits_i a_i. \tag{2}$$

Modulo this change, the remaining definition of the complete $\mathbb{M}$-module structure on $\bar{\mathbb{M}}_X$ is the same as for $\tilde{\mathbb{M}}_X$. The following characterization is a counterpart of Theorem 14 and Proposition 16.

▶ **Theorem 21.** *In classical set theory,* $((\bar{\mathbb{M}}_X, \rhd, \bot, \lessapprox, \tilde{\tilde{\bigvee}}), \eta)$ *is the free conservatively complete* $\mathbb{M}$*-module on* X *with* $\eta(x) = [(0, \mathsf{inl}\, x)]_i$*, and* $\bar{\mathbb{M}}_X \cong \mathbb{M} \times X \cup \bar{\mathbb{M}}$*.*

Proof. The fact that under the above strengthening of $\sim$ the definition of the complete $\mathbb{M}$-module structure of $\bar{\mathbb{M}}_X$ remains valid relies on Assumption 20. For example, we need to make sure that $(a \rhd_X (b_i, \mathsf{inr}\,\bot))_i \sim (a \rhd_X (b, \mathsf{inr}\,\bot))_i$ for all $a, b_i \colon \mathbb{M}$ whenever $b = \bigvee_i b_i$. Equivalently, we need to show $(a + b_i, \mathsf{inr}\,\bot)_i \sim (a + b, \mathsf{inr}\,\bot)_i$, which follows from Assumption 20 (1). Similarly, correctness of $\tilde{\tilde{\bigvee}}$ requires Assumption 20 (2). ◀

▶ **Example 22.** Let us revisit some previous examples, again, assuming that **Set** is a classical set theory. Analogously to the case of $\tilde{\mathbf{L}}$, for $\bar{\mathbf{L}}$ we again obtain the property that $\bar{L}X$ is isomorphic to $\mathbb{M} \times X \cup \bar{\mathbb{M}}$, hence it suffices to consider $\bar{\mathbb{M}}$.

Note that the new law (2) is often ineffective, hence e.g. $\tilde{\mathbb{N}} = \bar{\mathbb{N}}$, and $\widetilde{A^\star} = \overline{A^\star}$. However, as expected, $\overline{\mathbb{R}_+}$ consists precisely of extended reals, i.e. $\overline{\mathbb{R}_+} = \mathbb{R}_+ \cup \{\infty\}$. Analogously, $\overline{\mathbb{Q}_+} = \overline{\mathbb{R}_+}$. Finally, for the monoid of trajectories, $\overline{A^{[0,\mathbb{R}_+)}} \cong A^{[0,\bar{\mathbb{R}}_+)}$.

7 Formalization in HoTT/Cubical Agda

We next embark on the details of our formalization of the material of Sections 4 and 6 using the means of homotopy type theory (HoTT). The latter is an extension of intensional Martin-Löf type theory (MLTT) obtained by interpreting types A as topological spaces, inhabitants of types $a \colon A$ as the corresponding points, and identity types $\mathsf{Id}_A(a, b)$ as spaces of continuous paths from $a \colon A$ to $b \colon A$ within A, subject to homotopy equivalence. We use the standard Agda notation $a \;\equiv\; b$ for the identity type $\mathsf{Id}_A(a, b)$ from now on.

Among various benefits and far reaching implications of HoTT, the critical feature we need here are higher inductive-inductive types (HIITs), which in particular enable construction of free objects in the style of category theory. We carry out our formalization in the recently emerged cubical extension [22] of the Agda proof assistant – while Agda is generally based on MLTT, the cubical extension adds full support of HoTT (in the form of *cubical type theory* [7]). As a result, cubical Agda provides a rather accurate way for designing machine checked proofs in the style of HoTT, and here we dim the distinction between HoTT and cubical Agda as much as possible. We note that we only involve particular HIITs, called *quotient inductive-inductive types* (QIITs) [3], which are a special case of HIITs – this specialization is completely explicit in cubical Agda, i.e. HIITs are available via native language primitives while QIITs are expressible as certain HIITs.

In our formalization we closely follow the previous work on constructing the partiality monad (which is in our setting the duration monad over the trivial monoid) and the subsequent formalization in cubical Agda by Danielsson [8].

Since Agda supports the *propositions-as-types* discipline, types can be read as propositions and the corresponding terms as proofs. Hence, universal $\forall$ and existential $\exists$ quantifiers have the same meaning as dependent product Π and dependent sum Σ operators correspondingly. This is a standard convention for Agda, which we apply to improve readability (for technical reasons we use slightly unusual syntax for existential quantification: $\exists[\ x\ ]\ \phi$ instead of $\exists x\ \phi$). Moreover, for the same purpose, we use the disjunction symbol $\vee$ for coproducts $\uplus$ and the conjunction symbol $\wedge$ for products $\times$. For example, the following self-explanatory Agda code

```
IsProp A = ∀ (x y : A) → x ≡ y
IsSet A = ∀ (x y : A) → IsProp (x ≡ y)
IsDec A = A ∨ ¬ A
```

defines correspondingly (mere) propositions, sets and decidable types.

A derivable facility of HoTT is the *propositional truncation* operator ‖_‖ sending any type A to the type $\|A\|$ obtained by quotienting A under the equality $x \equiv y$ for all $x\ y : A$, which is implemented as follows:

```
data ‖_‖ (A : Set ℓ) : Set ℓ where
  |_| : A → ‖ A ‖
  ‖‖-prop : IsProp ‖ A ‖
```

This provides a simple example of a quotient inductive type (QIT), i.e. an inductively defined set with constructors for equalities. Such types already generally go beyond MLTT. Next, given an infinite sequence $\sigma : \mathbb{N} \to A$ over a partially ordered set A, the following definitions

```
Inc σ = ∀ (n : ℕ) → σ n ⩽ σ (suc n)
Dir σ = ∀ (n m : ℕ) → ∃[ k ] (σ n ⩽ σ k ∧ σ m ⩽ σ k)
‖Dir‖ σ = ∀ (n m : ℕ) → ‖ ∃[ k ] (σ n ⩽ σ k ∧ σ m ⩽ σ k) ‖
```

identify *monotone* (increasing), *intensionally directed* and *extensionally directed* sequences correspondingly. The intensional version of directedness for any two numbers n and m produces a number k with an obvious property. The extensional version ensures that such a number exists, without producing it. Observe that Inc, Dir and ‖Dir‖ are arranged by strength: if σ is monotone, then it is intensionally directed (k is the maximum of n and m), and if σ is intensionally directed then it is extensionally directed (by using |_| to forget the choice of k). Furthermore, observe that Inc, Dir and ‖Dir‖ induce the corresponding notions of Inc-complete, Dir-complete and ‖Dir‖-complete partial orders, i.e. those partial orders in which all least upper bounds of the corresponding sequences exist. These notions are therefore arranged in the opposite direction: ‖Dir‖-completeness implies Dir-completeness, and the latter implies Inc-completeness.

The discrepancy between Dir and ‖Dir‖ can be clarified in terms of the axiom of countable choice, which can be expressed e.g. as follows:

```
ACω {ℓ} = ∀ (P : ℕ → Set ℓ) → (∀ n → ‖ P n ‖) → ‖ (∀ n → P n) ‖
```

This can be read as the statement that any proof of inhabitance of $P\ n$ for every n, can be converted into a proof of existence of a corresponding choice function. Intuitively, under ACω one should be able to convert an extensionally directed sequence to an intensionally directed one by successively pushing the truncation operator ‖_‖ upwards and then applying the elimination principle for ‖_‖. This indeed works, and in summary we have the following set of results:

▶ **Proposition 23.** *Let (a), (b) and (c) stand for completeness of a fixed set A w.r.t. ∥Dir∥, Dir and Inc correspondingly. Then*

- *(a) ⇒ (b) ⇒ (c);*
- *(b) ⇒ (a) under countable choice;*
- *(c) ⇒ (a) under the decidability of ⩽ on A (i.e. under ∀ (x y : A) → IsDec (x ⩽ y)).*

Proof Sketch. Consider the last clause, which is the one we did not discuss yet. The idea is based on Exercise 3.19. from the HoTT book [20], which can be formalized as follows:

```
restore-ωC : ∀ (P : ℕ → Set ℓ) → (∀ (n : ℕ) → IsDec (P n)) → ∥ ∃[ n ] P n ∥ → ∃[ n ] P n
```

That is, under decidability of all $P\,n$, the fact that there exists n satisfying P implies a constructive procedure for producing such an n. In our implementation such a procedure simply finds the first n that satisfies P, and that critically depends on the decidability assumption – otherwise the very concept "first n satisfying P" cannot be realized. Using restore-ωC, and the decidability assumption for ⩽, it is easy to select a monotone subsequence from any extensionally directed sequence and show coincidence of the corresponding least upper bounds. ◀

For decidable ⩽, ∥Dir∥-, Dir- and Inc-completeness are therefore equivalent, which is the case of the partiality monad. While the partiality monad is based on Inc-completeness, our implementation of $\tilde{\mathbf{L}}$ and $\bar{\mathbf{L}}$ is based on Dir-completeness, as we explain next.

For $\tilde{\mathbf{L}}$ we introduce an HIIT of a mutually dependent carrier (implicitly parametrized by an argument A of type Set (ℓ ⊔ ℓ′)) and a binary relation ⊑ on it, w.r.t. an ordered monoid 𝕄 with a carrier from Set ℓ and a partial order relation on 𝕄 from Set ℓ′:

```
data L̃ : Set (ℓ ⊔ ℓ′)
data _⊑_ : L̃ → L̃ → Set (ℓ ⊔ ℓ′)
```

The following forward reference asserts that $\tilde{L}$ will be a partial order

```
PO-⊑ : PartialOrder L̃
```

Then we introduce the constructors for $\tilde{L}$:

```
data L̃ where
  _▷_ : 𝕄 → L̃ → L̃
  ⊥ : L̃
  ⨆ : DirSeq PO-⊑ → L̃
  η : A → L̃
  ⊑-antisym : ∀ (x y : L̃) → x ⊑ y → y ⊑ x → x ≡ y
```

where DirSeq PO-⊑ is the type of intensionally directed sequences over $\tilde{L}$. The corresponding definition of _⊑_ is more technical, and we omit it here. This definition contains all the necessary axioms and in particular allows us to define PO-⊑. Moreover, it is asserted that IsProp (x ⊑ y) is inhabited, which implies that our carrier is a set (Theorem 7.2.2 of the HoTT book), i.e. the HIIT we define is indeed a QIIT. The most technically involved remaining part of the construction is the definition of the elimination principle together with the proof that it implies initiality of $\tilde{L}$ which in the categorical sense is the same as freeness of each $\tilde{L}X$ as an 𝕄-module on X. The construction of $\bar{L}$ is analogous – essentially we add the conservative completeness property as a new axiom to the definition of _⊑_.

Our definitions of $\tilde{L}$ and $\bar{L}$ are based on the intensional notion of directedness. It is currently not clear to us if those can also be based on the corresponding extensional notion, and whether this would bring any benefits over the present formalization. We regard this as an issue for further work. For another alternative, it would be perfectly possible to base $\widetilde{L}$ and $\bar{L}$ on Inc-completeness. We avoided that for a strategic reason: although the results of Section 5 currently hold under rather strong classicality assumptions, some of them must hold under weaker assumptions such as the axiom of countable choice. The challenges we would face then would be similar to those one faces when proving completeness of Cauchy reals. That standardly relies on a diagonalization argument roughly stating that a directed sequence of directed sequences can be converted to a directed sequence. This argument is constructively valid for Dir-completeness, but for Inc-completeness it seems to indispensably rely on decidability of inequality of the underlying set.

8 Conclusions and Further Work

We proposed a constructive formalization of hybrid semantics by combining ideas from category theory, type theory and domain theory and justified the results by an implementation in the Agda proof assistant extended with support of cubical type theory [7]. On the one hand, we closely followed the previous work on constructing the partiality monad, and on the other hand, complemented this construction with an explicit "time" dimension in the form of an ordered monoid. We thus reinforce the importance of quotient inductive-inductive types (QIITs) [3] which have previously been used for defining constructive counterparts of important semantic notions such as Cauchy reals and non-terminating computations so that even the principle of countable choice is avoided. Our analysis also identifies the importance of the notion of conservative completion [23], which seems to be little known in the computer science community, possibly because this notion only becomes relevant when dealing with non-discrete data types, once Zeno effects come into play. In contrast to the partiality monad case, our characterization in terms of directed sequence completion is only established under strong classicality assumptions and not with countable choice as the only additional axiom. We leave it as an important pending question for further work to check if our results to this effect can be improved. In a nutshell, a potential positive answer would amount to implementing the relevant parts of Section 5 in cubical Agda with the axiom of countable choice postulated.

Our work is motivated by a simple deterministic hybrid language HybCore for hybrid computation [10], which we now provided with a constructive semantics. As a next step, we are planning to extend HybCore with further features such as concurrency and non-determinism in a principled fashion. In fact, the majority of existing work on hybrid systems intertwine hybridness and non-determinism (even though, conceptually, these are independent computational effects). From a type-theoretic perspective, an interesting insight into relating partiality and nondeterminism is provided by Veltri [21], who proposed to build a constructive analogue of the countable powerset monad as a quotient inductive type (instead of a QIIT!), from which the partiality monad is then ingeniously carved out. The technical benefit of this approach is that it enables construction of the partiality monad in environments which do not support QIITs. We consider this approach more broadly as a way to relate partiality and non-determinism, and in particular, we plan to investigate its potential for constructing non-deterministic hybrid monads.

References

1 Samson Abramsky and Achim Jung. Domain theory. In *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Oxford University Press, 1994.

2 Jiří Adámek, Horst Herrlich, and George Strecker. *Abstract and concrete categories*. John Wiley & Sons Inc., New York, 1990.

3 Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, pages 293–310, Cham, 2018. Springer International Publishing.

4 Thorsten Altenkirch, Nils Danielsson, and Nicolai Kraus. Partiality, revisited - the partiality monad as a quotient inductive-inductive type. In Javier Esparza and Andrzej Murawski, editors, *Foundations of Software Science and Computation Structures, FOSSACS 2017*, volume 10203 of *LNCS*, pages 534–549, 2017.

5 Abhishek Anand and Ross Knepper. ROSCoq: Robots powered by constructive reals. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving*, pages 34–50, Cham, 2015. Springer International Publishing.

6 James Chapman, Tarmo Uustalu, and Niccolò Veltri. Quotienting the delay monad by weak bisimilarity. In *Theoretical Aspects of Computing, ICTAC 2015*, volume 9399 of *LNCS*, pages 110–125. Springer, 2015.

7 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

8 Nils Anders Danielsson. Code related to the paper "Partiality, Revisited: The Partiality Monad as a Quotient Inductive-Inductive Type". `http://www.cse.chalmers.se/~nad/listings/partiality-monad/`. Accessed: 2020-02-12.

9 Sergey Goncharov, Julian Jakob, and Renato Neves. A semantics for hybrid iteration. In Sven Schewe and Lijun Zhang, editors, *29th International Conference on Concurrency Theory (CONCUR 2018)*, LNCS. Springer, 2018.

10 Sergey Goncharov and Renato Neves. An adequate while-language for hybrid computation. In Ekaterina Komendantskaya, editor, *Proceedings of the 21th International Symposium on Principles and Practice of Declarative Programming, (PPDP 2019)*. ACM, 2019.

11 Sergey Goncharov, Lutz Schröder, Christoph Rauch, and Julian Jakob. Unguarded recursion on coinductive resumptions. *Logical Methods in Computer Science*, 14(3), 2018.

12 Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Inf. & Comp*, 185:2003, 2002.

13 Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, 1971.

14 Georg Markowsky. Chain-complete posets and directed sets with applications. *Algebra universalis*, 6:53–68, 1976.

15 John C. Mitchell. *Foundations of Programming Languages*. MIT Press, Cambridge, MA, USA, 1996.

16 Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93:55–92, 1991.

17 G. Plotkin. Post graduate lecture notes on advanced domain theory (incorporating the “Pisa notes”). Dept. of Computer Science, Univ. Edinburgh, 1981.

18 J. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

19 Benjamin Sherman, Luke Sciarappa, Adam Chlipala, and Michael Carbin. Computable decision making on the reals and other spaces: Via partiality and nondeterminism. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS ’18, pages 859–868, New York, NY, USA, 2018. ACM.

20 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

21 Niccoló Veltri. *A Type-Theoretical Study of Nontermination*. PhD thesis, Tallinn University of Technology, 2017.

22 Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *Proceedings of the ACM on Programming Languages*, 3(ICFP):87, 2019.

23 Wolfgang Wechler. *Universal Algebra for Computer Scientists*, volume 25 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1992.

24 G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1993.

A Omitted Proofs

A.1 Proof of Theorem 14

We fix a complete $\mathbb{M}$-module $(\mathbb{E}, \rhd, \bot, \sqsubseteq, \bigsqcup)$ together with a map $f\colon X \to \mathbb{E}$. Our goal is to construct a unique complete $\mathbb{M}$-module morphism $f^\star\colon \widetilde{\mathbb{M}}_X \to \mathbb{E}$ satisfying $f^\star(\eta(x)) = f(x)$. To this purpose we define an auxiliary function $\ell\colon X \uplus \{\bot\} \to \mathbb{E}$ as follows:

$$\ell(\mathsf{inl}\, x) = f(x), \qquad \ell(\mathsf{inr}\, \bot) = \bot.$$

Now, we define $f^\star\colon \widetilde{\mathbb{M}}_X \to \mathbb{E}$ as follows:

$$f^\star[(a_i, x_i)]_i = \bigsqcup\nolimits_i a_i \rhd \ell(x_i).$$

This definition is only valid if $(a_i \rhd \ell(x_i))_i$ is directed and $\bigsqcup_i a_i \rhd \ell(x_i)$ does not depend on the specific representatives $(a_i, x_i)_i$ of the corresponding equivalence classes. Both these properties require us to prove that $(a, x) \sqsubseteq_X (b, y)$ implies $a \rhd \ell(x) \sqsubseteq b \rhd \ell(y)$, which is shown as follows:

- if $(a, \mathsf{inl}\, x) \sqsubseteq_X (b, \mathsf{inl}\, y)$ then $(a, \mathsf{inl}\, x) = (b, \mathsf{inl}\, y)$ and thus $a \rhd \ell(\mathsf{inl}\, x) \sqsubseteq b \rhd \ell(\mathsf{inl}\, y)$;
- if $(a, \mathsf{inr}\, \bot) \sqsubseteq_X (b, \mathsf{inl}\, y)$ then $a \leqslant b$ and thus $a \rhd \ell(\mathsf{inr}\, \bot) = a \rhd \bot \sqsubseteq b \rhd \bot \sqsubseteq b \rhd f(y) = b \rhd \ell(\mathsf{inl}\, y)$ by the *restricted left monotonicity*, *least element* and *right monotonicity* axioms;
- if $(a, \mathsf{inr}\, \bot) \sqsubseteq_X (b, \mathsf{inr}\, \bot)$ then $a \leqslant b$ and thus $a \rhd \ell(\mathsf{inr}\, \bot) = a \rhd \bot \sqsubseteq b \rhd \bot = b \rhd \ell(\mathsf{inr}\, \bot)$ by *restricted left monotonicity.*

Next, let us show that $f^\star$ is a complete $\mathbb{M}$-module morphism, by verifying the corresponding preservation properties:

- $f^\star(a \rhd [(b_i, x_i)]_i) = a \rhd f^\star[(b_i, x_i)]_i$: the requisite calculation runs as follows:

$$\begin{aligned}
f^\star(a \rhd [(b_i, x_i)]_i) &= f^\star[a \rhd_X (b_i, x_i)]_i && \textit{(definition of } \rhd\textit{)}\\
&= f^\star[(a + b_i,\, x_i)]_i && \textit{(definition of } \rhd_X\textit{)}\\
&= \bigsqcup\nolimits_i (a + b_i) \rhd \ell(x_i) && \textit{(definition of } f^\star\textit{)}\\
&= \bigsqcup\nolimits_i a \rhd (b_i \rhd \ell(x_i)) && \textit{(monoid action)}\\
&= a \rhd \bigsqcup\nolimits_i b_i \rhd \ell(x_i) && \textit{(right continuity of } \rhd\textit{)}\\
&= a \rhd f^\star[(b_i, x_i)]_i. && \textit{(definition of } f^\star\textit{)}
\end{aligned}$$

- $f^\star(\bot) = \bot$: $f^\star(\bot) = f^\star[(0, \mathsf{inr}\, \bot)]_i = \bigsqcup_i 0 \rhd \ell(\mathsf{inr}\, \bot) = \bigsqcup_i 0 \rhd \bot = \bigsqcup_i \bot = \bot$ using the definition of monoidal action.

- $[(a_i,x_i)]_i \lessapprox [(b_i,y_i)]_i$ implies $f^\star[(a_i,x_i)]_i \sqsubseteq f^\star[(b_i,y_i)]_i$: If $[(a_i,x_i)]_i \lessapprox [(b_i,y_i)]_i$ then $(a_i,x_i)_i \lesssim (b_i,y_i)_i$ meaning that for every $i\colon \mathbb{N}$ there exists a $j\colon \mathbb{N}$ such that $(a_i,x_i) \sqsubseteq_X (b_j,y_j)$. This implies $a_i \triangleright \ell(x_i) \sqsubseteq b_j \triangleright \ell(y_j) \sqsubseteq \bigsqcup_i b_i \triangleright \ell(y_i)$ for every $i\colon \mathbb{N}$ by *upper bound* and thus $f^\star[(a_i,x_i)]_i = \bigsqcup_i a_i \triangleright \ell(x_i) \sqsubseteq \bigsqcup_i b_i \triangleright \ell(y_i) = f^\star[(b_i,y_i)]_i$ by *least upper bound.*
- $f^\star\left(\overset{\approx}{\bigvee}_i[(a_{i,j},x_{i,j})]_j\right) = \bigsqcup_i f^\star[(a_{i,j},x_{i,j})]_j$: This is obtained as follows.

$$\begin{aligned} f^\star\left(\overset{\approx}{\bigvee}_i[(a_{i,j},x_{i,j})]_j\right) &= f^\star[(a_{\pi_1^{-1}(i),\pi_2^{-1}(i)},x_{\pi_1^{-1}(i),\pi_2^{-1}(i)})]_i \\ &= \bigsqcup_i a_{\pi_1^{-1}(i),\pi_2^{-1}(i)} \triangleright \ell(x_{\pi_1^{-1}(i),\pi_2^{-1}(i)}) \\ &= \bigsqcup_i \bigsqcup_j a_{i,j} \triangleright \ell(x_{i,j}) \qquad (*) \\ &= \bigsqcup_i f^\star[(a_{i,j},x_{i,j})]_j. \end{aligned}$$

The only step that does not follow by definition is $(*)$, and we show it by *antisymmetry* of $\sqsubseteq$ as follows.

- $(\sqsubseteq)$: Let us fix $i\colon \mathbb{N}$ and let $k_1 = \pi_1^{-1}(i)$, $k_2 = \pi_2^{-1}(i)$, and then

$$\begin{aligned} a_{\pi_1^{-1}(i),\pi_2^{-1}(i)} \triangleright \ell(x_{\pi_1^{-1}(i),\pi_2^{-1}(i)}) &= a_{k_1,k_2} \triangleright \ell(x_{k_1,k_2}) \\ &\sqsubseteq \bigsqcup_j a_{k_1,j} \triangleright \ell(x_{k_1,j}) \\ &\sqsubseteq \bigsqcup_i \bigsqcup_j a_{i,j} \triangleright \ell(x_{i,j}). \end{aligned}$$

  Since i is arbitrary, by the *least upper bound* property, $\bigsqcup_i a_{\pi_1^{-1}(i),\pi_2^{-1}(i)} \triangleright \ell(x_{\pi_1^{-1}(i),\pi_2^{-1}(i)}) \sqsubseteq \bigsqcup_i \bigsqcup_j a_{i,j} \triangleright \ell(x_{i,j})$.
- $(\sqsupseteq)$: For all $i,j\colon \mathbb{N}$, $a_{i,j} \triangleright \ell(x_{i,j}) = a_{\pi^{-1}(\pi(i,j))} \triangleright \ell(x_{\pi^{-1}(\pi(i,j))}) \sqsubseteq \bigsqcup_i a_{\pi^{-1}(i)} \triangleright \ell(x_{\pi^{-1}(i)})$ and hence $\bigsqcup_i \bigsqcup_j a_{i,j} \triangleright \ell(x_{i,j}) \sqsubseteq \bigsqcup_i a_{\pi_1^{-1}(i),\pi_2^{-1}(i)} \triangleright \ell(x_{\pi_1^{-1}(i),\pi_2^{-1}(i)})$ by the *least upper bound* property.

Next we show commutativity of (1), i.e. that $f^\star(\eta(x)) = f(x)$, as follows: $f^\star(\eta(x)) = f^\star[(0,\mathsf{inl}\,x)]_i = \bigsqcup_i 0 \triangleright \ell(\mathsf{inl}\,x) = \bigsqcup_i 0 \triangleright f(x) = \bigsqcup_i f(x) = f(x)$ using the definition of monoidal action.

Finally, we show that the constructed morphism $f^\star$ is unique. That is, given another complete $\mathbb{M}$-module morphism $g\colon \widetilde{\mathbb{M}}_X \to \mathbb{E}$ satisfying $g(\eta(x)) = f(x)$, we show that $f^\star$ and g are equal. First, let $h\colon X \uplus \{\bot\} \to \widetilde{\mathbb{M}}_X$ be defined as $h(x) = [(0,x)]_i$. Note that $g(h(x)) = \ell(x)$ for any $x\colon X \uplus \{\bot\}$:

$$\begin{aligned} g(h(\mathsf{inl}\,x)) &= g[(0,\mathsf{inl}\,x)]_j = g(\eta(x)) = f(x) = \ell(\mathsf{inl}\,x), \\ g(h(\mathsf{inr}\,\bot)) &= g[(0,\mathsf{inr}\,\bot)]_j = g(\bot) = \bot = \ell(\mathsf{inr}\,\bot). \end{aligned}$$

The desired equation $f^\star = g$ is obtained as follows:

$$\begin{aligned} g[(a_i,x_i)]_i &= g\Big(\overset{\approx}{\bigvee}_i a_i \triangleright h(x_i)\Big) = \bigsqcup_i g(a_i \triangleright h(x_i)) = \bigsqcup_i a_i \triangleright g(h(x_i)) \\ &= \bigsqcup_i a_i \triangleright \ell(x_i) = f^\star[(a_i,x_i)]_i. \end{aligned}$$

Here, only the first step is not by definition. Let us show that, in fact, $[(a_i,x_i)]_i = \overset{\approx}{\bigvee}_i a_i \triangleright h(x_i)$. Note that $(a_i,x_i)_i \sim (a_{\pi_1^{-1}(i)},x_{\pi_1^{-1}(i)})_i$ basically because by definition $\pi_1^{-1}(i)$ can be made arbitrary large by choosing a suitable i. Therefore $[(a_i,x_i)]_i = [(a_{\pi_1^{-1}(i)},x_{\pi_1^{-1}(i)})]_i$ and we have

$$[(a_i,x_i)]_i = [(a_{\pi_1^{-1}(i)},x_{\pi_1^{-1}(i)})]_i = \overset{\approx}{\bigvee}_i[(a_i,x_i)]_j = \overset{\approx}{\bigvee}_i a_i \triangleright [(0,x_i)]_j = \overset{\approx}{\bigvee}_i a_i \triangleright h(x_i),$$

which completes the proof. ◀

A Gentzen-Style Monadic Translation of Gödel's System T

Chuangjie Xu
Ludwig-Maximilians-Universität München, Germany
http://cj-xu.github.io/
cj-xu@outlook.com

Abstract

We introduce a syntactic translation of Gödel's System T parametrized by a weak notion of a monad, and prove a corresponding fundamental theorem of logical relation. Our translation structurally corresponds to Gentzen's negative translation of classical logic. By instantiating the monad and the logical relation, we reveal the well-known properties and structures of T-definable functionals including majorizability, continuity and bar recursion. Our development has been formalized in the Agda proof assistant.

2012 ACM Subject Classification Theory of computation → Program analysis; Theory of computation → Type theory; Theory of computation → Proof theory

Keywords and phrases monadic translation, Gödel's System T, logical relation, negative translation, majorizability, continuity, bar recursion, Agda

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.25

Supplementary Material Agda development: https://github.com/cj-xu/GentzenTrans

Funding *Chuangjie Xu*: Alexander von Humboldt Foundation and LMUexcellent initiative.

Acknowledgements The author is grateful to Martín Escardó, Hajime Ishihara, Nils Köpp, Paulo Oliva, Thomas Powell, Benno van den Berg, Helmut Schwichtenberg for many fruitful discussions and the anonymous reviewers for their valuable suggestions and comments.

1 Introduction

Via a syntactic translation of Gödel's System T, Oliva and Steila [17] construct functionals of bar recursion whose terminating functional is given by a closed term in System T. The author adapts their method to compute moduli of (uniform) continuity of functions $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ that are definable in System T [27]. Inspired by the generalizations of negative translations which replace double negation by an arbitrary nucleus [8, 12, 25], we introduce a monadic translation of System T into itself which unifies those in [17, 27]. This monadic translation structurally corresponds to Gentzen's negative translation.

Our translation is parametrized by a monad-like structure, which we call a nucleus, but without the restriction of satisfying the monad laws. We adopt the standard technique of logical relations to show the soundness of the translation in the sense that each term of T is related to its translation. Because the translation is parametrized by a nucleus, we have to assume that the logical relation holds for the nucleus. Such a soundness theorem is an instance of the *fundamental theorem of logical relation* [21] stating that if a logical relation holds for all constants then so does it for all terms.

Monadic translations have been widely used for assigning semantics to impure languages. Our goal is instead to reveal properties enjoyed by terms of T and to extract witnesses of these properties. For this purpose, the nuclei we work with are not extensions of T, but just simple structures given by types and terms of T, so that the translation remains in T and the extracted witnesses are terms of T. The Gentzen-style translation looks simpler

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 25; pp. 25:1–25:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

than the other variants [19, 24]. But we demonstrate its power and elegance via its various applications including majorizability, (uniform) continuity and bar recursion. Of course these properties of T-definable functionals are well known [5, 7, 11, 13, 17, 19, 27]. The main contribution of the paper, however, is in obtaining these results in a single framework simply by choosing a suitable nucleus that satisfies the logical relation for the target property.

All the results in the paper are formalized in the Agda proof assistant [2], except the introductory section on negative translations of predicate logic. There are some differences in the Agda development. Firstly, it works with de Bruijn indices when representing the syntax of T to avoid handling variable names. Moreover, all the logical relations are defined between the Agda (or type-theoretic) interpretations of T-types. What we have proved in Agda is that the interpretation of any T-term is related to the one of its translation. In this way, we avoid dealing with the computation rules of T because they all hold judgmentally in the Agda interpretation. The Agda development is available at the author's GitHub page to which the link is given above the introduction.

1.1 Proof-theoretic translations

Recall that Gentzen's translation[1] simply places a double negation in front of atomic formulas, disjunctions and existential quantifiers [22]. One can replace double negation with a *nucleus*, that is, an endofunction j on formulas such that for any formulas A, B the following statements are provable:

$$A \to jA \qquad (A \to jB) \to jA \to jB \qquad (jA)[t/x] \leftrightarrow j(A[t/x]).$$

Nuclei are also known as *lax modalities* [1] and *strong monad* [8]. But in this paper we adopt the terminology and definition from [25] which brought the technical motivation to this work. Each nucleus determines a proof-theoretic translation of intuitionistic predicate logic IQL into itself, consisting of a formula translation $A \mapsto A^{\mathrm{G}}_j$ defined as follows

$$\begin{aligned}
(A \to B)^{\mathrm{G}}_j &:= A^{\mathrm{G}}_j \to B^{\mathrm{G}}_j & P^{\mathrm{G}}_j &:= jP \quad \text{for primitive } P\\
(A \wedge B)^{\mathrm{G}}_j &:= A^{\mathrm{G}}_j \wedge B^{\mathrm{G}}_j & (A \vee B)^{\mathrm{G}}_j &:= j(A^{\mathrm{G}}_j \vee B^{\mathrm{G}}_j)\\
(\forall x A)^{\mathrm{G}}_j &:= \forall x A^{\mathrm{G}}_j & (\exists x A)^{\mathrm{G}}_j &:= j\exists x A^{\mathrm{G}}_j
\end{aligned}$$

and a soundness theorem stating that IQL $\vdash A$ implies IQL $\vdash A^{\mathrm{G}}_j$. Working with different nuclei, one embed a logic system into another:

- if $jA = (A \to \bot) \to \bot$, then CQL $\vdash A$ implies MQL $\vdash A^{\mathrm{G}}_j$;
- if $jA = (A \to R) \to R$ for some predicate variable R, then CQL $\vdash A$ implies IQL $\vdash A^{\mathrm{G}}_j$;
- if $jA = A \vee \bot$, then IQL $\vdash A$ implies MQL $\vdash A^{\mathrm{G}}_j$;

where CQL stands for classical predicate logic and MQL for minimal predicate logic. These results are well-known (see *e.g.* [12, 25]) and various instances of the translation have been applied in term extraction (see *e.g.* [8, 14])

Under the viewpoint of the proofs-as-programs correspondence, our translation of Gödel's System T presented in Section 2 is exactly a term/program version of the above proof-theoretic translation on minimal propositional logic.

[1] Nowadays it is known as the Gödel-Gentzen negative translation. Gödel's translation places a double negation also in front of the clause for implication, which makes it different from Gentzen's one in affine logic [3].

1.2 Gödel's System T

Recall that the term language of Gödel's System T can be given by the following grammar

$$\begin{array}{ll} \text{Type} & \sigma, \tau ::= \mathbb{N} \mid \sigma \to \tau \\ \text{Term} & t, u ::= x \mid \lambda x^{\sigma}.t \mid tu \mid 0 \mid \mathsf{suc} \mid \mathsf{rec}_{\sigma} \end{array}$$

where $\mathbb{N}$ is the base type of natural numbers and $\sigma \to \tau$ the type of functions from σ to τ. A typing judgment takes the form $\Gamma \vdash t : \tau$, where Γ is a context (*i.e.* a list of distinct typed variables $x : \sigma$), t is a term and τ is a type. Here are the typing rules:

$$\Gamma, x : \sigma \vdash x : \sigma \qquad \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x^{\sigma}.t : \sigma \to \tau} \qquad \frac{\Gamma \vdash t : \sigma \to \tau \qquad \Gamma \vdash u : \sigma}{\Gamma \vdash tu : \tau}$$

$$\Gamma \vdash 0 : \mathbb{N} \qquad \Gamma \vdash \mathsf{suc} : \mathbb{N} \to \mathbb{N} \qquad \Gamma \vdash \mathsf{rec}_{\sigma} : \sigma \to (\mathbb{N} \to \sigma \to \sigma) \to \mathbb{N} \to \sigma$$

We call $\Gamma \vdash t : \tau$ a *well-typed* term if it is derivable. We may omit the context Γ and simply write $t : \tau$ or t^{τ} if it is unambiguous. When mentioning terms of T in the paper, we refer to only the well-typed ones. We often omit superscript and subscript types if they can be easily inferred, and may write:

- $\lambda x_1 x_2 \cdots x_n.t$ instead of $\lambda x_1.\lambda x_2.\cdots \lambda x_n.t$,
- $f(a_1, a_2, \cdots, a_n)$ instead of $(((f a_1) a_2) \cdots) a_n$,
- $n + 1$ instead of $\mathsf{suc}\, n$,
- τ^{σ} instead of $\sigma \to \tau$, and
- $f \circ g$ instead of $\lambda x.f(gx)$.

Using the primitive recursor, we can for instance define the function $\max : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ that returns the greater argument as follows:

$$\max := \mathsf{rec}_{\mathbb{N}\to\mathbb{N}}(\lambda n^{\mathbb{N}}.n,\ \lambda n^{\mathbb{N}} f^{\mathbb{N}\to\mathbb{N}}.\mathsf{rec}_{\mathbb{N}}(\mathsf{suc}\, n,\ \lambda m^{\mathbb{N}} g^{\mathbb{N}\to\mathbb{N}}.\mathsf{suc}(fm))).$$

One can easily verify that the usual defining equations of max

$$\max(0, n) = n \qquad \max(m, 0) = m \qquad \max(\mathsf{suc}\, m, \mathsf{suc}\, n) = \mathsf{suc}(\max(m, n))$$

hold using the computation rules of rec

$$\mathsf{rec}_{\sigma}(a, f, 0) = a \qquad \mathsf{rec}_{\sigma}(a, f, \mathsf{suc}\, n) = f(n, \mathsf{rec}_{\sigma}(a, f, n))$$

where $a : \sigma$ and $f : \mathbb{N} \to \sigma \to \sigma$. For the ease of understanding, we will use defining equations rather than T-terms involving rec in the paper.

2 A monadic translation of System T

Our syntactic translation of System T is parametrized by a nucleus, that is, a monad-like structure without the restriction of satisfying the monad laws.

▶ **Definition 1** (nuclei). *A* nucleus *relative to* T *is a triple* $(\mathsf{J}\mathbb{N}, \eta, \kappa)$ *consisting of a type* $\mathsf{J}\mathbb{N}$ *and two terms*

$$\eta : \mathbb{N} \to \mathsf{J}\mathbb{N} \qquad \kappa : (\mathbb{N} \to \mathsf{J}\mathbb{N}) \to \mathsf{J}\mathbb{N} \to \mathsf{J}\mathbb{N}$$

of System T.

Note that a nucleus is *not* an extension of T, but instead a simple structure given by a type and two terms of T. Therefore, our translation of any term of T remains in T rather than some monadic metatheory such as in [19, 24]. The simplest example is the identity nucleus where $\mathsf{J}\mathbb{N}$ is just $\mathbb{N}$ and η, κ are the identity functions of suitable types. More examples are available in Section 3. Though the first component of a nucleus is just a type, we denote it as $\mathsf{J}\mathbb{N}$ because in the generalized notion of a nucleus discussed in Section 4 it will be a map J on all the types of T.

We are now ready to construct a syntactic translation of T into itself:

▶ **Definition 2** (J-translation). *Given a nucleus $(\mathsf{J}\mathbb{N}, \eta, \kappa)$, we assign to each type ρ of T a type ρ^{J} as follows:*

$$\begin{aligned} \mathbb{N}^{\mathsf{J}} &:= \mathsf{J}\mathbb{N} \\ (\sigma \to \tau)^{\mathsf{J}} &:= \sigma^{\mathsf{J}} \to \tau^{\mathsf{J}}. \end{aligned}$$

Each term $\Gamma \vdash t : \rho$ is translated to a term $\Gamma^{\mathsf{J}} \vdash t^{\mathsf{J}} : \rho^{\mathsf{J}}$, where Γ^{J} is a new context assigning each $x : \sigma \in \Gamma$ to a fresh variable $x^{\mathsf{J}} : \sigma^{\mathsf{J}}$, and t^{J} is translated inductively as follows:

$$\begin{aligned} (x)^{\mathsf{J}} &:= x^{\mathsf{J}} & 0^{\mathsf{J}} &:= \eta 0 \\ (\lambda x.t)^{\mathsf{J}} &:= \lambda x^{\mathsf{J}}.t^{\mathsf{J}} & \mathsf{suc}^{\mathsf{J}} &:= \kappa(\eta \circ \mathsf{suc}) \\ (tu)^{\mathsf{J}} &:= t^{\mathsf{J}} u^{\mathsf{J}} & (\mathsf{rec}_\sigma)^{\mathsf{J}} &:= \lambda x^{\sigma^{\mathsf{J}}} f^{\mathsf{J}\mathbb{N} \to \sigma^{\mathsf{J}} \to \sigma^{\mathsf{J}}}.\mathrm{ke}_\sigma \left(\mathsf{rec}_{\sigma^{\mathsf{J}}} (x, f \circ \eta)\right) \end{aligned}$$

where $\mathrm{ke}_\sigma : (\mathbb{N} \to \sigma^{\mathsf{J}}) \to \mathsf{J}\mathbb{N} \to \sigma^{\mathsf{J}}$ is an extension of κ defined inductively on σ:

$$\begin{aligned} \mathrm{ke}_{\mathbb{N}} &:= \kappa \\ \mathrm{ke}_{\sigma \to \tau} &:= \lambda g^{\mathbb{N} \to \sigma^{\mathsf{J}} \to \tau^{\mathsf{J}}} a^{\mathsf{J}\mathbb{N}} x^{\sigma^{\mathsf{J}}}.\mathrm{ke}_\tau \left(\lambda n^{\mathbb{N}}.g(n, x), a\right). \end{aligned}$$

We often write J to denote the nucleus $(\mathsf{J}\mathbb{N}, \eta, \kappa)$ and call the above the J-translation *of* T.

Thanks to the inductive translation of function types into function types, the translation of the simply-typed-λ-calculus fragment of T is straightforward. There is no need of introducing a nonstandard, monadic notion of function application which plays an essential role in the other monadic translations [19, 24] as discussed in Section 4.

The more interesting part is the translation of the constants. Viewing η as a *unit* operator and κ as a *bind* operator in a monad J on $\mathbb{N}$ may reveal some intuition behind the translation of 0 and suc: It is natural to expect $\underline{n}^{\mathsf{J}} = \eta \underline{n}$ for each numeral $\underline{n} := \mathsf{suc}^n(0)$. This is indeed the case if the monad laws are satisfied, because $\kappa(\eta \circ -) : (\mathbb{N} \to \mathbb{N}) \to \mathsf{J}\mathbb{N} \to \mathsf{J}\mathbb{N}$ which is used to translate suc recovers exactly the "functoriality" of J. It is also natural to expect $(\mathsf{rec}_\sigma)^{\mathsf{J}}$ to preserve the computation rules, *i.e.*

$$(\mathsf{rec}_\sigma)^{\mathsf{J}}(x, f, 0^{\mathsf{J}}) = x \qquad (\mathsf{rec}_\sigma)^{\mathsf{J}}(x, f, (\mathsf{suc}\, n)^{\mathsf{J}}) = f(n^{\mathsf{J}}, (\mathsf{rec}_\sigma)^{\mathsf{J}}(x, f, n^{\mathsf{J}})).$$

A promising candidate of such $(\mathsf{rec}_\sigma)^{\mathsf{J}}(x, f) : \mathsf{J}\mathbb{N} \to \sigma^{\mathsf{J}}$ is $\mathsf{rec}_{\sigma^{\mathsf{J}}}(x, f \circ \eta) : \mathbb{N} \to \sigma^{\mathsf{J}}$. Hence, we extend κ to $\mathrm{ke}_\sigma : (\mathbb{N} \to \sigma^{\mathsf{J}}) \to \mathsf{J}\mathbb{N} \to \sigma^{\mathsf{J}}$ to complete the translation of rec_σ.

We adopt the standard technique of logical relations to show that the above translation is sound in the sense that each term of T is related to its translation[2]. Because the translation is parametrized by a nucleus, we have to assume that the logical relation holds for the nucleus.

[2] We owe the idea of proving a unified theorem of logical relation to Thomas Powell.

▶ **Theorem 3** (Fundamental Theorem of Logical Relation). *Let* $(\mathsf{J}\mathbb{N}, \eta, \kappa)$ *be a nucleus. Given a binary relation* $\mathrm{R}_{\mathbb{N}} \subseteq \mathbb{N} \times \mathsf{J}\mathbb{N}$ *between terms of* T*, we extend it to* $\mathrm{R}_\rho \subseteq \rho \times \rho^{\mathsf{J}}$ *for arbitrary type* ρ *of* T *by defining*

$$f \; \mathrm{R}_{\sigma\to\tau} \; g :\equiv \forall x^\sigma a^{\sigma^{\mathsf{J}}} \, (x \; \mathrm{R}_\sigma \; a \to fx \; \mathrm{R}_\tau \; ga) \, .$$

If $\mathrm{R}_{\mathbb{N}}$ *satisfies*

$$\forall n^{\mathbb{N}} \, (n \; \mathrm{R}_{\mathbb{N}} \; \eta n) \qquad \textit{and} \qquad \forall f^{\mathbb{N}\to\mathbb{N}} g^{\mathbb{N}\to\mathsf{J}\mathbb{N}} \left(\forall n^{\mathbb{N}} \, (fn \; \mathrm{R}_{\mathbb{N}} \; gn) \to f \; \mathrm{R}_{\mathbb{N}\to\mathbb{N}} \; \kappa g \right) \qquad (\dagger)$$

then $t \; \mathrm{R}_\rho \; t^{\mathsf{J}}$ *for any closed term* $t : \rho$ *of* T*.*

Proof. We prove a more general statement that

for any term $\Gamma \vdash t : \rho$ of T, if $\Gamma \; \mathrm{R} \; \Gamma^{\mathsf{J}}$ then $t \; \mathrm{R}_\rho \; t^{\mathsf{J}}$

where $(x_1 : \sigma_1, \ldots, x_n : \sigma_n) \; \mathrm{R} \; (x_1^{\mathsf{J}} : \sigma_1^{\mathsf{J}}, \ldots, x_n^{\mathsf{J}} : \sigma_n^{\mathsf{J}})$ stands for $x_1 \; \mathrm{R}_{\sigma_1} \; x_1^{\mathsf{J}} \wedge \ldots \wedge x_n \; \mathrm{R}_{\sigma_n} \; x_n^{\mathsf{J}}$, by structural induction over t.

- $t = x$. By the assumption $\Gamma \; \mathrm{R} \; \Gamma^{\mathsf{J}}$.
- $t = \lambda x.u$. Assume $\Gamma \; \mathrm{R} \; \Gamma^{\mathsf{J}}$ and $x \; \mathrm{R}_\sigma \; x^{\mathsf{J}}$. We have $u \; \mathrm{R} \; u^{\mathsf{J}}$ by induction hypothesis.
- $t = uv$. By induction hypothesis we have $u \; \mathrm{R}_{\sigma\to\tau} \; u^{\mathsf{J}}$ and $v \; \mathrm{R}_\sigma \; v^{\mathsf{J}}$. Then, by the definition of $\mathrm{R}_{\sigma\to\tau}$, we have $uv \; \mathrm{R}_\tau \; u^{\mathsf{J}} v^{\mathsf{J}}$.
- $t = 0$. By the assumption (†) of η.
- $t = \mathsf{suc}$. By the assumption (†) of η, we have $\mathsf{suc}(n) \; \mathrm{R}_{\mathbb{N}} \; \eta(\mathsf{suc}(n))$ for all $n : \mathbb{N}$. Then by the assumption (†) of κ, we have $\mathsf{suc} \; \mathrm{R}_{\mathbb{N}} \; \kappa(\eta \circ \mathsf{suc})$.
- $t = \mathsf{rec}$. We prove $\mathsf{rec} \; \mathrm{R} \; \mathsf{rec}^{\mathsf{J}}$ with the following claims:
  1. For any type σ of T, the term ke_σ preserves the logical relation in the following sense:

     $$\forall f^{\mathbb{N}\to\sigma} g^{\mathbb{N}\to\sigma^{\mathsf{J}}} \left(\forall n^{\mathbb{N}} \, (fn \; \mathrm{R}_\sigma \; gn) \to f \; \mathrm{R}_{\mathbb{N}\to\sigma} \; \mathrm{ke}_\sigma(g) \right) .$$

     Proof. By induction on σ. ◀
  2. For any x^σ and $y^{\sigma^{\mathsf{J}}}$ with $x \; \mathrm{R} \; y$, and any $f^{\mathbb{N}\to\sigma\to\sigma}$ and $g^{\mathsf{J}\mathbb{N}\to\sigma^{\mathsf{J}}\to\sigma^{\mathsf{J}}}$ with $f \; \mathrm{R} \; g$,

     $$\forall n^{\mathbb{N}} \, (\mathsf{rec}_\sigma(x, f, n) \; \mathrm{R}_\sigma \; \mathsf{rec}_{\sigma^{\mathsf{J}}}(y, g \circ \eta, n)) \, .$$

     Proof. By induction on n. ◀

  We get a proof of $\mathsf{rec} \; \mathrm{R} \; \mathsf{rec}^{\mathsf{J}}$ simply by applying (1) to (2). ◀

▶ Remark 4. The above proof can be carried out in the intuitionistic Heyting arithmetic in finite types HA^ω [23], with the theorem formulated as

if HA^ω proves (†) then, for each closed term t of T, HA^ω proves $t \; \mathrm{R} \; t^{\mathsf{J}}$.

So are all the results in Section 3. Hence, the verification system here can be HA^ω. We leave it unspecified in the theorem for several reasons. Firstly, we hope to study other properties whose verification may require a stronger system as in [19]. Moreover, what we have proved in the Agda formalization is a version of the theorem for the Agda embedding of System T, namely that the Agda interpretations of any T-term and its translation are related. But that is only an implementation choice as explained in the introduction.

3 Applications of the monadic translation

We now apply the above framework to reveal various properties and structures of T-definable functions including majorizability, (uniform) continuity and bar recursion. Each example consists of an algorithm to construct the desired structure given by the monadic translation, and a correctness proof of the algorithm given by the fundamental theorem of logical relation. For this, one only needs to choose a suitable nucleus that satisfies the logical relation for the target property.

3.1 Majorizability

Our first application is to recover Howard's majorizability proof of System T [11]. Majorizability plays an important role in models of higher-order calculi and more recently in the proof mining program [14]. Howard's majorizability relation extends the usual ordering $\leq$ on natural numbers to the one $\lhd_\rho$ on functionals of arbitrary finite type ρ in the same way as in Theorem 3. Specifically $\lhd_\rho$ is defined inductively on ρ as follows:

$$\begin{aligned} n \lhd_{\mathbb{N}} m &:= n \leq m \\ f \lhd_{\sigma\to\tau} g &:= \forall x^\sigma y^\sigma \, (x \lhd_\sigma y \to fx \lhd_\tau gy) \,. \end{aligned}$$

We say t is *majorized* by u if $t \lhd u$, and call u a *majorant* of t. Howard shows that each closed term of T is majorized by some closed term of T, which fits perfectly into our framework: Let us take $\mathsf{J}\mathbb{N} = \mathbb{N}$ and define $\eta : \mathbb{N} \to \mathbb{N}$ and $\kappa : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$ by

$$\eta(n) := n \qquad \begin{aligned} \kappa(g, 0) &:= g(0) \\ \kappa(g, n+1) &:= \max\left(\kappa(g,n), g(n+1)\right) . \end{aligned}$$

The max function can be defined in T using rec as shown in Section 1.2, and thus so is κ. Intuitively $\kappa(g, n)$ is the maximum of the values $g0, g1, \ldots, gn$. Therefore, it satisfies the following property:

▶ **Lemma 5.** *For any $g : \mathbb{N} \to \mathbb{N}$, we have $gm \leq \kappa(g, n)$ whenever $m \leq n$.*

Proof. By induction on n. If $n = 0$, we are done because m has to be 0. If $m \leq n + 1$, we have two cases: (i) If $m = n + 1$, then $g(n + 1) \leq \kappa(g, n + 1)$ by definition. (ii) If $m \leq n$, then $g(m) \leq \kappa(g, n) \leq \kappa(g, n + 1)$ by induction hypothesis and definition. ◀

▶ **Corollary 6.** *Each closed term $t : \rho$ of T is majorized by its translation t^{J}.*

Proof. We only need to check that the two conditions (†) are fulfilled. The first one holds because the ordering $\leq$ is reflexive. For the second, let us assume $\forall n\,(fn \leq gn)$ and $n \leq m$. We have $gn \leq \kappa(g, m)$ by Lemma 5, and thus $fn \leq \kappa(g, m)$ by the transitivity of $\leq$. ◀

We draw the reader's attention to this simple example also because the nucleus defined above does *not* satisfy the monad laws: Here η is the identity function on $\mathbb{N}$. If the left-identity law $\kappa(f, \eta x) = fx$ holds, then κ has to be the identity function on $\mathbb{N}^{\mathbb{N}}$, which is not the case.

3.2 Lifting to higher-order functionals

In the previous example, we extend a relation on natural numbers to arbitrary finite types and then show that the resulting logical relation holds for all terms of T. However, if one wants to prove a certain property P of functions $X \to \mathbb{N}$, the above syntactic method may not work directly, because the property P may not be captured by the inductively defined logical relation. Our monadic translation can serve a preliminary step to solve the problem by lifting natural numbers to functions $X \to \mathbb{N}$ so that the desired property P becomes the base case of the logical relation.

Let X be a type of T. Consider the nucleus $(\mathsf{J}\mathbb{N}, \eta, \kappa)$ with $\mathsf{J}\mathbb{N} = X \to \mathbb{N}$ and $\eta : \mathbb{N} \to \mathsf{J}\mathbb{N}$ and $\kappa : (\mathbb{N} \to \mathsf{J}\mathbb{N}) \to \mathsf{J}\mathbb{N} \to \mathsf{J}\mathbb{N}$ defined by

$$\eta(n) :\equiv \lambda x.n \qquad \kappa(g, f) :\equiv \lambda x.g(fx, x).$$

Clearly η maps a natural number n to a constant function with value n. The intuition of $\kappa(g, f) : X \to \mathbb{N}$ is the following: Given an input x, we have an index fx to get a function $g(fx)$ from the sequence g. Then we apply it to the input x to get the final value.

Given $x : X$, we define a logical relation $\mathrm{R}^x_\rho \subseteq \rho \times \rho^{\mathsf{J}}$ inductively as in Theorem 3:

$$\begin{aligned} n \;\mathrm{R}^x_{\mathbb{N}}\; f &:\equiv n = fx \\ g \;\mathrm{R}^x_{\sigma\to\tau}\; h &:\equiv \forall y^\sigma, z^{\sigma^{\mathsf{J}}} \left(y \;\mathrm{R}^x_\sigma\; z \to gy \;\mathrm{R}^x_\tau\; hz\right). \end{aligned}$$

Clearly the conditions (†) hold; thus by Theorem 3 we have $t \;\mathrm{R}^x_\rho\; t^{\mathsf{J}}$ for any closed term t of T. In particular, for any closed term $f : X \to \mathbb{N}$ of T, we have

$$\forall \Omega^{X^{\mathsf{J}}} \left(x \;\mathrm{R}^x_X\; \Omega \to fx = f^{\mathsf{J}}(\Omega, x)\right).$$

For some type X of T, we may be able to construct a closed term $\Omega : X^{\mathsf{J}}$ such that $x \;\mathrm{R}^x_X\; \Omega$ for all $x : X$, by unfolding the statement $x \;\mathrm{R}^x_X\; \Omega$. For example, if $X = \mathbb{N}^{\mathbb{N}}$, then $x \;\mathrm{R}^x_X\; \Omega$ is unfolded to $\forall n^{\mathbb{N}}, f^{\mathbb{N}^{\mathbb{N}}\to\mathbb{N}} (n = fx \to xn = \Omega(f, x))$; we thus define $\Omega(f, x) :\equiv x(fx)$ as $fx = n$ by assumption and then have $x \;\mathrm{R}^x_X\; \Omega$ by definition. Once we construct such a term $\Omega : X^{\mathsf{J}}$, we have $f = f^{\mathsf{J}}\Omega$ (up to pointwise equality). The term $\Omega : X^{\mathsf{J}}$ which preserves the logical relation in the sense of $x \;\mathrm{R}^x_X\; \Omega$ for all $x : X$ is known as a *generic element* [6, 7].

Given a property P of functions $X \to \mathbb{N}$, we define a predicate $Q_\rho \subseteq \rho^{\mathsf{J}}$ on elements of the translated type ρ^{J} inductively on ρ:

$$\begin{aligned} Q_{\mathbb{N}}(f) &:\equiv P(f) \\ Q_{\sigma\to\tau}(h) &:\equiv \forall z^{\sigma^{\mathsf{J}}} \left(Q_\sigma(z) \to Q_\tau(hz)\right). \end{aligned}$$

Note that Q is just an instance of the binary relation defined in Theorem 3. Once we prove the conditions (†) for Q, *i.e.*

$$\forall n^{\mathbb{N}} Q_{\mathbb{N}}(\eta n) \qquad \text{and} \qquad \forall g^{\mathbb{N}\to X\to\mathbb{N}} \left(\forall n^{\mathbb{N}} Q_{\mathbb{N}}(gn) \to Q_{\mathbb{N}\to\mathbb{N}}(\kappa g)\right)$$

we have $Q(t^{\mathsf{J}})$ for any closed term t of T. If we prove also $Q_X(\Omega)$, then we have $P(f)$ for all closed terms $f : \mathbb{N} \to X$ of T because $Q_{\mathbb{N}}(f^{\mathsf{J}}\Omega)$ and $f = f^{\mathsf{J}}\Omega$.

All the remaining examples are about properties of T-definable functions $\mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ which can be proved following the above steps. We instead enrich the "lifting" nucleus to reflect the computational content of the properties so that witnesses of the properties can be obtained as terms of T directly via the translation.

3.3 Continuity

The next applications of our monadic translation are to recover the well-known results that every T-definable function $\mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ is pointwise continuous and its restriction to any compact subspace is uniformly continuous [5].

There are various approaches to continuity: Kohlenbach [13] extracts a term from the extensionality proof via the Dialectica interpretation, and then uses the majorant of this term to construct a modulus of uniform continuity. Coquand and Jaber [6] extend type theory with a new constant for a generic element, decorate the operational semantics with forcing information, and then extract continuity information of a functional by applying it to the generic element. Escardó [7] also employs a generic element to compute continuity but in his model of dialogue trees, which is closely related to our syntactic approach as discussed in Remark 8. There are also various sheaf models [9, 10, 26] in which all functionals from the Baire space $\mathbb{N}^{\mathbb{N}}$ are continuous and those from the Cantor space $\mathbf{2}^{\mathbb{N}}$ are uniformly continuous. Powell [19] introduces a monadic translation for some call-by-value functional languages, one of whose instantiations tackles also continuity of T-definable functionals. His method corresponds to Kuroda's negative translation as discussed in Section 4.3.

We enrich the "lifting" nucleus (Section 3.2) so that moduli of (uniform) continuity are obtained directly from the translation. For the sake of convenience, we extend System T with products. Such extension can be avoided by working with sequences of types and terms as in the literature of functional interpretations such as [15].

3.3.1 Translating products

We extend System T with product type $\sigma \times \tau$ and constants

$$\mathsf{pair} : \sigma \to \tau \to \sigma \times \tau \qquad \mathsf{pr}_1 : \sigma \times \tau \to \sigma \qquad \mathsf{pr}_2 : \sigma \times \tau \to \tau$$

satisfying the usual computation rules. Similarly to Gentzen's translation of conjunction, we translate product type component-wise, *i.e.* $(\sigma \times \tau)^{\mathsf{J}} := \sigma^{\mathsf{J}} \times \tau^{\mathsf{J}}$. Then the above constants are translated into themselves but of the translated types, *e.g.* $\mathsf{pr}_1^{\mathsf{J}} := \mathsf{pr}_1 : \sigma^{\mathsf{J}} \times \tau^{\mathsf{J}} \to \sigma^{\mathsf{J}}$. Recall that the primitive recursor is translated using $\mathrm{ke}_\sigma : (\mathbb{N} \to \sigma^{\mathsf{J}}) \to \mathsf{J}\mathbb{N} \to \sigma^{\mathsf{J}}$ which is defined inductively on σ. So we have to add the following case

$$\mathrm{ke}_{\sigma\times\tau} := \lambda g^{\mathbb{N}\to\sigma^{\mathsf{J}}\times\tau^{\mathsf{J}}} a^{\mathsf{J}\mathbb{N}}.\, \mathsf{pair}\,(\mathrm{ke}_\sigma(\mathsf{pr}_1 \circ g, a), \mathrm{ke}_\tau(\mathsf{pr}_2 \circ g, a))$$

into the definition of ke in order to complete the translation. For the fundamental theorem of logical relation, when extending a relation $\mathrm{R}_{\mathbb{N}} \subseteq \mathbb{N} \times \mathsf{J}\mathbb{N}$ to $\mathrm{R}_\rho \subseteq \rho \times \rho^{\mathsf{J}}$, we add the following case for product type

$$u\ \mathrm{R}_{\sigma\times\tau}\ v := (\mathsf{pr}_1 u\ \mathrm{R}_\sigma\ \mathsf{pr}_1 v) \wedge (\mathsf{pr}_2 u\ \mathrm{R}_\tau\ \mathsf{pr}_2 v)\,.$$

and can easily show that the constants of product types are related to their translations. We often write $\langle a,\, b\rangle$ instead of $\mathsf{pair}(a, b)$ for the sake of readability.

3.3.2 Pointwise continuity

Recall that a function $M : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ is a *modulus of continuity* of $f : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ if

$$\forall \alpha^{\mathbb{N}^{\mathbb{N}}} \beta^{\mathbb{N}^{\mathbb{N}}}\, (\alpha =_{M\alpha} \beta \to f\alpha = f\beta)$$

where $\alpha =_m \beta$ stands for $\forall i{<}m\, (\alpha i = \beta i)$. Our goal is to find a suitable nucleus J so that we can obtain such a functional M from the J-translation of f and then verify its correctness using the fundamental theorem of logical relation.

Let $\mathsf{J}\mathbb{N} = (\mathbb{N}^{\mathbb{N}} \to \mathbb{N}) \times (\mathbb{N}^{\mathbb{N}} \to \mathbb{N})$. For $w : \mathsf{J}\mathbb{N}$ we write V_w to denote its first component and M_w the second, due to the intuition that M_w is a modulus of continuity of the value component V_w. Then we define $\eta : \mathbb{N} \to \mathsf{J}\mathbb{N}$ by

$$\eta(n) := \langle \lambda\alpha.n,\ \lambda\alpha.0\rangle$$

and $\kappa : (\mathbb{N} \to \mathsf{J}\mathbb{N}) \to \mathsf{J}\mathbb{N} \to \mathsf{J}\mathbb{N}$ by

$$\kappa(g, w) :\equiv \big\langle \lambda\alpha.\mathrm{V}_{g(\mathrm{V}_w(\alpha))}(\alpha),\ \lambda\alpha.\max(\mathrm{M}_{g(\mathrm{V}_w(\alpha))}(\alpha), \mathrm{M}_w(\alpha))\big\rangle\,.$$

Note that the "value" components form a "lifting" nucleus in the sense of Section 3.2 so that natural numbers are lifted to functions $\mathbb{N}^{\mathbb{N}} \to \mathbb{N}$. And the "modulus" components will allow the translation to equip a continuity structure to the values. Reasonably $\eta(n)$ equips the constantly zero function as a modulus of continuity to the constant function $\lambda\alpha.n$ since its input is never accessed. As to $\kappa(g, w)$, its value at a point α has two possible moduli: one given by $g(\mathrm{V}_w(\alpha))$ and the other by w; thus the greater one is a modulus of continuity at α.

We work with a logical relation $\mathrm{R}^{\alpha}_{\rho} \subseteq \rho \times \rho^{\mathsf{J}}$ which is parametrized by $\alpha : \mathbb{N}^{\mathbb{N}}$. Specifically, its base case $\mathrm{R}^{\alpha}_{\mathbb{N}} \subseteq \mathbb{N} \times \mathsf{J}\mathbb{N}$ is defined by

$$n\ \mathrm{R}^{\alpha}_{\mathbb{N}}\ w :\equiv n = \mathrm{V}_w(\alpha) \wedge \forall\beta \left(\alpha =_{\mathrm{M}_w(\alpha)} \beta \to \mathrm{V}_w(\alpha) = \mathrm{V}_w(\beta)\right).$$

The first component of $n\ \mathrm{R}^{\alpha}_{\mathbb{N}}\ w$ states that the value of w at α is n, while the second explains exactly the intuition of the type $\mathsf{J}\mathbb{N}$, namely that $\mathrm{M}_w(\alpha)$ is a modulus of continuity of V_w at α. We leave the proof of (†) to the reader. By Theorem 3, we have $t\ \mathrm{R}^{\alpha}_{\rho}\ t^{\mathsf{J}}$ for any $\alpha : \mathbb{N}^{\mathbb{N}}$ and for any closed term $t : \rho$ of T. In particular, we have $f\ \mathrm{R}^{\alpha}_{\mathbb{N}^{\mathbb{N}} \to \mathbb{N}}\ f^{\mathsf{J}}$ for every closed term $f : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ of T.

The last step is to construct the generic element $\Omega : \mathsf{J}\mathbb{N} \to \mathsf{J}\mathbb{N}$ such that $\alpha\ \mathrm{R}^{\alpha}_{\mathbb{N}^{\mathbb{N}}}\ \Omega$ for all $\alpha : \mathbb{N}^{\mathbb{N}}$. Once we unfold $\alpha\ \mathrm{R}^{\alpha}_{\mathbb{N}^{\mathbb{N}}}\ \Omega$, we can see that, for any $w : \mathsf{J}\mathbb{N}$, the value of $\Omega(w)$ has to be $\lambda\alpha.\alpha(\mathrm{V}_w(\alpha))$ as discussed in Section 3.2. Then we also need to construct its modulus of continuity. There are two possible moduli at α: one is $\mathrm{V}_w(\alpha) + 1$ because the modulus of continuity of $\lambda\alpha.\alpha n$ at α is $n + 1$, and the other is $\mathrm{M}_w(\alpha)$. We just take the greater one and then end up with the following definition:

$$\Omega(w) :\equiv \langle \lambda\alpha.\alpha(\mathrm{V}_w(\alpha)),\ \lambda\alpha.\max(\mathrm{V}_w(\alpha) + 1, \mathrm{M}_w(\alpha))\rangle\,.$$

One may have noticed that the above is highly similar to the definition of κ. Indeed, we have $\Omega = \kappa(\lambda n.\langle\lambda\alpha.\alpha n,\ \lambda\alpha.n + 1\rangle)$.

▶ **Theorem 7.** *Every closed term $f : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ of T has a modulus of continuity given by the term $\mathrm{M}_{f^{\mathsf{J}}(\Omega)}$.*

Proof. Because $f\ \mathrm{R}^{\alpha}_{\mathbb{N}^{\mathbb{N}} \to \mathbb{N}}\ f^{\mathsf{J}}$ and $\alpha\ \mathrm{R}^{\alpha}_{\mathbb{N}^{\mathbb{N}}}\ \Omega$, we have $f\alpha\ \mathrm{R}^{\alpha}_{\mathbb{N}}\ f^{\mathsf{J}}(\Omega)$ for any $\alpha : \mathbb{N}^{\mathbb{N}}$, which implies (i) $f = \mathrm{V}_{f^{\mathsf{J}}(\Omega)}$ up to pointwise equality, and (ii) $\mathrm{M}_{f^{\mathsf{J}}(\Omega)}$ is a modulus of continuity of $\mathrm{V}_{f^{\mathsf{J}}(\Omega)}$. Therefore, $\mathrm{M}_{f^{\mathsf{J}}(\Omega)}$ is also a modulus of continuity of f. ◀

▶ Remark 8. The above development can be viewed as a syntactic (and simplified) version of Escardó's approach via dialogue trees [7]. The algorithms to construct moduli of continuity in these two methods are exactly the same. On the other hand, though Powell works also with a monadic translation [19], his algorithm is different because he translates terms in the call-by-value manner. We will look into this in more detail in Section 4.3.

3.3.3 Uniform continuity

The objective here is to, for each closed term $f : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ of T, construct a *modulus of uniform continuity* $M : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$, *i.e.*

$$\forall\delta^{\mathbb{N}^{\mathbb{N}}}\alpha^{\mathbb{N}^{\mathbb{N}}}\beta^{\mathbb{N}^{\mathbb{N}}}\left(\alpha \le^1 \delta \wedge \beta \le^1 \delta \wedge \alpha =_{M\delta} \beta \to f\alpha = f\beta\right)$$

where $\alpha \le^1 \beta$ stands for $\forall i(\alpha i \le \beta i)$. The value $M\delta$ is called a modulus of uniform continuity of f on $\{\alpha : \mathbb{N}^{\mathbb{N}} \mid \alpha \le^1 \delta\}$. The following fact of uniform continuity plays an important role in the construction:

▶ **Lemma 9.** *If $f : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ is uniformly continuous on $\{\alpha : \mathbb{N}^{\mathbb{N}} \mid \alpha \leq^1 \delta\}$ with a modulus m, then it has a maximum image on $\{\alpha : \mathbb{N}^{\mathbb{N}} \mid \alpha \leq^1 \delta\}$.*

Proof. We compute the maximum image $\Theta(m, f, \delta)$ by induction on the modulus m:

$$\begin{aligned} \Theta(0, f, \delta) &\coloneqq f\delta \\ \Theta(m+1, f, \delta) &\coloneqq \Phi\left(\lambda i.\Theta\left(m, \lambda\alpha.f(i * \alpha), \delta \circ \mathsf{suc}\right), \delta 0\right) \end{aligned}$$

where $i * \alpha$ is an infinite sequence with head i and tail α, and $\Phi : \mathbb{N}^{\mathbb{N}} \to \mathbb{N} \to \mathbb{N}$ defined by

$$\begin{aligned} \Phi(\alpha, 0) &\coloneqq \alpha 0 \\ \Phi(\alpha, n+1) &\coloneqq \max\left(\Phi(\alpha, n), \alpha(n+1)\right) \end{aligned}$$

i.e. $\Phi(\alpha, n)$ is the greatest αi for $i \leq n$ (note that Φ is the κ of the "majorizability" nucleus introduced in Section 3.1). In the base case of Θ, the modulus is 0 and thus f is constant with the value $f\delta$. To compute $\Theta(m+1, f, \delta)$, by induction hypothesis we have for each $i : \mathbb{N}$ the maximum image $\Theta\left(m, \lambda\alpha.f(i * \alpha), \delta \circ \mathsf{suc}\right)$ of the function $\lambda\alpha.f(i*\alpha)$ with a modulus m on the inputs bounded by $\delta \circ \mathsf{suc}$. Because the inputs of f are bounded by δ, the greatest $\Theta\left(m, \lambda\alpha.f(i * \alpha), \delta \circ \mathsf{suc}\right)$ for $i < \delta 0$ is the maximum image of f, and we use Φ to find it. Note that both Φ and Θ can be defined in T using rec. ◀

We are now ready to construct the nucleus. Let $\mathsf{J}\mathbb{N} = (\mathbb{N}^{\mathbb{N}} \to \mathbb{N}) \times (\mathbb{N}^{\mathbb{N}} \to \mathbb{N})$. The ieda is exactly the same as in the previous treatment to pointwise continuity: For any $w : \mathsf{J}\mathbb{N}$, its second component M_w is (expected to be) a modulus of uniform continuity of the first component V_w. Then we define $\eta : \mathbb{N} \to \mathsf{J}\mathbb{N}$ by

$$\eta(n) \coloneqq \langle \lambda\alpha.n,\ \lambda\alpha.0 \rangle$$

and $\kappa : (\mathbb{N} \to \mathsf{J}\mathbb{N}) \to \mathsf{J}\mathbb{N} \to \mathsf{J}\mathbb{N}$ by

$$\kappa(g, w) \coloneqq \left\langle \lambda\alpha.\mathrm{V}_{g(\mathrm{V}_w(\alpha))}(\alpha),\ \lambda\delta.\max(\Phi(\lambda i.\mathrm{M}_{gi}(\delta), \Theta(\mathrm{M}_w(\delta), \mathrm{V}_w, \delta)), \mathrm{M}_w(\delta)) \right\rangle$$

where Φ and Θ are defined in the proof of Lemma 9. Specifically, we construct a modulus of uniform continuity for the value of $\kappa(g, w)$ as follows: Given $\delta : \mathbb{N}^{\mathbb{N}}$, we have two possible moduli given by those of g and w at δ and thus we choose the larger one. Actually the only complication comes from the calculation of the modulus given by g. For each $i : \mathbb{N}$ we have a modulus $\mathrm{M}_{gi}(\delta)$. In the value of $\kappa(g, w)$, we apply g to $\mathrm{V}_w(\alpha)$ for input α. Because V_w has a maximum image computed using Θ, we only need to find the greatest modulus $\mathrm{M}_{gi}(\delta)$ for i not greater than the maximum image of V_w, and we use Φ for this purpose.

Given $\delta : \mathbb{N}^{\mathbb{N}}$, the base case $\mathrm{R}^{\delta}_{\mathbb{N}} \subseteq \mathbb{N} \times \mathsf{J}\mathbb{N}$ of the logical relation is defined by

$$n \mathrel{\mathrm{R}^{\delta}_{\mathbb{N}}} w \coloneqq n = \mathrm{V}_w(\delta) \wedge \forall\alpha, \beta \left(\alpha \leq^1 \delta \wedge \beta \leq^1 \delta \wedge \alpha =_{\mathrm{M}_w(\delta)} \beta \to \mathrm{V}_w(\alpha) = \mathrm{V}_w(\beta)\right).$$

In words, $n \mathrel{\mathrm{R}^{\delta}_{\mathbb{N}}} w$ means that n is the value of w at δ and that $\mathrm{M}_w(\delta)$ is a modulus of uniform continuity of V_w on $\{\alpha : \mathbb{N}^{\mathbb{N}} \mid \alpha \leq^1 \delta\}$. It is routine to check that both η and κ preserve the logical relation in the sense of (†), which we again leave to the reader. By Theorem 3 we have $t \mathrel{\mathrm{R}^{\delta}_{\rho}} t^{\mathsf{J}}$ for any $\delta : \mathbb{N}^{\mathbb{N}}$ and for any closed term $t : \rho$ of T. Moreover, the generic element $\Omega : \mathsf{J}\mathbb{N} \to \mathsf{J}\mathbb{N}$ defined by

$$\Omega \coloneqq \kappa(\lambda n. \langle \lambda\alpha.\alpha n,\ \lambda\alpha.n+1 \rangle)$$

also preserves the logical relation in the sense of $\delta \mathrel{\mathrm{R}^{\delta}_{\mathbb{N}^{\mathbb{N}}}} \Omega$ for all $\delta : \mathbb{N}^{\mathbb{N}}$.

With a proof similar to the one of Theorem 7, we get the following result:

▶ **Theorem 10.** *Every closed term $f : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ of T has a modulus of uniform continuity given by the term $\mathrm{M}_{f^{\mathsf{J}}(\Omega)}$.*

3.4 General bar recursion

To prove Schwichtenberg's theorem [20] that the System T definable functionals are closed under a rule-like version Spector's bar recursion of type levels 0 and 1, Oliva and Steila [17] introduce a notion of general bar recursion whose termination condition is given by decidable monotone predicates on finite sequences. As the last example, we recover their construction of general-bar-recursion functionals ([17, Definitions 3.1 & 3.3]) via an instantiation of our translation. For this, we need the following notations:

- We represent decidable predicates as functions $\mathbb{N}^* \to \mathbf{2}$, where $\mathbb{N}^*$ is the type of finite sequences of natural numbers and $\mathbf{2} = \{0,1\}$ is the type of booleans.
- For any $S : \mathbb{N}^* \to \mathbf{2}$ and $s : \mathbb{N}^*$, we write $S(s)$ instead of $S(s) = 1$.
- For any $s : \mathbb{N}^*$, we write $|s| : \mathbb{N}$ to denote its length and $\hat{s} : \mathbb{N}^\mathbb{N}$ the extension of s with infinitely many 0's.
- For any $s : \mathbb{N}^*$ and $n : \mathbb{N}$, we write $s * n : \mathbb{N}^*$ to denote appending n to s.
- For any $s : \mathbb{N}^*$ and $\alpha : \mathbb{N}^\mathbb{N}$, we write $s * \alpha : \mathbb{N}^\mathbb{N}$ to denote their concatenation.

Note that the treatment of $\mathbb{N}^*$ and $\mathbf{2}$ is not essential. For instance, we can represent a finite sequence s by a pair $\langle \alpha, n \rangle$ and consider s as the prefix of the infinite sequence α of length n as in our Agda implementation. All the above operations on sequences are definable in T. We also need the following definitions:

- We call $\xi : (\mathbb{N}^* \to \sigma) \to (\mathbb{N}^* \to \sigma^\mathbb{N} \to \sigma) \to \mathbb{N}^* \to \sigma$ a functional of *general bar recursion* for $S : \mathbb{N}^* \to \mathbf{2}$ if $\mathcal{GBR}_S(\xi)$ holds where $\mathcal{GBR}_S(\xi)$ is defined by

$$\mathcal{GBR}_S(\xi) :\equiv \forall\, G^{\mathbb{N}^* \to \sigma} H^{\mathbb{N}^* \to \sigma^\mathbb{N} \to \sigma} s^{\mathbb{N}^*} \left\{ \begin{array}{c} S(s) \to \xi(G,H,s) = G(s) \\ \wedge \\ \neg S(s) \to \xi(G,H,s) = H(s, \lambda n^\mathbb{N}.\xi(G,H,s*n)) \end{array} \right\}.$$

- A predicate S is *monotone* if $S(s)$ implies $S(s*n)$ for all $s : \mathbb{N}^*$ and $n : \mathbb{N}$.
- For $Y : \mathbb{N}^\mathbb{N} \to \mathbb{N}$, we say S *secures* Y if

$$\forall s^{\mathbb{N}^*} \Big(S(s) \to \forall \alpha^{\mathbb{N}^\mathbb{N}}\, Y(s*\alpha) = Y(\hat{s}) \Big).$$

Let $Y : \mathbb{N}^\mathbb{N} \to \mathbb{N}$ be a closed term of T. Oliva and Steila show (i) for any S securing Y, from a functional of general bar recursion for S we can construct a functional of Spector's bar recursion for Y [17, Theorem 2.4], and (ii) we can construct a monotone predicate S that secures Y and a functional of general bar recursion for S [17, Theorem 3.4]. In this way, they give a new proof of Schwichtenberg's bar recursion closure theorem with an explicit construction of Spector's bar-recursion functionals.

We firstly construct a nucleus for general bar recursion. Fix a type σ of T. Let $\mathsf{J}\mathbb{N} = (\mathbb{N}^\mathbb{N} \to \mathbb{N}) \times (\mathbb{N}^* \to \mathbf{2}) \times ((\mathbb{N}^* \to \sigma) \to (\mathbb{N}^* \to \sigma^\mathbb{N} \to \sigma) \to \mathbb{N}^* \to \sigma)$. Given $w : \mathsf{J}\mathbb{N}$, we write $\mathrm{V}_w, \mathrm{S}_w, \mathrm{B}_w$ to denote its three components. The intuition is that S_w is a monotone predicate securing V_w and B_w is a functional of general bar recursion for S_w. We define $\eta : \mathbb{N} \to \mathsf{J}\mathbb{N}$ by

$$\eta(n) :\equiv \langle \lambda\alpha.n,\ \lambda s.1,\ \lambda GH.G \rangle$$

and $\kappa : (\mathbb{N} \to \mathsf{J}\mathbb{N}) \to \mathsf{J}\mathbb{N} \to \mathsf{J}\mathbb{N}$ by

$$\kappa(g,w) :\equiv \big\langle \lambda\alpha.\mathrm{V}_{g(\mathrm{V}_w\alpha)}\alpha,\ \lambda s.\min(\mathrm{S}_w(s), \mathrm{S}_{g(\mathrm{V}_w\hat{s})}(s)),\ \lambda GH.\mathrm{B}_w(\lambda s.\mathrm{B}_{g(\mathrm{V}_w\hat{s})}(G,H,s), H) \big\rangle$$

where $\min : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ returns the smaller argument. Lastly, we define the generic element $\Omega : \mathsf{J}\mathbb{N} \to \mathsf{J}\mathbb{N}$ by

$$\Omega :\equiv \kappa(\lambda n.\, \langle \lambda\alpha.\alpha n,\ \lambda s.\mathrm{Le}(n, |s|),\ \Psi n \rangle)$$

where $\mathrm{Le} : \mathbb{N} \to \mathbb{N} \to \mathbf{2}$ has value 1 iff its first argument is strictly smaller than the second, and $\Psi n : (\mathbb{N}^* \to \sigma) \to (\mathbb{N}^* \to \sigma^{\mathbb{N}} \to \sigma) \to \mathbb{N}^* \to \sigma$ is a T-definable functional of bar recursion for the constant function $Y = \lambda\alpha.n$ [17, Lemma 2.1], *i.e.*

$$\forall\, G^{\mathbb{N}^* \to \sigma} H^{\mathbb{N}^* \to \sigma^{\mathbb{N}} \to \sigma} s^{\mathbb{N}^*} \begin{Bmatrix} n < |s| \to \Psi n(G, H, s) = G(s) \\ \wedge \\ n \geq |s| \to \Psi n(G, H, s) = H(s, \lambda m.\Psi n(G, H, s * m)) \end{Bmatrix}.$$

For any $n : \mathbb{N}$, it is clear that $\lambda s.\mathrm{Le}(n, |s|)$ is a monotone predicate that secures $\lambda\alpha.\alpha n$, and that Ψn is a functional of general bar recursion for $\lambda s.\mathrm{Le}(n, |s|)$.

▶ **Theorem 11.** *For any closed term $Y : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ of T,*

1. $\mathrm{S}_{Y^{\mathsf{J}}\Omega}$ *is a monotone predicate securing Y, and*
2. $\mathrm{B}_{Y^{\mathsf{J}}\Omega}$ *is a functional of general bar recursion of* $\mathrm{S}_{Y^{\mathsf{J}}\Omega}$.

Proof. Given $\alpha : \mathbb{N}^{\mathbb{N}}$, we define the base case $\mathrm{R}^{\alpha}_{\mathbb{N}} \subseteq \mathbb{N} \times \mathsf{J}\mathbb{N}$ of the logical relation by

$$n\ \mathrm{R}^{\alpha}_{\mathbb{N}}\ w :\equiv n = \mathrm{V}_w(\alpha) \wedge \mathrm{S}_w \text{ is monotone} \wedge \mathrm{S}_w \text{ secures } \mathrm{V}_w \wedge \mathcal{GBR}_{\mathrm{S}_w}(\mathrm{B}_w).$$

To apply the fundamental theorem of logical relation, we need to check the conditions (†):

- It is trivial to prove $n\ \mathrm{R}^{\alpha}_{\mathbb{N}}\ \eta n$ for all $n : \mathbb{N}$.
- As to κ, given $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathsf{J}\mathbb{N}$ such that $fi\ \mathrm{R}^{\alpha}_{\mathbb{N}}\ gi$ for all $i : \mathbb{N}$, our goal is to prove $f\ \mathrm{R}^{\alpha}_{\mathbb{N}\to\mathbb{N}}\ \kappa g$. Let $n : \mathbb{N}$ and $w : \mathsf{J}\mathbb{N}$ with $n\ \mathrm{R}^{\alpha}_{\mathbb{N}}\ w$ be given.
  - We have $fn = \mathrm{V}_{gn}(\alpha) = \mathrm{V}_{g(\mathrm{V}_w(\alpha))}(\alpha) = \mathrm{V}_{\kappa(g,w)}(\alpha)$ as in the previous examples.
  - Because S_w is monotone and so is S_{gi} for all $i : \mathbb{N}$ by assumption, the predicate $\mathrm{S}_{\kappa(g,w)}$ is also monotone.
  - If $\mathrm{S}_{\kappa(g,w)}(s)$, then $\mathrm{S}_w(s)$ and $\mathrm{S}_{g(\mathrm{V}_w\hat{s})}(s)$ by definition. Given $\alpha : \mathbb{N}^{\mathbb{N}}$, we have $\mathrm{V}_w(s*\alpha) = \mathrm{V}_w(\hat{s})$ because S_w secures V_w. Then we have $\mathrm{V}_{\kappa(g,w)}(s * \alpha) = \mathrm{V}_{g(\mathrm{V}_w(s*\alpha))}(s * \alpha) = \mathrm{V}_{g(\mathrm{V}_w(\hat{s}))}(s * \alpha) = \mathrm{V}_{g(\mathrm{V}_w(\hat{s}))}(\hat{s}) = \mathrm{V}_{\kappa(g,w)}(\hat{s})$ because $\mathrm{S}_{g(\mathrm{V}_w\hat{s})}$ secures $\mathrm{V}_{g(\mathrm{V}_w\hat{s})}$. Hence $\mathrm{S}_{\kappa(g,w)}$ secures $\mathrm{V}_{\kappa(g,w)}$.
  - Lastly we show that $\mathrm{B}_{\kappa(g,w)}$ is a functional of general bar recursion for $\mathrm{S}_{\kappa(g,w)}$. Let $G : \mathbb{N}^* \to \sigma$, $H : \mathbb{N}^* \to \sigma^{\mathbb{N}} \to \sigma$ and $s : \mathbb{N}^*$ be given. (1) If $\mathrm{S}_{\kappa(g,w)}(s)$, then $\mathrm{S}_w(s)$ and $\mathrm{S}_{g(\mathrm{V}_w\hat{s})}(s)$, and thus we have $\mathrm{B}_{\kappa(g,w)}(G, H, s) = G(s)$. (2) If $\neg\mathrm{S}_{\kappa(g,w)}(s)$, then we have two cases to check: (2.1) If $\mathrm{S}_w(s)$, then $\neg\mathrm{S}_{g(\mathrm{V}_w\hat{s})}(s)$ by definition. It is not hard to show $\mathrm{B}_{\kappa(g,w)}(G, H, s) = H(s, \lambda n.\mathrm{B}_{\kappa(g,w)}(G, H, s * n))$. (2.2) If $\neg(\mathrm{S}_w(s))$, then we always have $\mathrm{B}_{\kappa(g,w)}(G, H, s) = H(s, \lambda n.\mathrm{B}_{\kappa(g,w)}(G, H, s * n))$ no matter if $\mathrm{S}_{g(\mathrm{V}_w\hat{s})}(s)$ holds or not.

  Hence, we have $fn\ \mathrm{R}^{\alpha}_{\mathbb{N}}\ \kappa(g, w)$.

Given a closed term $Y : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ of T, for any $\alpha : \mathbb{N}^{\mathbb{N}}$ we have $Y\ \mathrm{R}^{\alpha}_{\mathbb{N}^{\mathbb{N}}\to\mathbb{N}}\ Y^{\mathsf{J}}$ by Theorem 3. For any $n : \mathbb{N}$, we have $n\ \mathrm{R}^{\alpha}_{\mathbb{N}}\ \langle\lambda\alpha.\alpha n,\ \lambda s.\mathrm{Le}(n, |s|),\ \Psi n\rangle$ by definition; thus, $\alpha\ \mathrm{R}^{\alpha}_{\mathbb{N}\to\mathbb{N}}\ \Omega$ holds by (†) for κ which we have just proved. Hence we have $Y\alpha\ \mathrm{R}^{\alpha}_{\mathbb{N}}\ Y^{\mathsf{J}}\Omega$ for any $\alpha : \mathbb{N}^{\mathbb{N}}$. From this, we get (i) $Y = \mathrm{V}_{Y^{\mathsf{J}}\Omega}$ up to pointwise equality, (ii) $\mathrm{S}_{Y^{\mathsf{J}}\Omega}$ is a monotone predicate securing $\mathrm{V}_{Y^{\mathsf{J}}\Omega}$ and thus also Y, and (iii) $\mathrm{B}_{Y^{\mathsf{J}}\Omega}$ is a functional of general bar recursion for $\mathrm{S}_{Y^{\mathsf{J}}\Omega}$. ◀

The above development is just a restructuring of the work of Oliva and Steila [17] that fits into our framework. But there are some small differences (or simplifications):

- [17] requires the predicate S to satisfy the bar condition $\forall\alpha^{\mathbb{N}^{\mathbb{N}}}\exists n^{\mathbb{N}} S(\bar{\alpha}n)$, where $\bar{\alpha}n : \mathbb{N}^*$ is the prefix of α of length n. As pointed out by Makoto Fujiwara in a personal discussion, this condition is not needed for the result. So we remove it in the above development.

- [17] assumes that the closed terms $Y : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ are of the form $\lambda\alpha.t$ where $\alpha : \mathbb{N}^{\mathbb{N}}$ is the only free variable in $t : \mathbb{N}$, and treats α as a special constant (for the generic element). Motivated by a version of Escardó's Agda development of [7], we avoid such extension by using the lifting nucleus that is introduced in Section 3.2.
- The syntactic translation of T in [17, Definitions 3.1 & 3.3] contains only the construction of general-bar-recursion functionals while the one of monotone securing predicates is given in the proof of the main theorem [17, Theorem 3.4]. We combine both in the translation in order to split the constructions from the correctness proof.

As pointed out by an anonymous reviewer, this section unifies the results in Section 3.3 in the sense that moduli of (uniform) continuity can be defined from monotone securing bars and general-bar-recursion functionals: Let $Y : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ be a closed term of T. The monotone predicate $\mathrm{S}_{Y^{\mathsf{J}}\Omega}$ is a bar, *i.e.* $\forall\alpha^{\mathbb{N}^{\mathbb{N}}}\exists n^{\mathbb{N}}\mathrm{S}_{Y^{\mathsf{J}}\Omega}(\bar{\alpha}n)$, as shown in [17, Theorem 3.4]. This together with the fact that it secures Y implies the pointwise continuity of Y. The witness of the fact that $\mathrm{S}_{Y^{\mathsf{J}}\Omega}$ is a bar obtained via *e.g.* modified realizability [14] is a continuity modulus of Y. Our translation in Section 3.3 is just an explicit procedure to get these witnesses that are blurred in the proof of [17, Theorem 3.4]. On the other hand, the reviewer points out that we can construct a modulus $M : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ of uniform continuity of Y by

$$M(\delta) :\equiv \mathrm{B}_{Y^{\mathsf{J}}\Omega}(G, H^{\delta}, \mathsf{nil})$$

where $G(s) :\equiv 0$, $H^{\delta}(s, f) :\equiv 1 + \max\{fn \mid n \leq \delta(|s|)\}$ and $\mathsf{nil} : \mathbb{N}^*$ is the empty sequence. The idea is that if s is a prefix of δ then $\mathrm{B}_{Y^{\mathsf{J}}\Omega}(G, H^{\delta}, s)$ is a modulus of uniform continuity of the function $\lambda\alpha.Y(s * \alpha)$ on $\{\alpha : \mathbb{N}^{\mathbb{N}} \mid \alpha \leq^1 \delta\}$: If $\mathrm{S}_{Y^{\mathsf{J}}\Omega}(s)$, we know $\lambda\alpha.Y(s * \alpha)$ is a constant function because $\mathrm{S}_{Y^{\mathsf{J}}\Omega}$ secures Y. Then $\mathrm{B}_{Y^{\mathsf{J}}\Omega}(G, H^{\delta}, s) = G(s) = 0$ is a proper modulus. If $\neg\mathrm{S}_{Y^{\mathsf{J}}\Omega}(s)$, then the step functional H^{δ} finds the greatest value of the moduli of $\lambda\alpha.Y(s * n * \alpha)$ given by $\mathrm{B}_{Y^{\mathsf{J}}\Omega}(G, H^{\delta}, s * n)$ for $n \leq \delta(|s|)$ and then adds 1 to get a modulus of $\lambda\alpha.Y(s * \alpha)$.

4 Generalization and variants of the monadic translation

We have developed a self-translation of System T in the spirit of Gentzen's negative translation. We conclude the paper by generalizing it to translate sums and comparing it with another two monadic translations.

4.1 Translating sums

We generalize our Gentzen-style translation to translate also sums. Let us extend T with sum type $\sigma + \tau$ and the following constants

$$\mathsf{inj}_1 : \sigma \to \sigma + \tau \qquad \mathsf{inj}_2 : \tau \to \sigma + \tau \qquad \mathsf{case} : (\sigma \to \rho) \to (\tau \to \rho) \to \sigma + \tau \to \rho.$$

In his translation, Gentzen places a double negation in front of disjunctions (see Section 1.1). Following this inspiration, the sum type $\sigma + \tau$ should be translated into $\mathsf{J}(\sigma^{\mathsf{J}} + \tau^{\mathsf{J}})$. But the simple notion of a nucleus given by a type and two terms does not suffice. We have to work with the following more general notion: A nucleus $(\mathsf{J}, \eta, \kappa)$ consists of an endofunction J on types of (the extension of) T, and terms

$$\eta : \sigma \to \mathsf{J}\sigma \qquad \kappa : (\sigma \to \mathsf{J}\tau) \to \mathsf{J}\sigma \to \mathsf{J}\tau$$

for any types σ, τ of (the extension of) T. Then we add the following clauses to Definition 2 to complete the translation:

$$\begin{aligned} (\sigma+\tau)^{\mathsf{J}} &:= \mathsf{J}(\sigma^{\mathsf{J}}+\tau^{\mathsf{J}}) \qquad & \mathsf{inj}_i{}^{\mathsf{J}} &:= \eta \circ \mathsf{inj}_i \qquad \text{for } i \in \{1,2\} \\ \mathrm{ke}_{\sigma+\tau} &:= \kappa & \mathsf{case}^{\mathsf{J}} &:= \lambda fg.\mathrm{ke}(\mathsf{case}(f,g)). \end{aligned}$$

Generalizing the fundamental theorem of logical relation to cover sums is remained as a future task. Both natural numbers and sums, as instances of inductive types, are translated in a very similar way. Another future task is to generalize the translation to cover all (strictly positive) inductive and coinductive types.

4.2 The Kolmogorov-style monadic translation

Barthe and Uustalu's call-by-name continuation passing style transformation [4] corresponds to Kolmogorov's negative translation. By replacing the continuation monad with a nucleus, one obtains a Kolmogorov-style monadic translation which is studied in [24]. Kolmogorov places a double negation in front of every subformula. Similarly we place the nucleus in front of every subtype. Hence we have to work with the more general notion of a nucleus $(\mathsf{J}, \eta, \kappa)$ where J is an endofunction on types. Specifically, each type ρ of T is translated to $\mathsf{J}\langle\rho\rangle$ where $\langle\rho\rangle$ is defined inductively as follows

$$\begin{aligned} \langle\mathbb{N}\rangle &:= \mathbb{N} \\ \langle\sigma \mathbin{\square} \tau\rangle &:= \mathsf{J}\langle\sigma\rangle \mathbin{\square} \mathsf{J}\langle\tau\rangle \qquad \text{for } \square \in \{\to, \times, +\}. \end{aligned}$$

Each term $t : \rho$ is translated to a term $\langle t\rangle : \mathsf{J}\langle\rho\rangle$. In order to translate function application, we have to consider a monadic notion of application. Given $f : \mathsf{J}(\sigma \to \mathsf{J}\tau)$ and $a : \sigma$, we "apply" f to a as

$$f \diamond a := \kappa(\lambda g^{\sigma\to\mathsf{J}\tau}.ga,\ f).$$

Then for any terms $t : \sigma \to \tau$ and $u : \sigma$, we define $\langle tu\rangle := \langle t\rangle \diamond \langle u\rangle$. The rest of the translation can be found in the appendix.

4.3 The Kuroda-style monadic translation

There is also a Kuroda-style monadic translation of System T which has been studied by Powell in [18, 19], where each type ρ is translated to $\mathsf{J}[\rho]$ with $[\rho]$ defined by

$$\begin{aligned} [\mathbb{N}] &:= \mathbb{N} \qquad & [\sigma\times\tau] &:= [\sigma]\times[\tau] \\ [\sigma\to\tau] &:= [\sigma] \to \mathsf{J}[\tau] & [\sigma+\tau] &:= [\sigma]+[\tau]. \end{aligned}$$

Note that it actually corresponds to the variant of Kuroda's negative translation where double negations are placed also in front of conclusions of implications (see [16, Section 6]). Here we need another notion of application for the term translation: Given $f : \mathsf{J}(\sigma \to \mathsf{J}\tau)$ and $a : \mathsf{J}\sigma$, we "apply" f to a as

$$f \bullet a := \kappa(\lambda g^{\sigma\to\mathsf{J}\tau}.\kappa(g,a),\ f).$$

The complete translation can be found in the appendix.

Powell makes use of the Kuroda-style translation to extract moduli of continuity [19, Section 5], similarly to our development in Section 3.3. However, our algorithms are different because the Kuroda-style translation is call-by-value while our Gentzen-style one is call-by-name. Consider the following example. Let $t = \lambda\alpha.\mathsf{rec}(\alpha 0, \lambda nm.0, 1) : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ which is a constant function. If we apply the Kuroda-style translation to t with the continuity nucleus (similar to the one given in Section 3.3 but generalized to arbitrary types of T), then we get a modulus $\lambda\alpha.1$, because in the call-by-value strategy all the inputs of `rec` including $\alpha 0$ are evaluated. With the Gentzen-style translation, we get $\lambda\alpha.0$ because $\alpha 0$ is never evaluated in the call-by-name strategy.

References

1 Peter Aczel. The Russell–Prawitz modality. *Mathematical Structures in Computer Science*, 11(4):541–554, 2001. doi:10.1017/S0960129501003309.

2 Agda community. The Agda Wiki. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php.

3 Rob Arthan and Paulo Oliva. On affine logic and łukasiewicz logic, 2014. arXiv:1404.0570 [cs.LO].

4 Gilles Barthe and Tarmo Uustalu. CPS translating inductive and coinductive types. In *2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*, volume 37 of *SIGPLAN Notices*, pages 131–142. ACM Press, New York, 2002. doi:10.1145/503032.503043.

5 Michael J. Beeson. *Foundations of Constructive Mathematics.* Springer, 1985. doi:10.1007/978-3-642-68952-9.

6 Thierry Coquand and Guilhem Jaber. A note on forcing and type theory. *Fundamenta Informaticae*, 100(1-4):43–52, 2010. doi:10.3233/FI-2010-262.

7 Martín H. Escardó. Continuity of Gödel's system T functionals via effectful forcing. In *Proceedings of the Twenty-ninth Conference on the Mathematical Foundations of Programming Semantics (MFPS'2013)*, volume 298 of *Electronic Notes in Theoretical Computer Science*, pages 119–141. Elsevier, 2013. doi:10.1016/j.entcs.2013.09.010.

8 Martín H. Escardó and Paulo Oliva. The Peirce translation. *Annals of Pure and Applied Logic*, 163(6):681–692, 2012. doi:10.1016/j.apal.2011.11.002.

9 Martín H. Escardó and Chuangjie Xu. A constructive manifestation of the Kleene–Kreisel continuous functionals. *Annals of Pure and Applied Logic*, 167(9):770–793, 2016. Fourth Workshop on Formal Topology (4WFTop). doi:10.1016/j.apal.2016.04.011.

10 Michael P. Fourman. Notions of choice sequence. In *The L. E. J. Brouwer Centenary Symposium*, volume 110 of *Studies in Logic and the Foundations of Mathematics*, pages 91–105. Elsevier, 1982. doi:10.1016/S0049-237X(09)70125-9.

11 William A. Howard. Hereditarily majorizable functionals of finite type. In *Metamathematical investigation of intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*, pages 454–461. Springer, Berlin, Heidelberg, 1973. doi:10.1007/BFb0066739.

12 Hajime Ishihara. A note on the Gödel-Gentzen translation. *Mathematical Logic Quarterly*, 46(1):135–137, 2000. doi:10.1002/(SICI)1521-3870(200001)46:1<135::AID-MALQ135>3.0.CO;2-R.

13 Ulrich Kohlenbach. Pointwise hereditary majorization and some applications. *Archive for Mathematical Logic*, 31(4):227–241, 1992. doi:10.1007/BF01794980.

14 Ulrich Kohlenbach. *Applied Proof Theory: Proof Interpretations and their Use in Mathematics.* Springer Monographs in Mathematics. Springer, 2008. doi:10.1007/978-3-540-77533-1.

15 Paulo Oliva. Unifying functional interpretations. *Notre Dame Journal of Formal Logic*, 47(2):263–290, 2006. doi:10.1305/ndjfl/1153858651.

16 Paulo Oliva and Gilda Ferreira. On the relation between various negative translations. In *Logic, Construction, Computation*, volume 3 of *Mathematical Logic Series*, pages 227–258. Ontos-Verlag, 2012. doi:10.1515/9783110324921.

17 Paulo Oliva and Silvia Steila. A direct proof of Schwichtenberg's bar recursion closure theorem. *The Journal of Symbolic Logic*, 83(1):70–83, 2018. doi:10.1017/jsl.2017.33.

18 Thomas Powell. A functional interpretation with state. In *Proceedings of the Thirty third Annual IEEE Symposium on Logic in Computer Science (LICS 2018)*, pages 839–848. IEEE Computer Society Press, July 2018. `doi:10.1145/3209108.3209134`.

19 Thomas Powell. A unifying framework for continuity and complexity in higher types, 2019. `arXiv:1906.10719 [cs.LO]`.

20 Helmut Schwichtenberg. On bar recursion of types 0 and 1. *The Journal of Symbolic Logic*, 44(3):325–329, 1979. `doi:10.2307/2273126`.

21 Richard Statman. Logical relations and the typed lambda-calculus. *Information and Control*, 65(2/3):85–97, 1985. `doi:10.1016/S0019-9958(85)80001-2`.

22 Anne Sjerp Troelstra and Helmut Schwichtenberg. *Basic Proof Theory*, volume 43 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 2nd edition, 2000.

23 Anne Sjerp Troelstra and Dirk van Dalen. *Constructivism in mathematics, Vol. II*, volume 123 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam, 1988.

24 Tarmo Uustalu. Monad translating inductive and coinductive types. In *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *Lecture Notes in Computer Science*, pages 299–315. Springer, 2002. `doi:10.1007/3-540-39185-1_17`.

25 Benno van den Berg. A Kuroda-style j-translation. *Archive for Mathematical Logic*, 58(5-6):627–634, 2019. `doi:10.1007/s00153-018-0656-x`.

26 Gerrit van der Hoeven and Ieke Moerdijk. Sheaf models for choice sequences. *Annals of Pure and Applied Logic*, 27(1):63–107, 1984. `doi:10.1016/0168-0072(84)90035-6`.

27 Chuangjie Xu. A syntactic approach to continuity of T-definable functionals. *Logical Methods in Computer Science*, 16(1):22:1–22:11, 2020. URL: `https://lmcs.episciences.org/6130`.

A The Kolmogorov- and Kuroda-style monadic translations of T

We present the Kolmogorov- and Kuroda-style monadic translations of System T extended with products and sums, parametrized by a nucleus $(\mathsf{J}, \eta, \kappa)$, where J is an endofunction on types of T and $\eta : \sigma \to \mathsf{J}\sigma$ and $\kappa : (\sigma \to \mathsf{J}\tau) \to \mathsf{J}\sigma \to \mathsf{J}\tau$ are terms of T. Recall the following notions of monadic application that will be needed in the translations:

- Given $f : \mathsf{J}(\sigma \to \mathsf{J}\tau)$ and $a : \sigma$, we define

$$f \diamond a := \kappa(\lambda g^{\sigma \to \mathsf{J}\tau}.ga,\ f).$$

- Given $f : \mathsf{J}(\sigma \to \mathsf{J}\tau)$ and $a : \mathsf{J}\sigma$, we define

$$f \bullet a := \kappa(\lambda g^{\sigma \to \mathsf{J}\tau}.\kappa(g, a),\ f).$$

▶ **Definition 12** (Kolmogorov-style monadic translation)**.** *We assign to each type ρ a type $\mathsf{J}\langle\rho\rangle$ where $\langle\rho\rangle$ is defined as follows*

$$\begin{aligned} \langle\mathbb{N}\rangle &:= \mathbb{N} \\ \langle\sigma \mathbin{\square} \tau\rangle &:= \mathsf{J}\langle\sigma\rangle \mathbin{\square} \mathsf{J}\langle\tau\rangle \qquad \textit{for } \square \in \{\to, \times, +\}. \end{aligned}$$

Each term $\Gamma \vdash t : \rho$ is translated to a term $\langle\Gamma\rangle \vdash \langle t\rangle : \mathsf{J}\langle\rho\rangle$, where $\langle\Gamma\rangle$ is a new context assigning each $x : \sigma \in \Gamma$ to a fresh variable $\hat{x} : \mathsf{J}\langle\sigma\rangle$, and $\langle t\rangle$ is defined inductively as follows:

$$\begin{array}{rlrl} \langle x\rangle &:= \hat{x} & \langle 0\rangle &:= \eta(0) \\ \langle\lambda x.t\rangle &:= \eta(\lambda\hat{x}.\langle t\rangle) & \langle\mathsf{suc}\rangle &:= \eta(\kappa(\eta \circ \mathsf{suc})) \\ \langle tu\rangle &:= \langle t\rangle \diamond \langle u\rangle & \langle\mathsf{rec}\rangle &:= \eta(\kappa(\lambda a.\eta(\kappa(\lambda f.\eta(\kappa(\mathsf{rec}^{\diamond}(a, f))))))) \\ \langle\mathsf{pair}\rangle &:= \eta(\lambda a.\eta(\lambda b.\eta(\mathsf{pair}(a, b)))) & \langle\mathsf{inj}_i\rangle &:= \eta(\eta \circ \mathsf{inj}_i) \\ \langle\mathsf{pr}_i\rangle &:= \eta(\kappa(\mathsf{pr}_i)) & \langle\mathsf{case}\rangle &:= \eta(\kappa(\lambda f.\eta(\kappa(\lambda g.\eta(\kappa(\mathsf{case}(f, g))))))) \end{array}$$

where $\mathsf{rec}^{\diamond} : \sigma \to (\mathsf{J}\mathbb{N} \to \mathsf{J}(\mathsf{J}\sigma \to \mathsf{J}\sigma)) \to \mathbb{N} \to \mathsf{J}\sigma$ *is defined by*

$$\begin{aligned} \mathsf{rec}^{\diamond}(a, f, 0) &:= \eta(a) \\ \mathsf{rec}^{\diamond}(a, f, n+1) &:= f(\eta n) \diamond \mathsf{rec}^{\diamond}(a, f, n). \end{aligned}$$

▶ **Definition 13** (Kuroda-style monadic translation)**.** *We assign to each type* ρ *a type* $\mathsf{J}[\rho]$ *where* $[\rho]$ *is defined as follows*

$$\begin{aligned} [\mathbb{N}] &:= \mathbb{N} & [\sigma \times \tau] &:= [\sigma] \times [\tau] \\ [\sigma \to \tau] &:= [\sigma] \to \mathsf{J}[\tau] & [\sigma + \tau] &:= [\sigma] + [\tau]. \end{aligned}$$

Each term $\Gamma \vdash t : \rho$ *is translated to a term* $[\Gamma] \vdash [t] : \mathsf{J}[\rho]$*, where* $[\Gamma]$ *is a new context assigning each* $x : \sigma \in \Gamma$ *to a fresh variable* $\bar{x} : [\sigma]$*, and* $[t]$ *is defined inductively as follows:*

$$\begin{aligned} [x] &:= \eta(\bar{x}) & [0] &:= \eta(0) \\ [\lambda x.t] &:= \eta(\lambda \bar{x}.[t]) & [\mathsf{suc}] &:= \eta(\eta \circ \mathsf{suc}) \\ [tu] &:= [t] \bullet [u] & [\mathsf{rec}] &:= \eta(\lambda a.\eta(\lambda f.\eta(\mathsf{rec}^{\bullet}(a, f)))) \\ [\mathsf{pair}] &:= \eta(\lambda a.\eta(\lambda b.\eta(\mathsf{pair}(a, b)))) & [\mathsf{inj}_i] &:= \eta(\eta \circ \mathsf{inj}_i) \\ [\mathsf{pr}_i] &:= \eta(\eta \circ \mathsf{pr}_i) & [\mathsf{case}] &:= \eta(\lambda f.\eta(\lambda g.\eta(\mathsf{case}(f, g)))) \end{aligned}$$

where $\mathsf{rec}^{\bullet} : \sigma \to (\mathbb{N} \to \mathsf{J}(\sigma \to \mathsf{J}\sigma)) \to \mathbb{N} \to \mathsf{J}\sigma$ *is defined by*

$$\begin{aligned} \mathsf{rec}^{\bullet}(a, f, 0) &:= \eta(a) \\ \mathsf{rec}^{\bullet}(a, f, n+1) &:= fn \bullet \mathsf{rec}^{\bullet}(a, f, n). \end{aligned}$$

Unital Anti-Unification: Type and Algorithms

David M. Cerna
Johannes Kepler Univerisity Linz, Austria
david.cerna@risc.jku.at

Temur Kutsia
Johannes Kepler Univerisity Linz, Austria
kutsia@risc.jku.at

Abstract

Unital equational theories are defined by axioms that assert the existence of the unit element for some function symbols. We study anti-unification (AU) in unital theories and address the problems of establishing generalization type and designing anti-unification algorithms. First, we prove that when the term signature contains at least two unital functions, anti-unification is of the nullary type by showing that there exists an AU problem, which does not have a minimal complete set of generalizations. Next, we consider two special cases: the linear variant and the fragment with only one unital symbol, and design AU algorithms for them. The algorithms are terminating, sound, complete, and return tree grammars from which the set of generalizations can be constructed. Anti-unification for both special cases is finitary. Further, the algorithm for the one-unital fragment is extended to the unrestricted case. It terminates and returns a tree grammar which produces an infinite set of generalizations. At the end, we discuss how the nullary type of unital anti-unification might affect the anti-unification problem in some combined theories, and list some open questions.

2012 ACM Subject Classification Theory of computation → Rewrite systems; Theory of computation → Tree languages; Theory of computation → Equational logic and rewriting

Keywords and phrases Anti-unification, tree grammars, unital theories, collapse theories

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.26

Supplementary Material Implementation: `https://github.com/Ermine516/UnitAU`

Funding Supported by the Austrian Science Fund (FWF) under the project P 28789-N32.

1 Introduction

We consider the equational theory of function symbols with unit element (also know as identity), U, which is defined by the axioms $f(x, \epsilon_f) \approx x$ and $f(\epsilon_f, x) \approx x$, where ϵ_f is a special constant, the unit element, associated with the function f. These axioms state that the function symbol f is *unital* and that its unit is ϵ_f. We refer to such theories, containing only these type of axioms, as *unital theories.* This property is ubiquitous in algebra, and is essential to the two basic arithmetic operations + and $\cdot$ as well as the union ($\cup$) and intersection ($\cap$) operations on sets. Furthermore, it is an example of a regular collapse theory [16], which means that the variable sets of both sides of the defining axiom(s) are the same (the regularity property), and it contains an axiom of the form $t \approx x$, where t is a non-variable term and x is a variable (the collapse property). Besides idempotency [8, 10], it is the simplest well-known such theory.

Unification and matching in unital theories has been shown to be NP-complete [17]. Otherwise, investigations concerning unital unification mostly focused on its combination with well known equational theories such as associativity (A), commutativity (C), idempotency (I), see, e.g., [2] for a survey.

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 26; pp. 26:1–26:20
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

As for anti-unification in unital theories, one of the earliest examples is generalization in free monoids [7]. More recent work [1] considers problems over arbitrary term alphabets with some binary symbols being unital, and proposes a modular algorithm for anti-unification in A, C, U theories and their combinations. The set of generalizations computed by the unital anti-unification algorithm there is not complete in general (as one can see from Example 16 below), but completeness would hold if one restricts the result to linear generalizations.

The problems we address in this paper concern the unital anti-unification type and algorithms. We prove that when the term signature contains at least two unital functions, anti-unification is of type zero (nullary) by showing that there exists an AU problem which does not have a minimal complete set of generalizations. Next, we consider two special cases: the linear variant and one-unital fragment and design algorithms for them incrementally: The one-unital fragment algorithm is obtained by extending the rule set used in the linear variant algorithm. The latter uses a modification of rules from [1]. The algorithms are terminating, sound, complete, and return tree grammars from which a set of generalizations can be constructed. For the linear variant, the language of generalizations generated by the grammar is finite. In the one-unital fragment, the language might be infinite, but it contains a finite minimal complete set of generalizations. It follows that both linear and one-unital anti-unification are finitary.

The algorithm for one-unital fragment is further extended for the unrestricted case. It terminates and returns a tree grammar which produces an infinite set of generalizations. It remains to be shown whether this set is always complete or not. At the end of the paper, we also discuss how the nullary type of unital anti-unification might affect the problems in theories that combine U with the properties such as A, C, or I.

Concerning applications, anti-unification has been used for recursion scheme detection in functional programs [4], inductive synthesis of recursive functions [15], learning fixes from software code repositories [3, 14], and for preventing bugs and misconfiguration [11], just to name a few. Given the prominence of algebraic structures, whose equational theory includes unit axioms, in programming language theory, understanding of anti-unification in the presence of such axioms is essential to future progress in this area. As an example of a possible application of this work, modern pure functional programming languages, such as Haskell, heavily rely on monads which are higher-order AU-functions. Clone analysis of code fragments which contain multiple monads used in conjunction would suffer from the nullary type of unital anti-unification. However, restricted procedures, especially for the linear variant, can provide useful substitutes to the less well behaved general procedure. Combining unit axioms with a higher-order term signature was partially address in [9].

The unital anti-unification algorithms described in the paper are implemented and can be accessed at `https://github.com/Ermine516/UnitAU`.

2 Preliminaries

We assume familiarity with the basic notions of unification theory, see, e.g., [2].

Terms and substitutions

We consider a ranked alphabet $\mathcal{A}$, consisting of the set $\mathcal{F}$ of function symbols with fixed arity and the set of variables $\mathcal{V}$. A term t over $\mathcal{A}$ is defined as $t ::= x \mid f(t_1, \dots, t_n)$, where $x \in \mathcal{V}$ and $f \in \mathcal{F}$ with the arity $n \geq 0$. The set of terms over the alphabet $\mathcal{A}$ is denoted by $\mathcal{T}(\mathcal{A})$. Nullary function symbols are called constants. We denote variables by x, y, z, u, v, constants by a, b, c, d, function symbols f, g, h, and terms by s, t, r. We denote the set of variables appearing in a term t by $var(t)$. The depth of a term t is defined inductively as $dep(x) = dep(a) = 1$ for variables and constants, and $dep(f(t_1, \ldots, t_n)) = \max\{dep(t_1), \ldots, dep(t_n)\}+1$ otherwise. The number of occurrences of s in t is defined inductively as follows $occ(s, s) = 1$, $occ(s, a) = occ(s, x) = 0$ if $x \neq a$ and $s \neq x$, $occ(s, f(t_1, \ldots, t_n)) = \sum_i occ(s, t_i)$.

The set of *positions* of a term t, denoted by $pos(t)$, is the set of strings of positive integers, defined as $pos(x) = \{\epsilon\}$ and $pos(f(t_1, \ldots, t_n)) = \{\epsilon\} \cup \bigcup_{i=1}^{n} \{i.p \mid p \in pos(t_i)\}$, where ϵ stands for the empty string. If p is a position in a term s and t is a term, then $s|_p$ denotes the subterm of s at position p and $s[t]_p$ denotes the term obtained from s by replacing the subterm $s|_p$ with t. The *head* of a term t is defined as $head(x) = x$ and $head(f(t_1, \ldots, t_n)) = f$.

A *substitution* is a mapping from variables to terms such that all but finitely many variables are mapped to themselves. Lower case Greek letters are used to denote them, except the identity substitution, which is denoted by Id. They are extended to terms in the usual way and we use the postfix notation for that, writing $t\sigma$ for an *instance* of a term t under a substitution σ. The *composition* of substitutions σ and ϑ, written as juxtaposition $\sigma\vartheta$, is the substitution defined as $x(\sigma\vartheta) = (x\sigma)\vartheta$ for all variables x.

The *domain* of a substitution σ is the set of variables which are not mapped to themselves by σ: $dom(\sigma) := \{x \mid x\sigma \neq x\}$. The restriction of σ to a set of variables X, denoted $\sigma|_X$, is the substitution defined as $x(\sigma|_X) = x\sigma$ if $x \in X$ and $x(\sigma|_X) = x$ otherwise.

A *binding* is a pair of a variable and a term, written as $x \mapsto t$. To explicitly write substitutions, we use the standard convention representing a substitution σ as a finite set of bindings $\{x \mapsto x\sigma \mid x \in dom(\sigma)\}$. *Application* of σ to a set of bindings B, written $B\sigma$, is defined as $B\sigma = \{x \mapsto t\sigma \mid x \mapsto t \in B\}$.

Equational anti-unification

Every function symbol f will have an associated set of axioms, denoted by $Ax(f)$. If $Ax(f)$ is empty, then f does not have any associated properties and is called *free*. Otherwise, $Ax(f) \subseteq \{\mathsf{A}, \mathsf{C}, \mathsf{U}, \mathsf{I}\}$ where A is *associativity*, i.e., $f(t_1, f(t_2, t_3)) \equiv f(f(t_1, t_2), t_3)$ for all t_1, t_2, t_3; C is *commutativity*, i.e., $f(t_1, t_2) \equiv f(t_2, t_1)$ for all t_1, t_2; U is *unital*, i.e., $f(t, \epsilon_f) \equiv f(\epsilon_f, t) \equiv t$ for all t, where ϵ_f is the unique unit element associated with the function constant f; and I is *idempotency*, i.e., $f(t, t) \equiv t$ for all t. Note that in these cases, only binary function symbols have equational properties. In the case of unit element, only function constants with arity 0 can be ϵ_f. For each $\mathcal{E} \subseteq \{\mathsf{A}, \mathsf{C}, \mathsf{U}, \mathsf{I}\}$ we denote the equational theory generated by $\mathcal{E}$ by $\approx_{\mathcal{E}}$. For particular equational theories such as U we can denote which function constants have this property, writing, e.g., $\approx_{\mathsf{U}(f,g,\ldots)}$. The majority of this paper focuses on *unital equational theories*. However, in later sections we consider combinations between unital theories and the other above mentioned theories.

In the rest of the paper, every non-unital function symbol is free unless otherwise specified.

We say that a term is in *unital normal form* (*U-normal form*) if it does not contain a subterm of the form $f(t, \epsilon_f)$ or $f(\epsilon_f, t)$ for any unital symbol f. To get an U-normal form of a term, all the subterms of the form $f(t, \epsilon_f)$ and $f(\epsilon_f, t)$ are replaced by t repeatedly as long as possible, for each unital symbol f. We write $nf_U(s)$ for the U-normal form of s, and for a set of terms S, $nf_U(S)$ denotes the set $nf_U(S) := \{nf_U(s) \mid s \in S\}$.

A term r is *more general* than s modulo $\mathcal{E}$ (r is an *$\mathcal{E}$-generalization* of s) if there exists a substitution σ such that $r\sigma \approx_{\mathcal{E}} s$. It is written as $r \preceq_{\mathcal{E}} s$. The relation $\preceq_{\mathcal{E}}$ is a quasi-ordering. Its strict part is denoted by $\prec_{\mathcal{E}}$, and the equivalence relation it induces by $\simeq_{\mathcal{E}}$.

Given two terms t and s, and their generalization r, we say that it is their *least general generalization* modulo $\mathcal{E}$ ($\mathcal{E}$-lgg or just lgg in short), if there is no generalization r' of t and s which satisfies $r \prec_{\mathcal{E}} r'$.

A *minimal and complete set of $\mathcal{E}$-generalizations* of two terms t and s is the set G with the following three properties:

1. Each element of G is an $\mathcal{E}$-generalization of t and s (soundness of G).
2. For each $\mathcal{E}$-generalization r' of t and s, there exists $r \in G$ such that $r' \preceq_{\mathcal{E}} r$, i.e., r is less general than r' modulo $\mathcal{E}$ (completeness of G).
3. No two distinct elements of G are $\preceq_{\mathcal{E}}$-comparable: If $r_1, r_2 \in G$ such that $r_1 \preceq_{\mathcal{E}} r_2$, then $r_1 = r_2$ (minimality of G).

We write $mcsg_{\mathcal{E}}(t, s)$ for the minimal complete set of $\mathcal{E}$-generalizations of t and s if it exists.

Often we just say generalization, lgg, etc. instead of $\mathcal{E}$-generalization, $\mathcal{E}$-lgg and so on when the equational theory being discussed is clear from context.

The *Anti-unification type* of equational theories are defined similarly (but dually) to unification type, based on the existence and cardinality of a minimal complete set of generalizations. We assume here no restriction on the signature, i.e., the problems and generalizations may contain arbitrary function symbols. Then the types are defined as follows:

- Unitary type: Any anti-unification problem in the theory has a singleton *mcsg*.
- Finitary type: Any anti-unification problem in the theory has an *mcsg* of finite cardinality, for at least one problem having it greater than 1.
- Infinitary type: For any anti-unification problem in the theory there exists an *mcsg*, and for at least one problem this set is infinite.
- Nullary type (or type zero): There exists an anti-unification problem in the theory which does not have an *mcsg*, i.e., every complete set of generalizations for this problem contains two distinct elements such that one is more general than the other.

For each of these types, there exists a corresponding instance of an equational theory. The syntactic first-order anti-unification [12,13] is unitary; commutative anti-unification [1] is finitary; idempotent anti-unification is infinitary [8]; nominal anti-unification with infinitely many atoms is nullary [5,6]. In this paper we illustrate that unital anti-unification is nullary over a term alphabet with as least two unital function symbols, and study anti-unification type for some other theories, which are combined with the unital one.

We represent anti-unification problems in the form of *$\mathcal{E}$-anti-unification triples* ($\mathcal{E}$-AUTs). An $\mathcal{E}$-AUT is a triple of a variable and two terms, written as $x : t \triangleq_{\mathcal{E}} s$. Here x is a fresh variable which stands for the most general $\mathcal{E}$-generalization of t and s. Any $\mathcal{E}$-generalization r of t and s is then an instance of x, witnessed by a substitution σ such that $x\sigma \approx_{\mathcal{E}} r$.

Sometimes, when we want to anti-unify s and t, we simply say that we have an *anti-unification problem* (AUP) modulo $\mathcal{E}$, $s \triangleq_{\mathcal{E}} t$.

In all the notations, we omit $\mathcal{E}$ when it is clear from the context.

Regular tree grammars

A *regular tree grammar* is a tuple $\langle \alpha, N, T, R \rangle$, where the symbol α is called the *axiom*, N is the set of *non-terminal* symbols with arity 0 such that $\alpha \in N$, T is the set of terminal symbols with $T \cap N = \emptyset$, and R is the set of production rules of the form $\beta \mapsto t$ where $\beta \in N$ and $t \in \mathcal{T}(T \cup N)$. Given a regular tree grammar $\mathcal{G} = \langle \alpha, N, T, R \rangle$, the *derivation relation* $\rightarrow_G$ is a relation on pairs of terms of $\mathcal{T}(T \cup N)$ such that $s \rightarrow_G t$ if and only if there exists a position p in s and a rule $\nu \rightarrow r \in R$ such that $s|_p = \nu$ and $t = s[r]_p$. The *language generated by $\mathcal{G}$* from the nonterminal β is the set of terms $\mathcal{L}(\mathcal{G}, \beta) := \{t \mid t \in \mathcal{T}(T) \text{ and } \beta \rightarrow_G^+ t\}$, where $\rightarrow_G^+$ is the transitive closure of the relation $\rightarrow_G$. The language generated by $\mathcal{G}$ is defined as the language generated bt $\mathcal{G}$ from α: $\mathcal{L}(\mathcal{G}) := \mathcal{L}(\mathcal{G}, \alpha)$. Given a grammar $\mathcal{G}$, the set of nonterminals of $\mathcal{G}$ that appear in a syntactic object (term, rule, AUT, etc.) O is denoted by $\mathit{nter}(\mathcal{G}, O)$. For a grammar $\mathcal{G}$, the set of nonterminals that *can be reached* from a nonterminal ν, denoted by $\mathit{reach}(\mathcal{G}, \nu)$, is defined as $\mathit{reach}(\mathcal{G}, \nu) := \{\mu \mid \nu \to^*_{\mathcal{G}} t \text{ and } \mu \in \mathit{nter}(\mathcal{G}, t)\}$, where $\to^*_{\mathcal{G}}$ is reflexive and transitive closure of $\to_{\mathcal{G}}$. When the grammar is clear from the context, we write $\to$ instead of $\to_{\mathcal{G}}$.

Our next step is to connect sets of bindings and regular tree grammars, defining how to construct grammars from binding sets. The reasoning behind such a correspondence is the following: our goal is to represent complete sets of unital generalizations by finite means with the help of regular tree grammars. Hence, we want to develop a U-generalization algorithm which gives us such a representation. The mentioned correspondence will make this task easier, because it will allow us to design a simpler algorithm. It computes a set of bindings, from which one can directly construct the desired grammar, based on the correspondence we define below in Definition 1.

We assume that each nonempty set of bindings B contains a designated binding, which we call the *root binding*. Its left hand side is called *the root* of B. It is required that the root occurs only once in the grammar, in the left hand side of the root binding.

▶ **Definition 1** (Regular tree grammar corresponding to a set of bindings)**.** *Given a (nonempty) set of bindings B, the* corresponding regular tree grammar *$\mathcal{G}(B) = \langle \alpha, N, T, B \rangle$ is defined by the following construction:*

- *The axiom α is the root of B.*
- *$N = \{x \mid x \mapsto r \in B$ for some $r\}$.*
- *$T = F \cup V$, where F is the set of all function symbols that appear in terms of the right hand sides of B, and $V = \{\mathit{var}(r) \mid x \mapsto r \in B$ for some $x\} \setminus N$.*

The language of a tree grammar $\mathcal{G}$ is denoted by $\mathcal{L}(\mathcal{G})$.

A motivating Example

Let us consider the term $g(f(a, c), a) \triangleq g(c, b)$ where $Ax(g) = \emptyset$ and $Ax(f) = \{\mathsf{U}\}$. Using the methods discussed in [1] the computed generalization is $g(f(x, c), y)$. This seems reasonable because after decomposing $g(f(a, c), a) \triangleq g(c, b)$ once, we get two AUPs $f(a, c) \triangleq c$ and $a \triangleq b$. The latter is solvable while the former can benefit from a single application of unit introduction, i.e. $f(a, c) \triangleq f(\epsilon_f, c)$, resulting in the AUPs $a \triangleq \epsilon_f$ and $c \triangleq c$. However, if we apply unit introduction to $a \triangleq b$ twice, resulting in $f(a, \epsilon_f) \triangleq f(\epsilon_f, b)$, we can merge variables and get the generalization $g(f(x, c), f(x, y))$ which is less general than $g(f(x, c), y)$. This observation motivated us to investigate the type in greater detail because it seems to imply the possibility of an arbitrary number of variable introductions and merges.

3 General case: unital anti-unification is nullary

We formulate the first main result of this paper: generalization in theories with at least two unital function symbols is of type zero.

In this section all terms are taken from the set $\mathcal{T}(\{f, g, \epsilon_f, \epsilon_g\}, \mathcal{V})$, where both f and g are unital with units ϵ_f and ϵ_g respectively. That means, we have no other function symbols except f, g, ϵ_f, and ϵ_g. Furthermore, we will denote generalizations by bold face $\mathbf{g}$.

▶ **Definition 2.** *Let $\mathbf{g}$ be a generalization in U-normal form of $t \triangleq s$. We refer to σ_1 and σ_2 as* generalizing substitutions *of $\mathbf{g}$ if $\mathbf{g}\sigma_1 \approx_U t$, $\mathbf{g}\sigma_2 \approx_U s$, and for every $\{x \mapsto u\} \in \sigma_i$, for $i \in \{1, 2\}$, u is in U-normal form.*

▶ **Definition 3.** *Let* $\mathbf{g}$ *be a generalization in U-normal form of* $t \triangleq s$*, and let* σ_1 *and* σ_2 *be generalizing substitutions. We say that* $\mathbf{g}$ *is in* reduced form *if the following conditions hold:*

1. *For every* $x \in \mathit{var}(\mathbf{g})$*,* $x\sigma_1 \not\approx_{\mathsf{U}} x\sigma_2$*.*
2. *For all* $x, y \in \mathit{var}(\mathbf{g})$ *either* $x = y$*, or for some* $\theta \in \{\sigma_1, \sigma_2\}$*,* $x\theta \not\approx_{\mathsf{U}} y\theta$*.*

▶ **Theorem 4.** *There exists a reduced generalization* $\mathbf{g}$ *of* $\epsilon_f \triangleq \epsilon_g$ *such that* $\mathbf{g}$ *is not equal modulo* U *to a variable.*

Proof. Take $\mathbf{g} = f(x, g(x, y))$. Then $\sigma_1 = \{x \mapsto \epsilon_f, y \mapsto \epsilon_g\}$ and $\sigma_2 = \{x \mapsto \epsilon_g, y \mapsto \epsilon_f\}$ are the generalizing substitutions. Obviously, $\mathbf{g}$ is not equal modulo U to a variable. ◀

▶ **Theorem 5.** *Any reduced generalization of* $\epsilon_f \triangleq \epsilon_g$ *is either a variable or contains two distinct variables (maybe with multiple occurrences).*

Proof. Let $\mathbf{g}$ be a reduced generalization of $\epsilon_f \triangleq \epsilon_g$, and σ_1 an σ_2 be generalizing substitutions. If $\mathbf{g}$ is a variable, the theorem trivially holds. By Theorem 4, there exist also nonvariable reduced generalizations of $\epsilon_f \triangleq \epsilon_g$. Notice that for all $x \in \mathit{var}(\mathbf{g})$ we have either (a) $x\sigma_1 = \epsilon_f$ and $x\sigma_2 = \epsilon_g$, or (b) $x\sigma_1 = \epsilon_g$ and $x\sigma_2 = \epsilon_f$, for otherwise either $\mathbf{g}$ would not be a generalization of $\epsilon_f \triangleq \epsilon_g$ (we would be introducing new symbols not occurring in the initial terms), or for some $\{x \mapsto s\} \in \sigma_i$, $i \in \{1, 2\}$, s would not be in U-normal form. If the latter is the case we may just replace the offending binding by $\{x \mapsto s'\}$ where s' is the U-normalized version of s. But since $\mathbf{g}$ is reduced, we do not have two distinct $x, y \in \mathit{var}(\mathbf{g})$ with $x\sigma_i \approx_{\mathsf{U}} y\sigma_i$. Hence, when $\mathbf{g}$ is not a variable, then it must contain two distinct variables: one that satisfies (a), and the other one that satisfies (b). ◀

▶ **Theorem 6.** *For every generalization* $\mathbf{g}$ *in U-normal form of* $\epsilon_f \triangleq \epsilon_g$ *there exists a substitution* ϑ *such that* $\mathbf{g}\vartheta$ *is a reduced generalization of* $\epsilon_f \triangleq \epsilon_g$*.*

Proof. Let σ_1 and σ_2 be its generalizing substitutions. If $\mathbf{g}$ is reduced, then the theorem trivially holds and $\vartheta = \mathit{Id}$. Assume $\mathbf{g}$ is not in reduced form. (Therefore, it can not be a variable.) We will construct ϑ as a composition of two substitutions ϑ_1 and ϑ_2, which we define below. Since $\mathbf{g}$ is not reduced, it violates one of the two conditions of Definition 3.

If $\mathbf{g}$ does not violate the first condition, we take $\vartheta_1 = \mathit{Id}$ and continue with checking the second one. If $\mathbf{g}$ violates the first condition, then there exists $x \in \mathit{var}(\mathbf{g})$ such that $x\sigma_1 = x\sigma_2$, i.e., x is an overgeneralization. We can assume that $x\sigma_1 = x\sigma_2 = \epsilon_w$, where w is either f or g, because if $\mathit{dep}(x\sigma_1) > 1$, then either $x\sigma_1$ is not in U-normal form or $\mathbf{g}\sigma_1 \not\approx_{\mathsf{U}} \epsilon_w$.

Assume $\{x_1, \ldots, x_n, y_1, \ldots, y_m\} \subseteq \mathit{var}(\mathbf{g})$ are all those variables in $\mathbf{g}$ that violate the first condition of Definition 3 such that $x_i\sigma_1 = x_i\sigma_2 = \epsilon_f$ for all $1 \le i \le n$, and $y_j\sigma_1 = y_j\sigma_2 = \epsilon_g$ for all $1 \le j \le m$. Then we take $z_1, z_2 \notin \mathit{var}(\mathbf{g})$ and consider three substitutions

$$\vartheta_1 = \{x_1 \mapsto g(z_1, z_2)\} \cdots \{x_n \mapsto g(z_1, z_2)\}\{y_1 \mapsto f(z_1, z_2)\} \cdots \{y_m \mapsto g(z_1, z_2)\},$$
$$\sigma_1' = \{z_1 \mapsto \epsilon_f, z_2 \mapsto \epsilon_g\}\sigma_1, \qquad \sigma_2' = \{z_1 \mapsto \epsilon_g, z_2 \mapsto \epsilon_f\}\sigma_2.$$

$\mathbf{g}\vartheta_1$ is a generalization of $\epsilon_f \triangleq \epsilon_g$ and σ_1' and σ_2' are generalizing substitutions, because

$$\mathbf{g}\vartheta_1\sigma_1' = \mathbf{g}\{x_1 \mapsto \epsilon_f\} \cdots \{x_n \mapsto \epsilon_f\}\{y_1 \mapsto \epsilon_g\} \cdots \{y_m \mapsto \epsilon_g\}\sigma_1 = \mathbf{g}\sigma_1 = \epsilon_f.$$
$$\mathbf{g}\vartheta_1\sigma_2' = \mathbf{g}\{x_1 \mapsto \epsilon_f\} \cdots \{x_n \mapsto \epsilon_f\}\{y_1 \mapsto \epsilon_g\} \cdots \{y_m \mapsto \epsilon_g\}\sigma_2 = \mathbf{g}\sigma_2 = \epsilon_g.$$

However, in $\mathbf{g}\vartheta_1$ we do not have variables that violate the first condition of Definition 3: all such variables from $\mathbf{g}$ are now replaced by terms containing z_1 and z_2 only, and these new variables do not violate the condition as one can see from σ_1' and σ_2'.

Hence, we got $\mathbf{g}\vartheta_1$ that does not violate the first condition of Definition 3. If $\mathbf{g}\vartheta_1$ fulfills the second one too, then we take $\vartheta_2 = \mathit{Id}$ and obtain $\vartheta = \vartheta_1$. Otherwise there exist two distinct variables $x, y \in \mathit{var}(\mathbf{g}\vartheta_1)$ such that $x\sigma_i \approx_{\mathsf{U}} y\sigma_i$, $i = 1, 2$. We take the renaming substitution $\{x \mapsto y\}$ and obtain $\mathbf{g}\vartheta_1\{x \mapsto y\}$, which is obviously a generalization again, but replaces the violating variable pair by a single variable. We can repeat this process iteratively for all variable pairs violating the second condition of Definition 3. Let ϑ_2 be the composition of all renaming substitutions used in this process. The obtained generalization $\mathbf{g}\vartheta_1\vartheta_2$ is in reduced form. Taking $\vartheta = \vartheta_1\vartheta_2$ finishes the proof. ◀

From this proof we see that if $\mathbf{g}$ is a reduced generalization of $\epsilon_f \triangleq \epsilon_g$ with variables $\mathit{var}(\mathbf{g}) = \{x, y\}$, then $\sigma_1 = \{x \mapsto \epsilon_f, y \mapsto \epsilon_g\}$ and $\sigma_2 = \{x \mapsto \epsilon_g, y \mapsto \epsilon_f\}$ can be taken as the generalizing substitutions.

▶ **Theorem 7.** *Let $\mathbf{g}$ be a reduced generalization of $\epsilon_f \triangleq \epsilon_g$. Then there exists a reduced generalization $\mathbf{g}'$ of $\epsilon_f \triangleq \epsilon_g$ such that $\mathbf{g} \prec_U \mathbf{g}'$.*

Proof. By Theorem 5, since $\mathbf{g}$ is reduced, it is either a single variable x, or contains exactly two variables x and y.

First assume $\mathbf{g} = x$. Then $\mathbf{g}' = \mathbf{g}\{x \mapsto f(x, g(x, y))\} = f(x, g(x, y))$ is also a reduced generalization of $\epsilon_f \triangleq \epsilon_g$. However, for no θ we have $\mathbf{g}'\theta \approx_{\mathsf{U}} \mathbf{g}$. Hence, $\mathbf{g} \prec_U \mathbf{g}'$ in this case.

Now let $\mathbf{g}$ be such that $\{x, y\} = \mathit{var}(\mathbf{g})$ and $\mathbf{g}' = \mathbf{g}\{x \mapsto f(x, g(x, y))\}$. Furthermore, let $\mathit{occ}(x, \mathbf{g}) = n$ and $\mathit{occ}(y, \mathbf{g}) = m$. Then we get $\mathit{occ}(x, \mathbf{g}') = 2n$ and $\mathit{occ}(y, \mathbf{g}') = n + m$. By the proof of Theorem 5, $n > 0$ and $m > 0$. Assume by contradiction that $\mathbf{g} \not\prec_U \mathbf{g}'$, i.e. there exists $\theta = \{x \mapsto t, y \mapsto s\}$ such that $\mathbf{g}\{x \mapsto f(x, g(x, y))\}\theta = \mathbf{g}$.

If $x \in \mathit{var}(\mathbf{g}'\theta|_x)$ then $x \in \mathit{var}(t)$ implies that $\mathit{occ}(x, \mathbf{g}'\theta|_x) \geq 2n$. Thus, $x \notin \mathit{var}(t)$. This implies that $x \in \mathit{var}(\mathbf{g}'\theta)$ iff $x \in \mathit{var}(s)$. Therefore, $\mathit{occ}(x, \mathbf{g}'\theta) \geq n + m$. On the other hand, $\mathit{occ}(x, \mathbf{g}'\theta) = \mathit{occ}(x, \mathbf{g}) = n$ and from $n \geq n + m$ we get $m = 0$. But it is a contradiction with $m > 0$.

We can apply similar reasoning to the case when $\mathbf{g}' = \mathbf{g}\{y \mapsto f(y, g(y, x))\}$. Hence, $\mathbf{g} \prec_U \mathbf{g}'$ also when $\mathbf{g}$ contains exactly two variables. ◀

▶ **Theorem 8.** *Let $\mathcal{C}$ be a complete set of generalizations of $\epsilon_f \triangleq \epsilon_g$ which are in U-normal form. Then $\mathcal{C}$ contains $\mathbf{g}$ and $\mathbf{g}'$ such that $\mathbf{g} \prec_U \mathbf{g}'$.*

Proof. Let $\mathbf{g} \in \mathcal{C}$. By Theorem 6, $\mathbf{g}\vartheta$ a reduced generalization of $\epsilon_f \triangleq \epsilon_g$ for some ϑ. By Theorem 7 there exists a substitution φ such that $\mathbf{g}\vartheta \prec_U \mathbf{g}\vartheta\varphi$ and $\mathbf{g}\vartheta\varphi$ is a reduced generalization of $\epsilon_f \triangleq \epsilon_g$. By completeness of the set $\mathcal{C}$, there exists a substitution μ such that $\mathbf{g}\vartheta\varphi\mu \in \mathcal{C}$. Taking $\mathbf{g}' = \mathbf{g}\vartheta\varphi\mu$, we get $\mathbf{g}, \mathbf{g}' \in \mathcal{C}$ and $\mathbf{g} \prec_U \mathbf{g}'$. ◀

▶ **Corollary 9.** *Unital anti-unification is nullary.*

Proof. Follows from Theorem 8. ◀

In the rest of the paper we consider two special cases of unital anti-unification for which minimal complete set of generalizations exist, i.e., which are not nullary. These special cases and the linear variant and the fragment with one unital symbol.

4 Linear variant

In linear variant we are looking for unital generalizations in which no variable occurs more than once. Input is not restricted. In particular, the language may contain one or more unital function symbols.

We start by formulating the rules of an algorithm which is supposed to compute linear U-generalizations. The rules transform configurations into configurations. A *configuration* is a quadruple $A; S; L; B$, where A is a set of anti-unification triples to be solved, S is a set of already solved anti-unification triples (called the store), L is a set of pairs of an anti-unification triple and a set of unit elements denoting the start of cycles in B, and B is a set of bindings, representing the generalizations "computed so far". The intuitive idea is to take the obtained B at the end and construct from it a regular tree grammar, from which one can read off each generalization. The set L is not used in the linear variant, but we will need it in later cases when introducing cycles into the constructed grammar. We elaborate on the details later, after the rules are formulated. Configurations are denoted by $\mathbf{C}$.

It is assumed that all terms in A and S are in U-normal form and if $\mathsf{U} \in Ax(f)$ then ϵ_f is the unit element of f. Also, when bindings of the form $\{x \mapsto x\}$ occur in B they will automatically be dropped. The rules are defined as follows ($\uplus$ stands for disjoint union):

Dec: **Decomposition**

$$\{x : f(s_1, \dots, s_n) \triangleq f(t_1, \dots, t_n)\} \uplus A;\ S;\ L;\ B \Longrightarrow$$
$$\{y_1 : s_1 \triangleq t_1, \dots, y_n : s_n \triangleq t_n\} \cup A;\ S;\ L;\ B\{x \mapsto f(y_1, \dots, y_n)\}$$

where $n \geq 0$, and $y_1, \dots, y_n$ are fresh variables.

Exp-U-Both: **Expansion for Unit, Both**

$$\{x : t \triangleq s\} \uplus A;\ S;\ L;\ B \Longrightarrow$$
$$\{x_1 : g(t, \epsilon_g) \triangleq s,\ x_2 : g(\epsilon_g, t) \triangleq s,\ y_1 : t \triangleq f(s, \epsilon_f), y_2 : t \triangleq f(\epsilon_f, s)\} \cup A;\ S;\ L;$$
$$B \cup \{x \mapsto x_1\} \cup \{x \mapsto x_2\} \cup \{x \mapsto y_1\} \cup \{x \mapsto y_2\},$$

where $head(t) = f \neq g = head(s)$, $\mathsf{U} \in Ax(f) \cap Ax(g)$, and x_1, x_2, y_1, y_2 are fresh variables.

Exp-U-L: **Expansion for Unit, Left**

$$\{x : t \triangleq f(s_1, s_2)\} \uplus A;\ S;\ L;\ B \Longrightarrow$$
$$\{x_1 : f(t, \epsilon_f) \triangleq f(s_1, s_2),\ x_2 : f(\epsilon_f, t) \triangleq f(s_1, s_2)\} \cup A;\ S;\ L;\ B \cup \{x \mapsto x_1\} \cup \{x \mapsto x_2\},$$

where $f \neq head(t)$, $\mathsf{U} \in Ax(f)$, $\mathsf{U} \notin Ax(head(t))$, and x_1, x_2 are fresh variables.

Exp-U-R: **Expansion for Unit, Right**

$$\{x : f(t_1, t_2) \triangleq s\} \uplus A;\ S;\ L;\ B \Longrightarrow$$
$$\{x_1 : f(t_1, t_2) \triangleq f(s, \epsilon_f), x_2 : f(t_1, t_2) \triangleq f(\epsilon_f, s)\} \cup A;\ S;\ L;\ B \cup \{x \mapsto x_1\} \cup \{x \mapsto x_2\},$$

where $f \neq head(s)$, $\mathsf{U} \in Ax(f)$, $\mathsf{U} \notin Ax(head(s))$, and x_1, x_2 are fresh variables.

Solve: **Solve**

$$\{x : s \triangleq t\} \uplus A;\ S;\ L;\ B \Longrightarrow A;\ \{x : s \triangleq t\} \cup S; L;\ B,$$

where $head(s) \neq head(t)$ and $\mathsf{U} \notin Ax(head(t)) \cup Ax((head(s)))$.

We denote this set of rules by $\mathcal{R}_{\mathsf{lin}}$. In order to compute linear U-generalizations of two terms t and s, we create an initial configuration $\{x : t \triangleq s\}; \emptyset; \emptyset; \{x_{\text{root}} \to x\}$, where x_{root} and x are fresh variables, and apply the following strategy as long as possible:

- Select an AUT $\mathbf{a}$ arbitrarily from the first component of the configuration.
- Apply a rule in $\mathcal{R}_{\mathsf{lin}}$, applicable to $\mathbf{a}$. (There is only one such rule for each $\mathbf{a}$ in $\mathcal{R}_{\mathsf{lin}}$.)
- If the applied rule is Exp-U-Both, transform all four new AUTs by the Dec rule.
- If the applied rule is Exp-U-L or Exp-U-R, transform both new AUTs by the Dec rule.

This strategy, called Step, will be used in other algorithms below as well. Therefore, we describe it in Algorithm 1. It takes a configuration and an AUT, and returns back a new configuration. In the algorithm, instead of writing "apply rule R to the configuration $\mathbf{C} = A; S; L; B$ with the AUT $\mathbf{a}$ selected in A", we simply write "apply rule R to $\mathbf{a}$".

Algorithm 1 Procedure Step.

```
Require: A configuration $\mathbf{C} = A; S; L; B$ and an AUT $\mathbf{a} = x : t \triangleq s \in A$.
if $head(t) = head(s)$ then
    Apply Dec to $\mathbf{a}$, resulting in $\mathbf{C}'$. Update $\mathbf{C} \leftarrow \mathbf{C}'$
else if $\exists f, g \in \mathcal{F} : (\mathsf{U} \in (Ax(f) \cap Ax(g)) \wedge head(s) = f \neq g = head(t))$ then
    Apply Exp-U-Both to $\mathbf{a}$ resulting in $\mathbf{C}' = \{\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3, \mathbf{a}_4\} \cup A; S; L; B'$
    Apply Dec to $\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_1, \mathbf{a}_2$ resulting in $\mathbf{C}''$. Update $\mathbf{C} \leftarrow \mathbf{C}''$
else if $head(t) \neq head(s) \wedge \exists f \in \mathcal{F} : (\mathsf{U} \in Ax(f) \wedge head(s) = f)$ then
    Apply Exp-U-L to $\mathbf{a}$ resulting in $\mathbf{C} = \{\mathbf{a}_1, \mathbf{a}_2\} \cup A; S; L; B'$
    Apply Dec to $\mathbf{a}_1, \mathbf{a}_2$ resulting in $\mathbf{C}''$. Update $\mathbf{C} \leftarrow \mathbf{C}''$
else if $head(t) \neq head(s) \wedge \exists f \in \mathcal{F} : (\mathsf{U} \in Ax(f) \wedge head(t) = f)$ then
    Apply Exp-U-R to $\mathbf{a}$ resulting in $\{\mathbf{a}_1, \mathbf{a}_2\} \cup A; S; L; B'$
    Apply Dec to $\mathbf{a}_1, \mathbf{a}_2$ resulting in $\mathbf{C}''$. Update $\mathbf{C} \leftarrow \mathbf{C}''$
else
    Apply Solve to $\mathbf{a}$ resulting in $\mathbf{C}'$. Update $\mathbf{C} \leftarrow \mathbf{C}'$
end if
return $\mathbf{C}$
```

The linear U-generalization algorithm, $\mathfrak{G}_{\mathsf{U\text{-}lin}}$, is then an iterative application of Step, as one can see in Algorithm 2.[1] However, in that work we refrained from using a tree grammar-based procedure. In Example 10 below, we apply $\mathfrak{G}_{\mathsf{U\text{-}lin}}$ to the AUP $x : g(f(a, c), a) \triangleq g(c, b)$ over the alphabet $\{f, g, a, b, c, \epsilon_f\}$, where a, b, and c are constants and g is a binary free function symbol.

Algorithm 2 Procedure $\mathfrak{G}_{\mathsf{U\text{-}lin}}$.

```
Require: A configuration $\mathbf{C} = A; S; L; B$
  while $A \neq \emptyset$ do
    $\mathbf{a} \leftarrow x : t \triangleq s \in A$
    $\mathbf{C} \leftarrow$ Step$(\mathbf{C}, \mathbf{a})$ (See Algorithm 1)
  end while
  return $\mathbf{C}$
```

▶ **Example 10.**

$$\{x : g(f(a, c), a) \triangleq g(c, b)\};\ \emptyset;\ \emptyset;\ \{x_{\text{root}} \mapsto x\} \Longrightarrow_{\mathsf{Dec}}$$

$$\{x_1 : f(a, c) \triangleq c, x_2 : a \triangleq b\};\ \emptyset;\ \emptyset;\ \{x_{\text{root}} \mapsto g(x_1, x_2)\} \Longrightarrow_{\mathsf{Exp\text{-}U\text{-}L,\, Dec}\times 2}$$

$$\{x_3 : a \triangleq \epsilon_f, x_4 : c \triangleq c, x_5 : a \triangleq c, x_6 : c \triangleq \epsilon_f, x_2 : a \triangleq b\};\ \emptyset; \emptyset;$$
$$\{x_{\text{root}} \mapsto g(x_1, x_2), x_1 \mapsto f(x_3, x_4), x_1 \mapsto f(x_5, x_6)\} \Longrightarrow_{\mathsf{Dec}}$$

$$\{x_3 : a \triangleq \epsilon_f, x_5 : a \triangleq c, x_6 : c \triangleq \epsilon_f, x_2 : a \triangleq b\};\ \emptyset;\ \emptyset;$$
$$\{x_{\text{root}} \mapsto g(x_1, x_2), x_1 \mapsto f(x_3, c), x_1 \mapsto f(x_5, x_6)\} \Longrightarrow_{\mathsf{Solve}\times 4}$$

$$\emptyset;\ \{x_3 : a \triangleq \epsilon_f, x_5 : a \triangleq c, x_6 : c \triangleq \epsilon_f, x_2 : a \triangleq b\};\ \emptyset;$$
$$\{x_{\text{root}} \mapsto g(x_1, x_2),\ x_1 \mapsto f(x_3, c),\ x_1 \mapsto f(x_5, x_6)\}$$

[1] Linear U-anti-unification is discussed in [9].

We refer to the final binding set as B. Thus, $\mathcal{L}(\mathcal{G}(B)) \approx_U \{g(f(x_3, c), x_2), g(f(x_5, x_6), x_2)\}$. Note that $g(f(x_5, x_6), x_2) \prec_{\mathsf{U}} g(f(x_3, c), x_2)$.

▶ **Theorem 11** (Termination). *The procedure $\mathfrak{G}_{\mathsf{U\text{-}lin}}$ is terminating.*

Proof. Let the depth of an AUP be $dep(x : t \triangleq s) = dep(t) + dep(s)$, and the complexity measure of a configuration $A; S; L; B$ be the multiset of depths of AUPs in A. We compare measures by multiset extension of the standard ordering on natural numbers. The extension is well-founded. After each iteration of the loop in Algorithm 2, the complexity measure of $\mathbf{C}$ strictly decreases. Hence, the algorithm terminates. ◀

Termination of $\mathfrak{G}_{\mathsf{U\text{-}lin}}$ means that any sequence of rule transformations, starting from the initial configuration, is finite: $\{x : t \triangleq s\}; \emptyset; \emptyset; \{x_{\text{root}} \mapsto x\} \Longrightarrow^* \emptyset; S; L; B$. In the terminal configuration the first component is empty, for otherwise there is always an applicable rule. The set of bindings B at the end is called the $\mathfrak{G}_{\mathsf{U\text{-}lin}}$-computed set of bindings.

▶ **Theorem 12** (Soundness). *If $\{x : t \triangleq s\}; \emptyset; \emptyset; \{x_{\text{root}} \mapsto x\} \Longrightarrow^* \emptyset; S; L; B$ is a transformation sequence of $\mathfrak{G}_{\mathsf{U\text{-}lin}}$, then for every $r \in \mathcal{L}(\mathcal{G}(B))$, $r \preceq_U t$ and $r \preceq_U s$.*

Proof. We can prove soundness by induction over the length of the derivation, based on the fact that if $\mathcal{L}(\mathcal{G}(B))$ is a set of generalizations of an AUT $x : t \triangleq s$ and $\{x : t \triangleq s\} \cup A; S; L; B \Longrightarrow A'; S'; L'; B'$ is a transformation step, then $\mathcal{L}(\mathcal{G}(B'))$ is also a set of generalizations of $x : t \triangleq s$. For a transformation with Dec rule the proof of this property is standard. For Solve rule it is obvious. For the expansion rules it follows from two facts: first, B' is obtained from B by bindings of a variable to a variable (e.g., x to x_1) and second, all new AUTs obtained by these rules are U-equivalent to the original one (e.g., an AUT whose generalization is x_1 is U-equivalent to the AUT whose generalization was x). ◀

For the set B computed by the procedure, we call $\mathcal{L}(\mathcal{G}(B))$ the *set of generalizations computed by* $\mathfrak{G}_{\mathsf{U\text{-}lin}}$.

▶ **Theorem 13** (Completeness of $\mathfrak{G}_{\mathsf{U\text{-}lin}}$). *Let s be a linear U-generalization of two terms t_1 and t_2. Then there exists a transformation sequence $\{x : t_1 \triangleq t_2\}; \emptyset; \emptyset; \{x_{\text{root}} \mapsto x\} \Longrightarrow^* \emptyset; S; L; B$ in $\mathfrak{G}_{\mathsf{U\text{-}lin}}$ such that for some term $r \in \mathcal{L}(\mathcal{G}(B))$, $s \preceq_U r$.*

Proof. See Appendix A. ◀

▶ **Theorem 14.** *The set $\mathcal{L}(\mathcal{G}(B))$ computed by $\mathfrak{G}_{\mathsf{U\text{-}lin}}$ is finite for any input.*

Proof. At every step of $\mathfrak{G}_{\mathsf{U\text{-}lin}}$ only one of the inference rules is applicable to the current configuration. None of the rules used in the $\mathfrak{G}_{\mathsf{U\text{-}lin}}$ procedure introduce cycles into the grammar. Thus, the final set of bindings produces a tree grammar with a finite language. ◀

▶ **Theorem 15.** *Linear unital anti-unification is finitary.*

Proof. By Theorem 13 & 14. ◀

5 One-unital fragment

The next special case of U-anti-unification allows arbitrary generalizations (not only linear ones), but takes input from a language with only one unital function. We call this special case a one-unital fragment, and the corresponding alphabet one-unital alphabet.

Lifting the linearity restriction leads to an extension of the rule system. If two variables generalize the same AUTs, they should be merged. Besides, cycles should be permitted in the grammar. These changes are reflected in the set of rules $\mathcal{R}_{\mathsf{one(f)}}$ given below. They will be used together with the $\mathcal{R}_{\mathsf{lin}}$ rules to solve generalization problems with one unital symbol.

One will probably notice that the cycle rules allows the construction of a grammar with an infinite language, however, as shown in Theorem 20, only a finite number of these terms are least general generalization. In some sense the cycle rules allow for the construction of more expressive tree grammars than necessary for finding the minimal complete set of generalizations. It is reasonable to expect that less expressive versions of the rules may be developed specifically for the one-unital fragment. However, as presented here we highlight the relationship between this fragment and the algorithm we present for the general procedure. Essentially in the one-unital fragment only a finite portion of the terms generated by the cycles are least general generalizations where in the general case all the terms resulting from a cycle may be ordered by generality.

Start-Cycle-U: **Cycle introduction for Unit**

$$\{x : t \triangleq s\} \uplus A;\ S;\ L;\ B \Longrightarrow \{y_1 : f(t, \epsilon_f) \triangleq f(\epsilon_f, s),\ y_2 : f(\epsilon_f, t) \triangleq f(s, \epsilon_f), y_3 : t \triangleq s\} \cup A;$$
$$S;\ \{(\{x : t \triangleq s\}, \{\epsilon_f\})\} \cup L;\ B \cup \{x \mapsto y_1\} \cup \{x \mapsto y_2\},$$

where $\mathsf{U} \in Ax(f)$, $(\{y : t \triangleq s\}, Un) \notin L$ for any y and Un, $head(t) \neq \epsilon_f$ or $head(s) \neq \epsilon_f$, $\mathsf{U} \notin Ax(head(t)) \cup Ax(head(s))$, and y_1 and y_2 are fresh variables.

Sat-Cycle-U: **Cycle Saturation for Unit**

$$\{x : t \triangleq s\} \uplus A;\ S;\ \{(\{y : t \triangleq s\}, Un)\} \cup L;\ B \Longrightarrow$$
$$\{x : t \triangleq s\} \cup A;\ S;\ (\{y : t \triangleq s\}, Un) \cup L;\ B\{x \mapsto y\} \cup \{y \mapsto x\},$$

where $x \neq y$ and $\{y \mapsto x\} \notin B$.

Merge: **Merge**

$$\emptyset;\ \{x_1 : s_1 \triangleq t_1,\ x_2 : s_2 \triangleq t_2\} \uplus S;\ L;\ B \Longrightarrow \emptyset;\ \{x_1 : s_1 \triangleq t_1\} \cup S;\ L;\ B\{x_2 \mapsto x_1\},$$

where $s_1 \approx_{\mathsf{U}} s_2$ and $t_1 \approx_{\mathsf{U}} t_2$.

For a given AUT, the Start-Cycle-U rule adds two new AUTs, which are U-equivalent to the given one. The original AUT is still present, just with a renamed generalization variable. It will be used for saturation. In Algorithm 3, we define a strategy for applying the new cycle rules. We "exhaustively" (see line 6) apply Sat-Cycle-U because applying Dec to the AUPs resulting from Start-Cycle-U may result in AUPs present in the cycle set L.

Algorithm 3 Procedure Cycle($\mathbf{C}, \mathbf{a}$).

```
Require: A configuration $\mathbf{C} = A; S; L; B$, an AUT $\mathbf{a} = x : t \triangleq s$
1: if $\exists f \in \mathcal{F} : (\mathsf{U} \in Ax(f) \wedge (\{y : t \triangleq s\}, Un) \notin L)$ then
2:     Apply Start-Cycle-U to $\mathbf{a}$ resulting in $\mathbf{C}' = \{\mathbf{a}_1, \mathbf{a}_2, x' : t \triangleq s\} \cup A; S; L'; B'$
3:     Apply Dec to $\mathbf{a}_1, \mathbf{a}_2$ resulting in $\mathbf{C}''$. Update $\mathbf{C} \leftarrow \mathbf{C}''$ and $\mathbf{a} \leftarrow x' : t \triangleq s$
4: end if
5: Exhaustively apply Sat-Cycle-U to $\mathbf{C}$ resulting in $\mathbf{C}^*$. Update $\mathbf{C} \leftarrow \mathbf{C}^*$
6: return $(\mathbf{C}, \mathbf{a})$
```

The one-unital-function anti-unification algorithm $\mathfrak{G}_{\mathsf{U(f)}}$ is a strategy of applying the rules in $\mathcal{R}_{\mathsf{lin}} \cup \mathcal{R}_{\mathsf{one(f)}}$ as defined in Algorithm 4.

Algorithm 4 Procedure for $\mathfrak{G}_{\mathsf{U(f)}}$.

```
Require: A configuration $\mathbf{C} = A; S; L; B$
  while $A \neq \emptyset$ do
    $\mathbf{a} \leftarrow x : t \triangleq s \in A$
    $(\mathbf{C}, \mathbf{a}) \leftarrow \mathsf{Cycle}(\mathbf{C}, \mathbf{a})$ (See Algorithm 3)
    $\mathbf{C} \leftarrow \mathsf{Step}(\mathbf{C}, \mathbf{a})$ (See Algorithm 1)
    Exhaustively apply Sat-Cycle-U to $\mathbf{C}$ resulting in $\mathbf{C}^*$. Update $\mathbf{C} \leftarrow \mathbf{C}^*$
  end while
  Exhaustively apply Merge to $\mathbf{C}$ resulting in $\mathbf{C}^*$. Update $\mathbf{C} \leftarrow \mathbf{C}^*$
  return $\mathbf{C}$
```

▶ **Example 16.** Observe that the AUP addressed in Example 10 is solved over an alphabet with a single unital function symbol. Now we try to solve it using $\mathfrak{G}_{\mathsf{U(f)}}$.

$$\{x : g(f(a,c),a) \triangleq g(c,b)\}; \emptyset; \emptyset; \{x_{\text{root}} \mapsto x\} \Longrightarrow_{\mathsf{Start\text{-}Cycle\text{-}U}}$$

$$\{x_1 : g(f(a,c),a) \triangleq g(c,b), x_2 : f(g(f(a,c),a),\epsilon_f) \triangleq f(\epsilon_f, g(c,b)),$$
$$x_3 : f(\epsilon_f, g(f(a,c),a)) \triangleq f(g(c,b),\epsilon_f)\}; \emptyset; \{(x : g(f(a,c),a) \triangleq g(c,b), \{\epsilon_f\})\};$$
$$\{x_{\text{root}} \mapsto x, x \mapsto x_2, x \mapsto x_3\} \Longrightarrow_{\mathsf{Dec}}$$

$$\{x_1 : g(f(a,c),a) \triangleq g(c,b), x_2 : f(g(f(a,c),a),\epsilon_f) \triangleq f(\epsilon_f, g(c,b)), x_4 : \epsilon_f \triangleq g(c,b),$$
$$x_5 : g(f(a,c),a) \triangleq \epsilon_f\}; \emptyset; \{(x : g(f(a,c),a) \triangleq g(c,b), \{\epsilon_f\})\};$$
$$\{x_{\text{root}} \mapsto x, x \mapsto x_2, x \mapsto f(x_4, x_5)\} \Longrightarrow_{\mathsf{Dec}}$$

$$\{x_1 : g(f(a,c),a) \triangleq g(c,b), x_4 : \epsilon_f \triangleq g(c,b), x_5 : g(f(a,c),a) \triangleq \epsilon_f,$$
$$x_6 : g(f(a,c),a) \triangleq \epsilon_f, x_7 : \epsilon_f \triangleq g(c,b)\}; \emptyset; \{(x : g(f(a,c),a) \triangleq g(c,b), \{\epsilon_f\})\};$$
$$\{x_{\text{root}} \mapsto x, x \mapsto f(x_6, x_7), x \mapsto f(x_4, x_5)\}, \Longrightarrow_{\mathsf{Sat\text{-}Cycle\text{-}U}}$$

$$\{x_1 : g(f(a,c),a) \triangleq g(c,b), x_4 : \epsilon_f \triangleq g(c,b), x_5 : g(f(a,c),a) \triangleq \epsilon_f,$$
$$x_6 : g(f(a,c),a) \triangleq \epsilon_f, x_7 : \epsilon_f \triangleq g(c,b)\}; \emptyset; \{(x : g(f(a,c),a) \triangleq g(c,b), \{\epsilon_f\})\};$$
$$\{x_{\text{root}} \mapsto x, x \mapsto f(x_6, x_7), x \mapsto f(x_4, x_5), x \mapsto x_1\} \Longrightarrow_{\mathsf{Dec}}$$

$$\{x_4 : \epsilon_f \triangleq g(c,b), x_5 : g(f(a,c),a) \triangleq \epsilon_f, x_6 : g(f(a,c),a) \triangleq \epsilon_f, x_7 : \epsilon_f \triangleq g(c,b),$$
$$x_8 : f(a,c) \triangleq c, x_9 : a \triangleq b\}; \emptyset; \{(x : g(f(a,c),a) \triangleq g(c,b), \{\epsilon_f\})\};$$
$$\{x_{\text{root}} \mapsto x, x \mapsto f(x_6, x_7), x \mapsto f(x_4, x_5), x \mapsto g(x_8, x_9)\} \Longrightarrow_{\mathsf{Start\text{-}Cycle\text{-}U}}$$

$$\ldots$$

$$\emptyset; \{x_{10} : \epsilon_f \triangleq g(c,b), x_{17} : g(f(a,c),a) \triangleq \epsilon_f, x_{33} : a \triangleq b, x_{40} : \epsilon_f \triangleq c, x_{76} : \epsilon_f \triangleq b,$$
$$x_{83} : a \triangleq \epsilon_f, x_{146} : a \triangleq c, x_{153} : c \triangleq \epsilon_f\}; L; \{x_{\text{root}} \mapsto x,\ x \mapsto g(f(x_{28}, x_{61}), f(x_{83}, x_{76})),$$
$$x \mapsto g(f(x_{83}, f(x_{153}, x_{40})), x_{33}), \ldots, \underline{x \mapsto g(f(x_{83}, c), f(x_{76}, x_{83}))},$$
$$x \mapsto g(f(x_{70}, x_{28}), x_{33}), x \mapsto g(f(x_{83}, f(x_{40}, x_{153})), x_{33}), \ldots,$$
$$x \mapsto g(f(x_{61}, x_{28}), f(x_{76}, x_{83})), \underline{x \mapsto g(f(x_{83}, c), f(x_{83}, x_{76}))}, \ldots\}.$$

The complete derivation contains 217 rule applications. Here we skipped most of them. The final binding set, after removing useless bindings, has 26 bindings together with a single non-terminal.[2] However, the majority of the generalizations contained in the language of this grammar are comparable. We underline the two incomparable generalizations produced by the algorithm, and refer to them as $\mathbf{g}_1$ and $\mathbf{g}_2$. In fact, the set $\{\mathbf{g}_1, \mathbf{g}_2\}$ forms

[2] See Section C for the grammar generated by our implementation.

$mcsg_{\mathsf{U}}(g(f(a,c),a),g(c,b))$.[3] Observe that they are *less general* than the terms computed in Example 10, indicating that the expansion rules are not enough to construct all non-linear generalizations even when only one function symbol is unital. We did not even need the Merge rule to obtain those *nonlinear* generalizations. The cycle rules created them.

▶ **Theorem 17** (Termination). *$\mathfrak{G}_{U(f)}$ is terminating for AUPs over an one-unital alphabet.*

Proof. To a given AUT, Cycle can apply only once, because afterwards the AUT is put in the set L. To each of the AUTs obtained by the application of the Start-Cycle-U the same rule can apply again at most once, since the further obtained AUTs are either of the form $x : \epsilon_f \triangleq \epsilon_f$, or are already placed in L. The saturation rule applies once to each element in L. Hence, the cycle rules can apply only finitely many times. The other rules strictly decrease the measure as defined in the proof of Theorem 11. It implies that $\mathfrak{G}_{\mathsf{U(f)}}$ terminates. ◀

▶ **Theorem 18** (Soundness). *If $\{x : t \triangleq s\}; \emptyset; \emptyset; \{x_{\mathrm{root}} \mapsto x\} \Longrightarrow^* \emptyset; S; L; B$ is a transformation sequence of $\mathfrak{G}_{U(f)}$ for AUPs over an one-unital alphabet, then for every $g \in \mathcal{L}(\mathcal{G}(B))$, $g \preceq_U t$ and $g \preceq_U s$.*

Proof. Similar to Theorem 12. For the cycle rules, the argument is the same as for the expansion rules. ◀

The notion of computed grammar is defined for $\mathfrak{G}_{\mathsf{U(f)}}$ in the same way as for $\mathfrak{G}_{\mathsf{U\text{-}lin}}$.

▶ **Theorem 19** (Completeness of $\mathfrak{G}_{\mathsf{U(f)}}$). *Let t_1, t_2, and s be terms over an one-unital alphabet such that s is a U-generalization of t_1 and t_2. Then there exists a transformation sequence $\{x : t_1 \triangleq t_2\}; \emptyset; \emptyset; \{x_{\mathrm{root}} \mapsto x\} \Longrightarrow^* \emptyset; S; L; B$ using the procedure $\mathfrak{G}_{U(f)}$ such that for some term $r \in \mathcal{L}(\mathcal{G}(B))$, $s \preceq_U r$.*

Proof. We assume that t_1, t_2, and s are in U-normal form. We prove the theorem by induction on $dep(t_1) + dep(t_2)$ which we denote by n. Furthermore we will denote the unital function by f and its unit by ϵ_f.
Case 1: $n = 2$, i.e., t_1 and t_2 are constants.

a) The case $dep(s) = 1$ is handled in a similar way as case 1 a) of the proof of Theorem 13.

b) Now assume as the induction hypothesis that for every generalization s of t_1 and t_2 of depth at most k, either $s \preceq_{\mathsf{U}} t_1$ and $t_1 = t_2$, or $s \preceq_{\mathsf{U}} x$ and $t_1 \neq t_2$. We show that this holds for a generalization s' of depth $k+1$. By our assumptions, $s' = f(s_1, s_2)$ for some terms s_1 and s_2.

Let σ_1 and σ_2 be substitutions such that $s'\sigma_1 = t_1$ and $s'\sigma_2 = t_2$. If $s_1\sigma_1 = s_1\sigma_2 = \epsilon_f$ (resp. if $s_2\sigma_1 = s_2\sigma_2 = \epsilon_f$), then, by the induction hypothesis, $s_2 \preceq_{\mathsf{U}} t_1$ (resp., $s_1 \preceq_{\mathsf{U}} t_1$) when $t_1 = t_2$, or $s_2 \preceq_{\mathsf{U}} x$ (resp., $s_1 \preceq_{\mathsf{U}} x$) when $t_1 \neq t_2$. Without loss of generality, this implies that for every $x \in var(s_1)$, $x\sigma_1 = x\sigma_2 = \epsilon_f$, being that f is the only unital function. Thus, there exists a substitution ϑ such that $s_1\vartheta = \epsilon_f$ and $s_2\vartheta \approx_{\mathsf{U}} s_2'$ where s_2' is still a generalization of t_1 and t_2, i.e., $s'\vartheta = s_2'$ or $s' \prec_{\mathsf{U}} s_2'$.

However, if $s_2\sigma_1 = \epsilon_f$ and $s_1\sigma_2 = \epsilon_f$, or vice versa, then additional observations are required. We assume the former case, without loss of generality.

If $t_1 = t_2$ then both s_1 and s_2 are generalizations of $t_1 \triangleq t_2$ and by the induction hypothesis $s_1 \preceq_{\mathsf{U}} t_1$ and $s_2 \preceq_{\mathsf{U}} t_1$. If $t_1 \neq t_2$ then we need to make a distinction:

[3] The algorithm in [1] computes generalizations that are more general than $\mathbf{g}_1$ and $\mathbf{g}_2$.

- **b1.** If neither t_1 nor t_2 is ϵ_f, then there exists a variable y occurring in s_1 such that $y\sigma_1 = t_1$ and a variable y' occurring in s_2 such that $y'\sigma_2 = t_2$. Note that if either t_1 or t_2 occurs in s' then s' is not a generalization $t_1 \triangleq t_2$. Let use assume that either y or y' occurs in s_2 or s_1, respectively. without loss of generality we assume that y occurs in s_2. However, this would imply that $s_2\sigma_1 = t_1$ resulting in the term $f(t_1, t_1)$ unless $t_1 = \epsilon_f$, which contradicts our assumptions. Thus, y cannot occur in s_2. This implies that there exist two substitutions σ_1' and σ_2' which coincide everywhere with σ_1 and σ_2 except on y and y' respectively. That is, $y\sigma_1' = t_1$, $y\sigma_2' = t_2$, $y'\sigma_1' = y\sigma_2' = \epsilon_f$. This implies that s_1 is a generalization of $t_1 \triangleq t_2$ which has depth $< k + 1$. Thus, $s_1 \preceq_{\mathsf{U}} x$.
- **b2.** Either t_1 or t_2 is ϵ_f. The proof is similar to the case b1 by showing that the variable generalizing the term which is not equivalent to ϵ_f cannot occur in both s_1 and s_2.

Case 2: $n > 2$.

- **a)** Assume that $t_1 = g(w_1, \dots, w_m)$ and $t_2 = g(r_1, \dots, r_m)$, such that $\mathsf{U} \notin \mathsf{Ax}(g)$. Then $\mathfrak{G}_{\mathsf{U(f)}}$ performs the following rule applications to the initial configuration:

$$
\begin{aligned}
&\{x : t_1 \triangleq t_2\}; \emptyset; \emptyset; \{x_{\text{root}} \mapsto x\} \Longrightarrow_{\mathsf{Start\text{-}Cycle\text{-}U},\ (\mathsf{Dec}\times 2)}\\
&\{x_1 : t_1 \triangleq \epsilon_f, x_2 : \epsilon_f \triangleq t_2, y_1 : t_1 \triangleq \epsilon_f, y_2 : \epsilon_f \triangleq t_2, x_3 : t_1 \triangleq t_2\}; \emptyset;\\
&\quad \{(x : t_1 \triangleq t_2, \{\epsilon_f\})\}; \{x_{\text{root}} \mapsto x, x \mapsto f(x_1, x_2), x \mapsto f(y_2, y_1)\} \Longrightarrow_{\mathsf{Sat\text{-}Cycle\text{-}U}}\\
&\{x_1 : t_1 \triangleq \epsilon_f, x_2 : \epsilon_f \triangleq t_2, y_1 : t_1 \triangleq \epsilon_f, y_2 : \epsilon_f \triangleq t_2, x_3 : t_1 \triangleq t_2\}; \emptyset;\\
&\quad \{(x : t_1 \triangleq t_2, \{\epsilon_f\})\}; \{x_{\text{root}} \mapsto x, x \mapsto f(x_1, x_2), x \mapsto f(x_2, x_1), x \mapsto x_3\} \Longrightarrow_{\mathsf{Dec}}\\
&\{x_1 : t_1 \triangleq \epsilon_f, x_2 : \epsilon_f \triangleq t_2, y_1 : t_1 \triangleq \epsilon_f, y_2 : \epsilon_f \triangleq t_2, z_1 : w_1 \triangleq r_1, \dots, z_m : w_1 \triangleq r_m\};\\
&\quad \emptyset; \{(x : t_1 \triangleq t_2, \{\epsilon_f\})\}; \{x_{\text{root}} \mapsto x, x \mapsto f(x_1, x_2), x \mapsto f(x_2, x_1), x \mapsto g(z_1, \dots, z_m)\}
\end{aligned}
$$

  The case when $s = g(s_1, \dots, s_m)$ is handled in a similar fashion as in case 2a) of the proof of Theorem 13, though we may need to apply additional Merges.
  If $s = f(s_1, s_2)$ then it may be the case, without loss of generality, that s_1 generalizes $t_1 \triangleq \epsilon_f$ and s_2 generalizes $\epsilon_f \triangleq t_2$. This case may also be handled in a similar fashion as in case 2a) of the proof of Theorem 13, though we may need to apply additional Merges.
  The final case to consider is $s = f(s_1, s_2)$ and, without loss of generality, s_2 generalizes $\epsilon_f \triangleq \epsilon_f$. This implies that for all $x \in \mathit{var}(s_2)$, $x\sigma_1 = x\sigma_2 = \epsilon_f$. Similar to case 1b) above we can reconstruct the substitutions such that $s \preceq_{\mathsf{U}} s_1$.
- **b)** Assume that $t_1 = f(w_1, w_2)$ and $t_2 = f(r_1, r_2)$, such that $\mathsf{U} \in \mathsf{Ax}(f)$. We can proceed in a similar fashion as in case 2a).
- **c)** Assume that $t_i = f(w_1, w_2)$ and $t_{(i+1 \ mod \ 2)} = g(r_1, \dots, r_k)$, where $i \in \{1, 2\}$. we can proceed in a similar fashion as in case 2b) except that we apply Exp-U-Both, Exp-U-L or Exp-U-R prior to applying Dec. ◀

▶ **Theorem 20.** *The set $\mathcal{L}(\mathcal{G}(B))$ computed by $\mathfrak{G}_{U(f)}$ contains only finitely many incomparable generalizations.*

Proof. Notice that in case 1 of Theorem 19 only one generalization exists for a given AUP whose left and right term are constant. In case 2 of Theorem 19 we show that the generalizations of a given AUP can be constructed from the generalizations of the direct subterms. The only point which makes reference to possibly infinite chains of generalizations comes at the end of case 2a). However, it was shown that this case is degenerate. Thus, we can redo the inductive construction of Theorem 19 to prove that $\mathcal{L}(\mathcal{G}(B))$ contains only finitely many non-comparable generalizations. To show that it is not unitary we need only to consider the $f(a, a) \triangleq a$ where $\mathsf{U} \in \mathit{Ax}(f)$, which has two generalizations. ◀

▶ **Theorem 21.** *Anti-unification over an one-unital alphabet is finitary.*

Proof. By Theorem 19 & 20. ◀

A problem one might have noticed concerning $\mathfrak{G}_{\mathsf{U(f)}}$ is that the computed bindings produce a verbose grammar. Most of the generalizations in the language of the grammar are comparable. However, prior to termination, it is not clear which paths may be pruned from the search. The binding set produced by $\mathfrak{G}_{\mathsf{U(f)}}$ almost always produces a tree grammar with an infinite language which contains a finite set of incomparable generalizations. Possible ways of pruning need further investigations.

6 An algorithm for unrestricted unital anti-unification

The unrestricted case generalizes one-unital anti-unification by permitting more than one unital symbol. To accommodate them in cycles, we need an extra rule, which resembles to Start-Cycle-U in that it extends the set L, but only for AUTs already existing there, by adding a new unit element.

Branch-Cycle-U: **Branching Cycle for Unit**

$$\{x : t \triangleq s\} \cup A;\ S;\ \{(\{y : t \triangleq s\}, Un)\} \cup L;\ B \Longrightarrow$$
$$\{y_1 : f(t, \epsilon_f) \triangleq f(\epsilon_f, s), y_2 : f(\epsilon_f, t) \triangleq f(s, \epsilon_f), y_3 : t \triangleq s\} \cup A;\ S;$$
$$\{(\{y : t \triangleq s\}, \{\epsilon_f\} \cup Un)\} \cup L;\ B\{x \mapsto y\} \cup \{y \mapsto y_1\} \cup \{y \mapsto y_2\},$$

where $\mathsf{U} \in Ax(f)$, $\epsilon_f \notin Un$, $head(t) \neq \epsilon_f$ or $head(s) \neq \epsilon_f$, $\mathsf{U} \notin Ax(head(t)) \cup Ax(head(s))$, and y_1 and y_2 are fresh variables.

We get the set of all rules for unital generalization $\mathcal{R}_{\mathsf{U}} := \mathcal{R}_{\mathsf{lin}} \cup \mathcal{R}_{\mathsf{one(f)}} \cup \{\mathsf{Branch\text{-}Cycle\text{-}U}\}$, and the procedure that is based on them is denoted by $\mathfrak{G}_{\mathsf{U}}$. It is formulated in Algorithm 5.

Algorithm 5 Procedure $\mathfrak{G}_{\mathsf{U}}$.

```
Require: A configuration $\mathbf{C} = A; S; L; B$
while $A \neq \emptyset$ do
    $\mathbf{a} \leftarrow x : t \triangleq s \in A$
    $(\mathbf{C}, \mathbf{a}) \leftarrow \mathsf{Cycle}(\mathbf{C}, \mathbf{a})$ (See Algorithm 3)
    if $\exists f \in \mathcal{A} : (\mathsf{U} \in Ax(f) \wedge (\{y : t \triangleq s\}, Un) \in L \wedge \epsilon_f \notin Un)$ then
        repeat
            Apply Branch-Cycle-U to $\mathbf{a}$ resulting in $\mathbf{C}' = \{\mathbf{a}_1, \mathbf{a}_2, x' : t \triangleq s\} \cup A; S; L'; B'$
            Apply Dec to $\mathbf{a}_1, \mathbf{a}_2$ resulting in $\mathbf{C}''$. Update $\mathbf{C} \leftarrow \mathbf{C}''$ and $\mathbf{a} \leftarrow x' : t \triangleq s$
            Exhaustively apply Sat-Cycle-U to $\mathbf{C}$ resulting in $\mathbf{C}^*$. Update $\mathbf{C} \leftarrow \mathbf{C}^*$
        until $\forall f \in \mathcal{A} : (\mathsf{U} \in Ax(f) \wedge (\{y : t \triangleq s\}, Un) \in L) \Rightarrow \epsilon_f \in Un)$
    end if
    $\mathbf{C} \leftarrow \mathsf{Step}(\mathbf{C}, \mathbf{a})$ (See Algorithm 1)
    Exhaustively apply Sat-Cycle-U to $\mathbf{C}$ resulting in $\mathbf{C}^*$. Update $\mathbf{C} \leftarrow \mathbf{C}^*$
end while
Exhaustively apply Merge to $\mathbf{C}$ resulting in $\mathbf{C}^*$. Update $\mathbf{C} \leftarrow \mathbf{C}^*$
return $\mathbf{C}$
```

Note that at each step in the procedures outlined in Algorithms 2, 4, and 5, there is only one rule applicable to the current configuration. Thus, each procedure produces a single tree grammar whose language is the computed generalizations of the initial AUP. Termination and soundness of $\mathfrak{G}_{\mathsf{U}}$ depends on termination and soundness of Branch-Cycle-U, which can be established similarly to Start-Cycle-U. Completeness of $\mathfrak{G}_{\mathsf{U}}$ needs further study.

▶ **Theorem 22.** *The algorithm $\mathfrak{G}_U$ is terminating and sound.*

We have seen in Section 3 that unital anti-unification with two unital symbols is nullary, based on the AUPs $\epsilon_f \triangleq \epsilon_g$. Such AUPs can be generated with the help of Branch-Cycle-U even from such trivial problems as, e.g., $a \triangleq a$.

7 Combined theories

In this section we consider the combination of unit element theories with other common equational theories such as A (Associativity), C (Commutativity), and I (Idempotency).

Observe that the anti-unification problems used to prove Theorem 9, i.e., $\epsilon_f \triangleq \epsilon_g$ and $\epsilon_g \triangleq \epsilon_f$, are still problematic when considering the combined theories CU, AU, ACU. For example, modulo CU, AU, and ACU, $f(x, g(x, y)) \not\preceq_u x$, for $u \in \{\mathsf{CU}, \mathsf{AU}, \mathsf{ACU}\}$, when $\mathsf{U} \in Ax(f)$ and $\mathsf{U} \in Ax(g)$. Thus, the argument outlined in Section 3 still applies to these cases. However, for UI we have $f(x, g(x, y)) \preceq_{\mathsf{UI}} x$, i.e., $f(x, g(x, y))\{y \mapsto x\} = f(x, g(x, x)) \preceq_{\mathsf{UI}} f(x, x) \preceq_{\mathsf{UI}} x$ where $\mathsf{U}, \mathsf{I} \in Ax(f)$ and $\mathsf{U}, \mathsf{I} \in Ax(g)$. Thus, our proof of nullarity for unital theories cannot be extended to UI. As it was shown in [8], a theory with a single idempotent function is infinitary if there is an AUP with a so called base set of generalizations of size at least two. It is not completely clear that a similar result will hold for UI and ACUI.

Concerning the special cases, since C, A, and AC are finitary [1], we expect that their linear variant and one-unital fragment remain finitary, despite the fact that the existing algorithms are not based on the tree grammar representation and would require reworking. This can be done in a straightforward manner similar to our handling of the U-decomposition rules we define above. When using the tree grammar formulation described in this paper or as described in [8], one either needs to describe how to join tree grammars as in [8], or write rules in such a way that all possibilities are exhausted by a single rule application. Notice the U-decomposition rules introduce all possible decompositions modulo U into the configuration. The existing rules for C, A, and AC can be adjusted to our framework in a similar way, i.e., it would require writing a rule which adds all decomposition paths simultaneously to the current configuration.

8 Discussion

In this work we showed that unital anti-unification is of type zero. We also distinguished two cases the problem is finitary: linear variant and one-unital fragment. We provided procedures for solving those special cases, and proved their termination, soundness, and completeness. Besides, we provide a terminating and sound general procedure for computing unrestricted unital generalizations. These procedures are based on tree grammar construction in a similar fashion as in earlier work on idempotent equational theories [8]. We also briefly discussed generalization type in combined theories such as CU, AU, ACU, ACUI, and UI.

We end the paper with the following list of open questions:

- Is the general procedure $\mathfrak{G}_{\mathsf{U}}$ complete for arbitrary unital theories?
- Modify the one-unital procedure $\mathfrak{G}_{\mathsf{U(f)}}$ so that it produces less verbose tree grammars.
- Can the rules outlined in [1] be joined with the rules from $\mathcal{R}_{\mathsf{one(f)}}$ to produce minimal complete procedures for restrictions of CU, AU, ACU.
- Are unrestricted ACUI and UI infinitary or nullary?
- Can the techniques used here and [8] be generalized to AU for any collapse theory?
- Are there non-trivial collapse theories with unitary or finitary AU type?

References

1 María Alpuente, Santiago Escobar, Javier Espert, and José Meseguer. A modular order-sorted equational generalization algorithm. *Inf. Comput.*, 235:98–136, 2014. doi:10.1016/j.ic.2014.01.006.

2 Franz Baader and Wayne Snyder. Unification theory. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–533. North-Holland, Amsterdam, 2001. doi:10.1016/B978-044450813-3/50010-2.

3 Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA):159:1–159:27, 2019. doi:10.1145/3360585.

4 Adam D. Barwell, Christopher Brown, and Kevin Hammond. Finding parallel functional pearls: Automatic parallel recursion scheme detection in Haskell functions via anti-unification. *Future Generation Comp. Syst.*, 79:669–686, 2018. doi:10.1016/j.future.2017.07.024.

5 Alexander Baumgartner. *Anti-Unification Algorithms: Design, Analysis, and Implementation.* PhD thesis, Johannes Kepler University Linz, 2015. Available from http://www.risc.jku.at/publications/download/risc_5180/phd-thesis.pdf.

6 Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. Nominal anti-unification. In Maribel Fernández, editor, *RTA 2015, June 29 to July 1, 2015, Warsaw, Poland*, volume 36 of *LIPIcs*, pages 57–73. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPIcs.RTA.2015.57.

7 Armin Biere. Normalisation, unification and generalisation in free monoids. Master's thesis, University of Karlsruhe, 1993. (in German).

8 David Cerna and Temur Kutsia. Idempotent anti-unification. *ACM Trans. Comput. Logic*, 21(2), November 2019. doi:10.1145/3359060.

9 David M. Cerna and Temur Kutsia. Higher-order pattern generalization modulo equational theories. *Mathematical Structures in Computer Science*, 2020. Accepted.

10 Stefan Kühner, Chris Mathis, Peter Raulefs, and Jörg H. Siekmann. Unification of idempotent functions. In Raj Reddy, editor, *Proceedings of the 5th International Joint Conference on Artificial Intelligence. Cambridge, MA, USA, August 22-25, 1977*, page 528. William Kaufmann, 1977. URL: http://ijcai.org/Proceedings/77-1/Papers/092.pdf.

11 Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, Chetan Bansal, Chandra Maddila, B. Ashok, Sumit Asthana, Christian Bird, and Aditya Kumar. Rex: Preventing bugs and misconfiguration in large services using correlated change analysis. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 435–448, Santa Clara, CA, February 2020. USENIX Association. URL: https://www.usenix.org/conference/nsdi20/presentation/mehta.

12 Gordon D. Plotkin. A note on inductive generalization. *Machine Intell.*, 5(1):153–163, 1970.

13 John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intel.*, 5(1):135–151, 1970.

14 Reudismam Rolim, Gustavo Soares, Rohit Gheyi, and Loris D'Antoni. Learning quick fixes from code repositories. *CoRR*, abs/1803.03806, 2018. arXiv:1803.03806.

15 Ute Schmid. *Inductive Synthesis of Functional Programs, Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*, volume 2654 of *Lecture Notes in Computer Science*. Springer, 2003.

16 Jörg H. Siekmann. Unification theory. *J. Symb. Comput.*, 7(3/4):207–274, 1989. doi:10.1016/S0747-7171(89)80012-4.

17 Erik Tidén and Stefan Arnborg. Unification problems with one-sided distributivity. *J. Symb. Comput.*, 3(1/2):183–202, 1987. doi:10.1016/S0747-7171(87)80026-3.

A Proof of Theorem 13

Proof. We assume that t_1, t_2, and s are in U-normal form. We prove the theorem by induction on $dep(t_1) + dep(t_2)$ which we denote by n.

Case 1: $n = 2$, i.e., t_1 and t_2 are constants.

a) First, assume that $dep(s) = 1$. If $t_1 = t_2$, then $s = t_1 = t_2$ and s is computed by the derivation $\{x : t_1 \triangleq t_2\}; \emptyset; \emptyset; \{x_{\text{root}} \mapsto x\} \Longrightarrow_{\mathsf{Dec}} \emptyset; \emptyset; \emptyset; \{x_{\text{root}} \mapsto t_1\}$. If $t_1 \neq t_2$, then s must be a variable, computed by the derivation $\{x : t_1 \triangleq t_2\}; \emptyset; \emptyset; \{x_{\text{root}} \mapsto x\} \Longrightarrow_{\mathsf{Sol}} \emptyset; \{x : t_1 \triangleq t_2\}; \emptyset; \{x_{\text{root}} \mapsto x\}$. Note that, in both cases the resulting tree grammars are trivial, both have a language of size 1. Thus, we will refer to the members of these languages directly rather than evoking the tree grammar itself.

b) Now assume as the induction hypothesis that for every generalization s of t_1 and t_2 of depth at most k, either $s \preceq t_1$ and $t_1 = t_2$, or $s \preceq x$ and $t_1 \neq t_2$. We show that this holds for a generalization s' of depth $k + 1$. Let $head(s') = f$. Our assumptions imply that $\mathsf{U} \in \mathsf{Ax}(f)$ because both t_1 and t_2 are of depth 1. Thus, $s' = f(s_1, s_2)$.

By the definition of a generalization, there must exists two substitutions σ_1 and σ_2 such that $s'\sigma_1 = t_1$ and $s'\sigma_2 = t_2$. If $s_1\sigma_1 = s_1\sigma_2 = \epsilon_f$ (resp. if $s_2\sigma_1 = s_2\sigma_2 = \epsilon_f$), then s_2 (resp., s_1) is, by the induction hypothesis, more general than t_1 when $t_1 = t_2$, or more general than x when $t_1 \neq t_2$. This implies, by the linearity assumption that there exists a substitution ϑ such that $s_2\vartheta = s_2$ and $s_1\vartheta = \epsilon_f$. Thus, $s'\vartheta = s_2$, i.e. $s' \prec s_2$.

However, if $s_2\sigma_1 = \epsilon_f$ and $s_1\sigma_2 = \epsilon_f$, or vice versa, then additional observations are required. We assume without loss of generality the former case.

If $t_1 = t_2$ then both s_1 and s_2 are generalizations of $t_1 \triangleq t_2$ and by the induction hypothesis $s_1 \preceq t_1$ and $s_2 \preceq t_1$. If $t_1 \neq t_2$ then we need to make a distinction:

b1. If neither t_1 nor t_2 are units of function constants f_{t_1} and f_{t_2}, respectively, which may appear in s, then there exists a variable y occurring in s_1 such that $y\sigma_1 = t_1$ and a variable y' occurring in s_2 such that $y'\sigma_2 = t_2$. However, by the linearity of S, this implies that there exist two substitutions σ_1' and σ_2' which coincide everywhere with σ_1 and σ_2 except on y and y' respectively. That is, $y\sigma_1' = t_2$ and $y'\sigma_2' = t_1$. This implies that both s_1 and s_2 are generalizations of $t_1 \triangleq t_2$ which have depth $\leq k+1$. Thus, $s_1 \preceq x$ and $s_2 \preceq x$.

b2. If either t_1 or t_2 is a unit of the function constants f_{t_1} and f_{t_2}, respectively, which may appear in s, then additional observations are necessary. If neither t_1 or t_2 occurs in s then we have the same situation as in case b1. Otherwise, if f_{t_1} occurs in s_1 (respectively f_{t_2} in s_2) then it must occur as the head symbol of a term with t_1 as a subterm because $s_1\sigma_2 = \epsilon_{f_{t_1}}$. This implies that there must be a variable y in s_1 which σ_1 maps to t_1. Similar can be said concerning s_2, t_2, and σ_2. We can construct a new substitution which coincides with σ_1 (respectively, with σ_2) everywhere but on the variable y (resp. y') which it maps to t_2 (resp. to t_1). This means that s_1 and s_2 are generalizations of $t_1 \triangleq t_2$ and by the induction hypothesis $s_1 \preceq x$ $s_2 \preceq x$. This completes the case 1.

Case 2: $n > 2$.

a) Let us assume that $t_1 = f(w_1, \ldots, w_m)$ and $t_2 = f(r_1, \ldots, r_m)$, such that $\mathsf{U} \notin \mathsf{Ax}(f)$. Then by applying the Dec rule to the AUP $x : t_1 \triangleq t_2$ we get m AUPs $x_1 : w_1 \triangleq r_1, \ldots, x_m : w_1 \triangleq r_1$ each of which has a depth sum $\leq n-1$. Thus, by the induction hypothesis, for each generalization s' generalizing $X_i : w_i \triangleq r_i$ there exists a generalization $s_i^* \in \mathcal{L}(\mathcal{G}(B_i))$, where B_i is the final set of bindings computed using $\mathfrak{G}_{\mathsf{U\text{-}lin}}$, such that, $s' \preceq s_i^*$. Now let S_i^* be the set of all such generalizations computed using $\mathfrak{G}_{\mathsf{U\text{-}lin}}$. We may now define the set of generalizations S^* as $S^* = \{f(s_1^*, \ldots, s_m^*) \mid s_i^* \in S_i^* \text{ for all } 1 \leq i \leq m\}$. Note that each term in S^* is a generalization of $X : t_1 \triangleq t_2$ computed using $\mathfrak{G}_{\mathsf{U\text{-}lin}}$ in is contained in $\mathcal{L}(\mathcal{G}(B))$, where B is the final set of bindings computed using $\mathfrak{G}_{\mathsf{U\text{-}lin}}$. Thus, any generalization s' of $X : t_1 \triangleq t_2$ such that $\mathit{head}(s') = f$ is more general than some generalization of S^*. Thus we need only to consider generalization s' such that $\mathit{head}(s') \neq f$. This implies that $\mathsf{U} \in \mathsf{Ax}(\mathit{head}(s'))$.

If s' does not contain f, then $s' \preceq X$. Thus let us assume that $s' = g(s_1', s_2')$ where $\mathsf{U} \in \mathsf{Ax}(g)$ and without loss of generality $\mathit{head}(s_1') = f$. This implies that $s_2' \preceq \epsilon_g$ (note that s' is linear) and thus $s_1' \preceq s'$. This reduction can be performed inductively thus showing that for any generalization s' with $\mathit{head}(s') \neq f$ there exists $s'' \in S^*$ such that $s' \preceq s''$.

b) Let us assume that $t_1 = f(w_1, w_2)$ and $t_2 = f(r_1, r_2)$, such that $\mathsf{U} \in \mathsf{Ax}(f)$. Then we can proceed in a similar fashion as in case b) by constructing S^*. Thus, any generalization s' of $X : t_1 \triangleq t_2$ such that $\mathit{head}(s') = f$ and $s' = f(d_1, d_2)$, where d_1 is a generalization of $w_1 \triangleq r_1$, d_2 a generalization of $w_2 \triangleq r_2$, is more general than some generalization of S^*. When $\mathsf{U} \in \mathsf{Ax}(\mathit{head}(s'))$ and some generalization s'' is a subterm of s' such that there exists $s^* \in S^*$ with $s'' \preceq s^*$, a similar approach can be taken as in the second half of case 2a).

c) Let us assume that $t_1 = f(w_1, \ldots, w_m)$ and $t_2 = g(r_1, \ldots, r_k)$, where either $\mathsf{U} \in \mathsf{Ax}(f)$ or $\mathsf{U} \in \mathsf{Ax}(g)$, or both. By an application of $\mathsf{Exp\text{-}U\text{-}Both}$, $\mathsf{Exp\text{-}U\text{-}L}$, or $\mathsf{Exp\text{-}U\text{-}R}$ this case can be reduced to two (possibly four) instances of case 2b). ◀

B Example used for the proof of nullarity

Below is the tree grammar computed from the final configuration of $\mathfrak{G}_{\mathsf{U}}$ applied to $\epsilon_g \triangleq \epsilon_f$. Computation of the final binding set required the application of 86 rules to the initial configuration.

$$\mathcal{G} = \left(\{\mathbf{x}\}, \left\{ \begin{array}{c} \mathbf{x}, \mathbf{x}_1, \\ \mathbf{x}_5, \mathbf{x}_{11} \\ \mathbf{x}_{18}, \mathbf{x}_{29} \end{array} \right\}, \left\{ \begin{array}{c} f, g, \\ \epsilon_f, \epsilon_g, \\ x_8, x_{36} \end{array} \right\}, \left\{ \begin{array}{ll} \mathbf{x} \mapsto g(\mathbf{x}, \mathbf{x}_5), & \mathbf{x} \mapsto g(\mathbf{x}_5, \mathbf{x}) \\ \mathbf{x} \mapsto \mathbf{x}_1, & \mathbf{x} \mapsto x_8 \\ \mathbf{x}_1 \mapsto f(\mathbf{x}, \mathbf{x}_{11}), & \mathbf{x}_1 \mapsto f(\mathbf{x}_{11}, \mathbf{x}) \\ \mathbf{x}_5 \mapsto f(\mathbf{x}, \mathbf{x}_{18}), & \mathbf{x}_5 \mapsto f(\mathbf{x}_{18}, \mathbf{x}) \\ \mathbf{x}_5 \mapsto \epsilon_g, & \mathbf{x}_{11} \mapsto g(\mathbf{x}_{18}, \mathbf{x}) \\ \mathbf{x}_{11} \mapsto g(\mathbf{x}, \mathbf{x}_{18}), & \mathbf{x}_{11} \mapsto \epsilon_f \\ \mathbf{x}_{18} \mapsto \mathbf{x}_{29}, & \mathbf{x}_{18} \mapsto x_{36} \\ \mathbf{x}_{18} \mapsto g(\mathbf{x}_5, \mathbf{x}_{18}), & \mathbf{x}_{18} \mapsto g(\mathbf{x}_{18}, \mathbf{x}_5) \\ \mathbf{x}_{29} \mapsto f(\mathbf{x}_{18}, \mathbf{x}_{11}), & \mathbf{x}_{29} \mapsto g(\mathbf{x}_{11}, \mathbf{x}_{18}) \end{array} \right\} \right).$$

If we clean the grammar by removing redundant bindings we get the tree grammar $\mathcal{G}'$:

$$\mathcal{G}' = \left(\{\mathbf{x}\}, \left\{ \begin{array}{c} \mathbf{x}, \\ \mathbf{y} \end{array} \right\}, \left\{ \begin{array}{c} f, g, \\ \epsilon_f, \epsilon_g, \\ y, z \end{array} \right\}, \left\{ \begin{array}{ll} \mathbf{x} \mapsto g(\mathbf{x}, f(\mathbf{x}, \mathbf{y})), & \mathbf{x} \mapsto f(\mathbf{x}, g(\mathbf{x}, \mathbf{y})) \\ \mathbf{x} \mapsto f(g(\mathbf{y}, \mathbf{x}), \mathbf{x}), & \mathbf{x} \mapsto x \\ \mathbf{x} \mapsto g(\mathbf{x}, f(\mathbf{y}, \mathbf{x})), & \mathbf{x} \mapsto f(\mathbf{x}, g(\mathbf{y}, \mathbf{x})) \\ \mathbf{x} \mapsto f(g(\mathbf{x}, \mathbf{y}), \mathbf{x}), & \mathbf{x} \mapsto g(f(\mathbf{y}, \mathbf{x}), \mathbf{x}) \\ \mathbf{x} \mapsto g(f(\mathbf{x}, \mathbf{y}), \mathbf{x}), & \mathbf{y} \mapsto f(g(\mathbf{y}, \mathbf{x}), \mathbf{y}) \\ \mathbf{y} \mapsto g(\mathbf{y}, f(\mathbf{y}, \mathbf{x})), & \mathbf{y} \mapsto f(\mathbf{y}, g(\mathbf{y}, \mathbf{x})) \\ \mathbf{y} \mapsto g(f(\mathbf{y}, \mathbf{x}), \mathbf{y}), & \mathbf{y} \mapsto y \\ \mathbf{y} \mapsto f(\mathbf{y}, g(\mathbf{x}, \mathbf{y})), & \mathbf{y} \mapsto g(\mathbf{y}, f(\mathbf{x}, \mathbf{y})) \\ \mathbf{y} \mapsto f(g(\mathbf{x}, \mathbf{y}), \mathbf{y}), & \mathbf{y} \mapsto g(f(\mathbf{x}, \mathbf{y}), \mathbf{y}) \end{array} \right\} \right).$$

Some of the generalizations contained in the language of this grammar are x, $f(x, g(x, y))$, $f(x, g(y, x))$, $f(g(y, x), x)$, $f(g(y, x), f(x, g(x, y)))$, $f(g(y, f(x, g(x, y))), f(x, g(x, y)))$, and $f(f(x, g(x, y)), g(f(x, g(x, y)), y))$. Observe that some of these generalizations are comparable and form a subsequence of an infinite chain of less generality.

C Grammar generated for Example 16

Below is the tree grammar computed from the final configuration of $\mathfrak{G}_{\mathsf{U(f)}}$ applied to $g(f(a, c), a) \triangleq g(c, b)$. Note that g is non-unital and no unit elements show up in the initial AUP. Computation of the final binding set required the application of 217 rules to the initial configuration. We only provide the cleaned version of the tree grammar. Note that the language of the resulting tree grammar is finite.

$$\mathcal{G} = \left(\{\mathbf{x}\}, \{\ \mathbf{x}\ \}, \left\{ \begin{array}{c} f, g, \epsilon_f, a, b, \\ c, y, z, y', z' \end{array} \right\}, B\right),$$

where B is the set

$$\left\{ \begin{array}{lll}
\mathbf{x} \mapsto g(f(f(y, z), y'), z') & \mathbf{x} \mapsto g(f(y, z), f(y', z')) & \mathbf{x} \mapsto g(f(f(z, y'), y), f(z, z')) \\
\mathbf{x} \mapsto g(f(f(z, y), y'), f(z, z')) & \mathbf{x} \mapsto g(f(y, y'), z') & \mathbf{x} \mapsto g(f(f(y, z), y'), f(z', z)) \\
\mathbf{x} \mapsto g(f(y, f(z, y')), z') & \mathbf{x} \mapsto g(f(z, f(y, y')), z') & \mathbf{x} \mapsto g(f(z, f(y', y)), z') \\
\mathbf{x} \mapsto g(f(f(z, y), y'), f(z', z)) & \mathbf{x} \mapsto g(f(f(z, y'), y), f(z', z)) & \mathbf{x} \mapsto f(y, z) \\
\boxed{\mathbf{x} \mapsto g(f(z, c), f(y, z))} & \mathbf{x} \mapsto g(f(y, y'), f(z', z)) & \mathbf{x} \mapsto g(f(z, f(y, y')), f(z, z')) \\
\mathbf{x} \mapsto g(f(y, f(z, y')), f(z, z')) & \mathbf{x} \mapsto g(f(z, f(y', y)), f(z, z')) & \mathbf{x} \mapsto g(f(z, c), z') \\
\mathbf{x} \mapsto g(f(f(z, y'), y), z') & \mathbf{x} \mapsto g(f(z, f(y, y')), f(z', z)) & \mathbf{x} \mapsto g(f(y, f(z, y')), f(z', z)) \\
\mathbf{x} \mapsto g(f(f(y, z), y'), f(z, z')) & \boxed{\mathbf{x} \mapsto g(f(z, c), f(z, y))} & \mathbf{x} \mapsto g(f(z, f(y', y)), f(z', z)) \\
\mathbf{x} \mapsto f(y, z) & \mathbf{x} \mapsto g(f(f(z, y), y'), z') &
\end{array} \right\}.$$

Observe that of the 26 terms contained in $\mathcal{L}(\mathcal{G})$, there are only two incomparable terms, $g(f(z, c), f(y, z))$ and $g(f(z, c), f(z, y))$.

Symbolic Execution Game Semantics

Yu-Yang Lin
Queen Mary University of London, UK

Nikos Tzevelekos
Queen Mary University of London, UK

Abstract

We present a framework for symbolically executing and model checking higher-order programs with external (open) methods. We focus on the client-library paradigm and in particular we aim to check libraries with respect to any definable client. We combine traditional symbolic execution techniques with operational game semantics to build a symbolic execution semantics that captures arbitrary external behaviour. We prove the symbolic semantics to be sound and complete. This yields a bounded technique by imposing bounds on the depth of recursion and callbacks. We provide an implementation of our technique in the $\mathbb{K}$ framework and showcase its performance on a custom benchmark based on higher-order coding errors such as reentrancy bugs.

2012 ACM Subject Classification Theory of computation → Semantics and reasoning

Keywords and phrases game semantics, symbolic execution, higher-order open programs

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.27

Supplementary Material The tool and its benchmarks can be found at: `https://github.com/LaifsV1/HOLiK`.

Funding Research funded by EPSRC (EP/P004172/1).

Acknowledgements We would like to thank members of the K framework for their consistent support with K, and anonymous reviewers for their insightful reviews.

1 Introduction

Two important challenges in program verification are state-space explosion and the environment problem. The former refers to the need to investigate infeasibly many states, while the latter concerns cases where the code depends on an environment that is not available for analysis. State-space explosion has been approached with a range of techniques, which have led to verification tools being nowadays routinely used on industrial-scale code (e.g. [10, 5, 7]). The environment problem, however, remains largely unanswered: verification techniques often require the whole code to be present for the analysis and, in particular, cannot analyse components like libraries where parts of the code are missing (e.g. the client using the library). This problem is particularly acute in higher-order programs, where the interaction between a program and its environment can be intricate and e.g. involve callbacks or reentrant calls. In this paper we address this latter problem by combining *game semantics*, a semantics theory for higher-order programs, with *symbolic execution*, a technique that uses *symbolic values* to explore multiple execution paths of a program.

To showcase the importance and challenges of the environment problem, following is a simple example of a library written in a sugared version of HOLi, the vehicle language of this paper. The example is a simplified implementation of "The DAO" smart contract, a failed decentralised autonomous organisation on the Ethereum blockchain platform [12]. As with libraries, the challenge in analysing smart contracts is that the client code is not available. We must thus generate all possible contexts in which the contract can be called. In this case, the error is caused by a reentrant call from the `send()` method, which is provided by the

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 27; pp. 27:1–27:24
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

environment. When this method is called, the environment takes control and is allowed to call any method in the library. If a client were to call `withdraw()` within its `send()` method, the recursive call would drain all the funds available, which is simulated in this example by a negative balance. This happens because the method is manipulating a global state, and is updating it after the external call. We can see that an analysis capturing this error would need to be able to predict an intricate environment behaviour. Moreover, such an analysis should ideally only predict realisable environment behaviours.

```
1 import send:(int → unit)
2 int balance := 100;
3
4 public withdraw (m:int) :(unit) =
5   if (not (!balance < m)) then
6        send(m);
7        balance := !balance - m;
8        assert(not(!balance < 0))
9   else ();
```

Symbolic execution [33, 13, 19] explores all paths of a program using symbolic values instead of concrete input values. Each symbolic path holds a path condition (a SAT formula) that is satisfiable if and only if the path can be concretely executed. While the resulting analysis is unbounded in general, by restricting our focus to bounded paths we can soundly catch errors, or affirm the absence thereof up to the used bound. Game semantics [2, 14], on the other hand, models higher-order program phrases in isolation as 2-player games: sequences of computational *moves* (method calls and returns) between the program and its hypothetical environment. The power of the technique lies in its use of combinatorial conditions to precisely allow those game plays that can be realised by including the program in an actual environment. Moreover, the theory can be formulated operationally in terms of a trace semantics for open terms [18, 21, 16] which, in turn, lends itself to a symbolic representation. The latter yields a symbolic execution technique that is *sound and complete* in the following sense: given an open program, its symbolic traces match its concrete traces, which match its realisable traces in some environment.

Returning to the DAO example, we can model the ensuing interaction as a sequence of moves, alternating between the environment and the library. Any finite sequence of moves (that leads to an assertion violation) is a trace defining a counterexample. Running the example in HOLiK, our implementation of the symbolic semantics in the $\mathbb{K}$ Framework [32], the following minimal symbolic trace is automatically found:

$$\begin{aligned}&\mathit{call}\langle \mathit{withdraw}, x_1\rangle \cdot \mathit{call}\langle \mathit{send}, x_1\rangle \cdot \mathit{call}\langle \mathit{withdraw}, x_2\rangle\\ &\qquad \cdot \mathit{call}\langle \mathit{send}, x_2\rangle \cdot \mathit{ret}\langle \mathit{send}, ()\rangle \cdot \mathit{ret}\langle \mathit{withdraw}, ()\rangle \cdot \mathit{ret}\langle \mathit{send}, ()\rangle\end{aligned}$$

where x_1 is the original call parameter, and x_2 is the parameter for the reentrant call, satisfiable with values $x_1 = 100$ and $x_2 = 1$. A fix would be to swap line 6 and 7, to update internal state before passing control.

In Appendix A we look at a few more examples of libraries that exhibit errors due to high-order behaviours. We provide three examples: a file lock example, a double deallocation example, and an unsafe implementation of flat-combining.

Overall, this paper contributes a novel symbolic execution technique based on game semantics to precisely model the behaviour of higher-order stateful programs. Specifically:

- We present a symbolic trace semantics for higher-order libraries that captures the behaviour of an unknown environment, and prove it sound and complete: i.e. it produces no spurious error traces, and is able to produce the complete execution tree of any library.

$$\begin{array}{rrl@{\qquad}rrl}
\textit{Libraries} & L ::= & B \mid \texttt{abstract}\ m; L & \textit{Terms} & M ::= & m \mid i \mid () \mid x \mid \lambda x.M \mid r := M \mid !r \\
\textit{Blocks} & B ::= & \varepsilon \mid \texttt{public}\ m = \lambda x.M; B & & & \mid M \oplus M \mid \langle M, M\rangle \mid \pi_1 M \mid \pi_2 M \\
& & \mid m = \lambda x.M; B \mid \texttt{global}\ r := i; B & & & \mid MM \mid \texttt{if}\ M\ \texttt{then}\ M\ \texttt{else}\ M \\
& & \mid \texttt{global}\ r := \lambda x.M; B & & & \mid \texttt{letrec}\ x = \lambda x.M\ \texttt{in}\ M \\
\textit{Clients} & C\ ::= & L; \texttt{main} = M & & & \mid \texttt{let}\ x = M\ \texttt{in}\ M \mid \texttt{assert}(M)
\end{array}$$

$$\frac{}{():\texttt{unit}} \qquad \frac{}{i:\texttt{int}} \qquad \frac{x \in \texttt{Vars}_\theta}{x:\theta} \qquad \frac{m \in \texttt{Meths}_{\theta,\theta'}}{m:\theta\to\theta'} \qquad \frac{M, M':\texttt{int}}{M \oplus M':\texttt{int}} \qquad \frac{M:\texttt{int} \qquad M_1, M_0:\theta}{\texttt{if}\ M\ \texttt{then}\ M_1\ \texttt{else}\ M_0:\theta}$$

$$\frac{M:\theta_1 \quad M':\theta_2}{\langle M, M'\rangle:\theta_1\times\theta_2} \qquad \frac{\langle M, M'\rangle:\theta_1\times\theta_2}{\pi_i\langle M, M'\rangle:\theta_i} \qquad \frac{r\in\texttt{Refs}_\theta}{!r:\theta} \qquad \frac{r\in\texttt{Refs}_\theta \quad M:\theta}{r := M:\texttt{unit}} \qquad \frac{M':\theta\to\theta' \quad M:\theta}{M'\,M:\theta'}$$

$$\frac{M:\theta' \quad x:\theta}{\lambda x.M:\theta\to\theta'} \qquad \frac{x, M:\theta \quad M':\theta'}{\texttt{let}\ x = M\ \texttt{in}\ M':\theta'} \qquad \frac{x, \lambda y.M:\theta\to\theta'' \quad M':\theta'}{\texttt{letrec}\ x = \lambda y.M\ \texttt{in}\ M':\theta'} \qquad \frac{M:\texttt{int}}{\texttt{assert}(M):\texttt{unit}}$$

Figure 1 Syntax and typing rules of HOLi.

- By bounding the depth of nested calls and the *insistence* of the environment in calling library methods, we derive a sound and bounded-complete technique to check higher-order libraries for errors.
- We implement the latter in the $\mathbb{K}$ semantical framework [32] to produce a sound and bounded-complete tool for higher-order libraries as a proof of concept. We test our implementation with benchmarks adapted from the literature.

Some material has been delegated to an Appendix.

2 A Language for Higher-Order Libraries: HOLi

We introduce HOLi, a language for higher-order libraries which define methods to be used by an external client, and in turn require external methods (provided by the client). We give in HOLi an operational semantics for terms that integrates a counter for the depth of nested calls that a program phrase can make. We then extend this counting semantics to open terms by means of a trace semantics. We show that the trace semantics of libraries is sound and complete for reachability of errors under any external client.

2.1 Syntax and operational semantics

A library in HOLi is a collection of typed higher-order methods. A client is simply a library with a main body. Types are given by the grammar:

$$\theta \ ::= \ \texttt{unit} \mid \texttt{int} \mid \theta\times\theta \mid \theta\to\theta$$

We use countably infinite sets Meths, Refs and Vars for method, global reference and variable names, ranged over by m, r and x respectively, and variants thereof; while i is for ranging over the integers. We use $\oplus$ to range over a set of binary integer operations, which we leave unspecified. Each set of names is typed, that is, it can be expressed as a disjoint union as follows: $\texttt{Meths} = \biguplus_{\theta,\theta'}\texttt{Meths}_{\theta,\theta'}$, $\texttt{Refs} = \biguplus_{\theta\neq\theta_1\times\theta_2}\texttt{Refs}_\theta$, $\texttt{Vars} = \biguplus_\theta \texttt{Vars}_\theta$.

The full syntax and typing rules are given in Figure 1. Thus, a library consists of abstract method declarations, followed by blocks of public and private method and reference definitions. A method is considered private unless it is declared `public`. Each public/private method and reference is defined once. Abstract methods are not given definitions: these methods are external to the library. Public, private and abstract methods are all disjoint.

$$\begin{array}{ll}
(E[\texttt{let}\ x = v\ \texttt{in}\ M], R, S, k) \to (E[M\{v/x\}], R, S, k) & (E[\pi_j\langle v_1, v_2\rangle], R, S, k) \to (E[v_j], R, S, k) \\
(E[r := v], R, S, k) \to (E[()], R, S[r \mapsto v], k) & (E[!r], R, S, k) \to (E[S(r)], R, S, k) \\
(E[\texttt{if}\ i\ \texttt{then}\ M_1\ \texttt{else}\ M_0], R, S, k) \to (E[M_j], R, S, k)\ \ (1) & (E[i_1 \oplus i_2], R, S, k) \to (E[i], R, S, k)\ \ (2) \\
(E[\lambda x.M], R, S, k) \to (E[m], R \uplus \{m \mapsto \lambda x.M\}, S, k) & (E[\texttt{assert}(i)], R, S, k) \to (E[()], R, S, k)\ \ (3) \\
(E[mv], R, S, k) \to (E[\llparenthesis M\{v/x\}\rrparenthesis], R, S, k+1)\ \ (4) & (E[\llparenthesis v\rrparenthesis], R, S, k+1) \to (E[v], R, S, k) \\
\multicolumn{2}{l}{(E[\texttt{letrec}\ f = \lambda x.M\ \texttt{in}\ M'], R, S, k) \to (E[M'\{m/f\}], R \uplus \{m \mapsto \lambda x.M\{m/f\}\}, S, k)} \\
\multicolumn{2}{l}{\text{Conditions: } (1): j = 1 \text{ iff } i \neq 0, \quad (2): i = i_1 \oplus i_2, \quad (3): i \neq 0, \quad (4): R(m) = \lambda x.M.}
\end{array}$$

$$\begin{array}{ll}
\textit{Values}\ \ v\ ::=\ m \mid i \mid () \mid \langle v, v\rangle & \textit{Terms (extended)}\ \ M ::=\ \cdots \mid \llparenthesis M \rrparenthesis \\
\multicolumn{2}{l}{\textit{Eval. Contexts}\ \ E ::=\ \bullet \mid \texttt{assert}(E) \mid r := E \mid E \oplus M \mid v \oplus E \mid \langle E, M\rangle \mid \langle v, E\rangle \mid \pi_j E} \\
\multicolumn{2}{l}{\qquad\qquad EM \mid mE \mid \texttt{let}\ x = E\ \texttt{in}\ M \mid \texttt{if}\ E\ \texttt{then}\ M\ \texttt{else}\ M \mid \llparenthesis E \rrparenthesis}
\end{array}$$

$$\begin{aligned}
(\texttt{abstract}\ m; L, R, S, \mathcal{P}, \mathcal{A}) &\xrightarrow{bld} (L, R, S, \mathcal{P}, \mathcal{A} \uplus \{m\}) \\
(\texttt{public}\ m = \lambda x.M; B, R, S, \mathcal{P}, \mathcal{A}) &\xrightarrow{bld} (B, R \uplus \{m \mapsto \lambda x.M\}, S, \mathcal{P} \uplus \{m\}, \mathcal{A}) \\
(m = \lambda x.M; B, R, S, \mathcal{P}, \mathcal{A}) &\xrightarrow{bld} (B, R \uplus \{m \mapsto \lambda x.M\}, S, \mathcal{P}, \mathcal{A}) \\
(\texttt{global}\ r := i; B, R, S, \mathcal{P}, \mathcal{A}) &\xrightarrow{bld} (B, R, S \uplus \{r \mapsto i\}, \mathcal{P}, \mathcal{A}) \\
(\texttt{global}\ r := \lambda x.M; B, R, S, \mathcal{P}, \mathcal{A}) &\xrightarrow{bld} (B, R \uplus \{m \mapsto \lambda x.M\}, S \uplus \{r \mapsto m\}, \mathcal{P}, \mathcal{A})
\end{aligned}$$

Figure 2 Operational semantics (top); values and evaluation contexts (mid); library build (bottom).

Libraries are well typed if all their method and reference definitions are well typed (e.g. $\texttt{public}\ m = \lambda x.M$ is well typed if $m : \theta$ and $\lambda x.M : \theta$ are both valid for the same type θ) and only mention methods and references that are defined or abstract. A client $L; \texttt{main} = M$ is well typed if $M : \texttt{unit}$ is valid and $L; m = \lambda x.M$ is well typed for some fresh x, m. A library/client is *open* if it contains abstract methods. This is different to open/closed terms: we call a term *open* if it contains free variables.

▶ Remark 1. By typing variable, reference and method names, we do not need to provide a context in typing judgements. Note that the references we use are of non-product type and, more importantly, **global** to the library: a term can use references but not create them locally or pass them as arguments (we discuss how to include such references in Appendix C).

▶ **Example 2.** The DAO-attack example from the Introduction can be written in HOLi as:

$$\begin{aligned}
&\texttt{abstract}\ \textit{send};\ \ \texttt{global}\ \textit{bal} := 100; \\
&\texttt{public}\ \textit{wdraw} = \\
&\qquad \lambda x.\ \texttt{if}\ !\textit{bal} \geq x\ \texttt{then}\ (\textit{send}(x); \textit{bal} := !\textit{bal} - x; \texttt{assert}(!\textit{bal} \geq 0))\ \texttt{else}\ ()
\end{aligned}$$

where $\textit{send}, \textit{wdraw} \in \texttt{Meths}_{\texttt{int},\texttt{unit}}$, $\textit{bal} \in \texttt{Refs}_{\texttt{int}}$, and $M; M'$ stands for $\texttt{let}\ \_ = M\ \texttt{in}\ M'$.

A library contains public methods that can be called by a client. On the other hand, a client contains a main body that can be executed. These two scenarios constitute the operational semantics of HOLi. Both are based on evaluating (closed) terms, which we define next. Term evaluation requires: the closed term being evaluated; method definitions, provided by a method repository; reference values, provided by a store; and a call-depth counter (a natural number). Since method application is the only source of infinite behaviour in HOLi, bounding the depth of nested calls is enough to guarantee termination in program analysis. Hence we provide a mechanism to keep track of call depth.

The operational semantics is given in Figure 2. The evaluation of terms (top part) involves configurations of the form (M, R, S, k), where:

- M is a closed term which may contain *evaluation boxes*, i.e. points inside a term where a method call has been made and has not yet returned, and is taken from the syntax extending the one of Figure 1 with the rule: $M ::= \cdots \mid (\!|M|\!)$
- R is a *method repository*, i.e. a partial map from method names to their bodies
- S is a *store*, i.e. a partial map from reference names to their stored values
- k is a *counter*, i.e. a natural number.

Most of the rules are standard, but it is worth noting that lambdas are not values themselves but, rather, evaluate to method names that are freshly stored in the repository. Moreover, evaluation boxes interplay with the counter k in the semantics: they mark places where the depth has increased because of a nested call. The penultimate line of rules in the operational semantics keeps track of call depth, and illustrates the utility of evaluation boxes: making a call increases the counter and leaves behind an evaluation box; returning form the call removes the box and decreases the counter again.

A library L *builds* into a configuration of the form $(\varepsilon, R, S, \mathcal{P}, \mathcal{A})$, which includes its public methods according to the rules in Figure 2 (bottom). More precisely, R and S are as above, while $\mathcal{P}, \mathcal{A} \subseteq \texttt{Meths}$ are (disjoint) sets of *public* and *abstract* method names. We say that (a well typed) L builds to $(\varepsilon, R, S, \mathcal{P}, \mathcal{A})$ if $(L, \emptyset, \emptyset, \emptyset, \emptyset) \xrightarrow{bld}{}^{*} (\varepsilon, R, S, \mathcal{P}, \mathcal{A})$. If L builds to $(\varepsilon, R, S, \mathcal{P}, \mathcal{A})$ then the client L; $\texttt{main} = M$ builds to $(M, R, S, \mathcal{P}, \mathcal{A})$. Moreover, we can link libraries to clients and evaluate them, as in the following definition.

▶ **Definition 3.**

1. *Library L and client C are* compiatible *if L builds to some $(\varepsilon, R, S, \mathcal{P}, \mathcal{A})$ and C builds to some $(M, R', S', \mathcal{P}', \mathcal{A}')$ such that: $\mathcal{P} \supseteq \mathcal{A}'$ and $\mathcal{A} \supseteq \mathcal{P}'$ (*complementation*); $\mathrm{dom}(S) \cap \mathrm{dom}(S') = \emptyset$ (*disjoint state*); and $\mathrm{dom}(R) \cap \mathrm{dom}(R') = \emptyset$ (*method ownership*).*
2. *For a library L, we let $\hat{L}$ be L with all its abstract method declarations and* `public` *keywords removed; and similarly for $\hat{C}$. Given compatible library L and client C, we let their* composition *be the client: L;$C = \hat{L}; \hat{C}$.*
3. *Given compatible L, C, the semantics of L;C is:*

$$[\![L;C]\!] = \{\rho \mid L;C \textit{ builds to } (M, R, S, \emptyset, \emptyset) \wedge (M, R, S, 0) \to^* \rho\}$$

We say that $[\![L;C]\!]$ fails *if it contains some $(E[\texttt{assert}(0)], \cdots)$.*

▶ **Example 4.** To illustrate how libraries and clients are used, consider the DAO example again as a library $L_{\texttt{DAO}}$. We can define a client $C_{\texttt{atk}}$:

$\texttt{abstract}\ wdraw;\ \texttt{global}\ wlet := 0;$

$\texttt{public}\ send = \lambda x.wlet := !wlet + x;\ \texttt{if}\ !wlet < 100\ \texttt{then}\ wdraw(x)\ \texttt{else}\ ();$

$\texttt{main} = wdraw(1)$

to produce the following linked client $L_{\texttt{DAO}}$;$C_{\texttt{atk}}$ (modulo re-ordering):

$\texttt{global}\ bal := 100;\ \texttt{global}\ wlet := 0;$

$wdraw = \lambda x.\ \texttt{if}\ !bal \geq x\ \texttt{then}\ (send(x); bal := !bal - x; \texttt{assert}(!bal > 0))\ \texttt{else}\ ();$

$\texttt{public}\ send = \lambda x.wlet := !wlet + x;\ \texttt{if}\ !wlet < 100\ \texttt{then}\ wdraw(x)\ \texttt{else}\ ();$

$\texttt{main} = wdraw(1)$

We can see how $L_{\texttt{DAO}}$ is vulnerable to an attacker such as $C_{\texttt{atk}}$ after linking them. The aim is thus to use bounded analysis to find counterexamples that define clients such as this one.

$$\text{(INT)}\quad \frac{(M,R,S,k) \to (M',R',S',k')}{(\mathcal{E},M,R,S,\mathcal{P},\mathcal{A},k)_p \to (\mathcal{E},M',R',S',\mathcal{P},\mathcal{A},k')_p}$$

$$\text{(PQ)}\quad (\mathcal{E},E[mv],R,S,\mathcal{P},\mathcal{A},k)_p \xrightarrow{\texttt{call}(m,v)} ((m,E) :: \mathcal{E},0,R,S,\mathcal{P}',\mathcal{A},k)_o$$

$$\text{(OQ)}\quad (\mathcal{E},l,R,S,\mathcal{P},\mathcal{A},k)_o \xrightarrow{\texttt{call}(m,v)} ((m,l+1) :: \mathcal{E},mv,R,S,\mathcal{P},\mathcal{A}',k)_p \text{ if } R(m)=\lambda x.M$$

$$\text{(PA)}\quad ((m,l) :: \mathcal{E},v,R,S,\mathcal{P},\mathcal{A},k)_p \xrightarrow{\texttt{ret}(m,v)} (\mathcal{E},l,R,S,\mathcal{P}',\mathcal{A},k)_o$$

$$\text{(OA)}\quad ((m,E) :: \mathcal{E},l,R,S,\mathcal{P},\mathcal{A},k)_o \xrightarrow{\texttt{ret}(m,v)} (\mathcal{E},E[v],R,\mathcal{P},\mathcal{A}',k)_p$$

$$(\mathbf{PC}):\ m \in \mathcal{A} \wedge \mathcal{P}' = \mathcal{P} \cup (\texttt{Meths}(v) \cap dom(R)),\ \ (\mathbf{OC}):\ m \in \mathcal{P} \wedge \mathcal{A}' = \mathcal{A} \cup (\texttt{Meths}(v) \setminus dom(R)).$$

Figure 3 Trace semantics rules. Rules (PQ), (PA) assume the condition (**PC**), and similarly for (OQ),(OA) and (**OC**). $\texttt{Meths}(v)$ contains all method names appearing in v. INT stands for *internal* transition; PQ for *P-question* (i.e. call); PA for *P-answer* (i.e. return). Similarly for OQ and OA.

2.2 Trace Semantics

The semantics we defined only allows us to evaluate terms, and only so long as their method applications only involve methods that can be found in the repository R. We next extend this semantics to encompass libraries and terms that can also call abstract methods. The approach we follow is based on operational game semantics [18, 21, 16] and in particular the semantics is given by means of traces of method calls and returns (called *moves* in game semantics jargon), between the library and its client. In between such moves, the semantics evolves as the operational semantics we already saw.

To maintain a terminating analysis, we need to keep track of an added source of infinite execution, namely endless consecutive calls from an external component: a library will never terminate if its client keeps calling its methods. This leads us to a semantics with two counters, k and l, where k keeps track of internal nested method calls and l records the number of consecutive calls made from the external component. This counter l is orthogonal to k and is refreshed at every call to the external context.

When computing the semantics of a library, the library and its methods are the *Player (P)* of the computation game, while the (intended) client is the *Opponent (O)*. As the semantics is given in absence of an actual client, O actually represents every possible client. When computing the semantics of a client, the roles are reversed. In both cases, though, the same sets of rules is used and there is no need to specify who is P and O in the semantics.

The trace semantics uses *game configurations*, which are divided into *P-configurations* and *O-configurations* given respectively as:

$$(\mathcal{E},M,R,S,\mathcal{P},\mathcal{A},k)_p \quad \text{and} \quad (\mathcal{E},l,R,S,\mathcal{P},\mathcal{A},k)_o\,.$$

In a P-configuration, a term M is being evaluated – this is P's role. In an O-configuration, an external call has been made and the semantics waits for O to either return that call, or reply itself with another call. The components $M,R,S,\mathcal{P},\mathcal{A},k,l$ are as above, while $\mathcal{E}$ is an *evaluation stack*:

$$\mathcal{E} \ ::= \ \varepsilon \mid (m,E) :: \mathcal{E} \mid (m,l) :: \mathcal{E}$$

which keeps track of the computations that are on hold due to external calls. The trace semantics is generated by the rules given in Figure 3.

The formulation follows closely the operational game semantics technique. For example, from a P-configuration $(\mathcal{E},M,R,S,\mathcal{P},\mathcal{A},k)_p$, there are 3 options:

1. If M can make an internal reduction, i.e. in the operational semantics in context (R, S, k), then $(\mathcal{E}, M, R, S, \mathcal{P}, \mathcal{A}, k)_p$ performs this reduction (via (INT)).
2. If M is stuck at a method application for a method that is not in the repository R, then that method must be abstract (i.e. external) and needs to be called externally. This is achieved be issuing a call move and moving to an O-configuration (via (PQ)). The current evaluation context and the called method name are stored, in order to resume once the call is returned (via (OA)).
3. If M is a value and the evaluation stack is non-empty, then P has completed a method call that was issued by O (via (OQ)) and can now return (via (PA)).

On the other hand, from an O-configuration $(\mathcal{E}, l, R, S, \mathcal{P}, \mathcal{A}, k)_o$, there are 2 options:

1. either return the last open method call (made by P) via (OA), or
2. call one of the public methods (from $\mathcal{P}$) using (OQ).

The role of conditions (PC) and (OC) is to ensure that each player calls the methods owned by the other, or returns their own, and update the sets of public and abstract names according to the method names passed inside v.

▶ Remark 5. The novelty of Figure 3 with respect to previous work on trace semantics for open libraries (e.g. [25]) lies in the use of l in order to bound the ability of O to ask repeated questions for finite analysis. The way rules (OQ) and (PA) are designed is such that any sequence of consecutive O-calls and P-returns has maximum length $2n$ if we bound l to n (i.e. $l \leq n$), as each such pair of moves increases l by 1. On the other hand, each P-call supplies to O a fresh counter ($l = 0$) to be used in contiguous (OQ)-(PA)'s. Thus, l can be seen as keeping track of the *insistence* of O in calling.

Finally, we can define the trace semantics of libraries.

▶ **Definition 6.** *Let L be a library. The semantics of L is :*

$$[\![L]\!] = \{(\tau, \rho) \mid (L, \emptyset, \emptyset, \emptyset, \emptyset) \xrightarrow{bld}{}^{*} (\varepsilon, R, S, \mathcal{P}, \mathcal{A}) \wedge (\varepsilon, 0, R, S, \mathcal{P}, \mathcal{A}, 0)_o \xrightarrow{\tau} \rho\}$$

We say that $[\![L]\!]$ fails *if it contains some* $(\tau, (\mathcal{E}, E[\texttt{assert}(0)], \cdots))$.

▶ **Example 7.** Consider the DAO example as library $L_{\texttt{DAO}}$ once again. Evaluating the game semantics we know the following sequence is in $[\![L_{\texttt{DAO}}]\!]$. For economy, we hide $R, \mathcal{P}, \mathcal{A}$ and show only the top of the stack in the configurations. We also use $m(v)?$ and $m(v)!$ for calls and returns. We write S_i for the store $[bal \mapsto i]$.

$$\begin{aligned}
&(\varepsilon, 0, S_{100}, 0)_o \xrightarrow{wdraw(42)?} ((wdraw, 1), wdraw(42), S_{100}, 0)_p \\
&\quad \to^* ((wdraw, 1), E[send(42)], S_{100}, 1)_p \xrightarrow{send(42)?} ((send, E), 2, S_{100}, 1)_o \\
&\quad \xrightarrow{wdraw(100)?} ((wdraw, 1), wdraw(100), S_{100}, 1)_p \\
&\quad \to^* ((wdraw, 1), E'[send(100)], S_{100}, 2)_p \xrightarrow{send(100)?} ((send, E), 2, S_{100}, 2)_o \\
&\quad \xrightarrow{send(())!} ((wdraw, 1), E'[()], S_{100}, 2)_p \to^* ((wdraw, 1), (), S_0, 2)_p \\
&\quad \xrightarrow{wdraw(())!} ((send, E), 1, S_0, 2)_o \xrightarrow{send(())!} ((wdraw, 1), E[()], S_0, 1)_p \\
&\quad \to^* ((wdraw, 1), E[\texttt{assert}(-42 \geq 0)], S_{-42}, 1)_p
\end{aligned}$$

This transition sequence is an instance of the symbolic trace provided in the Introduction. Here, a call is made with parameter 42, and a reentrant call with 100, which leads to the assertion violation $\texttt{assert}(-42 \geq 0)$. Note that a bound of $k \leq 2$ is sufficient to find this assertion violation.

We next establish two focal properties of the trace semantics: bounding k and l ensures termination (Theorem 8), and that it is sound and complete with respect to library errors (Theorem 9). Notice Theorem 9 captures both soundness and completeness as it states that the game semantics eventually reaches every error that is concretely reachable for any client while finding only errors that can be reached concretely by a definable client.

▶ **Theorem 8** (Boundedness). *For any game configuration ρ, provided an upper bound k_0 and l_0 for call counters k and l, the labelled transition system starting from ρ is strongly normalising.*

Proof. For any transition sequence $\rho = \rho_0 \to \cdots \to \rho_i \to \dots$ and each $i > 0$, we set the following two classes of configurations:

$$(A) = \{\rho_i \mid |\rho_i| < |\rho_{i-1}|\} \qquad (B) = \{\rho_i \mid \exists j < i-1.\ |\rho_i| < |\rho_j|\}$$

where $|\rho| = (k_0 - k, |M|, l_0 - l)$ is the *size* of ρ, and $|\rho| < |\rho'|$ is defined by the lexicographic ordering of the triple $(k_0 - k, |M|, l_0 - l)$, with bounds k_0 and l_0 such that $k \leq k_0$ and $l \leq l_0$ for semantic transitions to be applicable. If not present in the configuration, we look at the evaluation stack $\mathcal{E}$ to find the top-most missing component. In other words, opponent configurations will have size $(k_0 - k, |E|, l_0 - l)$ where E is the top-most one in $\mathcal{E}$, whereas proponent configurations will have size $(k_0 - k, |M|, l_0 - l)$ where l is the top-most one in $\mathcal{E}$.

We approach the proof in two steps: (1) we show that, for any transition sequence out of ρ, each reachable configuration belongs to (at least) one of the above classes; and (2) prove that the classes form a terminating sequence. For (1), considering all moves available to ρ, we have the following cases.

1. If $\rho \to \rho'$ is an (Int) move, we have two possibilities.
   a. For a transition $(E[(\!|v|\!)], R, S, k) \to (E[v], R, S, k+1)$, where $k+1 \leq k_0$, we have a class (B) configuration since there must be a $(E[mv], R, S, k)$ such that $(E[mv], R, S, k) \to^* (E[v], R, S, k)$ which is lexicographically ordered since $|v| < |mv|$.
   b. Every other transition sequence is class (A) since they reduce the size of the term.
2. If $\rho \to \rho'$ is a (Pq) move, we have that ρ' is a class (A) configuration since $(k, |E|, l_0) < (k, |E[mv]|, l_0 - l)$ by lexicographic ordering.
3. If $\rho \to \rho'$ is an (Oa) move, we have a transition

   $$((m, E) :: \mathcal{E}, l, \dots, k)_o \xrightarrow{ret(m,v)} (\mathcal{E}, E[v], \dots, k)_p$$

   which must be a result of the prior proponent question, meaning $\mathcal{E}$ holds an l' on top. We thus have the following sequence

   $$(\mathcal{E}, E[mv], \dots, k)_p \to^* (\mathcal{E}, E[v], \dots, k)_o$$

   where $(k, |E[v]|, l) < (k, |E[mv]|, l')$, so ρ' is a class (B) configuration.
4. If $\rho \to \rho'$ is an (Oq) move, we have the transition

   $$\begin{aligned}(\mathcal{E}, l, \dots, k)_o &\xrightarrow{call(m,v)} ((m, l+1) :: \mathcal{E}, mv, \dots, k)_p \\ &\to ((m, l+1) :: \mathcal{E}, (\!|M\{v/x\}|\!), \dots, k+1)\end{aligned}$$

   Simplifying the transition, we remove the configuration in between and take

   $$(\mathcal{E}, l, R, S, \mathcal{P}, \mathcal{A}, k)_o \xrightarrow{call(m,v)} ((m, l+1) :: \mathcal{E}, (\!|M\{v/x\}|\!), R, S, \mathcal{P}, \mathcal{A}, k+1)_p$$

   to be our new equivalent transition. We thus have that ρ' is a class (A) configuration since $(k_0 - (k+1), |(\!|M\{v/x\}|\!)|, l_0 - (l+1)) < (k_0 - k, |E|, l_0 - l)$ by lexicographic ordering.

5. If $\rho \to \rho'$ is a (PA) move, we have the transition

$$((m,l) :: \mathcal{E}, v, \dots, k)_p \xrightarrow{ret(m,v)} (\mathcal{E}, l, \dots, k)_o$$

which must be the result of a prior opponent question

$$(\mathcal{E}, l+1, \dots, k)_o \xrightarrow{call(m,v)} ((m,l) :: \mathcal{E}, (\!|M\{v/x\}|\!), \dots, k+1)_p$$
$$\to^* ((m,l) :: \mathcal{E}, (\!|v|\!), \dots, k+1)_p \to ((m,l) :: \mathcal{E}, v, \dots, k)_p \xrightarrow{ret(m,v)} (\mathcal{E}, l, \dots, k)_o$$

where E' is the topmost evaluation context in $\mathcal{E}$. We thus have that $(k_0 - k, E', l_0 - l) < (k_0 - k, E', l_0 - (l+1))$, so ρ' is a class (B) configuration.

For (2), let us assume there is an infinite sequence

$$\rho_0 \to \dots \to \rho_j \to \dots \to \rho_i \to \dots$$

Since all reachable configurations fall into either (A) or (B) class, we know that the sequence must comprise only (A) and (B) configurations. In this infinite sequence, we know that all sequences of (A) configurations are in descending size, so (A) sequences cannot be infinite. We also observe that (B) configurations are padded with (A) sequences. For instance, if ρ_i is a (B) configuration, and ρ_j is its matching configuration, there may exist nested (B) configurations between ρ_j and ρ_i, as well as (A) sequences padding these.

Additionally, these (B) configurations can only occur as a return to a call, so we know they only occur together with the introduction of evaluation boxes $(\!|\bullet|\!)$. Since these brackets occur in pairs and are introduced in a nested fashion, we know $\mathcal{E}$ can only contain evaluation contexts with well-bracketed evaluation boxes, meaning that there cannot be interleaved sequences of (B) configurations where their target configurations intersect. More specifically, the sequence

$$\rho_0 \to \dots \to \rho_j \to \dots \to \rho'_j \to \dots \to \rho_i \to \dots \to \rho'_i \to \dots$$

where ρ'_i matches ρ'_j and ρ_i matches ρ_j is not possible.

Now, ignoring all (A) and nested (B) sequences, we are left with an infinite stream of top-level (B) sequences which are also in descending order. Since starting size is finite, we cannot have an infinite stream of (B) sequences. Thus, the assumption that the sequence is infinite does not hold, meaning our semantics is strongly normalising. ◀

▶ **Theorem 9** (S and C). *We call a client* good *if it contains no assertions. For any library L, the following are equivalent:*

1. *$[\![L]\!]$ fails (reaches an assertion violation)*
2. *there exists a good client C such that $[\![L;C]\!]$ fails*

Proof. 1 to 2: Suppose now that $(\tau, \rho) \in [\![L]\!]$ for some trace τ and failed ρ. By Theorem 11, we have that there is a good client C realising the trace τ. So then, by Lemma 10, we have that $[\![L;C]\!]$ fails.

2 to 1: Suppose $[\![L;C]\!]$ fails for some good client C. Then, by Lemma 10, there are τ, ρ, ρ' such that $(\tau, \rho) \in [\![L]\!]$, $(\tau, \rho') \in [\![C]\!]$, and ρ is failed (i.e. is of the shape $(\mathcal{E}, E[\texttt{assert}(0)], \cdots)$). ◀

The latter relies on an auxiliary lemma (well-composing of libraries and clients), and a definability result akin to game semantics definability arguments (see Appendix D).

▶ **Lemma 10** (L-C Compositionality). *For any library L and compatible good client C, $[\![L;C]\!]$ fails if and only if there exist $(\tau_1, \rho_1) \in [\![L]\!]$ and $(\tau_2, \rho_2) \in [\![C]\!]$ such that $\tau_1 = \tau_2$ and $\rho_1 = (\mathcal{E}, E[\texttt{assert}(0)], \cdots)$.*

▶ **Theorem 11** (Definability). *Let L be a library and $(\tau, \rho) \in [\![L]\!]$. There is a good client C compatible with L such that $(\tau, \rho') \in [\![C]\!]$ for some ρ'.*

3 Symbolic Semantics

Checking libraries for errors using the semantics of the previous section is infeasible, even when the traces are bounded in length, as ground values are concretely represented. In particular, integer values provided by O as arguments to calls or return values range over all integers. The typical way to mitigate this limitation is to execute the semantics symbolically, using symbolic variables for integers and path conditions to bind these variables to plausible values. We use this technique to devise a symbolic version of the trace semantics, corresponding to a symbolic execution which will enable us in the next sections to introduce a practical method and implementation for checking libraries for errors. The symbolic semantics is fully formal, closely following the developments of the previous section, and allows us to prove a strong form of correspondence between concrete and symbolic semantics (a bisimulation).

Apart from integers, another class of concrete values provided by O are method names. For them, the semantics we defined is symbolic by design: all method names played by O are going to be fresh and therefore picking just one of those fresh choices is sufficient (formally speaking, the semantics lives in nominal sets [31]). The reason why using fresh names for methods played by O is sound is that the effect of O calling a higher-order public method with an argument m (where m is another public method), and with $\lambda x.mx$, is equivalent as far as reachability of an error is concerned. In the latter case, the client semantics would create a fresh name m', bind it to $\lambda x.mx$, and pass m' as an argument. We therefore just focus on this latter case.

The symbolic semantics involves terms that may contain symbolic values for integers. We therefore extend the syntax for values and terms to include such values, and abuse notation by continuing to use M to range over them. We let `SInts` be a set of symbolic integers ranged over by κ and variants, and define:

$$\begin{array}{rrcl} \textit{Sym.Values} & \tilde{v} & ::= & m \mid i \mid () \mid \kappa \mid \tilde{v} \oplus \tilde{v} \mid \langle \tilde{v}, \tilde{v} \rangle \\ \textit{Sym.Terms} & M & ::= & \cdots \mid \kappa \end{array}$$

where, in $\tilde{v} \oplus \tilde{v}$, not both $\tilde{v}$ can be integers. We moreover use a symbolic environment to store symbolic values for references, but also to keep track of arithmetic performed with symbolic integers. More precisely, we let σ be a finite partial map from the set $\texttt{SInts} \cup \texttt{Refs}$ to symbolic values. Finally, we use pc to range over program conditions, which will be quantifier-free first-order formulas with variables taken from `SInts`, and with $\top, \bot$ denoting true and false respectively.

The semantics for closed symbolic terms involves configurations of the form (M, R, σ, pc, k). Its rules include copies of those from Figure 2 (top) where the pc and σ are simply carried over. For example:

$$(E[\lambda x.M], R, \sigma, pc, k) \rightarrow_s (E[m], R \uplus \{m \mapsto \lambda x.M\}, \sigma, pc, k)$$

where m is fresh. On the other hand, the following rules directly involve symbolic reasoning:

$$(E[\texttt{assert}(\kappa)], R, \sigma, pc, k) \rightarrow_s (E[\texttt{assert}(0)], \sigma, pc \wedge (\kappa = 0), k)$$

$$
\begin{array}{ll}
(\widetilde{\text{INT}}) & \dfrac{(M,R,\sigma,pc,k) \rightarrow_s \ (M',R',\sigma,pc',k')}{(\mathcal{E},M,R,\mathcal{P},\mathcal{A},\sigma,pc,k)_p \rightarrow_s (\mathcal{E},M',R',\mathcal{P},\mathcal{A},\sigma',pc',k')_p} \\
(\widetilde{\text{PQ}}) & (\mathcal{E},E[m\tilde{v}],R,\mathcal{P},\mathcal{A},\sigma,pc,k)_p \xrightarrow{\texttt{call}(m,\tilde{v})}_s ((m,E) :: \mathcal{E},0,R,\mathcal{P}',\mathcal{A},\sigma,k)_o \\
(\widetilde{\text{OQ}}) & (\mathcal{E},l,R,\mathcal{P},\mathcal{A},\sigma,pc,k)_o \xrightarrow{\texttt{call}(m,\tilde{v})}_s ((m,l+1) :: \mathcal{E},m\tilde{v},R,\mathcal{P},\mathcal{A}',\sigma,pc,k)_p \\
(\widetilde{\text{PA}}) & ((m,l) :: \mathcal{E},\tilde{v},R,\mathcal{P},\mathcal{A},\sigma,pc,k)_p \xrightarrow{\texttt{ret}(m,\tilde{v})}_s (\mathcal{E},l,R,\mathcal{P}',\mathcal{A},\sigma,pc,k)_o \\
(\widetilde{\text{OA}}) & ((m,E) :: \mathcal{E},l,R,\mathcal{P},\mathcal{A},\sigma,pc,k)_o \xrightarrow{\texttt{ret}(m,\tilde{v})}_s (\mathcal{E},E[\tilde{v}],R,\mathcal{P},\mathcal{A}',\sigma,pc,k)_p
\end{array}
$$

$(\widetilde{\mathbf{PC}})$ $m \in \mathcal{A}$ and $\mathcal{P}' = \mathcal{P} \cup (\texttt{Meths}(\tilde{v}) \cap \mathit{dom}(R))$.

$(\widetilde{\mathbf{OC}})$ $m \in \mathcal{P}$ and $(\tilde{v}',\mathcal{A}') \in \mathsf{symval}(\theta,\mathcal{A})$ where θ is the expected type of $\tilde{v}$. Moreover:

$$
\texttt{symval}(\theta,\mathcal{A}) = \begin{cases}
\{((),\mathcal{A})\} & \text{if } \theta = \texttt{unit} \\
\{(\kappa,\mathcal{A} \uplus \{\kappa\}) \mid \ \kappa \text{ is fresh in } \mathit{dom}(\sigma) \uplus \mathcal{A}\} & \text{if } \theta = \texttt{int} \\
\{(m,\mathcal{A} \uplus \{m\}) \mid m \text{ is fresh in } \mathit{dom}(R) \uplus \mathcal{A}\} & \text{if } \theta = \theta_1 \rightarrow \theta_2 \\
\{(\langle \tilde{v}_1,\tilde{v}_2\rangle,\mathcal{A}_2) \mid (\tilde{v}_1,\mathcal{A}_1) \in \texttt{symval}(\theta_1,\mathcal{A}) & \text{if } \theta = \theta_1 \times \theta_2 \\
\qquad\qquad\qquad (\tilde{v}_2,\mathcal{A}_2) \in \texttt{symval}(\theta_2,\mathcal{A}_1)\} &
\end{cases}
$$

Figure 4 Symbolic trace semantics rules. Rules $(\widetilde{\text{PQ}})$, $(\widetilde{\text{PA}})$ assume the condition $(\widetilde{\mathbf{PC}})$, and similarly for $(\widetilde{\text{OQ}})$,$(\widetilde{\text{OA}})$ and $(\widetilde{\mathbf{OC}})$. $(\widetilde{\text{OQ}})$,$(\widetilde{\text{OA}})$ introduce $\tilde{v}$ and thus are non-deterministic.

$$
\begin{array}{l}
(E[\texttt{assert}(\kappa)],R,\sigma,pc,k) \rightarrow_s (E[()],R,\sigma,pc \wedge (\kappa \neq 0),k) \\
(E[!r],R,\sigma,pc,k) \rightarrow_s (E[\sigma(r)],R,\sigma,pc,k) \\
(E[r := \tilde{v}],R,\sigma,pc,k) \rightarrow_s (E[()],R,\sigma[r \mapsto \tilde{v}],pc,k) \\
(E[\tilde{v}_1 \oplus \tilde{v}_2],R,\sigma,pc,k) \rightarrow_s (E[\kappa],R,\sigma \uplus \{\kappa \mapsto \tilde{v}_1 \oplus \tilde{v}_2\},pc,k) \quad \text{where } \kappa \text{ is fresh} \\
(E[\texttt{if } \kappa \texttt{ then } M_1 \texttt{ else } M_0],R,\sigma,pc,k) \rightarrow_s (E[M_0],R,\sigma,pc \wedge (\kappa = 0),k) \\
(E[\texttt{if } \kappa \texttt{ then } M_1 \texttt{ else } M_0],R,\sigma,pc,k) \rightarrow_s (E[M_1],R,\sigma,pc \wedge (\kappa \neq 0),k)
\end{array}
$$

and where $\tilde{v}_1 \oplus \tilde{v}_2$ is a symbolic value (for $i_i \oplus i_2$ the rule from Figure 1 applies).

We now extend the symbolic setting to the trace semantics. We define symbolic configurations for P and O respectively as:

$$
(\mathcal{E},M,R,\mathcal{P},\mathcal{A},\sigma,pc,k)_p \qquad\qquad (\mathcal{E},l,R,\mathcal{P},\mathcal{A},\sigma,pc,k)_o
$$

with evaluation stack $\mathcal{E}$, proponent term M, counters $k,l \in \mathbb{N}$, method repository R, public method name set $\mathcal{P}$, σ and pc as previously. The abstract name set $\mathcal{A}$ is now a finite subset of $\texttt{Meths} \cup \texttt{SInts}$, as we also need to keep track of the symbolic integers introduced by O (in order to be able to introduce fresh such names). The rules for the symbolic trace semantics are given in Figure 4. Note that O always refreshes names it passes. This is a sound overapproximation of all names passed for the sake of analysis.

Similarly to Definition 6, we can define the symbolic semantics of libraries.

▶ **Definition 12.** *Given library L, the symbolic semantics of L is:*

$$
\begin{array}{l}
[\![L]\!]_s = \{(\tau,\rho) \mid (L,\emptyset,\emptyset,\emptyset,\emptyset) \xrightarrow{bld}{}^* (\varepsilon,R,S,\mathcal{P},\mathcal{A}) \\
\qquad\qquad \wedge\, (\varepsilon,0,R,\mathcal{P},\mathcal{A},S,\top,0)_o \xrightarrow{\tau}_s \rho \ \wedge \ \exists \mathcal{M}.\, \mathcal{M} \vDash \rho(\sigma)^\circ \wedge \rho(pc)\}
\end{array}
$$

where $\rho(\chi)$ is component χ in configuration ρ, and $\mathcal{M}$ is a model as defined in the next section. We say that $[\![L]\!]_s$ fails *if it contains some $(\tau,(\mathcal{E},E[\texttt{assert}(0)],\cdots))$.*

The symbolic rules follow those of the concrete semantics, the biggest change being the treatment of symbolic values played by O. Condition $(\widetilde{\mathbf{OC}})$ stipulates that O plays distinct fresh symbolic integers as well as fresh method names, in each appropriate position in $\tilde{v}$, and all these names are included in the set $\mathcal{A}$.

▶ **Example 13.** As with Example 7, we consider the DAO attack. Running the symbolic semantics, we find the following minimal class of errors. We write $\sigma_{\tilde{v}}$ for a symbolic environment $[\mathit{bal} \mapsto \tilde{v}]$.

$$\begin{aligned}
&(\varepsilon, 2, \sigma_{100}, k_0)_o \xrightarrow{\mathit{wdraw}(\kappa_1)?} ((\mathit{wdraw}, 1), \mathit{wdraw}(\kappa_1), \sigma_{100}, 2)_p \\
&\quad \to^* ((\mathit{wdraw}, 1), E[\mathit{send}(\kappa_1)], \sigma_{100}, 1)_p \xrightarrow{\mathit{send}(\kappa_1)?} ((\mathit{send}, E), 2, \sigma_{100}, 1)_o \\
&\quad \xrightarrow{\mathit{wdraw}(\kappa_2)?} ((\mathit{wdraw}, 1), \mathit{wdraw}(\kappa_2), \sigma_{100}, 1)_p \\
&\quad \to^* ((\mathit{wdraw}, 1), E'[\mathit{send}(\kappa_2)], \sigma_{100}, 0)_p \xrightarrow{\mathit{send}(\kappa_2)?} ((\mathit{send}, E), 2, \sigma_{100}, 0)_o \\
&\quad \xrightarrow{\mathit{send}(())!} ((\mathit{wdraw}, 1), E'[()], \sigma_{100}, 0)_p \\
&\quad \to^* ((\mathit{wdraw}, 1), (), \sigma_{100-\kappa_2}, 0)_p \xrightarrow{\mathit{wdraw}(())!} ((\mathit{send}, E), 1, \sigma_{100-\kappa_2}, 0)_o \\
&\quad \xrightarrow{\mathit{send}(())!} ((\mathit{wdraw}, 1), E[()], \sigma_{100-\kappa_2}, 1)_p \\
&\quad \to^* ((\mathit{wdraw}, 1), E[\mathtt{assert}(!\mathit{bal} \geq 0)], \sigma_{100-\kappa_2-\kappa_1}, 1)_p
\end{aligned}$$

For this to be a valid error, we require $(\kappa_1, \kappa_2 \leq 100) \wedge (100 - \kappa_2 - \kappa_1 < 0)$ to be satisfiable. Taking assignment $\{\kappa_1 \mapsto 100, \kappa_2 \mapsto 1\}$, we show the path is valid.

3.1 Soundness

The main result of this section is establishing the soundness of the symbolic semantics: a trace and a specific configuration can be achieved symbolically iff they can be achieved concretely as well. In fact, we will need to quantify this statement as, by construction, the symbolic semantics requires O to always place fresh method names, whereas in the concrete semantics O is given the freedom to play old names as well. What we show is that the symbolic semantics corresponds (via *bisimilarity*) to a restriction of the concrete semantics where O plays fresh names only. This restriction is sound, in the sense that it is sufficient for identifying when a configuration can fail. We make this precise below.

A ***model*** $\mathcal{M}$ is a finite partial map from symbolic integers to concrete integers. Given such an $\mathcal{M}$ and a formula ϕ, we define $\mathcal{M} \models \phi$ using a standard first-order logic interpretation with integers and arithmetic operators (in particular, we require that all symbolic integers in ϕ are in the domain of $\mathcal{M}$). Moreover, for any symbolic term M (or trace, move, etc.), we denote by $M\{\mathcal{M}\}$ the concrete term we obtain by substituting any symbolic integer κ of M with its corresponding concrete integer $\mathcal{M}(\kappa)$. Finally, given a symbolic environment σ, we define its formula representation σ° recursively by:

$$\emptyset^\circ = \top, \quad (\sigma \uplus \{r \mapsto v\})^\circ = \sigma^\circ, \quad (\sigma \uplus \{\kappa \mapsto v\})^\circ = \sigma^\circ \wedge (\kappa = v).$$

We now define notions for equivalence between symbolic and concrete configurations. Let $\mathcal{M}$ be a model. For any concrete configuration $\rho = (\mathcal{E}, \chi, R, S, \mathcal{P}, \mathcal{A}, k)$ and symbolic configuration $\rho_s = (\mathcal{E}', \chi', R', \mathcal{P}', \mathcal{A}', \sigma, pc, k')$, we say they are *equivalent in* $\mathcal{M}$, written $\rho =_{\mathcal{M}} \rho_s$, if:

- $(\mathcal{E}, \chi, R) = (\mathcal{E}', \chi', R')\{\mathcal{M}\}, \mathcal{P} = \mathcal{P}', \mathcal{A} = \mathcal{A}' \cap \mathtt{Meths}$ and $S = (\sigma \restriction \mathtt{Refs})\{\mathcal{M}\}$;
- $\mathrm{dom}(\mathcal{M}) = (\mathcal{A}' \cup \mathrm{dom}(\sigma)) \cap \mathtt{SInts}$ and $\mathcal{M} \models pc \wedge \sigma^\circ$.

The notion of equivalence we require between concrete configurations and their symbolic counterparts is behavioural equivalence, modulo O playing fresh names.

More precisely, a transition $\rho \xrightarrow{\chi} \rho'$ is called *O-refreshing* if, when ρ is an O-configuration and $\chi = \mathtt{call}/\mathtt{ret}(m, v)$ then all names in v are fresh and distinct. A set $\mathcal{R}$ with elements of the form $(\rho, \mathcal{M}, \rho_s)$ is a ***bisimulation*** if, whenever $(\rho, \mathcal{M}, \rho_s) \in \mathcal{R}$, written $\rho\, \mathcal{R}_{\mathcal{M}}\, \rho_s$ then $\rho =_{\mathcal{M}} \rho_s$ and, using χ to range over moves and ε (i.e. no move):

- if $\rho \xrightarrow{\chi} \rho'$ is O-refreshing then there exists $\mathcal{M}' \supseteq \mathcal{M}$ such that $\rho_s \xrightarrow{\chi_s}_s \rho'_s$, with $\chi = \chi_s\{\mathcal{M}'\}$, and $\rho' \mathcal{R}_{\mathcal{M}'} \rho'_s$;
- if $\rho_s \xrightarrow{\chi}_s \rho'_s$ then there exists $\mathcal{M}' \supseteq \mathcal{M}$ such that $\rho \xrightarrow{\chi\{\mathcal{M}'\}}_G \rho'$ and $\rho' \mathcal{R}_{\mathcal{M}'} \rho'_s$.

We let $\sim$ be the largest bisimulation relation: $\rho \sim_{\mathcal{M}} \rho_s$ iff there is bisimulation $\mathcal{R}$ such that $\rho \mathcal{R}_{\mathcal{M}} \rho_s$.

We can show that concrete and symbolic configurations are bisimilar.

▶ **Lemma 14.** *Given ρ, ρ_s a concrete and symbolic configuration respectively, and $\mathcal{M}$ a model such that $\rho =_{\mathcal{M}} (\rho')$, we have $\rho \sim_{\mathcal{M}} \rho_s$.*

Proof (sketch). We show that $\{(\rho, \mathcal{M}, \rho') \mid \rho =_{\mathcal{M}} \rho'\}$ is a bisimulation. ◀

Next, we argue that O-refreshing transitions suffice for examining failure of concrete configurations. Indeed, suppose τ is a trace leading to fail, and where O plays an old name m in argument position in a given move. Then, τ can be simulated by a trace τ' that uses a fresh m' in place of m. If m is an O-name, we obtain τ' from τ by following exactly the same transitions, only that some P-calls to m are replaced by calls to m' (and accordingly for returns). If, on the other hand, m is a P-name, then the simulation performed by τ' is somewhat more elaborate: some internal calls to m will be replaced by P-calls to m', immediately followed by the required calls to m (and dually for returns).

▶ **Lemma 15** (O-Refreshing)**.** *Let ρ be a concrete configuration. Then, ρ fails iff it fails using only O-refreshing transitions.*

With the above, we can prove soundness.

▶ **Theorem 16** (Soundness)**.** *For any L, $[\![L]\!]$ fails iff $[\![L]\!]_s$ fails.*

Proof. Lemma 14 implies that $[\![L]\!]_s$ fails iff $[\![L]\!]$ fails with O-refreshing transitions, which in turns occurs iff $[\![L]\!]$ fails, by Lemma 15. ◀

3.2 Bounded Analysis for Libraries

Definition 12 states how the symbolic trace semantics can be used to independently check libraries for errors. As with the trace semantics in Definition 6, this is strongly normalising when given an upper limit to the call counters. As such, $[\![L]\!]_s$ with counter bounds $k_0, l_0 \in \mathbb{N}$, for k, l respectively, defines a finite set (modulo selecting of fresh names) of reachable valid configurations within $k \leq k_0, l \leq l_0$, where validity is defined by the satisfiability of the symbolic environment σ and the path condition *pc* of the configuration reached. By virtue of Theorems 9 and 16, every valid reachable configuration that is failed (evaluates an invalid assertion) is realisable by some client. And viceversa.

Given a library L, taking $\mathcal{F}[\![L]\!]_s$ to be all reachable final configurations, we have the exhaustive set of paths L can reach. In $\mathcal{F}[\![L]\!]_s$, every failed configuration (τ, ρ), i.e. such that ρ holds a term $E[\mathtt{assert}(0)]$, defines a reachable assertion violation, where τ is a true counterexample. Hence, to check L for assertion violations it suffices to produce a finite representation of the set $\mathcal{F}[\![L]\!]_s$. One approach is to bound the depth of analysis by setting an

Table 1 Table recording performance of HOLiK on our benchmarks.

	$l \leq 1$	$l \leq 2$	$l \leq 3$
$k \leq 2$	226/70/45 (555s)	5708/60/44 (4710s)	9656/3/23 (12471s)
$k \leq 3$	1254/67/51 (1475s)	4092/27/18 (13482s)	4187/17/12 (16649s)
$k \leq 4$	3392/63/48 (3180s)	3069/19/14 (15903s)	1335/12/10 (17765s)
$k \leq 5$	3659/57/45 (4787s)	895/15/10 (16757s)	215/11/9 (17796s)

$a/b/c$ (d) for a traces found in b successful runs taking d seconds in total where c out of 59 unsafe files were found to have bugs, per bound.
59 of 59 unsafe files found to have bugs over the various bounds checked

upper bound to the call counters, using a name generator to make deterministic the creation of fresh names, and then exhaustively search all final configurations for failed elements. In the following section we implement this routine and test it.

4 Implementation and Experiments

We implemented the syntax and symbolic trace semantics (symbolic games) for HOLi in the $\mathbb{K}$ semantic framework [32] as a proof of concept, and tested it on 70 sample libraries.[1] Using $\mathbb{K}$'s option to exhaustively expand all transitions, $\mathbb{K}$ is able to build a closure of all applicable rules. By providing a bound on the call counters, we produce a finite set of all reachable valid symbolic configurations up to the given depth (equivalent to finding every valid $\rho \in \mathcal{F}[\![L]\!]_s$) which thus implements our bounded symbolic execution.

We wrote and adapted examples of coding errors into a set of 70 sample libraries written in HOLi, totalling 6,510 lines of code (LoC). Examples adapted from literature include: reentrancy bugs from smart contracts [3, 23]; variations of the "awkward example" [30]; various programs from the MoCHi benchmark [35]; and simple implementations related to concurrent programming (e.g. flat combining and race conditions) where errors may occur in a single thread due to higher-order behaviour. We also combined several libraries, by concatenating refactored method and reference definitions, to generate larger libraries that are harder to solve. Combined files range from 150 to 520 LoC.

We ran HOLiK on all sample libraries, lexicographically increasing the bounds from $k \leq 2, l \leq 1$ to $k \leq 5, l \leq 3$ (totalling 78,120 LoC checked), with a timeout set to five minutes per library. We start from $k \leq 2$ because it provides the minimum nesting needed to observe higher-order semantics. All experiments ran on an Ubuntu 19.04 machine with 16GB RAM, Intel Core i7 3.40GHz CPU, with intermediate calls to Z3 to prune invalid configurations. Per bound, the number of counterexamples found, the time taken in seconds, and the execution status, i.e. whether it terminated or not, are recorded in Table 1.

We can observe that independently increasing the bounds for k and l causes exponential growth in the total time taken, which is expected from symbolic execution. Note that the time tends towards 21000 seconds because of the timeout set to 5 minutes for 70 programs. In particular, while the number of errors found grows exponentially with respect to the increase in bounds – which is due to the exponential growth in paths – this trend does not continue indefinitely because programs start timing out without reporting any errors as the bounds

[1] The tool and its benchmarks can be found at: `https://github.com/LaifsV1/HOLiK`.

grow. With bounds $k \leq 2$ and $l \leq 1$, all 70 programs in our benchmark were successfully analysed, though not all minimal errors were found until the bounds were increased further. Cumulatively, all unsafe programs in our benchmark were correctly identified.

While the table may suggest that increasing bound for l is more beneficial than that for k, the number of errors reported does not imply every trace is useful. For instance, increasing the bound for l can lead to errors re-merging in a higher-order version, which suggests potential gain from a partial order reduction. Overall, the k and l counters are incomparable as they keep track of different behaviours. Finally, since HOLiK was able to handle every file and correctly identified all unsafe files in the benchmark, we conclude that HOLiK, as a proof of concept, captures the full range of behaviours in higher-order libraries. Results suggest that the tool scales up to at least medium-sized programs (<1000 LoC), which is promising because real-world medium-size higher-order programs have been proven infeasible to check with standard techniques (e.g. the DAO withdraw contract was approximately 100 LoC).

5 Related Work

Game semantics techniques have been applied to program equivalence verification by reducing program equivalence to language equivalence in a decidable automata class [15, 1]. Equivalence tools can be used for reachability but, as they perform full verification, they can only cover lower-order recursion-free language fragments. For example, the Coneqct [24] tool can verify the simplified DAO attack, but cannot check higher-order or recursive functions (e.g. the "file lock" and "flat combiner" examples), and operates on integers concretely. Close to our approach is also Symbolic GameChecker [11], which performs symbolic model checking by using a representation of games based on symbolic finite-state automata. The tool works on recursion-free Idealized Algol with first-order functions, which supports only integer references. On the other hand, it is complete (not bounded) on the fragment that it covers.

Besides games techniques, a recent line of work on verification of contracts in Racket [27, 26] is the work closest to ours. Racket contracts exist in a higher-order setting similar to ours, and generalise higher-order pre and post conditions, and thus specify safety. To verify these, [27] defines a symbolic execution based on what they call "demonic context" in prior work [38]. This either returns a symbolic value to a call, or performs a call to a known method within some unknown context, thus approximating all the possible higher-order behaviours, and is equivalent to the role the opponent plays in our games. In [26], the technique is extended to handle state, and finitised for total verification. The approaches are notionally similar to ours, since both amount to Symbolic Execution for an unknown environment. In substance, the techniques are very different and in particular ours is based on a semantics theory which allows us to obtain compositionality and definability results, which are not proven for [26] and proven for [27] only in a stateless setting. On the other hand, Racket contracts can be used for richer verification questions than assertion violations. In terms of tool performance, we provide a comparison of the techniques in Appendix B.

Another relevant line of work is that of verifying programs in the Ethereum Platform. Smart contracts call for techniques that handle the environment, with a focus on reentrancy. Tools like Oyente [23] and Majan [28] use pre-defined patterns to find bugs in the transaction order, but are not sound or complete. ReGuard [22] finds sound reentrancy bugs using a fuzzing engine to generate random transactions to check with a reentrancy automaton. In principle, it may detect reentrancy faster than symbolic execution (native execution is faster [40]), but, is incomplete even in a bounded setting. More closely related to our approach,

[17] considers the possibility of an *unknown* contract $c?$ calling a *known* contract $c*$ at each higher call level. This can be generalised in our game semantics as *abstract* and *public* names calling each other, but their focus is on modelling reentrancy, while we handle the full range of higher-order behaviours.

Like KLEE [4] and jCUTE [36], our implementation is a symbolic execution tool. These are generally able to find first-order counterexamples, but are unable to produce higher-order traces involving unknown code. Particularly, KLEE and jCUTE only handle symbolic calls provided these can be concretised. This partially models the environment, but calls are often impossible to concretise with libraries. The CBMC [6, 20] bounded model checking approach, which also bounds function application to a fixed depth, partially handle calls to unknown code by returning a non-deterministic value to such calls. This is equivalent to a game where only move available to the opponent is to answer questions. This restriction allows CBMC to find some bugs caused by interaction with the environment, but misses errors that arise from transferring flow of control (e.g. reentrancy). The typical BMC approach also misses bugs involving disclosure of names.

Higher-order model checking tools like MoCHi [35] are also related. MoCHi model checks a pure subset of OCaml and is based on predicate abstraction and CEGAR and higher-order recursion scheme model checkers. The modular approach [34] further extends this idea with modular analysis that guesses refinement intersection types for each top-level function. Although generally incomparable, HOLiK covers program features that MoCHi does not: MoCHi does not handle references and support for open code is limited (from experiments, and private communication with the authors).

6 Future Directions

Observing errors resurface deeper in the trace suggests the possibility of defining a partial order for our semantics to obtain equivalence classes for configurations and thus eliminate paths that involve known errors [29, 39]. Additionally, while k and l successfully bound infinite behaviour, a notion of bounding can be arbitrarily chosen. In fact, while we chose to directly bound the sources of infinite behaviour in method calls for simplicity of proofs and implementation, the theory does not prevent the generalisation of k and l as a monotonic cost function that bounds the semantics. It may also be worth considering the elimination of bounds entirely for the sake of unbounded verification. For this, one direction is abstract interpretation [9, 8], which amounts to defining overapproximations for values in our language to then attempt to compute a fixpoint for the range of values that assertions may take. However, defining and using abstract domains that maintain enough precision to check higher-order behaviours, such as reentrancy, is not a simple extension of the theory. Another direction, similar to Coneqct [24], is to define a push-down system for our semantics. Particularly, the approach in [24] is based on the decidability of reachability in fresh-register pushdown automata, and would require overapproximations for methods and integers. As with abstract interpretation, this would require defining abstract domains for methods and integers. While methods could be approximated using a finite set of names, as with k-CFA [37], an extension using integer abstract domains would need refinement to tackle reentrancy attacks. Finally, MoCHi [35] shows that it is possible to use CEGAR and higher-order recursion schemes for unbounded verification of higher-order programs. However, an extension of the MoCHi approach to include references and open code is not obvious.

References

1 S. Abramsky, D. R. Ghica, L. Ong, and A. Murawski. Algorithmic game semantics and component-based verification. In *Proceedings of SAVBCS 2003: Specification and Verification of Component-Based Systems, Workshop at ESEC/FASE 2003*, pages 66–74, 2003. published as Technical Report 03-11, Department of Computer Science, Iowa State University. URL: `http://www.cs.iastate.edu/~leavens/SAVBCS/2003/papers/SAVCBS03.pdf`.

2 Samson Abramsky and Guy McCusker. Game semantics. In Ulrich Berger and Helmut Schwichtenberg, editors, *Computational Logic*, pages 1–55, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

3 Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts sok. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, pages 164–186, New York, NY, USA, 2017. Springer-Verlag New York, Inc. `doi:10.1007/978-3-662-54455-6_8`.

4 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, Berkeley, CA, USA, 2008. USENIX Association. URL: `http://dl.acm.org/citation.cfm?id=1855741.1855756`.

5 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.

6 Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. `doi:10.1007/978-3-540-24730-2_15`.

7 Byron Cook, Kareem Khazem, Daniel Kroening, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. Model checking boot code from AWS data centers. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 467–486. Springer, 2018.

8 Agostino Cortesi and Matteo Zanioli. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems & Structures*, 37(1):24–42, 2011. `doi:10.1016/j.cl.2010.09.001`.

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, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.

11 Aleksandar S. Dimovski. Program verification using symbolic game semantics. *Theor. Comput. Sci.*, 560:364–379, 2014. `doi:10.1016/j.tcs.2014.01.016`.

12 Quinn Dupont. *Experiments in Algorithmic Governance: A history and ethnography of "The DAO," a failed Decentralized Autonomous Organization*, chapter 8, pages 157–177. Routledge, 2017.

13 William E. Howden. Symbolic testing and the dissect symbolic evaluation system. *Software Engineering, IEEE Transactions on*, SE-3:266–278, August 1977. doi:10.1109/TSE.1977.231144.

14 Dan R. Ghica. Applications of game semantics: From program analysis to hardware synthesis. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*, pages 17–26. IEEE Computer Society, 2009. doi:10.1109/LICS.2009.26.

15 Dan R. Ghica and Guy McCusker. Reasoning about idealized ALGOL using regular languages. In Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors, *Automata, Languages and Programming, 27th International Colloquium, ICALP 2000, Geneva, Switzerland, July 9-15, 2000, Proceedings*, volume 1853 of *Lecture Notes in Computer Science*, pages 103–115. Springer, 2000. doi:10.1007/3-540-45022-X_10.

16 Dan R. Ghica and Nikos Tzevelekos. A system-level game semantics. *Electr. Notes Theor. Comput. Sci.*, 286:191–211, 2012. doi:10.1016/j.entcs.2012.08.013.

17 Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. Foundations and tools for the static analysis of ethereum smart contracts. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 51–78. Springer, 2018. doi:10.1007/978-3-319-96145-3_4.

18 A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 101–112, July 2002. doi:10.1109/LICS.2002.1029820.

19 James C. King. A new approach to program testing. *SIGPLAN Not.*, 10(6):228–233, April 1975. doi:10.1145/390016.808444.

20 Daniel Kroening. The CBMC homepage. http://www.cprover.org/cbmc/, 2017. [Online; accessed 13-Jun-2017].

21 James Laird. A fully abstract trace semantics for general references. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, volume 4596 of *Lecture Notes in Computer Science*, pages 667–679. Springer, 2007. doi:10.1007/978-3-540-73420-8_58.

22 C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe. Reguard: Finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68, May 2018.

23 Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, 2016. ACM. doi:10.1145/2976749.2978309.

24 Andrzej S. Murawski, Steven J. Ramsay, and Nikos Tzevelekos. A contextual equivalence checker for IMJ*. In Bernd Finkbeiner, Geguang Pu, and Lijun Zhang, editors, *Automated Technology for Verification and Analysis*, pages 234–240, Cham, 2015. Springer International Publishing.

25 Andrzej S. Murawski and Nikos Tzevelekos. Higher-order linearisability. In *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany*, pages 34:1–34:18, 2017. doi:10.4230/LIPIcs.CONCUR.2017.34.

26 Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. Soft contract verification for higher-order stateful programs. *PACMPL*, 2(POPL):51:1–51:30, 2018. doi:10.1145/3158139.

27 Phuc C. Nguyen and David Van Horn. Relatively complete counterexamples for higher-order programs. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM*

SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015, pages 446–456. ACM, 2015. doi:10.1145/2737924.2737971.

28 Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pages 653–663, New York, NY, USA, 2018. ACM. doi:10.1145/3274694.3274743.

29 Doron A. Peled. All from one, one for all: on model checking using representatives. In Costas Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1993. doi:10.1007/3-540-56922-7_34.

30 Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*, pages 227–273. Cambridge University Press, 1998.

31 Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, New York, NY, USA, 2013.

32 Grigore Roşu and Traian Şerbănuţă. An overview of the k semantic framework. *The Journal of Logic and Algebraic Programming*, 79:397–434, August 2010. doi:10.1016/j.jlap.2010.03.012.

33 Robert S. Boyer, Bernard Elspas, and Karl Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10:234–245, June 1975. doi:10.1145/390016.808445.

34 Ryosuke Sato and Naoki Kobayashi. Modular verification of higher-order functional programs. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017*, volume 10201 of *Lecture Notes in Computer Science*, pages 831–854. Springer, 2017. doi:10.1007/978-3-662-54434-1_31.

35 Ryosuke Sato, Hiroshi Unno, and Naoki Kobayashi. Towards a scalable software model checker for higher-order programs. In Elvira Albert and Shin-Cheng Mu, editors, *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM 2013, Rome, Italy, January 21-22, 2013*, pages 53–62. ACM, 2013. doi:10.1145/2426890.2426900.

36 Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, pages 419–423, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

37 Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, 1991.

38 Sam Tobin-Hochstadt and David Van Horn. Higher-order symbolic execution via contracts. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 537–554. ACM, 2012. doi:10.1145/2384616.2384655.

39 Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989. doi:10.1007/3-540-53863-1_36.

40 Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 745–761, Berkeley, CA, USA, 2018. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=3277203.3277260.

A Motivating examples

Our file lock example provides a scenario where the library makes it possible for the client to update a file without first reacquiring the lock for it. The library contains an empty private method `updateFile` that simulates file access. The library also provides a public method `openFile`, which locks the file, allows the user to update the file indirectly, and then releases the lock.

```
import userExec :((unit → unit) → unit)
int lock := 0;
private updateFile(x:unit) :(unit) = { () };
public openFile (u:unit) :(unit) = {
  if (!lock) then ()
  else (lock := 1;
      let write = fun(x:unit):(unit) → (assert(!lock);updateFile())
      in userExec(write); lock := 0) };
```

The bug here is that `openFile` creates a `write` method, which it then passes to the client, via `userExec(write)`, to use whenever they want. This provides the client indirect access to the private method `updateFile`, which it can call without first acquiring the lock. Running this example in HOLiK we obtain the following minimal trace:

$$\begin{aligned}&\mathit{call}\langle \mathit{openFile}, ()\rangle \cdot \mathit{call}\langle \mathit{userExec}, m_2\rangle \cdot \mathit{ret}\langle \mathit{userExec}, ()\rangle\\ &\quad\cdot \mathit{ret}\langle \mathit{openFile}, ()\rangle \cdot \mathit{call}\langle m_2, ()\rangle\end{aligned}$$

where m_2 is the method *name* generated by the library and bound to the variable `write`. This example serves as a representative of a class of bugs caused by revealing methods to the environment, a higher-order problem, in this case involving the second-order method `userExec` revealing m_2.

Next, we simulate double deallocation using a global reference `addr` as the memory address. The library defines private methods `alloc` and `free` to simulate allocation and freeing. The empty private method `doSthing` serves as a placeholder for internal computation that does not free memory.

```
1 import getInput :(unit → int)
2 int addr := 0; // 0 means address is free
3 private alloc (u:unit) :(unit) = {
4   if not(!addr) then addr := 1 else () };
5 private free (u:unit) :(unit) = {
6   assert(!addr); addr := 0 };
7 private doSthing (i:int) :(unit) = { () };
8 public run (u:unit) :(unit) = {
9   alloc(); doSthing(getInput ()); free() };
```

The error occurs in line 9, which calls the client method `getInput`. This passes control to the client, who can now call `run` again, thus causing `free` to be called twice. Executing the example on HOLiK, we obtain the following trace:

$$\begin{aligned}&\mathit{call}\langle \mathit{run}, ()\rangle \cdot \mathit{call}\langle \mathit{getInput}, ()\rangle \cdot \mathit{call}\langle \mathit{run}, ()\rangle \cdot \mathit{call}\langle \mathit{getInput}, ()\rangle\\ &\quad\cdot \mathit{ret}\langle \mathit{getInput}, x_1\rangle \cdot \mathit{ret}\langle \mathit{run}, ()\rangle \cdot \mathit{ret}\langle \mathit{getInput}, x_2\rangle\end{aligned}$$

As with the DAO attack, this is a reentrancy bug.

Finally, we have an unsafe implementation of a flat combiner. The library defines two public methods: `enlist`, which allows the client to add procedures to be executed by the library, and `run`, which lets the client run all procedures added so far. The higher-order global reference `list` implements a list of methods.

```
1  private empty(x:int) : (unit) = { () };
2  fun list := empty;
3  int cnt := 0; int running := 0;
4  public enlist(f:(unit → unit)) :(unit) = {
5    if (!running) then ()
6    else
7      cnt := !cnt + 1;
8      (let c = !cnt in let l = !list in
9       list := (fun(z:int):(unit) → if (z == c) then f() else l(z))};
10 public run(x:unit) :(unit) = {
11   running := 1;
12   if (0 < !cnt) then
13     (!list)(!cnt);
14     cnt := !cnt - 1; assert(not (!cnt < 0)); run()
15   else (list := empty; running := 0) };
```

The bug here is also due to a reentrant call in line 13. However, this is a much tougher example as it involves a higher-order reference `list`, a recursive method `run`, and a second-order method `enlist` that reveals client names to the library. With HOLiK, we obtain the following minimal counterexample:

$$
\begin{aligned}
&\mathit{call}\langle \mathit{enlist}, m_1\rangle \cdot \mathit{ret}\langle \mathit{enlist}, ()\rangle \cdot \mathit{call}\langle \mathit{run}, ()\rangle \cdot \mathit{call}\langle m_1, ()\rangle \\
&\qquad \cdot \mathit{call}\langle \mathit{run}, ()\rangle \cdot \mathit{call}\langle m_1, ()\rangle \cdot \mathit{ret}\langle m_1, ()\rangle \cdot \mathit{ret}\langle \mathit{run}, ()\rangle \cdot \mathit{ret}\langle m_1, ()\rangle
\end{aligned}
$$

where m_1 is a client name revealed to the library. In the trace above, `enlist` reveals the method m_1 to the library. This name is then added to the list of procedures to execute. In `run`, the library passes control to the client by calling m_1. At this point, the client is allowed to call `run` again before the list is updated.

B Comparison with Racket Contract Verification

We shall consider the latest version of the tool [26] since it handles state, which we refer to as SCV (Software Contract Verifier). A small benchmark (19 programs) based on HOLiK and SCV benchmarks was used for testing. Programs were manually translated between HOLi and Racket. Care was taken to translate programs whilst maintaining their semantics: contracts enforcing an input-output relation were translated into HOLi using wrapper functions that define the relation through an if statement. In the other direction, since contracts do not directly access references inside a term, stateful functions were translated from HOLi to return any references we wish to reason about.

Table 2 records the comparison. On one hand, HOLiK only found real errors, whereas SCV reported several spurious errors–a third of all errors were spurious. On the other hand, SCV was able to prove total correctness of 3 of the 7 safe files present. SCV also scales much better than HOLiK with respect to program size, which is in exchange of precision. The difference in time for small programs is mainly due to initialisation time. Subtle differences in the nature of each tool can also be observed. e.g., HOLiK reports 1 real error for `ack-simple-e`, whereas SCV reports 2 errors. The difference is because SCV takes into account constraints for integers (e.g. > 0 and $= 0$). More interestingly, for `various`,

Table 2 Comparison of HOLiK (left) and SCV (right). N/A is recorded for `ack` as in our attempts SCV crashed due to unknown reasons.

Program	LoC	Traces	Time (s)	LoC	Errors	Time (s)	False Errors
ack	17	0	6.0	9	N/A	2.4	N/A
ack-simple	13	0	6.5	9	0	2.4	0
ack-simple-e	13	1	6.5	9	2	2.5	0
dao	10	0	5.0	15	1	2.6	1
dao-e	16	1	5.5	15	1	2.7	0
dao-various	85	5	22.5	122	10	3.0	5
dao2-e	85	10	23.5	122	10	2.9	0
escape	9	0	5.0	9	0	2.6	0
escape-e	9	2	5.0	10	1	2.7	0
escape2-e	10	14	6.0	10	1	2.7	0
factorial	10	0	5.0	9	0	2.2	0
mc91	12	0	5.0	9	1	2.2	1
mc91-e	12	1	5.0	8	1	2.4	0
mult	14	0	5.0	11	2	2.7	2
mult-e	14	1	5.0	11	2	2.4	0
succ	7	0	5.0	7	1	2.5	1
succ-e	7	1	5.0	7	1	2.8	0
various	116	19	14.0	108	11	6.2	5
total	459	55	140.5	500	45	49.8	15

HOLiK reports 19 ways to reach assertion violations, whereas SCV reports only 6 real ways to violate contracts. The difference is because HOLiK reports paths through the execution tree that reach errors, whereas SCV reports a set of terms that may violate the contracts. For instance, independently safe methods A and B that may call an unsafe method C would be, from testing, reported as three valid traces ($call\langle A\rangle \cdot call\langle C\rangle$, $call\langle B\rangle \cdot call\langle C\rangle$ and $call\langle C\rangle$) by HOLiK. In contrast, SCV reports a single contract violation blaming C. Finally, `ack` failed to run on SCV due to unknown errors; Racket reported an error internal to the tool. Further testing proved the file is a valid Racket program that can be executed manually.

C ML-like References

HOLi has global higher-order references. These are enough for coding all of our examples and, moreover, allow us to prove completeness (every error has a realising client). We here present a sketch of how games can be extended with (locally created, scope extruding) ML-like references, following e.g. [21, 16]. First, the following extension to types and terms are required.

$$\theta ::= \cdots \mid \texttt{ref}\ \theta \qquad\qquad M ::= \cdots \mid !M \mid \texttt{ref}\ M \mid M = M \qquad\qquad v ::= \cdots \mid r$$

The term $!M$ allows dereferencing terms M which evaluate to references, while $\texttt{ref}\ v$ creates dynamically a fresh name $r \in \texttt{Refs}_\theta$ (if $v : \theta$), and the semantic purpose is to update the store $S \uplus \{r \mapsto v\}$ when evaluating $\texttt{ref}\ v$. Note that this allows us to store references to references, etc. Finally, the construct $M = M$ is for comparing references for name equality.

With terms handling general references concretely and symbolically, we extend game configurations with sets $\mathcal{L}_p, \mathcal{L}_o \subseteq \texttt{Refs}$ that keep track of reference names disclosed by the proponent and opponent respectively. References being passed as values means that the client can update the references belonging to the client, and viceversa. When making a move, for each reference r they own that is passed, the proponent adds r to $\mathcal{L}_p$. Passing of names in a move can be done either by method argument and return value, but also via the common part of the store (i.e. via the references known to both players). Similarly, opponent passes names in their moves, which are added to $\mathcal{L}_o$. Concretely, when the opponent passes control, all references in $\mathcal{L}_p$ are updated with opponent values. Symbolically, the references r are updated with distinct fresh symbolic integers κ if $r \in \texttt{Refs}_{\texttt{Int}}$, distinct fresh method names if $r \in \texttt{Refs}_{\theta_1 \to \theta_2}$, or to arbitrary reference names if $r \in \texttt{Refs}_{\texttt{Refs}_\theta}$.

D Definability

In this section we show that every trace τ in the semantics of a library L has a corresponding good client that realises the same trace in its semantics.

Let L be a library with public names $\mathcal{P}$ and abstract names $\mathcal{A}$. Given a trace τ produced by L, with $\mathcal{P}'$ and $\mathcal{A}'$ respectively the public and abstract names introduced in τ, we set:

$$\begin{aligned}\mathcal{N} &= \mathcal{P} \cup \mathcal{P}' \cup \mathcal{A} \cup \mathcal{A}' \\ \Theta_v &= \{\theta \mid \exists m \in \mathcal{N}.\ m : \theta' \wedge \theta \text{ a syntactic subtype of } \theta'\} \\ \Theta_m &= \{\theta \in \Theta \mid \theta \text{ a method type}\}\end{aligned}$$

Note that the above sets are finite, since $\tau, \mathcal{P}, \mathcal{A}$ are finite. We assume a fixed enumeration of $\mathcal{N} = \{m_1, m_2, \cdots, m_n\}$. Moreover, for each type θ, we let $\mathbf{defval}_\theta$ be a default value, and $\mathbf{diverge}_\theta$ a term that on evaluation diverges by infinite recursion. We then construct a client $C_{\tau,\mathcal{P},\mathcal{A}}$ as in Figure 5.

The code is structured as follows.

1. We start off by defining global references:
   - *cnt* counts the number of P (Library) moves played so far;
   - *meth* stores an index that records the move made by P: if the move was a return then *meth* stores 0; if it was call to m_i then *meth* stores i;
   - each ref_i will store the method $m_i \in \mathcal{P} \cup \mathcal{P}'$, either since the beginning (if $m_i \in \mathcal{P}$), or once P plays it (if $m_i \in \mathcal{P}'$);
   - each val_θ will be used for storing the value played by P in their last move.

   In the latter case above, there is a light abuse of syntax as θ can be a product type, of which HOLi does not have references. But we can in fact simulate references of arbitrary type by several HOLi references.
2. For each $m_i : \theta_1 \to \theta_2 \in \mathcal{A}$, we define a public method m_i that simulates the behaviour of O whenever m_i is called in τ:
   - it starts by increasing *cnt*, as a call to m_i corresponds to a P-move being played;
   - it continues by storing i and x in *meth* and val_{θ_1} respectively;
   - it calls the private method *oracle*, which is tasked with simulating the rest of τ and storing the value that m_i will return in val_{θ_2};
   - it returns the value in val_{θ_2}.
3. For each $m_i : \theta_1 \to \theta_2 \in \mathcal{A}'$ we produce a method just like above, but keep it private (for the time being).
4. The method *oracle* performs the bulk of the computations, by checking that the last move played by P was the expected one and selecting the next move to play (and playing it if is a call).
   - The oracle is called after each P-move is played, so it starts with increasing *cnt*.

```
global cnt := 0
global meth := 0
global ref_i := m_i              # for each m_i ∈ P
global ref_i := defval           # for each m_i ∈ P'
global val_θ := defval           # for each θ ∈ Θ_v
public m_i = λx.                 # for each m_i ∈ A
    cnt++; meth:=i; val_θ1:=x; oracle()
m_i = λx.                        # for each m_i ∈ A'
    cnt++; meth:=i; val_θ1:=x; oracle()
oracle = λ().
    match (!cnt) with       # number of P-moves played so far (max |τ|/2)
    | i →
      # if i > 0 and i-th P-move of τ is cr m_j(v), with m_j : θ1 → θ2, then
      # - if cr = ret then d = 0 and θ = θ2
      # - if cr = call then d = j and θ = θ1
      # diverge if the last P-move played is different from cr m_j(v)
      if not (!meth = d and !val_θ ≜_θ v) then diverge
      else for m_i in fresh(!val_θ) do ref_i := m_i
      # if (i+1)-th O-move of τ is cr' m_k(u), with m_k : θ1 → θ2, then
      # - if cr' = ret then c = 0
      # - if cr' = call then c = k
      if c then let x = (!ref_k)u in     # call m_k(u)
                cnt++; meth:=0; val_θ2:=x; oracle(); !val_θ2
      else val_θ2:=u                     # return u
main = oracle()
```

Figure 5 The client $C_{\tau,\mathcal{P},\mathcal{A}}$.

- It then performs a case analysis on the value of *cnt*, which above we denote collectively by assuming the value is i – this notation hides the fact that we have one case for each of the finitely many values of i.

  For each such i, the oracle first checks if the previous P-move (if there was one), was the expected one. If the move was a call, it checks whether the called method was the expected one (via an appropriate value of d), and also whether the value was the expected one. Value comparisons ($\triangleq_\theta$) only compare the integer components of θ, since we cannot compare method names. If this check is successful, the oracle extracts from u any method names played fresh by P and stores them in the corresponding ref_i.

  Next, the oracle prepares the next move. If, for the given i, the next move is a call, then the oracle issues the call, stores the return value of that call, increases *cnt* and recurs to itself – when the issued call returns, it would be through a P-move. If, on the other hand, the next move is a return, the oracle simply stores the value to be returned in the respective *val* reference – this would allow to the respective m_i to return that value.

5. The **main** method simply calls the oracle.

We can then show the following. For any library L and $(\tau, \rho) \in [\![L]\!]$, C_τ is such that $(\tau, \rho') \in [\![C_\tau]\!]$ for some ρ'.

Strongly Normalizing Higher-Order Relational Queries

Wilmer Ricciotti
Laboratory for Foundations of Computer Science, University of Edinburgh, UK
http://www.wilmer-ricciotti.net
research@wilmer-ricciotti.net

James Cheney
Laboratory for Foundations of Computer Science, University of Edinburgh, UK
https://homepages.inf.ed.ac.uk/jcheney/
jcheney@inf.ed.ac.uk

Abstract

Language-integrated query is a powerful programming construct allowing database queries and ordinary program code to interoperate seamlessly and safely. Language-integrated query techniques rely on classical results about monadic comprehension calculi, including the *conservativity theorem* for nested relational calculus. Conservativity implies that query expressions can freely use nesting and unnesting, yet as long as the query result type is a flat relation, these capabilities do not lead to an increase in expressiveness over flat relational queries. Wong showed how such queries can be translated to SQL via a constructive rewriting algorithm, and Cooper and others advocated *higher-order* nested relational calculi as a basis for language-integrated queries in functional languages such as Links and F#. However there is no published proof of the central *strong normalization* property for higher-order nested relational queries: a previous proof attempt does not deal correctly with rewrite rules that duplicate subterms. This paper fills the gap in the literature, explaining the difficulty with a previous proof attempt, and showing how to extend the ⊤⊤-*lifting* approach of Lindley and Stark to accommodate duplicating rewrites. We also sketch how to extend the proof to a recently-introduced calculus for *heterogeneous* queries mixing set and multiset semantics.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Strong normalization, ⊤⊤-lifting, Nested relational calculus, Language-integrated query

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.28

Funding This work was supported by ERC Consolidator Grant Skye (grant number ERC-682315), a grant from the UK Research Institute in Verified Trustworthy Software Systems (VeTSS), and by an ISCF Metrology Fellowship grant provided by the UK government's Department for Business, Energy and Industrial Strategy (BEIS).

Acknowledgements We are grateful to Philip Wadler, Sam Lindley, and the anonymous reviewers for their comments and suggestions.

1 Introduction

The nested relational calculus [2] provides a principled foundation for integrating database queries into programming languages. Wong's conservativity theorem [23] generalized the classic flat-flat theorem [17] to show that for any nesting depth d, a query expression over flat input tables returning collections of depth at most d can be expressed without constructing intermediate results of nesting depth greater than d. In the special case $d = 1$, this implies the flat-flat theorem, namely that a nested relational query mapping flat tables to flat tables can be expressed equivalently using the flat relational calculus. In addition, Wong's proof technique was constructive, and gave an easily-implemented terminating rewriting algorithm

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 28; pp. 28:1–28:22
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

for normalizing NRC queries to equivalent flat queries; these normal forms correspond closely to idiomatic SQL queries and translating from the former to the latter is straightforward. The basic approach has been extended in a number of directions, including to allow for (nonrecursive) higher-order functions in queries [7], and to allow for translating queries that return nested results to a bounded number of flat relational queries [4].

Normalization-based techniques are used in language-integrated query systems such as Kleisli [24] and Links [8], and can improve both performance and reliability of language-integrated query in F# [3]. However, most work on normalization considers *homogeneous* queries in which there is a single collection type (e.g. homogeneous sets or multisets). Recently, we considered a *heterogeneous* calculus for mixed set and bag queries [20], and conjectured that it too satisfies strong normalization and conservativity theorems. However, in attempting to extend Cooper's proof of normalization we discovered a subtle problem, which makes the original proof incomplete.

Most techniques to prove the strong normalization property for higher-order languages employ logical relations; among these, the Girard-Tait *reducibility* relation is particularly influential: reducibility interprets types as certain sets of strongly normalizing terms enjoying desirable closure properties with respect to reduction, called *candidates of reducibility* [9]. The fundamental theorem then proves that every well-typed term is reducible, hence also strongly normalizing. In its traditional form, reducibility has a limitation that makes it difficult to apply it to certain calculi: the elimination form of every type is expected to be a *neutral* term or, informally, an expression that, when placed in an arbitrary evaluation context, does not interact with it by creating new redexes. However, some calculi possess *commuting conversions*, i.e. reduction rules that apply to nested elimination forms: such rules usually arise when the elimination form for a type (say, pairs) is constructed by means of an auxiliary term of any arbitrary, unrelated type. In this case, we expect nested elimination forms to commute; for example, we could have the following commuting conversion hoisting the elimination of pairs out of case analysis on disjoint unions:

$$\begin{array}{l}\textbf{cases}\ (\textbf{let}\ (a,b)=p\ \textbf{in}\ t)\ \textbf{of}\ \textbf{inl}(x)\Rightarrow u;\textbf{inr}(y)\Rightarrow v\\ \leadsto \textbf{let}\ (a,b)=p\ \textbf{in}\ \textbf{cases}\ t\ \textbf{of}\ \textbf{inl}(x)\Rightarrow u;\textbf{inr}(y)\Rightarrow v\end{array}$$

where p has type $A \times B$, t has type $C + D$, u, v have type U, and the bound variables a, b are chosen fresh for u and v. Since in the presence of commuting conversions elimination forms are not neutral, a straightforward adaptation of reducibility to such languages is precluded.

1.1 $\top\top$-lifting and NRC_λ

Cooper's NRC_λ [6, 7] extends the simply typed lambda calculus with collection types whose elimination form is expressed by *comprehensions* $\bigcup\{M|x \leftarrow N\}$, where M and N have a collection type, and the bound variable x can appear in M:

$$\frac{\Gamma \vdash N : \{S\} \qquad \Gamma, x : S \vdash M : \{T\}}{\Gamma \vdash \bigcup\{M|x \leftarrow N\} : \{T\}}$$

(we use typewriter-style braces $\{\cdot\}$ to indicate collections as expressions or types of NRC_λ). In the rule above, we typecheck a comprehension destructuring collections of type $\{S\}$ to produce new collections in $\{T\}$, where T is an unrelated type: semantically, this corresponds to the union of all the collections $M[V/x]$, such that V is in N. According to the standard approach, we should attempt to define the reducibility predicate for the collection type $\{S\}$ as:

$$\mathsf{Red}_{\{S\}} \triangleq \{N : \forall x, T, \forall M \in \mathsf{Red}_{\{T\}}, \bigcup\{M|x \leftarrow N\} \in \mathsf{Red}_{\{T\}}\}$$

(we use roman-style braces $\{\cdot\}$ to express metalinguistic sets). Of course the definition above is circular, since it uses reducibility over collections to express reducibility over collections; however, this inconvenience could in principle be circumvented by means of impredicativity, replacing $\mathsf{Red}_{\{T\}}$ with a suitable, universally quantified candidate of reducibility (an approach we used in [19] in the context of justification logic). Unfortunately, the arbitrary return type of comprehensions is not the only problem: they are also involved in commuting conversions, such as:

$$\bigcup\{M|x \leftarrow \bigcup\{N|y \leftarrow P\}\} \rightsquigarrow \bigcup\{\bigcup\{M|x \leftarrow N\}|y \leftarrow P\} \qquad (y \notin FV(M))$$

Because of this rule, comprehensions are not neutral terms, thus we cannot use the closure properties of candidates of reducibility (in particular, CR3 [9]) to prove that a collection term is reducible. To address this problem, Lindley and Stark proposed a revised notion of reducibility based on a technique they called $\top\top$-lifting [15]. $\top\top$-lifting, which derives from Pitts's related notion of $\top\top$-closure [18], involves quantification over arbitrarily nested, reducible elimination contexts (*continuations*); the technique is actually composed of two steps: $\top$-lifting, used to define the set $\mathsf{Red}_T^\top$ of reducible continuations for collections of type T in terms of Red_T, and $\top\top$-lifting proper, defining $\mathsf{Red}_{\{T\}} = \mathsf{Red}_T^{\top\top}$ in terms of $\mathsf{Red}_T^\top$. In our setting, if we use $\mathcal{SN}$ to denote the set of strongly normalizing terms, the two operations can be defined as follows:

$$\begin{aligned}\mathsf{Red}_T^\top &\triangleq \{K : \forall M \in \mathsf{Red}_T, K[\{M\}] \in \mathcal{SN}\}\\ \mathsf{Red}_T^{\top\top} &\triangleq \{M : \forall K \in \mathsf{Red}_T^\top, K[M] \in \mathcal{SN}\}\end{aligned}$$

Notice that, in order to avoid a circularity between the definitions of reducible collection continuations and reducible collections, the former are defined by lifting a reducible term M of type T to a singleton collection.

In NRC_λ, besides commuting conversions, we come across an additional problem concerning the property of distributivity of comprehensions over unions, represented by the following reduction rule:

$$\bigcup\{M \cup N|x \leftarrow P\} \rightsquigarrow \bigcup\{M|x \leftarrow P\} \cup \bigcup\{N|x \leftarrow P\}$$

One can immediately see that in $\bigcup\{M \cup N|x \leftarrow \square\}$ the reduction above duplicates the hole, producing a multi-hole context that is not a continuation in the Lindley-Stark sense.

Cooper, in his work, attempted to reconcile continuations with duplicating reductions. While considering extensions to his language, we discovered that his proof of strong normalization presents a nontrivial lacuna which we could only fix by relaxing the definition of continuations to allow multiple holes. This problem affected both the proof of the original result and our attempt to extend it, and has an avalanche effect on definitions and proofs, yielding a more radical revision of the $\top\top$-lifting technique which is the subject of this paper.

The contribution of this paper is to place previous work on higher-order programming for language-integrated query on a solid foundation. As we will show, our approach also extends to proving normalization for a higher-order heterogeneous collection calculus $NRC_\lambda(Set, Bag)$ [20] and we believe our proof technique can be extended further.

1.2 Summary

Section 2 reviews presents NRC_λ and its rewrite system. Section 3 presents the refined approach to reducibility needed to handle rewrite rules with branching continuations. Section 4 presents the proof of strong normalization for NRC_λ. Section 5 outlines the extension to a higher-order calculus $NRC_\lambda(Set, Bag)$ providing heterogeneous set and bag queries. Sections 6 and 7 discuss related work and conclude. Some of the proofs which were omitted from the paper due to space constraints and are detailed in the appendix.

2 Higher-order NRC

NRC_λ, a nested relational calculus with non-recursive higher-order functions, is defined by the following grammar:

$$\begin{array}{llcl}
\textbf{types} & S,T & ::= & A \mid S \to T \mid \langle \overrightarrow{\ell : T} \rangle \mid \{T\} \\
\textbf{terms} & L,M,N & ::= & x \mid c(\overrightarrow{M}) \mid \langle \overrightarrow{\ell = M} \rangle \mid M.\ell \mid \lambda x.M \mid (M\ N) \\
 & & \mid & \emptyset \mid \{M\} \mid M \cup N \mid \bigcup\{M | x \leftarrow N\} \\
 & & \mid & \mathtt{empty}\ M \mid \mathtt{where}\ M\ \mathtt{do}\ N
\end{array}$$

Types include atomic types $A, B, \ldots$ (among which we have Booleans $\mathbf{B}$), record types with named fields $\langle \overrightarrow{\ell : T} \rangle$, collections $\{T\}$; we define *relation types* as those in the form $\{\langle \overrightarrow{\ell : A} \rangle\}$, i.e. collections of tuples of atomic types. Terms include applied constants $c(\overrightarrow{M})$, records with named fields and record projections ($\langle \overrightarrow{\ell = M} \rangle$, $M.\ell$), various collection terms (empty, singleton, union, and comprehension), the emptiness test `empty`, and one-sided conditional expressions for collection types `where` M `do` N. In this definition, x ranges over variable names, c over constants, and ℓ over record field names. We will allow ourselves to use sequences of generators in comprehensions, which are syntactic sugar for nested comprehensions, e.g.:

$$\bigcup\{M | x \leftarrow N, y \leftarrow R\} \triangleq \bigcup\{\bigcup\{M | y \leftarrow R\} | x \leftarrow N\}$$

The typing rules, shown in Figure 1, are largely standard, and we only mention those operators that are specific to our language: constants are typed according to a fixed signature Σ, prescribing the types of the n arguments and of the returned expression to be atomic; `empty` takes a collection and returns a Boolean indicating whether its argument is empty; `where` takes a Boolean condition and a collection and returns the second argument if the Boolean is true, otherwise the empty set. (Conventional two-way conditionals, at any type, are omitted for convenience but can be added without difficulty.)

2.1 Reduction and normalization

NRC_λ is equipped with a rewrite system whose purpose is to convert expressions of flat relation type into a sublanguage isomorphic to a fragment of SQL, even when the original expression contains subterms whose type is not available in SQL, such as nested collections. The rules for this rewrite system are shown in Figure 2.

Reduction on applied constants can happen when all of the arguments are values in normal form, and relies on a fixed semantics $[\![\cdot]\!]$ which assigns to each constant c of signature $\Sigma(c) = \overrightarrow{A_n} \to A'$ a function mapping sequences of values of type $\overrightarrow{A_n}$ to values of type A'. The rules for collections and conditionals are mostly standard. The reduction rule for the emptiness test is triggered when the argument M is not of relation type (but, for instance, of nested collection type) and employs comprehension to generate a (trivial) relation that is empty if and only if M is.

The normal forms of queries under these rewriting rules construct no intermediate nested structures, and are straightforward to translate to equivalent SQL queries. Cooper [7] and Lindley and Cheney [14] give details of such translations. Cheney et al. [3] showed

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Sigma(c) = \overrightarrow{A_n} \to A' \qquad (\Gamma \vdash M_i : A_i)_{i=1,\dots,n}}{\Gamma \vdash c(\overrightarrow{M_n}) : A'}$$

$$\frac{(\Gamma \vdash M_i : T_i)_{i=1,\dots,n}}{\Gamma \vdash \langle \overrightarrow{\ell_n = M_n} \rangle : \langle \overrightarrow{\ell_n : T_n} \rangle} \qquad \frac{\Gamma \vdash M : \langle \overrightarrow{\ell_n : T_n} \rangle \qquad i \in \{1, \dots, n\}}{\Gamma \vdash M.\ell_i : T_i}$$

$$\frac{\Gamma, x : S \vdash M : T}{\Gamma \vdash \lambda x.M : S \to T} \qquad \frac{\Gamma \vdash M : S \to T \qquad \Gamma \vdash N : S}{\Gamma \vdash (M\ N) : T}$$

$$\frac{}{\Gamma \vdash \emptyset : \{T\}} \qquad \frac{\Gamma \vdash M : T}{\Gamma \vdash \{M\} : \{T\}} \qquad \frac{\Gamma \vdash M : \{T\} \qquad \Gamma \vdash N : \{T\}}{\Gamma \vdash M \cup N : \{T\}}$$

$$\frac{\Gamma, x : T \vdash M : \{S\} \qquad \Gamma \vdash N : \{T\}}{\Gamma \vdash \bigcup \{M | x \leftarrow N\} : \{S\}}$$

$$\frac{\Gamma \vdash M : \{T\}}{\Gamma \vdash \texttt{empty}\ M : \mathbf{B}} \qquad \frac{\Gamma \vdash M : \mathbf{B} \qquad \Gamma \vdash N : \{T\}}{\Gamma \vdash \texttt{where}\ M\ \texttt{do}\ N : \{T\}}$$

Figure 1 Type system of NRC_λ.

$$(\lambda x.M)\ N \rightsquigarrow M[N/x] \qquad \langle \dots, \ell = M, \dots \rangle.\ell \rightsquigarrow M \qquad c(\overrightarrow{V}) \rightsquigarrow [\![c]\!](\overrightarrow{V})$$

$$\begin{array}{rcll}
\bigcup\{\emptyset | x \leftarrow M\} \rightsquigarrow \emptyset & & \bigcup\{M | x \leftarrow \emptyset\} \rightsquigarrow \emptyset \qquad \bigcup\{M | x \leftarrow \{N\}\} \rightsquigarrow M[N/x] & \\
\bigcup\{M \cup N | x \leftarrow R\} & \rightsquigarrow & \bigcup\{M | x \leftarrow R\} \cup \bigcup\{N | x \leftarrow R\} & \\
\bigcup\{M | x \leftarrow N \cup R\} & \rightsquigarrow & \bigcup\{M | x \leftarrow N\} \cup \bigcup\{M | x \leftarrow R\} & \\
\bigcup\{M | y \leftarrow \bigcup\{R | x \leftarrow N\}\} & \rightsquigarrow & \bigcup\{M | x \leftarrow N, y \leftarrow R\} & (\text{if } x \notin \mathrm{FV}(M)) \\
\bigcup\{M | x \leftarrow \texttt{where}\ N\ \texttt{do}\ R\} & \rightsquigarrow & \bigcup\{\texttt{where}\ N\ \texttt{do}\ M | x \leftarrow R\} & (\text{if } x \notin \mathrm{FV}(M))
\end{array}$$

$$\begin{array}{rcll}
\texttt{where}\ \mathsf{true}\ \texttt{do}\ M \rightsquigarrow M & & \texttt{where}\ \mathsf{false}\ \texttt{do}\ M \rightsquigarrow \emptyset \qquad \texttt{where}\ M\ \texttt{do}\ \emptyset \rightsquigarrow \emptyset & \\
\texttt{where}\ M\ \texttt{do}\ (N \cup R) & \rightsquigarrow & (\texttt{where}\ M\ \texttt{do}\ N) \cup (\texttt{where}\ M\ \texttt{do}\ R) & \\
\texttt{where}\ M\ \texttt{do}\ \bigcup\{N | x \leftarrow R\} & \rightsquigarrow & \bigcup\{\texttt{where}\ M\ \texttt{do}\ N | x \leftarrow R\} & \\
\texttt{where}\ M\ \texttt{do}\ \texttt{where}\ N\ \texttt{do}\ R & \rightsquigarrow & \texttt{where}\ (M \wedge N)\ \texttt{do}\ R & \\
\texttt{empty}\ M & \rightsquigarrow & \texttt{empty}\ (\bigcup\{\langle\rangle | x \leftarrow M\}) & (\text{if } M \text{ is not relation-typed})
\end{array}$$

Figure 2 Query normalization.

how to improve the performance and reliability of LINQ in F# using normalization and gave many examples showing how higher-order queries support a convenient, compositional language-integrated query programming style.

3 Reducibility with branching continuations

We introduce here the extension of $\top\top$-lifting we use to derive a proof of strong normalization for NRC_λ. The main contribution of this section is a refined definition of continuations with branching structure and multiple holes, as opposed to the linear structure with a single hole used by standard $\top\top$-lifting. In our definition, continuations (as well as the more general notion of context) are particular forms of terms: in this way, the notion of term reduction can be used for continuations as well, without need for auxiliary definitions.

3.1 Contexts and continuations

We start our discussion by introducing *contexts*, or terms with multiple, labelled holes that can be instantiated by plugging other terms (including other contexts) into them.

▶ **Definition 1** (context). *Let us fix a countably infinite set $\mathcal{P}$ of indices: a* context C *is a term that may contain distinguished free variables $[p]$, also called* holes, *where $p \in \mathcal{P}$.*

*Given a finite map from indices to terms $[p_1 \mapsto M_1, \ldots, p_n \mapsto M_n]$ (*context instantiation*), the notation $C[p_1 \mapsto M_1, \ldots, p_n \mapsto M_n]$ (*context application*) denotes the term obtained by simultaneously substituting $M_1, \ldots, M_n$ for the holes $[p_1], \ldots, [p_n]$.*

We will use metavariables η, θ to denote context instantiations.

▶ **Definition 2** (support). *Given a context C, its* support $\mathsf{supp}(C)$ *is defined as the set of the indices p such that $[p]$ occurs in C as a free variable:*

$$\mathsf{supp}(C) \triangleq \{p : [p] \in \mathrm{FV}(C)\}$$

When a term does not contain any $[p]$, we say that it is a *pure* term; when it is important that a term be pure, we will refer to it by using overlined metavariables $\overline{L}, \overline{M}, \overline{N}, \overline{R}, \ldots$. Under the appropriate assumptions, a multiple context instantiation can be decomposed.

▶ **Definition 3.** *An instantiation η is* permutable *iff for all $p \in \mathrm{dom}(\eta)$ we have $\mathrm{FV}(\eta(p)) \cap \mathrm{dom}(\eta) = \emptyset$.*

▶ **Lemma 4.** *Let η be permutable and let us denote by $\eta_{\neg p}$ the restriction of η to indices other than p. Then for all $p \in \mathrm{dom}(\eta)$ we have:*

$$C\eta = C[p \mapsto \eta(p)]\eta_{\neg p} = C\eta_{\neg p}[p \mapsto \eta(p)]$$

We can now define continuations as certain contexts that capture how one or more collections can be used in a program.

▶ **Definition 5** (continuation). *Continuations K are defined as the following subset of contexts:*

$$K, H ::= \quad [p] \mid \overline{M} \mid K \cup K \mid \bigcup\{\overline{M} | x \leftarrow K\} \mid \texttt{where}\ \overline{B}\ \texttt{do}\ K$$

where for all indices p, $[p]$ can occur at most once.

This definition differs from the traditional one in two ways: first, holes are decorated with an index; secondly, and most importantly, the production $K \cup K$ allows continuations to branch and, as a consequence, to use more than one hole. Note that the grammar above is ambiguous, in the sense that certain expressions like $\texttt{where}\ \overline{B}\ \texttt{do}\ \overline{N}$ can be obtained either from the production $\texttt{where}\ \overline{B}\ \texttt{do}\ K$ with $K = \overline{N}$, or as pure terms by means of the production $\overline{M}$: we resolve this ambiguity by parsing these expressions as pure terms whenever possible, and as continuations only when they are proper continuations. An additional complication of NRC_λ when compared to the computational metalanguage for which $\top\top$-lifting was devised lies in the way conditional expressions can reduce when placed in an arbitrary context: continuations in the grammar above are not liberal enough to adapt to such reductions, therefore, like Cooper, we will need an additional definition of *auxiliary* continuations allowing holes to appear in the body of a comprehension (in addition to comprehension generators).

▶ **Definition 6** (auxiliary continuation). *Auxiliary continuations are defined as the following subset of contexts:*

$$Q, O ::= \quad [p] \mid \overline{M} \mid Q \cup Q \mid \bigcup\{Q | x \leftarrow Q\} \mid \texttt{where}\ \overline{B}\ \texttt{do}\ Q$$

where for all indices p, $[p]$ can occur at most once.

We can then see that regular continuations are a special case of auxiliary continuations; however, an auxiliary continuation is allowed to branch not only with unions, but also with comprehensions.[1] We use the following definition of *frames* to represent flat continuations with a distinguished hole.

▶ **Definition 7** (frame). *Frames are defined by the following grammar:*

$$F ::= \quad \bigcup\{Q|x\} \;\mid\; \bigcup\{x \leftarrow Q\} \;\mid\; \texttt{where}\ \overline{B}$$

where for all indices p, $[p]$ can occur at most once.

The operation F^p, lifting a frame to an auxiliary continuation with a distinguished hole $[p]$ is defined by the following rules

$$\begin{aligned} \bigcup\{Q|x\}^p &= \bigcup\{Q|x \leftarrow [p]\} && (p \notin \mathsf{supp}(Q)) \\ \bigcup\{x \leftarrow Q\}^p &= \bigcup\{[p]\,|x \leftarrow Q\} && (p \notin \mathsf{supp}(Q)) \\ (\texttt{where}\ B)^p &= \texttt{where}\ B\ \texttt{do}\ [p] \end{aligned}$$

The composition operation $Q \,\textcircled{\scriptsize p}\, F$ is defined as:

$$Q \,\textcircled{\scriptsize p}\, F = Q[p \mapsto F^p]$$

We generally use frames in conjunction with continuations or auxiliary continuations when we need to partially expose their leaves: for example, if we write $K = K_0 \,\textcircled{\scriptsize p} \bigcup\{\overline{M}|x\}$, we know that instantiating K at index p with (for example) a singleton term will create a redex: $K[p \mapsto \{\overline{L}\}] \rightsquigarrow K_0[p \mapsto \overline{M}\ [\overline{L}/x]]$. We say that such a reduction is a *reduction at the interface* between the continuation and the instantiation.

We introduce two measures $|\cdot|_p$ and $\|\cdot\|_p$ denoting the nesting depth of a hole $[p]$: the two measures differ in the treatment of nesting within the body of a comprehension.

▶ **Definition 8.** *The measures $|Q|_p$ and $\|Q\|_p$ are defined as follows:*

$$|[q]|_p = \|[q]\|_p = \begin{cases} 1 & \textit{if } p = q \\ 0 & \textit{else} \end{cases}$$

$$|\overline{M}|_p = \|\overline{M}\|_p = 0$$

$$|Q_1 \cup Q_2|_p = \max(|Q_1|_p\,, |Q_2|_p) \qquad \|Q_1 \cup Q_2\|_p = \max(\|Q_1\|_p\,, \|Q_2\|_p)$$

$$|\texttt{where}\ B\ Q|_p = |Q|_p + 1 \qquad \|\texttt{where}\ B\ Q\|_p = \|Q\|_p + 1$$

$$|\bigcup\{Q_1|x \mapsto Q_2\}|_p = \begin{cases} |Q_1|_p & \textit{if } p \in \mathsf{supp}(Q_1) \\ |Q_2|_p + 1 & \textit{if } p \in \mathsf{supp}(Q_2) \\ 0 & \textit{else} \end{cases}$$

$$\|\bigcup\{Q_1|x \mapsto Q_2\}\|_p = \begin{cases} \|Q_1\|_p + 1 & \textit{if } p \in \mathsf{supp}(Q_1) \\ \|Q_2\|_p + 1 & \textit{if } p \in \mathsf{supp}(Q_2) \\ 0 & \textit{else} \end{cases}$$

[1] It is worth noting that Cooper's original definition of auxiliary continuation does not use branching comprehension (nor branching unions), but is linear just like the original definition of continuation. The only difference between regular and auxiliary continuations in his work is that the latter allowed nesting not just within comprehension generators, but also within comprehension bodies (in our notation, this would correspond to two separate productions $\bigcup\{\overline{M}|x \leftarrow Q\}$ and $\bigcup\{Q|x \leftarrow \overline{N}\}$).

NRC_λ reduction can be used immediately on contexts (including regular and auxiliary continuations) since these are simply terms with distinguished free variables; we will also abuse notation to allow ourselves to specify reduction on hole instantiations: whenever $\eta(p) \leadsto N$ and $\eta' = \eta_{\neg p}[p \mapsto N]$, we can write $\eta \leadsto \eta'$.

We will denote the set of strongly normalizing terms by $\mathcal{SN}$. For strongly normalizing terms we can introduce the concept of maximal reduction length.

▶ **Definition 9** (maximal reduction length). *Let $M \in \mathcal{SN}$: we define $\nu(M)$ as the maximum length of all reduction sequences starting with M. We also define $\nu(\eta)$ as $\sum_{p \in \mathrm{dom}(\eta)} \nu(\eta(p))$, whenever this value is defined.*

▶ **Lemma 10.** *For all strongly normalizing terms M, if $M \leadsto M'$, then $\nu(M') < \nu(M)$.*

▶ **Lemma 11.** *If $Q \circledcirc_p F \in \mathcal{SN}$, then $\nu(Q) \leq \nu(Q \circledcirc_p F)$.*

3.2 Renaming reduction

Reducing a plain or auxiliary continuation will yield a context that is not necessarily in the same class because certain holes may have been duplicated. For this reason, we introduce a refined notion of renaming reduction which we can use to rename holes in the results so that each of them occurs at most one time.

▶ **Definition 12.** *Given a term M with holes and a finite map $\sigma : \mathcal{P} \to \mathcal{P}$, we write $M\sigma$ for the term obtained from M by replacing each hole $[p]$ such that $\sigma(p)$ is defined with $[\sigma(p)]$.*

Even though finite renaming maps are partial functions, it is convenient to extend them to total functions by taking $\sigma(p) = p$ whenever $p \notin \mathrm{dom}(\sigma)$; we will write id to denote the empty renaming map, whose total extension is the identity function on $\mathcal{P}$.

▶ **Definition 13** (renaming reduction). *M σ-reduces to N (notation: $M \overset{\sigma}{\leadsto} N$) iff $M \leadsto N\sigma$.*

Conveniently, it can be shown that every renaming reduction chain can be simulated by a plain reduction chain of the same length and vice-versa: therefore the notion of strongly normalizing term and the maximal reduction length $\nu(M)$ do not depend on whether we use plain or renaming reduction (this simple result is described in the appendix).

Our goal is to describe the reduction of pure terms expressed in the form of instantiated continuations. One first difficulty we need to overcome is that, as we noted, the sets of continuations (both regular and auxiliary) are not closed under reduction; thankfully, we can prove they are closed under renaming reduction.

▶ **Lemma 14.**

1. *For all continuations K, if $K \leadsto C$, there exist a continuation K' and a finite map σ such that $K \overset{\sigma}{\leadsto} K'$ and $K'\sigma = C$.*
2. *For all auxiliary continuations Q, if $Q \leadsto C$, there exist an auxiliary continuation Q' and a finite map σ such that $Q \overset{\sigma}{\leadsto} Q'$ and $Q'\sigma = C$.*

Proof sketch. For all C we can find C', σ such that $C = C'\sigma$ and all the holes in C' are linear. For case 1, we can show by induction on the derivation of $K \leadsto C'\sigma$ that C' satisfies the grammar for continuations. Case 2 is similar. ◀

Secondly, given a renaming reduction $C \overset{\sigma}{\leadsto} C'$, we want to be able to express the corresponding reduction on $C\eta$: due to the renaming σ, it is not enough to change C to C', but we also need to construct some η' containing precisely those mappings $[q \mapsto M]$ such that, if $\sigma(q) = p$, then $p \in \mathrm{dom}(\eta)$ and $\eta(p) = M$. This construction is expressed by means of the following operation.

▶ **Definition 15.** *For all pure hole instantiations η and renamings σ, we define η^σ as the hole instantiation such that:*

- *if $\sigma(p) \in \mathrm{dom}(\eta)$ then $\eta^\sigma(p) = \eta(\sigma(p))$;*
- *in all other cases, $\eta^\sigma(p) = \eta(\sigma)$.*

The results above allow us to express what happens when a reduction duplicates the holes in a continuation which is then combined with a hole instantiation.

▶ **Lemma 16.** *For all contexts C, renamings σ, and hole instantiations η such that, for all $p \in \mathrm{dom}(\eta)$, $\mathsf{supp}(\eta(p)) \cap \mathrm{dom}(\sigma) = \emptyset$, if $C \overset{\sigma}{\rightsquigarrow} C'$, then $C\eta \overset{\sigma}{\rightsquigarrow} C'\eta^\sigma$.*

Remark. In [6], Cooper attempts to prove strong normalization for NRC_λ using a similar, but weaker result:

> If $K \rightsquigarrow C$, then for all terms M there exists K'_M such that $C[M] = K'_M[M]$ and $K[M] \rightsquigarrow K'_M[M]$.

Since he does not have branching continuations and renaming reductions, whenever a hole is duplicated, e.g.

$$K = \bigcup\{N_1 \cup N_2 | x \leftarrow \Box\} \rightsquigarrow \bigcup\{N_1 | x \leftarrow \Box\} \cup \bigcup\{N_2 | x \leftarrow \Box\} = C$$

he resorts to obtaining a continuation from C simply by filling one of the holes with the instantiation M:

$$K'_M = \bigcup\{N_1 | x \leftarrow M\} \cup \bigcup\{N_2 | x \leftarrow \Box\}$$

Hence, $K'_M[M] = C[M]$. Unfortunately, subsequent proofs rely on the fact that $\nu(K)$ must decrease under reduction: since we have no control over $\nu(M)$, which could potentially be much greater than $\nu(K)$, it may be that $\nu(K'_M) \geq \nu(K)$.

In our setting, by combining Lemmas 14 and 16, we can find a K' which is a proper contractum of K. By Lemma 10, we get $\nu(K') < \nu(K)$, as required by subsequent proofs.

The following result, like many other in the rest of this section, proceeds by well-founded induction; we will use the following notation to represent well-founded relations:

- $<$ stands for the standard less-than relation on $\mathbb{N}$, which is well-founded;
- $\lessdot$ is the lexicographic extension of $<$ to k-tuples in $\mathbb{N}^k$ (for a given k), also well-founded;
- $\prec$ will be used to provide a decreasing metric that depends on the specific proof: such metrics are defined as subsets of $\lessdot$ and are thus well-founded.

▶ **Lemma 17.** *Let Q be an auxiliary continuation, and let η, θ be context instantiations s.t. their union is permutable. If $Q\eta \in \mathcal{SN}$ and $Q\theta \in \mathcal{SN}$, then $Q\eta\theta \in \mathcal{SN}$.*

Proof. We assume that $\mathrm{dom}(\eta) \cup \mathrm{dom}(\theta) \subseteq \mathsf{supp}(Q)$ (otherwise, we can find strictly smaller permutable η', θ' such that $Q\eta\theta = Q\eta'\theta'$, and their domains are subsets of $\mathsf{supp}(Q)$). We show $Q\eta \in \mathcal{SN}$ and $Q\theta \in \mathcal{SN}$ imply $Q \in \mathcal{SN}$, $\eta \in \mathcal{SN}$ and $\theta \in \mathcal{SN}$; thus we can then prove the theorem by well-founded induction on (Q, η, θ) using the following metric:

$$(Q_1, \eta_1, \theta_1) \prec (Q_2, \eta_2, \theta_2) \iff (\nu(Q_1), \|Q_1\|, \nu(\eta_1) + \nu(\theta_1)) \lessdot (\nu(Q_2), \|Q_2\|, \nu(\eta_2) + \nu(\theta_2))$$

We show that all of the possible contracta of $Q\eta\theta$ are s.n. by case analysis on the contraction. The important cases are the following:

- $Q'\eta^\sigma\theta^\sigma$, where $Q \overset{\sigma}{\rightsquigarrow} Q'$: it is easy to see that $\nu(\eta^\sigma)$ and $\nu(\theta^\sigma)$ are defined because $\nu(\eta)$ and $\nu(\theta)$ are; then the thesis follows from the induction hypothesis, knowing that $\nu(Q') < \nu(Q)$ (Lemma 10).
- $Q_0[p \mapsto N]\eta_0\theta$ where $Q = Q_0 \circledcirc_p F$, $\eta = [p \mapsto M]\eta_0$, and $F^p[p \mapsto M] \rightsquigarrow N$ by means of a reduction at the interface. By Lemma 11 we know $\nu(Q_0) \leq \nu(Q)$; we can easily prove $\|Q_0\| < \|Q\|$. We take $\eta' = [p \mapsto N]\eta_0$: since $Q\eta$ reduces to $Q_0\eta'$ and both terms are strongly normalizing, we have that $\nu(\eta')$ is defined. Then we observe $(Q_0, \eta', \theta) \prec (Q, \eta, \theta)$ and obtain the thesis by induction hypothesis. A symmetric case with $p \in \mathrm{dom}(\theta)$ is proved similarly. ◀

▶ **Corollary 18.** *$Q[p \mapsto M]^\sigma \in \mathcal{SN}$ iff for all q s.t. $\sigma(q) = p$, we have $Q[q \mapsto M] \in \mathcal{SN}$.*

The following property (whose proof we detail in the appendix) tells us that instantiating a continuation never shortens the maximal reduction chain.

▶ **Lemma 19.** *If $Q\eta \in \mathcal{SN}$, then $Q \in \mathcal{SN}$ and $\nu(Q) \leq \nu(Q\eta)$.*

3.3 Candidates of reducibility

We here define the notion of *candidates of reducibility*: sets of strongly normalizing terms enjoying certain closure properties that can be used to overapproximate the sets of terms of a certain type. Our version of candidates for *NRC*$_\lambda$ is a straightforward adaptation of the standard definition given by Girard and like that one is based on a notion of *neutral terms*, i.e. those terms that, when placed in an arbitrary context, do not create additional redexes. The set of neutral terms is denoted by $\mathcal{NT}$. Let us introduce the following notation for Girard's CRx properties of sets [9]:

- $\mathrm{CR1}(\mathcal{C}) \triangleq \mathcal{C} \subseteq \mathcal{SN}$
- $\mathrm{CR2}(\mathcal{C}) \triangleq \forall M \in \mathcal{C}, \forall M'.M \rightsquigarrow M' \implies M' \in \mathcal{C}$
- $\mathrm{CR3}(\mathcal{C}) \triangleq \forall M \in \mathcal{NT}.(\forall M'.M \rightsquigarrow M' \implies M' \in \mathcal{C}) \implies M \in \mathcal{C}$

The set $\mathcal{CR}$ of the candidates of reducibility is then defined as the collection of those sets of terms which satisfy all the CRx properties. Some standard results include the non-emptiness of candidates (in particular, all free variables are in every candidate) and that $\mathcal{SN} \in \mathcal{CR}$.

3.4 Reducibility sets

In this section we introduce *reducibility sets*, which are sets of terms that we will use to provide an interpretation of the types of *NRC*$_\lambda$; we will then prove that reducibility sets are candidates of reducibility, hence they only contain strongly normalizing terms. The following notation will be useful as a shorthand for certain operations on sets of terms that are used to define reducibility sets:

- $\mathcal{C} \to \mathcal{D} \triangleq \{M : \forall N \in \mathcal{C}, (M\ N) \in \mathcal{D}\}$
- $\langle \overrightarrow{\ell_k : \mathcal{C}_k} \rangle \triangleq \{M : \forall i = 1, \ldots, k, M.\ell_i \in \mathcal{C}_i\}$
- $(p : \mathcal{C})^\top \triangleq \{K : \forall M \in \mathcal{C}.K[p \mapsto \{M\}] \in \mathcal{SN}\}$
- $\mathcal{C}^{\top\top} \triangleq \{M : \forall p, \forall K \in (p : \mathcal{C})^\top, K[p \mapsto M] \in \mathcal{SN}\}$

The sets $(p : \mathcal{C})^\top$ and $\mathcal{C}^{\top\top}$ are called the $\top$-*lifting* and $\top\top$-*lifting* of $\mathcal{C}$. These definitions refine the ones used in the literature by using indices: $\top$-lifting is defined with respect to a given index p, while the definition of $\top\top$-lifting uses any index.

▶ **Definition 20** (reducibility). *For all types T, the set Red_T of reducible terms of type T is defined by recursion on T by means of the rules:*

$$\begin{array}{rclrcl} \mathsf{Red}_A & \triangleq & \mathcal{SN} & \mathsf{Red}_{S \to T} & \triangleq & \mathsf{Red}_S \to \mathsf{Red}_T \\ \mathsf{Red}_{\overrightarrow{\langle \ell_k : T_k \rangle}} & \triangleq & \overrightarrow{\langle \ell_k : \mathsf{Red}_{T_k} \rangle} & \mathsf{Red}_{\{T\}} & \triangleq & \mathsf{Red}_T^{\top\top} \end{array}$$

Let us use metavariables $\Theta, \Theta', \ldots$ to denote finite maps from indices to sets of terms in the form $(p_1 : \mathcal{C}_1, \ldots, p_k : \mathcal{C}_k)$. We extend the notion of $\top$-lifting to such maps by taking the intersection of all the $(p_i : \mathcal{C}_i)^\top$. This notation is useful to track Θ under renaming reduction.

▶ **Definition 21.** $\Theta^\top \triangleq \bigcap_{p \in \mathrm{dom}(\Theta)} (p : \Theta(p))^\top$.

▶ **Definition 22.** *Let Θ be a finite map from indices to sets of terms and σ a renaming: then we define Θ^σ as the finite map $\Theta^\sigma(p) = \Theta(\sigma(p))$, defined for all p such that $\sigma(p) \in \mathrm{dom}(\Theta)$.*

We now proceed with the proof that all the sets Red_T are candidates of reducibility: we will only focus on collections since for the other types the result is standard. The proofs of CR1 and CR2 do not differ much from the standard $\top\top$-lifting technique.

▶ **Lemma 23** (CR1 for continuations). *For all p and all non-empty $\mathcal{C}$, $(p : \mathcal{C})^\top \subseteq \mathcal{SN}$.*

▶ **Lemma 24** (CR1 for collections). *If* $\mathrm{CR1}(\mathcal{C})$, *then* $\mathrm{CR1}(\mathcal{C}^{\top\top})$.

▶ **Lemma 25** (CR2 for collections). *If $M \in \mathcal{C}^{\top\top}$ and $M \leadsto M'$, then $M' \in \mathcal{C}^{\top\top}$.*

In order to prove CR2 for all types (and particularly for collections), we do not need to establish an analogous property on continuations; however such a property is still useful for subsequent results (particularly CR3): its statement must, of course, consider that reduction may duplicate (or indeed delete) holes, and thus employs renaming reduction.

▶ **Lemma 26** (CR2 for continuations). *If $K \in \Theta^\top$ and $K \overset{\sigma}{\leadsto} K'$, then $K' \in (\Theta^\sigma)^\top$.*

The lemma above could have some unpleasant consequences for our proof: since reducing an applied continuation of the form $K[p \mapsto \overline{M}]$ can lead to the duplication of $\overline{M}$, every proof of a statement about the strong normalizability of such an expression that proceeds by induction on its reduction chains would need to be generalized to n-ary instantiations of n-ary continuations! Fortunately, instantiations to pure terms are always permutable, therefore we can simply consider each hole separately, as stated in the following lemma.

▶ **Lemma 27.** *$K \in (\Theta^\sigma)^\top$ if, and only if, for all $q \in \mathrm{dom}(\Theta^\sigma)$, we have $K \in (q : \Theta(\sigma(q)))^\top$. In particular, $K \in ((p : \mathcal{C})^\sigma)^\top$ if, and only if, for all q s.t. $\sigma(q) = p$, we have $K \in (q : \mathcal{C})^\top$.*

This is everything we need to prove CR3.

▶ **Lemma 28** (CR3 for collections). *Let $\mathcal{C} \in \mathcal{CR}$, and M a neutral term such that for all reductions $M \leadsto M'$ we have $M' \in \mathcal{C}^{\top\top}$. Then $M \in \mathcal{C}^{\top\top}$.*

Proof. By definition, we need to prove $K[p \mapsto M] \in \mathcal{SN}$ whenever $K \in (p : \mathcal{C})^\top$ for some index p. By Lemma 23, knowing that $\mathcal{C}$, being a candidate, is non-empty, we have $K \in \mathcal{SN}$. We can thus proceed by well-founded induction on $\nu(K)$ to prove the strengthened statement: for all indices q, if $K \in (q : \mathcal{C})^\top$, then $K[q \mapsto M] \in \mathcal{SN}$. Equivalently, we prove that all the contracta of $K[q \mapsto M]$ are s.n. by cases on the possible contracta:

- $K'[q \mapsto M]^\sigma$ (where $K \overset{\sigma}{\leadsto} K'$): to prove this term is s.n., by Lemma 18, we need to show $K'[q' \mapsto M] \in \mathcal{SN}$ whenever $\sigma(q') = q$; by Lemmas 26 and 27, we know $K' \in (q' : \mathcal{C})^\top$, and naturally $\nu(K') < \nu(K)$ (Lemma 10), thus the thesis follows by the IH.
- $K[p \mapsto M']$ (where $M \leadsto M'$): this is s.n. because $M' \in \mathcal{C}^{\top\top}$ by hypothesis.
- Since M is neutral, there are no reductions at the interface. ◀

▶ **Theorem 29.** *For all types T, $\mathsf{Red}_T \in \mathcal{CR}$.*

Proof. Standard by induction on T. For $T = \{T'\}$, we use Lemmas 24, 25, and 28. ◀

4 Strong normalization

Having proved that the reducibility sets of all types are candidates of reducibility, in order to obtain strong normalization we only need to know that every well-typed term is in the reducibility set corresponding to its type: this proof is by structural induction on the derivation of the typing judgment. Reducibility of singletons is trivial by definition, while that of empty collections is proved in the same style as [6], with the obvious adaptations.

▶ **Lemma 30** (reducibility for singletons). *For all $\mathcal{C}$, if $M \in \mathcal{C}$, then $\{M\} \in \mathcal{C}^{\top\top}$.*

▶ **Lemma 31** (reducibility for $\emptyset$). *For all $\mathcal{C}$, $\emptyset \in \mathcal{C}^{\top\top}$.*

As for unions, we will prove a more general statement on auxiliary continuations.

▶ **Lemma 32.**
For all auxiliary continuations Q, O_1, O_2 with pairwise disjoint supports, if $Q[p \mapsto O_1] \in \mathcal{SN}$ and $Q[p \mapsto O_2] \in \mathcal{SN}$, then $Q[p \mapsto O_1 \cup O_2] \in \mathcal{SN}$.

The proof of the lemma above follows the same style as [6]; however since our definition of auxiliary continuations is more general, the theorem statement mentions O_1, O_2 rather than pure terms: the hypothesis on the supports of the continuations being disjoint is required by this generalization.

▶ **Corollary 33** (reducibility for unions). *If $M \in \mathcal{C}^{\top\top}$ and $N \in \mathcal{C}^{\top\top}$, then $M \cup N \in \mathcal{C}^{\top\top}$.*

Like in proofs based on standard $\top\top$-lifting, the most challenging cases are those dealing with commuting conversions – in our case, comprehensions and conditionals.

▶ **Lemma 34.** *Let K, $\overline{L}$, $\overline{N}$ be such that $K[p \mapsto \overline{N}\,[\overline{L}/x]] \in \mathcal{SN}$ and $\overline{L} \in \mathcal{SN}$. Then $K[p \mapsto \bigcup\{\overline{N} | x \leftarrow \{\overline{L}\}\}] \in \mathcal{SN}$.*

Proof. In this proof, we assume the names of bound variables are chosen so as to avoid duplicates, and are distinct from the free variables. We proceed by well-founded induction on $(K, p, \overline{N}, \overline{L})$ using the following metric:

$$\begin{aligned}
&(K_1, p_1, \overline{N_1}, \overline{L_1}) \prec (K_2, p_2, \overline{N_2}, \overline{L_2}) \\
&\iff (\nu(K_1[p_1 \mapsto \overline{N_1}\,[\overline{L_1}/x]]) + \nu(\overline{L_1}), \|K_1\|_{p_1}, \mathsf{size}(\overline{N_1})) \\
&\qquad \lessdot (\nu(K_2[p_2 \mapsto \overline{N_2}\,[\overline{L_2}/x]]) + \nu(\overline{L_2}), \|K_2\|_{p_2}, \mathsf{size}(\overline{N_2}))
\end{aligned}$$

Now we show that every contractum must be a strongly normalizing:

- $K[p \mapsto \overline{N}\,[\overline{L}/x]]$: this term is s.n. by hypothesis.

- $K'[p \mapsto \bigcup\{N|x \leftarrow \{\overline{L}\}\}]^\sigma$, where $K \overset{\sigma}{\leadsto} K'$. Lemma 10 allows us to prove $\nu(K'[p \mapsto \overline{N}\,[\overline{L}/x]]^\sigma) < \nu(K[p \mapsto \overline{N}\,[\overline{L}/x]])$ (since the former is a contractum of the latter), which implies $\nu(K'[q \mapsto \overline{N}\,[\overline{L}/x]]) \leq \nu(K'[p \mapsto \overline{N}\,[\overline{L}/x]]^\sigma) < \nu(K[p \mapsto \overline{N}\,[\overline{L}/x]])$ for all q s.t. $\sigma(q) = p$ by means of Lemma 19 (because $[q \mapsto \overline{N}\,[\overline{L}/x]]$ is a subapplication of $[p \mapsto \overline{N}\,[\overline{L}/x]]^\sigma$); then we can apply the IH to obtain, for all q s.t. $\sigma(q) = p$, $K'[q \mapsto \bigcup\{\overline{N}|x \leftarrow \{\overline{L}\}\}] \in \mathcal{SN}$; by Lemma 18, this implies the thesis.
- $K[p \mapsto \emptyset]$ (when $N = \emptyset$): this is equal to $K[p \mapsto \emptyset\,[\overline{L}/x]]$, which is s.n. by hypothesis.
- $K[p \mapsto \bigcup\{\overline{N_1}|x \leftarrow \{\overline{L}\}\} \cup \bigcup\{\overline{N_2}|x \leftarrow \{\overline{L}\}\}]$ (when $\overline{N} = \overline{N_1} \cup \overline{N_2}$); by IH (since $\mathrm{size}(\overline{N_i}) < \mathrm{size}(\overline{N_1} \cup \overline{N_2})$, and all other metrics do not increase) we prove $K[p \mapsto \bigcup\{\overline{N_i}|x \leftarrow \{\overline{L}\}\}] \in \mathcal{SN}$ (for $i = 1, 2$), and consequently obtain the thesis by Lemma 32.
- $K_0[p \mapsto \bigcup\{\bigcup\{\overline{M}|y \leftarrow \overline{N}\}|x \leftarrow \{\overline{L}\}\}]$, where $K = K_0 \textcircled{\scriptsize p} \bigcup\{\overline{M}|y\}$; since we know, by the hypothesis on the choice of bound variables, that $x \notin \mathrm{FV}(\overline{M})$, we note that $K_0[p \mapsto \bigcup\{\overline{M}|y \leftarrow \overline{N}\}\,[\overline{L}/x]] = K[p \mapsto \overline{N}\,[\overline{L}/x]]$; furthermore, we know $\|K_0\|_p < \|K\|_p$; then we can apply the IH to obtain the thesis.
- $K_0[p \mapsto \bigcup\{\texttt{where}\ \overline{B}\ \texttt{do}\ \overline{N}|x \leftarrow \{\overline{L}\}\}]$ (when $K = K_0 \textcircled{\scriptsize p}\ \texttt{where}\ \overline{B}$): since we know, from the hypothesis on the choice of bound variables, that $x \notin \mathrm{FV}(B)$, we note that $K_0[p \mapsto (\texttt{where}\ \overline{B}\ \texttt{do}\ \overline{N})\,[\overline{L}/x]] = K[p \mapsto \overline{N}\,[\overline{L}/x]]$; furthermore, we know $\|K_0\|_p < \|K\|_p$; then we can apply the IH to obtain the thesis.
- reductions within N or L follow from the IH by reducing the induction metric. ◀

▶ **Lemma 35** (reducibility for comprehensions). *Assume* $\mathrm{CR1}(\mathcal{C})$, $\mathrm{CR1}(\mathcal{D})$, $\overline{M} \in \mathcal{C}^{\top\top}$ *and for all* $\overline{L} \in \mathcal{C}$, $\overline{N}\,[\overline{L}/x] \in \mathcal{D}^{\top\top}$. *Then* $\bigcup\{\overline{N}|x \leftarrow \overline{M}\} \in \mathcal{D}^{\top\top}$.

Proof. We assume p, $K \in (p : \mathcal{D})^\top$ and prove $K[p \mapsto \bigcup\{\overline{N}|x \leftarrow \overline{M}\}] \in \mathcal{SN}$. We start by showing that $K' = K \textcircled{\scriptsize p} \bigcup\{\overline{N}|x\} \in (p : \mathcal{C})^\top$, or equivalently that for all $\overline{L} \in \mathcal{C}$, $K'[p \mapsto \{\overline{L}\}] = K[p \mapsto \bigcup\{\overline{N}|x \leftarrow \{\overline{L}\}\}] \in \mathcal{SN}$: since $\mathrm{CR1}(\mathcal{C})$, we know $\overline{L} \in \mathcal{SN}$, and since $\overline{N}\,[\overline{L}/x] \in \mathcal{D}^{\top\top}$, $K[p \mapsto \overline{N}\,[\overline{L}/x]] \in \mathcal{SN}$; then we can apply Lemma 34 to obtain $K'[p \mapsto \{\overline{L}\}] \in \mathcal{SN}$ and consequently $K' \in (p : \mathcal{C})^\top$. But then, since $\overline{M} \in \mathcal{C}^{\top\top}$, we have $K'[p \mapsto \overline{M}] = K[p \mapsto \bigcup\{\overline{N}|x \leftarrow \overline{M}\}] \in \mathcal{SN}$, which is what we needed to prove. ◀

Reducibility for conditionals is proved in a similar manner. However, to consider all the conversions commuting with `where`, we need to use the more general auxiliary continuations.

▶ **Lemma 36.** *Let* Q, $\overline{B}$, O *such that* $Q[p \mapsto O] \in \mathcal{SN}$, $\overline{B} \in \mathcal{SN}$, *and* $\mathsf{supp}(Q) \cap \mathsf{supp}(O) = \emptyset$. *Then* $Q[p \mapsto \texttt{where}\ \overline{B}\ \texttt{do}\ O] \in \mathcal{SN}$.

Proof sketch. We proceed by well-founded induction on (Q, B, O, p) using the following metric:

$$\begin{aligned}&(Q_1, \overline{B_1}, O_1, p_1) \prec (Q_2, \overline{B_2}, O_2, p_2) \iff \\ &(\nu(Q_1[p_1 \mapsto O_1]) + \nu(\overline{B_1}), |Q_1|_{p_1}, \mathrm{size}(O_1)) \lessdot (\nu(Q_2[p_2 \mapsto O_2]) + \nu(\overline{B_2}), |Q_2|_{p_2}, \mathrm{size}(O_2))\end{aligned}$$

We show every contractum must be a strongly normalizing term; we apply the IH to new auxiliary continuations obtained by placing pieces of O into Q or vice-versa: the hypothesis on the supports of Q and O is used to ensure that the new continuations are well-formed. The use of $|\cdot|_p$ rather than $\|\cdot\|_p$ is needed to ensure that contractions in the form $Q[p \mapsto \texttt{where}\ \overline{B}\ \texttt{do}\ \bigcup\{O_1|x \leftarrow O_2\}] \leadsto (Q \textcircled{\scriptsize p} \bigcup\{x \leftarrow O_2\})[p \mapsto \texttt{where}\ \overline{B}\ \texttt{do}\ O_1]$ do not increase the metric. ◀

▶ **Corollary 37** (reducibility for conditionals).
If $\overline{B} \in \mathcal{SN}$ *and* $\overline{N} \in \mathsf{Red}_{\{T\}}$, *then* $\texttt{where}\ \overline{B}\ \texttt{do}\ \overline{N} \in \mathsf{Red}_{\{T\}}$.

Finally, reducibility for the emptiness test is proved in the same style as [6].

▶ **Lemma 38.** *For all M and T such that $\Gamma \vdash M : \{T\}$ and $M \in \mathsf{Red}_T^{\top\top}$, we have* $\texttt{empty}(M) \in \mathcal{SN}$*.*

4.1 Main theorem

Before stating and proving the main theorem, we introduce some auxiliary notation.

▶ **Definition 39.**

1. *A substitution ρ satisfies Γ (notation: $\rho \vDash \Gamma$) iff, for all $x \in \mathrm{dom}(\Gamma)$, $\rho(x) \in \mathsf{Red}_{\Gamma(x)}$.*
2. *A substitution ρ satisfies M with type T (notation: $\rho \vDash M : T$) iff $M\rho \in \mathsf{Red}_T$.*

As usual, the main result is obtained as a corollary of a stronger theorem generalized to substitutions into open terms, by using the identity substitution id_Γ.

▶ **Lemma 40.** *For all Γ, we have $\mathsf{id}_\Gamma \vDash \Gamma$.*

▶ **Theorem 41.** *If $\Gamma \vdash M : T$, then for all ρ such that $\rho \vDash \Gamma$, we have $\rho \vDash M : T$*

Proof. By induction on the derivation of $\Gamma \vdash M : T$. When M is empty, a singleton, a union, an emptiness test, or a conditional, we use Lemmas 31, 30, 33, 38, and 37. For comprehensions such that $\Gamma \vdash \bigcup\{M_1 | x \leftarrow M_2\} : \{T\}$, we know by IH that $\rho \vDash M_2 : \{S\}$ and for all $\rho' \vDash \Gamma, x : S$ we have $\rho' \vDash M_1 : \{T\}$: we prove that for all $L \in \mathsf{Red}_S$, $\rho[L/x] \vDash \Gamma, x : S$, hence $\rho[L/x] \vdash M_1 : \{T\}$; then we obtain $\rho \vDash \bigcup\{M_1 | x \leftarrow M_2\} : \{T\}$ by Lemma 35. Non-collection cases are standard. ◀

▶ **Corollary 42.** *If $\Gamma \vdash M : T$, then $M \in \mathcal{SN}$.*

5 Heterogeneous Collections

In a short paper [20], we introduced a generalization of *NRC* called *NRC*(*Set*, *Bag*), which contains both set-valued and bag-valued collections (with distinct types denoted by $\{T\}$ and $\Lbag T \Rbag$), along with mapping from bags to sets (deduplication δ) and from sets to bags (promotion ι). We conjectured that this language also satisfies a normalization property. Here, we prove this claim, even extending *NRC*(*Set*, *Bag*) to a richer language $NRC_\lambda(Set, Bag)$ with higher-order (nonrecursive) functions.

$$\begin{array}{rcl} L, M, N & ::= & \ldots \mid \mho \mid \Lbag M \Rbag \mid M \uplus N \mid \biguplus\Lbag M | x \leftarrow N \Rbag \\ & \mid & \texttt{where}_{\texttt{bag}}\ M\ \texttt{do}\ N \mid \texttt{empty}_{\texttt{bag}}\ M \mid \delta M \mid \iota M \end{array}$$

The notations $\mho$, $\Lbag M \Rbag$, $M \uplus N$, $\biguplus\Lbag M | x \leftarrow N \Rbag$ denote empty and singleton bags, bag disjoint union and bag comprehension; the language also includes conditionals and emptiness tests on bags. We omit the typing rules, and observe that the reduction rules involving bag operations correspond to those for set operations, and additionally include the following:

$$\begin{array}{c} \delta\mho \rightsquigarrow \emptyset \qquad \delta\Lbag M \Rbag \rightsquigarrow \{M\} \qquad \delta(M \uplus N) \rightsquigarrow \delta M \cup \delta N \qquad \delta\iota M \rightsquigarrow M \\ \delta \biguplus\Lbag M | x \leftarrow N \Rbag \rightsquigarrow \bigcup\{\delta M | x \leftarrow \delta N\} \qquad \delta(\texttt{where}_{\texttt{bag}}\ M\ \texttt{do}\ N) \rightsquigarrow \texttt{where}\ M\ \texttt{do}\ \delta N \\ \iota\emptyset \rightsquigarrow \mho \qquad \iota\{M\} \rightsquigarrow \Lbag M \Rbag \qquad \iota(\texttt{where}\ M\ \texttt{do}\ N) \rightsquigarrow \texttt{where}_{\texttt{bag}}\ M\ \texttt{do}\ \iota N \end{array}$$

SN for $NRC_\lambda(Set, Bag)$ is proved by first translating the language to a version of NRC_λ retaining the operations δ and ι that we call $NRC_{\lambda\delta\iota}$, by means of a forgetful translation $\lfloor \cdot \rfloor$ mapping empty bags, bag unions and bag comprehensions to the corresponding set constructs. We prove that every contraction in $NRC_\lambda(Set, Bag)$ is translated to a contraction in $NRC_{\lambda\delta\iota}$, and thus obtain SN for $NRC_\lambda(Set, Bag)$ as a corollary of SN for $NRC_{\lambda\delta\iota}$.

▶ **Theorem 43.** *If $\Gamma \vdash M : T$ in $NRC_\lambda(Set, Bag)$, then $\lfloor \Gamma \rfloor \vdash \lfloor M \rfloor : \lfloor T \rfloor$ in $NRC_{\lambda\delta\iota}$.*

▶ **Lemma 44.** *For all terms M of $NRC_\lambda(Set, Bag)$, if $M \rightsquigarrow M'$, we have $\lfloor M \rfloor \rightsquigarrow \lfloor M' \rfloor$ in $NRC_{\lambda\delta\iota}$. Consequently, if $\lfloor M' \rfloor \in \mathcal{SN}$ in $NRC_{\lambda\delta\iota}$, then $M' \in \mathcal{SN}$ in $NRC_\lambda(Set, Bag)$.*

▶ **Theorem 45.** *If $\Gamma \vdash M : T$ in $NRC_{\lambda\delta\iota}$, then $M \in \mathcal{SN}$ in $NRC_{\lambda\delta\iota}$.*

▶ **Corollary 46.** *If $\Gamma \vdash M : T$ in $NRC_\lambda(Set, Bag)$, then $M \in \mathcal{SN}$ in $NRC_\lambda(Set, Bag)$.*

6 Related Work

This paper builds on a long line of research on normalization of comprehension queries, a model of query languages popularized over 25 years ago by Buneman et al. [2] and inspired by Trinder and Wadler's work on comprehensions [21, 22]. Wong [23] proved conservativity via a strongly normalizing rewrite system, which was used in Kleisli [24], a functional query system, in which flat query expressions were normalized to SQL. Libkin and Wong [12, 13] investigated conservativity in the presence of aggregates, internal generic functions, and bag operations, and demonstrated that bag operations can be expressed using nested comprehensions. However, their normalization results studied bag queries by translating to relational queries with aggregation, and did not consider higher-order queries, so they do not imply the normalization results for $NRC_\lambda(Set, Bag)$ given here.

Cooper [7] first investigated query normalization (and hence conservativity) in the presence of higher-order functions. He gave a rewrite system showing how to normalize homogeneous (that is, pure set or pure bag) queries to eliminate intermediate occurrences of nesting or of function types. However, although Cooper claimed a proof (based on $\top\top$-lifting [15]) and provided proof details in his PhD thesis [6], there unfortunately turned out to be a nontrivial lacuna in that proof, and this paper therefore (in our opinion) contains the first *complete* proof of normalization for higher-order queries, even for the homogeneous case.

Since the fundamental work of Wong and others on the Kleisli system, language-integrated query has gradually made its way into other systems, most notably Microsoft's .NET framework languages C# and F# [16], and the Web programming language Links [8]. Cheney et al. [3] formally investigated the F# approach to language-integrated query and showed that normalization results due to Wong and Cooper could be adapted to improve it further; however, their work considered only homogeneous collections. In subsequent work, Cheney et al. [4] showed how use normalization to perform *query shredding* for multiset queries, in which a query returning a type with n nested collections can be implemented by combining the results of n flat queries; this has been implemented in Links [8].

Several recent efforts to formalize and reason about the semantics of SQL are complementary to our work. Guagliardo and Libkin [10] presented a semantics for SQL's actual behaviour in the presence of set and multiset operators (including bag intersection and difference) as well as incomplete information (nulls), and related the expressiveness of this fragment of SQL with that of an algebra over bags with nulls. Chu et al. [5] presented a formalised semantics for reasoning about SQL (including set and bag semantics as well as aggregation/grouping, but excluding nulls) using nested relational queries in Coq, while Benzaken and Contejean [1] presented a semantics including all of these SQL features (set, multiset, aggregation/grouping, nulls), and formalized the semantics in Coq. Kiselyov et al. [11] has proposed language-integrated query techniques that handle sorting operations (SQL's `ORDER BY`).

However, the above work on semantics has not considered query normalization, and to the best of our knowledge normalization results for query languages with more than one collection type were previously unknown even in the first-order case. We are interested in extending our results for mixed set and bag semantics to handle nulls, grouping/aggregation, and sorting, thus extending higher-order language integrated query to cover all of the most widely-used SQL features. Normalization of higher-order queries in the presence of all of these features simultaneously remains an open problem, which we plan to consider next. In addition, fully formalizing such normalization proofs also appears to be a nontrivial challenge.

7 Conclusions

Integrating database queries into programming languages has many benefits, such as type safety and avoidance of common SQL injection attacks, but also imposes limitations that prevent programmers from constructing queries dynamically as they could by concatenating SQL strings unsafely. Previous work has demonstrated that many useful dynamic queries can be constructed safely using *higher-order functions* inside language-integrated queries; provided such functions are not recursive, it was believed, query expressions can be normalized. Moreover, while it is common in practice for language-integrated query systems to provide support for SQL features such as mixed set and bag operators, it is not well understood in theory how to normalize these queries in the presence of higher-order functions. Previous work on higher-order query normalization has considered only homogeneous (that is, pure set or pure bag) queries, and in the process of attempting to generalize this work to a heterogeneous setting, we discovered a nontrivial gap in the previous proof of strong normalization. We therefore prove strong normalization for both homogeneous and heterogeneous queries for the first time.

As next steps, we intend to extend the Links implementation of language-integrated query with heterogeneous queries and normalization, and to investigate (higher-order) query normalization and conservativity for the remaining common SQL features, such as nulls, grouping/aggregation, and ordering.

References

1 Véronique Benzaken and Evelyne Contejean. A Coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra. In *CPP 2019*, pages 249–261, 2019. doi:10.1145/3293880.3294107.

2 Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1), 1995. doi:10.1016/0304-3975(95)00024-Q.

3 James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. In *ICFP*, 2013. doi:10.1145/2500365.2500586.

4 James Cheney, Sam Lindley, and Philip Wadler. Query shredding: efficient relational evaluation of queries over nested multisets. In *SIGMOD*, pages 1027–1038. ACM, 2014. doi:10.1145/2588555.2612186.

5 Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. HoTTSQL: Proving query rewrites with univalent SQL semantics. In *PLDI*, pages 510–524. ACM, 2017. doi:10.1145/3062341.3062348.

6 Ezra Cooper. *Programming language features for web application development*. PhD thesis, University of Edinburgh, 2009.

7 Ezra Cooper. The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed. In *DBPL*, 2009. doi:10.1007/978-3-642-03793-1_3.

8 Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: web programming without tiers. In *FMCO*, 2007. doi:10.1007/978-3-540-74792-5_12.

9 Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.

10 Paolo Guagliardo and Leonid Libkin. A formal semantics of SQL queries, its validation, and applications. *PVLDB*, 2017. doi:10.14778/3151113.3151116.

11 Oleg Kiselyov and Tatsuya Katsushima. Sound and efficient language-integrated query - maintaining the ORDER. In *APLAS 2017*, pages 364–383, 2017. doi:10.1007/978-3-319-71237-6_18.

12 Leonid Libkin and Limsoon Wong. Conservativity of nested relational calculi with internal generic functions. *Inf. Process. Lett.*, 49(6):273–280, 1994. doi:10.1016/0020-0190(94)90099-X.

13 Leonid Libkin and Limsoon Wong. Query languages for bags and aggregate functions. *J. Comput. Syst. Sci.*, 55(2), 1997. doi:10.1006/jcss.1997.1523.

14 Sam Lindley and James Cheney. Row-based effect types for database integration. In *TLDI*, 2012. doi:10.1145/2103786.2103798.

15 Sam Lindley and Ian Stark. Reducibility and $\top\top$-lifting for computation types. In *TLCA*, 2005. doi:10.1007/11417170_20.

16 Erik Meijer, Brian Beckman, and Gavin M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD*, 2006. doi:10.1145/1142473.1142552.

17 Jan Paredaens and Dirk Van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Trans. Database Syst.*, 17(1), 1992. doi:10.1145/128765.128768.

18 Andrew M. Pitts. Parametric polymorphism and operational equivalence (preliminary version). In *HOOTS II*, volume 10, pages 2–27, 1998. doi:10.1016/S1571-0661(05)80685-1.

19 W. Ricciotti and J. Cheney. Strongly Normalizing Audited Computation. In V. Goranko and M. Dam, editors, *CSL 2017*, volume 82 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 36:1–36:21. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.CSL.2017.36.

20 Wilmer Ricciotti and James Cheney. Mixing set and bag semantics. In *DBPL*, pages 70–73, 2019. doi:10.1145/3315507.3330202.

21 Philip Trinder and Philip Wadler. Improving list comprehension database queries. In *TENCON '89.*, 1989. doi:10.1109/TENCON.1989.176921.

22 Philip Wadler. Comprehending monads. *Math. Struct. in Comp. Sci.*, 2(4), 1992. doi:10.1017/S0960129500001560.

23 Limsoon Wong. Normal forms and conservative extension properties for query languages over collection types. *J. Comput. Syst. Sci.*, 52(3), 1996. doi:10.1006/jcss.1996.0037.

24 Limsoon Wong. Kleisli, a functional query system. *J. Funct. Programming*, 10(1), 2000. doi:10.1017/S0956796899003585.

A Proofs

This appendix expands on some results whose proofs were omitted or only sketched in the paper.

Since under plain reduction each term can be reduced only in a finite number of ways, it is easy to see that $\nu(M)$ is defined for any strongly normalizing term M; however, under renaming reduction, a term may be reduced in an infinite number of ways because, if $M \rightsquigarrow N$, there may be infinite R, σ such that $N = R\sigma$. Fortunately, we can prove that to any renaming reduction chain there corresponds a plain reduction chain of the same length, and vice-versa: consequently, the set of strongly normalizing terms is the same under the two notions of reduction, and $\nu(M)$ refers to the maximal length of reduction chains starting at M either with or without renaming.

▶ **Lemma 47.** *For all contexts C, terms N and indices p, if $C[p \mapsto N] \in \mathcal{SN}$, we have $C \in \mathcal{SN}$; if $p \in \mathsf{supp}(C)$, then $N \in \mathcal{SN}$.*

Proof of Lemma 11. *If $Q \mathbin{\textcircled{\scriptsize p}} F \in \mathcal{SN}$, then $\nu(Q) \leq \nu(Q \mathbin{\textcircled{\scriptsize p}} F)$.*

By induction on the possible reduction sequences in Q, we show there exists a corresponding reduction sequence with the same length in $Q \mathbin{\textcircled{\scriptsize p}} F$. ◀

▶ **Lemma 48.** *If $M \rightsquigarrow N$, then $M\sigma \rightsquigarrow N\sigma$.*

▶ **Lemma 49.**

1. *If $M \underbrace{\rightsquigarrow \cdots \rightsquigarrow}_{n \textit{ times}} N$, then $M \underbrace{\overset{\mathsf{id}}{\rightsquigarrow} \cdots \overset{\mathsf{id}}{\rightsquigarrow}}_{n \textit{ times}} N$*
2. *If $M \overset{\sigma_1}{\rightsquigarrow} \cdots \overset{\sigma_n}{\rightsquigarrow} N$, then $M \underbrace{\rightsquigarrow \cdots \rightsquigarrow}_{n \textit{ times}} N\sigma_n \cdots \sigma_1$*

Proof. The first part of the lemma is trivial. For the second part, proceed by induction on the length of the reduction chain: in the inductive case, we have $M \overset{\sigma_1}{\rightsquigarrow} \cdots \overset{\sigma_n}{\rightsquigarrow} M' \overset{\sigma_{n+1}}{\rightsquigarrow} N$ by hypothesis and $M \rightsquigarrow \cdots \rightsquigarrow M'\sigma_n \cdots \sigma_1$ by induction hypothesis; to obtain the thesis, we only need to prove that

$$M'\sigma_n \cdots \sigma_1 \rightsquigarrow N\sigma_{n+1} \cdots \sigma_1$$

In order for this to be true, by Lemma 48, it is sufficient to show that $M' \rightsquigarrow N\sigma_{n+1}$; this is by definition equivalent to $M' \overset{\sigma_{n+1}}{\rightsquigarrow} N$, which we know by hypothesis. ◀

▶ **Corollary 50.** *Suppose $M \in \mathcal{SN}$: if $M \overset{\sigma}{\rightsquigarrow} M'$, then $\nu(M')$ is defined and $\nu(M') < \nu(M)$.*

Proof. By Lemma 49, for any plain reduction chain there exists a renaming reduction chain of the same length, and vice-versa. Thus, since plain reduction lowers the length of the maximal reduction chain (Lemma 10), the same holds for renaming reduction. ◀

Proof of Lemma 14.

1. *For all continuations K, if $K \rightsquigarrow C$, there exist a continuation K' and a finite map σ such that $K \overset{\sigma}{\rightsquigarrow} K'$ and $K'\sigma = C$.*
2. *For all auxiliary continuation Q, if $Q \rightsquigarrow C$, there exist an auxiliary continuation Q' and a finite map σ such that $Q \overset{\sigma}{\rightsquigarrow} Q'$ and $Q'\sigma = C$.*

Let C be a contractum of the continuation we wish to reduce. This contractum will not, in general, satisfy the side condition that holes must be linear; however we can show that, for any context with duplicated holes, there exists a structurally equal context with linear holes. Operationally, if C contains n holes, we generate n different fresh indices in $\mathcal{P}$, and replace the index of each hole in C with a different fresh index to obtain a new context C': this induces a finite map $\sigma : \mathsf{supp}(C') \to \mathsf{supp}(C)$ such that $C'\sigma = C$. By structural induction on the derivation of the reduction and by case analysis on the structure of K (or on the structure of Q) we show that C' must also satisfy the grammar in Definition 5 (resp. Definition 6); furthermore, C' satisfies the linearity condition by construction, which proves it is a continuation K' (resp. an auxiliary continuation Q'). ◀

▶ **Lemma 51.** *For all contexts C and hole instantiations η, if $C \rightsquigarrow C'$, then $C\eta \rightsquigarrow C'\eta$.*

▶ **Lemma 52.** *For all contexts C, finite maps σ, and hole instantiations η such that, for all $p \in \mathrm{dom}(\eta)$, $\mathsf{supp}(\eta(p)) \cap \mathrm{dom}(\sigma) = \emptyset$, we have $C\sigma\eta = C\eta^\sigma\sigma$.*

Proof. By structural induction on C. The interesting case is when $C = [p]$. If $\sigma(p) \in \mathrm{dom}(\eta)$, we have $[p]\,\sigma\eta = [\sigma(p)]\,\eta = \eta(\sigma(p)) = \eta(\sigma(p))\sigma = [p]\,\eta^\sigma\sigma$; otherwise, $[p]\,\sigma\eta = [p] = [p]\,\eta^\sigma\sigma$. ◀

Proof of Lemma 16. *For all contexts C, renamings σ, and hole instantiations η such that, for all $p \in \mathrm{dom}(\eta)$, $\mathsf{supp}(\eta(p)) \cap \mathrm{dom}(\sigma) = \emptyset$, if $C \overset{\sigma}{\leadsto} C'$, then $C\eta \overset{\sigma}{\leadsto} C'\eta^\sigma$.*

By definition of $\overset{\sigma}{\leadsto}$, we have $C \leadsto C'\sigma$; then, by Lemma 51, we obtain $C\eta \leadsto C'\sigma\eta$; by Lemma 52, we know $C'\sigma\eta = C'\eta^\sigma\sigma$; then the thesis $C\eta \overset{\sigma}{\leadsto} C'\eta^\sigma$ follows immediately by the definition of $\overset{\sigma}{\leadsto}$. ◀

▶ **Lemma 53.**
If $M \leadsto M'$ and $p \in \mathsf{supp}(Q)$, then $Q[p \mapsto M] \overset{\mathsf{id}}{\leadsto} Q[p \mapsto M']$.

Proof. By induction on the structure of Q, we show that for each reduction in the hypothesis, we can construct a corresponding reduction proving the thesis. ◀

▶ **Lemma 54** (classification of reductions in applied continuations)**.** *Suppose $Q\eta \leadsto N$, where η is permutable, and $\mathrm{dom}(\eta) \subseteq \mathsf{supp}(Q)$; then one of the following holds:*

1. *there exist an auxiliary continuation Q' and a finite map σ such that $N = Q'\eta^\sigma$, where η^σ is permutable, and $Q \overset{\sigma}{\leadsto} Q'$: in this case, we say the reduction is* within *Q;*
2. *there exist auxiliary continuations Q_1, Q_2, an index $q \in \mathsf{supp}(Q_1)$, a variable x, and a term L such that $Q = (Q_1 \textcircled{q} \bigcup \{x \leftarrow \{\overline{L}\}\})[q \mapsto Q_2]$, and $N = Q_1[q \mapsto Q_2 \left[\overline{L}/x\right]]\eta^*$, where we define $\eta^*(p) = \eta(p) \left[\overline{L}/x\right]$ for all $p \in \mathsf{supp}(Q_2)$, otherwise $\eta^*(p) = \eta(p)$: this is a reduction within Q too;*
3. *there exists a permutable η' such that $N = Q\eta'$ and $\eta \leadsto \eta'$: in this case we say the reduction is* within *η;*
4. *there exist an auxiliary continuation Q_0, an index p such that $p \in \mathsf{supp}(Q_0)$ and $p \in \mathrm{dom}(\eta)$, a frame F and a term M such that $N = Q_0[p \mapsto M]\eta_{\neg p}$, $Q = Q_0 \textcircled{p} F$, and $F^p[p \mapsto \eta(p)] \leadsto M$: in this case we say the reduction is* at the interface*.*

Furthermore, if Q is a regular continuation K, then the Q' in case 1 can be chosen to be a regular continuation K', and case 2 cannot happen.

Proof. By induction on Q with a case analysis on the reduction rule applied. ◀

Proof of Lemma 19. *If $Q\eta \in \mathcal{SN}$, then $Q \in \mathcal{SN}$ and $\nu(Q) \leq \nu(Q\eta)$.*

We proceed by well-founded induction on (Q, η) using the metric:

$$(Q_1, \eta_1) \prec (Q_2, \eta_2) \iff \exists \sigma : Q\eta_1 \overset{\sigma}{\leadsto} Q'\eta_2$$

For all contractions $Q \overset{\sigma}{\leadsto} Q'$, by Lemma 53 we know $Q\eta \overset{\sigma}{\leadsto} Q'\eta^\sigma$: then we can apply the IH with (Q', η^σ) to prove Q': thus we conclude $Q \in \mathcal{SN}$.

To prove $\nu(Q) \leq \nu(Q\eta)$, it is sufficient to see that for each reduction step in Q we have a corresponding reduction step in $Q\eta$: thus the reduction chains starting in $Q\eta$ must be at least as long as those in Q. ◀

▶ **Lemma 55.** *Suppose* $\mathrm{CR1}(\mathcal{C})$*: then for all indices p, q, $[p] \in (q \mapsto \mathcal{C})^\top$.*

Proof. To prove the lemma, it is sufficient to show that for all $M \in \mathcal{C}$ we have $[p][q \mapsto \{M\}] \in \mathcal{SN}$. This term is equal to either $\{M\}$ (if $p = q$) or to $[p]$ (otherwise); both terms are s.n. (in the case of $\{M\}$, this is because CR1 holds for $\mathcal{C}$, thus $M \in \mathcal{SN}$). ◀

Proof of Lemma 23. *For all p and all non-empty $\mathcal{C}$, $(p : \mathcal{C})^\top \subseteq \mathcal{SN}$.*

We assume $K \in (p : \mathcal{C})^\top$ and $M \in \mathcal{C}$: by definition, we know that $K[p \mapsto \{M\}] \in \mathcal{SN}$; then we have $K \in \mathcal{SN}$ by Lemma 47. ◀

Proof of Lemma 24. *If* CR1($\mathcal{C}$)*, then* CR1($\mathcal{C}^{\top\top}$)*.*

We need to prove that if $M \in \mathcal{C}^{\top\top}$, then $M \in \mathcal{SN}$. By the definition of $\mathcal{C}^{\top\top}$, we know that for all p, $K[p \mapsto M] \in \mathcal{SN}$ whenever $K \in (p : \mathcal{C})^{\top}$. Now assume any p, and by Lemma 55 choose $K = [p]$: then $K[p \mapsto M] = M \in \mathcal{SN}$, which proves the thesis. ◀

▶ **Lemma 56.** *If $K \in \mathcal{SN}$ is a continuation, then for all indices p we have $K[p \mapsto \emptyset] \in \mathcal{SN}$.*

Proof. We proceed by well-founded induction, using the metric:

$$(K_1, p_1) \prec (K_2, p_2) \iff (\nu(K_1), \|K_1\|_{p_1}) \lessdot (\nu(K_2), \|K_2\|_{p_2})$$

- $K'[p \mapsto \emptyset]^{\sigma}$, where $K \overset{\sigma}{\leadsto} K'$: by Lemma 18, we need to show $K'[q \mapsto \emptyset] \in \mathcal{SN}$ whenever $\sigma(q) = p$; this follows from the IH, with $\nu(K') < \nu(K)$ by Lemma 10.
- $K_0[p \mapsto \emptyset]$, where $K = K_0 \textcircled{\scriptsize p} F$ for some frame F: by Lemma 11 we have $\nu(K_0) \leq \nu(K)$; furthermore, we can easily prove that $\|K_0\|_p < \|K\|_p$; then the thesis follows immediately from the IH. ◀

Proof of Lemma 31. *For all $\mathcal{C}$, $\emptyset \in \mathcal{C}^{\top\top}$.*

Immediate from Lemma 56, by unfolding the definition of $\mathcal{C}^{\top\top}$. ◀

Proof of Lemma 32. *For all Q-continuations Q, O_1, O_2 with pairwise disjoint supports, if $Q[p \mapsto O_1] \in \mathcal{SN}$ and $Q[p \mapsto O_2] \in \mathcal{SN}$, then $Q[p \mapsto O_1 \cup O_2] \in \mathcal{SN}$.*

We assume $p \in \mathsf{supp}(Q)$ (otherwise, $Q[p \mapsto O_1] = Q[p \mapsto O_2] = Q[p \mapsto O_1 \cup O_2]$, and the thesis holds trivially). Then, by Lemma 47, $Q[p \mapsto O_1] \in \mathcal{SN}$ and $Q[p \mapsto O_2] \in \mathcal{SN}$ imply $Q \in \mathcal{SN}$, $O_1 \in \mathcal{SN}$, and $O_2 \in \mathcal{SN}$: thus we can proceed by well-founded induction on (Q, p, O_1, O_2) using the following metric:

$$\begin{array}{l}(Q^1, p^1, O_1^1, O_2^1) \prec (Q^2, p^2, O_1^2, O_2^2) \\ \iff (\nu(Q^1), \left\|Q^1\right\|_{p^1}, \nu(O_1^1) + \nu(O_2^1)) \lessdot (\nu(Q^2), \left\|Q^2\right\|_{p^2}, \nu(O_1^2) + \nu(O_2^2))\end{array}$$

to prove that if $Q[p \mapsto O_1] \in \mathcal{SN}$ and $Q[p \mapsto O_2] \in \mathcal{SN}$, then $Q[p \mapsto O_1 \cup O_2] \in \mathcal{SN}$. Equivalently, we will consider all possible contracta and show that each of them must be a strongly normalizing term; we will apply the induction hypothesis to new auxiliary continuations obtained by placing pieces of Q into O_1 and O_2: the hypothesis on the supports of the continuations being disjoint is used to make sure that the new continuations do not contain duplicate holes and are thus well-formed. By cases on the possible contracta:

- $Q_1[q \mapsto Q_2 \left[\overline{L}/x\right]][p \mapsto (O_1 \left[\overline{L}/x\right]) \cup (O_2 \left[\overline{L}/x\right])]$ (where $Q = (Q_1 \textcircled{\scriptsize q} \bigcup \{x \leftarrow \{\overline{L}\}\})[q \mapsto Q_2]$, $q \in \mathsf{supp}(Q_1)$, $p \in \mathsf{supp}(Q_2)$): let $Q' = Q_1[q \mapsto Q_2 \left[\overline{L}/x\right]]$, and note that $Q \leadsto Q'$, hence $\nu(Q') < \nu(Q)$; note $Q[p \mapsto O_1] \leadsto Q'[p \mapsto O_1 \left[\overline{L}/x\right]]$, hence since the former term is s.n., so must be the latter, and hence also $O_1 \left[\overline{L}/x\right] \in \mathcal{SN}$; similarly, $O_2 \left[\overline{L}/x\right]$; then we can apply the IH with $(Q', p, O_1 \left[\overline{L}/x\right], O_2 \left[\overline{L}/x\right])$ to obtain the thesis.
- $Q'[p \mapsto O_1 \cup O_2]^{\sigma}$ (where $Q \overset{\sigma}{\leadsto} Q'$): by Lemma 18, we need to prove that, for all q s.t. $\sigma(q) = p$, $Q'[q \mapsto O_1 \cup O_2] \in \mathcal{SN}$; since $Q[p \mapsto O_1] \in \mathcal{SN}$, we also have $Q'[p \mapsto O_1]^{\sigma} \in \mathcal{SN}$, which implies $Q'[q \mapsto O_1] \in \mathcal{SN}$ by Lemma 18; for the same reason, $Q'[q \mapsto O_2] \in \mathcal{SN}$; by Lemma 10, $\nu(Q') < \nu(Q)$, thus the thesis follows by IH.
- $Q_1[p \mapsto (\bigcup \{Q_2 | x \leftarrow O_1\}) \cup (\bigcup \{Q_2 | x \leftarrow O_2\})]$ (where $Q = Q_1 \textcircled{\scriptsize p} \bigcup \{Q_2 | x\}$): by Lemma 11, $\nu(Q_1) \leq \nu(Q)$; we also know $\|Q_1\|_p < \|Q\|_p$; take $O_1' := \bigcup \{Q_2 | x \leftarrow O_1\}$ and note that, since $Q[p \mapsto O_1] = Q_0[p \mapsto O_1']$, we have O_1' is a subterm of a strongly normalizing term, thus $O_1' \in \mathcal{SN}$; similarly, we define $O_2' := \bigcup \{Q_2 | x \leftarrow O_2\}$ and show it is s.n. in a similar way; then (Q_1, p, O_1', O_2') reduce the metric, and we can prove the thesis by IH.

- $Q_1[p \mapsto (\bigcup\{O_1|x \leftarrow Q_2\}) \cup (\bigcup\{O_2|x \leftarrow Q_2\})]$ (where $Q = Q_1 \textcircled{p} \bigcup\{x \leftarrow Q_2\}$): by Lemma 11, $\nu(Q_1) \leq \nu(Q)$; we also know $\|Q_1\|_p < \|Q\|_p$; take $O'_1 := \bigcup\{O_1|x \leftarrow Q_2\}$ and note that, since $Q[p \mapsto O_1] = Q_1[p \mapsto O'_1]$, we have O'_1 is a subterm of a strongly normalizing term, thus $O'_1 \in \mathcal{SN}$; similarly, we define $O'_2 := \bigcup\{O_2|x \leftarrow Q_2\}$ and show it is s.n. in a similar way; then (Q_1, p, O'_1, O'_2) reduce the metric, and we can prove the thesis by IH.
- $Q_0[p \mapsto (\texttt{where}\ \overline{B}\ \texttt{do}\ O_1) \cup (\texttt{where}\ \overline{B}\ \texttt{do}\ O_2)]$ (where $Q = Q_0 \textcircled{p} \texttt{where}\ \overline{B}$): by Lemma 11, $\nu(Q_0) \leq \nu(Q)$; we also know $\|Q_0\|_p < \|Q\|_p$; take $O'_1 := \texttt{where}\ B\ \texttt{do}\ O_1$ and note that, since $Q[p \mapsto O_1] = Q_0[p \mapsto O'_1]$, we have O'_1 is a subterm of a strongly normalizing term, thus $O'_1 \in \mathcal{SN}$; similarly, we define $O'_2 := \texttt{where}\ \overline{B}\ \texttt{do}\ O_2$ and prove it is strongly normalizing in the same way; then (Q_0, p, O'_1, O'_2) reduce the metric, and we can prove the thesis by IH.
- Contractions within O_1 or O_2 reduce $\nu(O_1) + \nu(O_2)$, thus the thesis follows by IH. ◀

Reducibility for conditionals similarly to comprehensions. However, to consider all the conversions commuting with `where`, we need to use the more general auxiliary continuations.

▶ **Lemma 57.** *If $Q[p \mapsto M \cup N] \in \mathcal{SN}$, then $Q[p \mapsto M] \in \mathcal{SN}$ and $Q[p \mapsto N] \in \mathcal{SN}$; furthermore, we have:*

$$\nu(Q[p \mapsto M]) \leq \nu(Q[p \mapsto M \cup N])$$
$$\nu(Q[p \mapsto N]) \leq \nu(Q[p \mapsto M \cup N])$$

Proof. We assume $p \in \mathsf{supp}(Q)$ (otherwise, $Q[p \mapsto M] = Q[p \mapsto N] = Q[p \mapsto M \cup N]$, and the thesis holds trivially), then we show that any contraction in $Q[p \mapsto M]$ has a corresponding non-empty reduction sequence in $Q[p \mapsto M \cup N]$, and the two reductions preserve the term form, therefore no reduction sequence of $Q[p \mapsto M]$ is longer than the maximal one in $Q[p \mapsto M \cup N]$. The same reasoning applies to $Q[p \mapsto N]$. ◀

Proof of Lemma 36. *Let Q, B, O such that $Q[p \mapsto O] \in \mathcal{SN}$, $B \in \mathcal{SN}$, and $\mathsf{supp}(Q) \cap \mathsf{supp}(O) = \emptyset$. Then $Q[p \mapsto \texttt{where}\ B\ \texttt{do}\ O] \in \mathcal{SN}$.*

In this proof, we assume the names of bound variables are chosen so as to avoid duplicates, and distinct from the free variables. We proceed by well-founded induction on (Q, B, O, p) using the following metric:

$$(Q_1, B_1, O_1, p_1) \prec (Q_2, B_2, O_2, p_2) \iff$$
$$(\nu(Q_1[p_1 \mapsto O_1]) + \nu(B_1), |Q_1|_{p_1}, \mathrm{size}(O_1)) \lessdot (\nu(Q_2[p_2 \mapsto O_2]) + \nu(B_2), |Q_2|_{p_2}, \mathrm{size}(O_2))$$

We will consider all possible contracta and show that each of them must be a strongly normalizing term; we will apply the induction hypothesis to new auxiliary continuations obtained by placing pieces of O into Q or vice-versa: the hypothesis on the supports of Q and O being disjoint is used to make sure that the new continuations do not contain duplicate holes and are thus well-formed. By cases on the possible contracta:

- $Q_1[q \mapsto Q_2 \left[\overline{L}/x\right]][p \mapsto (\texttt{where}\ B\ \texttt{do}\ O) \left[\overline{L}/x\right]]$, where $Q = (Q_1 \textcircled{q} \bigcup\{x \leftarrow \{\overline{L}\}\})[q \mapsto Q_2]$, $q \in \mathsf{supp}(Q_1)$, and $p \in \mathsf{supp}(Q_2)$; by the freshness condition we know $x \notin \mathrm{FV}(B)$, thus $(\texttt{where}\ B\ \texttt{do}\ O) \left[\overline{L}/x\right] = \texttt{where}\ B\ \texttt{do}\ (O \left[\overline{L}/x\right])$; we take $Q' = Q_1[q \mapsto Q_2 \left[\overline{L}/x\right]]$ and $O' = O \left[\overline{L}/x\right]$, and note that $\nu(Q'[p \mapsto O']) < \nu(Q[p \mapsto O])$, because the former term is a contractum of the latter: then we can apply the IH to prove $Q'[p \mapsto \texttt{where}\ B\ \texttt{do}\ O'] \in \mathcal{SN}$, as needed.

- $Q'[p \mapsto \texttt{where}\ B\ \texttt{do}\ O]^\sigma$, where $Q \overset{\sigma}{\rightsquigarrow} Q'$. We know $\nu(Q'[p \mapsto O]^\sigma) < \nu(Q[p \mapsto O])$ by Lemma 10 since the latter is a contractum of the former. By Lemma 18, for all q s.t. $\sigma(q) = p$ we have $\nu(Q'[q \mapsto O]) \leq \nu(Q'[p \mapsto O]^\sigma)$; we can thus apply the IH to obtain $Q[q \mapsto \texttt{where}\ B\ \texttt{do}\ O] \in \mathcal{SN}$ whenever $\sigma(q) = p$. By Lemma 18, this implies the thesis.
- $Q_1[p \mapsto \texttt{where}\ B\ \texttt{do}\ \bigcup\{Q_2|x \leftarrow O\}]$, where $Q = Q_1 \textcircled{\scriptsize p} \bigcup\{Q_2|x\}$; we take $O' = \bigcup\{Q_2|x \leftarrow O\}$, and we note that $Q[p \mapsto O] = Q_1[p \mapsto O']$ and $|Q_1|_p < |Q|_p$; we can thus apply the IH to prove $Q_1[p \mapsto \texttt{where}\ B\ \texttt{do}\ O'] \in \mathcal{SN}$, as needed.
- $Q[p \mapsto \emptyset]$, where $O = \emptyset$: this term is strongly normalizing by hypothesis.
- $Q[p \mapsto (\texttt{where}\ B\ \texttt{do}\ O_1) \cup (\texttt{where}\ B\ \texttt{do}\ O_2)]$, where $O = O_1 \cup O_2$; for $i = 1, 2$, we prove $Q[p \mapsto O_i] \in \mathcal{SN}$ and $\nu(Q[p \mapsto O_i]) \leq \nu(Q[p \mapsto O])$ by Lemma 32, and we also note $\text{size}(O_i) < \text{size}(O)$; then we can apply the IH to prove $Q[p \mapsto \texttt{where}\ B\ \texttt{do}\ O_i] \in \mathcal{SN}$, which implies the thesis by Lemma 32.
- $Q[p \mapsto \bigcup\{\texttt{where}\ B\ \texttt{do}\ O_1|x \leftarrow O_2\}]$, where $O = \bigcup\{O_1|x \leftarrow O_2\}$; we take $Q' = Q \textcircled{\scriptsize p} \bigcup\{x \leftarrow O_2\}$ and we have that $Q'[p \mapsto \texttt{where}\ B\ \texttt{do}\ O_1]$ is equal to $Q[p \mapsto \bigcup\{\texttt{where}\ B\ \texttt{do}\ O_1|x \leftarrow O_2\}]$; we thus note $\nu(Q'[p \mapsto O_1]) = \nu(Q[p \mapsto O])$, $|Q'|_p = |Q|_p$, and $\text{size}(O_1) < \text{size}(O)$, thus we can apply the IH to prove $Q'[p \mapsto \texttt{where}\ B\ \texttt{do}\ O_1] \in \mathcal{SN}$, as needed.
- Reductions within B or O make the induction metric smaller, thus follow immediately from the IH. ◀

Semi-Axiomatic Sequent Calculus

Henry DeYoung
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA
hdeyoung@andrew.cmu.edu

Frank Pfenning
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA
fp@cs.cmu.edu

Klaas Pruiksma
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA
kpruiksm@andrew.cmu.edu

Abstract

We present the semi-axiomatic sequent calculus (SAX) that blends features of Gentzen's sequent calculus with an axiomatic formulation of intuitionistic logic. We develop and prove a suitable analogue to cut elimination and then show that a natural computational interpretation of SAX provides a simple form of shared memory concurrency.

2012 ACM Subject Classification Theory of computation → Proof theory; Computing methodologies → Concurrent programming languages

Keywords and phrases Sequent calculus, Curry-Howard isomorphism, shared memory concurrency

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.29

Funding This material is based upon work supported by the National Science Foundation under Grant No. 1718276.

Acknowledgements We would like to thank Siva Somayyajula and the anonymous reviewers for suggestions on an earlier version of this paper.

1 Introduction

The celebrated Curry-Howard isomorphism [8, 14] establishes a close connection between logic and computation. There are three interrelated components that define this correspondence: (1) propositions in the logic are interpreted as types, (2) proofs in the logic are interpreted as programs, and (3) proof reduction is interpreted as computation. Parts (2) and (3) mean that we must pay attention to the specific formulation of the logic. Curry used combinatory logic [9] which derives from an axiomatic formulation of inference and arrives at combinatory reduction. Howard [14] established the close connection between natural deduction and the simply-typed λ-calculus. As a further example, Herbelin [13] introduced LJT, a formulation of the sequent calculus with a stoup whose computational interpretation uses explicit substitutions. In all three of these cases, the logic is simply intuitionistic logic but its computational interpretation is quite different.

In this paper we add another entry to this list of correspondences. We introduce the semi-axiomatic sequent calculus (Sax) which blends features of the sequent calculus with axiomatic presentations of intuitionistic logic. We show that Sax satisfies a version of cut elimination, where cut-free proofs have the subformula property, thereby establishing the basics of the proof theory for Sax.

As for other inference systems, the proof of cut elimination contains the seeds of its operational interpretation. For example, normalization for natural deduction is based on a substitution operation that corresponds to β-reduction, and cut reduction on LJT corresponds to $\overline{\lambda}$-reduction in a calculus of explicit substitution. In Sax, a cut disappears entirely during

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 29; pp. 29:1–29:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

a principal cut reduction, which is only possible because certain special forms of analytic cuts we call *snips* are allowed to remain in cut-free proofs. This disappearing act can be explained when we think of the cut reduction itself as reading from or writing to memory, which are atomic actions. Under this view, we recognize that one operational interpretation of SAX equates proofs with processes where some actively compute and others are passive, thereby representing shared memory cells. We formalize the concurrent reduction strategy (that is, the evaluation relation) using techniques from substructural operational semantics [27, 5, 6], which can be mapped back directly to proofs (with a loss of readability). As a concurrent programming language, the interpretation of SAX then has the desired properties such as type preservation, progress, confluence, and termination.

Our results are yet another illustration of the flexibility of the sequent calculus as a foundation for computation at a high level of abstraction. SAX is remarkably simple: it requires no stoup or other structural devices, just the initial leap of faith to replace noninvertible rules by axioms. It also provides a path towards understanding a simple, logically grounded form of shared memory concurrency under a Curry-Howard interpretation.

In Section 2, we provide an overview of the standard sequent calculus G_3 for background, before relating it to SAX in Section 3. This section contains our first set of contributions: SAX itself, translations between G_3 and SAX, and a cut elimination result for SAX. We briefly discuss some example SAX proofs in Section 4, which also serve as examples of computations later on. Section 5 contains our second major set of contributions. We provide an operational interpretation of SAX using *shared memory* and prove the basic theorems of progress, preservation, and (because the semantics are nondeterministic) confluence. A termination theorem, analagous to normalization for natural deduction, is more involved, and is delayed to its own section (Section 6).

In prior work we have given a different interpretation of SAX (without investigating its metatheory from a logical and proof-theoretic perspective) using session types and message passing, starting with purely linear intuitionistic logic and generalizing all the way to adjoint logic, combining structural and substructural intuitionistic logics [24]. This in turn built on the Curry-Howard interpretation of linear logic as session-typed processes using *message-passing* concurrency [3, 29, 4]. (Modeling asynchronous communication in a Curry-Howard interpretation of linear logic was the original motivation behind SAX.) Starting from Herbelin's seminal work, others have also given computational interpretations of intuitionistic and classical sequent calculi at various level of abstraction (for example, with multiple conclusions and stoups [7] or with de Bruijn indices [2]). These are, however, quite different from the interpretation presented here.

2 Ordinary Sequent Calculus

For reference, we provide the standard sequent calculus G_3 [15] in Figure 1 so we can explicitly relate it to SAX. In G_3 all structural rules remain implicit, and antecedents Γ of a sequent $\Gamma \vdash A$ are propagated to all premises. For brevity, we omit the logical constants falsehood ($\bot$) and truth ($\top$) since they are not particularly interesting, especially from the computational perspective. We do, however, present two forms of conjunction: $A \otimes B$ and $A \mathbin{\&} B$, which are distinguished here by their left rules extrapolated from linear logic. Although these conjunctions are logically equivalent (that is, $A \otimes B \dashv\vdash A \mathbin{\&} B$), they are computationally distinct: functionally, $A \otimes B$ corresponds to eager pairs whereas $A \mathbin{\&} B$ corresponds to lazy pairs (see, for example, call-by-push-value [16]).

$$\frac{}{\Gamma, A \vdash A}\ \mathsf{Id}_A \qquad \frac{\Gamma \vdash A \quad \Gamma, A \vdash C}{\Gamma \vdash C}\ \mathsf{Cut}_A$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}\ {\supset}R \qquad \frac{\Gamma, A \supset B \vdash A \quad \Gamma, A \supset B, B \vdash C}{\Gamma, A \supset B \vdash C}\ {\supset}L$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \otimes B}\ {\otimes}R \qquad \frac{\Gamma, A \otimes B, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\ {\otimes}L$$

$$\frac{}{\Gamma \vdash \mathbf{1}}\ \mathbf{1}R \qquad \frac{\Gamma, \mathbf{1} \vdash C}{\Gamma, \mathbf{1} \vdash C}\ \mathbf{1}L$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ {\vee}R_1 \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ {\vee}R_2 \qquad \frac{\Gamma, A \vee B, A \vdash C \quad \Gamma, A \vee B, B \vdash C}{\Gamma, A \vee B \vdash C}\ {\vee}L$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \mathbin{\&} B}\ {\&}R \qquad \frac{\Gamma, A \mathbin{\&} B, A \vdash C}{\Gamma, A \mathbin{\&} B \vdash C}\ {\&}L_1 \quad \frac{\Gamma, A \mathbin{\&} B, B \vdash C}{\Gamma, A \mathbin{\&} B \vdash C}\ {\&}L_2$$

Figure 1 The Sequent Calculus (G_3).

We also include $\mathbf{1}$, the unit of $A \otimes B$, which can be thought of as the nullary form of the binary $A \otimes B$. The $\mathbf{1}L$ rule may look a bit surprising: we might expect the antecedent $\mathbf{1}$ to be deleted in its premise. But since we have implicit weakening and contraction, the principal formula of every left rule, $\mathbf{1}L$ included, must be preserved. Preserving the antecedents turns out to be computationally significant in the shared memory interpretation that we present in Section 5.

We have the following standard theorems with standard proofs. Gentzen's original sequent calculus had explicit structural rules and used an intermediate system with a rule called Mix in the proof of cut elimination [11]. We sketch the proofs of admissibility of cut and cut elimination [21] that are more closely related in structure to the proof in SAX in Section 3.

▶ **Theorem 1** (Admissibility of Cut in G_3 [11, 21]). *If there are cut-free derivations $\Gamma \vdash A$ and $\Gamma, A \vdash C$ then there is a cut-free derivation of $\Gamma \vdash C$.*

Proof. By nested induction on the structure of A and then simultaneously on the derivations of $\Gamma \vdash A$ and $\Gamma, A \vdash C$. ◀

▶ **Theorem 2** (Cut Elimination [11, 21]). *If $\Gamma \vdash A$ then there is a cut-free derivation of $\Gamma \vdash A$.*

Proof. By induction on the structure of the given derivation, using Theorem 1 for cut. ◀

It is evident that the cut-free sequent calculus has the subformula property since, except for cut, the premises of all rules are subformulas of the propositions in the conclusions. This allows us to view the right- and left-rules for the logical connectives in G_3 as a compositional explanation for the meaning of propositions [10, 19]. Another component of such an interpretation is identity expansion (we require the identity rule Id_A only for atomic propositions), which is significant from the foundational perspective but not computationally interesting since identity expansions correspond to extensional equalities.

$$\frac{}{\Gamma, A \vdash A}\ \mathsf{Id}_A \qquad \frac{\Gamma \vdash A \quad \Gamma, A \vdash C}{\Gamma \vdash C}\ \mathsf{Cut}_A$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}\ {\supset}R \qquad \frac{}{\Gamma, A, A \supset B \vdash B}\ {\supset}L^0$$

$$\frac{}{\Gamma, A, B \vdash A \otimes B}\ {\otimes}R^0 \qquad \frac{\Gamma, A \otimes B, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\ {\otimes}L$$

$$\frac{}{\Gamma \vdash \mathbf{1}}\ \mathbf{1}R^0 \qquad \frac{\Gamma, \mathbf{1} \vdash C}{\Gamma, \mathbf{1} \vdash C}\ \mathbf{1}L$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}\ \&R \qquad \frac{}{\Gamma, A \,\&\, B \vdash A}\ \&L_1^0 \quad \frac{}{\Gamma, A \,\&\, B \vdash B}\ \&L_1^0$$

$$\frac{}{\Gamma, A \vdash A \vee B}\ {\vee}R_1^0 \quad \frac{}{\Gamma, B \vdash A \vee B}\ {\vee}R_2^0 \qquad \frac{\Gamma, A \vee B, A \vdash C \quad \Gamma, A \vee B, B \vdash C}{\Gamma, A \vee B \vdash C}\ {\vee}L$$

Figure 2 The Semi-Axiomatic Sequent Calculus (SAX).

3 The Semi-Axiomatic Sequent Calculus

The connectives in intuitionistic logic can be divided into positive ($A \otimes B$, $\mathbf{1}$, $A \vee B$) and negative ($A \supset B$, $A \,\&\, B$), where we have split conjunction into two. The right rules for negative connectives and the left rules for the positive connectives are asynchronous (in the terminology of Andreoli [1]) in the sense that a negative connective in the succedent and a positive connective in the antecedent can always be broken down eagerly with the corresponding rule when constructing a proof bottom-up. Conversely, the right rules for positive connectives and the left rules for negative connectives may have to be postponed until they can be applied. Even though in the formulation in G_3 this is not strictly accurate, we say that negative right and positive left rules are *invertible* while positive right and negative left rules are *noninvertible* (see, for example, the analysis by Liang and Miller [17]).

The semi-axiomatic sequent calculus SAX arises from G_3 by replacing all the noninvertible rules (${\supset}L, \&L, {\otimes}R, \mathbf{1}R, {\vee}R$) by corresponding axioms while leaving all the invertible rules (${\supset}R, \&R, {\otimes}L, \mathbf{1}L, {\vee}L$) unchanged. Identity and cut also remain unchanged. We annotate all the rules that are now axioms with the superscript 0, indicating the zero premises of the rule. A summary of the rules can be found in Figure 2.

First, we should convince ourselves that G_3 and SAX have the same derivable sequents.

▶ **Theorem 3** (Translations between G_3 and SAX). *$\Gamma \vdash A$ in G_3 iff $\Gamma \vdash A$ in SAX.*

Proof. In one direction, we can derive each new SAX axiom using the corresponding rules of G_3 and the identity. For example, we can derive ${\supset}L^0$ in G_3 as follows:

$$\frac{\dfrac{}{\Gamma, A, A \supset B \vdash A}\ \mathsf{Id}_A \quad \dfrac{}{\Gamma, A, A \supset B, B \vdash B}\ \mathsf{Id}_B}{\Gamma, A, A \supset B \vdash B}\ {\supset}L$$

The other direction requires the uses of cut in SAX to derive the rules of G_3. For example, we can derive $\supset L$ in Sax as follows:

$$\dfrac{\dfrac{\Gamma, A \supset B \vdash A \qquad \dfrac{}{\Gamma, A, A \supset B \vdash B}\,{\supset}L^0}{\Gamma, A \supset B \vdash B}\,\mathsf{Cut} \qquad \Gamma, A \supset B, B \vdash C}{\Gamma, A \supset B \vdash C}\,\mathsf{Cut}$$

All other cases are similar. ◀

Sax does not satisfy the standard cut elimination theorem. For example, there is no cut-free proof of $A \supset B, B \supset C \vdash A \supset C$. Instead, we have the following proof (omitting some unused antecedents):

$$\dfrac{\dfrac{\dfrac{}{A, A \supset B \vdash B}\,{\supset}L^0 \qquad \dfrac{}{B, B \supset C \vdash C}\,{\supset}L^0}{A, A \supset B, B \supset C \vdash C}\,\mathsf{Cut}_B}{A \supset B, B \supset C \vdash A \supset C}\,{\supset}R$$

Despite not being cut-free, this proof does exhibit the subformula property – its only cut is *analytic*. However, we find admitting arbitrary analytic cuts in the normal forms of proofs is too lenient and does not provide a good correspondence to operational behavior under the Curry-Howard interpretation. Instead, we would like to recognize the subformulas of the principal formula of each axiom as specific formulas that we may cut without losing the subformula property. We then have a restricted form of cut we call Snip which requires that the cut formula in one (or both) of the premises originates from an axiom in this way.

In order to make this precise we need to track specific formula occurrences, so we label each antecedent and the succedent of each sequent in a derivation with variables, where all variables in a sequent are distinct. As a representative example of an axiom, we will examine the $\supset L^0$ axiom. We call the variables $y : A$ and $z : B$ in

$$\dfrac{}{\Gamma, y : A, x : A \supset B \vdash z : B}\,{\supset}L^0$$

eligible for a Snip (or just *eligible*) and propagate this information through the derivation. In a rule with two premises a variable is eligible in the conclusion if it is eligible in at least one of the premises. If we assume or prove that an antecedent or succedent is eligible we mark it with $x^* : A$ (although the absence of a $*$ does not mean that it is ineligible). This mark is not part of the syntax of sequents, just expressing an assumed or known property of its derivation.

Since we are not in a linear logic, the cut elimination proof also seems to require that we track *irrelevant* antecedents in sequents, those that are never used. These are the (ineligible) side formulas of axioms and Id, which then propagate through the derivation. In a rule with two premises, a variable is only irrelevant in the conclusion if it is irrelevant in both premises. If we assume or prove that an antecedent is irrelevant, we mark it with $x^0 : A$ (although again the absence of the mark does not mean it is relevant). We also write Γ^0 if all variables in Γ are irrelevant. Again, the superscript 0 is not part of the syntax of sequents, just expressing an assumed or known property of its derivation.

We refer to the *status* of variables as *irrelevant* or *relevant* and *eligible* or *ineligible*, where, by its definition, an eligible variable is automatically relevant.

In the normal form of derivations we now permit cuts where at least one of the two occurrences of the cut formula is eligible for a snip. The rules can be found in Figure 3.

$$\frac{}{\Gamma^0, x : A \vdash y : A}\,\mathsf{Id}_A \qquad \frac{\Gamma \vdash x : A \quad \Gamma, x : A \vdash z : C}{\Gamma \vdash z : C}\,\mathsf{Cut}_A$$

$$\frac{\Gamma \vdash x^* : A \quad \Gamma, x : A \vdash z : C}{\Gamma \vdash z : C}\,\mathsf{Snip}^1_A \qquad \frac{\Gamma \vdash x : A \quad \Gamma, x^* : A \vdash z : C}{\Gamma \vdash z : C}\,\mathsf{Snip}^2_A$$

$$\frac{\Gamma, y : A \vdash z : B}{\Gamma \vdash x : A \supset B}\,{\supset}R \qquad \frac{}{\Gamma^0, y^* : A, x : A \supset B \vdash z^* : B}\,{\supset}L^0$$

$$\frac{}{\Gamma^0, y^* : A, z^* : B \vdash x : A \otimes B}\,{\otimes}R^0 \qquad \frac{\Gamma, x : A \otimes B, y : A, z : B \vdash w : C}{\Gamma, x : A \otimes B \vdash w : C}\,{\otimes}L$$

$$\frac{}{\Gamma^0 \vdash x : \mathbf{1}}\,\mathbf{1}R^0 \qquad \frac{\Gamma, x : \mathbf{1} \vdash w : C}{\Gamma, x : \mathbf{1} \vdash w : C}\,\mathbf{1}L$$

$$\frac{\Gamma \vdash y : A \quad \Gamma \vdash z : B}{\Gamma \vdash x : A \mathbin{\&} B}\,{\&}R \qquad \frac{}{\Gamma^0, x : A \mathbin{\&} B \vdash y^* : A}\,{\&}L^0_1 \qquad \frac{}{\Gamma^0, x : A \mathbin{\&} B \vdash z^* : B}\,{\&}L^0_2$$

$$\frac{}{\Gamma^0, x^* : A \vdash z : A \vee B}\,{\vee}R^0_1 \qquad \frac{}{\Gamma^0, y^* : B \vdash z : A \vee B}\,{\vee}R^0_2 \qquad \frac{\Gamma, z : A \vee B, x : A \vdash w : C \quad \Gamma, z : A \vee B, y : B \vdash w : C}{\Gamma, z : A \vee B \vdash w : C}\,{\vee}L$$

Figure 3 Sax in Labeled Form.

The implementation of $\supset L$ from $\supset L^0$ has two uses of cuts, but both of them are on eligible formulas and are therefore snips.

$$\frac{\dfrac{\Gamma, y : A \supset B \vdash x : A \quad \dfrac{}{\Gamma, x^* : A, y : A \supset B \vdash z^* : B}\,{\supset}L^0}{\Gamma, y : A \supset B \vdash z^* : B}\,\mathsf{Snip}^2_A \quad \Gamma, y : A \supset B, z : B \vdash w : C}{\Gamma, y : A \supset B \vdash w : C}\,\mathsf{Snip}^1_B$$

From now on when we say "cut-free" we mean that a derivation may not use Cut, but is allowed to use Snip in both of its forms. In its most pedantic version, the cut-free proof of our example would be

$$\frac{\dfrac{\dfrac{}{u^* : A, x : A \supset B, y^0 : B \supset C \vdash t^* : B}\,{\supset}L^0 \quad \dfrac{}{u^0 : A, x^0 : A \supset B, t^* : B, y : B \supset C \vdash w^* : C}\,{\supset}L^0}{u^* : A, x : A \supset B, y : B \supset C \vdash w^* : C}\,\mathsf{Snip}_B}{x : A \supset B, y : B \supset C \vdash z : A \supset C}\,{\supset}R$$

It so happens that in this example, the Snip is an instance of both Snip^1 and Snip^2. Also, in the final conclusion none of the variables are eligible or irrelevant.

The translations between Sax and G_3 give us one way to prove cut elimination for Sax.

▶ **Theorem 4** (Cut Elimination in Sax, v1). *If $\Gamma \vdash x : A$ in Sax then there is a cut-free derivation of $\Gamma \vdash x : A$ in Sax.*

Proof. We translate the given derivation from Sax to G_3. This introduces additional uses of the identity but mostly preserves the structure of the derivation. Now we apply cut elimination (Theorem 2) to obtain a cut-free derivation in G_3. The backwards translation (see the proof of Theorem 3) of the result to Sax introduces some snips but no cuts. The result is therefore cut-free in Sax. ◀

This proof induces a simple algorithm for cut elimination but it does so in an indirect way, via two translations. We are instead interested in understanding the computational behavior of Sax directly, so we look for a direct algorithm for cut elimination. This proof (and the algorithm it embodies) is somewhat more complex for Sax than for G_3 because it needs to allow snips but not general cuts.

As with G_3, we proceed in two steps: first, we show the admissibility of cut in the cut-free calculus, and then we prove cut elimination using the admissibility of cut.

▶ **Theorem 5** (Admissibility of Cut in Sax). *If there are cut-free derivations $\Gamma \vdash x : A$ and $\Gamma, x : A \vdash z : C$ then there exists a cut-free derivation of $\Gamma \vdash z : C$.*

Proof. For readability, we express the construction of $\mathcal{F}$ from $\mathcal{D}$ and $\mathcal{E}$ in the form

$$\dfrac{\overset{\mathcal{D}}{\Gamma \vdash x : A} \quad \overset{\mathcal{E}}{\Gamma, x : A \vdash z : C}}{\Gamma \vdash z : C}\ \mathsf{Cut}_A \qquad \Longrightarrow \qquad \overset{\mathcal{F}}{\Gamma \vdash z : C}$$

The proof proceeds by a nested induction, first on the structure of A and then on the structure of the first and second given derivations, $\mathcal{D}$ and $\mathcal{E}$. We exploit that adding or subtracting an irrelevant antecedent ($\mathcal{D} + \{y^0 : B\}$ and $\mathcal{D} - \{y^0 : B\}$) does not change the structure of a derivation.

However, a direct proof does not work – we need to generalize our induction hypothesis. We observe that in the resulting derivation $\mathcal{F}$, the status of variables in Γ and $z : C$ may be different from their status in $\mathcal{D}$ and $\mathcal{E}$. If a variable becomes *irrelevant* we impose no condition. If a variable $y : B$ transitions from *eligible* in $\mathcal{D}$ or $\mathcal{E}$ to *ineligible but relevant* in $\mathcal{F}$, then we demand that $B < A$, that is, B be a strict subformula of A. This will allow us to apply the induction hypothesis to B in these cases, but it also requires that this condition is preserved in each case of the proof. See Appendix A for additional proof details. ◀

▶ **Theorem 6** (Cut Elimination in Sax, v2). *If there is a derivation of $\Gamma \vdash x : A$ then there is a cut-free derivation of the same sequent.*

Proof. By a standard induction on the structure of the deduction, appealing to the admissibility of cut in the case of a cut. A cut in the original derivation could turn into a snip or be eliminated entirely, based on the eligibility of the cut formula in the two premises after appeals to the induction hypothesis. ◀

▶ **Theorem 7** (Subformula Property in Sax). *All formulas in a cut-free derivation of $\Gamma \vdash x : A$ are subformulas of formulas in Γ or A.*

Proof. We generalize the induction hypothesis to include:

> For any eligible antecedent or succedent $y^* : B$ in any sequent in the derivation, B is a subformula of at least one of the remaining (ineligible) formulas in the sequent.

Then we proceed by induction over the structure of the derivation. For a snip, we use the eligibility requirement and the second part of the induction hypothesis to conclude that the cut formula is a subformula in both premises. Also, the second part of the induction hypothesis is manifestly true in all axioms and propagated by all the rules. ◀

4 Some Example Derivations

As an example, we show that $A \otimes B \dashv\vdash A \mathbin{\&} B$. These proofs will have some interesting computational content examined in Appendix B. The first proof has three axioms and two uses of snip.

$$\dfrac{\dfrac{}{x : A \mathbin{\&} B \vdash y^* : A}\,{\&}L_1^0 \qquad \dfrac{\dfrac{}{x : A \mathbin{\&} B \vdash z^* : B}\,{\&}L_2^0 \quad \dfrac{}{x^0 : A \mathbin{\&} B, y^* : A, z^* : B \vdash w : A \otimes B}\,{\otimes}R^0}{x : A \mathbin{\&} B, y^* : A \vdash w : A \otimes B}\,\mathsf{Snip}}{x : A \mathbin{\&} B \vdash w : A \otimes B}\,\mathsf{Snip}$$

We have annotated the proof with eligibility information and notice that in both snips it so happens the variables are eligible on both sides. This proof is cut-free according to our criterion since it only contains snips and not general cuts. The following proof for the other direction is also cut-free, but contains only rules from the usual sequent calculus.

$$\dfrac{\dfrac{\dfrac{}{x^0 : A \otimes B, y : A, z^0 : B \vdash u : A}\,\mathsf{Id}_A \qquad \dfrac{}{x^0 : A \otimes B, y^0 : A, z : B \vdash v : B}\,\mathsf{Id}_B}{x^0 : A \otimes B, y : A, z : B \vdash w : A \mathbin{\&} B}\,{\&}R}{x : A \otimes B \vdash w : A \mathbin{\&} B}\,{\otimes}L$$

As a final example, consider (this portion of) the proof of $(A \supset C) \mathbin{\&} (B \supset C) \vdash (A \vee B) \supset C$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p : (A \supset C) \mathbin{\&} (B \supset C) \vdash r^* : A \supset C}\,{\&}L_1^0 \quad \dfrac{}{r : A \supset C, x^* : A \vdash z^* : C}\,{\supset}L^0}{p : (A \supset C) \mathbin{\&} (B \supset C), x^* : A \vdash z^* : C}\,\mathsf{Snip} \qquad \vdots}{p : (A \supset C) \mathbin{\&} (B \supset C), s : A \vee B \vdash z^* : C}\,{\vee}L}{p : (A \supset C) \mathbin{\&} (B \supset C) \vdash q : (A \vee B) \supset C}\,{\supset}R$$

5 A Shared Memory Interpretation

The key idea behind the shared memory interpretation is that at runtime, variables will be substituted by *addresses in shared memory*. Moreover, a sequent

$$x_1 : A_1, \ldots, x_n : A_n \vdash z : C$$

defines the interface to a process P that *reads* from addresses $x_1, \ldots, x_n$ and *writes* to address z. The types of the variables define the shape of the contents of memory at the given address. Once z has been written to, the process P terminates because it has completed its task. We sometimes refer to $x_1, \ldots, x_n$ as the *sources* and z as the *destination* for P.

True to the Curry-Howard interpretation, P can be read off from the derivation of the sequent. We incorporate a process expression P into the judgment by writing

$$x_1 : A_1, \ldots, x_n : A_n \vdash P :: (z : C)$$

Now, the sequent can be seen as a typing judgment for the process expression P.

Cut. To understand the operational behavior of the processes assigned to sequents, we have to study the cut reductions. We begin with the rule of cut itself, without distinguishing snip as a special case. Cut is a first-class rule (unlike in the proof of admissibility of cut), so it has a corresponding process. This is because cut reduction corresponds to computation, so if we did not have cut we would have no computation.

$$\dfrac{\Gamma \vdash P :: (x : A) \qquad \Gamma, x : A \vdash Q :: (z : C)}{\Gamma \vdash (x \leftarrow P \,;\, Q) :: (z : C)}\,\mathsf{Cut}_A$$

A process executing $x \leftarrow P \,;\, Q$ will allocate a new cell in memory with address a, then spawn a new process $[a/x]P$ (which will write a), and continue as $[a/x]Q$ (which may read from a). Reading from a will be an act of *synchronization*, because $[a/x]Q$ cannot read from a until the value has been written by $[a/x]P$.

This is the only point in the dynamics where a new memory cell is allocated. Initially, it is shared between two processes, $[a/x]P$ and $[a/x]Q$. However, we also notice that Γ, in accordance with the usual presentation of (nonlinear) intuitionistic sequent calculus, is propagated to both premises. Dynamically, this means any cell with an address in Γ is accessible to both new processes. On the other hand, the succedent of a sequent is always a singleton, which leads us to the conclusion:

> *A cell with address a will be written by one distinguished process and may be read by many different processes.*

This observation will have consequences when we consider other rules.

Identity. The rule of identity has a straightforward operational interpretation.

$$\frac{}{\Gamma, x : A \vdash (y^W \leftarrow x^R) :: (y : A)}\ \mathsf{Id}_A$$

The process we assign reads from x and writes to y which amounts to just copying the value at address x to memory at address y. After it has written to y it terminates. The superscripts W and R would presumably not be part of the concrete syntax of a language, but remind us that this process reads from x and writes to y.

If we examine the cut reductions _/Id and Id/_ in the proof of Theorem 5 (shown in Appendix A) we see that this is a considerable restriction of the general reduction rules. This exemplifies a common phenomenon when we relate pure proof theory to computation: some rules of cut reduction may be entirely dropped (such as the so-called permutative reductions), while others are restricted to superimpose a particular strategy on the general notion of reduction.

Logical Rules. The general interpretation of the left and right rules is:

> *Process expressions assigned to right rules will* write *to memory while expressions assigned to left rules will* read *from memory.*

The question in each case is what to write to or read from memory, and how to subsequently continue execution. We will examine this for each connective in turn.

Positive Conjunction. We start with the positive conjunction $A \otimes B$ because it is a little easier to understand than implication. As one might expect given the general guideline, the right rule should write a pair to memory, and it does!

$$\frac{}{\Gamma^0, x^* : A, y^* : B \vdash z^W.\langle x, y\rangle :: (z : A \otimes B)}\ {\otimes}R^0$$

The expression $z^W.\langle x, y\rangle$ writes the pair $\langle x, y\rangle$ to the cell at location z and terminates. The superscript W is there to remind us that we write to the cell z. No other cell is written to or read from. It may also be helpful to directly think of $z^W.\langle x, y\rangle$ as a representation of the memory cell z with contents $\langle x, y\rangle$. Note that the value $\langle x, y\rangle$ itself just contains two addresses x and y, not complex terms. The cells with these addresses may still be empty when we write the pair to z, which allows for a high degree of parallelism.

Conversely, the expression assigned to the rule $\otimes L$

$$\frac{\Gamma, z : A \otimes B, x : A, y : B \vdash P :: (w : C)}{\Gamma, z : A \otimes B \vdash \textbf{case}\ z^R\ (\langle x, y \rangle \Rightarrow P) :: (w : C)}\ \otimes L$$

reads such a pair from memory at location z, matches, it against $\langle x, y\rangle$ to extract the components (say addresses a and b) and continues with $[a/x, b/y]P$. The cell z is persistent, so it may be read again later, either by this process or by another one. Again, the cells at addresses x and y may not yet have been filled, but we can nevertheless extract and manipulate their addresses.

The principal cut reduction of $\otimes R^0$ against $\otimes L$, expressed on processes, becomes

$$z \leftarrow (z^W.\langle a, b\rangle) \mathrel{;} \textbf{case}\ z^R\ (\langle x, y\rangle \Rightarrow P) \quad \longrightarrow \quad [a/x, b/y]P$$

which is precisely the intended operational semantics.

Implication. Implication represents somewhat of a challenge to intuition, and is perhaps the reason that this form of sequent calculus and its shared memory interpretation has been overlooked. We start with the left rule $\supset L^0$ which, according to our guiding principle, should read from memory.

$$\frac{}{\Gamma, y^* : A, x : A \supset B \vdash x^R.\langle y, z\rangle :: (z^* : B)}\ \supset L^0$$

The process expression $x^R.\langle y, z\rangle$ should read a value of type $A \supset B$ from location x and pass it the pair y and z. But what does this pair represent? y is the (usual) argument to the function, having type A. And z is the *destination for the result of the function.* As such, every function takes an additional argument. This is reminiscent of continuation-passing style [28] except that instead of passing a *continuation to accept the function's result* we pass a *destination address to store the function's result.*

Note that we have reused the syntax for pairs, except that the process assigned here *reads* from x. It is economical for the $\supset R$ rule to also reuse the same syntax to describe the augmented functions which we call *continuations.*

$$\frac{\Gamma, y : A \vdash P :: (z : B)}{\Gamma \vdash \textbf{case}\ x^W\ (\langle y, z\rangle \Rightarrow P) :: (x : A \supset B)}\ \supset R$$

The process **case** c^W $(\langle y, z\rangle \Rightarrow P)$ writes the continuation $(\langle y, z\rangle \Rightarrow P)$ to the cell at address c and terminates.

The cut reduction for $\supset R$ against $\supset L^0$, expressed directly on processes, is symmetric to the reduction for $\otimes R^0$ against $\otimes L$:

$$z \leftarrow (\textbf{case}\ z^W\ (\langle x, y\rangle \Rightarrow P)) \mathrel{;} z^R.\langle a, b\rangle \quad \longrightarrow \quad [a/x, b/y]P$$

Disjunction and Negative Conjunction. Just like implication and positive conjunction form a symmetric pair of expressions, reversing the role of read and write, so do disjunction and negative conjunction. The rules can be found in Figure 4, which summarizes all of the rules for SAX. We show here the reductions for $A \mathbin{\&} B$, and the reductions for $A \vee B$ are symmetric, as they were for $A \supset B$ and $A \otimes B$.

$$\begin{array}{lcl} z \leftarrow (\textbf{case}\ z^W\ (\pi_1(x) \Rightarrow P \mid \pi_2(y) \Rightarrow Q)) \mathrel{;} z^R.\pi_1(a) & \longrightarrow & [a/x]P \\ z \leftarrow (\textbf{case}\ z^W\ (\pi_1(x) \Rightarrow P \mid \pi_2(y) \Rightarrow Q)) \mathrel{;} z^R.\pi_2(b) & \longrightarrow & [b/x]Q \end{array}$$

$$\frac{}{\Gamma, x : A \vdash (y^W \leftarrow x^R) :: (y : A)}\ \mathsf{Id}_A \qquad \frac{\Gamma \vdash P :: (x : A) \quad \Gamma, x : A \vdash Q :: (z : C)}{\Gamma \vdash (x \leftarrow P \,;\, Q) :: (z : C)}\ \mathsf{Cut}_A$$

$$\frac{\Gamma, y : A \vdash P :: (z : B)}{\Gamma \vdash \textbf{case}\ x^W\ (\langle y, z\rangle \Rightarrow P) :: (x : A \supset B)}\ {\supset}R \qquad \frac{}{\Gamma, y^* : A, x : A \supset B \vdash (x^R.\langle y, z\rangle) :: (z^* : B)}\ {\supset}L^0$$

$$\frac{}{\Gamma^0, y^* : A, z^* : B \vdash x^W.\langle y, z\rangle :: (x : A \otimes B)}\ {\otimes}R^0 \qquad \frac{\Gamma, x : A \otimes B, y : A, z : B \vdash P :: (w : C)}{\Gamma, x : A \otimes B \vdash \textbf{case}\ x^R\ (\langle y, z\rangle \Rightarrow P) :: (w : C)}\ {\otimes}L$$

$$\frac{}{\Gamma^0 \vdash x^W.\langle\,\rangle :: (x : \mathbf{1})}\ \mathbf{1}R^0 \qquad \frac{\Gamma, x : \mathbf{1} \vdash P :: (w : C)}{\Gamma, x : \mathbf{1} \vdash \textbf{case}\ x^R\ (\langle\,\rangle \Rightarrow P) :: (w : C)}\ \mathbf{1}L$$

$$\frac{\Gamma \vdash P :: (y : A) \quad \Gamma \vdash Q :: (z : B)}{\Gamma \vdash \textbf{case}\ x^W\ (\pi_1(y) \Rightarrow P \mid \pi_2(z) \Rightarrow Q) :: (x : A \mathbin{\&} B)}\ {\&}R \qquad \begin{array}{c} \dfrac{}{\Gamma^0, x : A \mathbin{\&} B \vdash x^R.\pi_1(y) :: (y^* : A)}\ {\&}L_1^0 \\[2ex] \dfrac{}{\Gamma^0, x : A \mathbin{\&} B \vdash x^R.\pi_2(z) :: (z^* : B)}\ {\&}L_2^0 \end{array}$$

$$\frac{}{\Gamma^0, x^* : A \vdash z^W.\pi_1(x) :: (z : A \vee B)}\ {\vee}R_1^0$$

$$\frac{}{\Gamma^0, y^* : B \vdash z^W.\pi_2(y) :: (z : A \vee B)}\ {\vee}R_2^0$$

$$\frac{\Gamma, z : A \vee B, x : A \vdash P :: (w : C) \quad \Gamma, z : A \vee B, y : B \vdash Q :: (w : C)}{\Gamma, z : A \vee B \vdash \textbf{case}\ z^R\ (\pi_1(x) \Rightarrow P \mid \pi_2(y) \Rightarrow Q) :: (w : C)}\ {\vee}L$$

Figure 4 Process Expression Assignment for Sax.

Processes	P	$::= x \leftarrow P \,;\, Q$	allocate a, spawn $[a/x]P$, continue as $[a/x]Q$
		$\mid\ x^W \leftarrow y^R$	copy contents of cell y into cell x
		$\mid\ x^W.V$	write V to cell x
		$\mid\ \textbf{case}\ x^R\ K$	read value V from cell x and pass it to K
		$\mid\ \textbf{case}\ x^W\ K$	write continuation K to cell x
		$\mid\ x^R.V$	read continuation K from x and pass V to it
Continuations	K	$::= (\langle x, y\rangle \Rightarrow P)$	$({\otimes}L, {\supset}R)$
		$\mid\ (\langle\,\rangle \Rightarrow P)$	$(\mathbf{1}L)$
		$\mid\ (\pi_1(x) \Rightarrow P \mid \pi_2(x) \Rightarrow Q)$	$({\&}R, {\vee}L)$
Values	V	$::= \langle x, y\rangle$	$({\otimes}R^0, {\supset}L^0)$
		$\mid\ \langle\,\rangle$	$(\mathbf{1}R^0)$
		$\mid\ \pi_1(x) \mid \pi_2(x)$	$({\&}L^0, {\vee}R^0)$
Cell Contents	W	$::= V \mid K$	

Figure 5 The Grammar for Sax Process Expressions.

5.1 Values and Continuations, Cells and Processes

The language of process expressions is now complete, but the immediate cut reductions do not yet fully capture the intended semantics. We first refactor the definition of our language slightly, by separating small values V from continuations K. Figure 5 also now shows what the new axioms of Sax represent: *values*. Meanwhile, the unchanged invertible rules represent *continuations*.

A key operation is in the semantics is passing a value to a continuation, which we write as $V \triangleright K$. It is defined by the following clauses:

$$\begin{array}{rcl@{\quad}c@{\quad}l}
\langle a, b\rangle & \triangleright & (\langle x, y\rangle \Rightarrow P) & = & [a/x, b/y]P \\
\langle\,\rangle & \triangleright & (\langle\,\rangle \Rightarrow P) & = & P \\
\pi_1(a) & \triangleright & (\pi_1(x) \Rightarrow P \mid \pi_2(y) \Rightarrow Q) & = & [a/x]P \\
\pi_2(b) & \triangleright & (\pi_1(x) \Rightarrow P \mid \pi_2(y) \Rightarrow Q) & = & [b/y]Q
\end{array}$$

The difficulty with the raw cut reductions in the presence of contraction (whether implicit as in G_3 and Sax, or with an explicit rule) is that some of them duplicate derivations. Instead, we would like them to be *shared*, which is exactly what the notion of shared memory allows us to do. One can feed this back into proof theory using the notion of multicut [24]. Here, we represent multiple cuts, and simultaneous cuts of one derivation against multiple others with the concept of a *configuration*. Such a configuration can be unravelled back into ordinary cuts (and therefore ordinary derivations), but at the cost of duplicating derivations.

A *configuration* consists of allocated memory cells (some of which may have been written to and some not) and executing processes. We present it as a substructural operational semantics (SSOS) [5, 27] in the form of multiset rewriting rules [6], using the following semantic objects.

$\mathsf{proc}\ c\ P$	process P with destination c
$\mathsf{cell}\ c\ \_$	memory cell c, allocated but not yet written
$!\mathsf{cell}\ c\ W$	cell c with contents W

In a configuration, a process $\mathsf{proc}\ c\ P$ is always paired with an allocated but not yet written $\mathsf{cell}\ c\ \_$. We also have cells $!\mathsf{cell}\ c\ W$ that have been written already. They are *persistent* (indicated by the exclamation mark) since they may be read multiple times but cannot be written again. In the multiset rewriting rules, a left-hand side of the form $!\phi$ will remain in the configuration, while objects ψ are removed and replaced by the objects on the right-hand side of the rule. All addresses c with objects $\mathsf{cell}\ c\ \_$ or $!\mathsf{cell}\ c\ W$ in a given configuration must be distinct, that is, no two cells in a configuration may share the same address.

The transitions in Figure 6 are multiset rewriting rules describing the dynamics of configurations. They can be applied to any subconfiguration, which induces a form of concurrency. However, the rules are confluent (see Theorem 10) so the result of reducing a configuration via these rules is ultimately deterministic, modulo the names of freshly introduced variables.

$$\begin{array}{lcll}
\mathsf{proc}\ c\ (x \leftarrow P \;;\, Q) & \longrightarrow & \mathsf{proc}\ a\ ([a/x]P), \mathsf{cell}\ a\ \_, \mathsf{proc}\ c\ ([a/x]Q) & (a \text{ fresh}) \\
!\mathsf{cell}\ b\ W, \mathsf{proc}\ a\ (a^W \leftarrow b^R), \mathsf{cell}\ a\ \_ & \longrightarrow & !\mathsf{cell}\ a\ W & \\
\mathsf{proc}\ a\ (\mathbf{case}\ a^W\ K), \mathsf{cell}\ a\ \_ & \longrightarrow & !\mathsf{cell}\ a\ K & \\
!\mathsf{cell}\ a\ K, \mathsf{proc}\ c\ (a^R.V) & \longrightarrow & \mathsf{proc}\ c\ (V \triangleright K) & \\
 & & & \\
!\mathsf{cell}\ a\ V, \mathsf{proc}\ a\ (\mathbf{case}\ a^R\ K) & \longrightarrow & \mathsf{proc}\ a\ (V \triangleright K) & \\
\mathsf{proc}\ a\ (a.V), \mathsf{cell}\ a\ \_ & \longrightarrow & !\mathsf{cell}\ a\ V &
\end{array}$$

Figure 6 Dynamic Semantics of Configurations.

$$\frac{}{\Gamma \vdash (\cdot) :: \Gamma}\ \mathsf{Empty} \qquad \frac{\Gamma_0 \vdash \mathcal{C}_1 :: \Gamma_1 \quad \Gamma_1 \vdash \mathcal{C}_2 :: \Gamma_2}{\Gamma_0 \vdash (\mathcal{C}_1, \mathcal{C}_2) :: \Gamma_2}\ \mathsf{Join}$$

$$\frac{\Gamma \vdash P :: (a : A)}{\Gamma \vdash (\mathsf{proc}\ a\ P, \mathsf{cell}\ a\ \_) :: (\Gamma, a : A)}\ \mathsf{Proc}$$

$$\frac{\Gamma \vdash a^W.V :: (a : A)}{\Gamma \vdash \mathsf{!cell}\ a\ V :: (\Gamma, a : A)}\ \mathsf{Cell}_V \qquad \frac{\Gamma \vdash \mathbf{case}\ a^W\ K :: (a : A)}{\Gamma \vdash \mathsf{!cell}\ a\ K :: (\Gamma, a : A)}\ \mathsf{Cell}_K$$

Figure 7 Typing Rules for Configurations.

5.2 Typing Configurations

Configurations, like cells and processes, are runtime artifacts which can nevertheless be typed and also put into correspondence with the sequent calculus. First, typing. We have

$$\begin{array}{llll} \text{Configurations} & \mathcal{C} ::= & \mathsf{proc}\ a\ P, \mathsf{cell}\ a\ \_ & \text{process with destination } a \\ & \quad\ \mid & \mathsf{!cell}\ a\ W & \text{cell with contents } W \\ & \quad\ \mid & \cdot & \text{empty configuration} \\ & \quad\ \mid & \mathcal{C}_1, \mathcal{C}_2 & \text{joining configurations} \end{array}$$

The concatenation operator "," for configurations is commutative and associative with unit "·", which makes it a suitable basis for multiset rewriting. However, a typing derivation for a configuration imposes an order by requiring that the writer of a cell a comes before all the readers of the cell. We include the rules for typing the contents of cells below those for typing configurations in Figure 7.

We can now state and prove several key results regarding our dynamics.

▶ **Theorem 8** (Preservation). *If $\Gamma_0 \vdash \mathcal{C} :: \Gamma$ and $\mathcal{C} \longrightarrow \mathcal{C}'$ then $\Gamma_0 \vdash \mathcal{C} :: \Gamma'$ for some $\Gamma' \supseteq \Gamma$.*

Proof. A first key property is that if $\Gamma_0 \vdash \mathcal{C} :: \Gamma$ then $\Gamma_0 \subseteq \Gamma$, which is easily proved by induction on the typing derivation. Therefore, a cell with address a and any process tasked with writing to a (which need not exist if the cell has already been filled) will always come to the left of any reader of a. This and the persistence of $\mathsf{!cell}\ a\ W$ easily yield preservation by induction on the typing derivation and inversion on the typing of processes and cell contents. ◀

We say a configuration is *final* if it consists only of cells $\mathsf{!cell}\ a\ W$. In other words, there are no longer any running processes. We prove progress for *closed* configurations $(\cdot \vdash \mathcal{C} :: \Gamma)$ that do not depend on any undefined addresses.

Note that the typing derivation for a configuration is associative with respect to rule Join and unit Empty. This has two useful consequences. First, this provides a simple way to reconstruct a proof tree from a (well-typed) configuration – an empty configuration yields an empty tree, the Proc and Cell rules are base cases, each becoming the proof tree in the premise, and the Join rule simply cuts together the proof trees on the left and the right. Second, because of the associativity, we may perform induction where we isolate the rightmost cell or process and apply the inductive hypothesis to the remaining configuration to the left.

▶ **Theorem 9** (Progress). *If $\cdot \vdash \mathcal{C} :: \Gamma$ then either $\mathcal{C}$ is final or $\mathcal{C} \longrightarrow \mathcal{C}'$ for some $\mathcal{C}'$.*

Proof. By right-to-left induction over the typing derivation for $\mathcal{C}$.
Case: The rightmost rule is Cell, so $\mathcal{C} = (\mathcal{C}_1, \mathsf{!cell}\ a\ W)$. By induction hypothesis, either $\mathcal{C}_1$ is final (and so is then $\mathcal{C}$) or $\mathcal{C}_1 \longrightarrow \mathcal{C}_1'$ and therefore also $\mathcal{C} \longrightarrow \mathcal{C}_1', \mathsf{!cell}\ a\ W$.

Case: The rightmost rule is Proc, so $\mathcal{C} = (\mathcal{C}_1, \mathsf{proc}\ a\ P, \mathsf{cell}\ a\ \_)$. If $\mathcal{C}_1 \longrightarrow \mathcal{C}_1'$ then also $\mathcal{C} \longrightarrow (\mathcal{C}_1', \mathsf{proc}\ a\ P, \mathsf{cell}\ a\ \_)$. If $\mathcal{C}_1$ is final then we distinguish cases on P. If P is alloc/spawn, copy, or a process that writes, then $\mathcal{C} \longrightarrow \mathcal{C}'$ for some $\mathcal{C}'$. If P is a process that reads, then we apply inversion on the typing derivations of the cell a and the process P to show that a reduction is once again possible. ◀

We say $\mathcal{C}_1 \sim \mathcal{C}_2$ if there is a renaming ρ such that $\rho\mathcal{C}_1 = \mathcal{C}_2$.

▶ **Theorem 10** (Diamond Property). *Assume $\Delta \vdash \mathcal{C} :: \Gamma$. If $\mathcal{C} \longrightarrow \mathcal{C}_1$ and $\mathcal{C} \longrightarrow \mathcal{C}_2$ such that $\mathcal{C}_1 \not\sim \mathcal{C}_2$. Then there exist $\mathcal{C}_1'$ and $\mathcal{C}_2'$ such that $\mathcal{C}_1 \longrightarrow \mathcal{C}_1'$ and $\mathcal{C}_2 \longrightarrow \mathcal{C}_2'$ with $\mathcal{C}_1' \sim \mathcal{C}_2'$.*

Proof. The proof is straightforward by cases. There are no critical pairs involving ephemeral (that is, non-persistent) objects in the left-hand sides. ◀

As usual, confluence of multistep reduction follows by two standard inductions from the diamond property.

We now return to the distinction between general cuts and snips, which we introduced in order to establish cut elimination. First, we note that the transition rule for cut applies equally whether the cut $x \leftarrow P\ ;\ Q$ is a snip or not. In order to discuss the other rules, we define that an occurrence of an *address a is eligible* if it is eligible in the typing derivation according to the rules in Figure 7. This coincides with saying that the corresponding variable would be eligible if we unwound the configuration into a collection of proofs. We notice that in an object $!\mathsf{cell}\ a\ W$ the address a is never eligible because it labels the principal formula of an inference. Furthermore, none of the reductions besides cut involve an eligible address a, since they all label either an application of identity or the principal proposition of an inference.

It remains to characterize final configurations, that is, those consisting only of cells. We might at first suspect that all remaining cuts are snips, but that is not true because the dynamics does not reduce continuations K. This reflects a common difference between pure proof theory (where we show full normalization or cut elimination) and the dynamics of programming languages (where we do not evaluate under abstractions). We call addresses that occur in values V *observable* and those that occur in continuations K *hidden*.

▶ **Theorem 11** (Observable Addresses). *All observable addresses in a well-typed final configuration are eligible.*

Proof. By inversion on the typing of cells containing values V. ◀

From this, we obtain a simple corollary for *purely positive types*. In a functional language, *values* of purely positive type are *observable* in their entirety, without any functions or closures with hidden structure. Here, such values are allocated and distributed into memory cells, but nevertheless observable in their entirety.

$$\begin{array}{lll} \text{Purely Positive Type} & A^+ & ::= \mathbf{1} \mid A_1^+ \otimes A_2^+ \mid A_1^+ \vee A_2^+ \\ \text{Purely Positive Context} & \Gamma^+ & ::= \cdot \mid \Gamma^+, x : A^+ \end{array}$$

We then have the following characterization.

▶ **Corollary 12** (Final Configurations of Purely Positive Type). *All cells in a final configuration $\cdot \vdash \mathcal{F} :: \Gamma^+$ have the form $!\mathsf{cell}\ a\ V$, and therefore all addresses in V are observable and eligible.*

This means if we reconstitute a final configuration into a collection of proofs (one for each $a : A^+$ in Γ^+) by introducing cuts, then all these cuts will be snips.

6 Termination

We prove termination by means of a logical relation (predicate), which we first define for closed configurations (with no free addresses), and then extend to open configurations. The definition and proof incorporate ideas from standard logical relations for natural deduction into those of Pérez et al. [20] in the context of synchronous message-passing concurrency.

▶ **Definition 13.** *We define two predicates on configurations, $[\![a : A]\!]$ and $[a : A]$, by mutual induction on the structure of the type A.*

1. $\mathcal{C} \in [\![a : A]\!]$ *iff* $\mathcal{C} \longrightarrow^* \mathcal{F}$ *where* $\mathcal{F}$ *is final and* $\mathcal{F} \in [a : A]$.

2.a $\mathcal{F} \in [a : B \otimes C]$ *iff* $\mathcal{F} = \mathcal{F}', !\mathsf{cell}\ a\ \langle b, c\rangle$, $\mathcal{F}' \in [b : B]$, *and* $\mathcal{F}' \in [c : C]$.
2.b $\mathcal{F} \in [a : \mathbf{1}]$ *iff* $\mathcal{F} = \mathcal{F}', !\mathsf{cell}\ a\ \langle\rangle$.
2.c $\mathcal{F} \in [a : B \supset C]$ *iff for all* $\mathcal{F}'$ *such that* $\mathcal{F}, \mathcal{F}' \in [b : B]$,
we have $\mathcal{F}, \mathcal{F}', (\mathsf{proc}\ c\ (a.\langle b, c\rangle), \mathsf{cell}\ c\ \_) \in [\![c : C]\!]$.
2.d $\mathcal{F} \in [a : B \mathbin{\&} C]$ *iff both* $\mathcal{F}, (\mathsf{proc}\ b\ (a.\pi_1(b)), \mathsf{cell}\ b\ \_) \in [\![b : B]\!]$
and $\mathcal{F}, (\mathsf{proc}\ c\ (a.\pi_2(c)), \mathsf{cell}\ c\ \_) \in [\![c : C]\!]$.
2.e $\mathcal{F} \in [a : B \vee C]$ *iff either* $\mathcal{F} = \mathcal{F}_B, !\mathsf{cell}\ a\ (\pi_1(b))$ *with* $\mathcal{F}_B \in [b : B]$
or $\mathcal{F} = \mathcal{F}_C, !\mathsf{cell}\ a\ (\pi_2(c))$ *with* $\mathcal{F}_C \in [c : C]$.

We can then extend this definition to specify the behavior of a configuration providing more than one address.

▶ **Definition 14.** *We define* $\mathcal{C} \in [\![\Gamma]\!]$ *iff for all* $a : A \in \Gamma$, $\mathcal{C} \in [\![a : A]\!]$.

Finally, using the above definitions, we can extend the predicate to open configurations.

▶ **Definition 15.** *We define* $\mathcal{C} \in [\![\Gamma \vdash a : A]\!]$ *iff for all* $\mathcal{C}' \in [\![\Gamma]\!]$, $\mathcal{C}', \mathcal{C} \in [\![a : A]\!]$. *Note that* $\mathcal{C} \in [\![\cdot \vdash a : A]\!]$ *iff* $\mathcal{C} \in [\![a : A]\!]$.

Given these definitions, we have three key lemmas for the proof of termination.

▶ **Lemma 16** (Weakening). *If* $\mathcal{C} \in [\![a : A]\!]$, *then for all* b, B, *and* $\mathcal{C}' \in [\![b : B]\!]$ *we have* $\mathcal{C}', \mathcal{C} \in [\![a : A]\!]$.

As a corollary, if $\mathcal{C} \in [\![a : A]\!]$ then $\mathcal{C} \in [\![\Gamma \vdash a : A]\!]$ for any Γ.

▶ **Lemma 17** (Closure). *If* $\mathcal{C} \longrightarrow^* \mathcal{C}'$ *then* $\mathcal{C} \in [\![\Gamma \vdash a : A]\!]$ *iff* $\mathcal{C}' \in [\![\Gamma \vdash a : A]\!]$.

▶ **Lemma 18** (Inversion). *If a final* $\mathcal{F} \in [a : A]$ *then* $\mathcal{F} = \mathcal{F}', !\mathsf{cell}\ a\ W$ *for some* $\mathcal{F}'$ *and* W.

Finally, using these lemmas, we can go on to prove the main theorem of this section, which state that all well-typed configurations satisfy the termination predicate. We can apply this to any of the succedents in the general typing judgment for configurations.

▶ **Theorem 19.** *If* $\Gamma \vdash \mathcal{C} :: \Delta$, *then* $\mathcal{C} \in [\![\Gamma \vdash a : A]\!]$ *for every* $(a : A) \in \Delta$.

Proof. This proof proceeds by induction on the multiset $\mathcal{M}$ of derivations $\Gamma' \vdash P :: (b : B)$ and $\Gamma' \vdash W : B$ used in the derivation of $\Gamma \vdash \mathcal{C} :: \Delta$. Derivations are ordered in the standard way, and we use the multiset ordering derived from this as the basis of our induction. As noted in the proof of Theorem 9, the typing derivation for a configuration induces an order on that configuration and we can examine it from right to left. ◀

A corollary of this theorem is that any closed well-typed configuration terminates in a final configuration.

7 Conclusion

We defined Sax, a new hybrid form of sequent calculus in which right rules for positive connectives and left rules for negative connectives are replaced by axioms, that is, inference rules with no premises. This calculus satisfies a modified cut elimination theorem in which certain analytic cuts which preserve the subformula property are allowed. We showed how to assign process expression to derivations in Sax and provided a simple shared memory semantics for them: cut allocates memory cells, identity copies contents from one cell to another, processes assigned to right rules write to cells and those assigned to left rules read from them. Cells may be written at most once, but read many times, which means that they provide synchronization points between concurrent processes. This seems quite similar to futures [12], an analogy we have substantiated in an unpublished report [25]. This report does not investigate the proof-theoretic foundations of Sax but further generalizes the type system to *adjoint logic* [26, 18, 23, 24], adds recursive types, and shows that futures can be embedded into an adjoint formulation of Sax. We further conjecture that functional programs can be compiled directly into Sax, where a particular schedule for the resulting concurrent programs corresponds to their sequential execution. This has been developed (without proof) in some unpublished lecture notes [22, Lectures 19–21].

References

1 Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

2 Zena M. Ariola, Aaron Bohannon, and Amr Sabry. Sequent calculi and abstract machines. *ACM Transactions on Programming Languages and Systems*, 31(4):13:1–13:48, May 2009.

3 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, pages 222–236, Paris, France, August 2010. Springer LNCS 6269.

4 Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016. Special Issue on Behavioural Types.

5 Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

6 Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Information and Computation*, 207(10):1044–1077, October 2009.

7 Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *International Conference on Functional Programming (ICFP 2000)*, pages 233–243. ACM Press, September 2000.

8 H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences, U.S.A.*, 20:584–590, 1934.

9 H. B. Curry and R. Feys. *Combinatory Logic.* North-Holland, Amsterdam, 1958.

10 Michael Dummett. *The Logical Basis of Metaphysics.* Harvard University Press, Cambridge, Massachusetts, 1991. The William James Lectures, 1976.

11 Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

12 Robert H. Halstead. Multilisp: A language for parallel symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, October 1985.

13 Hugo Herbelin. A lambda-calculus structure isomorphic to Gentzen-style sequent calculus structure. In L. Pacholski and J. Tiuryn, editors, *8th International Workshop on Computer Science Logic*, pages 61–75, Kazimierz, Poland, September 1994. Springer LNCS 933.

14 W. A. Howard. The formulae-as-types notion of construction. Unpublished note. An annotated version appeared in: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 479–490, Academic Press (1980), 1969.

15 Stephen Cole Kleene. *Introduction to Metamathematics.* North-Holland, 1952.

16 Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation*, 19(4):377–414, 2006.

17 Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, November 2009.

18 Daniel R. Licata and Michael Shulman. Adjoint logic with a 2-category of modes. In *International Symposium on Logical Foundations of Computer Science (LFCS)*, pages 219–235. Springer LNCS 9537, January 2016.

19 Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Notes for three lectures given in Siena, Italy. Published in *Nordic Journal of Philosophical Logic*, 1(1):11-60, 1996, April 1983.

20 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.

21 Frank Pfenning. Structural cut elimination I. Intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, March 2000.

22 Frank Pfenning. Types and programming languages. Lecture Notes, 2019. URL: `http://www.cs.cmu.edu/~fp/courses/15814-f19`.

23 Klaas Pruiksma, William Chargin, Frank Pfenning, and Jason Reed. Adjoint logic. Unpublished manuscript, April 2018. URL: `http://www.cs.cmu.edu/~fp/papers/adjoint18b.pdf`.

24 Klaas Pruiksma and Frank Pfenning. A message-passing interpretation of adjoint logic. In F. Martins and D. Orchard, editors, *Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software (PLACES)*, pages 60–79, Prague, Czech Republic, April 2019. EPTCS 291.

25 Klaas Pruiksma and Frank Pfenning. Back to futures. *CoRR*, abs/2002.04607, February 2020. URL: `http://arxiv.org/abs/2002.04607`.

26 Jason Reed. A judgmental deconstruction of modal logic. Unpublished manuscript, May 2009. URL: `http://www.cs.cmu.edu/~jcreed/papers/jdml2.pdf`.

27 Robert J. Simmons. *Substructural Logical Specifications.* PhD thesis, Carnegie Mellon University, November 2012. Available as Technical Report CMU-CS-12-142.

28 Gerald Jay Sussman and Guy L. Steele. Scheme: An intepreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, December 1998. Reprint of the MIT AI Memo 349:19, December 1975.

29 Philip Wadler. Propositions as sessions. In *Proceedings of the 17th International Conference on Functional Programming*, ICFP 2012, pages 273–286, Copenhagen, Denmark, September 2012. ACM Press.

A Some Cases in the Proof of Cut Admissibility

Proof of Theorem 5. We consider various classes of cases. These cases are not mutually exclusive, which means that the algorithm for the construction of $\mathcal{F}$ from $\mathcal{D}$ and $\mathcal{E}$ induced by this constructive proof is naturally nondeterministic. We restrict ourselves to implication only, but show all the cases relevant to this fragment. The cases for other connectives follow similar patterns.

The first two cases show the admissibility of cut by building a snip when one (or both) of the cut formulas are eligible.

Case: $*/\_$ (which also covers the cases $\supset L^0/\_$)

$$\dfrac{\overset{\mathcal{D}}{\Gamma \vdash x^* : A} \quad \overset{\mathcal{E}}{\Gamma, x : A \vdash z : C}}{\Gamma \vdash z : C}\,\mathsf{Cut}_A \quad \Longrightarrow \quad \dfrac{\overset{\mathcal{D}}{\Gamma \vdash x^* : A} \quad \overset{\mathcal{E}}{\Gamma, x : A \vdash z : C}}{\Gamma \vdash z : C}\,\mathsf{Snip}^1_A$$

Case: $\_/*$ (which also covers a subcase of $\_/\supset^0$)

$$\dfrac{\overset{\mathcal{D}}{\Gamma \vdash x : A} \quad \overset{\mathcal{E}}{\Gamma, x^* : A \vdash z : C}}{\Gamma \vdash z : C}\,\mathsf{Cut}_A \quad \Longrightarrow \quad \dfrac{\overset{\mathcal{D}}{\Gamma \vdash x : A} \quad \overset{\mathcal{E}}{\Gamma, x^* : A \vdash z : C}}{\Gamma \vdash z : C}\,\mathsf{Snip}^2_A$$

The next case covers a cut against an irrelevant antecedent. This case is not strictly necessary and could be replaced by several more specialized ones if desired.

Case: $\_/0$ (also covers $\_/\supset L^0$ and $\_/\mathsf{Id}$ where the cut formula is a side formula in $\mathcal{E}$)

$$\dfrac{\overset{\mathcal{D}}{\Gamma \vdash x : A} \quad \overset{\mathcal{E}}{\Gamma, x^0 : A \vdash z : C}}{\Gamma \vdash z : C}\,\mathsf{Cut}_A \quad \Longrightarrow \quad \overset{\mathcal{E} - \{x^0 : A\}}{\Gamma \vdash z : C}$$

In the next two cases the cut formula is the principal formula of an identity, either in $\mathcal{D}$ or $\mathcal{E}$.

Case: $\_/\mathsf{Id}$

$$\dfrac{\overset{\mathcal{D}}{\Gamma \vdash x : A} \quad \dfrac{}{\Gamma, x : A \vdash z : A}\,\mathsf{Id}}{\Gamma \vdash z : A}\,\mathsf{Cut}_A \quad \Longrightarrow \quad \overset{[z/x]\mathcal{D}}{\Gamma \vdash z : A}$$

Next is a case where $z : C$ may be eligible in $\mathcal{F}$ even if it is not in $\mathcal{E}$. Our proof is not concerned with variables that may gain eligibility.

Case: $\mathsf{Id}/\_$

$$\dfrac{\dfrac{}{\Gamma', y : A \vdash x : A}\,\mathsf{id} \quad \overset{\mathcal{E}}{\Gamma', y : A, x : A \vdash z : C}}{\Gamma', y : A \vdash z : C}\,\mathsf{Cut}_A \quad \Longrightarrow \quad \overset{[y/x]\mathcal{E}}{\Gamma', y : A \vdash z : C}$$

The next case is the principal case where the cut formula is inferred in the last inference of both premises of the cut.

Case: $\supset R/\supset L^0$

$$\dfrac{\dfrac{\overset{\mathcal{D}_0}{\Gamma', y : A_1, x_1 : A_1 \vdash x_2 : A_2}}{\Gamma', y : A_1 \vdash x : A_1 \supset A_2}\,\supset R \quad \dfrac{}{\Gamma', y^* : A_1, x : A_1 \supset A_2 \vdash z^* : A_2}\,\supset L^0}{\Gamma', y : A_1 \vdash z : A_2}\,\mathsf{Cut}_{A_1 \supset A_2} \Longrightarrow \overset{[y/x_1, z/x_2]\mathcal{D}_0}{\Gamma', y : A_1 \vdash z : A_2}$$

Note that in this case $y : A_1$ and $z : A_2$ may lose eligibility. However, $A_1 < A_1 \supset A_2$ and $A_2 < A_1 \supset A_2$ so our requirements are satisfied in case they remain relevant.

In the next group of cases, the first derivation $\mathcal{D}$ is arbitrary and the cut formula is a side formula of the last inference in $\mathcal{E}$. Because $\mathcal{E}$ is assumed to be cut-free, we get five cases $\_/\supset R$, $\_/\mathsf{Snip}^1$, $\_/\mathsf{Snip}^2$, $\_/\supset L^0$, and $\_/\mathsf{Id}$, where the last two are already covered by $\_/0$.

Case: $\_/\supset R$

$$\dfrac{\overset{\mathcal{D}}{\Gamma \vdash x : A} \quad \dfrac{\overset{\mathcal{E}_0}{\Gamma, x : A, z_1 : C_1 \vdash z_2 : C_2}}{\Gamma, x : A \vdash z : C_1 \supset C_2}\,\supset R}{\Gamma \vdash z : C_1 \supset C_2}\,\mathsf{Cut}_A \Longrightarrow \dfrac{\dfrac{\overset{\mathcal{D} + \{z_1^0 : C_1\}}{\Gamma, z_1^0 : C_1 \vdash x : A} \quad \overset{\mathcal{E}_0}{\Gamma, x : A, z_1 : C_1 \vdash z_2 : C_2}}{\Gamma, z_1 : C_1 \vdash z_2 : C_2}\,\mathsf{Cut}_A}{\Gamma \vdash z : C_1 \supset C_2}\,\supset R$$

Case: $\_/\mathsf{Snip}^1$ with two subcases.

$$\dfrac{\overset{\mathcal{D}}{\Gamma \vdash x : A} \qquad \dfrac{\overset{\mathcal{E}_1}{\Gamma, x : A \vdash y^* : B} \quad \overset{\mathcal{E}_2}{\Gamma, x : A, y : B \vdash z : C}}{\Gamma, x : A \vdash z : C}\,\mathsf{Snip}^1_B}{\Gamma \vdash z : C}\,\mathsf{Cut}_A \qquad \Longrightarrow \quad \overset{\mathcal{F}}{\Gamma \vdash z : C}$$

where $\mathcal{F}$ is defined in each subcase below from

$$\dfrac{\overset{\mathcal{D}}{\Gamma \vdash x : A} \quad \overset{\mathcal{E}_1}{\Gamma, x : A \vdash y^* : B}}{\Gamma \vdash y : B}\,\mathsf{Cut}_A \quad \Longrightarrow \quad \overset{\mathcal{F}_1}{\Gamma \vdash y : B}$$

and

$$\dfrac{\overset{\mathcal{D} + \{y : B\}}{\Gamma, y : B \vdash x : A} \quad \overset{\mathcal{E}_2}{\Gamma, y : B, x : A \vdash z : C}}{\Gamma, y : B \vdash z : C}\,\mathsf{Cut}_A \quad \Longrightarrow \quad \overset{\mathcal{F}_2}{\Gamma, y : B \vdash z : C}$$

There are two subcases depending on properties of $\mathcal{F}_1$, obtained from the induction hypothesis.

Subcase: y is still eligible in $\mathcal{F}_1$.

Subcase: y is not eligible in $\mathcal{F}_1$ and $B < A$. Note that in this case we can apply the induction hypothesis once more.

These two subcases yield the following two definitions of $\mathcal{F}$, respectively:

$$\mathcal{F} \;=\; \dfrac{\overset{\mathcal{F}_1}{\Gamma \vdash y^* : B} \quad \overset{\mathcal{F}_2}{\Gamma, y : B \vdash z : C}}{\Gamma \vdash z : C}\,\mathsf{Snip}^1 \qquad \mathcal{F} \;=\; \dfrac{\overset{\mathcal{F}_1}{\Gamma \vdash y : B} \quad \overset{\mathcal{F}_2}{\Gamma, y : B \vdash z : C}}{\Gamma \vdash z : C}\,\mathsf{Cut}_B$$

Case: $\_/\mathsf{Snip}^2$ with three subcases.

$$\dfrac{\overset{\mathcal{D}}{\Gamma \vdash x : A} \qquad \dfrac{\overset{\mathcal{E}_1}{\Gamma, x : A \vdash y : B} \quad \overset{\mathcal{E}_2}{\Gamma, x : A, y^* : B \vdash z : C}}{\Gamma, x : A \vdash z : C}\,\mathsf{Snip}^1_B}{\Gamma \vdash z : C}\,\mathsf{Cut}_A \qquad \Longrightarrow \quad \mathcal{F}$$

We first apply the induction hypothesis twice on smaller derivations.

$$\dfrac{\overset{\mathcal{D}}{\Gamma \vdash x : A} \quad \overset{\mathcal{E}_1}{\Gamma, x : A \vdash y : B}}{\Gamma \vdash y : B}\,\mathsf{Cut}_A \quad \Longrightarrow \quad \overset{\mathcal{F}_1}{\Gamma \vdash y : B}$$

and

$$\dfrac{\overset{\mathcal{D} + \{y^0 : B\}}{\Gamma, y^0 : B \vdash x : A} \quad \overset{\mathcal{E}_2}{\Gamma, y^* : B, x : A \vdash z : C}}{\Gamma, y : B \vdash z : C}\,\mathsf{Cut}_A \quad \Longrightarrow \quad \overset{\mathcal{F}_2}{\Gamma, y : B \vdash z : C}$$

There are three subcases for the status of $y : B$ in $\mathcal{F}_2$.

Subcase: $y : B$ becomes irrelevant.

Subcase: $y : B$ remains eligible.

Subcase: $y : B$ is ineligible but relevant and $B < A$.

These subcases yield the following three definitions for $\mathcal{F}$, respectively:

$$\mathcal{F} = \begin{matrix}\mathcal{F}_2 - \{y^0 : B\} \\ \Gamma \vdash z : C\end{matrix} \qquad \mathcal{F} = \dfrac{\overset{\mathcal{F}_1}{\Gamma \vdash y : B} \quad \overset{\mathcal{F}_2}{\Gamma, y^* : B \vdash z : C}}{\Gamma \vdash z : C}\,\mathsf{Snip}^2_B \qquad \mathcal{F} = \dfrac{\overset{\mathcal{F}_1}{\Gamma \vdash y : B} \quad \overset{\mathcal{F}_2}{\Gamma, y : B \vdash z : C}}{\Gamma \vdash z : C}\,\mathsf{Cut}_B$$

The next set of cases the cut formula is a side formula of the inference in $\mathcal{D}$ and $\mathcal{E}$ is arbitrary.
Case: $\mathsf{Snip}^1/\_$.

$$\dfrac{\dfrac{\overset{\mathcal{D}_1}{\Gamma \vdash y^* : B} \quad \overset{\mathcal{D}_2}{\Gamma, y : B \vdash x : A}}{\Gamma \vdash x : A}\,\mathsf{Snip}^1_B \quad \overset{\mathcal{E}}{\Gamma, x : A \vdash z : C}}{\Gamma \vdash z : C}\,\mathsf{Cut}_A$$

$$\Longrightarrow \quad \dfrac{\overset{\mathcal{D}_1}{\Gamma \vdash y^* : B} \quad \dfrac{\overset{\mathcal{D}_2}{\Gamma, y : B \vdash x : A} \quad \overset{\mathcal{E} + \{y^0 : B\}}{\Gamma, y^0 : B, x : A \vdash z : C}}{\Gamma, y : B \vdash z : C}\,\mathsf{Cut}_A}{\Gamma \vdash z : C}\,\mathsf{Snip}^1_B$$

Case: $\mathsf{Snip}^2/\_$

$$\dfrac{\dfrac{\overset{\mathcal{D}_1}{\Gamma \vdash y : B} \quad \overset{\mathcal{D}_2}{\Gamma, y^* : B \vdash x : A}}{\Gamma \vdash x : A}\,\mathsf{Snip}^2_B \quad \overset{\mathcal{E}}{\Gamma, x : A \vdash z : C}}{\Gamma \vdash z : C}\,\mathsf{Cut}_A \quad \Longrightarrow \quad \overset{\mathcal{F}}{\Gamma \vdash z : C}$$

where

$$\mathcal{F}_2 = \dfrac{\overset{\mathcal{D}_2}{\Gamma, y^* : B \vdash x : A} \quad \overset{\mathcal{E} + \{y^0 : B\}}{\Gamma, y^0 : B, x : A \vdash z : C}}{\Gamma, y : B \vdash z : C}\,\mathsf{Cut}_A$$

Again, there are three subcases.
Subcase: $y^0 : B$ is irrelevant in $\mathcal{F}_2$.
Subcase: $y^* : B$ is eligible in $\mathcal{F}_2$.
Subcase: $y : B$ is relevant but not eligible in $\mathcal{F}_2$ and $B < A$.
These three subcases yield the following three definitions for $\mathcal{F}$, respectively:

$$\mathcal{F} = \begin{matrix}\mathcal{F}_2 - \{y^0 : B\} \\ \Gamma \vdash z : C\end{matrix} \qquad \mathcal{F} = \dfrac{\overset{\mathcal{D}_1}{\Gamma \vdash y : B} \quad \overset{\mathcal{F}_2}{\Gamma, y^* : B \vdash z : C}}{\Gamma \vdash z : C}\,\mathsf{Snip}^2_B \qquad \mathcal{F} = \dfrac{\overset{\mathcal{D}_1}{\Gamma \vdash y : B} \quad \overset{\mathcal{F}_2}{\Gamma, y : B \vdash z : C}}{\Gamma \vdash z : C}\,\mathsf{Cut}_B$$

◀

B Some Example Programs

The following are the process expressions assigned to the proofs in Section 3 and Section 4.

$$x : A \supset B, y : B \supset C \vdash z : A \supset C$$

$$\begin{aligned}&\textbf{case } z^W\ (\langle u, w\rangle \Rightarrow b \leftarrow x^R.\langle u, b\rangle\ ;\\ &\qquad\qquad y^R.\langle b, w\rangle)\end{aligned}$$

This process terminates immediately after writing a continuation to z. When this continuation is called with an argument u and destination w, it allocates a fresh cell b and passes u to x with instructions to place the result in b. While this continuation executes, it passes the address b (which may not be filled yet) to y and jumps to *that* continuation in order to write *its* answer into w. Note that two processes may execute concurrently here: intuitively, one executing the function x and the other executing the function y. They are connected via a common reference to a fresh cell b, to be written by x and read by y.

$$x : A \mathbin{\&} B \vdash w : A \otimes B$$

$$\begin{array}{l} y \leftarrow x^R.\pi_1(y) \ ; \\ z \leftarrow x^R.\pi_2(z) \ ; \\ w^W.\langle y, z\rangle \end{array}$$

This process reads the continuation at x twice, requesting that the first component of the pair be written to a freshly allocated cell y, the second to a freshly allocated cell z. While these processes execute, it writes the pair $\langle y, z\rangle$ to the required destination w and terminates. Note that both y and z have been allocated, but neither needs to have been filled by the time this process terminates, since the two processes executing with the destinations y and z can continue to run.

$$x : A \otimes B \vdash w : A \mathbin{\&} B$$

$$\begin{array}{l} \textbf{case}\ x^R\ (\langle y, z\rangle \Rightarrow \\ \qquad \textbf{case}\ w^W\ (\,\pi_1(u) \Rightarrow u^W \leftarrow y^R \\ \qquad\qquad\quad \mid \pi_2(v) \Rightarrow v^W \leftarrow z^R\,)) \end{array}$$

This process reads the pair $\langle y, z\rangle$ from x and then terminates by writing a continuation to w. If this continuation is invoked by a reader requesting the first component of $A \mathbin{\&} B$ to be put into u, this is satisfied by copying the contents of y; if the second component is requested we copy the contents of z. This program is essentially sequential.

$$p : (A \supset C) \mathbin{\&} (B \supset C) \vdash q : (A \vee B) \supset C$$

$$\begin{array}{l} \textbf{case}\ q^W\ (\langle s, z\rangle \Rightarrow \\ \qquad \textbf{case}\ s^R\ (\,\pi_1(x) \Rightarrow r \leftarrow p^R.\pi_1(r)\ ;\ r^R.\langle x, z\rangle \\ \qquad\qquad\quad \mid \pi_2(y) \Rightarrow t \leftarrow p^R.\pi_2(t)\ ;\ t^R.\langle y, z\rangle\,)) \end{array}$$

This process writes a continuation to q and terminates. This continuation represents a function with argument s and destination z. If it is invoked, it distinguishes two cases for the contents of s. If it is $\pi_1(x)$ for some address x, it obtains the first component of p (call it r) and invokes that with x, requesting the result to be put into z. If it is $\pi_2(y)$, it takes a symmetric action with the second component of p. This example has little parallelism: even though a fresh process is spawned to fill r and t in the two branches of the inner case, they are immediately read which will block until r and t have been filled, respectively.

C More Details in the Termination Proof

Proof Sketch of Theorem 19. The case of an empty configuration is straightforward, as is the case where the rightmost object in $\mathcal{C}$ provides an address $d \neq a$. In all remaining cases, we may therefore assume that $\mathcal{C}$ is non-empty, and that its rightmost part (either a cell $!\mathsf{cell}\ a\ W$ or a process with its unfilled destination cell at a) provides a. We refer to this component as ϕ_a and the remainder of $\mathcal{C}$ as $\mathcal{C}'$.

If the typing derivation for ϕ_a ends with an id rule, then $\phi_a = (\mathsf{proc}\ a\ a \leftarrow b, \mathsf{cell}\ a\ \_)$ for some $b : B$ in Δ. We therefore have (by the inductive hypothesis) that $\mathcal{C}' \in [\![\Gamma \vdash b : A]\!]$. Let $\mathcal{C}'' \in [\![\Gamma]\!]$. Now, by definition, $(\mathcal{C}'', \mathcal{C}') \in [\![b : A]\!]$, and so $(\mathcal{C}'', \mathcal{C}') \longrightarrow^* \mathcal{F}$ for some $\mathcal{F}$ final and $\mathcal{F} \in [b : A]$. Lemma 18 gives us that $\mathcal{F}$ contains $!\mathsf{cell}\ b\ W$ for some W. We therefore have that $(\mathcal{C}'', \mathcal{C}) \longrightarrow^* \mathcal{F}, !\mathsf{cell}\ a\ W$. Now, split $\mathcal{C}'$ into $\mathcal{C}_1, \mathcal{C}_2$, where $\mathcal{C}_2$ contains exactly the objects which mention or provide b. We may now apply the inductive hypothesis to $\mathcal{C}_1, !\mathsf{cell}\ a\ W$ to get that $\mathcal{C}_1, !\mathsf{cell}\ a\ W \in [\![\Gamma \vdash a : A]\!]$. Lemma 16 then allows us to conclude that $\mathcal{C}', !\mathsf{cell}\ a\ W \in [\![\Gamma \vdash a : A]\!]$. Thus, $\mathcal{C}'', \mathcal{C}', !\mathsf{cell}\ a\ W \longrightarrow^* \mathcal{F}', !\mathsf{cell}\ a\ W$ and $\mathcal{F}', !\mathsf{cell}\ a\ W \in [a : A]$. Confluence (an easy consequence of Theorem 10) gives us that $\mathcal{F} = \mathcal{F}'$, and Lemma 17 completes this case.

If the typing derivation for ϕ_a ends with a cut rule, we may simply apply to the inductive hypothesis after taking a step.

If the typing derivation for ϕ_a ends with $\supset L$, $\& L$, $\mathbf{1}R$, $\otimes R$, or $\vee R$, then we simply take either zero or one steps (depending on exactly which rule it is and what form $\mathcal{C}$ has), invoke the inductive hypothesis up to two times on $\mathcal{C}'$, and then conclude using Lemma 17.

If the typing derivation for ϕ_a ends with $\mathbf{1}L$, $\otimes L$, or $\vee L$, we proceed much as in the case of id. We set up a configuration $\widehat{\mathcal{C}}$ which is our candidate for what $\mathcal{C}$ would look like if we were able to step in ϕ_a, and using the inductive hypothesis both on $\widehat{\mathcal{C}}$ and on $\mathcal{C}'$, we are able to show that given $\mathcal{C}'' \in [\![\Gamma]\!]$, both $(\mathcal{C}'', \mathcal{C})$ and $(\mathcal{C}'', \widehat{\mathcal{C}})$ reduce to the same configuration $\widetilde{\mathcal{C}}$ (taking advantage of confluence and Lemma 18 to do so). As in the identity case, Lemma 17 completes these cases.

If the typing derivation for ϕ_a ends with $\supset R$ or $\& R$, we proceed largely similarly, constructing a candidate $\widehat{\mathcal{C}}$ for the result of stepping ϕ_a in $\mathcal{C}$. The only difference is that here, we need to augment $\mathcal{C}$ with an additional process which reads from the cell a, rather than working with $\mathcal{C}$ on its own. ◀

Constraint Solving over Multiple Similarity Relations

Besik Dundua
FBT, International Black Sea University, Tbilisi, Georgia
VIAM, Ivane Javakhishvili Tbilisi State University, Georgia
bdundua@gmail.com

Temur Kutsia
Johannes Kepler University, Research Institute for Symbolic Computation, Linz, Austria
http://www.risc.jku.at/people/tkutsia/
kutsia@risc.jku.at

Mircea Marin
West University of Timişoara, Romania
http://staff.fmi.uvt.ro/~mircea.marin/
mircea.marin@e-uvt.ro

Cleopatra Pau
Johannes Kepler University, Research Institute for Symbolic Computation, Linz, Austria
ipau@risc.jku.at

Abstract

Similarity relations are reflexive, symmetric, and transitive fuzzy relations. They help to make approximate inferences, replacing the notion of equality. Similarity-based unification has been quite intensively investigated, as a core computational method for approximate reasoning and declarative programming. In this paper we consider solving constraints over several similarity relations, instead of a single one. Multiple similarities pose challenges to constraint solving, since we can not rely on the transitivity property anymore. Existing methods for unification with fuzzy proximity relations (reflexive, symmetric, non-transitive relations) do not provide a solution that would adequately reflect particularities of dealing with multiple similarities. To address this problem, we develop a constraint solving algorithm for multiple similarity relations, prove its termination, soundness, and completeness properties, and discuss applications.

2012 ACM Subject Classification Theory of computation → Logic; Computing methodologies → Symbolic and algebraic manipulation; Theory of computation → Semantics and reasoning

Keywords and phrases Fuzzy relations, similarity, constraint solving

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.30

Funding This research has been partially supported by the Austrian Science Fund (FWF) under the project 28789-N32 and the Shota Rustaveli National Science Foundation of Georgia under the grant YS-18-1480.

1 Introduction

Reasoning with incomplete, imperfect information is very common in human communication. Its modeling is a highly nontrivial task, and remains an important issue in applications of artificial intelligence. There are various notions associated to such information (e.g., uncertainty, imprecision, vagueness, fuzziness) and different methodologies have been proposed to deal with them (e.g., approaches based on default logic, probability, fuzzy sets, etc.)

For many problems in this area, exact equality is replaced by its approximation. Several approaches use similarity relations to express the approximation, modeling the corresponding imprecise information. Similarity relations are fuzzy binary relations, specifying to which

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 30; pp. 30:1–30:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

degree two objects are similar to each other. They satisfy reflexivity, symmetry, and fuzzy min-transitivity properties, and can be characterized by "level-wise" equivalence. Once the level $\lambda > 0$ of similarity is fixed, the set of all object-pairs, which have the similarity degree at least λ, form a classical equivalence relation.

Reasoning with similarity relations requires approximate inference techniques. Solving similarity-based constraints is the central computational mechanism for such inferences. Several approaches to unification modulo similarity have been proposed, see, e.g., [1, 2, 5–9, 14, 16, 17, 20, 21]. Recently, unification was studied also for proximity relations, which generalize similarity in the sense they are reflexive and symmetric but non-transitive fuzzy relations [10, 12, 15]. The techniques studied in these papers usually assume a single fuzzy (similarity or proximity) relation. However, in many practical situations, one needs to deal with several similarities between the objects from the same set, see, e.g. [18, 19], where examples about building online fashion compatibility representation and understanding visual similarities are considered in the context of learning image embeddings.

Multiple similarities pose challenges to constraint solving, since we can not rely on the transitivity property anymore. Note that proximity relations are not transitive either, but their unification methods have some limitations in dealing with multiple similarities simultaneously.

We address this problem, proposing an algorithm for solving constraints over multiple similarity relations. A simple example below illustrates the problem together with the results of different approaches, and motivates the development of a dedicated technique for it.

▶ **Example 1.** Let *white-circle*, *white-ellipse*, *gray-circle* and *gray-ellipse* be four symbols and $\mathcal{R}_1$ and $\mathcal{R}_2$ be two similarity relations, where $\mathcal{R}_1$ stands for "similar color, same shape" and $\mathcal{R}_2$ denotes "same color, similar shape". They are defined as

- $\mathcal{R}_1(\textit{white-circle}, \textit{gray-circle}) = \mathcal{R}_1(\textit{white-ellipse}, \textit{gray-ellipse}) = 0.5$, and
- $\mathcal{R}_2(\textit{white-circle}, \textit{white-ellipse}) = \mathcal{R}_2(\textit{gray-circle}, \textit{gray-ellipse}) = 0.7$.

Assume we want to find an object X such that from the color point of view, it is at least 0.4-similar to *white-circle* and from the shape point of view, it is at least 0.5-similar to *gray-ellipse*. The corresponding constraint is $X \simeq_{\mathcal{R}_1,0.4}$ *white-circle* and $X \simeq_{\mathcal{R}_2,0.5}$ *gray-ellipse*. The expected answer is $X =$ *gray-circle*. But it is problematic to compute it by the existing fuzzy unification techniques. The direct approach, trying to solve each equation separately by the weak unification algorithm from [17] leads to no solution in this case, because *white-circle* and *gray-ellipse* are not similar to each other by any of the given relations. An alternative way could be to consider the constraint over the relation $\mathcal{R}_1 \cup \mathcal{R}_2$, which is a proximity, not a similarity, since transitivity is not satisfied. However, the proximity unification algorithm from [12] gives no solution. We can try to use the algorithm for solving proximity constraints from [15], but it would give two answers instead of one: $X =$ *gray-circle* and $X =$ *white-ellipse*. On the other hand, the algorithm proposed in this paper computes the right solution $X =$ *gray-circle*. Its similarity degrees are 0.5 for the relation $\mathcal{R}_1$ and 0.7 for $\mathcal{R}_2$. ◀

It should be mentioned that the multi-adjoint framework [14, 16] is flexible enough to accommodate multiple similarities. It is a logic programming-based approach, where one needs to extend programs by fuzzy similarity axioms for each alphabet symbol and use classical unification. The authors show how to encode Sessa's algorithm [17] in this framework.

Our approach is different. We develop the solving algorithm directly, without being dependent on the implementation or application preferences. It can be incorporated in a modular way in the constraint logic programming schema, can be used for constrained rewriting, querying, or similar purposes. It combines three parts: solving syntactic equations, solving similarity problems for one relation, and solving mixed problems. Except variables for terms, we permit also variables for function symbols, since they are necessary in the process of finding an "intermediate object" between terms in different similarity relations.

The paper is organized as follows: In Section 2 we introduce the basic notions, define constraints and their solutions. Section 3 is the main section of the paper: it describes all three parts of our algorithm and presents its termination, soundness, and completeness results. In Section 4 we show how to include the computation of approximation degrees in the algorithm. Concluding discussion can be found in Section 5.

2 Preliminaries

Similarity relations

We define basic notions about similarity relations following [9, 17]. A binary *fuzzy relation* on a set S is a mapping from $S \times S$ to the real interval $[0,1]$. If $\mathcal{R}$ is a fuzzy relation on S and λ is a number $0 < \lambda \leq 1$ (called *cut value*), then the λ-*cut* of $\mathcal{R}$ on S, denoted $\mathcal{R}_\lambda$, is an ordinary (crisp) relation on S defined as $\mathcal{R}_\lambda := \{(s_1, s_2) \mid \mathcal{R}(s_1, s_2) \geq \lambda\}$.

A fuzzy relation $\mathcal{R}$ on a set S is called a *proximity relation*, if it is reflexive and symmetric:

Reflexivity: $\mathcal{R}(s, s) = 1$ for all $s \in S$;
Symmetry: $\mathcal{R}(s_1, s_2) = \mathcal{R}(s_2, s_1)$ for all $s_1, s_2 \in S$.

Let $\wedge$ be a T-norm: an associative, commutative, non-decreasing binary operation on $[0,1]$ with 1 as the unit element. A proximity relation (on S) is called a *similarity relation* (on S) iff it is transitive:

Transitivity $\mathcal{R}(s_1, s_2) \geq \mathcal{R}(s_1, s) \wedge \mathcal{R}(s, s_2)$ for any $s_1, s_2, s \in S$.

In this paper, in the role of T-norm we take the *minimum* of two numbers, and write min instead of $\wedge$. In the role of S we take a syntactic domain, defined in the next section.

Terms, atoms, substitutions

Our alphabet $\mathbf{A}$ consists of the following pairwise disjoint sets of symbols:

- $\mathbf{V}_\mathsf{T}$: term variables, denoted by X, Y, Z,
- $\mathbf{V}_\mathsf{F}$: function variables, denoted by F, G, H,
- $\mathbf{C}_\mathsf{F}$: function constants, denoted by f, g, h,

By $\mathbf{V}$ we denote the set of variables $\mathbf{V} = \mathbf{V}_\mathsf{T} \cup \mathbf{V}_\mathsf{F}$, and V is used for its elements.

A *function symbol* is a function variable or a function constant, i.e., an element of the set $\mathbf{F} = \mathbf{C}_\mathsf{F} \cup \mathbf{V}_\mathsf{F}$. We use the letters $\mathsf{f}, \mathsf{g}, \mathsf{h}$ to denote function symbols. Each function symbol has a fixed arity.

Terms over $\mathbf{A}$ are defined by the grammar $t := X \mid \mathsf{f}(t_1, \ldots, t_n)$, where f is an n-ary function symbol. For terms we use the letters t, s, r. The set of terms over $\mathbf{A}$ is denoted by $\mathsf{Terms}(\mathbf{A})$.

For a term $\mathsf{f}(t_1, \ldots, t_n)$, if $n = 0$, we write just f instead of $\mathsf{f}()$. Usually, from the context it is clear whether we are talking about a symbol or about a term.

A *substitution* σ is a mapping from $\mathbf{V}$ to $\mathbf{F} \cup \mathsf{Terms}(\mathbf{A})$ such that

- $\sigma(X) \in \mathsf{Terms}(\mathbf{A})$ for all $X \in \mathbf{V}_\mathsf{T}$,
- $\sigma(F) \in \mathbf{F}$ for all $F \in \mathbf{V}_\mathsf{F}$,
- $\sigma(V) = V$ for all but finitely many variables $V \in \mathbf{V}$.

Substitutions are denoted by Greek letters σ, ϑ, φ. The identity substitution is denoted by *Id*. The domain of a substitution σ is the set $dom(\sigma) = \{V \mid V \in \mathbf{V}, \sigma(V) \neq V\}$. The *restriction* of σ to a set of variables $\mathcal{V}$ is the substitution $\sigma|_{\mathcal{V}}$ defined as $\sigma|_{\mathcal{V}}(V) = \sigma(V)$ if $V \in \mathcal{V}$ and $\sigma|_{\mathcal{V}}(V) = V$ otherwise. We will use the usual set representation of substitutions, writing σ as $\{V \mapsto \sigma(V) \mid V \in dom(\sigma)\}$.

Substitution application to variables, constants, and terms is defined as follows: $c\sigma = c$ for all $c \in \mathbf{C}_{\mathsf{F}}$, $V\sigma = \sigma(V)$ for all $V \in \mathbf{V}$, and $\mathsf{f}(t_1, \dots, t_n)\sigma = (\mathsf{f}\sigma)(t_1\sigma, \dots, t_n\sigma)$.

Similarity relations on syntactic domains

Our similarity relations are defined on the set of constants $\mathbf{C}_{\mathsf{F}}$. Any such relation $\mathcal{R}$ should satisfy the restriction: $\mathcal{R}(f, g) = 0$, if f and g have different arity.

Given an $\mathcal{R}$ defined on $\mathbf{C}_{\mathsf{F}}$, we extend it to $\mathbf{F} \cup \mathsf{Terms}(\mathbf{A})$:

- For variables: $\mathcal{R}(V, V) = 1$.
- For nonvar. terms: $\mathcal{R}(\mathsf{f}(t_1, \dots, t_n), \mathsf{g}(s_1, \dots, s_n)) = \min(\mathcal{R}(\mathsf{f}, \mathsf{g}), \mathcal{R}(t_1, s_1), \dots, \mathcal{R}(t_n, s_n))$, when f and g are both n-ary.
- In all other cases, $\mathcal{R}(\tau_1, \tau_2) = 0$ for $\tau_1, \tau_2 \in \mathbf{F} \cup \mathsf{Terms}(\mathbf{A})$.

Given a similarity relation $\mathcal{R}$ and the cut value $\lambda \in (0, 1]$, we define $(\mathcal{R}, \lambda)$*-neighborhood* of τ as $\mathsf{N}(\tau, \mathcal{R}, \lambda) := \{\tau' \mid \mathcal{R}(\tau, \tau') \geq \lambda\}$, where $\tau, \tau' \in \mathbf{F} \cup \mathsf{Terms}(\mathbf{A})$. Based on the definition of similarity relations above, it is obvious that neighborhoods of function constants (resp. variables) contain only function constants (resp. variables) of the same arity. Neighborhoods of terms contain only terms. All terms in the same neighborhood have the same structure (same set of positions). We require for each $f \in \mathbf{C}_{\mathsf{F}}$, $\mathcal{R}$, and λ, the set $\mathsf{N}(f, \mathcal{R}, \lambda)$ to be finite. It implies that term neighborhoods are finite as well.

Constraints

In our constraint language, the elements of $\mathbf{F} \cup \mathsf{Terms}(\mathbf{A})$ are the basic objects. In the rest of the paper, the letter τ is used to denote its elements. Besides, we have the equality predicate constant $\doteq$ (interpreted as syntactic equality), one or more similarity predicate constants $\simeq_1$, $\simeq_2$, ..., (interpreted as similarity relations on $\mathbf{F} \cup \mathsf{Terms}(\mathbf{A})$), propositional constants true and false, connectives $\wedge, \vee$, and the quantifier $\exists$.

Primitive constraints $\mathcal{P}$ are defined by the grammar

$$\mathcal{P} ::= \ \mathsf{true} \mid \mathsf{false} \mid t \doteq s \mid t \simeq s \mid \mathsf{f} \doteq \mathsf{g} \mid \mathsf{f} \simeq \mathsf{g},$$

where $\simeq\ \in \{\simeq_1, \simeq_2 \dots\}$. Primitive $\doteq$- and $\simeq$-constraints are called *primitive equality constraints* and *primitive similarity constraints*, respectively. A *literal* L is an atom or a primitive constraint. A (positive) *constraint* $\mathcal{C}$ over $\mathbf{A}$ is defined as $\mathcal{C} ::= \mathcal{P} \mid \mathcal{C} \wedge \mathcal{C} \mid \mathcal{C} \vee \mathcal{C} \mid \exists x.\mathcal{C}$. In this paper we consider only positive constraints.

The domain of the *intended interpretation* of our constraint language is its Herbrand universe (the set of ground terms). The predicate constant $\doteq$ is interpreted as syntactic equality. Each similarity predicate constant $\simeq$ is interpreted as a similarity relation on the domain as defined in the previous section. When a predicate constant $\simeq$ is to be interpreted by a relation $\mathcal{R}$ with the cut value $\lambda \in (0, 1]$, we write $\simeq_{\mathcal{R},\lambda}$ instead of $\simeq$.

A *variable-predicate pair (VP-pair)* is either $\langle V, \simeq_{\mathcal{R},\lambda}\rangle$ or $\langle V, \doteq\rangle$. We say that a substitution σ is *more general* than ϑ on a set of VP-pairs $\mathcal{W}$ iff there exists a substitution φ such that $\mathcal{R}(V\sigma\varphi, V\vartheta) \geq \lambda$ for all $\langle V, \simeq_{\mathcal{R},\lambda}\rangle \in \mathcal{W}$ and $V\sigma\varphi = V\vartheta$ for all $\langle V, \doteq\rangle \in \mathcal{W}$. In this case we write $\sigma \preceq_{\mathcal{W}} \vartheta$.

▶ **Example 2.** Let $\mathcal{R}_1(a,b) = 0.7$, $\mathcal{R}_1(b,c) = 0.7$, $\mathcal{R}_1(a,c) = 0.8$, $\mathcal{R}_2(b,c) = 0.9$, and $\mathcal{W} = \{\langle X, \simeq_{\mathcal{R}_1,0.5}\rangle, \langle Y, \simeq_{\mathcal{R}_1,0.6}\rangle, \langle Y, \simeq_{\mathcal{R}_2,0.7}\rangle\}$.

- Let $\sigma = \{X \mapsto Y\}$ and $\vartheta = \{X \mapsto a, Y \mapsto b\}$. Then $\sigma \preceq_{\mathcal{W}} \vartheta$, because for $\varphi = \{X \mapsto b, Y \mapsto b\}$ we have $X\sigma\varphi = b \simeq_{\mathcal{R}_1,0.5} a = X\vartheta$, $Y\sigma\varphi = b \simeq_{\mathcal{R}_1,0.6} b = Y\vartheta$, and $Y\sigma\varphi = b \simeq_{\mathcal{R}_2,0.7} b = Y\vartheta$.
- Let $\sigma = \{X \mapsto Y\}$ and $\vartheta = \{X \mapsto a, Y \mapsto c\}$. Then $\sigma \preceq_{\mathcal{W}} \vartheta$, because for $\varphi = \{X \mapsto b, Y \mapsto b\}$ we have $X\sigma\varphi = b \simeq_{\mathcal{R}_1,0.5} a = X\vartheta$, $Y\sigma\varphi = b \simeq_{\mathcal{R}_1,0.6} c = Y\vartheta$, and $Y\sigma\varphi = b \simeq_{\mathcal{R}_2,0.7} c = Y\vartheta$.
- Let $\sigma = \{X \mapsto f(Y), Y \mapsto Z\}$ and $\vartheta = \{X \mapsto f(Z), Y \mapsto a, Z \mapsto X\}$. Then $\sigma \preceq_{\mathcal{W}} \vartheta$, because for $\varphi = \{Y \mapsto Z, Z \mapsto a\}$ we have $X\sigma\varphi = f(Z) \simeq_{\mathcal{R}_1,0.5} f(Z) = X\vartheta$, $Y\sigma\varphi = a \simeq_{\mathcal{R}_1,0.6} a = Y\vartheta$, and $Y\sigma\varphi = a \simeq_{\mathcal{R}_2,0.7} a = Y\vartheta$.

▶ **Theorem 3.** *$\preceq_{\mathcal{W}}$ is a quasi-ordering for all $\mathcal{W}$.*

Proof. Reflexivity is obvious. For transitivity, assume $\sigma_1 \preceq_{\mathcal{W}} \sigma_2$ and $\sigma_2 \preceq_{\mathcal{W}} \sigma_3$. We will show $\sigma_1 \preceq_{\mathcal{W}} \sigma_3$. Take $\langle V, \simeq_{\mathcal{R},\lambda}\rangle \in \mathcal{W}$. Then for some φ_1 and φ_2 we have $\mathcal{R}(V\sigma_1\varphi_1, V\sigma_2) \geq \lambda$ and $\mathcal{R}(V\sigma_2\varphi_2, V\sigma_3) \geq \lambda$. Since similarity is stable for substitutions [17, Proposition 3.1], we have $\mathcal{R}(V\sigma_1\varphi_1\varphi_2, V\sigma_2\varphi_2) \geq \lambda$. By transitivity of similarity, we get $\mathcal{R}(V\sigma_1\varphi_1\varphi_2, V\sigma_3) \geq \min(\mathcal{R}(V\sigma_1\varphi_1\varphi_2, V\sigma_2\varphi_2), \mathcal{R}(V\sigma_2\varphi_2, V\sigma_3)) \geq \lambda$, which implies that $\sigma_1 \preceq_{\mathcal{W}} \sigma_3$. ◀

We denote the equivalence relation induced by $\preceq_{\mathcal{W}}$ by $\cong$.

The notation $\mathcal{K}_{\doteq}$ denotes a conjunction of primitive equality constraints. By $\mathcal{K}_{\mathcal{R},\lambda}$ we denote a conjunction of primitive similarity constraints, all with the same relation $\mathcal{R}$ and the same λ-cut: $\mathcal{K}_{\mathcal{R},\lambda} = \tau_1 \simeq_{\mathcal{R},\lambda} \tau_1' \wedge \cdots \wedge \tau_n \simeq_{\mathcal{R},\lambda} \tau_n'$.

Given a constraint $\mathcal{K} = \mathcal{K}_{\doteq} \wedge \mathcal{K}_{\mathcal{R}_1,\lambda_1} \wedge \cdots \wedge \mathcal{K}_{\mathcal{R}_m,\lambda_m}$, we denote by $\mathcal{W}(\mathcal{K})$ the set of VP-pairs $\mathcal{W}(\mathcal{K}) := \{\langle V, \doteq\rangle \mid V \in var(\mathcal{K}_{\doteq})\} \cup \{\langle V, \simeq_{\mathcal{R}_i,\lambda_i}\rangle \mid V \in var(\mathcal{K}_{\mathcal{R}_i,\lambda_i}), 1 \leq i \leq m\}$.

▶ **Definition 4** (Solution). *A substitution σ is called a* solution *of a primitive constraint $\mathcal{P}$, if*

- *$\mathcal{P} = \tau_1 \doteq \tau_2$ and $\tau_1\sigma = \tau_2\sigma$, or*
- *$\mathcal{P} = \tau_1 \simeq_{\mathcal{R},\lambda} \tau_2$ and $\mathcal{R}(\tau_1\sigma, \tau_2\sigma) \geq \lambda$.*

Any substitution is a solution of true*, while* false *has no solution.*

A substitution σ is a solution of a conjunction of primitive constraints *$\mathcal{K}$ iff it solves each primitive constraint in $\mathcal{K}$. We denote the* set of all solutions of *$\mathcal{K}$ by $Sol(\mathcal{K})$. For a constraint $\mathcal{C} = \mathcal{K}_1 \wedge \cdots \wedge \mathcal{K}_n$ in disjunctive normal form (DNF), we define $Sol(\mathcal{C}) = \cup_{i=1}^n Sol(\mathcal{K}_i)$.*

Given similarity relations $\mathcal{R}_1, \ldots, \mathcal{R}_n$, a conjunction of primitive constraints $\mathcal{K}$, and its solution σ, we say that σ solves *$\mathcal{K}$* with approximation degrees *$\mathfrak{D} = \{\langle \mathcal{R}_1, \mathfrak{d}_1\rangle, \ldots, \langle \mathcal{R}_n, \mathfrak{d}_n\rangle\}$ if*

- *$\mathcal{K} =$ true or $\mathcal{K} = \mathcal{K}_{\doteq}$ and $\mathfrak{d}_1 = \cdots = \mathfrak{d}_n = 1$,*
- *$\mathcal{K} = \tau_1 \simeq_{\mathcal{R}_j,\lambda_j} \tau_2$ for some $1 \leq j \leq n$, $\mathfrak{d}_j = \mathcal{R}_j(\tau_1\sigma, \tau_2\sigma) \geq \lambda_j$, and $\mathfrak{d}_i = 1$ for all $1 \leq i \leq n$, $i \neq j$,*
- *$\mathcal{K} = \mathcal{P} \wedge \mathcal{K}'$, σ solves $\mathcal{P}$ and $\mathcal{K}'$ with approximation degrees $\{\langle \mathcal{R}_1, \mathfrak{d}_1^{\mathcal{P}}\rangle, \ldots, \langle \mathcal{R}_n, \mathfrak{d}_n^{\mathcal{P}}\rangle\}$ and $\{\langle \mathcal{R}_1, \mathfrak{d}_1'\rangle, \ldots, \langle \mathcal{R}_n, \mathfrak{d}_n'\rangle\}$, respectively, and $\mathfrak{d}_i = \min\{\mathfrak{d}_i^{\mathcal{P}}, \mathfrak{d}_i'\}$ for all $1 \leq i \leq n$.*

Such a definition of approximation degrees gives the flexibility to characterize approximations with respect to each involved relation independently from each other.

▶ **Theorem 5.** *Let $\mathcal{K} = \mathcal{K}_{\doteq} \wedge \mathcal{K}_{\mathcal{R}_1,\lambda_1} \wedge \cdots \wedge \mathcal{K}_{\mathcal{R}_m,\lambda_m}$ be a constraint. If σ is a solution of $\mathcal{K}$ and $\sigma \preceq_{\mathcal{W}(\mathcal{K})} \vartheta$, then ϑ is a solution of $\mathcal{K}$.*

Proof. Let $s_1 \simeq_{\mathcal{R}_i,\lambda_i} s_2 \in \mathcal{K}_{\mathcal{R}_i,\lambda_i}$. From $\sigma \preceq_{\mathcal{W}(\mathcal{K})} \vartheta$, by definition of $\preceq_{\mathcal{W}(\mathcal{K})}$, there exists a φ such that $\mathcal{R}(V\sigma\varphi, V\vartheta) \geq \lambda_i$ for each $V \in \mathit{var}(\mathcal{K}_{\mathcal{R}_i,\lambda_i})$. It implies that

$$\mathcal{R}(s_j\sigma\varphi, s_j\vartheta) \geq \lambda_i, \quad j = 1, 2. \tag{1}$$

On the other hand, for similarity relations $\mathcal{R}(s_1\sigma\varphi, s_2\sigma\varphi) = \mathcal{R}(s_1\sigma, s_2\sigma)$ (see [17]). Since σ is a solution of $\mathcal{K}$, $\mathcal{R}(s_1\sigma, s_2\sigma) \geq \lambda_i$. Hence, we have $\mathcal{R}(s_1\sigma\varphi, s_2\sigma\varphi) \geq \lambda_i$. From this inequality and (1), by symmetry and transitivity of $\mathcal{R}$, we get $\mathcal{R}(s_1\vartheta, s_2\vartheta) \geq \lambda_i$. Hence, ϑ is a solution of $s_1 \simeq_{\mathcal{R}_i,\lambda_i} s_2$.

It is straightforward that ϑ is a solution of any equation from $\mathcal{K}_{\doteq}$. Hence, ϑ is a solution of $\mathcal{K}$. ◀

▶ **Definition 6** (Solved form, approximately solved form). *A conjunction of primitive constraints $\mathcal{K}$ is in* solved form, *if $\mathcal{K}$ is either* true *or each primitive constraint in $\mathcal{K}$ has a form $V \doteq \mathrm{t}$ or $V \simeq_{\mathcal{R},\lambda} \mathrm{t}$, where V appears only once in $\mathcal{K}$. A constraint in DNF $\mathcal{K}_1 \vee \cdots \vee \mathcal{K}_n$ is in solved form, if each $\mathcal{K}_i$ is in solved form.*

A conjunction of primitive constraints $\mathcal{K}_{\text{sol}} \wedge \mathcal{K}_{\text{var}}$ is in approximately solved form (appr-solved form) *if $\mathcal{K}_{\text{sol}}$ is in solved form, and $\mathcal{K}_{\text{var}}$ is a conjunction of primitive similarity constraints between variables $V_1 \simeq_{\mathcal{R},\lambda} V_2$ such that neither V_1 nor V_2 appear in the left hand side of any primitive constraint in $\mathcal{K}_{\text{sol}}$. A constraint in DNF $\mathcal{K}_1 \vee \cdots \vee \mathcal{K}_n$ is in appr-solved form, if each $\mathcal{K}_i$ is in appr-solved form.*

Appr-solved forms are also solved forms, but not vice versa. Each solved form $\mathcal{K}$ *induces* a substitution, denoted by $\sigma_{\mathcal{K}}$: if $\mathcal{K} =$ true, then $\sigma_{\mathcal{K}} = \mathit{Id}$, otherwise $\sigma_{\mathcal{K}} = \{V \mapsto \mathrm{t} \mid V \doteq \mathrm{t} \in \mathcal{K}$ or $V \simeq_{\mathcal{R},\lambda} \mathrm{t} \in \mathcal{K}\}$. Obviously, $\sigma_{\mathcal{K}}$ is a solution of $\mathcal{K}$. A constraint $\mathcal{K}_{\text{sol}} \wedge \mathcal{K}_{\text{var}}$ in appr-solved form is also solvable, because $\sigma_{\mathcal{K}_{\text{sol}}}$ solves $\mathcal{K}_{\text{sol}}$ and $\mathcal{K}_{\text{var}}$ always has at least a trivial solution mapping all terms variables to the same term variable and all function variables to the same function variable.

▶ **Example 7.** Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be defined as in Example 1 and $\mathcal{K} = \mathcal{K}_{\mathcal{R}_1,0.4} \wedge \mathcal{K}_{\mathcal{R}_2,0.5}$ be a constraint, where $\mathcal{K}_{\mathcal{R}_1,0.4} = X \simeq_{\mathcal{R}_1,0.4}$ *white-circle* $\wedge X \simeq_{\mathcal{R}_1,0.4} Y$ and $\mathcal{K}_{\mathcal{R}_2,0.5} = X \simeq_{\mathcal{R}_2,0.5}$ *gray-ellipse* $\wedge Y \simeq_{\mathcal{R}_2,0.5}$ *white-ellipse.*

One can bring $\mathcal{K}_{\mathcal{R}_1,0.4}$ to its equivalent solved form (e.g., by an algorithm along the lines of the weak unification algorithm in [17]). $\mathcal{K}_{\mathcal{R}_2,0.5}$ is already in the solved form. Hence, $\mathcal{K}$ is equivalent to the constraint

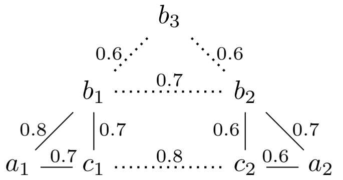
$$X \simeq_{\mathcal{R}_1,0.4} \textit{white-circle} \wedge Y \simeq_{\mathcal{R}_1,0.4} \textit{white-circle} \wedge$$
$$X \simeq_{\mathcal{R}_2,0.5} \textit{gray-ellipse} \wedge Y \simeq_{\mathcal{R}_2,0.5} \textit{white-ellipse},$$

which is not yet in a solved form. A solved form, equivalent to $\mathcal{K}$, would be $X \doteq$ *gray-circle* $\wedge$ $Y \doteq$ *white-circle*. It induces the substitution $\sigma = \{X \mapsto$ *gray-circle*$, Y \mapsto$ *white-circle*$\}$.

▶ **Example 8.** Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two similarity relations defined as

$$\begin{aligned}
\mathcal{R}_1 : \quad & \mathcal{R}_1(a_1, c_1) = \mathcal{R}_1(b_1, c_1) = 0.7, \quad \mathcal{R}_1(a_1, b_1) = 0.8, \\
& \mathcal{R}_1(a_2, c_2) = \mathcal{R}_1(b_2, c_2) = 0.6, \quad \mathcal{R}_1(a_2, b_2) = 0.7, \\
\mathcal{R}_2 : \quad & \mathcal{R}_2(b_1, b_3) = \mathcal{R}_2(b_2, b_3) = 0.6, \quad \mathcal{R}_2(b_1, b_2) = 0.7, \quad \mathcal{R}_2(c_1, c_2) = 0.8.
\end{aligned}$$

Visually:



$\mathcal{R}_1$: the solid lines, $\mathcal{R}_2$: the dotted lines.

Let $\mathcal{K} = X \simeq_{\mathcal{R}_1,0.5} f(a_1, a_2) \wedge X \simeq_{\mathcal{R}_2,0.6} f(Y, Y)$. It is equivalent to the disjunction of two solved forms, e.g., $(X \doteq f(b_1, b_2) \wedge Y \simeq_{\mathcal{R}_2,0.6} b_1) \vee (X \doteq f(c_1, c_2) \wedge Y \simeq_{\mathcal{R}_2,0.6} c_1)$. The solved forms induce two substitutions: $\sigma_1 = \{X \mapsto f(b_1, b_2),\, Y \mapsto b_1\}$ and $\sigma_2 = \{X \mapsto f(c_1, c_2),\, Y \mapsto c_1\}$. They are solutions of $\mathcal{K}$. There are other solutions of $\mathcal{K}$ that are $\cong$-equivalent to σ_1 or σ_2: $\vartheta_1 = \{X \mapsto f(b_1, b_2),\, Y \mapsto b_2\} \cong \sigma_1$, $\vartheta_2 = \{X \mapsto f(b_1, b_2),\, Y \mapsto b_3\} \cong \sigma_1$, and $\vartheta_3 = \{X \mapsto f(c_1, c_2),\, Y \mapsto c_2\} \cong \sigma_2$.

Now let $\mathcal{K} = X \simeq_{\mathcal{R}_1,0.8} g(Y) \wedge X \simeq_{\mathcal{R}_2,0.6} g(Z)$. A solved form $\mathcal{K}_{\mathrm{s}} = X \doteq g(Z) \wedge Y \doteq Z$ implies $\mathcal{K}$, but is not equivalent to it, because $\mathcal{K}$ has solutions $\{X \mapsto g(b_1), Y \mapsto a_1, Z \mapsto b_2\}$ and $\{X \mapsto g(b_1), Y \mapsto a_1, Z \mapsto b_3\}$, which do not solve $\mathcal{K}_{\mathrm{s}}$. On the other hand, if we take the approximate solved form $\mathcal{K}_{\mathrm{as}} = X \doteq g(X_1) \wedge X_1 \simeq_{\mathcal{R}_1,0.8} Y \wedge X_1 \simeq_{\mathcal{R}_2,0.6} Z$, then every solution of $\mathcal{K}_{\mathrm{as}}$ solves $\mathcal{K}$, and $(\exists X_1.\mathcal{K}_{\mathrm{as}})\sigma$ holds for any solution σ of $\mathcal{K}$. (Substitution application to a quantified constraint avoids variable capture.) Alternatively, we could have taken another solved form $\mathcal{K}'_{\mathrm{s}} = X \doteq g(X_1) \wedge Y \simeq_{\mathcal{R}_1,0.8} X_1 \wedge Z \simeq_{\mathcal{R}_2,0.6} X_1$ which has the same properties as $\mathcal{K}_{\mathrm{as}}$.

▶ **Example 9.** Let $\mathcal{R}_1(a, b_1) = \mathcal{R}_1(b_1, b_2) = \mathcal{R}_1(a, b_2) = 0.8$, $\mathcal{R}_2(c, b_1) = \mathcal{R}_2(b_1, b_2) = \mathcal{R}_2(b_2, c) = 0.7$ and consider a constraint $\mathcal{K} = X \simeq_{\mathcal{R}_1,0.6} f(Y, Y) \wedge X \simeq_{\mathcal{R}_2,0.5} f(Z, Z)$. The straightforward solved form $X \doteq f(Z, Z) \wedge Y \doteq Z$, as in the previous example, has fewer solutions than $\mathcal{K}$, e.g., $\{X \mapsto f(b_1, b_2), Y \mapsto a, Z \mapsto c\}$ would be lost. If we take an appr-solved form $\mathcal{K}_{\mathrm{as}} = X \doteq f(X_1, X_2) \wedge X_1 \simeq_{\mathcal{R}_1,0.6} Y \wedge X_1 \simeq_{\mathcal{R}_2,0.5} Z \wedge X_2 \simeq_{\mathcal{R}_1,0.6} Y \wedge X_2 \simeq_{\mathcal{R}_2,0.5} Z$, then all solutions of $\mathcal{K}_{\mathrm{as}}$ solve $\mathcal{K}$ and for each solution σ of $\mathcal{K}$, we have $(\exists X_1, X_2.\mathcal{K}_{\mathrm{as}})\sigma$. Unlike the previous example, we can not turn this appr-solved form into a solved from by swapping sides of variables-only equations.

3 Constraint solving

The constraint solving algorithm Solve presented in this section works on constraints in DNF. Its rules are divided into three groups: for equalities, for similarities, and for mixed problems. They are applied modulo associativity, commutativity, and idempotence of $\wedge$ and $\vee$, treating false as the unit element of $\vee$. We first introduce the rules and then show how Solve is defined using them.

In the rules, the superscript ? indicates that the constraints are supposed to be solved. The sides of an equation $V \doteq^? \tau$ belong to the same syntactic category, i.e., it stands either for $X \doteq^? t$ or for $F \doteq^? \mathsf{f}$. The same holds for $V \simeq^?_{\mathcal{R},\lambda} \tau$.

Equality rules

In this subsection we describe the rules that solve equality constraints. Essentially, these are first-order unification rules with a slight modification, which concerns dealing with function and predicate variables. The rules have the form $\mathcal{K} \rightsquigarrow \mathcal{K}'$, which defines the transformation $\mathcal{K} \vee \mathcal{C} \rightsquigarrow \mathcal{K}' \vee \mathcal{C}$. Note that $\mathcal{C}$ does not change.

The rules are Del-eq (deletion), Dec-eq (decomposition), Ori-eq (orientation), Elim-eq (variable elimination), Confl-eq (conflict), Mism-eq (arity mismatch), Occ-eq (occurrence check), all formulated for the equality relation $\doteq$.

$$\mathsf{Del\text{-}eq}: \quad \tau \doteq^? \tau \wedge \mathcal{K} \rightsquigarrow \mathcal{K}, \quad \text{where } \tau \in \mathbf{C}_{\mathsf{F}} \cup \mathbf{V}_{\mathsf{F}} \cup \mathbf{V}_{\mathsf{T}}.$$

$$\mathsf{Dec\text{-}eq}: \quad \mathsf{f}(t_1,\dots,t_n) \doteq^? \mathsf{g}(s_1,\dots,s_n) \wedge \mathcal{K} \rightsquigarrow \mathsf{f} \doteq^? \mathsf{g} \wedge t_1 \doteq^? s_1 \wedge \cdots \wedge t_n \doteq^? s_n \wedge \mathcal{K}, \quad \text{where } n > 0.$$

$$\mathsf{Ori\text{-}eq}: \quad \tau \doteq^? V \wedge \mathcal{K} \rightsquigarrow V \doteq^? \tau \wedge \mathcal{K}, \quad \text{if } \tau \notin \mathbf{V}.$$

$$\mathsf{Elim\text{-}eq}: \quad V \doteq^? \tau \wedge \mathcal{K} \rightsquigarrow V \doteq^? \tau \wedge \mathcal{K}\{V \mapsto \tau\}, \quad \text{if } V \notin \mathit{var}(\tau) \text{ and } V \in \mathit{var}(\mathcal{K}).$$

$$\mathsf{Confl\text{-}eq}: \quad f \doteq^? g \wedge \mathcal{K} \rightsquigarrow \mathsf{false}, \quad \text{where } f \neq g.$$

$$\mathsf{Mism\text{-}eq}: \quad \mathsf{f}(t_1,\dots,t_n) \doteq^? \mathsf{g}(s_1,\dots,s_m) \wedge \mathcal{K} \rightsquigarrow \mathsf{false}, \text{ if } n \neq m.$$

$$\mathsf{Occ\text{-}eq}: \quad X \doteq^? t \wedge \mathcal{K} \rightsquigarrow \mathsf{false}, \text{ if } X \in \mathit{var}(t) \text{ and } X \neq t.$$

Note that the Elim-eq rule replaces occurrences of a variable in the whole $\mathcal{K}$, i.e., the variable gets replaced both in equality and similarity constraints.

We define the algorithm Unif, which applies the equality rules as long as possible. When there are more than one applicable rule, the algorithm may choose one arbitrarily.

▶ **Theorem 10.** Unif *is terminating.*

Proof. Similar to the proof of termination of the unification algorithm in [3]. ◀

▶ **Lemma 11** (Soundness lemma for Unif)**.** *If $\mathcal{K} \rightsquigarrow \mathcal{K}'$ is a step performed by a rule in* Unif*, then $Sol(\mathcal{K}) = Sol(\mathcal{K}')$.*

Proof. When $\mathcal{K}$ consists of equational constraints only, then so is $\mathcal{K}'$ and the lemma can be proved as the analogous property of the unification algorithm in [3]. If $\mathcal{K}$ contains similarity constraints as well, the only nontrivial case to consider is the Elim-eq rule. We will need the fact that for any σ and ϑ, $\vartheta\sigma \in Sol(\mathcal{K})$ iff $\sigma \in Sol(\mathcal{K}\vartheta)$ (which is straightforward to show).

Let $\mathcal{K} = \{V \doteq^? \tau\} \wedge \mathcal{K}_0$ and $\vartheta = \{V \mapsto \tau\}$. Then $\mathcal{K}' = \{V \doteq^? \tau\} \wedge \mathcal{K}_0\vartheta$ and we have

$$\begin{aligned}
&\sigma \in Sol(\mathcal{K}) \text{iff } \sigma \in Sol(\{V \doteq^? \tau\} \wedge \mathcal{K}_0) \text{ iff}\\
&\quad V\sigma = \tau\sigma \wedge \sigma \in Sol(\mathcal{K}_0) \text{ iff (because } V\sigma = \tau\sigma \text{ implies } \sigma = \vartheta\sigma)\\
&\quad V\sigma = \tau\sigma \wedge \vartheta\sigma \in Sol(\mathcal{K}_0) \text{ iff}\\
&\quad V\sigma = \tau\sigma \wedge \sigma \in Sol(\mathcal{K}_0\vartheta) \text{ iff}\\
&\quad \sigma \in Sol(\{V \doteq^? \tau\} \wedge \mathcal{K}_0\vartheta) \text{ iff}\\
&\sigma \in Sol(\mathcal{K}').
\end{aligned}$$

◀

Similarity rules

The rules in this section are designed for similarity relations. They resemble weak unification rules [9, 17], with the difference that function variables and multiple similarity relations are permitted, and aims at computing the answer in solved form, instead of a substitution.

In the Elim-sim rule, the variable V is replaced by τ only in the constraints for the same similarity relation. This is justified by the fact that although a λ-cut of each similarity relation is transitive, from $t \simeq_{\mathcal{R}_1,\lambda_1} s$, $s \simeq_{\mathcal{R}_2,\lambda_2} r$ we can not conclude anything about similarity between t and r.

The similarity rules have the same form as the equality rules: $\mathcal{K} \rightsquigarrow \mathcal{K}'$, which defines the transformation $\mathcal{K} \vee \mathcal{C} \rightsquigarrow \mathcal{K}' \vee \mathcal{C}$. The names are also similar to those for equalities, using sim instead of eq.

Del-sim : $\tau_1 \simeq^?_{\mathcal{R},\lambda} \tau_2 \wedge \mathcal{K} \rightsquigarrow \mathcal{K}$, where $\tau_1, \tau_2 \in \mathbf{C}_\mathsf{F} \cup \mathbf{V}_\mathsf{F} \cup \mathbf{V}_\mathsf{T}$ and $\mathcal{R}(\tau_1, \tau_2) \geq \lambda$.

Dec-sim : $\mathsf{f}(t_1, \dots, t_n) \simeq^?_{\mathcal{R},\lambda} \mathsf{g}(s_1, \dots, s_n) \wedge \mathcal{K} \rightsquigarrow$
$\mathsf{f} \simeq^?_{\mathcal{R},\lambda} \mathsf{g} \wedge t_1 \simeq^?_{\mathcal{R},\lambda} s_1 \wedge \cdots \wedge t_n \simeq^?_{\mathcal{R},\lambda} s_n \wedge \mathcal{K}$, where $n > 0$.

Ori-sim : $\tau \simeq^?_{\mathcal{R},\lambda} V \wedge \mathcal{K} \rightsquigarrow V \simeq^?_{\mathcal{R},\lambda} \tau \wedge \mathcal{K}$, where $\tau \notin \mathbf{V}$.

Elim-sim : $V \simeq^?_{\mathcal{R},\lambda} \tau \wedge \mathcal{K}_{\mathcal{R},\lambda} \wedge \mathcal{K} \rightsquigarrow V \simeq^?_{\mathcal{R},\lambda} \tau \wedge \mathcal{K}_{\mathcal{R},\lambda}\{V \mapsto \tau\} \wedge \mathcal{K}$
where $\mathcal{K}$ does not contain primitive $\simeq^?_{\mathcal{R},\lambda}$-constraints, $V \notin \mathit{var}(\tau)$, and $V \in \mathcal{K}_{\mathcal{R},\lambda}$.

Confl-sim : $\tau_1 \simeq^?_{\mathcal{R},\lambda} \tau_2 \wedge \mathcal{K} \rightsquigarrow$ false, where $\tau_1, \tau_2 \in \mathbf{C}_\mathsf{F} \cup \mathbf{V}_\mathsf{F} \cup \mathbf{V}_\mathsf{T}$, and $\mathcal{R}(\tau_1, \tau_2) < \lambda$.

Mism-sim : $\mathsf{f}(t_1, \dots, t_n) \simeq^?_{\mathcal{R},\lambda} \mathsf{g}(s_1, \dots, s_m) \wedge \mathcal{K} \rightsquigarrow$ false, if $n \neq m$.

Occ-sim : $X \simeq^?_{\mathcal{R},\lambda} t \wedge \mathcal{K} \rightsquigarrow$ false, if $X \in \mathit{var}(t)$ and $X \neq t$.

The algorithm Sim applies the similarity rules as long as possible. When there are more than one applicable rule, the algorithm may choose one nondeterministically.

Termination of Sim can be proved as termination of Unif:

▶ **Theorem 12.** Sim *is terminating.*

▶ **Lemma 13** (Soundness lemma for Sim)**.** *If $\mathcal{K} \rightsquigarrow \mathcal{K}'$ is a step performed by a rule in* Sim*, then $\mathit{Sol}(\mathcal{K}) = \mathit{Sol}(\mathcal{K}')$.*

Proof. When we have only one similarity relation, soundness follows from soundness of weak unification algorithm [17]. For the extension to multiple similarity relations, the only nontrivial rule is Elim-sim. (For the others, $\mathit{Sol}(\mathcal{K}) = \mathit{Sol}(\mathcal{K}')$ holds directly.) It is important to notice that in this rule, $\{V \mapsto \tau\}$ applies only to $\mathcal{K}_{\mathcal{R},\lambda}$. Then for $V \simeq^?_{\mathcal{R},\lambda} \tau \wedge \mathcal{K}_{\mathcal{R},\lambda}$ we have $\mathit{Sol}(V \simeq^?_{\mathcal{R},\lambda} \tau \wedge \mathcal{K}_{\mathcal{R},\lambda}) = \mathit{Sol}(V \simeq^?_{\mathcal{R},\lambda} \tau \wedge \mathcal{K}_{\mathcal{R},\lambda}\{V \mapsto \tau\})$. (It follows from soundness of weak unification algorithm [17], since the constraint is over a single similarity relation.) Constraints for all other relations remain unchanged. It implies that the solution sets for constraints in both sides of the Elim-sim rule are the same. ◀

▶ **Example 14.** let $\mathcal{K}_{\mathcal{R}_1,0.4} = X \simeq_{\mathcal{R}_1,0.4}$ *white-circle* $\wedge X \simeq_{\mathcal{R}_1,0.4} Y$ and $\mathcal{K}_{\mathcal{R}_2,0.5} = X \simeq_{\mathcal{R}_2,0.5}$ *gray-ellipse* $\wedge\, Y \simeq_{\mathcal{R}_2,0.5}$ *white-ellipse* be the constraints from Example 7. The reduction mentioned in that example is modeled by performing the Elim-sim step and replacing X by *white-circle* in $\mathcal{K}_{\mathcal{R}_1,0.4}$:

$$\begin{aligned}
&X \simeq_{\mathcal{R}_1,0.4} \mathit{white\text{-}circle} \wedge X \simeq_{\mathcal{R}_1,0.4} Y \wedge \\
&\quad X \simeq_{\mathcal{R}_2,0.5} \mathit{gray\text{-}ellipse} \wedge Y \simeq_{\mathcal{R}_2,0.5} \mathit{white\text{-}ellipse} \rightsquigarrow_{\mathsf{Elim\text{-}sim}} \\
&X \simeq_{\mathcal{R}_1,0.4} \mathit{white\text{-}circle} \wedge \mathit{white\text{-}circle} \simeq_{\mathcal{R}_1,0.4} Y \wedge \\
&\quad X \simeq_{\mathcal{R}_2,0.5} \mathit{gray\text{-}ellipse} \wedge Y \simeq_{\mathcal{R}_2,0.5} \mathit{white\text{-}ellipse}.
\end{aligned}$$

If Elim-sim permitted to replace X not only in $\mathcal{K}_{\mathcal{R}_1,0.4}$, but also in $\mathcal{K}_{\mathcal{R}_2,0.5}$, we would get

$$X \simeq_{\mathcal{R}_1,0.4} \textit{white-circle} \wedge \textit{white-circle} \simeq_{\mathcal{R}_1,0.4} Y \wedge$$
$$\textit{white-circle} \simeq_{\mathcal{R}_2,0.5} \textit{gray-ellipse} \wedge Y \simeq_{\mathcal{R}_2,0.5} \textit{white-ellipse},$$

but *white-circle* $\simeq_{\mathcal{R}_2,0.5}$ *gray-ellipse* is unsolvable. Hence, we would lose a solution.

Mixed rules

The rules in this section apply when there are at least two primitive constraints over different similarity relations. The notation $t[X]$ below means that the variable X occurs in the term t.

▶ **Definition 15** (Occurrence cycle). *An* occurrence cycle *for a variable X_1 is called the conjunction of primitive constraints $X_1 \simeq^?_{\mathcal{R}_1,\lambda_1} t_1[X_2] \wedge X_2 \simeq^?_{\mathcal{R}_2,\lambda_2} t_2[X_3] \wedge \cdots \wedge X_n \simeq^?_{\mathcal{R}_n,\lambda_n} t_n[X_1]$, where $n > 1$, $\mathcal{R}_i \neq \mathcal{R}_{i+1}$ for all $1 \leq i \leq n-1$, $\mathcal{R}_n \neq \mathcal{R}_1$, and at least one t is not a variable.*

▶ Remark 16. Note that in the definition of occurrence cycle, if two neighboring primitive similarity constraints use the same relation, they can be contracted into one constraint by transitivity, i.e., instead of $X_i \simeq^?_{\mathcal{R}_i,\lambda_i} t_i[X_{i+1}] \wedge X_{i+1} \simeq^?_{\mathcal{R}_i,\lambda_i} t_{i+1}[X_{i+2}]$ we can have $X_i \simeq^?_{\mathcal{R}_i,\lambda_i} t_i[t_{i+1}[X_{i+2}]]$, getting rid of consecutive identical similarity relations. The same is true for the last and the first constraints.

▶ **Theorem 17.** *If a conjunction of primitive constraints contains an occurrence cycle modulo symmetry of $\simeq^?_{\mathcal{R},\lambda}$, then it has no solution.*

Proof. In similarity relations, symbols of different arities can not be similar. Therefore, similar terms have the same set of positions, i.e., as trees they are the same up to renaming of nodes.

Assume by contradiction that the given occurrence cycle has a solution ϑ. It means that the following term pairs have the same structure: $X_1\vartheta$ and $t_1[X_2]\vartheta$, $X_2\vartheta$ and $t_2[X_3]\vartheta$, …, $X_n\vartheta$ and $t_n[X_1]\vartheta$. Then $X_1\vartheta$ and $t_1[t_2[\cdots[t_n[X_1]]\cdots]]\vartheta$ have the same structure. Since at least one of t_i's is not a variable, $X_1\vartheta$ is a proper subterm of $t_1[t_2[\cdots[t_n[X_1]]\cdots]]\vartheta$. But a term and its proper subterm can not have the same structure. A contradiction.

The phrase "modulo symmetry of $\simeq^?_{\mathcal{R},\lambda}$" in the theorem means that the sides of primitive constraints can be swapped, in order to detect an occurrence cycle. Since side swapping does not affect solvability of constraints, the theorem remains true if an occurrence cycle is not in the explicit form in the constraint. ◀

Below, when we talk about existence of an occurrence cycle in a constraint, we mean existence modulo symmetry of the similarity predicate.

The rules in the mixed group are rules for occurrence check (Occ-mix), mismatch (Mism-mix), and elimination of term and function variables (TVE-mix and FVE-mix, respectively). All of them except FVE-mix have the form $\mathcal{K} \rightsquigarrow \mathcal{K}'$. As usual, they define the constraint transformation $\mathcal{K} \vee \mathcal{C} \rightsquigarrow \mathcal{K}' \vee \mathcal{C}$. As for FVE-mix, its form is $\mathcal{K} \rightsquigarrow \mathcal{K}'_1 \vee \cdots \vee \mathcal{K}'_n$, defining a transformation $\mathcal{K} \vee \mathcal{C} \rightsquigarrow \mathcal{K}'_1 \vee \cdots \vee \mathcal{K}'_n \vee \mathcal{C}$. Note that in any of these rules, $\mathcal{C}$ does not change.

In all the rules it is assumed that the constraint to be transformed (i.e., the constraint in the left hand side of $\rightsquigarrow$) has the form $\mathcal{K}_{\doteq} \wedge \mathcal{K}_{\mathcal{R}_1,\lambda_1} \wedge \cdots \wedge \mathcal{K}_{\mathcal{R}_m,\lambda_m}$, where $\mathcal{K}_{\doteq}$ and each $\mathcal{K}_{\mathcal{R}_i,\lambda_i}$, $1 \leq i \leq m$, are in *solved form.*

The TVE-mix rule uses the *renaming function* ρ. Applied to a term, ρ gives its fresh copy, obtained by replacing each occurrence of a constant from $\mathbf{C}_\mathsf{F}$ by a new function variable, each occurrence of a term variable by a fresh term variable, and each occurrence of a function variable by a fresh function variable. For instance, if the term is $f(F(a, X, X, f(a)))$, we have $\rho(f(F(a, X, X, f(a)))) = G_1(G_2(G_3(), Y_1, Y_2, G_4(G_5())))$, where $G_1, G_2, G_3, G_4, G_5 \in \mathbf{V}_\mathsf{F}$ are new function variables and $Y_1, Y_2 \in \mathbf{V}_\mathsf{T}$ are new term variables.

Occ-mix : $X \simeq^?_{\mathcal{R},\lambda} t \wedge \mathcal{K} \rightsquigarrow \mathsf{false}$, if $X \simeq^?_{\mathcal{R},\lambda} t \wedge \mathcal{K}$ contains an occurrence cycle for X.

Mism-mix : $X \simeq^?_{\mathcal{R}_1,\lambda_1} \mathsf{f}(t_1, \dots, t_n) \wedge X \simeq^?_{\mathcal{R}_2,\lambda_2} \mathsf{g}(s_1, \dots, s_m) \wedge \mathcal{K} \rightsquigarrow \mathsf{false}$,
if $\mathcal{R}_1 \neq \mathcal{R}_2$ and $m \neq n$.

TVE-mix : $X \simeq^?_{\mathcal{R},\lambda} \mathsf{f}(t_1, \dots, t_n) \wedge \mathcal{K} \rightsquigarrow$
$$X \doteq F(t'_1, \dots, t'_n) \wedge F \simeq^?_{\mathcal{R},\lambda} \mathsf{f} \wedge t'_1 \simeq^?_{\mathcal{R},\lambda} t_1 \wedge \cdots \wedge t'_n \simeq^?_{\mathcal{R},\lambda} t_n \wedge \mathcal{K}\vartheta,$$
where $X \in \mathit{var}(\mathcal{K})$, $X \simeq^?_{\mathcal{R}_1,\lambda_1} \mathsf{f}(t_1, \dots, t_n) \wedge \mathcal{K}$ does not contain an occurrence cycle for X, $F(t'_1, \dots, t'_n) = \rho(\mathsf{f}(t_1, \dots, t_n))$, and $\vartheta = \{X \mapsto F(t'_1, \dots, t'_n)\}$.

FVE-mix : $F \simeq^?_{\mathcal{R},\lambda} f \wedge \mathcal{K} \rightsquigarrow \bigvee_{g \in \mathsf{N}(f,\mathcal{R},\lambda)} \big(F \doteq g \wedge \mathcal{K}\{F \mapsto g\}\big)$, where $F \in \mathit{var}(\mathcal{K})$.

By Mix we denote one application of any of the mixed rules.

▶ **Lemma 18** (Soundness lemma for Mix). *If $\mathcal{K} \rightsquigarrow \mathcal{C}$ is a step performed by a rule in* Mix*, and $\sigma \in \mathit{Sol}(\mathcal{C})$, then $\sigma \in \mathit{Sol}(\mathcal{K})$.*

Proof. For failing rules it is trivial as false has no solution. For FVE-mix, the definition of neighborhood implies it. For TVE-mix we reason as follows: Let $\mathcal{K} = \{X \simeq^?_{\mathcal{R},\lambda} \mathsf{f}(t_1, \dots, t_n)\} \wedge \mathcal{K}_1$ and σ be a solution of the right hand side of this rule. Then $X\sigma = F(t'_1, \dots, t'_n)\sigma \simeq_{\mathcal{R},\lambda} \mathsf{f}(t_1, \dots, t_n)\sigma$ and σ solves $X \simeq^?_{\mathcal{R},\lambda} \mathsf{f}(t_1, \dots, t_n)$. As for any other equation $\mathit{eq} \in \mathcal{K}_1$, we have $\mathit{eq}\vartheta$ in the right hand side, where $\vartheta = \{X \mapsto F(t'_1, \dots, t'_n)\}$. Moreover, σ is a solution of $\mathit{eq}\vartheta$ iff $\vartheta\sigma$ is a solution of eq. The equality $X\sigma = F(t'_1, \dots, t'_n)\sigma$ implies $\vartheta\sigma = \sigma$. Hence, σ is a solution of eq. ◀

Our constraint solving algorithm Solve is designed as a strategy of applying Unif, Sim, and Mix. To solve a conjunction of primitive equality and similarity constraints $\mathcal{K} = \mathcal{K}_{\doteq} \wedge \mathcal{K}_{\mathcal{R}_1,\lambda_1} \wedge \cdots \wedge \mathcal{K}_{\mathcal{R}_m,\lambda_m}$, it performs the following steps:

```
$\mathcal{C} := \mathcal{K}$
while $\mathcal{C}$ is not in the appr-solved form do
    $\mathcal{C} :=$ Unif($\mathcal{C}$), if $\mathcal{C} =$ false, return false
    $\mathcal{C} :=$ Sim($\mathcal{C}$), if $\mathcal{C} =$ false, return false
    $\mathcal{C} :=$ Mix($\mathcal{C}$), if $\mathcal{C} =$ false, return false
return $\mathcal{C}$
```

We write Solve($\mathcal{K}$) $= \mathcal{C}$, if the algorithm returns $\mathcal{C}$ for the input $\mathcal{K}$. Respectively, Solve($\mathcal{K}$) $=$ false if false is returned.

▶ **Example 19.** Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be the relations defined in Example 8 and illustrate the steps Solve would make to solve $X \simeq^?_{\mathcal{R}_1,0.5} f(a_1, a_2) \wedge X \simeq^?_{\mathcal{R}_2,0.6} f(Y, Y)$. We will explicitly distinguish between function variables and terms made of a function variable only, i.e., between F and $F()$. For the same reason, we write constant-terms a_1 and a_2 in their full form $a_1()$ and $a_2()$. Primitive constraints selected to perform a particular step are underlined.

$$\begin{aligned}
&\underline{X \simeq^?_{\mathcal{R}_1,0.5} f(a_1(),a_2())} \wedge X \simeq^?_{\mathcal{R}_2,0.6} f(Y,Y) \rightsquigarrow_{\mathsf{TVE\text{-}mix}}\\
&X \doteq F(G_1(),G_2()) \wedge F \simeq^?_{\mathcal{R}_1,0.5} f \wedge \underline{G_1() \simeq^?_{\mathcal{R}_1,0.5} a_1()} \wedge \underline{G_2() \simeq^?_{\mathcal{R}_1,0.5} a_2()} \wedge\\
&\qquad F(G_1(),G_2()) \simeq^?_{\mathcal{R}_2,0.6} f(Y,Y) \rightsquigarrow_{\mathsf{Dec\text{-}sim}\times 2}\\
&X \doteq F(G_1(),G_2()) \wedge F \simeq^?_{\mathcal{R}_1,0.5} f \wedge G_1 \simeq^?_{\mathcal{R}_1,0.5} a_1 \wedge G_2 \simeq^?_{\mathcal{R}_1,0.5} a_2 \wedge\\
&\qquad \underline{F(G_1(),G_2()) \simeq^?_{\mathcal{R}_2,0.6} f(Y,Y)} \rightsquigarrow_{\mathsf{Dec\text{-}sim,Ori\text{-}sim}}\\
&X \doteq F(G_1(),G_2()) \wedge F \simeq^?_{\mathcal{R}_1,0.5} f \wedge G_1 \simeq^?_{\mathcal{R}_1,0.5} a_1 \wedge G_2 \simeq^?_{\mathcal{R}_1,0.5} a_2 \wedge\\
&\qquad F \simeq^?_{\mathcal{R}_2,0.6} f \wedge \underline{Y \simeq^?_{\mathcal{R}_2,0.6} G_1()} \wedge G_2() \simeq^?_{\mathcal{R}_2,0.6} Y \rightsquigarrow_{\mathsf{Elim\text{-}sim}}\\
&X \doteq F(G_1(),G_2()) \wedge F \simeq^?_{\mathcal{R}_1,0.5} f \wedge G_1 \simeq^?_{\mathcal{R}_1,0.5} a_1 \wedge G_2 \simeq^?_{\mathcal{R}_1,0.5} a_2 \wedge\\
&\qquad F \simeq^?_{\mathcal{R}_2,0.6} f \wedge Y \simeq^?_{\mathcal{R}_2,0.6} G_1() \wedge \underline{G_2() \simeq^?_{\mathcal{R}_2,0.6} G_1()} \rightsquigarrow_{\mathsf{Dec\text{-}sim}}\\
&X \doteq F(G_1(),G_2()) \wedge \underline{F \simeq^?_{\mathcal{R}_1,0.5} f} \wedge G_1 \simeq^?_{\mathcal{R}_1,0.5} a_1 \wedge G_2 \simeq^?_{\mathcal{R}_1,0.5} a_2 \wedge\\
&\qquad F \simeq^?_{\mathcal{R}_2,0.6} f \wedge Y \simeq^?_{\mathcal{R}_2,0.6} G_1() \wedge G_2 \simeq^?_{\mathcal{R}_2,0.6} G_1 \rightsquigarrow_{\mathsf{FVE\text{-}mix}}\\
&X \doteq f(G_1(),G_2()) \wedge F \doteq f \wedge G_1 \simeq^?_{\mathcal{R}_1,0.5} a_1 \wedge G_2 \simeq^?_{\mathcal{R}_1,0.5} a_2 \wedge\\
&\qquad \underline{f \simeq^?_{\mathcal{R}_2,0.6} f} \wedge Y \simeq^?_{\mathcal{R}_2,0.6} G_1() \wedge G_2 \simeq^?_{\mathcal{R}_2,0.6} G_1 \rightsquigarrow_{\mathsf{Del\text{-}sim}}\\
&X \doteq f(G_1(),G_2()) \wedge F \doteq f \wedge \underline{G_1 \simeq^?_{\mathcal{R}_1,0.5} a_1} \wedge \underline{G_2 \simeq^?_{\mathcal{R}_1,0.5} a_2} \wedge\\
&\qquad Y \simeq^?_{\mathcal{R}_2,0.6} G_1() \wedge G_2 \simeq^?_{\mathcal{R}_2,0.6} G_1 \rightsquigarrow_{\mathsf{FVE\text{-}mix}\times 2} \text{(showing only successful branches)}\\
&\big(X \doteq f(b_1(),b_2()) \wedge F \doteq f \wedge G_1 \doteq b_1 \wedge G_2 \doteq b_2 \wedge\\
&\qquad Y \simeq^?_{\mathcal{R}_2,0.6} b_1() \wedge \underline{b_2 \simeq^?_{\mathcal{R}_2,0.6} b_1}\big) \vee\\
&\big(X \doteq f(c_1(),c_2()) \wedge F \doteq f \wedge G_1 \doteq c_1 \wedge G_2 \doteq c_2 \wedge\\
&\qquad Y \simeq^?_{\mathcal{R}_2,0.6} c_1() \wedge \underline{c_2 \simeq^?_{\mathcal{R}_2,0.6} c_1}\big) \rightsquigarrow_{\mathsf{Del\text{-}sim}\times 2}\\
&\big(X \doteq f(b_1(),b_2()) \wedge F \doteq f \wedge G_1 \doteq b_1 \wedge G_2 \doteq b_2 \wedge Y \simeq^?_{\mathcal{R}_2,0.6} b_1()\big) \vee\\
&\big(X \doteq f(c_1(),c_2()) \wedge F \doteq f \wedge G_1 \doteq c_1 \wedge G_2 \doteq c_2 \wedge Y \simeq^?_{\mathcal{R}_2,0.6} c_1()\big).
\end{aligned}$$

Restricting the obtained result to the original variables (and writing constant-terms in the conventional way), we get the solved form

$$(X \doteq f(b_1,b_2) \wedge Y \simeq_{\mathcal{R}_2,0.6} b_1) \vee (X \doteq f(c_1,c_2) \wedge Y \simeq_{\mathcal{R}_2,0.6} c_1).$$

▶ **Example 20.** Now we show how Solve computes an appr-solved form for the constraint from Example 9:

$$\begin{aligned}
&\underline{X \simeq^?_{\mathcal{R}_1,0.6} f(Y,Y)} \wedge X \simeq^?_{\mathcal{R}_2,0.5} f(Z,Z) \rightsquigarrow_{\mathsf{TVE\text{-}mix}}\\
&X \doteq F(X_1,X_2) \wedge \underline{F \simeq^?_{\mathcal{R}_1,0.6} f} \wedge X_1 \simeq^?_{\mathcal{R}_1,0.6} Y \wedge X_2 \simeq^?_{\mathcal{R}_1,0.6} Y \wedge\\
&\qquad F(X_1,X_2) \simeq^?_{\mathcal{R}_2,0.5} g(Z,Z) \rightsquigarrow_{\mathsf{FVE\text{-}mix}}\\
&X \doteq f(X_1,X_2) \wedge F \simeq_{\mathcal{R}_1,0.6} f \wedge X_1 \simeq^?_{\mathcal{R}_1,0.6} Y \wedge X_2 \simeq^?_{\mathcal{R}_1,0.6} Y \wedge\\
&\qquad \underline{f(X_1,X_2) \simeq^?_{\mathcal{R}_2,0.5} g(Z,Z)} \rightsquigarrow_{\mathsf{Dec}}\\
&X \doteq f(X_1,X_2) \wedge F \simeq^?_{\mathcal{R}_1,0.6} f \wedge X_1 \simeq^?_{\mathcal{R}_1,0.6} Y \wedge X_2 \simeq^?_{\mathcal{R}_1,0.6} Y \wedge\\
&\qquad X_1 \simeq^?_{\mathcal{R}_2,0.5} Z \wedge X_2 \simeq^?_{\mathcal{R}_2,0.5} Z.
\end{aligned}$$

The result gives an appr-solved form. If we did not generate new copies for each variable occurrence in the TVE-mix rule, we would end up with $X \doteq^? f(Z,Z) \wedge F \doteq^? f \wedge Y \doteq^? Z$. As we saw in Example 9, some solutions would be lost in this case.

To prove termination of Solve, we will need an ordering on directed acyclic graphs (dags).

▶ **Definition 21** (The relation $<_{\text{dag}}$). *We consider dags, which have a finite set of symbols associated to each vertex. These sets are called the marks of vertices. For a graph G and a vertex v, we denote the mark of v in G by $mark(v, G)$. The relation $<_{\text{dag}}$ is defined on such graphs, having the same set of vertices.*

Let $G_1 = (Vert, E_1)$ and $G_2 = (Vert, E_2)$ be two dags with the same set of vertices $Vert$. The vertices are marked in G_1 and G_2. Then $G_1 <_{\text{dag}} G_2$ iff $E_1 \supseteq E_2$ and the following condition holds:

- *Let $\emptyset \neq D \subseteq Vert$ be the set of all vertices, for which the marks in the graphs differ (i.e., $D := \{v \in Vert \mid mark(v, G_1) \neq mark(v, G_2)\}$: same vertex, different markings), and $\emptyset \neq M \subseteq D$ be the set of those elements of D, which are not reachable from any of the elements in D in G_1. (Such a subset of D exists, because the graphs are acyclic.) Then $mark(v, G_1) \subset mark(v, G_2)$ for all $v \in M$.*

We write $G >_{\text{dag}} G'$ if $G' <_{\text{dag}} G$.

▶ **Theorem 22.** *The relation $>_{\text{dag}}$ is a well-founded ordering on dags.*

Proof. First, we show that $>_{\text{dag}}$ is a strict partial order (irreflexive and transitive relation). Irreflexivity is obvious. For transitivity, assume $G_1 >_{\text{dag}} G_2$, $G_2 >_{\text{dag}} G_3$ and show $G_1 >_{\text{dag}} G_3$. Let $G_i = (Vert, E_i)$ for $i = 1, 2, 3$. By transitivity of set inclusion, we have $E_1 \subseteq E_3$.

Let $D(G_i, G_j) := \{v \in Vert \mid mark(v, G_i) \neq mark(v, G_j)\}$ and $M(G_i, G_j)$ be the set of all those elements in $D(G_i, G_j)$ that are not reachable in G_j from $D(G_i, G_j)$, $1 \leq i < j \leq 3$. Assume first that $D(G_1, G_3) \neq \emptyset$ and take $v \in M(G_1, G_3)$. We want to show that $mark(v, G_1) \supset mark(v, G_3)$. The possible cases are

i. $mark(v, G_1) \neq mark(v, G_2)$ and $mark(v, G_2) = mark(v, G_3)$.
ii. $mark(v, G_1) = mark(v, G_2)$ and $mark(v, G_2) \neq mark(v, G_3)$.
iii. $mark(v, G_1) \neq mark(v, G_2)$ and $mark(v, G_2) \neq mark(v, G_3)$.

In case **i**, $v \in D(G_1, G_2)$. If $v \in M(G_1, G_2)$, then $mark(v, G_1) \supset mark(v, G_2) = mark(v, G_3)$. Now we show that the case $v \notin M(G_1, G_2)$ is impossible. Assume by contradiction that $v \notin M(G_1, G_2)$. It means that there exists $v' \in M(G_1, G_2)$ such that $v' \to^+ v$ in G_2. But then, since $E_2 \subseteq E_3$, we have $v' \to^+ v$ in G_3. Since $v \in M(G_1, G_3)$, we should have $v' \notin D(G_1, G_3)$, i.e., $mark(v', G_1) = mark(v', G_3)$. On the other hand, from $G_1 >_{\text{dag}} G_2$ and $v' \in M(G_1, G_2)$, we have $mark(v', G_1) \supset mark(v', G_2)$, which implies $mark(v', G_2) \subset mark(v', G_3)$, $v' \in D(G_2, G_3)$ and $v' \notin M(G_2, G_3)$. Then there should exist $v'' \in M(G_2, G_3)$ such that $v'' \to^+ v'$ in G_3. Hence, we get $v'' \to^+ v' \to^+ v$ in G_3. Therefore, we can not have $v'' \in D(G_1, G_3)$, because there would be a contradiction: $v \in M(G_1, G_3)$ and v is reachable in G_3 from $v'' \in D(G_1, G_3)$. Hence, we get $mark(v'', G_1) = mark(v'', G_3)$, which, together with $v'' \in M(G_2, G_3)$ implies that $mark(v'', G_1) \subset mark(v'', G_2)$. Hence, $v'' \in D(G_1, G_2)$ and since $G_1 >_{\text{dag}} G_2$, there should exist $v''' \in M(G_1, G_2)$ such that $mark(v''', G_1) \supset mark(v''', G_2)$ and $v''' \to^+ v''$ in G_2. Then we have also $v''' \to^+ v''$ in G_3, since $E_2 \subseteq E_3$. Moreover, $mark(v''', G_2) = mark(v''', G_3)$, because otherwise we would have a contradiction with $v'' \in M(G_2, G_3)$ (there would be $v''' \in D(G_2, G_3)$ with $v''' \to^+ v''$ in G_3). Hence, $mark(v''', G_1) \supset mark(v''', G_2) = mark(v''', G_3)$ and we get $v''' \in D(G_1, G_3)$. But it contradicts our assumption that $v \in M(G_1, G_3)$, because we got $v''' \in D(G_1, G_3)$ with $v''' \to^+ v$ in G_3. The obtained contradiction shows that the case $v \notin M(G_1, G_2)$ is impossible. (See also the diagram in Appendix A.)

The case **ii** can be proved analogously. In case **iii**, if we have $\mathit{mark}(v, G_1) \supset \mathit{mark}(v, G_2)$ and $\mathit{mark}(v, G_2) \supset \mathit{mark}(v, G_3)$, we immediately get $\mathit{mark}(v, G_1) \supset \mathit{mark}(v, G_3)$. The other cases are not possible. For instance, assuming $\mathit{mark}(v, G_1) \supset \mathit{mark}(v, G_2)$ and $\mathit{mark}(v, G_2) \subset \mathit{mark}(v, G_3)$ will lead to contradiction by the same reasoning as in the proof of case **i**, where we reached $\mathit{mark}(v', G_1) \supset \mathit{mark}(v', G_2)$ and $\mathit{mark}(v', G_2) \subset \mathit{mark}(v', G_3)$.

For the assumption $D(G_1, G_3) = \emptyset$ we get a contradiction analogously.

Well-foundedness follows from well-foundnedness of $\supset$, from the facts that the set of vertices is fixed and the set of edges can not be infinitely increased, and from boundedness of the length of paths due to acyclicity. ◀

▶ **Theorem 23** (Termination of Solve). *The algorithm* Solve *terminates and gives either* false *or a constraint in appr-solved form.*

Proof. Let $\mathcal{K}_0$ be a given conjunction of primitive constraints. We build a term variable dependency graph $G = (\mathit{Vert}, E)$ from $\mathcal{K}_0$ and maintain it during the process of solving. The vertices of G correspond to term variables in $\mathcal{K}_0$ so that to each variable a single vertex is assigned. For instance, if $\mathcal{K}_0$ contains three term variables X_1, X_2, and Y_3, then $\mathit{Vert} = \{v_{X_1}, v_{X_2}, v_{X_3}\}$. The initial marking is defined as $\mathit{mark}(v_X, G) = \{X\}$ for each $v_X \in \mathit{Vert}$. Next, to define E, we do the following: If $\mathcal{K}_0$ contains a solved primitive constraint $X \doteq^? t$ or $X \simeq^?_{\mathcal{R},\lambda} t$ (i.e., if X occurs in $\mathcal{K}_0$ once), then $(v_X, v_Y) \in E$ for all $Y \in \mathit{var}(t)$ and $\mathit{mark}(v_X, G)$ is updated as $\mathit{mark}(v_X, G) \setminus \{X\}$. This is how the initial version of the graph is created. It is denoted by $G_{\mathcal{K}_0}$.

In the process of the application of Solve, the graph gets modified as follows:

a) *The applied rule is of the form $\mathcal{K} \rightsquigarrow \mathcal{K}'$.* Then from the graph $G_{\mathcal{K}}$ we obtain the graph $G_{\mathcal{K}'}$ depending on the rule:
   - Elim-eq with $X \doteq^? t$ adds edges and removes X from the marking set of the vertex v_X exactly as described above.
   - Elim-sim with $X \simeq^?_{\mathcal{R},\lambda} t$ either keeps the graph unchanged (when X occurs more than once in the resulting constraint after the application of Elim-sim), or modifies it as described above (when X occurs exactly once in the resulting constraint after the application of Elim-sim).
   - TVE-mix with $X \simeq^?_{\mathcal{R},\lambda} t$ does the same modification as Elim-eq and, in addition, modifies marking: Let $Y_1, \dots, Y_m$, $m \geq 0$, be all the copies of a term variable $Y \in \mathit{var}(t)$ created by the renaming function ρ in the TVE-mix step. Then each Y_i is associated with the vertex v_Y (i.e., $v_{Y_i} = v_Y$) and $\mathit{mark}(v_Y, G)$ gets updated as $\mathit{mark}(v_Y, G) \cup \{Y_1, \dots, Y_m\}$.
   - No other rule of the form $\mathcal{K} \rightsquigarrow \mathcal{K}'$ modifies the graph.

b) *The applied rule is of the form $\mathcal{K} \rightsquigarrow \mathcal{K}'_1 \vee \cdots \vee \mathcal{K}'_n$, $n > 1$.* Then $G_{\mathcal{K}} = G_{\mathcal{K}'_1} = \cdots = G_{\mathcal{K}'_n}$.

One can see that the set Vert remains unchanged during the process.

Let $G_1 = (\mathit{Vert}, E_1)$ be the graph before applying a rule, and $G_2 = (\mathit{Vert}, E_2)$ be the one after a rule application so that $G_1 \neq G_2$, i.e., Elim-eq, Elim-sim, or TVE-mix is applied. Let the chosen primitive constraint be $X \doteq^? t$ (for Elim-eq) or $X \simeq^?_{\mathcal{R},\lambda} t$ (for Elim-sim and TVE-mix). Then $E_1 \subseteq E_2$, because new edges from v_X to v_Y for each $Y \in \mathit{var}(t)$ is created and none are removed. Besides, $\mathit{mark}(X, G_1) \supset \mathit{mark}(X, G_2) = \mathit{mark}(X, G_1) \setminus \{X\}$, and markings in G_2 are not changed for any of the vertices which is not reachable from X in G_2. Hence, $G_1 >_{\mathrm{dag}} G_2$.

Let us consider the pair $(G_{\mathcal{K}}, \mathbf{V}_{\mathsf{F}}(\mathcal{K}))$ of such a term variable dependency graph $G_{\mathcal{K}}$ associated to a constraint $\mathcal{K}$ and a set of function variables $\mathbf{V}_{\mathsf{F}}(\mathcal{K})$ occurring in $\mathcal{K}$. These pairs are ordered lexicographically by $>_{\mathrm{dag}}$ and $>$. By Theorem 22, $>_{\mathrm{dag}}$ is well-founded. The relation $>$ on natural numbers is well-founded. Therefore, their lexicographic combination is well-founded. Since Unif and Sim are terminating (Theorems 10 and 12), each iteration of the **while** loop in the definition of Solve either stops with false, or reaches the application of Mix. In this process, measure of the pair $(G_{\mathcal{K}}, \mathbf{V}_{\mathsf{F}}(\mathcal{K}))$ does not increase, because the Unif and Sim rules, as we have already seen, either decrease or keep unchanged $G_{\mathcal{K}}$, and do not add new function variables. The application of Mix either fails, or strictly reduces $(G_{\mathcal{K}}, \mathbf{V}_{\mathsf{F}}(\mathcal{K}))$: TVE-mix strictly decreases $G_{\mathcal{K}}$, and FVE-mix does not change $G_{\mathcal{K}}$ but strictly decreases $\mathbf{V}_{\mathsf{F}}(\mathcal{K})$. Hence, the **while** loop in Solve can be executed only finitely many times.

If Solve does not stop with false, the only possible non-solved primitive constraints are those between variables, whose left hand side has occurrences in at least two different kind of constraints. For any other case, there is an applicable rule. Hence, the obtained constraint is in appr-solved form. ◀

▶ **Theorem 24** (Soundness of Solve)**.** *Let $\mathcal{K}$ be a conjunction of primitive constraints. Then every solution of the constraint* $\mathsf{Solve}(\mathcal{K})$ *is a solution of $\mathcal{K}$.*

Proof. By induction on the length of a rule application sequence leading from $\mathcal{K}$ to $\mathsf{Solve}(\mathcal{K})$, using the soundness lemmas for equality, similarity, and mixed rules (Lemmas 11, 13, 18). ◀

▶ **Theorem 25** (Completeness of Solve)**.** *Let $\mathcal{K}$ be a conjunction of primitive constraints, and ϑ be its solution. Then* $\mathsf{Solve}(\mathcal{K})$ *is a constraint $(\mathcal{K}_{\mathrm{sol}} \wedge \mathcal{K}_{\mathrm{var}}) \vee \mathcal{C}$, where $\mathcal{K}_{\mathrm{sol}} \wedge \mathcal{K}_{\mathrm{var}}$ is in appr-solved form, and $\sigma_{\mathcal{K}_{\mathrm{sol}}}\sigma_{\mathcal{K}_{\mathrm{var}}} \preceq_{\mathcal{W}(\mathcal{K})} \vartheta$, where $\sigma_{\mathcal{K}_{\mathrm{sol}}}$ is the substitution induced by $\mathcal{K}_{\mathrm{sol}}$, and $\sigma_{\mathcal{K}_{\mathrm{var}}}$ is a solution of $\mathcal{K}_{\mathrm{var}}$.*

Proof. In the proof we use completeness of unification and weak unification algorithms [4,9,17]. First, note that if one of the failure rules is applicable to a constraint, then it has no solution. For Occ-mix it follows from Theorem 17. For Mism-mix, it is guaranteed by the fact that symbols with different arities are not similar. For failure rules in Unif and Sim it is known from their completeness results.

Application of Unif to $\mathcal{K}$ leads to a new constraint $\mathcal{C}_{\mathrm{un}}$, which contains a solved from $\mathcal{K}_{\text{un-sol}}$ such that $\sigma_{\mathcal{K}_{\text{un-sol}}} \preceq_{\mathcal{W}(\mathcal{K})} \vartheta$. Application of Sim to $\mathcal{C}_{\mathrm{un}}$ gives $\mathcal{C}_{\mathrm{sim}}$, which contains a solved form $\mathcal{K}_{\text{sim-sol}}$ (an extension of $\mathcal{K}_{\text{un-sol}}$) such that $\sigma_{\mathcal{K}_{\text{sim-sol}}} \preceq_{\mathcal{W}(\mathcal{K})} \vartheta$. After that, if TVE-mix is applicable, we have an equation $X \simeq^{?}_{\mathcal{R},\lambda} \mathsf{f}(t_1, \ldots, t_n)$. TVE-mix extends the solved form by a new equation $X \doteq^{?} \mathsf{\rho}(\mathsf{f}(t_1, \ldots, t_n))$, obtaining $\mathcal{K}_{\text{tve-sol}}$. By definition of $\mathsf{\rho}$, the term $\mathsf{\rho}(\mathsf{f}(t_1, \ldots, t_n))$ contains fresh variables for each symbol in $\mathsf{f}(t_1, \ldots, t_n)$ and, hence, $\sigma_{\mathcal{K}_{\text{tve-sol}}} \preceq_{\mathcal{W}(\mathcal{K})} \vartheta$. It is important at this step to record which fresh variable is a copy of which original variable, maintaining an function $\mathit{original\text{-}of}(V') = V$, where $V \in \mathit{var}(\mathcal{K})$ and V' is zero or more applications of $\mathsf{\rho}$ to it (i.e., V' is V, or its copy, or a copy of its copy etc.). If the rule FVE-mix is applicable, we have an equation $F \simeq^{?}_{\mathcal{R},\lambda} f$. We make a step by this rule, adding a new equation $F \doteq^{?} \mathit{original\text{-}of}(F)\vartheta$ and obtaining a new solved form $\mathcal{K}_{\text{fve-sol}}$. Let φ be the substitution $\{\mathit{original\text{-}of}(F) \mapsto \mathit{original\text{-}of}(F)\vartheta\}$. Then we have $\sigma_{\mathcal{K}_{\text{fve-sol}}}\varphi \preceq_{\mathcal{W}(\mathcal{K})} \vartheta$. Iterating this process, we do not get false, since $\mathcal{K}$ was solvable. By Theorem 23, the process terminates with an appr-solved form $\mathcal{K}_{\mathrm{sol}} \wedge \mathcal{K}_{\mathrm{var}}$ such that $\sigma_{\mathcal{K}_{\mathrm{sol}}}\varphi_1 \cdots \varphi_k \preceq_{\mathcal{W}(\mathcal{K})} \vartheta$, where the φ's are substitutions of the form $\{\mathit{original\text{-}of}(V) \mapsto \mathit{original\text{-}of}(V)\vartheta\}$. Let $\sigma_{\mathcal{K}_{\mathrm{var}}}$ be the restriction of ϑ to the variables of $\mathcal{K}_{\mathrm{var}}$. Then $\sigma_{\mathcal{K}_{\mathrm{var}}}$ is a solution of $\mathcal{K}_{\mathrm{var}}$. And we have $\sigma_{\mathcal{K}_{\mathrm{sol}}}\varphi_1 \cdots \varphi_k\sigma_{\mathcal{K}_{\mathrm{var}}} \preceq_{\mathcal{W}(\mathcal{K})} \vartheta$.

For the φ's, composition is commutative, because they are ground substitutions with disjoint domains. For some of them we have $\sigma_{\mathcal{K}_{\mathrm{sol}}}\varphi_i = \sigma_{\mathcal{K}_{\mathrm{sol}}}$, because at some step we might have solved an equation with $\mathit{original\text{-}of}(V)$ variable in its left for $\mathit{original\text{-}of}(V) \in \mathit{dom}(\varphi_i)$. We assume that the φ's in the composition are rearranged so that $\sigma_{\mathcal{K}_{\mathrm{sol}}}\varphi_1 \cdots \varphi_i =$

$\sigma_{\mathcal{K}_{\mathrm{sol}}}\varphi_{i+1}\cdots\varphi_k$. These remaining φ's are those for which the algorithm reached a variables-only equation containing *original-of*(V), which occurs in the domain of one of the φ's. But then $\varphi_{i+1}\cdots\varphi_k$ is a part of $\sigma_{\mathcal{K}_{\mathrm{var}}}$. Hence, we can get rid of them, obtaining $\sigma_{\mathcal{K}_{\mathrm{sol}}}\sigma_{\mathcal{K}_{\mathrm{var}}}$. Hence, we get $\sigma_{\mathcal{K}_{\mathrm{sol}}}\sigma_{\mathcal{K}_{\mathrm{var}}} \preceq_{\mathcal{W}(\mathcal{K})} \vartheta$. ◀

4 Computing approximation degrees

In the algorithm, we have not included the computation of approximation degrees, but it can be done easily. Instead of constraints in DNF of the form $\mathcal{K}_1 \vee \cdots \vee \mathcal{K}_n$, we will be working with expressions (we call them extended constraints) $(K_1, \mathfrak{D}_1) \vee \cdots \vee (K_n, \mathfrak{D}_n)$, where $\mathfrak{D}_1, \ldots, \mathfrak{D}_n$ are approximation degrees. The rules will carry the degree ("computed so far") as an additional parameter, but only two rules would change them: Del-sim and FVE-mix. Their variants with degree modification would work on constraint-degree pairs ($\uplus$ stands for disjoint union):

Del-sim-deg : $\big(\tau_1 \simeq^{?}_{\mathcal{R},\lambda} \tau_2 \wedge \mathcal{K},\ \{\langle \mathcal{R}, \mathfrak{d}\rangle\} \uplus \mathfrak{D}\big) \rightsquigarrow (\mathcal{K},\ \{\langle \mathcal{R}, \min\{\mathfrak{d}, \mathcal{R}(\tau_1,\tau_2)\}\rangle\} \cup \mathfrak{D})$
where $\tau_1, \tau_2 \in \mathbf{C}_{\mathsf{F}} \cup \mathbf{V}_{\mathsf{F}} \cup \mathbf{V}_{\mathsf{T}}$ and $\mathcal{R}(\tau_1,\tau_2) \geq \lambda$.

FVE-mix-deg : $\big(F \simeq^{?}_{\mathcal{R},\lambda} f \wedge \mathcal{K},\ \{\langle \mathcal{R}, \mathfrak{d}\rangle\} \uplus \mathfrak{D}\big) \rightsquigarrow$
$\bigvee_{g \in \mathsf{N}(f,\mathcal{R},\lambda)} \big(F \doteq g \wedge \mathcal{K}\{F \mapsto g\},\ \{\langle \mathcal{R}, \min\{\mathfrak{d}, \mathcal{R}(f,g)\}\rangle\} \cup \mathfrak{D}\big)$,
where $F \in \mathit{var}(\mathcal{K})$.

For any other rule R of the form $\mathcal{K} \rightsquigarrow \mathcal{K}_1 \vee \cdots \vee \mathcal{K}_n$, $n \geq 1$, its degree variant R-deg will have the form $(\mathcal{K}, \mathfrak{D}) \rightsquigarrow (\mathcal{K}_1, \mathfrak{D}) \vee \cdots \vee (\mathcal{K}_n, \mathfrak{D})$, i.e., $\mathfrak{D}$ will not change. Let us denote the corresponding versions of Unif, Sim, and Mix by Unif-deg, Sim-deg, and Mix-deg. The notions of solved and approx-solved forms generalize directly to extended constraints. Then we can define Solve-deg along the lines of Solve: To solve a conjunction of primitive equality and similarity constraints $\mathcal{K}$ with respect to similarity relations $\mathcal{R}_1, \ldots, \mathcal{R}_m$, it performs the following steps:

```
$\mathcal{C} := (\mathcal{K}, \{\langle \mathcal{R}_1, 1\rangle, \ldots, \langle \mathcal{R}_m, 1\rangle\})$
while $\mathcal{C}$ is not in the appr-solved form do
    $\mathcal{C} :=$ Unif-deg($\mathcal{C}$), if $\mathcal{C} =$ false, return false
    $\mathcal{C} :=$ Sim-deg($\mathcal{C}$), if $\mathcal{C} =$ false, return false
    $\mathcal{C} :=$ Mix-deg($\mathcal{C}$), if $\mathcal{C} =$ false, return false
return $\mathcal{C}$
```

5 Discussion and summary

The proposed solver can be used in constraint-based formalisms such as, for instance, constraint logic programming [11] or term rewriting with constraints [13]. We can envisage an instance of the CLP schema with constraints over multiple similarities. Without going into much details, a simple constraint logic program below can illustrate this possibility:

▶ **Example 26.** The letter P in the program stands for a predicate variable. In constraints, it is treated as a function variable.

$$P(X,Y) \leftarrow P \simeq_{\mathcal{R}_1,\lambda_1} p,\ X \simeq_{\mathcal{R}_1,\lambda_1} F(Y),\ Y \simeq_{\mathcal{R}_2,\lambda_2} c,\ r(X),\ r(F(Y)).$$
$$r(F(X)) \leftarrow F \simeq_{\mathcal{R}_2,\lambda_2} h,\ X \simeq_{\mathcal{R}_1,\lambda_1} a.$$

Assume $\mathcal{R}_1(p,q) = 0.9$, $\mathcal{R}_1(a,b) = 0.8$, $\mathcal{R}_1(f,g) = 0.6$, $\mathcal{R}_2(g,h) = 0.5$, $\mathcal{R}_2(b,c) = 0.7$, $\lambda_1 = \lambda_2 = 0.4$. Then by performing the usual CLP inference (i.e., syntactically unifying the selected query and the head of the corresponding clause) for the query $\leftarrow q(X,Y)$, the resulting constraint (together with the approximation degrees) will be $(X \doteq g(b) \wedge Y \doteq b;$ $\{\langle\mathcal{R}_1, 0.8\rangle, \langle\mathcal{R}_2, 0.5\rangle\}) \vee (X \doteq h(b) \wedge Y \doteq b; \{\langle\mathcal{R}_1, 0.8\rangle, \langle\mathcal{R}_2, 0.7\rangle\})$.

To summarize, the algorithm Solve presented in the paper solves positive equational and similarity constraints, where multiple similarity relations are permitted. Given such a constraint in DNF, it computes disjunction of approximately solved forms, from which solution substitutions can be read off. It can be easily extended to include the computation of approximation degrees of the solutions. The algorithm is terminating, sound, and complete.

References

1 Hassan Aït-Kaci and Gabriella Pasi. Fuzzy unification and generalization of first-order terms over similar signatures. In Fabio Fioravanti and John P. Gallagher, editors, *Logic-Based Program Synthesis and Transformation - 27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017, Revised Selected Papers*, volume 10855 of *Lecture Notes in Computer Science*, pages 218–234. Springer, 2017. doi:10.1007/978-3-319-94460-9_13.

2 Hassan Aït-Kaci and Gabriella Pasi. Lattice operations on terms with fuzzy signatures. *CoRR*, abs/1709.00964, 2017. arXiv:1709.00964.

3 Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

4 Franz Baader and Wayne Snyder. Unification theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 445–532. Elsevier and MIT Press, 2001. doi:10.1016/b978-044450813-3/50010-2.

5 Francesca Arcelli Fontana and Ferrante Formato. Likelog: A logic programming language for flexible data retrieval. In Barrett R. Bryant, Gary B. Lamont, Hisham Haddad, and Janice H. Carroll, editors, *Proceedings of the 1999 ACM Symposium on Applied Computing, SAC'99, San Antonio, Texas, USA, February 28 - March 2, 1999*, pages 260–267. ACM, 1999. doi:10.1145/298151.298348.

6 Francesca Arcelli Fontana and Ferrante Formato. A similarity-based resolution rule. *Int. J. Intell. Syst.*, 17(9):853–872, 2002. doi:10.1002/int.10067.

7 Ferrante Formato, Giangiacomo Gerla, and Maria I. Sessa. Extension of logic programming by similarity. In Maria Chiara Meo and Manuel Vilares Ferro, editors, *1999 Joint Conference on Declarative Programming, AGP'99, L'Aquila, Italy, September 6-9, 1999*, pages 397–410, 1999.

8 Ferrante Formato, Giangiacomo Gerla, and Maria I. Sessa. Similarity-based unification. *Fundam. Inform.*, 41(4):393–414, 2000. doi:10.3233/FI-2000-41402.

9 Pascual Julián Iranzo and Clemente Rubio-Manzano. A sound and complete semantics for a similarity-based logic programming language. *Fuzzy Sets and Systems*, 317:1–26, 2017. doi:10.1016/j.fss.2016.12.016.

10 Pascual Julián Iranzo and Fernando Sáenz-Pérez. An efficient proximity-based unification algorithm. In *2018 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2018, Rio de Janeiro, Brazil, July 8-13, 2018*, pages 1–8. IEEE, 2018. doi:10.1109/FUZZ-IEEE.2018.8491593.

11 Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994. doi:10.1016/0743-1066(94)90033-7.

12 Pascual Julián-Iranzo and Clemente Rubio-Manzano. Proximity-based unification theory. *Fuzzy Sets and Systems*, 262:21–43, 2015. doi:10.1016/j.fss.2014.07.006.

13 Cynthia Kop and Naoki Nishida. Term rewriting with logical constraints. In Pascal Fontaine, Christophe Ringeissen, and Renate A. Schmidt, editors, *Frontiers of Combining Systems - 9th International Symposium, FroCoS 2013, Nancy, France, September 18-20, 2013. Proceedings*, volume 8152 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2013. doi:10.1007/978-3-642-40885-4_24.

14 Stanislav Krajci, Rastislav Lencses, Jesús Medina, Manuel Ojeda-Aciego, and Peter Vojtás. A similarity-based unification model for flexible querying. In Troels Andreasen, Amihai Motro, Henning Christiansen, and Henrik Legind Larsen, editors, *Flexible Query Answering Systems, 5th International Conference, FQAS 2002, Copenhagen, Denmark, October 27-29, 2002, Proceedings*, volume 2522 of *Lecture Notes in Computer Science*, pages 263–273. Springer, 2002. doi:10.1007/3-540-36109-X_21.

15 Temur Kutsia and Cleo Pau. Solving proximity constraints. In Maurizio Gabbrielli, editor, *Logic-Based Program Synthesis and Transformation - 29th International Symposium, LOPSTR 2019, Porto, Portugal, October 8-10, 2019, Revised Selected Papers*, volume 12042 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2019. doi:10.1007/978-3-030-45260-5_7.

16 Jesús Medina, Manuel Ojeda-Aciego, and Peter Vojtás. Similarity-based unification: a multi-adjoint approach. *Fuzzy Sets and Systems*, 146(1):43–62, 2004. doi:10.1016/j.fss.2003.11.005.

17 Maria I. Sessa. Approximate reasoning by similarity-based SLD resolution. *Theor. Comput. Sci.*, 275(1-2):389–426, 2002. doi:10.1016/S0304-3975(01)00188-8.

18 Mariya I. Vasileva, Bryan A. Plummer, Krishna Dusad, Shreya Rajpal, Ranjitha Kumar, and David A. Forsyth. Learning type-aware embeddings for fashion compatibility. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors, *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XVI*, volume 11220 of *Lecture Notes in Computer Science*, pages 405–421. Springer, 2018. doi:10.1007/978-3-030-01270-0_24.

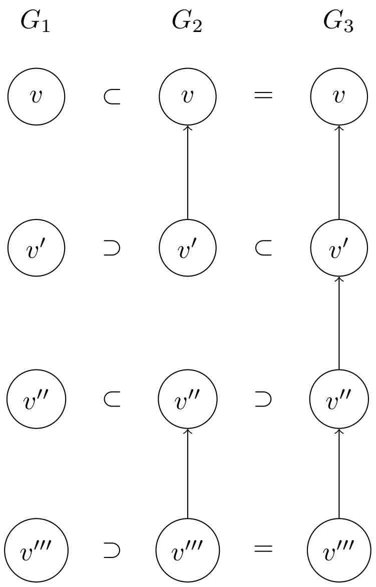
19 Andreas Veit, Serge J. Belongie, and Theofanis Karaletsos. Conditional similarity networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 1781–1789. IEEE Computer Society, 2017. doi:10.1109/CVPR.2017.193.

20 Harry Virtanen. Vague domains, s-unification, logic programming. *Electr. Notes Theor. Comput. Sci.*, 66(5):86–103, 2002. doi:10.1016/S1571-0661(04)80516-4.

21 Peter Vojtás. Declarative and procedural semantics of fuzzy similarity based unification. *Kybernetika*, 36(6):707–720, 2000. URL: http://www.kybernetika.cz/content/2000/6/707.

A Intuitive explanation of a part of the proof of Theorem 3

In the proof of Theorem 3, case **i**, we assumed $v \notin M(G_1, G_2)$ and reached a contradiction. The reasoning line there can be illustrated by the diagram below, where we step by step construct the vertices v', v'', v''' until we reach a contradiction with another assumption $v \in M(G_1, G_3)$. The symbols $\subset$, $\supset$, $=$ between the vertices show the relation between the marked sets of those vertices. The leftmost graph is a part of G_1, the middle one of G_2, and the rightmost one of G_3.



$$mark(v, G_1) \subset mark(v, G_2) = mark(v, G_3)$$

$$mark(v, G_1) \supset mark(v, G_2) \subset mark(v, G_3)$$

$$mark(v, G_1) \subset mark(v, G_2) \supset mark(v, G_3)$$

$$mark(v, G_1) \supset mark(v, G_2) = mark(v, G_3)$$

Encoding Agda Programs Using Rewriting

Guillaume Genestier
Université Paris-Saclay, ENS Paris-Saclay, Inria, CNRS, LSV, France
MINES ParisTech, PSL University, France

Abstract

We present in this paper an encoding in an extension with rewriting of the Edimburgh Logical Framework (LF) [13] of two common features: universe polymorphism and eta-convertibility. This encoding is at the root of the translator between Agda and Dedukti developped by the author.

2012 ACM Subject Classification Theory of computation → Equational logic and rewriting; Theory of computation → Type theory

Keywords and phrases Logical Frameworks, Rewriting, Universe Polymorphism, Eta Conversion

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.31

Supplementary Material The translator is available at `https://github.com/Deducteam/Agda2Dedukti`, the directory `theory/` contains the encoding presented in Sections 3 and 4. It is able to translate and type-check 162 files of Agda's standard library [9].

Funding Part of this work was carried out during a stay at Chalmers University of Technology, Sweden, funded by the Cost Action CA15123: EUTypes.

Acknowledgements I am grateful to Jesper Cockx and Andreas Abel, who received me in Chalmers and helped me to find my way in the hostile jungle the Agda implementation would have been without their help. Then I would like to thank Frédéric Blanqui, Olivier Hermant, Kostia Chardonnet and the anonymous reviewers for their thorough comments, both on the form and content of this article, which greatly improved its quality.

1 Introduction

With the multiplication of proof assistants, interoperability has became a main obstacle preventing the dissemination of formally verified software among industrial companies.

Indeed, a lot of mathematical results have been formalized, using many different proof assistants. Hence, if one want to use two already proved theorems in her development, there is a high risk that these two proofs are in different systems.

To avoid the community the burden of redevelopping the same proofs in each system, the Logipedia project aims at building an encyclopedia of formal proofs, agnostic in the system they were developped in. To do so, the logics of the proof assistants can be encoded in the same *Logical Framework*: Dedukti, which is based of the $\lambda\Pi$-calculus modulo rewriting. Once all the logics are encoded in the same framework, it becomes easier to compare them, and so to export to a target system proofs originally made in another system.

In this article, we present an encoding of two common features, shared by many proof assistants.

The first one is universe polymorphism. Introduced by Harper and Pollack [14], this allows the user to declare a symbol only once for all universe levels, and then to instantiate it several times with concrete levels.

The second one is equality modulo η. In set theory, a function is identified with its graph, hence two functions outputing the same result when fed with the same data are equal. In type theory, it is not the case. η-conversion is a weak form of this principle of extensionality, which just states that f is equal to the function associating to any x the result of f applied to x.

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 31; pp. 31:1–31:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Developped for twenty years, AGDA is a dependently-typed functional programming language based on an extension Martin-Löf's type theory. Thanks to Curry-Howard correspondence, it is often used as a proof assistant. Furthermore, it features the two ingredients this article focuses on. Hence, the author developed, in collaboration with Jesper Cockx, an automatic translator from a fragment of AGDA to DEDUKTI.

Outline

After a brief presentation of the λΠ-calculus modulo rewriting, Section 2 introduces the Cousineau-Dowek's encoding of *Pure Type Systems*. Section 3 presents a general encoding of universe polymorphism and an instantiation of this encoding in the special case of the predicative two-ladder universe system behind AGDA. The main theorem of this section is the preservation of typability of this encoding. Then, Section 4 explains how to encode η-conversion using rewriting. Preservation of the conversion is the main result of this section. Finally, after a presentation of the implementation in Section 5, Section 6 summarizes our result and provides hints on future extensions.

2 Encoding Pure Type Systems in λΠ-modulo Rewriting

In [3], Barendregt presents the λ-cube, a classification of eight widely used type systems, distinguishing themselves from each other by the possibility they offer (or not) to quantify on a type, a term to construct a type, or a term.

Those constructions of systems in the λ-cube were generalized by Terlouw and Berardi [5], giving birth to what they called "generalized type system", nowadays more often called *Pure Type Systems* (PTS).

Every PTS shares the same typing rules. The only difference between them are the relations $\mathcal{A}$ and $\mathcal{R}$. $\mathcal{A}$, called axioms, states inhabitation between sorts and $\mathcal{R}$, called rules, controls on which sort one can quantify.

▶ **Definition 1** (Syntax and typing of PTS). *Let $\mathcal{X}$ be an infinite set of variables and $\mathcal{S}$ be the set of sorts.*

$$t, u ::= s \mid x \mid (x : t) \to u \mid \lambda x^t.u \mid t\,u \qquad \textit{with } s \in \mathcal{S} \textit{ and } x \in \mathcal{X}$$

The typing rules include 5 introduction rules related to the syntax, and 2 structural rules.

$$\textit{(var)} \quad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\ x \notin \mathrm{dom}(\Gamma)$$

$$\textit{(ax)} \quad \frac{}{\vdash s_1 : s_2}\ (s_1, s_2) \in \mathcal{A} \qquad \textit{(prod)} \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (x : A) \to B : s_3}\ (s_1, s_2, s_3) \in \mathcal{R}$$

$$\textit{(app)} \quad \frac{\Gamma \vdash t : (x : A) \to B \quad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B\,[u/_x]} \qquad \textit{(abs)} \quad \frac{\Gamma \vdash (x : A) \to B : s \quad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.t : (x : A) \to B}$$

$$\textit{(conv)} \quad \frac{\Gamma \vdash t : A \quad \Gamma \vdash B : s}{\Gamma \vdash t : B}\ A \leftrightsquigarrow^*_\beta B \qquad \textit{(weak)} \quad \frac{\Gamma \vdash A : s \quad \Gamma \vdash t : B}{\Gamma, x : A \vdash t : B}\ x \notin \mathrm{dom}(\Gamma)$$

▶ **Definition 2** (Functional Pure Type System). *A PTS is called* functional *if axioms and rules are functional relations, respectively from $\mathcal{S}$ and $\mathcal{S} \times \mathcal{S}$ to $\mathcal{S}$.*

One can be even more restrictive on the class of PTS's considered, by defining a special case of *functional PTS*, the *full PTS*.

▶ **Definition 3** (Full Pure Type System). *A PTS is called* full *if axioms and rules are total functions, respectively from $\mathcal{S}$ and $\mathcal{S} \times \mathcal{S}$ to $\mathcal{S}$.*

▶ **Example 4** ($\mathcal{P}^\infty$ and $\mathcal{C}^\infty$). The predicative and impredicative infinite hierarchies, are two full PTS: $\mathcal{P}^\infty$ is $\mathcal{S} = \{ *_i | \, i \in \mathbb{N}\}\,; \mathcal{A} = \{(*_i, *_{i+1})\}\,; \mathcal{R} = \{\,(*_i, *_j, *_k)|\, k = \max(i,j)\}$ whereas $\mathcal{C}^\infty$ is $\mathcal{S} = \{ *_i | \, i \in \mathbb{N}\}\,; \mathcal{A} = \{(*_i, *_{i+1})\}\,; \mathcal{R} = \{\,(*_i, *_j, *_k)|\, j \geqslant 1 \text{ and } k = \max(i,j)\} \cup \{(*_i, *_0, *_0)\}$.

▶ **Definition 5** (Embedding of PTS). *Given $P_1 = (\mathcal{S}_1; \mathcal{A}_1; \mathcal{R}_1)$ and $P_2 = (\mathcal{S}_2; \mathcal{A}_2; \mathcal{R}_2)$ two PTS, $f : \mathcal{S}_1 \to \mathcal{S}_2$ is an embedding of P_1 in P_2 if for all $(s, s') \in \mathcal{A}_1$, we have $(f(s), f(s')) \in \mathcal{A}_2$ and for all $(s, s', s'') \in \mathcal{R}_1$, we have $(f(s), f(s'), f(s'')) \in \mathcal{R}_2$.*

f is extended to terms of P_1, by: $f(x) = x$, *if* $x \in \mathcal{X}$; $f(\lambda x^A.t) = \lambda x^{f(A)}.f(t)$; $f(t\,u) = f(t)\,f(u)$; $f((x : A) \to B) = (x : f(A)) \to f(B)$.

▶ **Proposition 6** (Soundness of the Embedding). *If f is an embedding from a PTS P_1 to P_2, if $\Gamma \vdash_{P_1} t : A$, then $f(\Gamma) \vdash_{P_2} f(t) : f(A)$.*

Proof. By induction on the proof tree. Since f preserves $\mathcal{A}$ and $\mathcal{R}$, the (ax) and $(prod)$ cases are satisfied. All the other cases are direct, since f does not act on the shape of terms. ◀

The Edimburgh Logical Framework [13] (LF), denoted λP in Barendregt's λ-cube is the minimal PTS including dependent types. It has two sorts $\mathcal{S} = \{\star, \Box\}$, with the axioms $\mathcal{A} = \{(\star, \Box)\}$ and the rules $\mathcal{R} = \{(\star, \star, \star), (\star, \Box, \Box)\}$. It is well-known to be "a framework for defining logics", since it allows to encode most of the proof systems. One can note, LF is not a Full PTS, since $\Box$ is the left-hand side of no axioms.

The logic behind the *Logical Framework* Dedukti is the λΠ-calculus modulo rewriting [2, 6], an extension of the Edimburg Logical Framework with user-defined rewrite rules used not only to define functions, but also types, allowing for shallow embedding of various type systems. Indeed, even if one can encode many logics in LF, those encodings are deep, meaning that applications, λ-abstractions and variables of the encoded system are not translated directly by their equivalent in LF, but by using explicit symbols `App`, `Lam` and `Var`. Using rewriting, the introduction of those extra symbols can be avoided, allowing for more reasonable size translations.

▶ **Definition 7** (Signature in λΠ-modulo rewriting). *A signature in λΠ-modulo rewriting is $(\Sigma, \Theta, \mathbb{R})$ where Σ is a set of symbols, disjoint of $\mathcal{X}$, Θ is a function from Σ to terms and $\mathbb{R}$ is a set of rewriting rules, i.e. a set of pair of terms of the form $f\,\vec{l} \hookrightarrow r$, with $f \in \Sigma$ and all l_i's are Miller's pattern [16].*

We say that t rewrites to u, denoted $t \rightsquigarrow u$ if there is a rule $f\,\vec{l} \hookrightarrow r$, a substitution σ and a "term with a hole" $C[]$, such that $t = C[(f\,\vec{l})\sigma]$ and $u = C[r\sigma]$. $\rightsquigarrow$ is the smallest relation containing $\hookrightarrow$ and stable by substitution and context. We denote by $\rightsquigarrow^*$ the reflexive transitive closure of $\rightsquigarrow$ and by $\leftrightsquigarrow^*$ the convertibility relation, which is the reflexive symmetric and transitive closure of $\rightsquigarrow$.

▶ **Definition 8** (Typing rules of λΠ-modulo rewriting). *They are the one of LF (those of Def. 1, instantiated with $\mathcal{S} = \{\star, \Box\}$, $\mathcal{A} = \{(\star, \Box)\}$ and $\mathcal{R} = \{(\star, \star, \star), (\star, \Box, \Box)\}$.), but with a rule to introduce symbols of Σ and enrichment of the conversion, to include both β-reduction and the user-defined rewriting rules.*

$$(sig)\ \frac{\Gamma \vdash \Theta(f) : s \quad f \in \Sigma}{\Gamma \vdash f : \Theta(f)} \qquad (conv)\ \frac{\Gamma \vdash t : A \quad \Gamma \vdash B : s}{\Gamma \vdash t : B}\ A \leftrightsquigarrow^*_{\beta \cup \mathbb{R}} B$$

In 2007, Cousineau and Dowek [8] proposed an encoding of any functional PTS in Dedukti. Their encoding contained two symbols for each sort, and one symbol for each axiom or rule. However, having an infinite number of symbols and rules is not well-suited for implementations. Hence, to encode *Pure Type Systems* with an infinite number of sorts, one prefers to have a type `Sort` for sorts and only one symbol for products [1]. For *full Pure Type Systems*, this extension is quite straightforward. The general encoding of full PTS is:

First the PTS specifification: a type of sorts and two functions for $\mathcal{A}$ and $\mathcal{R}$.

```
constant Sort : TYPE.
symbol axiom : Sort ⇒ Sort.          symbol rule : Sort ⇒ Sort ⇒ Sort.
```

For each sort `s`, a type `Univ s` containing the codes of its elements. Indeed, since the $\lambda\Pi$-calculus, does not allow to quantify over types, one needs to declare the type of the logic we are encoding, not directly as a type, but as a code, which can be decoded to a type using rewriting rules.

```
constant Univ : (s : Sort) ⇒ TYPE.
```

Then a symbol to decode the elements of `Univ s` as type of $\lambda\Pi$-modulo rewriting.

```
symbol Term : (s : Sort) ⇒ Univ s ⇒ TYPE.
```

The encoding of sorts and the rewrite rule to decode it. (Simulates the rule (ax) of a PTS).

```
constant code : (s : Sort) ⇒ Univ (axiom s).
Term _ (code s) ⟶ Univ s.
```

The encoding of products and its decoding rewrite rule. (Simulates the rule (*prod*) of a PTS).

```
constant prod : (s1 : Sort) ⇒ (s2 : Sort) ⇒
       (A : Univ s1) ⇒ (Term s1 A ⇒ Univ s2) ⇒ Univ (rule s1 s2).
Term _ (prod a b A B) ⟶ (x : Term a A) ⇒ Term b (B x).
```

Then the peculiarity of each PTS is reflected in the encoding of the elements of $\mathcal{S}$ as terms of `Sort`, and in the implementation of `axiom` and `rule` to encode $\mathcal{A}$ and $\mathcal{R}$ respectively.

3 Universe Polymorphism and its Encoding

It is quite common to enrich PTS with *Universe Polymorphism* [14], which consists in allowing the user to quantify over *universe levels*, allowing to declare simultaneously a symbol for several sorts. For instance, if the sorts are $\{\,\mathrm{Set}_i \,|\, i \in \mathbb{N}\}$, then one want to declare `List` in $\forall \ell, (A : \mathrm{Set}_\ell) \to \mathrm{Set}_\ell$. Indeed, just like polymorphism was used to avoid declaring a type of lists for each type of elements, one want to avoid one declaration of a new type of lists for each universe level.

We present here a definition of *universe polymorphism* inspired by the one given by Sozeau and Tabareau [19] for the proof assistant Coq. In this setting, the context contains three lists: a list Σ called signature, a list Θ of level variables, and a list Γ called local context. Both Σ and Γ contain pairs of a variable name and a type, but the variables in Γ can contain free level variables (those occuring in Θ), whereas all the level variables are bound by a prenex quantifier $\forall$ in the signature Σ. Unlike [19], we do not need to store constraints between universe levels, since those constraints are related to cumulativity, a feature we are not trying to encode here.

▶ **Definition 9** (Uniform Universe Polymorphic Full PTS). *We consider a set $\mathbb{L}$ of levels and a finite set $\mathcal{H}$ of sort constructors. Then the sorts are $\{s_\ell\}_{s\in\mathcal{H},\ell\in\mathbb{L}}$.*

In addition to functionality and totality of $\mathcal{A}$ and $\mathcal{R}$, we assume a uniformity in the hierarchy. Meaning that for all $s \in \mathcal{H}$, there is a $s' \in \mathcal{H}$, such that for all $\ell \in \mathbb{L}$, there is a $\ell' \in \mathbb{L}$, such that $(s_\ell, s'_{\ell'}) \in \mathcal{A}$ and for all $s^{(1)}, s^{(2)} \in \mathcal{H}$, there is a $s^{(3)} \in \mathcal{H}$, such that for all $\ell_1, \ell_2, \in \mathbb{L}$, there is $\ell_3 \in \mathbb{L}$ such that $(s^{(1)}_{\ell_1}, s^{(2)}_{\ell_2}, s^{(3)}_{\ell_3}) \in \mathcal{R}$.

We denote by $\bar{\mathcal{A}}$ the function $\left\{ (s,s') \in \mathcal{H}^2 \middle| \exists \ell, \ell', (s_\ell, s'_{\ell'}) \in \mathcal{A} \right\}$ and for all s by $\mathcal{A}_s$ the function $\left\{ (\ell,\ell') \in \mathbb{L}^2 \middle| \exists s', (s_\ell, s'_{\ell'}) \in \mathcal{A} \right\}$.

Analogously $\bar{\mathcal{R}}$ is the function $\left\{ (s^{(1)}, s^{(2)}, s') \in \mathcal{H}^3 \middle| \exists \ell_1, \ell_2, \ell', (s^{(1)}_{\ell_1}, s^{(2)}_{\ell_2}, s'_{\ell'}) \in \mathcal{R} \right\}$ and for all $(s^{(1)}, s^{(2)})$, $\mathcal{R}_{s^{(1)},s^{(2)}}$ is the function $\left\{ (\ell_1, \ell_2, \ell') \in \mathbb{L}^3 \middle| \exists s', (s^{(1)}_{\ell_1}, s^{(2)}_{\ell_2}, s'_{\ell'}) \in \mathcal{R} \right\}$.

The typing rules are:

$$\textit{(lvl)} \quad \frac{}{\Theta \vdash \ell \ \text{ isLvl}}\ \ell \in \mathbb{L} \qquad\qquad \textit{(ax)} \quad \frac{\Theta \vdash \gamma \ \text{ isLvl}}{[]; \Theta; [] \vdash s_\gamma : s'_{\mathcal{A}_s(\gamma)}}\ (s,s') \in \bar{\mathcal{A}}$$

$$\textit{(}\mathbb{L}\textit{var)} \quad \frac{}{\Theta \vdash i \ \text{ isLvl}}\ i \in \Theta \qquad\qquad \textit{(abs)} \quad \frac{\Sigma; \Theta, \Gamma \vdash (x : A) \to B : s_\gamma \quad \Sigma; \Theta; \Gamma, x : A \vdash t : B}{\Sigma; \Theta; \Gamma \vdash \lambda(x : A).t : (x : A) \to B}$$

$$\textit{(}\mathbb{L}\mathcal{A}\textit{)} \quad \frac{\Theta \vdash \ell \ \text{ isLvl}}{\Theta \vdash \mathcal{A}_s(\ell) \ \text{ isLvl}} \qquad\qquad \textit{(app)} \quad \frac{\Sigma; \Theta; \Gamma \vdash t : (x : A) \to B \quad \Sigma; \Theta; \Gamma \vdash u : A}{\Theta; \Gamma \vdash t\,u : B\,[{}^{u}/_{x}]}$$

$$\textit{(}\mathbb{L}\mathcal{R}\textit{)} \quad \frac{\Theta \vdash \ell_1 \ \text{ isLvl} \quad \Theta \vdash \ell_2 \ \text{ isLvl}}{\Theta \vdash \mathcal{R}_{ss'}(\ell_1, \ell_2) \ \text{ isLvl}} \qquad\qquad \textit{(conv)} \quad \frac{\Sigma; \Theta; \Gamma \vdash t : A \quad \Sigma; \Theta; \Gamma \vdash B : s_\gamma}{\Sigma; \Theta; \Gamma \vdash t : B}\ A \leftrightsquigarrow^*_\beta B$$

$$\textit{(var)} \quad \frac{\Sigma; \Theta; \Gamma \vdash A : s_\gamma}{\Sigma; \Theta; \Gamma, x : A \vdash x : A}\ x \notin \Sigma, \Gamma \qquad\qquad \textit{(sig)} \quad \frac{\Sigma; \Theta; [] \vdash A : s_\gamma}{\Sigma, x : \forall\Theta.A; \Theta'; [] \vdash x : \forall\Theta.A}\ x \notin \Sigma, \Gamma$$

$$\textit{(inst)} \quad \frac{\Sigma; \Theta; \Gamma \vdash t : \forall[i_1, \ldots, i_n], A \quad \Theta \vdash \gamma_1 \ \text{ isLvl} \quad \ldots \quad \Theta \vdash \gamma_n \ \text{ isLvl}}{\Sigma; \Theta; \Gamma \vdash t[\gamma_1, \ldots, \gamma_n] : A \left[{}^{\gamma_k}/_{i_k}\right]_k}$$

$$\textit{(prod)} \quad \frac{\Sigma; \Theta; \Gamma \vdash A : s_\gamma \quad \Sigma; \Theta; \Gamma, x : A \vdash B : s'_{\gamma'}}{\Sigma; \Theta; \Gamma \vdash (x : A) \to B : s''_{\mathcal{R}_{s,s'}(\gamma,\gamma')}}(s, s', s'') \in \bar{\mathcal{R}}$$

$$\textit{(ctx-weak)} \quad \frac{\Sigma; \Theta; \Gamma \vdash A : s_\gamma \quad \Sigma; \Theta; \Gamma \vdash t : B}{\Sigma; \Theta; \Gamma, x : A \vdash t : B}\ x \notin \Sigma, \Gamma$$

$$\textit{(sig-weak)} \quad \frac{\Sigma; \Theta; [] \vdash A : s_\gamma \quad \Sigma; \Theta'; [] \vdash t : B}{\Sigma, x : \forall\Theta.A; \Theta'; \Gamma \vdash t : B}\ x \notin \Sigma, \Gamma$$

In all those typing rules, $s, s' \in \mathcal{H}$ and $i, x \in \mathcal{X}$. Furthermore, we allowed ourselves to simply write $x \notin \Sigma, \Gamma$, rather than "for all A, $x : A$ is not in Σ, Γ".

One typical case of use, is to have only one hierarchy: $\mathcal{H} = \{\text{Set}\}$ and to use natural numbers for levels: $\mathbb{L} = \mathbb{N}$. But we do not want to restrict ourselves to have only one hierarchy, since some proof assistants feature several. For instance, in Agda and Coq, there are 2, called Set and Prop, and Type and SProp respectively.

The two rules modifying the signature Σ, allows to completely change the set Θ of names of local variables. Changing this set during the proof is not necessary, however, without this renewal of Θ, all the symbols in the signature would have been quantified over the same set Θ, no matter which variables occur really in it.

The universe polymorphism we are interested in is purely prenex. Furthermore, universally quantified types are not typed themselves and are only inhabited by variables. This form of universe polymorphism only provides ease of use, but it does not allow to prove more, meaning that it does not compromise the consistency of the logic.

To prove this, one can construct a new PTS $(\mathcal{S}^\Theta, \mathcal{A}^\Theta, \mathcal{R}^\Theta)$ simply by adding a brand new sort for every expression containing a level variable (such expressions are in $\mathbb{L}^+_\Theta$). Then embedding this newly-constructed PTS in the original one is defined just by interpreting level variables. Then using this interpretation of the variables, one can mimic the proofs done using universe polymorphism in the original PTS.

▶ **Proposition 10** (Conservativity of the universe polymorphism). *Let $P = (\mathbb{L}, \mathcal{H}, \mathcal{A}, \mathcal{R})$ be a uniform universe polymorphic full PTS and Θ be a subset of $\mathcal{X}$.*

Let $\mathbb{L}^+_\Theta$ be the smallest subset such that:

$$\mathbb{L}^+_\Theta = \Theta \cup \left\{ \mathcal{A}_s(l) \middle| s \in \mathcal{H}, l \in \mathbb{L}^+_\Theta \right\} \cup \left\{ \mathcal{R}_{ss'}(l_1, l_2) \middle| s, s' \in \mathcal{H}, (l_1, l_2) \in (\mathbb{L} \cup \mathbb{L}^+)^2 \setminus \mathbb{L}^2 \right\}.$$

Let $\mathcal{X}^+ = \mathcal{X} \cup \left\{ y[l_1, \dots, l_n] \middle| y \in \mathcal{X}, n \in \mathbb{N}, (l_1, \dots, l_n) \in (\mathbb{L} \cup \mathbb{L}^+_\mathcal{X})^n \right\}$ and P^Θ be the PTS:

$$\mathcal{S}^\Theta = \left\{ s_l \middle| s \in \mathcal{H}, l \in \mathbb{L} \cup \mathbb{L}^+_\Theta \right\}; \qquad \mathcal{A}^\Theta = \mathcal{A} \cup \left\{ \left(s_l, s'_{\mathcal{A}_s(l))} \right) \middle| (s, s') \in \bar{\mathcal{A}}, l \in \mathbb{L}^+_\Theta \right\}$$
$$\mathcal{R}^\Theta = \mathcal{R} \cup \left\{ \left(s_{l_1}, s'_{l_2}, s''_{\mathcal{R}_{ss'}(l_1, l_2))} \right) \middle| (s, s', s'') \in \bar{\mathcal{R}}, (l_1, l_2) \in (\mathbb{L} \cup \mathbb{L}^+)^2 \setminus \mathbb{L}^2 \right\}$$

a. *There is an embedding from P^Θ to the underlying PTS of P.*
b. *If $\Sigma; \Theta; \Gamma \vdash t : A$ in P and A is not a universal quantification, then there is a $\bar{\Sigma} \subset \left\{ x[l_1, \dots, l_n] : A' \middle| x : \forall [y_1, \dots, y_n].A \in \Sigma, A' = A \left[{}^{l_i}/_{y_i} \right]_{i=1\dots n} \text{ and all } l_i \in \mathbb{L} \cup \mathbb{L}^+_\Theta \right\}$ such that $\bar{\Sigma}, \Gamma \vdash_{P^\Theta} t : A$ using the enriched set of variables $\mathcal{X}^+$.*

Proof sketch. **a.** The embedding consists in just chosing a level for each variable in Θ.
b. Since A is not a universal quantification, in the proof of $\Sigma; \Theta; \Gamma \vdash t : A$, all the (*sig*) are followed directly by an arbitrary number of weakenings and a (*inst*). The weakenings can be anticipated and to create a proof in P^Θ, the (*sig*) and (*inst*) are compressed in a single introduction of a variable of $\bar{\Sigma}$. ◀

In a PTS, if $\Gamma \vdash t : A$, then there is a sort s such that $A = s$ or $\Gamma \vdash A : s$. In a full PTS, $\mathcal{A}$ is a total function, hence, all sorts inhabit a sort, allowing us to refer to s as the sort of a A. However, in the presentation of universe polymorphism of Def. 9, this property is lost because universally quantified types have no type. To overcome this issue, we assign artificially a type to those quantified types, using a brand new sort Sort_ω, which is not typable, is the type of no sort and over which one cannot quantify. Its only purpose is to make "the sort of A" well-defined whenever A is inhabited. It must be noted that Sort is not in $\mathcal{H}$ and ω is not a level.

To encode *Universe Polymorphic Full PTS*, one introduce a symbol `sortOmega` and a quantification symbol `∀𝕃` which takes as first argument the sort in which the term will live once instanciated. The definition of the decoding function `Term` is enriched with a new rule, specifying its behaviour when applied to a `∀𝕃`.

▶ **Definition 11** (Encoding).

```
constant sortOmega : Sort.
constant ∀𝕃 : (f:(𝕃⇒Sort)) ⇒ ((l:𝕃) ⇒ Univ (f l)) ⇒ Univ sortOmega.
Term _ (∀𝕃 f t) ⟶ (l : 𝕃) ⇒ Term (f l) (t l).
```

For instance, the encoding of $\forall \ell, \mathrm{Set}_\ell$ is `∀𝕃 (λ l, axiom (set l)) (λ l, code (set l))`, if `set` is a sort constructor in the encoding. And its decoding (when applying `Term sortOmega`) is, as expected, `(l:𝕃) ⇒ Univ (set l)`.

▶ **Example 12.** Consider the system $\mathcal{H} = \{s, \sigma\}$, $\mathcal{A} = \left\{ (A_i, s_{ax_A(i)}) \middle| A \in \mathcal{H} \right\}$ and $\mathcal{R} = \left\{ (A_i, B_j, B_{ru(i,j)}) \middle| A, B \in \mathcal{H} \right\}$, with ax_s, ax_σ and ru three functions remaining abstract here. ru could be indexed by two sorts, for ease of readibility, we have chosen not present such a general case.

```
(; one symbol for each sort constructor ;)
constant s : 𝕃 ⇒ Sort.                    constant σ : 𝕃 ⇒ Sort.
(; Function axiom ;)
symbol axiom : Sort ⇒ Sort.
symbol ax_s : 𝕃 ⇒ 𝕃.                      symbol ax_σ : 𝕃 ⇒ 𝕃.
axiom (s i) ⟶ s (ax_s i).                 axiom (σ i) ⟶ s (ax_σ i).
(; Function rule ;)
symbol rule : Sort ⇒ Sort ⇒ Sort.         symbol ru : 𝕃 ⇒ 𝕃 ⇒ 𝕃.
rule (s i) (s j) ⟶ s (ru i j).            rule (s i) (σ j) ⟶ σ (ru i j).
rule (σ i) (s j) ⟶ s (ru i j).            rule (σ i) (σ j) ⟶ σ (ru i j).
```

▶ **Definition 13** (Translation). *We translate well-typed terms in a Universe Polymorphic Full Pure Type System by:* $\|x\| = \mathtt{x}$; $\|s_\ell\| = \mathtt{code}\ |s_\ell|_S$; $\|t\,u\| = \|t\|\ \|u\|$;
$\left\|\lambda x^A.t\right\| = \lambda(\mathtt{x} : \mathtt{Term}\ \ |s_A|_S\ \|A\|).\|t\|$;
$\|(x : A) \to B\| = \mathtt{prod}\,|s_A|_S\ |s_B|_S\ \|A\|\ (\lambda\mathtt{x} : \mathtt{Term}\ |s_1|_S\ \|A\|.\|B\|)$;
$\|\forall[\ell_1, \dots, \ell_n], A\| = \forall_{\mathbb{L}}\ (\lambda\ell_1 : \mathbb{L}.\ \ \mathtt{sortOmega})\ (\lambda\ell_1 \dots\ \forall_{\mathbb{L}}\ (\lambda\ell_n : \mathbb{L}.|s_A|_S)\ (\lambda\ell_n : \mathbb{L}.\|A\|)\dots)$;
$\|A[\gamma_1, \dots, \gamma_n]\| = \|A\|\ |\gamma_1|_L \dots |\gamma_n|_L$.
The translation of sorts is $|\mathrm{Sort}_\omega|_S = \mathtt{sortOmega}$, $|s_\gamma|_S = \mathtt{s}\ |\gamma|_L$.
And the translation of levels is $|i|_L = i$ *if* $i \in \mathcal{X}$;
$|\mathcal{A}_s(\ell)|_L = \mathtt{ax\_s}\ |\ell|_L$ *and* $|\mathcal{R}_{ss'}(\ell_1, \ell_2)|_L = \mathtt{ru\_ss'}\ |\ell_1|_L\ |\ell_2|_L$.
Wherever they are used, s_A *and* s_B *are respectively the sorts of* A *and* B*.*

It can be noted that the translation $|\ell|_L$ for $\ell \in \mathbb{L}$ is not given, since in general the number of level is infinite, hence, we do not want to introduce one new symbol per level. Furthermore, with universe polymorphism, universe levels are open terms, hence, convertibility between universe levels is now an issue. Fortunately, it is the last one, since once this issue is overcome, the encoding has one of the expected properties: we type check at least as much terms as in the original system.

To state this, we start with two useful lemmas:

▶ **Lemma 14** (Substitution and conversion).

a. *If* x *is a free variable in* t *such that* t *and* $t\,[{}^{u}/_{x}]$ *are well-typed,* $\|t\,[{}^{u}/_{x}]\| = \|t\|\ \left[{}^{\|u\|}/_{x}\right]$;

b. *If* ℓ *is a level variable in* t *such that* t *and* $t\,[{}^{u}/_{\ell}]$ *are well-typed,* $\|t\,[{}^{u}/_{x}]\| = \|t\|\ \left[{}^{|u|_L}/_{x}\right]$;

c. *If* $t \rightsquigarrow_\beta u$*, then* $\|t\| \rightsquigarrow_\beta \|u\|$.

Proof. a and b are proved by induction on the the term t. c is because a β-redex is translated as a β-redex. ◀

The proof of this property is only sketched, since Section 4 will contain detailled proofs on the conversion specifically.

▶ **Lemma 15** (Shape-preservation of type).

a. *If* s *is a sort,* $\mathtt{Term}\,|\mathcal{A}(s)|_S\ \|s\| \rightsquigarrow^* \mathtt{Univ}\,|s|_S$,

b. *If* $(x : A) \to B$ *is of sort* s*,* $\mathtt{Term}\,|s|_S\ \|(x : A) \to B\| \rightsquigarrow^* (x : \mathtt{Term}\,|s_A|_S\ \|A\|) \Rightarrow \mathtt{Term}\,|s_B|_S\ \|B\|$;

c. *If* $\ell_1 < \dots < \ell_n$*,* $\mathtt{Term}\ \mathtt{sortOmega}\ \|\forall\,\{\ell_i\}_i\,, A\| \rightsquigarrow^* (\ell_1 : \mathbb{L}) \Rightarrow \dots \Rightarrow (\ell_n : \mathbb{L}) \Rightarrow \|A\|$.

Proof. The three rules on `Term` are crafted to ensure those properties. ◀

To state properly the Correctness Theorem, one first has to define the translation of contexts:

▶ **Definition 16** (Context Translation). *If* $\Sigma = x_1 : T_1, \dots, x_l : T_l$, $\Theta = i_1, \dots, i_m$ *and* $\Gamma = y_1 : A_1, \dots, y_n : A_n$, *then the translation is* $\|\Sigma;\Theta;\Gamma\| = x_1 : \texttt{Term sortOmega } \|T_1\|, \dots, x_l : \texttt{Term sortOmega } \|T_l\|, i_1 : \mathbb{L}, \dots, i_m : \mathbb{L}, y_1 : \texttt{Term } |s_{A_1}|_S \ \|A_1\|, \dots, y_n : \texttt{Term } |s_{A_n}|_S \ \|A_n\|$.

▶ **Theorem 17** (Correctness). *Given a correct criterion for equality of levels (i.e. if two levels* ℓ_1 *and* ℓ_2 *are equals, their translations* $|\ell_i|_L$ *are convertible), for a Universe Polymorphic Full Pure Type System* P*, if* $\Sigma;\Theta;\Gamma \vdash t : A$*, then* $\|\Sigma;\Theta;\Gamma\| \vdash_{\lambda\Pi/P} \|t\| : \texttt{Term } |s|_S \ \|A\|$*, where* s *is the sort of* A*.*

Proof. By induction on the derivation. We assume that if $\Theta \vdash \gamma \,\mathrm{isLvl}$, then $\|[];\Theta;[]\| \vdash_{\lambda\Pi/P} |\gamma|_L : \mathbb{L}$, a property which can be proved by induction on the derivation, with the assumption that for all $\ell \in \mathbb{L}$, $\vdash_{\lambda\Pi/P} |\ell|_L : \mathbb{L}$. We then consider the 10 remaining cases:

(var) By induction hypothesis, $\|\Sigma;\Theta;\Gamma\| \vdash_{\lambda\Pi/P} \|A\| \,\texttt{:Univ}\, |s_\gamma|_S$. Hence $\|\Sigma;\Theta;\Gamma\| \vdash_{\lambda\Pi/P} \texttt{Term } |s_\gamma|_S \ \|A\|$: $\texttt{TYPE}$, so one can introduce a variable of this type.

(ax) The translation of s_γ is $\texttt{code (s } |\gamma|_L)$ which lives in $\texttt{Univ (s' (ax\_s } |\gamma|_L))$, which is the reduct of the translation as type of $s'_{\mathcal{A}_s(\gamma)}$.

(abs) By induction hypothesis, $\|\Sigma;\Theta;\Gamma\|,\texttt{x : Term } |s|_S \, \|A\| \vdash_{\lambda\Pi/P} \|t\| : \texttt{Term } |s'|_S \, \|B\|$, hence, one has that $\lambda(x : \texttt{Term } |s|_S \ \|A\|).t$ inhabits $(x : \texttt{Term } |s|_S \ \|A\|) \to \texttt{Term } |s'|_S \ \|B\|$, which is the reduct of the translation as type of $(x : A) \to B$. The other induction hypothesis $\|\Sigma;\Theta;\Gamma\| \vdash_{\lambda\Pi/P} \|(x : A) \to B\| : \texttt{Univ } |s_\gamma|_S$ ensures us that $\texttt{Term } |s|_S \ \|A\|$ lives in $\texttt{TYPE}$.

(app) By the induction hypothesis and the Lem. 15, one can apply the translation of t to the translation of u. The result lives in the translation of $B\left[{}^{u}/_{x}\right]$ thanks to Lem. 14.

(conv) This is a direct consequence of Lem. 14 and the induction hypotheses.

(sig) By induction hypothesis, $\|\Sigma;\Theta;[]\| \vdash_{\lambda\Pi/P} \|A\| : \texttt{Univ } |s_\gamma|_S$. Hence, one can use the (prod) rule of $\lambda\Pi$-modulo rewriting to move all the $i : \mathbb{L}$ from the context to the term. By Lem. 15, the product obtained is convertible with $\|\forall\Theta.A\|$, hence one can introduce a variable of this type. One must then use the weakening, to Re-invent the variables of type $\mathbb{L}$ corresponding to the Θ'.

(inst) Lem. 15 tells us that, after conversion, the induction hypothesis is $\|\Sigma;\Theta;\Gamma\| \vdash \|A\| : (\ell_1 : \mathbb{L}) \to \dots \to (\ell_n : \mathbb{L}) \to \|X\|$, hence, we can apply the γ_i's without type issues.

(prod) By induction hypothesis, we have $\|\Sigma;\Theta;\Gamma\| \vdash_{\lambda\Pi/P} \|A\| : \texttt{Univ } \|s_\gamma\|$ and also $\|\Sigma;\Theta;\Gamma, x : A\| \vdash_{\lambda\Pi/P} \|B\| : \texttt{Univ } \|s'_{\gamma'}\|$, so $\|\Sigma;\Theta;\Gamma\|, x : \texttt{Term } |s_\gamma|_S \ \|A\| \vdash_{\lambda\Pi/P} \|B\| : \texttt{Univ } \|s'_{\gamma'}\|$ and we can conclude by introducing the lambda and applying $\texttt{prod}$.

(ctx-weak) As before, we have $\|\Sigma;\Theta;\Gamma\| \vdash_{\lambda\Pi/P} \|A\| : \texttt{Univ } \|s_\gamma\|$, so $\|\Sigma;\Theta;\Gamma\| \vdash_{\lambda\Pi/P} \texttt{Term } |s_\gamma|_S \ \|A\|$: $\texttt{TYPE}$, so one can weaken with a variable of this type.

($\forall$weak) Like for the (sig) rule, one can empty the context of the variables of type $\mathbb{L}$ by applying the rule (prod) of $\lambda\Pi$-modulo rewriting. Then, one can weaken with a variable of this type and variables of type $\mathbb{L}$ to translate the Θ'. ◀

Now, we will more specifically focus on a specific hierarchy of levels, where $\mathbb{L} = \mathbb{N}$ and all the $\mathcal{A}_s$ are the successor function and all $\mathcal{R}_{ss'}$ are the maximum function. This is the predicative hierarchy of $\mathcal{P}^\infty$ (Expl. 4), used in Agda for instance.

The grammar of universe level we are interested in is: $t, u \in \mathcal{L} ::= x \in \mathcal{X} \mid 0 \mid \mathrm{s}\, t \mid \max t\, u$:

```
constant 𝕃 : TYPE.                  symbol 0 : 𝕃.
symbol s : 𝕃 ⇒ 𝕃.                  symbol max : 𝕃 ⇒ 𝕃 ⇒ 𝕃.
```

The question which arises in the translation is to have a convergent rewrite system such that for all t and u in $\mathcal{L}$:

$$t{\downarrow} = \ u{\downarrow} \ \text{ if and only if } \forall \sigma : \mathcal{X} \to \mathbb{N}, [\![t]\!]_\sigma = [\![u]\!]_\sigma$$

where $[\![\_]\!]_\_ : \mathcal{L} \to (\mathcal{X} \to \mathbb{N}) \to \mathbb{N}$ is the obvious interpretation in $\mathbb{N}$:

$$[\![0]\!]_\sigma = 0_\mathbb{N} \quad [\![x]\!]_\sigma = \sigma(x), \text{ if } x \in \mathcal{X} \quad [\![\mathrm{s}\, t]\!]_\sigma = [\![t]\!]_\sigma +_\mathbb{N} 1_\mathbb{N} \quad [\![\max\, t\, u]\!]_\sigma = \max{}_\mathbb{N}([\![t]\!]_\sigma, [\![u]\!]_\sigma)$$

Since max is associative and commutative (AC), we will propose an encoding having a weak version of this property: $t{\downarrow} \equiv_{AC} \ u{\downarrow}$ if and only if $\forall \sigma : \mathcal{X} \to \mathbb{N}, [\![t]\!]_\sigma = [\![u]\!]_\sigma$.

Since $[\![\mathrm{s}\,(\max\, t\, u]\!] = [\![\max\,(\mathrm{s}\, t)\,(\mathrm{s}\, u)]\!]$, one can consider having a Max acting on a set of terms, which do not contain max.

Furthermore, we have for all n the equality $[\![\max\,(\mathrm{s}^n\, x)\, x]\!] = [\![\mathrm{s}^n\, x]\!]$. To avoid declaring this rule infinitely often (once for every n), we add addition to our encoding. However, since this addition encodes iteration of the application of s, it is not an addition between two levels, but one between a ground natural number and a level. Furthermore, $[\![\max\,(\mathrm{s}^n\, x)\,(\mathrm{s}^m\, 0)]\!] = [\![\mathrm{s}^n\, x]\!]$, if $m < n$. Hence, the symbol Max will also collect the value of the smallest possible ground natural that the result can be.

Hence, in our encoding, the normal forms are the Max $i\ \{j_k + x_k\}_k$ where:
(1) $i, j_1, \ldots$ are ground naturals, (2) $x_1, \ldots$ are distinct variables, (3) for all k, $i \geqslant j_k$.
A separate type $\mathbb{N}$, containing only ground natural numbers, is declared, to avoid confusion with levels.

```
constant ℕ : TYPE.        constant 0ℕ : ℕ.       constant sℕ : ℕ ⇒ ℕ.
definition 1ℕ := sℕ 0ℕ.
symbol maxℕ : ℕ ⇒ ℕ ⇒ ℕ.           maxℕ 0ℕ y ⟶ y.
maxℕ x 0ℕ ⟶ x.                     maxℕ (sℕ x) (sℕ y) ⟶ sℕ (maxℕ x y).
infix +ℕ : ℕ ⇒ ℕ ⇒ ℕ.
0ℕ +ℕ y ⟶ y.                      (sℕ x) y ⟶ sℕ (x +ℕ y).
```

Sets can be empty or singleton or union of sets. This union operator is an associative and commutative symbol. Furthermore, since singletons are of the form $\{i + x\}$, the constructor of singletons is denoted $\oplus$.

```
symbol ∅: LSet.    infix ⊕: ℕ⇒𝕃⇒LSet.    infix ac ∪: LSet⇒LSet⇒LSet.
x ∪ ∅ ⟶ x.
```

Since constraint (1) is guaranteed by typing, we still have to implement the two constraints (2) and (3) presented in the description of the normal form:

- The only non-left-linear rule of the encoding eliminates redundancies, ensuring that all variables in the normal forms are distinct, in order to satisfy the invariant (2).

  ```
  (i ⊕ l) ∪ (j ⊕ l) ⟶ (maxℕ i j) ⊕ l.
  ```

- Intuitively, to flatten the entanglement of max and plus, we would like to have a rule stating that $a + \max(b, c) = \max(a + b, a + c)$.
  However, to fulfill constraint (3), we added the invariant that the first argument of Max is larger than all the first arguments of the $\oplus$ occuring directly under it. Hence, we do not declare the expected computation rule of $\oplus$, but enforce this computation to be performed under a Max.
  Furthermore, for typing distinction between $\mathbb{L}$ and LSet, we introduce an auxiliary function mapping (i $\oplus$ __) to all the elements of a set.

```
symbol mapPlus : ℕ ⇒ LSet ⇒ LSet.
mapPlus i ∅ ⟶ ∅.           mapPlus i (j ⊕ l) ⟶ (i +ℕ j) ⊕ l.
mapPlus i (l1 ∪ l2) ⟶ (mapPlus i l1) ∪ (mapPlus i l2).
symbol Max : ℕ ⇒ LSet ⇒ 𝕃          Max 0ℕ (0ℕ ⊕ x) ⟶ x.
Max i  (j ⊕ Max k l)         ⟶ Max (maxℕ i (j +ℕ k)) (mapPlus j l).
Max i ((j ⊕ Max k l) ∪ tl) ⟶
                          Max (maxℕ i (j +ℕ k)) ((mapPlus j l) ∪ tl).
```

And finally we give rewrite rules for the symbols of the syntax:

```
0 ⟶ Max 0ℕ ∅.                   s x ⟶ Max 1ℕ (1ℕ ⊕ x).
max x y ⟶ Max 0ℕ ((0ℕ ⊕ x) ∪ (0ℕ ⊕ y)).
```

This encoding is not confluent, as the following example illustrates:

```
Max i (j ⊕ (Max k (j2 ⊕ (Max k2 l))))
 ⇝o Max (maxℕ i (j +ℕ k)) (mapPlus j (j2 ⊕ (Max k2 l)))
  ⇝ Max (maxℕ i (j +ℕ k)) ((j +ℕ j2) ⊕ (Max k2 l))
  ⇝ Max (maxℕ (maxℕ i (j +ℕ k)) (j +ℕ j2 +ℕ k2)) (mapPlus (j +ℕ j2) l)
 ⇝i Max i (j ⊕ (Max (maxℕ k (j2 +ℕ k2)) (mapPlus j2 l)))
  ⇝ Max (maxℕ i (j +ℕ (maxℕ k (j2 +ℕ k2)))) (mapPlus j (mapPlus j2 l))
```

But this is not an issue, since we are only interested in reducts of elements of the syntax, meaning that all the variables are of type $\mathbb{L}$.

▶ **Proposition 18.** *The absence of variable of type* $\mathbb{N}$ *or* `LvlSet` *ensures the uniqueness of normal form (modulo AC) property.*

Proof. Since there are no variables of type $\mathbb{N}$ and `LSet`, the function $\mathtt{max}_{\mathbb{N}}$, $+_{\mathbb{N}}$ and `mapPlus` are fully defined and cannot occur in the normal forms.

Hence, normal forms contain only $0_{\mathbb{N}}$, $\mathtt{s}_{\mathbb{N}}$, `Max`, $\emptyset$, $\oplus$ and $\cup$. Among it, the only constructor of a $\mathbb{L}$ is `Max`, hence every level is either a variable or headed by `Max`.

If it contains a `Max`, there is one at the head. Hence the terms are of the form `Max n s` with `n` a closed natural and `s` a `LSet`. If there are more than one `Max`, it means that the `LSet` contains a level which is not a variable. This one, is headed by `Max`, so one of the rewrite rule regarding the interaction between `Max` and $\oplus$ can be applied.

Hence all normal forms are either a variable or of the form `Max n s`, with `n` closed natural and `s` a `LSet` where all levels are variable. The non-linear rule ensures us that the variables are all distinct.

One can check that the invariant that every natural which is the first argument of a $\oplus$ is smaller or equal to the first argument of the `Max` directly above the $\oplus$ is preserved by every rule and verified by the reducts of the syntax.

So, we can conclude that the normal forms have the shape announced.

To check that a term cannot have two distinct normal forms, the definition of the interpretation is extended to the symbols we introduced and one can verify that all the rules preserve the interpretation and that all the terms of the shape we decribed have a different interpretation. ◀

4 Eta-conversion

Many proof assistants implement, among other conversion rules, the η rule, which state that if f is a function, $f \equiv_\eta \lambda x. f\, x$.

At first sight, this conversion might look quite harmless, and one can hope to just add the corresponding rewrite rule. However, this conversion is an important issue for translation of systems in Dedukti. Indeed, the contraction rule cannot be stated, since $\lambda x. f\,x$ is not a Miller pattern: It requires to match on the fact that $f\,x$ is an application, which would be "meta-matching" and is not in the definition of $\lambda\Pi$-modulo rewriting. Furthermore, we could replace it by $\lambda x. f[x]$, but f is not a valid right-hand side anymore, since it is of arity one. On the other hand, to preserve typing, the expansion rule has to match on the type of a variable, and is not syntax-directed anymore.

Another natural solution could be to define $\lambda\Pi$-modulo rewriting as a logical framework with η hard-coded in the conversion (just like β is). But this is a path *logical frameworks* want to avoid. Indeed, if η is hard-coded, it is impossible to have a shallow encoding of the λ-calculus without η-conversion.

One could expect that η-expanding every term during the translation phase, could allow us to completely ignore η-conversion in the $\lambda\Pi$-calculus modulo rewriting. Indeed, with dependent types it might happen than an η-long term has a non-η-long type. A situation that often breaks the *type preservation of the translation.*

▶ **Example 19.** To illustrate this, we start by defining a type, whose number of arrows depends on a natural number, with a constructor for this type.

```
symbol D : (x : ℕ) ⇒ TYPE.          constant d : (x : ℕ) ⇒ D x.
D 0 ⟶ ℕ.                            D (s x) ⟶ ℕ ⇒ D x.
```

We then define a new type depending on the first one and its constructor.

```
symbol E : (x : ℕ) ⇒ D x ⇒ TYPE.     symbol e : (x : ℕ) ⇒ E x (d x).
```

Now, the term `e 1` is η-long and has type `E 1 (d 1)`, but not `E 1 (λ x, d 1 x)` which is the η-long form of the type.

To overcome this issue, we propose to postpone η-expansion, until the type is fully instantiated. For this, we introduce in the translation a symbol ηE, which purpose is to tag with their types the subterms which may become η-expandable. Then some rewrite rules pattern match on this type annotation to decide when and how the expansion can be performed.

▶ **Definition 20** (Eta-expansion rewrite rules)**.** *ηE annotates terms with their types, to do so, it takes as arguments a sort, a code of type in this sort and the term to annotate. The rules state that η-expansion is the identity for inhabitant of sorts (ηS), and genesrates λ's for inhabitants of products (ηP). Furthermore, a rule state that η-expansion is an idempotent operation (ηI).*

```
symbol ηE : (s : Sort) ⇒ (A : Univ s) ⇒ Term s A ⇒ Term s A.
"ηS" ηE _ (code _)        t ⟶ t.
"ηP" ηE _ (prod a b A B) t ⟶
        λ (x : Term a A), ηE b (B (ηE a A x)) (t (ηE a A x)).
"ηI" ηE _ _ (ηE a A t)      ⟶ ηE a A t.
```

To prove that adding those annotations in the encoding enriches enough the conversion to simulate η-equality, we will also add those annotations in the system we are translating, just like what is done in [12, 11].

For sake of readibility, we will study in this section, terms typed in a full PTS embeddable in $\mathcal{C}^\infty$, like $\mathcal{P}^\infty$ and $\mathcal{C}^\infty$ defined in Expl. 4, in order to directly reuse the induction principle defined in [4].

Performing η-expansion can be required for variables or if an application instantiated a type, allowing it to reduce to a product. Hence, we will add those tags on the variable and application rules. Hence, one could imagine having the rules:

$$\text{(var')} \quad \frac{\Gamma \vdash A : s_i}{\Gamma, x : A \vdash x^A : A}\ x \notin \mathrm{dom}(\Gamma) \qquad \text{(app')} \quad \frac{\Gamma \vdash t : (x : A) \to B \quad \Gamma \vdash u : A}{\Gamma \vdash (t\,u)^{B\left[{}^{u}/_{x}\right]} : B\left[{}^{u}/_{x}\right]}$$

But those rules, do not have the property that if a term is well-type, its subterms are well-typed with a smaller tree, because of the substitution performed on B. Fortunately, the induction principle defined by Barthe, Hatcliff and Sørensen [4] ensures us that, if we annotate the applications with normal form, this property is verified, leading to:

$$\text{(app'')} \quad \frac{\Gamma \vdash t : (x : A) \to B \quad \Gamma \vdash u : A}{\Gamma \vdash (t\,u)^{B\left[{}^{u}/_{x}\right]\downarrow} : B\left[{}^{u}/_{x}\right]}$$

One must note here that the same tags can be added to the universe polymorph version of the full PTS considered. Indeed, Prop. 10 ensures us that the set of typable terms are the same in both systems. However, it would require to annotate the $x[l_1, \dots, l_n]$, generating an overweight in the proof, without introducing technicality.

▶ **Definition 21** (Translation). *Given an annotated well-typed term t in a Full Pure Type System, with the rules (var′) and (app″) and the conversion enriched with η, we translate t by:* $\left\|x^A\right\| = \eta E\ |s_A|_S\ \|A\|\ \mathtt{x}$; $\quad \|s\| = \mathtt{code}\ |s|_S$; $\quad \left\|(t\,u)^A\right\| = \eta E\ |s_A|_S\ \|A\|\ (\|t\|\ \|u\|)$;

$\left\|\lambda x^A.t\right\| = \lambda(\mathtt{x}\ \mathtt{:}\ \mathtt{Term}\ \ |s_A|_S\ \|A\|).\|t\|$;

$\|(x : A) \to B\| = \mathtt{prod}\,|s_A|_S\ |s_B|_S\ \|A\|\ (\lambda\mathtt{x}\ \mathtt{:}\ \mathtt{Term}\ |s_1|_S\ \|A\|.\|B\|)$;

s_A and s_B are respectively the sorts of A and B, and $|.|_S$ is the translation of sorts.

The correctness of our translation relies on the preservation of conversion. This result comes from the three following lemmas:

▶ **Lemma 22** (No ηE on translation). *If $\Gamma \vdash t : A$, then $\eta E\ |s_A|_S\ \|A\!\downarrow\|\ \|t\| \leftrightsquigarrow^* \|t\|$.*

▶ **Lemma 23** (Substitution). *If t is well-typed in the context $\Gamma, x_1 : A_1, \dots, x_n : A_n, \Gamma'$ and if $\Gamma \vdash u_1 : A_1$,…,$\Gamma \vdash u_n : A_n$ then* $\|t\| \left[{}^{\|u_i\|}/_{x_i}\right]_{i\in\{1,\dots,n\}} \leftrightsquigarrow^* \left\|t\left[{}^{u_i}/_{x_i}\right]_{i\in\{1,\dots,n\}}\right\|$.

▶ **Lemma 24** (Reduction). *If $\Gamma \vdash t : A$ and $t \rightsquigarrow u$, then $\|t\| \leftrightsquigarrow^* \|u\|$.*

We prove those three lemmas, in this order, by a mutual induction on the combination of the subterm ordering and reduction on a multiset of terms (this multiset is of size at most 2), called "measure" in the proofs.

Proof of Lem. 22. We use $\{\!|t|\!\}$ as the measure. If the normal form of A is a sort, then one can conclude using the rule ηS. We proceed by case on t for the remaining cases:

- If $t = x^B$, then $\eta E\ |s_A|_S\ \|A\!\downarrow\|\ \|t\| = \eta E\ |s_A|_S\ \|A\!\downarrow\|\ (\eta E\ |s_B|_S\ \|B\|\ x) \rightsquigarrow_{\eta I} \|t\|$.
- If $t = (u\,v)^B$, then it is again a direct consequence of the rule ηI
- If $t = \lambda x_1^{B_1} \dots \lambda x_n^{B_n}.u$, with u not a λ-abstraction.
  There is a C such that: $A\!\downarrow = (x_1 : B_1\!\downarrow) \to \dots \to (x_n : B_n\!\downarrow) \to C$. We denote by s_i the sort of $(x_i : B_i\!\downarrow) \to \dots \to (x_n : B_n\!\downarrow) \to C$. We have:

$$
\begin{aligned}
&\eta E\ |s_A|_S\ \|A{\downarrow}\|\ \|t\|\\
&= \ \eta E\ |s_A|_S\ (\texttt{prod}\ |s_{B_1}|_S\ |s_2|_S\ \|B_1{\downarrow}\|\ (\lambda(x_1 : \texttt{Term}\ |s_{B_1}|_S\ \|B_1{\downarrow}\|).\\
&\qquad \texttt{prod}\ldots |s_{B_n}|_S\ |s_C|_S\ \|B_n{\downarrow}\|\ (\lambda(x_n : \texttt{Term}\ |s_{B_n}|_S\ \|B_n{\downarrow}\|).\ \|C\|)\ldots))\\
&\qquad (\lambda(x_1 : \texttt{Term}\ |s_{B_1}|_S\ \|B_1\|)\ldots\lambda(x_n : \texttt{Term}\ |s_{B_n}|_S\ \|B_n\|).\ \|u\|)\\
&\rightsquigarrow_{\eta P}\ \lambda(x_1 : \texttt{Term}\ |s_1|_S\ \|B_1{\downarrow}\|).\eta E\ |s_2|_S\ ((\lambda\ldots\|C\|)(\eta E\ |s_1|_S\ \|B_1{\downarrow}\|\ x_1))\\
&\qquad ((\lambda x_1\ldots\|u\|)(\eta E\ |s_1|_S\ \|B_1{\downarrow}\|\ x_1))\\
&\rightsquigarrow_{\beta}^{2}\ \lambda(x_1 : \texttt{Term}\ |s_1|_S\ \|B_1{\downarrow}\|).\eta E\ |s_2|_S\ (\texttt{prod}\ |s_{B_2}|_S\ |s_3|_S\ \|B_2{\downarrow}\|\ldots\|C\|)\,\sigma\,(\lambda x_2\ldots\|u\|)\sigma\\
&\text{with } \sigma = \left[{}^{\eta E\ |s_1|_S\ \|B_1{\downarrow}\|\ x_1}/_{x_1}\right]\\
&(\rightsquigarrow_{\eta P}\rightsquigarrow_{\beta}^{2})^{n-1}\ \lambda(x_1 : \texttt{Term}\ |s_1|_S\ \|B_1{\downarrow}\|)\ldots\lambda(x_n : \texttt{Term}\ |s_n|_S\ \|B_n{\downarrow}\|).\eta E\ |s_C|_S\ \|C\|\,\tau\ \|u\|\,\tau\\
&\text{with } \tau = \left[{}^{\eta E\ |s_i|_S\ \|B_i{\downarrow}\|\ x_i}/_{x_i}\right]_{i\in\{1,\ldots,n\}}\\
&\leftrightsquigarrow^{*}_{Lem.23}\ \lambda(x_1 : \texttt{Term}\ |s_1|_S\ \|B_1{\downarrow}\|)\ldots\lambda(x_n : \texttt{Term}\ |s_n|_S\ \|B_n{\downarrow}\|).\eta E\ |s_C|_S\ \|C\tau'\|\ \|u\tau'\|\\
&\text{with } \tau' = \left[{}^{x_i^{B_i\downarrow}}/_{x_i}\right]_{i\in\{1,\ldots,n\}}\\
&\leftrightsquigarrow^{*}_{IH}\ \left\|\lambda x_1^{B_1}\ldots\lambda x_n^{B_n}.u\right\|
\end{aligned}
$$ ◀

Proof of Lem. 23. There, the measure is $\{\!|t, t\left[{}^{u_i}/_{x_i}\right]_{i\in\{1,\ldots,n\}}|\!\}$. Depending on the shape of t, we have:

- If t is a sort, the substitution does not have any impact.
- If $t = x_i^{A_i}$, $\|t\| = \eta E\ |s_{A_i}|_S\ \|A_i\|\ x_i$, so $\|t\|\left[{}^{\|u_i\|}/_{x_i}\right]_i = \eta E\ |s_{A_i}|_S\ \|A_i\|\ \|u_i\|$. By Lem. 24, $\|A_i\| \leftrightsquigarrow^{*} \|A_i{\downarrow}\|$ and one can conclude by Lem. 22 that $\|t\|\left[{}^{\|u_i\|}/_{x_i}\right]_i \leftrightsquigarrow^{*} \|u_i\|$.
- If $t = y^B$ with $y \notin \{x_i\}_i$, then $\|t\| = \eta E\ |s_B|_S\ \|B\|\ y$,so

$$\|t\|\left[{}^{\|u_i\|}/_{x_i}\right]_i = \eta E\ |s_B|_S\ \|B\|\left[{}^{\|u_i\|}/_{x_i}\right]_i\ y \leftrightsquigarrow^{*}_{IH} \left\|y^{B\left[{}^{u_i}/_{x_i}\right]_i}\right\| = \left\|t\left[{}^{u_i}/_{x_i}\right]_i\right\|.$$

- If $t = \lambda y^B.v$, then $\|t\| = \lambda(y : \texttt{Term}\ |s_B|_S\ \|B\|).\ \|v\|$, so

$$
\begin{aligned}
\|t\|\left[{}^{\|u_i\|}/_{x_i}\right]_i &= \lambda(y : \texttt{Term}\ |s_B|_S\ \|B\|\left[{}^{\|u_i\|}/_{x_i}\right]_i).\ \|v\|\left[{}^{\|u_i\|}/_{x_i}\right]_i\\
&\leftrightsquigarrow^{*}_{IH}\ \lambda(y : \texttt{Term}\ |s_B|_S\ \left\|B\left[{}^{u_i}/_{x_i}\right]_i\right\|).\ \left\|v\left[{}^{u_i}/_{x_i}\right]_i\right\| = \left\|(\lambda y^B.v)\left[{}^{u_i}/_{x_i}\right]_i\right\|
\end{aligned}
$$

The other cases are straightforward, just like the previous two. ◀

Proof of Lem. 24. We use $\{\!|t|\!\}$ as the measure. If the reduction is not at the head of t, then the result follows by the induction hypothesis.

Otherwise, the reduction occurs at the head of the term. It can be either η or β reduction.

(η) Then $t = \lambda x^A.(u\,x^A)^B$ and u is either a variable, an application or a λ-abstraction. In every case $\|t\| = \lambda(x : \texttt{Term}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\|\ (\|u\|\ (\eta E\ |s_A|_S\ \|A\|\ x))$.

- If $u = y^C$, then $C{\downarrow} = (x : A{\downarrow}) \to B$.

$$
\begin{aligned}
\|u\| = \eta E\ |s_C|_S\ \|C\|\ y &\leftrightsquigarrow^{*}_{IH} \eta E\ |s_C|_S\ \|(x : A{\downarrow}) \to B\|\ y\\
&= \eta E\ |s_C|_S\ (\texttt{prod}\ |s_A|_S\ |s_B|_S\ \|A{\downarrow}\|\ (\lambda(x : \texttt{Term}\ |s_A|_S\ \|A{\downarrow}\|).\ \|B\|))\,y\\
&\rightsquigarrow_{\eta P} \lambda(x : \texttt{Term}\ |s_A|_S\ \|A{\downarrow}\|).\eta E\ |s_B|_S\ \|B\|\ (y\,(\eta E\ |s_A|_S\ \|A{\downarrow}\|\ x))
\end{aligned}
$$

When we instantiate $\|t\|$ in this case, we get:

$$\begin{aligned}\|t\| \rightsquigarrow_{\beta}\ & \lambda(x : \texttt{Term}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\| \\ & (\eta E\ |s_B|_S\ \|B\| \left[\eta E\ |s_A|_S\ \|A\|\ x/_{x}\right] (y\,(\eta E\ |s_A|_S\ \|A{\downarrow}\|\ (\eta E\ |s_A|_S\ \|A\|\ x)))) \\ \rightsquigarrow_{\eta I}\ & \lambda(x : \texttt{Term}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\|\ (y\,(\eta E\ |s_A|_S\ \|A{\downarrow}\|\ x)) \leftrightsquigarrow^{*}_{IH}\ \|u\|\end{aligned}$$

- If $u = (v\,w)^{(x:A\downarrow)\to B}$.

$$\begin{aligned}\|u\| &= \eta E\ |C|_S\ \|(x : A{\downarrow}) \to B\|\ (\|v\|\ \|w\|) \\ &\rightsquigarrow_{\eta P} \lambda(x : \texttt{Term}\ |s_A|_S\ \|A{\downarrow}\|).\eta E\ |s_B|_S\ \|B\|\ (\|v\|\ \|w\|\ (\eta E\ |s_A|_S\ \|A{\downarrow}\|\ x))\end{aligned}$$

Instantiating $\|t\|$ in this case give:

$$\begin{aligned}\|t\| \rightsquigarrow_{\beta}\ & \lambda(x : \texttt{Term}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\|\ (\eta E\ |s_B|_S\ \|B\| \left[\eta E\ |s_A|_S\ \|A\|\ x/_{x}\right] \\ & (\|v\|\ \|w\|\ (\eta E\ |s_A|_S\ \|A{\downarrow}\|\ (\eta E\ |s_A|_S\ \|A\|\ x))))\end{aligned}$$

Since v and w do not contain x free.

$$\rightsquigarrow_{\eta I}\ \lambda(x : \texttt{Term}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\|\ (\|v\|\ \|w\|\ (\eta E\ |s_A|_S\ \|A{\downarrow}\|\ x)) \leftrightsquigarrow^{*}_{IH}\ \|u\|$$

- If $u = \lambda y^{C}.v$, then $C{\downarrow} = A{\downarrow}$, then $\|u\| = \lambda(y : \texttt{Term}\ |s_C|_S\ \|C\|).\,\|v\|$. Then,

$$\begin{aligned}\|t\| & \rightsquigarrow_{\beta} \lambda(x : \texttt{Term}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\|\ \|v\| \left[(\eta E\ |s_A|_S\ \|A\|\ x)/_{y}\right] \\ & \leftrightsquigarrow^{*}_{Lem.23} \lambda(x : \texttt{Term}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\| \left\| v \left[{}^{x}/_{y}\right] \right\|\end{aligned}$$

$(\lambda y.v)\,x$ is a subterm of t.

$$\leftrightsquigarrow^{*}_{Lem.22} \lambda(x : \texttt{Term}\ |s_A|_S\ \|A\|).\left\| v \left[{}^{x}/_{y}\right] \right\| =_{\alpha} \|u\|$$

(β) Then $t = ((\lambda x^{A}.v)\,w)^{B}$ and $u = v\,[{}^{w}/_{x}]$. We have :

$$\begin{aligned}\|t\| &= \eta E\ |s_B|_S\ \|B\|\ ((\lambda(x : \texttt{Term}\ |s_A|_S\ \|A\|).\,\|v\|)\ \|w\|) \\ &\rightsquigarrow_{\beta} \eta E\ |s_B|_S\ \|B\|\ \|v\| \left[{}^{\|w\|}/_{x}\right] \\ &\leftrightsquigarrow^{*}_{Lem.23} \eta E\ |s_B|_S\ \|B\|\ \|v\,[{}^{w}/_{x}]\| \leftrightsquigarrow^{*}_{Lem.22}\ \|v\,[{}^{w}/_{x}]\|\end{aligned}$$

v and $v\,[{}^{w}/_{x}]$ are respectively subterm and reduct of t, hence Lem. 23 applies. ◀

From those three lemmas, one can conclude that

▶ **Theorem 25** (Correctness of the translation). *If $\Gamma \vdash t : A$ and $t \leftrightsquigarrow^{*} u$, then $\|t\| \leftrightsquigarrow^{*} \|u\|$.*

5 Implementation

Agda [18, 17] is a dependently-typed programming languages, based on an extension of Martin-Löf type theory, Luo's Unifying Theory of dependent Types [15, Chapter 9], which features both universe polymorphism and η-conversion. Dedukti [10, 2] is an implementation of the $\lambda\Pi$-calculus modulo rewriting, which was recently enriched with conversion modulo associativity and commutativity.

Developping a prototypical translator [7] from Agda to Dedukti allowed the author to give a concrete application to the ideas presented in Sections 3 and 4.

However, Agda offers its users a logic much richer than a universe polymorphic pure type system with η-conversion. First of all, Agda permits to declare inductive types and then to define functions using dependent pattern-matching on the constructors of this type. This behaviour can easily be replicated in Dedukti, by declaring new symbols for inductive types, constructors and functions and rewrite rules for each case of the dependent pattern-matching. Just like sorts and products have an encoded and a decoded version, linked by the application of the function `Term`, the type has two translation, one as code and one decoded, linked by a rewrite rule enriching the definition of `Term`. Analogously, one rewrite rule is added to enrich the definition of ηE.

▶ **Example 26.** The Agda declaration of the addition of natural numbers:

```
data Nat : Set where                    _+_ : Nat → Nat → Nat
  zero : Nat                            zero  + m = m
  suc  : (n : Nat) → Nat                suc n + m = suc (n + m)
```

is translated in Dedukti by:

```
constant TYPE__Nat : TYPE.           constant Nat : Univ (set 0).
Term _ Nat ⟶ TYPE__Nat.               ηE _ Nat t ⟶ t.
constant Nat__zero: Term (set 0) Nat.
constant Nat__suc: Term (set 0) (prod (set 0) (set 0) Nat (λ n, Nat)).
symbol {|_+_|} : Term (set 0) (prod (set 0) (set 0) Nat
                 (λ _0, prod (set 0) (set 0) Nat (λ _1, Nat))).
{|_+_|} Nat__zero    m ⟶ m.
{|_+_|} (Nat__suc n) m ⟶ Nat__suc ({|_+_|} n m).
```

We can observe, that `Nat` in Agda became `TYPE__Nat` and `Nat` in Dedukti, and two rules have been added: one to state that `TYPE__Nat` is the decoding of `Nat` and the other to extend the definition of ηE.

Each declaration of a new type consists in adding a new constructor to the type `Univ s`. The new rules on ηE and `Term` are here to ensure that the pattern-matching on this type remains exhaustive, in order to completely get rid of administrative encoding operators on the normal forms of values.

One can note, that the enrichment of the functions `Term` and ηE are left to the will of the author of the translation. This proves to be a good feature, since the η-conversion of Agda does not restrict to product types, but also concerns records (η-conversion of records is also sometimes called "surjective pairing" and means that if t lives in $\sum_{x:A} B$, then t and $(\mathrm{fst}\, t, \mathrm{snd}\, t)$ are convertible). This does not require to introduce a new symbol for this enrichment of the conversion, but just to define adequate rules on ηE.

▶ **Example 27.** The declaration of this record:

```
record r : Set1 where                 constructor cons
  field A : Set                       field b : A
```

is translated by:

```
constant TYPE__r : TYPE.            constant r : Univ (set (s 0)).
Term _ r ⟶ TYPE__r.
ηE _ r y ⟶ r__cons (r__A y) (ηE 0 (r__A y) (r__b y)).
constant r__cons : Term (set (s 0)) (prod (set (s 0)) (set (s 0))
          (code (set 0)) (λ A, prod (set 0) (set (s 0)) A (λ b, r))).
symbol r__A : Term (set (s 0))
              (prod (set (s 0)) (set (s 0)) r (λ r, code (set 0))).
symbol r__b : Term (set (s 0))
              (prod (set (s 0)) (set 0) r (λ r, r__A r)).
r__A (r__cons A b) ⟶ A.       r__b (r__cons A b) ⟶ ηE 0 A b.
```

The rule to define the η-expansion of an element of `r` states that if y is of type `r`, then $y \equiv \{a = y.a; b = y.b\}$.

This translator is available at `https://github.com/Deducteam/Agda2Dedukti`, the directory `theory/` contains the encoding presented in Sections 3 and 4. It is able to translate and type-check 162 files of Agda's standard library [9].

6 Conclusion and Future Work

We presented in this article a correct encoding of universe polymorphism in $\lambda\Pi$-modulo rewriting, meaning that every term typable in the original system is translated to a typable term. We also presented a rewrite system to decide equality in the max-plus algebra, which is a comon universe algebra.

Furthermore, we proposed an operator ηE to encode shallowly a type-directed rule, like η-conversion, since the translation of an application really involves the application of the translation of a term to the other one, reducing the interleaving between the computation steps coming from the original system and the steps related to the encoding.

Finally, we applied those results to the practical case of the translation of the proof system Agda, which offers, among others, the features we targeted, allowing us to provide Dedukti users with more than 500 declarations of types, constructors or functions, originating from Agda's standard library.

We proved that translation of well-typed terms remain typable in our encoding. However, it could be that our encoding is over-permissive and type-checks much more terms than the original system. Hence, one could envision a conservativity theorem, stating that if the translation of a type is inhabited, then the type is also inhabited in the original system. For implementability purposes, we have chosen an encoding with finitely many symbols. Such a theorem has only been proved [8, 1], for encodings of PTS with as many symbols as sorts, axioms and rules. Extending those theorems to our setting is a short-term goal.

Regarding the implementation, making the translator more complete is naturally an objective, however, it involves more theoretical problems, which are long run research programs. For instance, how size types or co-inductive types can be encoded in the $\lambda\Pi$-calculus modulo rewriting is not known yet.

Now that proofs have been translated to the logical framework Dedukti, they can be analysed, and (when it is possible) exported to other proof assistants, like what was done with proofs originating from the arithmetic library of Matita [20].

References

1 Ali Assaf. *A Framework for Defining Computational Higher-Order Logics*. PhD thesis, École polytechnique, France, 2015.

2 Ali Assaf, Guillaume Burel, Raphaël Cauderlier, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti: a logical framework based on the $\lambda\pi$-calculus modulo theory, 2019. URL: `http://www.lsv.fr/~dowek/Publi/expressing.pdf`.

3 Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 2)*, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.

4 Gilles Barthe, John Hatcliff, and Morten Heine Sørensen. An induction principle for pure type systems. *Theoretical Computer Science*, 266(1-2):773–818, 2001.

5 Stefano Berardi. *Type Dependence and Constructive Mathematics*. Phd, Carnegie Mellon University, Dept. Comp. Sci., Pittsburgh, Pennsylvania, USA and Torino University, Dipartimento di Informatica, Torino, Italy, 1990.

6 Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The $\lambda\Pi$-calculus Modulo as a Universal Proof Language. *PxTP*, page 16, 2012.

7 Jesper Cockx and Guillaume Genestier. Agda2dedukti. `https://github.com/Deducteam/Agda2Dedukti`, 2019.

8 Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-Pi-calculus modulo. In *Proceedings of the 8th International Conference on Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science 4583, 2007.

9 Nils Anders Danielsson, Matthew Daggitt, and Guillaume Allais. Agda standard library. `https://github.com/agda/agda-stdlib`, 2010-.

10 Deducteam. Dedukti. `https://deducteam.github.io/`, 2011-.

11 Gilles Dowek, Gérard Huet, and Benjamin Werner. On the Definition of the Eta-long Normal Form in Type Systems of the Cube. In *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 115–130, 1993.

12 Herman Geuvers and Mark-Jan Nederhof. Modular proof of strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2):155–189, 1991.

13 Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *Journal of the ACM*, 40(1):143–184, January 1993.

14 Robert Harper and Randy Pollack. Type Checking with Universes. *Theoretical Computer Science*, 89:107–136, 1991.

15 Zhaohui Luo. *Computation and reasoning - a type theory for computer science*, volume 11 of *International series of monographs on computer science*. Oxford University Press, 1994.

16 Dale Miller. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *J. Log. Comput.*, 1(4):497–536, 1991.

17 Ulf Norell. *Towards a practical programming language based on dependent type theory*. Phd, Chalmers University of Technology, Gothenburg, Sweden, 2007.

18 Ulf Norell, Andreas Abel, Niels Anders Danielsson, Makoto Takeyama, and Catarina Coquand. Agda. `https://github.com/agda/agda`, 2007-, v1.0 : 1999.

19 Matthieu Sozeau and Nicolas Tabareau. Universe Polymorphism in Coq. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, pages 499–514. Springer, 2014.

20 François Thiré. Sharing a Library between Proof Assistants: Reaching out to the HOL Family. In *Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP*, pages 57–71, 2018.

The Difference λ-Calculus: A Language for Difference Categories

Mario Alvarez-Picallo
University of Oxford, UK
mario.alvarez-picallo@cs.ox.ac.uk

C.-H. Luke Ong
University of Oxford, UK
luke.ong@cs.ox.ac.uk

Abstract

Cartesian difference categories are a recent generalisation of Cartesian differential categories which introduce a notion of "infinitesimal" arrows satisfying an analogue of the Kock-Lawvere axiom, with the axioms of a Cartesian differential category being satisfied only "up to an infinitesimal perturbation". In this work, we construct a simply-typed calculus in the spirit of the differential λ-calculus equipped with syntactic "infinitesimals" and show how its models correspond to difference λ-categories, a family of Cartesian difference categories equipped with suitably well-behaved exponentials.

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

Keywords and phrases Cartesian difference categories, Cartesian differential categories, Change actions, Differential lambda-calculus, Difference lambda-calculus

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.32

1 Introduction

A recent series of works introduced the concept of change actions and differential maps between them [5, 4] in order to account for settings equipped with derivative-like operations. Although the motivating example was the eminently practical field of incremental computation, these structures appear in more abstract settings such as the calculus of finite differences and Cartesian differential categories.

Of particular interest are Cartesian difference categories [2], a well-behaved class of change action models [4] that are much closer to the strong axioms of a Cartesian differential category [8] while remaining general enough for interpreting discrete calculus. A Cartesian difference category is a left additive category equipped with an "infinitesimal extension", an operation that sends an arrow f to an arrow $\varepsilon(f)$ which should be understood as f being multiplied by an "infinitesimal" element – infinitesimal in the sense that it verifies the Kock-Lawvere axiom from synthetic differential geometry (see [14] for an introduction to SDG).

The interest of Cartesian differential categories is in part motivated by the fact that they provide models for the differential λ-calculus [13, 11], which extends the λ-calculus with linear combinations of terms and an operator that differentiates arbitrary λ-abstractions. The claim that differentiation in the differential λ-calculus corresponds to the standard, "analytic" notion is then justified by its interpretation in (a well-behaved class of) Cartesian differential categories [9, 16].

It is reasonable to ask, then, whether there is a similar calculus that captures the behavior of derivatives in difference categories – especially since, as it has been shown, these subsume differential categories. The issue is far from trivial, as many of the properties of the differential λ-calculus crucially hinge on derivatives being linear. Through this work we provide an affirmative answer to this question by defining untyped and simply-typed variants for a simple calculus which extends the differential λ-calculus with a notion of derivative more suitable to the Cartesian difference setting.

For brevity most proofs have been omitted. Some of the most important proof sketches and the auxiliary results involved can be found in the appendices. The full details are available as part of the first author's doctoral dissertation [1].

2 Cartesian Difference Categories

The theory of Cartesian difference categories is developed and discussed at length in [2, 3], but we present here the main definitions and results which we will use throughout the paper, referring the reader to [3] for the proofs.

▶ **Definition 1.** A *Cartesian left additive category* ([8, Definition 1.1.1.]) $\mathbf{C}$ is a Cartesian category where every hom-set $\mathbf{C}[A, B]$ is endowed with the structure of a commutative monoid $(\mathbf{C}[A, B], +, 0)$ such that $0 \circ f = 0$, $(f + g) \circ h = (f \circ h) + (g \circ h)$ and $\langle f_1, f_2 \rangle + \langle g_1, g_2 \rangle = \langle f_1 + g_1, f_2 + g_2 \rangle$.

An *infinitesimal extension* ([2, Definition 8]) in a Cartesian left additive category $\mathbf{C}$ is a choice of a monoid homomorphism $\varepsilon : \mathbf{C}[A, B] \to \mathbf{C}[A, B]$ for every hom-set in $\mathbf{C}$. That is, $\varepsilon(f + g) = \varepsilon(f) + \varepsilon(g)$ and $\varepsilon(0) = 0$. Furthermore, we require that ε be compatible with the Cartesian structure, in the sense that $\varepsilon(\langle f, g \rangle) = \langle \varepsilon(f), \varepsilon(g) \rangle$.

▶ **Definition 2.** A *Cartesian difference category* ([2, Definition 9]) is a Cartesian left additive category with an infinitesimal extension ε which is equipped with a *difference combinator* $\partial[-]$ of the form:

$$\frac{f : A \to B}{\partial[f] : A \times A \to B}$$

satisfying the following coherence conditions (writing $\partial^2[f]$ for $\partial[\partial[f]]$):

[C∂.0] $f \circ (x + \varepsilon(u)) = f \circ x + \varepsilon(\partial[f] \circ \langle x, u \rangle)$
[C∂.1] $\partial[f + g] = \partial[f] + \partial[g]$, $\partial[0] = 0$, and $\partial[\varepsilon(f)] = \varepsilon(\partial[f])$
[C∂.2] $\partial[f] \circ \langle x, u + v \rangle = \partial[f] \circ \langle x, u \rangle + \partial[f] \circ \langle x + \varepsilon(u), v \rangle$ and $\partial[f] \circ \langle x, 0 \rangle = 0$
[C∂.3] $\partial[\mathbf{id}_A] = \pi_2$ and $\partial[\pi_1] = \pi_1 \circ \pi_2$ and $\partial[\pi_2] = \pi_2 \circ \pi_2$
[C∂.4] $\partial[\langle f, g \rangle] = \langle \partial[f], \partial[g] \rangle$ and $\partial[!_A] = !_{A \times A}$
[C∂.5] $\partial[g \circ f] = \partial[g] \circ \langle f \circ \pi_1, \partial[f] \rangle$
[C∂.6] $\partial^2[f] \circ \langle \langle x, u \rangle, \langle 0, v \rangle \rangle = \partial[f] \circ \langle x + \varepsilon(u), v \rangle$
[C∂.7] $\partial^2[f] \circ \langle \langle x, u \rangle, \langle v, 0 \rangle \rangle = \partial^2[f] \circ \langle \langle x, v \rangle, \langle u, 0 \rangle \rangle$

As noted in [2], the axioms in a Cartesian differential category (see e.g. **[C∂.1–7]** in [8]) correspond to the analogous axioms of the Cartesian difference operator, modulo certain "infinitesimal" terms, i.e. terms of the form $\varepsilon(f)$. We state here the following two properties, whose proofs can be found in [3].

▶ **Lemma 3.** Given any map $f : A \to B$ in a Cartesian difference category $\mathbf{C}$, its derivative $\partial[f]$ satisfies the following equations:

i. $\partial[f] \circ \langle x, \varepsilon(u) \rangle = \varepsilon(\partial[f]) \circ \langle x, u \rangle$
ii. $\varepsilon(\partial^2[f]) \circ \langle \langle x, u \rangle, \langle v, 0 \rangle \rangle = \varepsilon^2(\partial^2[f]) \circ \langle \langle x, u \rangle, \langle v, 0 \rangle \rangle$

3 Difference λ-Categories

In order to give a semantics for the differential λ-calculus, it does not suffice to ask for a Cartesian differential category equipped with exponentials – the exponential structure has to be compatible with both the additive and the differential structure, in the sense of [9, Definition 4.4]. For difference categories we will require an identical equation, together with a condition requiring higher-order functions to respect the infinitesimal extension.

▶ **Definition 4.** We remind the reader that a Cartesian left additive category is *Cartesian closed left additive* ([9, Definition 4.2]) whenever it is Cartesian closed and satisfies $\Lambda(f+g) = \Lambda(f) + \Lambda(g), \Lambda(0) = 0$.

A Cartesian difference category $\mathbf{C}$ is a *difference λ-category* if it Cartesian closed left additive and satisfies the following additional axioms:

$$[\partial\lambda.\mathbf{1}] \quad \partial\left[\Lambda(f)\right] = \Lambda\left(\partial\left[f\right] \circ \langle(\pi_1 \times \mathbf{id}), (\pi_2 \times 0)\rangle\right)$$
$$[\partial\lambda.\mathbf{2}] \quad \Lambda(\varepsilon(f)) = \varepsilon\left(\Lambda(f)\right)$$

Equivalently, let $\mathbf{sw}$ denote the map $\langle\langle\pi_{11}, \pi_2\rangle, \pi_{21}\rangle : (A \times B) \times C \to (A \times C) \times B$. Then the condition $[\partial\lambda.\mathbf{1}]$ can be written in terms of $\mathbf{sw}$ as:

$$\partial\left[\Lambda(f)\right] \coloneqq \Lambda\left(\partial\left[f\right] \circ (\mathbf{id} \times \langle\mathbf{id}, 0\rangle) \circ \mathbf{sw}\right)$$

Axiom $[\partial\lambda.\mathbf{1}]$ is identical to its differential analogue [9, Definition 4.4], and it follows the same broad intuition. Given a map $f : A \times B \to C$, we usually understand the composite $\partial\left[f\right] \circ (\mathbf{id}_{A\times B} \times (\mathbf{id}_A \times 0_B)) : (A \times B) \times A \to C$ as a partial derivative of f with respect to its first argument. Hence, just as it was with differential λ-categories, axiom $[\partial\lambda.\mathbf{1}]$ states that the derivative of a curried function is precisely the derivative of the uncurried function with respect to its first argument.

▶ **Example 5.** Let $\mathbf{C}$ be a differential λ-category. Then the trivial Cartesian difference category obtained by setting $\varepsilon(f) = 0$ (as in [2, Proposition 1]) is a difference λ-category. Furthermore, the Kleisli category $\mathbf{C}_\mathsf{T}$ induced by its tangent bundle monad (as in [2, Proposition 6]) is also a difference λ-category.

▶ **Example 6.** The category $\overline{\mathbf{Ab}}$ ([2, Section 5.2]), which has Abelian groups as objects and arbitrary functions between their carrier sets as morphisms, is a difference λ-category with infinitesimal extension $\varepsilon(f) = f$ and difference combinator $\partial\left[f\right](x, u) = f(x+u) - f(x)$. Given groups G, H, the exponential $G \Rightarrow H$ is the set of (set-theoretic) functions from G into H, endowed with the group structure of H lifted pointwise (that is, $(f+g)(x) = f(x)+g(x)$). Evidently the exponential respects the monoidal structure and the infinitesimal extension. We check that it also verifies axiom $[\partial\lambda.\mathbf{1}]$:

$$\begin{aligned}\partial\left[\Lambda(f)\right](x, u)(y) &= \Lambda(f)(x+u)(y) - \Lambda(f)(x)(y)\\ &= f(x+u, y) - f(x, y)\\ &= \Lambda(\partial\left[f\right] \circ (\mathbf{id} \times \langle\mathbf{id}, 0\rangle) \circ \mathbf{sw})(x, u)(y)\end{aligned}$$

A central property of differential λ-categories is a deep correspondence between differentiation and the evaluation map. As one would expect, the partial derivative of the evaluation map gives one a first-class derivative operator (see, for example, [9, Lemma 4.5], which provides an interpretation for the differential substitution operator in the differential λ-calculus). This property still holds in difference categories, although its formulation is somewhat more involved.

▶ **Lemma 7.** For any $\mathbf{C}$-morphisms $\Lambda(f) : A \to (B \Rightarrow C), e : A \to B$, the following identities hold:

i. $\partial\left[\mathbf{ev} \circ \langle \Lambda(f), e\rangle\right] = \mathbf{ev} \circ \langle \partial\left[\Lambda(f)\right], e \circ \pi_1\rangle + \partial\left[f\right] \circ \langle\langle \pi_1 + \varepsilon(\pi_2), e \circ \pi_1\rangle, \langle 0, \partial\left[e\right]\rangle\rangle$
ii. $\partial\left[\mathbf{ev} \circ \langle \Lambda(f), e\rangle\right] = \mathbf{ev} \circ \langle \partial\left[\Lambda(f)\right], e \circ \pi_1 + \varepsilon(\partial\left[e\right])\rangle + \partial\left[f\right] \circ \langle\langle \pi_1, e \circ \pi_1\rangle, \langle 0, \partial\left[e\right]\rangle\rangle$

A number of additional auxiliary results from the differential setting also hold for difference λ-categories, possibly with the introduction of some extra "infinitesimal" terms. Some of these will be stated in Appendix C.

4 The Difference λ-Calculus

We now set out to define λ_ε, a calculus in the vein of the differential λ-calculus which adds infinitesimal extensions and relaxes the linearity requirement. We proceed in a manner similar to Vaux [17, 18] in his treatment of the algebraic λ-calculus; that is, we will first define a set of "unrestricted" terms Λ_ε which we will later consider up to an equivalence relation arising from the theory of difference categories.

▶ **Definition 8.** The set Λ_ε of *unrestricted terms* of the λ_ε-calculus is given by the following inductive definition:

$$\text{Terms:} \quad s, t, e \quad \coloneqq \quad x \mid \lambda x.t \mid (s\ t) \mid \mathrm{D}(s) \cdot t \mid \varepsilon t \mid s + t \mid 0$$

assuming a countably infinite set of variables $x, y, z, \ldots$ is given. In what follows, we will speak of terms only up to α-equivalence, and assume by convention that all bound variables appearing in any term $t \in \Lambda_\varepsilon$ are different from its free variables.

Further to α-equivalence, we introduce here the notion of differential equivalence of terms. The role of this relation is, as in [18], to enforce the elementary algebraic properties of sums and actions. For example, we wish to treat the terms $\lambda x.(0 + \varepsilon(s + t))$ and $(\lambda x.\varepsilon t) + (\lambda x.\varepsilon s)$ as if they were equivalent (as it will be the case in the models). This equivalence relation also has the role of ensuring that the axioms of a Cartesian difference category are satisfied.

▶ **Definition 9.** A binary relation $\sim \subseteq \Lambda_\varepsilon \times \Lambda_\varepsilon$ is *contextual* whenever $t \sim t'$ and $s \sim s'$ implies

$$\lambda x.t \sim \lambda x.t' \quad \varepsilon t \sim \varepsilon t' \quad s\ t \sim s'\ t' \quad \mathrm{D}(s) \cdot t \sim \mathrm{D}(s') \cdot t' \quad s + t \sim s' + t'$$

▶ **Definition 10.** *Differential equivalence* $\sim_\varepsilon \subseteq \Lambda_\varepsilon \times \Lambda_\varepsilon$ is the least equivalence relation which is contextual and contains the relation $\sim_\varepsilon^1$ in Figure 1 below.

The above conditions can be separated in a number of conceptually distinct groups corresponding to their purpose. These are as follows:

- The first block of equations states that $+$ and 0 define a commutative monoid and that ε is a monoid homomorphism.
- The second block of equations amounts to stating that the monoid and infinitesimal extension structure on functions is pointwise.
- The third block of equations implies (and is equivalent to stating) that addition and infinitesimal extension are "linear", in the sense that they coincide with their own derivatives (that is, $\partial\left[+\right] = + \circ \pi_2$ and $\partial\left[\varepsilon\right] = \varepsilon$).

$$\begin{array}{rcl} (s+t)+e & \sim^1_\varepsilon & s+(t+e) \\ s+0 & \sim^1_\varepsilon & s \\ s+t & \sim^1_\varepsilon & t+s \\ \varepsilon 0 & \sim^1_\varepsilon & 0 \\ \varepsilon(s+t) & \sim^1_\varepsilon & \varepsilon s + \varepsilon t \end{array} \qquad \begin{array}{rcl} \lambda x.0 & \sim^1_\varepsilon & 0 \\ \lambda x.(s+t) & \sim^1_\varepsilon & (\lambda x.s)+(\lambda x.t) \\ \lambda x.\varepsilon t & \sim^1_\varepsilon & \varepsilon(\lambda x.t) \\ 0\ s & \sim^1_\varepsilon & 0 \\ (s+t)\ e & \sim^1_\varepsilon & (s\ e)+(t\ e) \\ (\varepsilon s)\ t & \sim^1_\varepsilon & \varepsilon(s\ t) \end{array}$$

$$\begin{array}{rcl} \mathrm{D}(0)\cdot e & \sim^1_\varepsilon & 0 \\ \mathrm{D}(s+t)\cdot e & \sim^1_\varepsilon & (\mathrm{D}(s)\cdot e)+(\mathrm{D}(t)\cdot e) \\ \mathrm{D}(\varepsilon t)\cdot e & \sim^1_\varepsilon & \varepsilon(\mathrm{D}(t)\cdot e) \end{array} \qquad \begin{array}{rcl} \mathrm{D}(s)\cdot 0 & \sim^1_\varepsilon & 0 \\ \mathrm{D}(s)\cdot(t+e) & \sim^1_\varepsilon & \mathrm{D}(s)\cdot t+\mathrm{D}(s)\cdot e+\varepsilon(\mathrm{D}(\mathrm{D}(s)\cdot t)\cdot e) \\ \mathrm{D}(s)\cdot(\varepsilon t) & \sim^1_\varepsilon & \varepsilon(\mathrm{D}(s)\cdot t) \\ \mathrm{D}(\mathrm{D}(s)\cdot t)\cdot e & \sim^1_\varepsilon & \mathrm{D}(\mathrm{D}(s)\cdot e)\cdot t \\ \varepsilon^2\mathrm{D}(\mathrm{D}(s)\cdot t)\cdot e & \sim^1_\varepsilon & \varepsilon\mathrm{D}(\mathrm{D}(s)\cdot t)\cdot e \\ s\ (t+\varepsilon e) & \sim^1_\varepsilon & (s\ t)+\varepsilon((\mathrm{D}(s)\cdot e)\ t) \end{array}$$

Figure 1 Differential equivalence on unrestricted Λ_ε-terms.

- The fourth block of equations states structural properties of the derivative, such as the derivative conditions and the commutativity of second derivatives. Similar equations are also present in the differential λ-calculus, where they state instead that the derivative is additive.

Most of these equations correspond directly to properties of Cartesian difference categories, with the only exception being the requirement that $\mathrm{D}(s)\cdot(\varepsilon t) \sim_\varepsilon \varepsilon(\mathrm{D}(s)\cdot t)$ and the "duplication of infinitesimals" in $\varepsilon\mathrm{D}(\mathrm{D}(s)\cdot t)\cdot e = \varepsilon^2\mathrm{D}(\mathrm{D}(s)\cdot t)\cdot e$, which should be read as a syntactic formulation of the equations in Lemma 3. It would be possible to give an alternative presentation of the calculus where these equivalences are oriented and understood as reduction rules, and thus part of the operational semantics (as in e.g. Arrighi et al.'s treatment of the linear λ-calculus [7, 6]). While such a formulation would better reflect how a real machine might evaluate these expressions, it would make the study of confluence and termination harder.

▶ **Definition 11.** The set λ_ε of *well-formed terms*, or simply *terms*, of the λ_ε-calculus is defined as the quotient set $\lambda_\varepsilon := \Lambda_\varepsilon/{\sim_\varepsilon}$. Whenever t is an unrestricted term, we write $\underline{t}$ to refer to the well-formed term represented by t, that is to say, the $\sim_\varepsilon$-equivalence class of t.

The notion of differential equivalence allows us to ensure that our calculus reflects the laws of the underlying models, but has the unintended consequence that our λ_ε-terms are equivalence classes, rather than purely syntactic objects. We will proceed by defining a notion of canonical form of a term and a canonicalization algorithm which explicitly constructs the canonical form of any given term, thus proving that $\sim_\varepsilon$ is decidable.

▶ **Definition 12.** We define the sets $\mathrm{B}_\varepsilon \subset \mathrm{B}^+_\varepsilon \subset \mathrm{B}^*_\varepsilon \subset \mathrm{C}^+_\varepsilon \subset \mathrm{C}_\varepsilon (\subset \Lambda_\varepsilon)$ of *basic*, *positive*, *additive*, *positive canonical* and *canonical* terms according to the following grammar:

$$\begin{array}{rrcl} \text{Basic terms:} & s^{\mathbf{b}}, t^{\mathbf{b}}, e^{\mathbf{b}} \in \mathrm{B}_\varepsilon & := & x \mid \lambda x.t^{\mathbf{b}} \mid (s^{\mathbf{b}}\ t^*) \mid \mathrm{D}(s^{\mathbf{b}})\cdot t^{\mathbf{b}} \\ \text{Positive terms:} & s^+, t^+, e^+ \in \mathrm{B}^+_\varepsilon & := & s^{\mathbf{b}} \mid s^{\mathbf{b}} + (t^+) \\ \text{Additive terms:} & s^*, t^*, e^* \in \mathrm{B}^*_\varepsilon & := & 0 \mid s^+ \\ \text{Positive canonical terms:} & S^+, T^+ \in \mathrm{C}_\varepsilon & := & \varepsilon^k s^{\mathbf{b}} \mid \varepsilon^k s^{\mathbf{b}} + (S^+) \\ \text{Canonical terms:} & S, T \in \mathrm{C}_\varepsilon & := & 0 \mid S^+ \end{array}$$

We will sometimes abuse the notation and write $\underline{t^*}$ or $\underline{t^{\mathbf{b}}}$ to denote well-formed terms whose canonical form is an additive or basic term respectively.

Since every syntactic construct is additive except for application, basic terms may only contain additive terms as the arguments to a function application. As infinitesimal extensions are themselves additive, we also want to disallow terms such as $\varepsilon(s+t)$, instead factoring out the extension into $\varepsilon s + \varepsilon t$. A general canonical term $T \in \mathrm{C}_\varepsilon$ then has the form:

$$T = \varepsilon^{k_1} t_1^{\mathbf{b}} + (\varepsilon^{k_2} t_2^{\mathbf{b}} + (\cdots + \varepsilon^{k_n} t_n^{\mathbf{b}}) \cdots)$$

That is to say, a canonical term is similar to a polynomial with coefficients in the set of basic terms and a variable ε (but note that canonical terms are always written in their "fully distributed" form, that is, we write $\varepsilon s + (\varepsilon t + \varepsilon^2 e)$ rather than $\varepsilon((s+t) + \varepsilon e)$).

We will freely abuse notation and write $\sum_{i=1}^{n} \varepsilon^{k_i} t_i^{\mathbf{b}}$ to denote a general canonical term, as this form is easier to manipulate in many cases. In particular, the canonical term 0 is precisely the sum of zero terms. We will also write $S + T$ to refer to the obvious canonical term obtained by adding S and T and associating all the additions to the right.

▶ **Definition 13.** Given an unrestricted λ_ε-term $t \in \Lambda_\varepsilon$, we define its *canonical form* $\mathbf{can}\,(t)$ by structural induction on t as follows:

- $\mathbf{can}\,(0) \coloneqq 0$
- $\mathbf{can}\,(x) \coloneqq x$
- $\mathbf{can}\,(s+t) \coloneqq \mathbf{can}\,(s) + \mathbf{can}\,(t)$
- $\mathbf{can}\,(\varepsilon t) \coloneqq \varepsilon^* \mathbf{can}\,(t)$, where:

$$\varepsilon^* T \coloneqq \begin{cases} T & \text{if } T = \varepsilon^k \mathrm{D}(\mathrm{D}(e) \cdot u) \cdot v \\ \varepsilon^{k+1} t^{\mathbf{b}} & \text{if } T = \varepsilon^k t^{\mathbf{b}} \neq \varepsilon^k \mathrm{D}(\mathrm{D}(e) \cdot u) \cdot v \\ \sum_{i=1}^{n} \varepsilon^* T_i & \text{if } T = \sum_{i=1}^{n} T_i \end{cases}$$

- If $\mathbf{can}\,(t) = \sum_{i=1}^{n} \varepsilon^{k_i} t_i^{\mathbf{b}}$ then:

$$\mathbf{can}\,(\lambda x.t) \coloneqq \sum_{i=1}^{n} \varepsilon^{k_i} (\lambda x.t_i^{\mathbf{b}})$$

- If $\mathbf{can}\,(s) = \sum_{i=1}^{n} \varepsilon^{k_i} s_i^{\mathbf{b}}$ and $\mathbf{can}\,(t) = T$ then:

$$\mathbf{can}\,(\mathrm{D}(s) \cdot t) \coloneqq \sum_{i=1}^{n} ((\varepsilon^*)^{k_i} \mathbf{reg}\,(s_i^{\mathbf{b}}, T))$$

where the *regularization* $\mathbf{reg}\,(s, T)$ is defined by structural induction on T:

$$\begin{aligned} \mathbf{reg}\,(s, 0) &\coloneqq 0 \\ \mathbf{reg}\,(s, \varepsilon^k t^{\mathbf{b}} + T') &\coloneqq \left[(\varepsilon^*)^k\, \mathrm{D}\,(s) \cdot t^{\mathbf{b}}\right] + [\mathbf{reg}\,(s, T')] + \left[(\varepsilon^*)^{k+1}\, \mathrm{D}^*\,(\mathbf{reg}\,(s, T')) \cdot t^{\mathbf{b}}\right] \end{aligned}$$

and D^* denotes the extension of D by additivity in its first argument, that is to say:

$$\mathrm{D}^* \left(\sum_{i=1}^{n} \varepsilon^{k_i} s_i^{\mathbf{b}} \right) \cdot t^{\mathbf{b}} \coloneqq \sum_{i=1}^{n} \varepsilon^{k_i} \left(\mathrm{D}\left(s_i^{\mathbf{b}}\right) \cdot t^{\mathbf{b}} \right)$$

Observe that, whenever S is canonical and $t^{\mathbf{b}}$ is basic, the term $\mathrm{D}^*(S) \cdot t^{\mathbf{b}}$ is also canonical. Therefore, by induction, the regularization $\mathbf{reg}\,(s^{\mathbf{b}}, T)$ is indeed a canonical term, since canonicity is preserved by $\varepsilon^*, +$.

- If $\mathbf{can}\,(s) = \sum_{i=1}^{n} \varepsilon^{k_i} s_i^{\mathbf{b}}$ and $\mathbf{can}\,(t) = T$, then:

$$\mathbf{can}\,(s\ t) := \left[\sum_{i=1}^{n} \varepsilon^{k_i} \left(s_i^{\mathbf{b}}\ \mathbf{pri}(T)\right)\right] + \left[\varepsilon^{*}\left(\sum_{i=1}^{n} \mathbf{ap}(\mathbf{reg}\left(s_i^{\mathbf{b}}, \mathbf{tan}(T)\right), \mathbf{pri}(T))\right)\right]$$

where $\mathbf{ap}$ is the additive extension of application: $\mathbf{ap}\left(\sum_{i=1}^{n} \varepsilon^{k_i} s_i^{\mathbf{b}}, t^{\mathbf{b}}\right) := \sum_{i=1}^{n} \varepsilon^{k_i}(s_i^{\mathbf{b}}\ t^{\mathbf{b}})$ and the primal $\mathbf{pri}$ and tangent $\mathbf{tan}$ components of a canonical term T correspond respectively to the basic terms with zero and non-zero ε coefficients:

$$\begin{array}{rcl} \mathbf{pri}\,(0) & := & 0 \\ \mathbf{pri}\left(\varepsilon^{k+1} t^{\mathbf{b}} + T'\right) & := & \mathbf{pri}\,(T') \\ \mathbf{pri}\left(\varepsilon^{0} t^{\mathbf{b}} + T'\right) & := & t^{\mathbf{b}} + \mathbf{pri}\,(T') \end{array} \qquad \begin{array}{rcl} \mathbf{tan}\,(0) & := & 0 \\ \mathbf{tan}\left(\varepsilon^{k+1} t^{\mathbf{b}} + T'\right) & := & \varepsilon^{k} t^{\mathbf{b}} + \mathbf{tan}\,(T') \\ \mathbf{tan}\left(\varepsilon^{0} t^{\mathbf{b}} + T'\right) & := & \mathbf{tan}\,(T') \end{array}$$

▶ **Theorem 14.** Every unrestricted λ_ε-term is differentially equivalent to its canonical form $\mathbf{can}\,(t)$. That is to say, for all $t \in \Lambda_\varepsilon$ we have $t \sim_\varepsilon \mathbf{can}\,(t)$.

This canonicalization procedure is the result of orienting the equivalences in Definition 10. Note, however, that while most of these equivalences have a "natural" orientation to them, two of them are entirely symmetrical: those being commutativity of the sum and the derivative. Barring the imposition of some arbitrary total ordering on terms which would allow us to prefer the term $x+y$ over $y+x$ (or vice versa), we settle for our canonical forms to be unique "up to" these commutativity conditions.

▶ **Definition 15.** *Permutative equivalence* $\sim_+ \subseteq \Lambda_\varepsilon \times \Lambda_\varepsilon$ is the least equivalence relation which is contextual and satisfies the following properties:

$$\begin{array}{rcl} s + (t + e) & \sim_+ & (s + t) + e \\ s + t & \sim_+ & t + s \\ \mathrm{D}(\mathrm{D}(s) \cdot t) \cdot e & \sim_+ & \mathrm{D}(\mathrm{D}(s) \cdot e) \cdot t \end{array}$$

▶ **Theorem 16.** Given unrestricted terms $s, t \in \Lambda_\varepsilon$, they are differentially equivalent if and only if their canonical forms are permutatively equivalent. More succinctly, $s \sim_\varepsilon t$ if and only if $\mathbf{can}\,(s) \sim_+ \mathbf{can}\,(t)$

▶ **Corollary 17.** The set λ_ε of well-formed terms corresponds precisely to the set of canonical terms up to permutative equivalence $\mathrm{C}_\varepsilon/{\sim_+}$.

4.1 Substitution

As is usual, our calculus features two different kinds of application: standard function application, represented as $(s\ t)$; and differential application, represented as $\mathrm{D}(s) \cdot t$. These two give rise to two different notions of substitution. The first is, of course, the usual capture-avoiding substitution. The second, differential substitution, is similar to the equivalent notion in the differential λ-calculus, as it arises from the same chain rule that is satisfied in both Cartesian differential categories and change action models.

▶ **Definition 18.** Given terms $s, t \in \Lambda_\varepsilon$ and a variable x, the *capture-avoiding substitution* of s for x in t (which we write as $t\,[s/x]$) is defined by induction on the structure of t:

$$\begin{array}{rclll} x\,[s/x] & := & s & & \\ y\,[s/x] & := & y & \text{if } x \neq y \\ (\lambda y.t)\,[s/x] & := & \lambda y.(t\,[s/x]) & \text{if } y \notin \mathrm{FV}(s) \\ (t\ e)\,[s/x] & := & (t\,[s/x])(e\,[s/x]) & \end{array} \qquad \begin{array}{rcl} (\mathrm{D}(t) \cdot e)\,[s/x] & := & \mathrm{D}(t\,[s/x]) \cdot (e\,[s/x]) \\ (\varepsilon t)\,[s/x] & := & \varepsilon(t\,[s/x]) \\ (t + e)\,[s/x] & := & (t\,[s/x]) + (e\,[s/x]) \\ 0\,[s/x] & := & 0 \end{array}$$

▶ **Proposition 19.** Capture-avoiding substitution respects differential equivalence. That is to say, whenever $s \sim_\varepsilon s'$ and $t \sim_\varepsilon t'$, it is the case that $t\,[s/x] \sim_\varepsilon t'\,[s'/x]$.

▶ **Definition 20.** Given terms $s, t \in \Lambda_\varepsilon$ and a variable x *which is not free in* s the *differential substitution* of s for x in t, which we write as $\frac{\partial t}{\partial x}\,(s)$, is defined by induction on the structure of t:

$$
\begin{array}{rcll}
\frac{\partial x}{\partial x}\,(s) & := & s & \\
\frac{\partial y}{\partial x}\,(s) & := & 0 & \text{if } x \neq y \\
\frac{\partial (\lambda y.t)}{\partial x}\,(s) & := & \lambda y.\left(\frac{\partial t}{\partial x}\,(s)\right) & \text{if } y \notin \mathrm{FV}(s) \\
\frac{\partial (t\ e)}{\partial x}\,(s) & := & \left[\mathrm{D}(t) \cdot \left(\frac{\partial e}{\partial x}\,(s)\right)\ e\right] + \left[\frac{\partial t}{\partial x}\,(s)\ \ (e\,[(x+\varepsilon s)/x])\right] & \\
\frac{\partial (\mathrm{D}(t)\cdot e)}{\partial x}\,(s) & := & \mathrm{D}(t) \cdot \left(\frac{\partial e}{\partial x}\,(s)\right) + \mathrm{D}\left(\frac{\partial t}{\partial x}\,(s)\right) \cdot (e\,[(x+\varepsilon s)/x]) & \\
 & & \quad + \varepsilon \mathrm{D}(\mathrm{D}(t) \cdot e) \cdot \left(\frac{\partial e}{\partial x}\,(s)\right) & \\
\frac{\partial (\varepsilon t)}{\partial x}\,(s) & := & \varepsilon\left(\frac{\partial t}{\partial x}\,(s)\right) & \\
\frac{\partial (t+e)}{\partial x}\,(s) & := & \left(\frac{\partial t}{\partial x}\,(s)\right) + \left(\frac{\partial e}{\partial x}\,(s)\right) & \\
\frac{\partial 0}{\partial x}\,(s) & := & 0 &
\end{array}
$$

We write $\frac{\partial^k t}{\partial(x_1,\ldots,x_k)}(u_1,\ldots,u_k)$ to denote a sequence of nested differential substitutions.

Most of the cases of differential substitution are identical to those in the differential λ-calculus – our definition in fact coincides exactly with the original notion of differential substitution in e.g [12], provided that one assumes the identity $\varepsilon t = 0$ for all terms. This reflects the fact that every Cartesian differential category is in fact a Cartesian difference category with trivial infinitesimal extension.

All the differences in this definition stem from the failure of derivatives to be additive in the setting of Cartesian difference categories. Consider the case for $\frac{\partial \mathrm{D}(t)\cdot s}{\partial x}\,(e)$, and remember that the "essence" of a derivative in the setting of difference categories lies in $[\mathbf{C}\boldsymbol{\partial}\mathbf{.0}]$, that is to say, if $t(x)$ is a term with a free variable x, we seek our notion of differential substitution to satisfy a condition akin to Taylor's formula:

$$t(x+\varepsilon y) \sim_\varepsilon t(x) + \varepsilon \frac{\partial t}{\partial x}\,(y)$$

When the term t is a differential application, and assuming the above "Taylor's formula" holds for all of its subterms (which we will show later), this leads us to the following informal argument:

$$
\begin{aligned}
&\mathrm{D}(t(x+\varepsilon y)) \cdot (s(x+\varepsilon y)) \sim_\varepsilon \mathrm{D}\left(t(x) + \varepsilon \tfrac{\partial t}{\partial x}\,(y)\right) \cdot (s(x+\varepsilon y)) \\
&\quad \sim_\varepsilon \mathrm{D}(t(x)) \cdot (s(x)) + \varepsilon \mathrm{D}(t(x)) \cdot \left(\tfrac{\partial s}{\partial x}\,(y)\right) + \varepsilon \mathrm{D}\left(\tfrac{\partial t}{\partial x}\,(y)\right) \cdot (s(x+\varepsilon y)) \\
&\qquad\qquad + \varepsilon^2 \mathrm{D}(\mathrm{D}(t(x)) \cdot (s(x))) \cdot \left(\tfrac{\partial s}{\partial x}\,(y)\right)
\end{aligned}
$$

From this calculation, the differential substitution for this case arises naturally as it results from factoring out the ε and noticing that the resulting expression has precisely the correct shape to be Taylor's formula for the case of differential application. The case for standard application can be derived similarly, although the involved terms are simpler. Differential substitution verifies some useful properties, which we state below (mechanised proofs are available as part of the author's doctoral dissertation [1], although the details are more cumbersome than enlightening).

▶ **Proposition 21.** Differential substitution respects differential equivalence. That is to say, whenever $s \sim_\varepsilon s'$ and $t \sim_\varepsilon t'$, it is the case that $\frac{\partial t}{\partial x}(s) \sim_\varepsilon \frac{\partial t'}{\partial x}(s')$.

▶ **Proposition 22.** Whenever x is not free in t, then $\frac{\partial t}{\partial x}(u) \sim_\varepsilon 0$.

▶ **Proposition 23.** Whenever x is not free in u, v, then:

$$\frac{\partial^2 t}{\partial x^2}(u, v) \sim_\varepsilon \frac{\partial^2 t}{\partial x^2}(v, u)$$

As we have previously mentioned, the rationale behind our specific definition of differential substitution is that it should satisfy some sort of "Taylor's formula" (or rather, Kock-Lawvere formula), in the following sense:

▶ **Theorem 24.** For any unrestricted terms s, t, e and any variable x which does not appear free in e, we have

$$s\left[(t + \varepsilon e)/x\right] \sim_\varepsilon s\left[t/x\right] + \varepsilon\left(\left(\frac{\partial s}{\partial x}(e)\right)\left[t/x\right]\right)$$

We will often refer to the right-hand side of the above equivalence as the *Taylor expansion* of the corresponding term in the left-hand side.

One consequence of this "syntactic Taylor's formula" is that derivatives in the difference λ-calculus can be computed by a sort of quasi-automatic-differentiation algorithm: given an expression of the form $\lambda x.s$, its derivative at point t along u can be computed by reducing the differential application $(\mathrm{D}(\lambda x.s) \cdot (u))\ t$ which, as we shall see later, reduces (by definition) to $\left(\frac{\partial s}{\partial x}(u)\right)[t/x]$. Alternatively, we can simply evaluate $(\lambda x.s)\ (t + \varepsilon(u))$ to compute $s\left[t + \varepsilon(u)/x\right]$ which, by Theorem 24 is equivalent to $s\left[t/x\right] + \varepsilon\left(\left(\frac{\partial s}{\partial x}(u)\right)[t/x]\right)$. In an appropriate setting (i.e. one where subtraction of terms is allowed and ε admits an inverse) the derivative can then be extracted from this result by extracting the term under the ε. This process is remarkably similar to forward-mode automatic differentiation, where derivatives are computed by adding "perturbations" to the program input.

▶ **Theorem 25.** Differential substitution is regular, that is, for any unrestricted terms s, u, v where x does not appear free in either u or v, we have:

$$\frac{\partial s}{\partial x}(0) \sim_\varepsilon 0$$

$$\frac{\partial s}{\partial x}(u + v) \sim_\varepsilon \frac{\partial s}{\partial x}(u) + \left(\frac{\partial s}{\partial x}(v)\right)\left[x + \varepsilon u/x\right]$$

4.2 The Operational Semantics of λ_ε

With the substitution operations we have introduced so far, we can now proceed to give a small-step operational semantics as a reduction system.

▶ **Definition 26.** The *one-step reduction relation* [1] $\rightsquigarrow\ \subseteq \Lambda_\varepsilon \times \Lambda_\varepsilon$ is the least contextual relation that contains the following reduction rules:

$$\begin{array}{rcl} (\lambda x.t)\ s & \rightsquigarrow_\beta & t\left[s/x\right] \\ \mathrm{D}(\lambda x.t) \cdot s & \rightsquigarrow_\partial & \lambda x.\left(\frac{\partial t}{\partial x}(s)\right) \end{array}$$

[1] While the one-step reduction rules for λ_ε may seem identical to those in the differential λ-calculus (compare [12, Section 3]), they are in fact not equivalent, as our notions of differential substitution and term equivalence differ substantially.

We write $\rightsquigarrow^+$ to denote the transitive closure of $\rightsquigarrow$, and $\rightsquigarrow^*$ to denote its transitive, reflexive closure.

The previous one-step reduction is defined as a relation from unrestricted terms to unrestricted terms, but it is not compatible with differential equivalence. That is to say, there may be differentially equivalent terms $t \sim_\varepsilon t'$ such that t' can be reduced but t cannot. For example, consider the term $(\lambda x.x + 0)\ 0$, which contains no β-redexes that can be reduced. This term is, however, equivalent to $(\lambda x.x)\ 0$, which clearly reduces to 0. Fortunately the canonical form of a term t gives us a representative of $\underline{t}$ which is "maximally reducible", that is to say, whenever any representative of $\underline{t}$ can be reduced to a representative of some $\underline{t'}$, then any canonical form $\mathbf{can}\,(t)$ for t can be reduced to a representative of the same $\underline{t'}$, possibly in zero reduction steps.

▶ **Theorem 27.** Reduction is compatible with canonicalization. That is to say, if $s \rightsquigarrow s'$, then $\mathbf{can}\,(s) \rightsquigarrow^* s''$ for some $s'' \sim_\varepsilon s'$.

This result then legitimises our proposed "existential" definition of reduction of well-formed terms, as it shows that, in order to reduce a given term, it suffices to reduce its canonical form. It also gets rid of the "reducing zero" problem, as canonical forms do not contain "spurious" representations of zero.

▶ **Definition 28.** Given well-formed terms $\underline{s}, \underline{s'}$, we say that $\underline{s}$ *reduces to* $\underline{s'}$ *in one step*, and write $\underline{s}\ \underline{\rightsquigarrow}\ \underline{t}$, whenever $\mathbf{can}\,(s) \rightsquigarrow s''$ and $s'' \sim_\varepsilon s'$, for some canonical form $\mathbf{can}\,(s)$ of $\underline{s}$.

▶ **Proposition 29.** Whenever $\underline{s}\ \underline{\rightsquigarrow}\ \underline{s'}$ then for any term $\underline{t}$ we have $\underline{s+t}\ \underline{\rightsquigarrow}\ \underline{s'+t}$.

If $t = t^*$ is an additive term, then additionally $\underline{s\ t^*}\ \underline{\rightsquigarrow}^+\ \underline{s'\ t^*}$.

Furthermore, when $t = t^{\mathbf{b}}$ is a basic term (in particular $t^{\mathbf{b}}$ is not differentially equivalent to zero), we also have $\underline{\mathrm{D}(s) \cdot t}\ \underline{\rightsquigarrow}^+\ \underline{\mathrm{D}(s') \cdot t}$.

Conversely, whenever s is not differentially equivalent to zero and $\underline{t}\ \underline{\rightsquigarrow}\ \underline{t'}$, then $\underline{s\ t}\ \underline{\rightsquigarrow}^+\ \underline{s\ t'}$ and $\underline{\mathrm{D}(s) \cdot t}\ \underline{\rightsquigarrow}^+\ \underline{\mathrm{D}(s) \cdot t'}$.

A proof of confluence for λ_ε is sketched in Appendix A This proof follows the standard Tait/Martin-Löf method by introducing a notion of parallel reduction on terms which is shown to have the diamond property although, due to the nature of our setting, the diamond property only holds up to differential equivalence.

▶ **Corollary 30.** The reduction relation $\underline{\rightsquigarrow}$ is confluent.

4.3 Encoding the Differential λ-Calculus

It is immediately clear, from simply inspecting the operational semantics for λ_ε, that it is closely related to the differential λ-calculus – indeed, every Cartesian differential category is a Cartesian difference category, and this connection should also be reflected in the syntax.

As it turns out, there is a clean translation that embeds λ_ε into the differential λ-calculus, which proceeds by deleting every term that contains an ε. The intuition behind this scheme should be apparent: every single differential substitution rule in λ_ε is identical to the corresponding case for the differential λ-calculus, once all the ε terms are cancelled out.

▶ **Definition 31.** Given an unrestricted λ_ε term t, its *ε-erasure* is the differential λ-term $\lceil t \rceil$ defined according to the rules in Figure 2.

▶ **Proposition 32.** The erasure $\lceil t \rceil$ is invariant under equivalence. That is to say, whenever $t \sim_\varepsilon t'$, it is the case that $\lceil t \rceil = \lceil t' \rceil$.

$$
\begin{array}{rcl@{\qquad}rcl}
\lceil x \rceil & := & x & \lceil \varepsilon t \rceil & := & 0 \\
\lceil 0 \rceil & := & 0 & \lceil s\ t \rceil & := & \lceil s \rceil\ \lceil t \rceil \\
\lceil s + t \rceil & := & \lceil s \rceil + \lceil t \rceil & \lceil \mathrm{D}(s) \cdot t \rceil & := & \mathrm{D}(\lceil s \rceil) \cdot \lceil t \rceil
\end{array}
$$

Figure 2 ε-erasure of a term t.

▶ **Proposition 33.** Erasure is compatible with standard and differential substitution. That is to say, for any terms s, t and a variable x, we have $\lceil s\,[t/x] \rceil = \lceil s \rceil\,[\lceil t \rceil / x]$ and $\left\lceil \frac{\partial s}{\partial x}\,(t) \right\rceil = \frac{\partial \lceil s \rceil}{\partial x}\,(\lceil t \rceil)$

▶ **Corollary 34.** Whenever $s \rightsquigarrow s'$, then $\lceil s \rceil \rightsquigarrow^* \lceil s' \rceil$.

These results form the syntactic obverse to the purely semantic correspondence between differential and difference categories [2, Proposition 1]: the former exhibits the differential λ-calculus as an instance of λ_ε where the ε operator is "degenerate", whereas the later shows that every Cartesian differential category can be understood as a "degenerate" Cartesian difference category, in the same sense that the corresponding infinitesimal extension is just the zero map.

5 Simple Types for λ_ε

Much like the differential λ-calculus, λ_ε can be endowed with a system of simple types, built from a set of basic types using the usual function type constructor.

▶ **Definition 35.** The set of *types* and *contexts* of the λ_ε-calculus is given by the following inductive definition:

$$
\begin{array}{rrcl}
\text{Types:} & \sigma, \tau & := & \mathbf{t} \mid \sigma \Rightarrow \tau \\
\text{Contexts:} & \Gamma & := & \emptyset \mid \Gamma, x : \tau
\end{array}
$$

assuming a countably infinite set of basic types $\mathbf{t}, \mathbf{s} \ldots$ is given.

The typing rules for the λ_ε-calculus are given in Figure 3 below, and should not be in the least surprising, as they are identical to the typing rules for the differential λ-calculus, with the addition of a typing rule for the infinitesimal extension of a term. As one would expect, our type system enjoys all the "usual" structural properties and their proofs follow by straightforward induction on the typing derivation. Note, however, that all of these typing rules operate on unrestricted terms, rather than on well-formed terms, for reasons that we will clarify later.

$$
\begin{array}{ccc}
\dfrac{}{\Gamma, x : \tau \vdash x : \tau} & \dfrac{\Gamma \vdash s : \tau \Rightarrow \sigma \qquad \Gamma \vdash t : \tau}{\Gamma \vdash (s\ t) : \sigma} & \dfrac{\Gamma, x : \tau \vdash t : \sigma}{\Gamma \vdash \lambda x.t : \sigma \Rightarrow \tau} \\[2ex]
\dfrac{}{\Gamma \vdash 0 : \tau} & \dfrac{\Gamma \vdash s : \tau \qquad \Gamma \vdash t : \tau}{\Gamma \vdash s + t : \tau} & \dfrac{\Gamma \vdash t : \tau}{\Gamma \vdash \varepsilon t : \tau} \\[2ex]
 & \dfrac{\Gamma \vdash s : \tau \Rightarrow \sigma \qquad \Gamma \vdash t : \tau}{\Gamma \vdash \mathrm{D}(s) \cdot t : \tau \Rightarrow \sigma} &
\end{array}
$$

Figure 3 Simple types for λ_ε.

One property that fails to hold is uniqueness of typings: indeed the term 0 admits any type, as do terms such as $0+0$ or $(\lambda x.0)\ y$. Typing judgements are nonetheless invertible. The following "standard" properties also hold, and can be proven by straightforward induction on the relevant typing derivation.

▶ **Proposition 36** (Weakening)**.** Whenever $\Gamma \vdash t : \tau$, then for any context Σ which is disjoint with Γ it is also the case that $\Gamma, \Sigma \vdash t : \tau$.

▶ **Proposition 37** (Substitution)**.** Whenever $\Gamma, x : \tau \vdash s : \sigma$ and $\Gamma \vdash t : \tau$, we have:

(i) $\Gamma \vdash s\,[t/x] : \sigma$
(ii) $\Gamma, x : \tau \vdash \frac{\partial s}{\partial x}\,(t) : \sigma$

▶ **Theorem 38** (Subject reduction)**.** Whenever $\Gamma \vdash t : \tau$ and $t \rightsquigarrow t'$ then $\Gamma \vdash t' : \tau$.

Since we have defined well-formed terms as equivalence classes of unrestricted terms, we might ask if typing is compatible with this equivalence relation. The answer is unfortunately no, that is to say, there are ill-typed terms that are differentially equivalent to well-typed terms. In particular, the term $(0\ t)$ is differentially equivalent to the term 0, but while the later is trivially well-typed, the former will not be typable for many choices of t (for example, whenever $t = (x\ x)$). A weaker version of this property does hold, however, that makes use of canonicity.

▶ **Proposition 39.** Whenever $\Gamma \vdash t : \tau$, then $\Gamma \vdash \mathbf{can}\,(t) : \tau$, and furthermore whenever $\Gamma \vdash \mathbf{can}\,(t) : \tau$ then every canonical form of t admits the same type.

Before stating a progress theorem for λ_ε, we must point out one small subtlety, as the definition of reduction of unrestricted terms depends on the particular representation chosen for the term. For example, the terms $((\lambda x.x) + 0)\ 0$ and $(\lambda x.x)\ 0$ are equivalent, but the first one contains no β-redexes, whereas the second one reduces to 0 in one step. We can prove that progress holds for canonical terms, however, as those are "maximally reducible".

▶ **Definition 40.** A canonical term T is a *canonical value* whenever it is of the form

$$T = \sum_{i=1}^{i} \varepsilon^{k_i}(\lambda x_i.t_i)$$

▶ **Theorem 41** (Progress)**.** Whenever a canonical term T admits a typing derivation $\vdash T : \tau$, then either T is a canonical value or there is some term t' with $T \rightsquigarrow t'$.

▶ **Definition 42.** We extend typing judgements to well-formed terms by setting $\Gamma \underline{\vdash}\ \underline{t} : \tau$ whenever $\Gamma \vdash \mathbf{can}\,(t) : \tau$.

▶ **Corollary 43** (Subject reduction for well-formed terms)**.** Whenever $\Gamma \underline{\vdash}\ \underline{t} : \tau$ and $\underline{t} \mathrel{\underline{\rightsquigarrow}} \underline{t'}$, then $\Gamma \underline{\vdash}\ \underline{t'} : \tau$.

▶ **Corollary 44** (Progress for well-formed terms)**.** Whenever $\Gamma \underline{\vdash}\ \underline{t} : \tau$ then either $\underline{t} \mathrel{\underline{\rightsquigarrow}} \underline{t'}$ or every canonical form $\mathbf{can}\,(\underline{t})$ is a canonical value.

Finally, strong normalisation can be shown by a proof similar to Ehrhard and Regnier's [12] and Vaux's [17], which themselves proceed by an argument similar to the standard reducibility candidates method. We defer the details to Appendix B.

▶ **Theorem 45** (Strong normalisation)**.** Whenever a closed well-formed term is typable with type $\underline{\vdash}\ \underline{t} : \tau$, it is strongly normalising.

6 Semantics

It is a well-known result that the simply-typed differential λ-calculus can be soundly interpreted in any differential λ-category, that is to say, any Cartesian differential category where differentiation "commutes with" abstraction (in the sense of [9, Definition 4.4]).

The exact same result holds for the difference λ-calculus and difference λ-categories. In what follows we will consider a fixed difference λ-category $\mathbf{C}$, and proceed to define interpretations for the types, contexts and terms of the simply-typed λ_ε-calculus.

▶ **Definition 46.** Given a $\mathbf{t}$-indexed family of objects $O_{\mathbf{t}}$, we define the interpretation $\llbracket \tau \rrbracket$ of a type τ by induction on its structure by setting $\llbracket \mathbf{t} \rrbracket \coloneqq O_{\mathbf{t}}$, $\llbracket \sigma \Rightarrow \tau \rrbracket \coloneqq \llbracket \sigma \rrbracket \Rightarrow \llbracket \tau \rrbracket$. We lift the interpretation of types to contexts in the usual way. Or, more formally, we have: $\llbracket \cdot \rrbracket \coloneqq \mathbf{1}$, $\llbracket \Gamma, x : \tau \rrbracket \coloneqq \llbracket \Gamma \rrbracket \times \llbracket \tau \rrbracket$.

As is the case in differential λ-categories, we can define a "differential substitution" operator on the semantic side. This operator is akin to post-composition with a partial derivative, and can be defined as follows.

▶ **Definition 47.** Given morphisms $s : A \times B \to C, u : A \to B$, we define their *differential composition* $s \star u : A \times B \to C$ by $s \star u \coloneqq \partial\,[s] \circ \langle \mathbf{id}_{A \times B}, \langle 0_A, u \circ \pi_1 \rangle \rangle$

▶ **Definition 48.** Given a well-typed unrestricted λ_ε-term $\Gamma \vdash t : \tau$, we define its interpretation $\llbracket t \rrbracket : \llbracket \Gamma \rrbracket \to \llbracket \tau \rrbracket$ inductively as in Figure 4 below. When Γ and τ are irrelevant or can be inferred from the context, we will simply write $\llbracket t \rrbracket$.

$$\begin{array}{rcll}
\llbracket (x_i : \tau_i)_{i=1}^n \vdash x_k : \tau_k \rrbracket & \coloneqq & \pi_2 \circ \pi_1^{n-k} & : \prod_{i=1}^n \llbracket \tau_i \rrbracket \to \llbracket \tau_k \rrbracket \\
\llbracket \Gamma \vdash 0 : \tau \rrbracket & \coloneqq & 0 & : \llbracket \Gamma \rrbracket \to \llbracket \tau \rrbracket \\
\llbracket \Gamma \vdash s + t : \tau \rrbracket & \coloneqq & \llbracket s \rrbracket + \llbracket t \rrbracket & : \llbracket \Gamma \rrbracket \to \llbracket \tau \rrbracket \\
\llbracket \Gamma \vdash \varepsilon t : \tau \rrbracket & \coloneqq & \varepsilon\,\llbracket t \rrbracket & : \llbracket \Gamma \rrbracket \to \llbracket \tau \rrbracket \\
\llbracket \Gamma \vdash \lambda x.t : \sigma \Rightarrow \tau \rrbracket & \coloneqq & \Lambda\,\llbracket t \rrbracket & : \llbracket \Gamma \rrbracket \to \llbracket \sigma \rrbracket \Rightarrow \llbracket \tau \rrbracket \\
\llbracket \Gamma \vdash (s\ t) : \tau \rrbracket & \coloneqq & \mathbf{ev} \circ \langle \llbracket s \rrbracket, \llbracket t \rrbracket \rangle & : \llbracket \Gamma \rrbracket \to \llbracket \tau \rrbracket \\
\llbracket \Gamma \vdash \mathrm{D}(s) \cdot t : \sigma \Rightarrow \tau \rrbracket & \coloneqq & \Lambda\,(\Lambda^-(\llbracket s \rrbracket) \star \llbracket t \rrbracket) & : \llbracket \Gamma \rrbracket \to \llbracket \tau \rrbracket
\end{array}$$

Figure 4 Interpreting λ_ε in $\mathbf{C}$.

▶ **Theorem 49.** Whenever $s \sim_\varepsilon t$ are equivalent unrestricted terms that admit typing derivations $\Gamma \vdash s : \tau$ and $\Gamma \vdash t : \tau$, then their interpretations are identical, that is to say:

$$\llbracket \Gamma \vdash s : \tau \rrbracket = \llbracket \Gamma \vdash t : \tau \rrbracket$$

▶ **Definition 50.** Given well-formed terms $\underline{s}, \underline{s'}$, we define the equivalence relation $\sim_{\beta\partial}$ as the least contextual equivalence relation that contains the one-step reduction relation $\underline{\rightsquigarrow}$.

▶ **Theorem 51.** The interpretation $\llbracket \cdot \rrbracket$ is *sound*, that is to say, whenever $\underline{s} \sim_{\beta\partial} \underline{s'}$ then $\llbracket s \rrbracket = \llbracket s' \rrbracket$, independently of the choice of representatives s, s'.

Proof. Straightforward consequence of the results in Appendix C. ◀

▶ **Definition 52.** Recall that a simply-typed theory is a collection of equational judgements of the form $\Gamma \vdash s = t : \sigma$ where $\Gamma \vdash s : \sigma$ and $\Gamma \vdash t : \sigma$ are derivable. We say that a simply-typed theory is a *difference λ-theory* if it is closed under all rules in the system $\lambda_\varepsilon^\times \beta\eta\partial$ (comprising the contextual rules for all the constructs of the λ_ε-calculus augmented by products, $\sim_\varepsilon$ equivalence, and the surjective pairing, β, η and ∂ laws, and last being the equational version of $\rightsquigarrow_\partial$).

Given an interpretation $[\![\cdot]\!]_{\mathcal{M}}$ of λ_ε in $\mathbf{C}$, we say that $\mathcal{M} = [\![\cdot]\!]_{\mathcal{M}}$ is a *model* of a difference λ-theory $\mathscr{T}$ if for every typed equational judgement $\Gamma \vdash s = t : \sigma$ in $\mathscr{T}$, we have that $[\![\Gamma \vdash s : \sigma]\!]_{\mathcal{M}}$ and $[\![\Gamma \vdash t : \sigma]\!]_{\mathcal{M}}$ are the same morphism.

A *model homomorphism* $h : \mathcal{M} \to \mathcal{N}$ is given by isomorphisms $h_{\mathbf{t}} : [\![\mathbf{t}]\!]_{\mathcal{M}} \to [\![\mathbf{t}]\!]_{\mathcal{N}}$ for each basic type $\mathbf{t}$, and $h_{\sigma\times\tau} := h_\sigma \times h_\tau$, and

$$h_{\sigma\Rightarrow\tau} := h_\sigma^{-1} \Rightarrow h_\tau := \Lambda(h_\tau \circ \mathbf{ev} \circ (\mathbf{id} \times h_\sigma^{-1})).$$

We write $\mathsf{Mod}_{\mathsf{Dif}\lambda}(\mathscr{T}, \mathbf{C})$ for the category whose objects are all models of difference λ-theory $\mathscr{T}$ in a difference λ-category $\mathbf{C}$, and whose morphisms are model homomorphisms.

▶ **Definition 53.** Let $\mathbf{C}$ and $\mathbf{D}$ be difference λ-categories. We say that a functor $F : \mathbf{C} \to \mathbf{D}$ is a *difference λ-functor* if F preserves the following:

- additive structure: $F(f + g) = F(f) + F(g)$, and $F(0) = 0$
- infinitesimal extension: $F(\varepsilon(f)) = \varepsilon(F(f))$
- products via the isomorphism $\Phi := \langle F(\pi_1), F(\pi_2)\rangle$
- exponentials via the isomorphism $\Psi := \Lambda(F(\mathbf{ev}) \circ \Phi)$
- difference combinator: $F(\partial[f]) = \partial[F(f)] \circ \Phi$.

We write $\mathsf{Dif}\lambda\text{-}\mathsf{Func}(\mathbf{C}, \mathbf{D})$ for the category of difference λ-functors $\mathbf{C} \to \mathbf{D}$ and natural isomorphisms.

▶ **Definition 54.** Given a difference λ-theory $\mathscr{T}$, we say that a category, denoted $\mathbf{Cl}(\mathscr{T})$, is *classifying* if there is a model of the theory in $\mathbf{Cl}(\mathscr{T})$, and this model is "generic", meaning that for every differential λ-category $\mathbf{D}$, there is a natural equivalence

$$\mathsf{Dif}\lambda\text{-}\mathsf{Func}(\mathbf{Cl}(\mathscr{T}), \mathbf{D}) \simeq \mathsf{Mod}_{\mathsf{Dif}\lambda}(\mathscr{T}, \mathbf{D}). \tag{1}$$

The classifying category (unique up to isomomrphism) is the "smallest" in the sense that given a model of the theory $[\![\cdot]\!]_{\mathbf{D}}$ in a difference λ-category $\mathbf{D}$, there is a difference λ-functor $F : \mathbf{Cl}(\mathscr{T}) \to \mathbf{D}$ such that the interpretation $[\![\cdot]\!]_{\mathbf{D}}$ can be factored through the canonical interpretation in the classifying category, i.e., $[\![\cdot]\!]_{\mathbf{D}} = F \circ [\![\cdot]\!]_{\mathbf{Cl}(\mathscr{T})}$.

▶ **Conjecture 6.1** (Completeness). Every difference λ-theory $\mathscr{T}$ has a classifying difference λ-category $\mathbf{Cl}(\mathscr{T})$.

7 Future Work

We have defined here the difference λ-calculus, which generalises the differential λ-calculus in exactly the same manner as Cartesian difference categories generalise their differential counterpart. While this calculus is of theoretical interest, it lacks most practical features, such as iteration or conditionals, and it is not immediately obvious how to extend it with these. It is not clear, for example, precisely when iteration combinators are differentiable in the difference category sense.

The problem of iteration is closely related to integration, which is itself the focus of current work on the differential side [10, 15]. Indeed, consider a hypothetical extension of the difference λ-calculus equipped with a type of natural numbers (with the identity as its corresponding infinitesimal extension, that is to say, $\varepsilon_{\mathbb{N}} = \mathbf{id}_{\mathbb{N}}$). How should an iteration operator **iter** be defined? The straightforward option would be to give it the usual behavior, that is to say:

$$\begin{aligned} \mathbf{iter}\ \mathbf{Z}\ z\ s &\rightsquigarrow z \\ \mathbf{iter}\ (\mathbf{S}\ n)\ z\ s &\rightsquigarrow s\ (\mathbf{iter}\ n\ z\ s) \end{aligned}$$

These reduction rules entail that every object involved must be complete, that is to say, for every $s, t : A$, there is some $u : A$ with $s + \varepsilon(u) = t$ – such an element is given by the term $((\mathrm{D}(\lambda n.\mathbf{iter}\ n\ s\ (\lambda x.t)) \cdot (\mathbf{S}\ \mathbf{Z}))\ \mathbf{Z})$.

This would rule out a number of interesting models and so it seems unsatisfactory. An alternative is to define the iteration operator by:

$$\begin{aligned} \mathbf{iter}\ \mathbf{Z}\ z\ s &\rightsquigarrow z \\ \mathbf{iter}\ (\mathbf{S}\ n)\ z\ s &\rightsquigarrow (\mathbf{iter}\ n\ z\ s) + \varepsilon(s\ (\mathbf{iter}\ n\ z\ s)) \end{aligned}$$

Fixed z, s, and defining the map $\mu(n) \coloneqq \mathbf{iter}\ n\ z\ s$, its derivative $\mathrm{D}[\mu](n, \mathbf{S}\ \mathbf{Z})$ is precisely $s(\mu(n))$. Or, in other words, the function $\mu : \mathbb{N} \to A$ is a "curve" which starts at z and whose derivative at a given point n is $s(\mu(n))$ – this boils down to stating that the curve μ is an integral curve for the vector field s satisfying the initial condition $\mu(\mathbf{Z}) = z$! Hence it may be possible to understand iteration as a discrete counterpart of the Picard-Lindelöf theorem, which states that such integral curves always exist (locally).

It would be of great interest to extend λ_ε with an interation operator and give its semantics in terms of differential (or difference) equations. Studying recurrence equations using the language of differential equations is a very useful tool in discrete analysis; for example, one can treat the recursive definition of the Fibonacci sequence as a discrete ODE and use differential equation methods to find a closed-form solution. We believe that in a language which frames iteration in such terms may be amenable to optimisation by similar analytic methods.

References

1 Mario Alvarez-Picallo. Change actions: from incremental computation to discrete derivatives, 2020. `arXiv:2002.05256`.

2 Mario Alvarez-Picallo and Jean-Simon P. Lemay. Cartesian difference categories. In *International Conference on Foundations of Software Science and Computation Structures*, page to appear. Springer, 2020.

3 Mario Alvarez-Picallo and Jean-Simon Pacaud Lemay. Cartesian difference categories: Extended report, 2020. `arXiv:2002.01091`.

4 Mario Alvarez-Picallo and C.-H. Luke Ong. Change actions: models of generalised differentiation. In *International Conference on Foundations of Software Science and Computation Structures*, pages 45–61. Springer, 2019.

5 Mario Alvarez-Picallo, Michael Peyton-Jones, Alexander Eyers-Taylor, and C.-H. Luke Ong. Fixing incremental computation. In *European Symposium on Programming*. Springer, 2019. in press.

6 P Arrighi and G Dowek. Linear-algebraic lambda-calculus: higher-order, encodings and confluence in A. Voronkov, Rewriting techniques and applications. *Lecture Notes in Computer Science, Springer-Verlag*, 2008.

7 Pablo Arrighi and Gilles Dowek. A computational definition of the notion of vectorial space. *Electronic Notes in Theoretical Computer Science*, 117:249–261, 2005.

8 Richard F. Blute, J. Robin B. Cockett, and Robert A. G. Seely. Cartesian differential categories. *Theory and Applications of Categories*, 22(23):622–672, 2009.

9 Antonio Bucciarelli, Thomas Ehrhard, and Giulio Manzonetto. Categorical models for simply typed resource calculi. *Electronic Notes in Theoretical Computer Science*, 265:213–230, 2010.

10 J Robin B Cockett and J-S Lemay. Integral categories and calculus categories. *Mathematical Structures in Computer Science*, 29(2):243–308, 2019.

11 Thomas Ehrhard. An introduction to differential linear logic: proof-nets, models and antiderivatives. *Mathematical Structures in Computer Science*, 28(7):995–1060, 2018.

12 Thomas Ehrhard and Laurent Regnier. The differential lambda-calculus. *Theoretical Computer Science*, 309(1-3):1–41, 2003. `doi:10.1016/S0304-3975(03)00392-X`.

13 Thomas Ehrhard and Laurent Regnier. Uniformity and the Taylor expansion of ordinary lambda-terms. *Theoretical Computer Science*, 403(2):347–372, 2008.

14 René Lavendhomme. *Basic concepts of synthetic differential geometry*, volume 13. Springer Science & Business Media, 2013.

15 Jean-Simon Pacaud Lemay. Exponential functions in cartesian differential categories. *arXiv preprint arXiv:1911.04790*, 2019.

16 Giulio Manzonetto. What is a Categorical Model of the Differential and the Resource λ-Calculi? *Mathematical Structures in Computer Science*, 22(03):451–520, 2012.

17 Lionel Vaux. λ-calculus in an algebraic setting. *unpublished note*, 2006.

18 Lionel Vaux. The algebraic lambda calculus. *Mathematical Structures in Computer Science*, 19(05):1029–1059, 2009.

A Confluence

The following lemmas relate differential substitution and standard substitution, and will be of much use later.

▶ **Lemma 55.** Whenever x, y are (distinct) variables then for any unrestricted terms t, u, v where x is not free in v we have:

$$\left(\frac{\partial t}{\partial x}(u)\right)[v/y] = \frac{\partial t\,[v/y]}{\partial x}(u\,[v/y])$$

▶ **Lemma 56.** Whenever x, y are (distinct) variables, with y not free in either u, v, we have:

$$\frac{\partial t\,[v/y]}{\partial x}(u) \sim_\varepsilon \left(\frac{\partial t}{\partial x}(u)\right)[(v\,[x + \varepsilon u/x])/y] + \left(\frac{\partial t}{\partial y}\left(\frac{\partial v}{\partial x}(u)\right)\right)[v/y]$$

▶ **Definition 57.** The *parallel reduction* relation between (unrestricted) terms is defined according to the deduction rules in Figure 5.

$$(\rightsquigarrow_x)\ \dfrac{}{x \rightsquigarrow x} \qquad (\rightsquigarrow_0)\ \dfrac{}{0 \rightsquigarrow 0} \qquad (\rightsquigarrow_\lambda)\ \dfrac{t \rightsquigarrow t'}{\lambda x.t \rightsquigarrow \lambda x.t'}$$

$$(\rightsquigarrow_\varepsilon)\ \dfrac{t \rightsquigarrow t'}{\varepsilon t \rightsquigarrow \varepsilon t'} \qquad (\rightsquigarrow_+)\ \dfrac{s \rightsquigarrow s' \qquad t \rightsquigarrow t'}{s + t \rightsquigarrow s' + t'}$$

$$(\rightsquigarrow_{\mathbf{ap}})\ \dfrac{s \rightsquigarrow s' \qquad t \rightsquigarrow t'}{s\ t \rightsquigarrow s'\ t'} \qquad (\rightsquigarrow_{\mathrm{D}})\ \dfrac{s \rightsquigarrow s' \qquad t \rightsquigarrow t'}{\mathrm{D}(s) \cdot t \rightsquigarrow \mathrm{D}(s') \cdot t'}$$

$$(\rightsquigarrow_\beta)\ \dfrac{s \rightsquigarrow \lambda x.s' \qquad t \rightsquigarrow t'}{s\ t \rightsquigarrow s'\,[t'/x]} \qquad (\rightsquigarrow_\partial)\ \dfrac{s \rightsquigarrow \lambda x.s' \qquad t \rightsquigarrow t'}{\mathrm{D}(s) \cdot t \rightsquigarrow \lambda x.\frac{\partial s'}{\partial x}(t')}$$

Figure 5 Parallel reduction rules for λ_ε.

The parallel reduction relation can be extended to well-formed terms by setting $\underline{t} \rightrightarrows \underline{t'}$ whenever $\mathbf{can}\,(t) \rightrightarrows t''$ with $t'' \sim_\varepsilon t'$ for some canonical form of $\underline{t}$.

▶ Remark 58. Our definition of parallel reduction differs slightly from the usual in the rule $(\rightrightarrows_\beta)$, which allows reducing a newly-formed λ-abstraction. This is necessary because our calculus contains terms of the shape $(\mathrm{D}(\lambda x.s) \cdot u)\ t$, which we need to parallel reduce in a single step to $\left(\frac{\partial s}{\partial x}\,(u)\right)[t/x]$. The original presentation of the differential λ-calculus opted instead for adding an extra parallel reduction rule to allow for the case of reducing an abstraction under a differential application. Similarly, our rule $(\rightrightarrows_\partial)$ allows reducing terms of the form $\mathrm{D}(\mathrm{D}(\lambda x.s) \cdot u) \cdot v$ in a single step.

One convenient property of the parallel reduction relation lies in its relation to canonical forms. As we saw in Theorem 27, canonical forms are "maximally reducible", but don't respect the number of reduction steps. This is no longer the case for parallel reduction: the process of canonicalization only duplicates regexes "in parallel" (that is, by copying them onto multiple separate summands) or in a "parallelizable series" (i.e. a differential application may be regularized into a term of the form $\mathrm{D}(\mathrm{D}(\ldots) \cdot u) \cdot v$, which can be entirely reduced in a single parallel reduction step).

▶ **Theorem 59.** Whenever $s \rightrightarrows s'$, then $\mathbf{can}\,(s) \rightrightarrows s''$ for some $s'' \sim_\varepsilon s'$.

We also state the following standard properties of parallel reduction, all of which can be proven by straightforward induction on the term.

▶ **Lemma 60.** Parallel reduction sits between one-step and many-step reduction. That is to say: $\rightsquigarrow\ \subseteq\ \rightrightarrows\ \subseteq\ \rightsquigarrow^*$, and furthermore $\underline{\rightsquigarrow}\ \subseteq\ \underline{\rightrightarrows}\ \subseteq\ \underline{\rightsquigarrow}^*$.

▶ **Lemma 61.** The parallel reduction relation is contextual. In particular, every term parallel-reduces to itself.

▶ **Lemma 62.** Parallel reduction cannot introduce free variables. That is to say: whenever $t \rightrightarrows t'$, we have $\mathrm{FV}(t') \subseteq \mathrm{FV}(t)$.

▶ **Lemma 63.** Whenever $\lambda x.t \rightrightarrows u$, it must be the case that $u = \lambda x.t'$ and $t \rightrightarrows t'$.

▶ **Lemma 64.** Whenever $s \rightrightarrows s'$ and $t \rightrightarrows t'$ then $s\,[t/x] \rightrightarrows s'\,[t'/x]$, and furthermore there is some w with $\frac{\partial s}{\partial x}\,(t) \rightrightarrows w \sim_\varepsilon \frac{\partial s'}{\partial x}\,(t')$.

We first prove that parallel reduction has the diamond property when applied to canonical terms, taking care that it holds up to differential equivalence (note that, much like one-step reduction, the result of parallel-reducing a canonical term need not be canonical). For this, we introduce the usual notion of a full parallel reduct of a term.

▶ **Definition 65.** Given an unrestricted term t, its *full parallel reduct* $t_\downarrow$ is defined inductively by:

$$\begin{aligned}
x_\downarrow &:= x & (\lambda x.t)_\downarrow &:= \lambda x.(t_\downarrow)\\
(\varepsilon t)_\downarrow &:= \varepsilon(t_\downarrow) & (s\ t)_\downarrow &:= \begin{cases} e\,[t_\downarrow/x] & \text{if } s_\downarrow = \lambda x.e\\ (s_\downarrow)\ (t_\downarrow) & \text{otherwise}\end{cases}\\
(s+t)_\downarrow &:= (s_\downarrow) + (t_\downarrow) & & \\
0_\downarrow &:= 0 & (\mathrm{D}(s)\cdot t)_\downarrow &:= \begin{cases} \lambda x.\frac{\partial e}{\partial x}\,(t_\downarrow) & \text{if } s_\downarrow = \lambda x.e\\ D(s_\downarrow)\cdot(t_\downarrow) & \text{otherwise}\end{cases}
\end{aligned}$$

▶ **Lemma 66.** Whenever $s \rightrightarrows \lambda x.v$, then $s_\downarrow$ is of the form $\lambda x.w$, for some term w.

▶ **Theorem 67.** For any unrestricted terms s, s' such that $s \rightsquigarrow s'$, there is an unrestricted term w such that $s' \rightsquigarrow w$ and $w \sim_\varepsilon s_\downarrow$.

▶ **Corollary 68.** Parallel reduction has the diamond property up to differential equivalence. That is to say, for any unrestricted term t and terms t_1, t_2 such that $t \rightsquigarrow t_1$ and $t \rightsquigarrow t_2$, there are terms u, v making the following diagram commute:

$$\begin{tikzcd}
t_1 \arrow[d, squiggly] & T \arrow[l, squiggly] \arrow[r, squiggly] & t_2 \arrow[d, squiggly] \\
u \arrow[rr, phantom, "\sim_\varepsilon"] & & v
\end{tikzcd}$$

▶ **Lemma 69.** Given unrestricted terms $s \sim_+ s'$ which are permutatively equivalent, that is, which differ only up to a reordering of their additions and differential applications, their full parallel reducts are differentially equivalent.

▶ **Theorem 70.** The reduction relation $\underline{\rightsquigarrow}$ has the diamond property. That is, whenever $\underline{s} \mathrel{\underline{\rightsquigarrow}} \underline{u}$ and $\underline{s} \mathrel{\underline{\rightsquigarrow}} \underline{v}$ there is a term $\underline{c}$ such that $\underline{u} \mathrel{\underline{\rightsquigarrow}} \underline{c}$ and $\underline{v} \mathrel{\underline{\rightsquigarrow}} \underline{c}$.

Proof. Consider a well-formed term $\underline{s}$, and suppose that $\underline{s} \mathrel{\underline{\rightsquigarrow}} \underline{u}$ and $\underline{s} \mathrel{\underline{\rightsquigarrow}} \underline{v}$. In particular, this means there are two canonical forms $\mathbf{can}\,(s)_1, \mathbf{can}\,(s)_2$ of s such that $\mathbf{can}\,(s)_1 \rightsquigarrow u$ and $\mathbf{can}\,(s)_2 \rightsquigarrow v$. These canonical forms $\mathbf{can}\,(s)_1, \mathbf{can}\,(s)_2$ are equivalent up to permutative equivalence, and so their full parallel reducts are differentially equivalent as per Lemma 69. Denote their $\sim_\varepsilon$-equivalence class by $\underline{c}$. Therefore since $\mathbf{can}\,(s)_1 \rightsquigarrow \mathbf{can}\,(s)_{1\downarrow} \sim_\varepsilon c$ and $\mathbf{can}\,(s)_2 \rightsquigarrow \mathbf{can}\,(s)_{2\downarrow} \sim_\varepsilon c$ it follows that $\underline{u} \mathrel{\underline{\rightsquigarrow}} \underline{c}$ and $\underline{v} \mathrel{\underline{\rightsquigarrow}} \underline{c}$. ◀

B Strong Normalisation

With our typing rules in place, we set out to show that λ_ε is strongly normalising. Our proof follows the structure of Ehrhard and Regnier's [12] and Vaux's[17], which use an adaptation of the well-known argument by reducibility candidates. Our proof will be somewhat simpler, however, due to two main reasons: first, we are not concerning ourselves with terms with coefficients on some general rig; and second, we have defined unrestricted and canonical terms as inductive types, and so we can freely use induction on the syntax of our terms. We will need some auxiliary results, which we prove now.

▶ **Lemma 71.** A term $\underline{s + t}$ is strongly normalising if and only if $\underline{s}, \underline{t}$ are strongly normalising. A term $\underline{\varepsilon s}$ is strongly normalising if and only if $\underline{s}$ is.

▶ **Definition 72.** For every type τ we define inductively a set $\mathcal{R}_\tau$ of well-formed terms of type τ.

- Whenever $\tau = \mathbf{t}$ is a primitive type, $\underline{s} \in \mathcal{R}_\mathbf{t}$ if and only if $\underline{s}$ is strongly normalising.
- Whenever $\tau = \sigma_1 \Rightarrow \sigma_2$, $\underline{s} \in \mathcal{R}_{\sigma_1 \Rightarrow \sigma_2}$ if and only if for any additive term $\underline{t^*} \in \mathcal{R}_{\sigma_1}$ and for any sequence $\underline{v_1^\mathbf{b}}, \ldots, \underline{v_n^\mathbf{b}}$ of basic terms $\underline{v_i^\mathbf{b}} \in \mathcal{R}_{\sigma_1}$ of length $n \geq 0$ we have $\underline{\left(\mathrm{D}^n(s) \cdot (v_1^\mathbf{b}, \ldots, v_i^\mathbf{b})\right)\ t^*} \in \mathcal{R}_{\sigma_2}$

If $\underline{t} \in \mathcal{R}_\tau$ we will often just say that $\underline{t}$ is *reducible* if the choice of τ is clear from the context.

▶ **Lemma 73.** Whenever $\underline{t} \in \mathcal{R}_\tau$, then for any two distinct variables x, y the renaming $\underline{t\,[y/x]}$ is also in $\mathcal{R}_\tau$.

▶ **Lemma 74.** Whenever $\underline{t} \in \mathcal{R}_\tau$, then $\underline{t}$ is strongly normalising.

▶ **Lemma 75.** Whenever $\underline{s}, \underline{t} \in \mathcal{R}_\tau$, then both $\underline{s+t}, \underline{\varepsilon s}$ are in $\mathcal{R}_\tau$. Conversely, whenever $\underline{s+t}$ is in $\mathcal{R}_\tau$ then so are $\underline{s}, \underline{t}$.

▶ **Lemma 76.** Whenever $\underline{s} \in \mathcal{R}_{\sigma \Rightarrow \tau}$ and $\underline{t} \in \mathcal{R}_\sigma$ then $\underline{\mathrm{D}(s) \cdot t} \in \mathcal{R}_{\sigma \Rightarrow \tau}$.

▶ **Corollary 77.** A well-formed term $\underline{t}$ is in $\mathcal{R}_\tau$ if and only if some canonical form $T = \mathbf{can}\left(\underline{t}\right)$ is of the form $\sum_{i=1}^{n} \varepsilon^{k_i} t_i^{\mathbf{b}}$ with $\underline{t_i^{\mathbf{b}}} \in \mathcal{R}_\tau$ for each $1 \leq i \leq n$.

▶ **Lemma 78.** Whenever $\underline{t} \in \mathcal{R}_\tau, \underline{t}\ \underline{\rightsquigarrow}^+\ \underline{t'}$, then $\underline{t'} \in \mathcal{R}_\tau$.

▶ **Definition 79.** A basic term $t^{\mathbf{b}}$ is *neutral* whenever it is not a λ-abstraction. In other words, a basic term is neutral whenever it is of the form $x, (s\ t)$ or $\mathrm{D}(s) \cdot u$. A canonical term T is neutral whenever it is of the form $\sum_{i=1}^{n} \varepsilon^{k_i} s_i^{\mathbf{b}}$, where each of the $s_i^{\mathbf{b}}$ are neutral. In particular, 0 is a neutral term. A well-formed term $\underline{t}$ is neutral whenever some (equivalently, all) canonical form is neutral.

▶ **Lemma 80.** Whenever $\underline{t}$ is neutral and every $\underline{t'}$ such that $\underline{t}\ \underline{\rightsquigarrow}^+\ \underline{t'}$ is in $\mathcal{R}_\tau$, then so is $\underline{t}$.

▶ **Lemma 81.** If, for all $\underline{t^*} \in \mathcal{R}_{\sigma_1}$ where x does not appear free, the term $\underline{s\left[t^*/x\right]}$ is in $\mathcal{R}_{\sigma_2}$ and, for all $\underline{u^{\mathbf{b}}}$ where x does not appear free, the term $\underline{\left(\frac{\partial s}{\partial x}\left(u^{\mathbf{b}}\right)\right)\left[t^*/x\right]}$ is in $\mathcal{R}_{\sigma_2}$, then the term $\underline{\lambda x.s}$ is in $\mathcal{R}_{\sigma_1 \Rightarrow \sigma_2}$.

▶ **Theorem 82.** Consider a well-formed term $\underline{t}$ which admits a typing of the form $x_1 : \sigma_1, \ldots, x_n : \sigma_n \underline{\vdash}\ \underline{t} : \tau$ and assume given the following data:

- A sequence of basic terms $\underline{d_1^{\mathbf{b}}} \in \mathcal{R}_{\sigma_1}, \ldots, \underline{d_n^{\mathbf{b}}} \in \mathcal{R}_{\sigma_n}$.
- An arbitrary sequence of indices $i_1, \ldots, i_k \in \{1, \ldots, n\}$ (possibly with repetitions).
- A sequence of additive terms $\underline{s_1^*} \in \mathcal{R}_{\sigma_{i_1}}, \ldots, \underline{s_k^*} \in \mathcal{R}_{\sigma_{i_k}}$.

such that none of the variables $x_1, \ldots, x_i$ appear free in the $d_i^{\mathbf{b}}, s_i^*$. Then the term

$$\underline{t'} = \underline{\left(\frac{\partial^k t}{\partial(x_{i_1}, \ldots, x_{i_k})}(d_1^{\mathbf{b}}, \ldots, d_k^{\mathbf{b}})\right)\left[s_1^*, \ldots, s_n^*/x_1, \ldots, x_n\right]}$$

is in $\mathcal{R}_\tau$.

C Soundness

The following result corresponds to well-known properties of differential λ-categories (see e.g. [9, Lemma 4.8]), the proof being identical to the differential case (unlike some other lemmas in that work which hinge on derivatives being additive).

▶ **Lemma 83.** Let $f : A \to B, g : A \to C, h : (A \times B) \times E \to F$ be arbitrary $\mathbf{C}$-morphisms. Then the following properties hold:

i. $\partial\left[\mathbf{sw}\right] = \mathbf{sw} \circ \pi_2$
ii. $(g \circ \pi_1) \star f = 0$
iii. $\Lambda(h) \star f = \Lambda(((h \circ \mathbf{sw}) \star (f \circ \pi_1)) \circ \mathbf{sw})$
iv. $\Lambda^-\left(\Lambda(h) \star f\right) = ((h \circ \mathbf{sw}) \star (f \circ \pi_1)) \circ \mathbf{sw}$

The results below also have rough analogues in the theory of Cartesian differential categories, but the correspondence starts growing a bit more distant as any result that hinges on derivatives being additive will, in general, only hold up to some second-order term in the theory of difference categories.

▶ **Lemma 84.** Let $f : A \times B \times C \to D, g : A \to B, g' : A \times B \to B, e : A \times B \to C$ be arbitrary $\mathbf{C}$-morphisms. Then the following identities hold:

i. $(\mathbf{ev} \circ \langle \Lambda(f), e\rangle) \star g = \mathbf{ev} \circ \langle \Lambda(f \star (e \star g)), e\rangle + \mathbf{ev} \circ \langle \Lambda(f) \star g, e \circ \langle \pi_1, \pi_2 + \varepsilon(g) \circ \pi_1\rangle\rangle$
ii. $\Lambda(f \star e) \star g = \Lambda\Big[\Lambda^-(\Lambda(f) \star g) \star (e \circ (\mathbf{id} + \langle 0, \varepsilon(g)\rangle))) + \varepsilon(f \star e) \star (e \star g) + (f \star (e \star g))\Big]$
iii. $\Lambda(f \star e) \circ \langle \pi_1, g'\rangle = \Lambda(\Lambda^-(\Lambda(f) \circ \langle \pi_1, g'\rangle) \star (e \circ \langle \pi_1, g'\rangle))$

▶ **Lemma 85.** Let t be some unrestricted λ_ε-term. The following properties hold:

i. If $\Gamma \vdash t : \tau$ and x does not appear in Γ then $[\![\Gamma, x : \sigma \vdash t : \tau]\!] = [\![\Gamma \vdash t : \tau]\!] \circ \pi_1$
ii. If $\Gamma, x : \sigma_1, y : \sigma_2 \vdash t : \tau$ then $[\![\Gamma, y : \sigma_2, x : \sigma_1 \vdash t : \tau]\!] = [\![\Gamma, x : \sigma_1, y : \sigma_2 \vdash t : \tau]\!] \circ \mathbf{sw}$

The morphism $\mathbf{sw}$ above is the obvious isomorphism between $(A \times B) \times C$ and $(A \times C) \times B$, which we can define explicitly by $\mathbf{sw} := \langle\langle \pi_{11}, \pi_2\rangle, \pi_{21}\rangle : (A \times B) \times C \to (A \times C) \times B$

▶ **Lemma 86.** Let $\Gamma, x : \tau \vdash s : \sigma$, with s some unrestricted λ_ε-term. Then:

i. Whenever $\Gamma, x : \tau \vdash t : \tau$, then $[\![s\,[t/x]]\!]_\Gamma = [\![s]\!]_{\Gamma, x:\tau} \circ \left\langle \pi_1, [\![t]\!]_{\Gamma, x:\tau} \right\rangle$
ii. Whenever $\Gamma \vdash t : \tau$, then $\left[\!\!\left[\frac{\partial s}{\partial x}\,(t)\right]\!\!\right]_{\Gamma, x:\tau} = \partial\left[[\![s]\!]_{\Gamma, x:\tau}\right] \circ \langle \mathbf{id}, \langle 0, [\![t]\!]_\Gamma \circ \pi_1\rangle\rangle$. Or, using the notation in Definition 47, $\left[\!\!\left[\frac{\partial s}{\partial x}\,(t)\right]\!\!\right] = [\![s]\!] \star [\![t]\!]$.

D Completeness

We set up the equivalence (1) in the forward direction via modelling functors. Using the preceding notations, let $\mathcal{M} = [\![\cdot]\!]_{\mathcal{M}}$ be a model of a difference λ-theory $\mathscr{T}$ in $\mathbf{C}$, we define a family of *modelling functors* $\mathsf{mod}_{\mathcal{M}} : \mathsf{Dif\lambda\text{-}Func}(\mathbf{C}, \mathbf{D}) \to \mathsf{Mod}_{\mathsf{Dif\lambda}}(\mathscr{T}, \mathbf{D})$ by $[\![\mathbf{t}]\!]_{\mathsf{mod}_{\mathcal{M}} F} := F([\![\mathbf{t}]\!]_{\mathcal{M}})$ where $F : \mathbf{C} \to \mathbf{D}$ is a difference λ-functor; and for any natural isomorphism $\phi : F \to G$, $\mathsf{mod}_{\mathcal{M}}\,\phi : \mathsf{mod}_{\mathcal{M}} F \to \mathsf{mod}_{\mathcal{M}} G$ is a model homomorphism where $(\mathsf{mod}_{\mathcal{M}}\,\phi)_{\mathbf{t}} := \phi_{[\![\mathbf{t}]\!]_{\mathcal{M}}}$.

We first build a syntactic difference λ-category using the difference λ-theory $\mathscr{T}$, and then prove that it is classifying by presenting the "inverse" for the functor

$$\mathsf{mod}_{\mathcal{G}} : \mathsf{Dif\lambda\text{-}Func}(\mathbf{Cl}(\mathscr{T}), \mathbf{D}) \to \mathsf{Mod}_{\mathsf{Dif\lambda}}(\mathscr{T}, \mathbf{D}),$$

writing $\mathcal{G} = [\![\cdot]\!]_{\mathcal{G}}$ for the canonical "generic" interpretation in $\mathbf{Cl}(\mathscr{T})$.

We build the difference λ-category, $\mathbf{Cl}(\mathscr{T})$, using a standard construction. Objects are types of $\mathscr{T}$, and morphisms $\boldsymbol{f} : \sigma \to \tau$ are equivalence classes of term-in-context judgements $[x : \sigma \vdash M : \tau]$, where two terms are equivalent if they are provably equal in $\mathscr{T}$. It remains to show that $\mathbf{Cl}(\mathscr{T})$ is classifying by exhibiting the "inverse" of the modelling functors.

It is straightforward to see that the canonical interpretation $\mathcal{G}$, that sets $[\![\mathbf{t}]\!]_{\mathcal{G}} := \mathbf{t}$, is a model of $\mathscr{T}$ in $\mathbf{Cl}(\mathscr{T})$. Take a difference λ-category $\mathbf{D}$. We define

$$\mathsf{mod}_{\mathcal{G}}^{-1} : \mathsf{Mod}_{\mathsf{Dif\lambda}}(\mathscr{T}, \mathbf{D}) \to \mathsf{Dif\lambda\text{-}Func}(\mathbf{Cl}(\mathscr{T}), \mathbf{D})$$

as follows. Given a model $\mathcal{M}$ of $\mathscr{T}$ in $\mathbf{D}$, the functor $\mathsf{mod}_{\mathcal{G}}^{-1}\,\mathcal{M} : \mathbf{Cl}(\mathscr{T}) \to \mathbf{D}$ is defined by

$$\begin{aligned} \sigma &\mapsto [\![\sigma]\!]_{\mathcal{M}} \\ [x : \sigma \vdash M : \tau] &\mapsto [\![x : \sigma \vdash M : \tau]\!]_{\mathcal{M}} : [\![\sigma]\!]_{\mathcal{M}} \to [\![\tau]\!]_{\mathcal{M}} \end{aligned}$$

Soundness of the interpretations tells us that $\mathsf{mod}_{\mathcal{G}}^{-1}\,\mathcal{M}$ is functorial. Note that Φ and Ψ are both identity functors, and so, $\mathsf{mod}_{\mathcal{G}}^{-1}\,\mathcal{M}$ preserves products and exponentials. It remains to check that $\mathsf{mod}_{\mathcal{G}}^{-1}\,\mathcal{M}$ is a difference λ-functor.

Given a model homomorphism $h : \mathcal{M} \to \mathcal{N}$ in $\mathsf{Mod}_{\mathsf{Dif}\lambda}(\mathscr{T}, \mathbf{D})$, we define the natural isomorphism $\mathsf{mod}_{\mathcal{G}}^{-1}\, h : \mathsf{mod}_{\mathcal{G}}^{-1}\, \mathcal{M} \to \mathsf{mod}_{\mathcal{G}}^{-1}\, \mathcal{N}$ by setting

$$(\mathsf{mod}_{\mathcal{G}}^{-1}\, h)_{\sigma} \coloneqq h_{\mathbf{t}} : [\![\sigma]\!]_{\mathcal{M}} \to [\![\sigma]\!]_{\mathcal{N}} .$$

We can easily check that $(\mathsf{mod}_{\mathcal{G}}^{-1}\, h)_{\sigma}$ is a natural transformation by induction on the length of the derivation of the $\lambda_{\varepsilon}^{\times}$-term $\Gamma \vdash M : \tau$. Since h_{σ} is an isomorphism for any type σ, $\mathsf{mod}_{\mathcal{G}}^{-1}$ is a natural isomorphism.

We check that $(\mathsf{mod}_{\mathcal{G}}^{-1}\, h)_{\sigma}$ is a natural transformation by induction on the length of the derivation of the $\lambda_{\varepsilon}^{\times}$-term-in-context, $\Gamma \vdash M : \tau$, that the following diagram commutes:

$$[\![\Gamma \vdash M : \sigma]\!]_{\mathcal{N}} \circ h_{\Gamma} = h_{\tau} \circ [\![\Gamma \vdash M : \sigma]\!]_{\mathcal{M}} .$$

Lastly we check that $\mathsf{mod}_{\mathcal{G}}$ and $\mathsf{mod}_{\mathcal{G}}^{-1}$ define an equivalence via the natural isomorphisms

$$\begin{aligned} &\mu : \mathsf{mod}_{\mathcal{G}} \circ \mathsf{mod}_{\mathcal{G}}^{-1} \simeq \mathbf{id}_{\mathsf{Mod}_{\mathsf{Dif}\lambda}(\mathscr{T},\mathbf{D})} \\ &\nu : \mathbf{id}_{\mathsf{Mod}_{\mathsf{Dif}\lambda}(\mathscr{T},\mathbf{D})} \simeq \mathsf{mod}_{\mathcal{G}}^{-1} \circ \mathsf{mod}_{\mathcal{G}} \end{aligned}$$

defined as follows. For any model $\mathcal{M}$ of $\mathscr{T}$ in $\mathbf{D}$, $\mu_{\mathcal{M}} : \mathsf{mod}_{\mathcal{G}}(\mathsf{mod}_{\mathcal{G}}^{-1}\, \mathcal{M}) \to \mathcal{M}$ is defined by

$$(\mu_{\mathcal{M}})_{\mathbf{t}} \coloneqq \mathbf{id}_{[\![\mathbf{t}]\!]_{\mathcal{M}}} : [\![\mathbf{t}]\!]_{\mathsf{mod}_{\mathcal{G}}(\mathsf{mod}_{\mathcal{G}}^{-1}\, \mathcal{M})} = [\![\mathbf{t}]\!]_{\mathcal{M}} \to [\![\mathbf{t}]\!]_{\mathcal{M}}$$

and for any difference λ-functor $F : \mathbf{Cl}(\mathscr{T}) \to \mathbf{D}$, we define

$$(\nu_F)_{\sigma} \coloneqq \mathbf{id}_{F\sigma} : F\,\sigma \to (\mathsf{mod}_{\mathcal{G}}^{-1}(\mathsf{mod}_{\mathcal{G}}\, F))\sigma = F([\![\sigma]\!]_{\mathcal{G}}) = F\,\sigma.$$

Obviously μ and ν are natural isomorphisms. Thus $\mathbf{Cl}(\mathscr{T})$ is a classifying category with the model $\mathcal{G}$.

Rast: Resource-Aware Session Types with Arithmetic Refinements

Ankush Das
Carnegie Mellon University, Pittsburgh, PA, USA
https://www.cs.cmu.edu/~ankushd/
ankushd@cs.cmu.edu

Frank Pfenning
Carnegie Mellon University, Pittsburgh, PA, USA
https://www.cs.cmu.edu/~fp/
fp@cs.cmu.edu

Abstract

Traditional session types prescribe bidirectional communication protocols for concurrent computations, where well-typed programs are guaranteed to adhere to the protocols. Recent work has extended session types with refinements from linear arithmetic, capturing intrinsic properties of processes and data. These refinements then play a central role in describing sequential and parallel complexity bounds on session-typed programs.

The Rast language and system provide an open-source implementation of session-typed concurrent programs extended with arithmetic refinements as well as ergometric and temporal types to capture work and span of program execution. Type checking relies on Cooper's algorithm for quantifier elimination in Presburger arithmetic with a few significant optimizations, and a heuristic extension to nonlinear constraints. Rast furthermore includes a reconstruction engine so that most program constructs pertaining the layers of refinements and resources are inserted automatically. We provide a variety of examples to demonstrate the expressivity of the language.

2012 ACM Subject Classification Theory of computation → Process calculi; Theory of computation → Linear logic; Theory of computation → Logic and verification; Computing methodologies → Concurrent programming languages; Theory of computation → Type theory

Keywords and phrases Session Types, Resource Analysis, Refinement Types

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.33

Category System Description

Supplementary Material https://bitbucket.org/fpfenning/rast/src/master/rast/

Funding This material is based upon work supported by the National Science Foundation under Grant No. 1718276.

Acknowledgements We would like to thank Farzaneh Derakhshan for contributions to the implementation and the anonymous reviewers for suggestions on an earlier version of this paper.

1 Introduction

Session types [13, 14, 17] provide a structured way of statically prescribing communication protocols in message-passing programs. In this system description we introduce the Rast programming language and implementation which is based on *binary session types* governing the interaction of two processes along a single channel, rather than *multi-party session types* [15] which take a more global view of computation. Nevertheless, during the execution of a Rast program complex networks of interacting processes arise. Recent work has placed binary session types without general recursion on a strong logical foundation by exhibiting a Curry-Howard isomorphism with linear logic [1, 18, 2]. Moreover, the cut reduction properties of linear logic entail type safety of session typed processes and guarantee *freedom from deadlocks* (global progress) and *session fidelity* (type preservation) ensuring adherence to the communication protocols at runtime.

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 33; pp. 33:1–33:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The Rast programming language is based on session types derived from intuitionistic linear logic, extended with equirecursive types and recursive process definitions. It furthermore supports arithmetic type refinements as well as ergometric and temporal types to measure the total work and span of Rast programs. The repository also contains a number of illustrative examples that highlight various language features, some of which we briefly sketch in this system description. The theory underlying Rast has been developed in several papers, starting with the Curry-Howard interpretation of linear logic as session-typed processes [1, 2], the treatment of general equirecursive types and type equality [10], asynchronous communication [11, 9], ergometric types [6], temporal types [5], indexed types and indexed type equality [12, 7].

We begin with motivation and a brief overview of the main features of the language using a concurrent queue data structure as a running example. The following type specifies the interface to a queue server in the system of basic recursive session types supporting the operations of insert (enqueue) and delete (dequeue).

$$\begin{aligned} \mathsf{queue}_A = \&\{&\mathbf{ins} : A \multimap \mathsf{queue}_A, \\ &\mathbf{del} : \oplus\{\mathbf{none} : \mathbf{1}, \\ &\qquad\qquad \mathbf{some} : A \otimes \mathsf{queue}_A\}\} \end{aligned}$$

The *external choice* operator & dictates that the process providing this data structure accepts either one of two messages: the labels **ins** or **del**. In the case of **ins**, it receives an element of type A denoted by the $\multimap$ operator, and then the type recurses back to queue_A. On receiving a **del** request, the process can respond with one of two labels (**none** or **some**), indicated by the *internal choice* operator $\oplus$. If the queue is empty, it responds with **none** and then *terminates* (indicated by **1**). If the queue is nonempty, it responds with **some** followed by the element of type A (expressed with the $\otimes$ operator) and recurses. However, the simple session type does not express the conditions under which the **none** and **some** branches must be chosen, which requires tracking the length of the queue.

Rast extends session types with arithmetic refinements [7] which can be used to express the length of a queue. The more precise type

$$\begin{aligned} \mathsf{queue}_A[n] = \&\{&\mathbf{ins} : A \multimap \mathsf{queue}_A[n+1], \\ &\mathbf{del} : \oplus\{\mathbf{none} : ?\{n = 0\}.\, \mathbf{1}, \\ &\qquad\qquad \mathbf{some} : ?\{n > 0\}.\, A \otimes \mathsf{queue}_A[n-1]\}\} \end{aligned}$$

uses the index refinement n to indicate the number of elements in the queue. In addition, the *type constraint* $?\{\phi\}.\, A$ read as "*there exists a proof of* ϕ" is analogous to the *assertion* of ϕ in imperative languages. Conceptually, the process providing the queue must provide a proof of $n = 0$ after sending **none**, and a proof of $n > 0$ after sending **some** respectively. It is therefore constrained in its choice between the two branches based on the value of the index n. Since the constraint domain is decidable and the actual form of a proof is irrelevant to the outcome of a computation, in the implementation no proof is actually sent.

As is standard in session types, the dual constraint to $?\{\phi\}.\, A$ is $!\{\phi\}.\, A$ (*for all proofs of* ϕ, analogous to the *assumption* of ϕ). We also add explicit quantifiers $\exists n.\, A$ and $\forall n.\, A$ that send and receive natural numbers, respectively.

Arithmetic refinements are instrumental in expressing *sequential* and *parallel complexity bounds.* These are captured with ergometric [6, 4] and temporal session types [5]. They rely on index refinements to express, for example, the size of lists, stacks, and queue data structures, or the height of trees and express work and time bounds as a function of these indices. Rast largely follows and extends prior work on session types with arithmetic refinements [7].

Revisiting the queue example, consider an implementation where each element in the queue corresponds to a process. Then insertion acts like a bucket brigade, passing the new element one by one to the end of the queue. Among multiple cost models provided by Rast is one where each *send* operation requires 1 unit of work (erg). In this cost model, such a bucket brigade requires $2n$ ergs because each process has to send **ins** and then the new element. On the other hand, responding to the **del** request requires only 2 ergs: we respond with **none** and close the channel, or **some** followed by the element. This gives us the following type

$$\begin{array}{l} \mathsf{queue}_A[n] = \mathbin{\&}\{\mathbf{ins} : \triangleleft^{2n}(A \multimap \mathsf{queue}_A[n+1]), \\ \qquad\qquad\quad \mathbf{del} : \triangleleft^{2} \oplus \{\mathbf{none} : ?\{n = 0\}.\,\mathbf{1}, \\ \qquad\qquad\qquad\qquad\qquad \mathbf{some} : ?\{n > 0\}.\,A \otimes \mathsf{queue}_A[n-1]\}\} \end{array}$$

which expresses that the client has to send $2n$ ergs to insert an element ($\triangleleft^{2n}$), and 2 ergs to delete an element ($\triangleleft^{2}$). The ergometric type system (described in Section 4) verifies this work bound using the potential operators as described in the type.

Temporal session types [5] capture the time complexity of session-typed programs assuming maximal parallelism on unboundedly many processors, often called the *span*. How does this work out in our example? We adopt a cost model where each send and receive action takes one unit of time (tick). First, we note that a use of a queue is at the client's discretion, so should be available at *any* point in the future, expressed by the type constructor $\Box$. Secondly, the queue does not interact at all with the elements it contains, so they have to be of type $\Box A$ for an arbitrary A. Since each interaction takes 1 tick, the next interaction requires at least 1 tick to elapse, captured by the next-time operator $\bigcirc$. During insertion, we need more time than this: a process needs 2 ticks to pass the element down the queue, so it takes 3 ticks overall until it can receive the next insert or delete request after an insertion. This reasoning yields the following temporal type:

$$\begin{array}{l} \mathsf{queue}_A[n] = \Box \mathbin{\&} \{\mathbf{ins} : \bigcirc(\Box A \multimap \bigcirc^{3}\mathsf{queue}_A[n+1]), \\ \qquad\qquad\qquad \mathbf{del} : \bigcirc \oplus \{\mathbf{none} : \bigcirc\ ?\{n = 0\}.\,\mathbf{1}, \\ \qquad\qquad\qquad\qquad\qquad\quad \mathbf{some} : \bigcirc\ ?\{n > 0\}.\,\Box A \otimes \bigcirc\mathsf{queue}_A[n-1]\}\} \end{array}$$

We see that even though the bucket brigade requires much work for every insertion (linear in the length of the queue), it has a lot of parallelism because there are only a constant number of required delays between consecutive insertions or deletions.

Rast follows the design principle that bases an *explicit language* directly on the correspondence with the sequent calculus for the underlying logic (such as linear logic, or temporal or ergometric linear logic), extended with recursively defined types and processes. Programming in this fully explicit form tends to be unnecessarily verbose, so Rast also provides an *implicit language* in which most constructs related to index refinements and amortized work analysis are omitted. Explicit programs are then recovered by a proof-theoretically motivated algorithm for *reconstruction* which is sound and complete on valid implicit programs.

Rast is implemented in SML, and allows the user to choose explicit or implicit syntax and the exact cost models for work and time analysis. The implementation consists of a lexer, parser, type checker, reconstruction engines, and an interpreter, with particular attention to providing precise error messages.

To summarize, our implementation makes the following contributions.

(i) A session-typed programming language with arithmetic refinements applied to ergometric and temporal types for parallel complexity analysis.

(ii) A type equality algorithm that works well in practice despite its theoretical undecidability [7] and uses Cooper's algorithm [3] with some small improvements to decide constraints in Presburger arithmetic (and heuristics for nonlinear constraints).

```
type queue{n} = &{ins : A -o queue{n+1},
                  del : +{none : ?{n = 0}. 1,
                          some : ?{n > 0}. A * queue{n-1}}}
decl empty : . |- (q : queue{0})
decl elem{n} : (x : A) (t : queue{n}) |- (q : queue{n+1})

proc q <- empty =
  case q (                      % receive a label along q
    ins => x <- recv q ;        % if ’ins’ receive a channel x along q
           e <- empty ;         % spawn a new empty process
           q <- elem{0} x e     % continue as an elem holding x
  | del => q.none ;             % if ’del’, respond with label ’none’
           assert q {0=0} ;     % assert that (n = 0)[0/n]
           close q )            % terminate by closing q

proc q <- elem{n} x t =
  case q (                      % receive a label along q
    ins => y <- recv q ;        % if ’ins’ receive a channel y along q
           t.ins ;              % send label ’ins’ along t
           send t y ;           % send the channel y along t
           q <- elem{n+1} x t   % recurse
  | del => q.some ;             % if ’del’, respond with label ’some’
           assert q {n+1>0} ;   % assert that (n > 0)[n+1/n]
           send q x ;           % send x along q
           q <-> t )            % identify q with t and terminate
```

Listing 1 Declaration and definition of queue processes, file `examples/list.rast`

(iii) A type checking algorithm that is sound and complete relative to type equality.

(iv) A sound and complete reconstruction algorithm for a process language where most index and ergometric constructs remain implicit.

(v) An interpreter for executing session-typed programs using the recently proposed shared memory semantics [16].

2 Example: An Implementation of Queues

We use the implementation of queues as sketched in the introduction as a first example program, starting with the indexed version. The concrete syntax of types is a straightforward rendering of their abstract syntax (Table 3), except that all arithmetic expressions are enclosed in braces to make them visually easily discernible.

```
type queue{n} = &{ins : A -o queue{n+1},
                  del : +{none : ?{n = 0}. 1,
                          some : ?{n > 0}. A * queue{n-1}}}
```

Each channel has exactly two endpoints: a *provider* and a *client*. Session fidelity ensures that provider and client always agree on the type of the channel and carry out complementary actions. The type of the channel evolves during communication, since it has to track where the processes are in the protocol as they exchange messages.

In our example, we need two kinds of processes: an *empty* process at the end of the queue, and an *elem* process that holds an element x. The empty process *provides* an empty queue, that is, a service of type `queue{0}` along a channel named `q`. It does not *use* any other services (indicated by ’`.`’), so its type is declared with

```
decl empty : . |- (q : queue{0})
```

An *elem* process *provides* a service of type `queue{n+1}` along a channel named `q` and *uses* a queue of type `queue{n}` along a channel named `t`. In addition, it holds (“owns”) an element `x` of type `A`.

```
decl elem{n} : (x : A) (t : queue{n}) |- (q : queue{n+1})
```

The turnstile ‘`|-`’ separates the channels used from the channel that is provided (which is always exactly one, roughly analogous to a value returned by a function). The notation `elem{n}` indicates that the natural number `n` is a parameter of this process.

Listing 1 shows the implementation of the two forms of processes in Rast. Comments, starting with a `%` character and extending to the end of the line, provide a brief explanation for the actions of each line of code. This code is in *explicit* form and contains two instances of `assert` to match the constraints `?{n = 0}` and `?{n > 0}` in the two possible responses to a delete request. These two lines would be omitted in implicit form since they can be read off the type at the corresponding place in the protocol. Of course, the type checker verifies that the assertion is justified and fails with an error message if it is not, whether the construct is explicit or implicit.

3 Basic and Refined Session Types

We present the basic system of session types and its arithmetic refinement, postponing ergometric and temporal types to Section 4.

$$\begin{array}{llll}
\text{Types} & A & ::= & \oplus\{\ell : A\}_{\ell \in L} \mid \&\{\ell : A\}_{\ell \in L} \mid A \otimes A \mid A \multimap A \mid \mathbf{1} \mid V\overline{[e]} \\
 & & \mid & ?\{\phi\}.\, A \mid !\{\phi\}.\, A \mid \exists n.\, A \mid \forall n.\, A \\
\text{Arith. Exps.} & e & ::= & i \mid e + e \mid e - e \mid e \times e \mid n \\
\text{Arith. Props.} & \phi & ::= & e < e \mid e \leq e \mid e = e \mid e \geq e \mid e > e \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg \phi \mid \phi \supset \phi
\end{array}$$

Here, i stands for a natural number, n for an arithmetic variable, L for a finite set of labels, V for a type identifier, and $\overline{[e]}$ for a sequence of arithmetic expressions. Arithmetic propositions could contain quantifiers, but at present the implementation only supports them at the level of types. Arithmetic expressions may be nonlinear, although a definitive outcome of type-checking is only guaranteed if they are lie within Presburger arithmetic.

Our implementation does not support type polymorphism which is convenient in some of the examples. We therefore allow definitions such as $\mathtt{queue}_A[n] = \ldots$ and interpret them as a family of definitions, one for each possible type A.

We review a few basic session type operators before introducing the quantified type constructors. Table 1 overviews the session types with their continuations, their associated process terms and operational description.

The complete typing judgment for process expressions has the form of a sequent

$$\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta \vdash^q_\Sigma P :: (x : A)$$

where $\mathcal{V}$ are index variables n, $\mathcal{C}$ are constraints over these variables expressed as a single proposition, Δ are the linear antecedents $x_i : A_i$, P is a process expression, and $x : A$ is the linear succedent. The *potential* q is explained in Section 4. We propose and maintain that the x_i and x are all distinct, and that all free index variables in $\mathcal{C}$, Δ, P, and A are contained among $\mathcal{V}$. Finally, Σ is a fixed signature containing type and process definitions (explained

Table 1 Basic session types with operational description.

Type	Cont.	Process Term	Cont.	Description
$c : \oplus\{\ell : A_\ell\}_{\ell \in L}$	$c : A_k$	$c.k \; ; \; P$	P	provider sends label k along c
		$\mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L}$	Q_k	client receives label k along c
$c : \&\{\ell : A_\ell\}_{\ell \in L}$	$c : A_k$	$\mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell \in L}$	P_k	provider receives label k along c
		$c.k \; ; \; Q$	Q	client sends label k along c
$c : A \otimes B$	$c : B$	$\mathsf{send}\ c\ w \; ; \; P$	P	provider sends chan. $w : A$ along c
		$y \leftarrow \mathsf{recv}\ c \; ; \; Q$	$Q[w/y]$	client receives chan. $w : A$ along c
$c : A \multimap B$	$c : B$	$y \leftarrow \mathsf{recv}\ c \; ; \; P$	$P[w/y]$	provider receives $w : A$ along c
		$\mathsf{send}\ c\ w \; ; \; Q$	Q	client sends $w : A$ along c
$c : \mathbf{1}$	—	$\mathsf{close}\ c$	—	provider sends *close* along c
		$\mathsf{wait}\ c \; ; \; Q$	Q	client receives *close* along c

in Section 3.1). Because it is fixed, we elide it from the presentation of the rules. In addition we write $\mathcal{V} \; ; \; \mathcal{C} \vDash \phi$ for semantic entailment (ϕ is true assuming $\mathcal{C}$) in the constraint domain where $\mathcal{V}$ contains all arithmetic variables in $\mathcal{C}$ and ϕ.

3.1 Basic Session Types

External and Internal Choice

The *external choice* type constructor $\&\{\ell : A_\ell\}_{\ell \in L}$ is an n-ary labeled generalization of the additive conjunction $A \,\&\, B$. Operationally, it requires the provider of $x : \&\{\ell : A_\ell\}_{\ell \in L}$ to branch based on the label $k \in L$ it receives from the client and continue to provide type A_k. The corresponding process term is written as $\mathsf{case}\ x\ (\ell \Rightarrow P)_{\ell \in L}$. Dually, the client must send one of the labels $k \in L$ using the process term $(x.k \; ; \; Q)$ where Q is the continuation. The *internal choice* constructor $\oplus\{\ell : A_\ell\}_{\ell \in L}$ is the dual of external choice requiring the provider to send one of the labels $k \in L$ that the client must branch on.

Channel Passing

The *tensor* operator $A \otimes B$ prescribes that the provider of $x : A \otimes B$ sends a channel w of type A and continues to provide type B. The corresponding process term is $\mathsf{send}\ x\ w \; ; \; P$ where P is the continuation. Correspondingly, its client must receive a channel using the term $y \leftarrow \mathsf{recv}\ x \; ; \; Q$, binding it to variable y and continuing to execute Q. The dual operator $A \multimap B$ allows the provider to receive a channel of type A and continue to provide type B. Finally, the type $\mathbf{1}$ indicates *termination*, operationally denoting that the provider sends a *close* message and terminates the communication.

A process $x \leftrightarrow y$ identifies the channels x and y so that any further communication along either x or y will be along the unified channel. Its typing rule corresponds to the logical rule of identity. Operationally, we refer to it as *forwarding*.

Table 2 Refined session types with operational description.

Type	Cont.	Process Term	Cont.	Description
$c : \exists n.\, A$	$c : A[i/n]$	$\mathsf{send}\ c\ \{e\}\ ;\ P$	P	provider sends the value i of e along c
		$\{n\} \leftarrow \mathsf{recv}\ c\ ;\ Q$	$Q[i/n]$	client receives number i along c
$c : \forall n.\, A$	$c : A[i/n]$	$\{n\} \leftarrow \mathsf{recv}\ c\ ;\ P$	$P[i/n]$	provider receives number i along c
		$\mathsf{send}\ c\ \{e\}\ ;\ Q$	Q	client sends value i of e along c
$c : ?\{\phi\}.\, A$	$c : A$	$\mathsf{assert}\ c\ \{\phi\}\ ;\ P$	P	provider asserts ϕ on channel c
		$\mathsf{assume}\ c\ \{\phi\}\ ;\ Q$	Q	client assumes ϕ on c
$c : !\{\phi\}.\, A$	$c : A$	$\mathsf{assume}\ c\ \{\phi\}\ ;\ P$	P	provider assumes ϕ on channel c
		$\mathsf{assert}\ c\ \{\phi\}\ ;\ Q$	Q	client asserts ϕ on c

Process and Type Definitions

Process definitions (possibly mutually recursive) have the form $\Delta \overset{q}{\vDash} f[\overline{n}] = P :: (x : A)$ where f is the name of the process and P its definition. In addition, $\overline{n}$ is a sequence of arithmetic variables that Δ, q, P, and A can refer to. Note that in the implementation a typed definition is split up into a declaration and a simple definition

```
decl f{n1}...{nk} : (x1 : A1) ... (xm : Am) |- (x : A)
proc x <- f{n1}...{nk} x1 ... xm = P
```

A new instance of a defined process f can be spawned with the expression $x \leftarrow f[\overline{e}]\ \overline{y}\ ;\ Q$ where $\overline{y}$ is a sequence of channels matching the antecedents Δ and $[\overline{e}]$ is a sequence of arithmetic expressions matching the variables $[\overline{n}]$. The newly spawned process will use all variables in $\overline{y}$ and provide x to the continuation Q. The declaration of f is looked up in the signature Σ, and $\overline{e}$ is substituted for $\overline{n}$ and $\overline{y}$ for Δ. Sometimes a process invocation is a tail call, written without a continuation as $x \leftarrow f[\overline{e}]\ \overline{y}$.

We allow (possibly mutually recursive) type definitions $V[\overline{n}] = A$, or, in concrete syntax

```
type v{n1}...{nk} = A
```

in the signature Σ. Here, $[\overline{n}]$ again denotes a sequence of arithmetic variables. We also require A to be *contractive* [10] meaning A should not itself be a type name. Our type definitions are *equirecursive* so we can silently replace type names $V[\overline{e}]$ indexed with arithmetic refinements by $A[\overline{e}/\overline{n}]$ during type checking.

All types in a signature must be *valid* which requires that all free arithmetic variables of $\mathcal{C}$ and A are contained in $\mathcal{V}$, and that for each arithmetic expression e in A we can prove $\mathcal{V}'\ ;\ \mathcal{C}' \vdash e : \mathsf{nat}$ for the constraints $\mathcal{C}'$ known at the occurrence of e (implying $e \geq 0$).

3.2 The Refinement Layer

We now describe quantifiers ($\exists n.\, A$, $\forall n.\, A$) and constraints ($?\{\phi\}.\, A$, $!\{\phi\}.\, A$). An overview of the types, process expressions, and their operational meaning can be found in Table 2.

Quantification

The provider of $(c : \exists n.\, A)$ should send a witness e along channel c and then continue as $A[e/n]$. From the typing perspective, we just need to check that the expression e denotes a natural number, using only the permitted variables in $\mathcal{V}$.

$$\frac{\mathcal{V} \; ; \; \mathcal{C} \vdash e : \mathsf{nat}\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta \vDash^{q} P :: (x : A[e/n])}{\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta \vDash^{q} \mathsf{send}\ x\ \{e\} \; ; \; P :: (x : \exists n.\, A)} \; \exists R$$

$$\frac{\mathcal{V}, n \; ; \; \mathcal{C} \; ; \; \Delta, (x : A) \vDash^{q} Q_n :: (z : C) \quad (n \text{ fresh})}{\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta, (x : \exists n.\, A) \vDash^{q} \{n\} \leftarrow \mathsf{recv}\ x \; ; \; Q_n :: (z : C)} \; \exists L$$

The dual type $\forall n.\, A$ reverses the role of the provider and client. The client sends (the value of) an arithmetic expression e which the provider receives and binds to n.

Constraints

Refined session types also allow constraints over index variables. From the message-passing perspective, the provider of $(c : ?\{\phi\}.\, A)$ should send a proof of ϕ along c and the client should receive such a proof. Statically, it is the provider's responsibility to ensure that ϕ holds, while the client is permitted to assume that ϕ is true. The dual operator $!\{\phi\}.\, A$ reverses the role of provider and client. The provider of $c : !\{\phi\}.\, A$ may assume the truth of ϕ, while the client must verify it. The typing rules for the ? type constructor are

$$\frac{\mathcal{V} \; ; \; \mathcal{C} \vDash \phi \quad \mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta \vDash^{q} P :: (x : A)}{\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta \vDash^{q} \mathsf{assert}\ x\ \{\phi\} \; ; \; P :: (x : ?\{\phi\}.\, A)} \; ?R$$

$$\frac{\mathcal{V} \; ; \; \mathcal{C} \wedge \phi \; ; \; \Delta, (x : A) \vDash^{q} Q :: (z : C)}{\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta, (x : ?\{\phi\}.\, A) \vDash^{q} \mathsf{assume}\ x\ \{\phi\} \; ; \; Q :: (z : C)} \; ?L$$

The remaining issue is how to type-check a branch that is impossible due to unsatisfiable constraints. A special impossibility construct is used to handle this situation (dead branches).

$$\frac{\mathcal{V} \; ; \; \mathcal{C} \vDash \bot}{\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta \vDash^{q} \mathsf{impossible} :: (x : A)} \; \mathsf{unsat}$$

There is no operational rule for this scenario since in well-typed configurations the process expression “impossible” is dead code and can never be reached. In practice, we almost never write this construct since reconstruction will fill in missing branches, whose impossibility is then verified by the type checker.

Example: Binary Numbers

As a second example consider natural numbers in binary representation. The idea is that, for example, the number 13 in binary $(1101)_2$ form is represented as a sequence of labels **b1**, **b0**, **b1**, **b1**, **e**, *close* sent or received on a given channel with the least significant bit first. Here **e** represents 0 (the empty sequence of bits), while **b0** and **b1** represent bits 0 and 1, respectively. Because (linear) arithmetic contains no division operator, we express the type $\mathsf{bin}[n]$ of binary numbers with value n using existential quantification, with the concrete syntax `?k. A` for $\exists k.\, A$.

```
type bin{n} = +{ b0 : ?{n > 0}. ?k. ?{n = 2*k}.   bin{k},
                 b1 : ?{n > 0}. ?k. ?{n = 2*k+1}. bin{k},
                 e  : ?{n = 0}. 1 }
```

The constraint that $n > 0$ in the case of **b0** ensures the representation is unique and there are no leading zeros; the same constraint for **b1** is in fact redundant. The `examples/arith.rast` contains several examples of processes over binary numbers like addition, multiplication, predecessor, equality and conversion to and from numbers in unary form.

4 Ergometric and Temporal Session Types

An important application of refinement types is complexity analysis. Prior work on resource-aware session types [6, 5, 4] crucially rely on arithmetic refinements to express work and time bounds. In this section, we review these type systems. The design principle we followed is that they should be *conservative* over the basic and indexed session types, so that previously defined programs and type-checking rules do not change.

4.1 Ergometric Types

The key idea is that *processes store potential* and *messages carry potential.* This potential can either be consumed to perform *work* or exchanged using special messages. The type system provides the programmer with the flexibility to specify what constitutes work. Thus, the programmer can choose to count the resource they are interested in, and the type system provides the corresponding upper bound. Our current examples assign unit cost to message sending operations (exempting those for index objects or potentials themselves) effectively counting the total number of "real" messages exchanged during a computation.

Two dual type constructors $\triangleright^r A$ and $\triangleleft^r A$ are used to exchange potential. The provider of $x : \triangleright^r A$ must *pay* r units of potential along x using process term $(\mathsf{pay}\, x\, \{r\} \; ; \; P)$, and continue to provide A by executing P. These r units are deducted from the potential stored inside the sender. Dually, the client must receive the r units of potential using the term $(\mathsf{get}\, x\, \{r\} \; ; \; Q)$ and add this to its internal stored potential. Finally, since processes are allowed to store potential, the typing judgment records the potential available to a process above the turnstile $\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta \overset{q}{\vdash}_{\Sigma} P :: (x : A)$. We allow potential q to refer to index variables in $\mathcal{V}$ to capture variable potential. The typing rules for $\triangleright^r A$ are

$$\frac{\mathcal{V} \; ; \; \mathcal{C} \vDash q \geq r_1 = r_2 \quad \mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta \vdash^{q-r_1} P :: (x : A)}{\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta \vdash^{q} \mathsf{pay}\, x\, \{r_1\} \; ; \; P :: (x : \triangleright^{r_2} A)} \; \triangleright R$$

$$\frac{\mathcal{V} \; ; \; \mathcal{C} \vDash r_1 = r_2 \quad \mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta, (x : A) \vdash^{q+r_1} Q :: (z : C)}{\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta, (x : \triangleright^{r_2} A) \vdash^{q} \mathsf{get}\, x\, \{r_1\} \; ; \; Q :: (z : C)} \; \triangleright L$$

In both cases, we check that the exchanged potential in the expression and type matches $(r_1 = r_2)$, and while paying, we ensure that the sender has sufficient potential to pay. The dual type $\triangleleft^r A$ enables the provider to receive potential that is sent by its client. Since the sent or received potential must match the one prescribed by the type, our reconstruction algorithm can insert the pay and get actions in a sound and complete way (get as soon as possible and pay as late as possible).

We use a special expression $\mathsf{work}\, \{r\} \; ; \; P$ to *perform work.* Usually, work actions are inserted by the Rast compiler based on a cost model selected by the programmer, such as paying one erg just before every send operation. The programmer can also select a model where all operations are free and manually insert calls to $\mathsf{work}\, \{r\}$. An example of this is given in the file `linlam-reds.rast` that counts the number of reductions necessary for the evaluation of an expression in the linear λ-calculus.

$$\frac{\mathcal{V} \; ; \; \mathcal{C} \vDash q \geq r \quad \mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta \vdash^{q-r} P :: (x : A)}{\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta \vdash^{q} \mathsf{work}\, \{r\} \; ; \; P :: (x : A)} \; \mathsf{work}$$

Work is *precise*, that is, before terminating a process must have 0 potential, which can be achieved by explicitly consuming any remaining potential.

Example: Queue Revisited

We have already seen the ergometric types of queues as a bucket brigade in the introduction. We show it now in concrete syntax, where `<{p}|` receives potential `p`.

```
type queue{n} = &{ins : <{2*n}| A -o queue{n+1},
                  del : <{2}| +{none : ?{n = 0}. 1,
                                some : ?{n > 0}. A * queue{n-1}}}

decl empty : . |- (q : queue{0})
decl elem{n} : (x : A) (r : queue{n}) |- (q : queue{n+1})
```

Interestingly, the exact code of Listing 1 will check against this more informative type (see file `examples/list-work.rast`). The cost model will insert the appropriate $\mathsf{work}\,\{r\}$ action and reconstruction will insert the actions to pay and get potential.

For a queue implemented internally as two stacks we can perform an amortized analysis. Briefly, the queue process maintains two lists: one (*in*) to store messages when they are enqueued, and a reversed list (*out*) from which they are dequeued. When the client wishes to dequeue an element and the *out* list is empty, the provider reverses the *in* list to serve as the new *out* list. A careful analysis shows that if this data structure is used linearly, both insert and delete have constant amortized time. More specifically we obtain the type

```
type queue{n} = &{enq : <{6}| nat -o queue{n+1},
                  deq : <{4}| +{none : ?{n = 0}. 1,
                                some : ?{n > 0}. nat * queue{n-1}}}
```

The program can be found in the file `list-work.rast` in the repository.

4.2 Temporal Types

Rast also supports *temporal modalities next* ($\bigcirc A$), *always* ($\Box A$), and *eventually* ($\Diamond A$), interpreted over a linear model of time. To model computation time, we use the syntactic form delay which advances time by one tick. A particular cost semantics is specified by taking an ordinary, non-temporal program and adding delays capturing the intended cost. For example, if only the blocking operations should cost one unit of time, a delay is added before the continuation of every receiving construct. For type checking, the delay construct subtracts one $\bigcirc$ operator from every channel it refers to. We denote consuming r units on the left of the context using $[A]^t_L$, and on the right by $[A]^t_R$. Briefly, $[\bigcirc^t A]^{-t}_L = [\bigcirc^t A]^{-t}_R = A$.

$$\frac{\mathcal{V} \;;\; \mathcal{C} \vDash t \geq 0 \mathcal{V} \;;\; \mathcal{C} \;;\; [\Delta]^{-t}_L \vdash^q Q :: (x : [A]^{-t}_R)}{\mathcal{V} \;;\; \mathcal{C} \;;\; \Delta \vdash^q \mathsf{delay}\,(t) \;;\; P :: (x : A)} \;\bigcirc LR$$

Always A

A process providing $x : \Box A$ promises to be available at any time in the future, including now. When the client would like to use this provider it (conceptually) sends a message $\mathsf{now!}$ along x and then continues to interact according to type A.

A process P providing $x : \Box A$ must be able to wait indefinitely. But this is only possible if all the channels that P uses can also wait indefinitely. This is enforced in the rule by the condition $\Delta\ \mathsf{delayed}^{\Box}$ which requires each antecedent to have the form $y_i : \bigcirc^{n_i} \Box B_i$.

$$\frac{\Delta\ \mathsf{delayed}^{\Box} \quad \Delta \vdash P :: (x : A)}{\Delta \vdash (\mathsf{when?}\,(x) \;;\; P) :: (x : \Box A)} \;\Box R \qquad \frac{\Delta, x : A \vdash Q :: (z : C)}{\Delta, x : \Box A \vdash (\mathsf{now!}\,(x) \;;\; Q) :: (z : C)} \;\Box L$$

Rast also has its dual modality $\Diamond A$, which communicates at some indeterminate future time. This is used when the time (span) of a computation is unpredictable or not expressible within the constraints of the language (more details in prior work [5]).

Example: Queue Revisited

We have already foreshadowed the temporal type of a queue, implemented as a bucket brigade. We show it now in concrete syntax, where `()` is the $\bigcirc$ modality and `[]` represents $\Box$. We also show the types of the `empty` and `elem` processes (see file `examples/time.rast`).

```
type queue{n} = [] {enq : () A -o ()()()queue{n+1},
                    deq : ()+{none: () ?{n = 0}. 1,
                              some: () ?{n > 0}. A * ()queue{n-1}}}
decl empty : . |- (q : ()()queue{0})
decl elem{n} : (x : A) (r : ()()queue{n}) |- (q : queue{n+1})
```

Because Rast currently does not have reconstruction for time we have to update the program with the five temporal actions presented in this section (two instances of `delay`, two of `when`, and one of `now`). A key observation here is that in the case of `elem` the process r does not need to be ready instantaneously, but can be ready after a delay of 2 ticks, because that is how long it takes to receive the **ins** label and the element along q. This slack is also reflected in the type of `empty` because it becomes then back of a new element when the end of the queue is reached.

5 Implementation

We have implemented a prototype for Rast in Standard ML (6700 lines of code). This implementation contains a lexer and parser (1355 lines), an arithmetic solver (1083 lines), a type checker (2852 lines), pretty printer (375 lines), reconstruction engine (880 lines), and interpreter (155 lines). The source code is well-documented and available open-source.

Syntax

Table 3 describes the syntax for Rast programs. Each row presents the abstract and concrete representation of a session type, and its corresponding providing expression. A program contains a series of mutually recursive type and process declarations and definitions.

```
type v{n} = A
decl f : (x1 : A1) ... (xn : An) |- (x : A)
proc x <- f x1 ... xn = P
```

Listing 2 Top-Level Declarations

The first line is a *type definition*, where v is the name with index variable n and A is its definition. The second line is a *process declaration*, where f is the process name, $(x_1 : A_1) \dots (x_n : A_n)$ are the used channels and corresponding types, while the offered channel is x of type A. Finally, the last line is a *process definition* for the same process f defined using the process expression P. We use a hand-written lexer and shift-reduce parser to read an input file and generate the corresponding abstract syntax tree of the program. The reason to use a hand-written parser instead of a parser generator is to anticipate the most common syntax errors that programmers make and respond with the best possible error messages.

Table 3 Abstract and Corresponding Concrete Syntax for Types and Expressions.

Abstract Types	Concrete Types	Abstract Syntax	Concrete Syntax
$\oplus\{l : A, \ldots\}$	`+{l : A, ...}`	$x.k$	`x.k`
$\&\{l : A, \ldots\}$	`&{l : A, ...}`	case x $(\ell \Rightarrow P)_{\ell \in L}$	`case x (l => P | ...)`
$A \otimes B$	`A * B`	send x w	`send x w`
$A \multimap B$	`A -o B`	$y \leftarrow$ recv x	`y <- recv x`
$\mathbf{1}$	`1`	close x	`close x`
		wait x	`wait x`
$\exists n. A$	`?n. A`	send x $\{e\}$	`send x {e}`
$\forall n. A$	`!n. A`	$\{n\} \leftarrow$ recv x	`{n} <- recv x`
$?\{n = 0\}. A$	`?{n = 0}. A`	assert x $\{n = 0\}$	`assert x {n = 0}`
$!\{n = 0\}. A$	`!{n = 0}. A`	assume x $\{n = 0\}$	`assume x {n = 0}`
$\triangleright^r A$	`|{r}> A`	pay x $\{r\}$	`pay x {r}`
$\triangleleft^r A$	`<{r}| A`	get x $\{r\}$	`get x {r}`
$\bigcirc^t A$	`({t}) A`	delay t	`delay {t}`
$\Box A$	`[] A`	when x	`when x`
$\Diamond A$	`<> A`	now x	`now x`
$V[\overline{e}]$	`V{e1}...{ek}`	$x \leftrightarrow y$	`x <-> y`
		$x \leftarrow f\ x_1 \ldots x_n$	`x <- f x1 ... xn`

Validity Checking

Once the program is parsed and its abstract syntax tree is extracted, we perform a validity check on it. We check that all index refinements, potentials, and delay operators are non-negative. We also check that all index expressions are closed with respect to the the index variables in scope. To simplify and improve the efficiency of the type equality algorithm, we also assign internal names to type subexpressions parameterized over their free index variables. These internal names are not visible to the programmer.

Cost Model

The cost model defines the execution cost of each construct. Since our type system is parametric in the cost model, we allow programmers to specify the cost model they want to use. Although programmers can create their own cost model (by inserting work or delay expressions in the process expressions), we provide three custom cost models: send, recv, and recvsend. If we are analyzing work (resp. time), the send cost model inserts a work$\{1\}$ (resp. delay$\{1\}$) before (resp. after) each send operation. Similarly, recv model assigns a cost of 1 to each receive operation. The recvsend cost model assigns a cost of 1 to each send and receive operation.

Reconstruction and Type Checking

The programmer can use a flag in the program file to indicate whether they are using *explicit* or *implicit* syntax. If the syntax is explicit, the reconstruction engine performs no program transformation. However, if the syntax is implicit, we use the implicit type system to approximately type-check the program. Once completed, we use the forcing calculus, introduced in prior work [7] to insert assert, assume, pay, get and work constructs. The core idea here is simple: insert assume or get constructs eagerly, i.e., as soon as available on a channel, and insert assert and pay lazily, i.e., just before communicating on that channel. The forcing calculus proves that this reconstruction technique is sound and complete in the absence of certain forms of quantifier alternations (which are checked before reconstruction is performed). We only perform reconstruction for proof constraints and ergometric types, leaving reconstruction of quantifiers and temporal constructs to future work.

The implementation takes some care to provide good error messages, in particular as session types (not to mention arithmetic refinements, ergometric types, and temporal types) are likely to be unfamiliar. One technique is staging: first check approximate type correctness, ignoring index, ergometric, and temporal types, and only if that check passes perform reconstruction and strict checking of type. Another particularly helpful technique has been type compression. Whenever the type checker expands a type $V[\overline{e}]$ with $V[\overline{n}] = A$ to $A[\overline{e}/\overline{n}]$ we record a reverse mapping from $A[\overline{e}/\overline{n}]$ to $V[\overline{e}]$. When printing types for error messages this mapping is consulted, and complex types may be compressed to much simpler forms, greatly aiding readability of error messages.

Type Equality

At the core of type checking lies type equality, defined coinductively [10]. With arithmetic refinements this equality is undecidable, but have found what seems to be a practical approximation [7], incrementally constructing a bisimulation closed under reflexivity. This algorithm always terminates, but may fail to establish an equality if the coinductive invariant is not general enough. Rast therefore allows the programmer to assert an arbitrary number of additional type equalities with the construct

```
eqtype V{e1}...{en} = V'{e1'}...{ek'}
```

These are then checked one by one, assuming all other asserted equalities. The default construction of the bisimulation is currently strong enough so that this feature has not been needed for any of our standard examples.

Arithmetic Solver

To determine the validity of arithmetic propositions that is used by our refinement layer, we use a straightforward implementation of Cooper's decision procedure [3] for Presburger arithmetic. We found a small number of optimizations were necessary, but the resulting algorithm has been quite efficient and robust.

(i) We eliminate constraints of the form $x = e$ (where x does not occur in e) by substituting e for x in all other constraints to reduce the total number of variables.

(ii) We exploit that we are working over natural numbers so all solutions have a natural lower bound, i.e., 0.

We also extend our solver to handle non-linear constraints. Since non-linear arithmetic is undecidable, in general, we use a normalizer which collects coefficients of each term in the multinomial expression.

(i) To check $e_1 = e_2$, we normalize $e_1 - e_2$ and check that each coefficient of the normal form is 0.

(ii) To check $e_1 \geq e_2$, we normalize $e_1 - e_2$ and check that each coefficient is non-negative.

(iii) If we know that $x \geq c$, we substitute $y + c$ for x in the constraint that we are checking with the knowledge that the fresh $y \geq 0$.

(iv) We try to find a quick counterexample to validity by plugging in 0 and 1 for the index variables (which can be improved in the future).

If the constraint does not fall in the above two categories, we print the constraint and trust that it holds. A user can then view these constraints manually and confirm their validity. At present, all of our examples pass without having to trust unsolvable constraints with our current set of heuristics beyond Presburger arithmetic.

Interpreter

The current version of the interpreter pursues a sequential schedule following a prior proposal [16]. We only execute programs that have no free index variables and only one externally visible channel, namely the one provided. When the computation finishes, the messages that were asynchronously sent along this distinguished channel are shown, while running processes waiting for input are displayed simply as a dash '-'.

The interpreter is surprisingly fast. For example, using a linear prime sieve to compute the status (prime or composite) or all number in the range $[2, 257]$ takes 27.172 milliseconds using MLton during our experiments (see machine specifications below).

6 Examples

We present several different kinds of examples from varying domains illustrating different features of the type system and algorithms. Table 4 describes the results: iLOC describes the lines of source code in implicit syntax, eLOC describes the lines of code after reconstruction (which inserts implicit constructs), #Defs shows the number of process definitions, R (ms) and T (ms) show the reconstruction and type-checking time in milliseconds respectively. Note that reconstruction is faster than type-checking since reconstruction does not involve solving any arithmetic propositions. The experiments were run on an Intel Core i5 2.7 GHz processor with 16 GB 1867 MHz DDR3 memory.

(i) **arithmetic**: natural numbers in unary and binary representation indexed by their value and processes implementing standard arithmetic operations.
(ii) **integers**: an integer counter represented using two indices x and y with value $x - y$.
(iii) **linlam**: expressions in the linear λ-calculus indexed by their size.
(iv) **list**: lists indexed by their size, and some standard operations such as *append, reverse, map, fold*, etc. Also provides and implementation of stacks and queues using lists.
(v) **primes**: the sieve of Eratosthenes to classify numbers as prime or composite.
(vi) **segments**: type $\mathsf{seg}[n] = \forall k.\mathsf{list}[k] \multimap \mathsf{list}[n+k]$ representing partial lists with constant-work append operation.
(vii) **ternary**: natural numbers and integers represented in balanced ternary form with digits $0, 1, -1$, indexed by their value, and a few standard operations on them.
(viii) **theorems**: processes representing valid circular [8] proofs of simple theorems such as $n(k+1) = nk + n, n + 0 = n, n * 0 = 0$, etc.
(ix) **tries**: a trie data structure to store multisets of binary numbers, with constant amortized work insertion and deletion verified with ergometric types.

We highlight interesting examples from some case studies showcasing the invariants that can be proved using arithmetic refinements.

Linear λ-Calculus

We demonstrate an implementation of the (untyped) linear λ-calculus, including evaluation, in which the index objects track the size of the expression. Type-checking verifies that the result of evaluating a linear λ-term is no larger than the original term. Our representation uses linear higher-order abstract syntax (see file `examples/linlam-size.rast`).

Table 4 Case Studies.

Module	iLOC	eLOC	#Defs	R (ms)	T (ms)
arithmetic	395	619	29	0.959	5.732
integers	90	125	8	0.488	0.659
linlam	88	112	10	0.549	1.072
list	341	642	37	3.164	4.637
primes	118	164	11	0.289	4.580
segments	48	76	8	0.183	0.225
ternary	270	406	20	0.947	140.765
theorems	79	156	13	0.182	1.095
tries	243	520	13	2.122	6.408
Total	**1672**	**2820**	**149**	**8.883**	**165.173**

```
type exp{n} = +{ lam : ?{n > 0}. !n1.exp{n1} -o exp{n1+n-1},
                 app : ?n1. ?n2. ?{n = n1+n2+1}. exp{n1} * exp{n2} }
type val{n} = +{ lam : ?{n > 0}. !n1.exp{n1} -o exp{n1+n-1} }
decl eval{n} : (e : exp{n}) |- (v : ?k. ?{k <= n}. val{k})
```

An expression of size n is either a λ (the label **lam**) or an application (label **app**). In case of **lam**, after proving $n > 0$, it expects an expression of size n_1 as an argument and then behaves like the body of the λ-abstraction of size $n_1 + n - 1$. In case of **app**, it sends two expressions of size n_1 and n_2 such that $n = n_1 + n_2 + 1$.

Interestingly, the result type of evaluation contains an existential quantifier since we do not know the precise size of the value –we just know it is bounded by n. Also, as exemplified in the type of `val{n}`, a value can only be a λ-expression (label **app** missing).

Trie Data Structure

We illustrate the data structure of a trie to maintain multisets of binary numbers. There is a fair amount of parallelism since consecutive requests to insert numbers into the trie can be carried out concurrently. We also obtain a good characterization of the necessary work –the data structure is quite efficient (in theoretical terms). We start with binary numbers where each bit carries potential p.

```
type bin{n}{p} = +{ b0 : ?{n > 0}. ?k. ?{n = 2*k}. |{p}> bin{k}{p},
                    b1 : ?{n > 0}. ?k. ?{n = 2*k+1}. |{p}> bin{k}{p},
                    e : ?{n = 0}. 1 }
```

A trie is represented by the type $\mathsf{trie}[n]$ where n is the number of elements in the current multiset. When inserting a number it updates to $\mathsf{trie}[n+1]$. When we delete a number x from the trie we delete all copies of x and return its multiplicity. If m is the multiplicity of the number, then after deletion the trie will have $\mathsf{trie}[n-m]$ elements. This requires the constraint that $m \leq n$: the multiplicity of an element cannot be greater than the total number of elements in the multiset.

When inserting a binary number into the trie that number can be of any value. Therefore, we must pass the index k representing that value, which is represented by a universal quantifier in the type. Conversely, when responding we need to return the unique binary number m which is of course not known statically and therefore is an existential quantifier.

The way we insert the binary number is starting at the root with the least significant bit and recursively insert the number into the left or right subtrie, depending on whether the bit is **b0** or **b1**. When we reach the end of the sequence of bits (**e**) we increase the multiplicity at the leaf we have reached. As we traverse the trie, we need to construct new intermediate nodes in case we encounter a leaf. These operations require 4 messages per bit, so the input number should have potential of 4 per bit. For deletion, we need one more because we need to communicate the answer back to the client, so 5 units per bit. For simplicity, we therefore uniformly require 5 units of potential per bit when adding a number to the trie and "burn" the extra unit during insertion.

```
type trie{n} =
  &{ ins: <{4}| !k. bin{k}{5} -o trie{n+1},
     del: <{5}| !k. bin{k}{5} -o ?m. ?{m<=n}. bin{m}{0} * trie{n-m}}
```

We have two kinds of nodes: leaf nodes (process $\mathsf{leaf}[0]$) not holding any elements and element nodes (process $\mathsf{nodes}[n_0, m, n_1]$) representing an element of multiplicity m with n_0 and n_1 elements in the left and right subtries, respectively. A node therefore has type $\mathsf{trie}[n_0 + m + n_1]$. Neither process carries any potential.

```
decl leaf : . |- (t : trie{0})
decl node{n0}{m}{n1} :
  (l : trie{n0}) (c : ctr{m}) (r : trie{n1}) |- (t : trie{n0+m+n1})
```

The source code is available at `examples/trie-work.rast`.

7 Conclusion

This paper describes the Rast programming language. In particular, we focused on the concrete syntax, type checker and equality, the refinement layer [7], and its applicability to work [6] and time analysis [5]. The refinements rely on an arithmetic solver based on Cooper's algorithm [3]. The interpreter uses the shared memory semantics introduced in recent work [16]. We concluded with several examples demonstrating the efficacy of the refined type system in expressing and verifying properties about data structure sizes and values. We also illustrated the work and time bounds for several examples, all of which have been verified with our system, and are available in an open-source repository.

References

1 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory*, CONCUR'10, pages 222–236, Berlin, Heidelberg, 2010. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1887654.1887670.

2 Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 760, November 2014. doi:10.1017/S0960129514000218.

3 David C Cooper. Theorem proving in arithmetic without multiplication. *Machine intelligence*, 7(91-99):300, 1972.

4 Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. Resource-aware session types for digital contracts, 2019. arXiv:1902.06056.

5 Ankush Das, Jan Hoffmann, and Frank Pfenning. Parallel complexity analysis with temporal session types. *Proc. ACM Program. Lang.*, 2(ICFP):91:1–91:30, July 2018. doi:10.1145/3236786.

6 Ankush Das, Jan Hoffmann, and Frank Pfenning. Work analysis with resource-aware session types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 305–314, New York, NY, USA, 2018. ACM. doi:10.1145/3209108.3209146.

7 Ankush Das and Frank Pfenning. Session types with arithmetic refinements and their application to work analysis, 2020. arXiv:2001.04439.

8 Farzaneh Derakhshan and Frank Pfenning. Circular Proofs as Session-Typed Processes: A Local Validity Condition. *arXiv e-prints*, page arXiv:1908.01909, August 2019. arXiv:1908.01909.

9 Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. Cut reduction in linear logic as asynchronous session-typed communication. In P. Cégielski and A. Durand, editors, *Proceedings of the 21st Annual Conference on Computer Science Logic (CSL 2012)*, pages 228–242, Fontainebleau, France, September 2012. LIPIcs 16.

10 Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2):191–225, November 2005. doi:10.1007/s00236-005-0177-z.

11 Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, January 2010.

12 Dennis Griffith and Elsa L. Gunter. Liquidpi: Inferrable dependent session types. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods*, pages 185–197, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

13 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, pages 509–523, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

14 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems*, pages 122–138, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

15 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 273–284, New York, NY, USA, 2008. ACM. doi:10.1145/1328438.1328472.

16 Klaas Pruiksma and Frank Pfenning. A shared-memory semantics for mixed linear and non-linear session types. unpublished, 2018.

17 Vasco T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70, 2012. doi:10.1016/j.ic.2012.05.002.

18 Philip Wadler. Propositions as sessions. *SIGPLAN Not.*, 47(9):273–286, September 2012. doi:10.1145/2398856.2364568.

Hierarchy Builder: Algebraic hierarchies Made Easy in Coq with Elpi

Cyril Cohen
Université Côte d'Azur, Inria, Sophia Antipolis, France
Cyril.Cohen@inria.fr

Kazuhiko Sakaguchi
University of Tsukuba, Japan
sakaguchi@logic.cs.tsukuba.ac.jp

Enrico Tassi
Université Côte d'Azur, Inria, Sophia Antipolis, France
Enrico.Tassi@inria.fr

Abstract

It is nowadays customary to organize libraries of machine checked proofs around hierarchies of algebraic structures [2, 6, 8, 16, 18, 23, 27]. One influential example is the Mathematical Components library on top of which the long and intricate proof of the Odd Order Theorem could be fully formalized [14].

Still, building algebraic hierarchies in a proof assistant such as Coq [9] requires a lot of manual labor and often a deep expertise in the internals of the prover [13, 17]. Moreover, according to our experience [26], making a hierarchy evolve without causing breakage in client code is equally tricky: even a simple refactoring such as splitting a structure into two simpler ones is hard to get right.

In this paper we describe $\mathcal{HB}$, a high level language to *build* hierarchies of algebraic structures and to make these hierarchies *evolve* without breaking user code. The key concepts are the ones of *factory*, *builder* and *abbreviation* that let the hierarchy developer describe an actual interface for their library. Behind that interface the developer can provide appropriate code to ensure backward compatibility. We implement the $\mathcal{HB}$ language in the `hierarchy-builder` addon for the Coq system using the Elpi [11, 28] extension language.

2012 ACM Subject Classification Software and its engineering → Formal language definitions; Theory of computation → Type theory; Theory of computation → Constraint and logic programming; Computing methodologies → Symbolic and algebraic manipulation

Keywords and phrases Algebraic Hierarchy, Packed Classes, Coq, Elpi, Metaprogramming, λProlog

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.34

Category System Description

Supplement Material Coq package source: `https://github.com/math-comp/hierarchy-builder`

Acknowledgements We are particularly grateful to Georges Gonthier, Assia Mahboubi and Florent Hivert for the many discussions we had that were instrumental in coining the concepts formalized here. We thank as well other developers of the Mathematical Components library and members of Dagstuhl seminar 18341, for the helpful discussions we had. We also thank many Coq developers and users who showed interest and support in this work. Finally, we thank the anonymous reviewers of this paper for their useful comments.

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 34; pp. 34:1–34:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Modern libraries of machine checked proofs are organized around hierarchies of algebraic structures [2, 6, 8, 16, 18, 23, 27]. For example the Mathematical Components library for the Coq system [9] provides a very rich, ever growing, hierarchy of structures such as group, ring, module, algebra, field, partial order, order, lattice... The hierarchy does not only serve the purpose of organizing knowledge, but also to make it easy to exploit it. Indeed the interactive prover can take advantage of the structure of the library and the relation between its concepts to infer part of information usually left implicit by the user, a capability that turned out to be key to tame the complexity and size of the formal proof of the Odd Order Theorem [14].

The hierarchy of the Mathematical Components library is implemented following the discipline of Packed Classes initially introduced by Garillot, Gonthier, Mahboubi and Rideau [13] and later also adopted by Affeldt, Nowak, and Saikawa to describe a hierarchy of monadic effects [2] and by Boldo, Lelay, and Melquiond in the Coquelicot library of real analysis [6]. We call Packed Classes a discipline, and not a language, because, in spite of its many virtues, it is unwieldy to use. In particular it leaks to the user many of the technical details of the Coq system. As a result, one needs to be a Coq expert in order to build or modify a hierarchy, and even experts make mistakes as shown in [26]. Another inconvenience of the Packed Classes discipline is that even simple changes to the hierarchy, such as splitting a structure into two simpler ones, break user code.

In this paper we describe $\mathcal{HB}$, a high level language to *build* hierarchies of algebraic structures and to make these hierarchies *evolve* without breaking user code. The key concepts are the ones of *factory*, *builder* and *abbreviation* that let the hierarchy developer describe an actual interface for their library. Behind that interface the developer can provide appropriate code to ensure backward compatibility. We implement the $\mathcal{HB}$ language by compiling it to a variant of the Packed Classes discipline, that we call *flat*, in the `hierarchy-builder` addon for Coq. We write this addon using the Elpi [11, 28] extension language.

To sum up, the main contributions of the paper are:

- the design of the $\mathcal{HB}$ language,
- the compilation of $\mathcal{HB}$ to the (flat) Packed Classes discipline, and
- the implementation of $\mathcal{HB}$ in the `hierarchy-builder` addon for Coq.

The paper is organized as follows. Via an example we introduce $\mathcal{HB}$ and its key ideas. We then describe the discipline of Packed Classes and we show how $\mathcal{HB}$ can be compiled to it. We then discuss the implementation of the Coq addon via the Elpi extension language and we position $\mathcal{HB}$ in the literature.

2 $\mathcal{HB}$ by examples: building and evolving a hierarchy

The first version of our hierarchy (that we name V1) features only two structures: `Monoid` and `Ring`. We use notations from Appendix A to define them.

```
HB.mixin Record Monoid_of_Type M := {
  zero : M;
  add : M -> M -> M;
  addrA : associative add;             (* `add` is associative.                   *)
  add0r : left_id zero add;            (* `zero` is the left and right neutral     *)
  addr0 : right_id zero add;           (*   element with respect to `add`.         *)
}.
HB.structure Definition Monoid := { M of Monoid_of_Type M }.

```

```
10 HB.mixin Record Ring_of_Monoid R of Monoid R := {
11   one : R;
12   opp : R -> R;
13   mul : R -> R -> R;
14   addNr : left_inverse zero opp add;   (* `opp x` is the left and right additive  *)
15   addrN : right_inverse zero opp add;  (*   inverse of `x`.                        *)
16   mulrA : associative mul;             (* `mul` is associative.                   *)
17   mul1r : left_id one mul;             (* `one` is the left and right neutral     *)
18   mulr1 : right_id one mul;            (*   element with respect to `mul`.        *)
19   mulrDl : left_distributive mul add;  (* `mul` is left and right distributive    *)
20   mulrDr : right_distributive mul add; (*   over `add`.                            *)
21 }.
22 HB.structure Definition Ring := { R of Monoid R & Ring_of_Monoid R }.
```

In order to build a structure we need to declare some factories and later assemble them. One kind of factory supported by $\mathcal{HB}$, the simplest one, is called *mixin* and is embodied by a record that gathers operations and properties.

Mixins are declared via the HB.mixin command that takes a record declaration with a type parameter and a possibly empty list of factories for that type. The code between lines 1 and 7 declares a mixin that can turn a naked type M into a monoid, hence we chose the name to be Monoid_of_Type.

The HB.structure command takes in input a definition for a sigma type S that equips a type with a list of factories. It registers a structure S in the hierarchy placing any definition specific to that structure inside a Coq module named S. Line 8 hence forges the Monoid structure.

Line 10 declares a second mixin collecting the operations and properties that are needed in order to enrich a monoid to a ring, hence the name Ring_of_Monoid. Indeed this time the type variable R is followed by Monoid that enriches R with the operations and properties of monoids. As a consequence add and zero can be used to express the new properties.

The last line declares the Ring structure to hold all the axioms declared so far. We can now inspect the contents of the hierarchy and then proceed to build a theory about abstract rings, register examples (instances) of ring structures and finally use the abstract theory on these examples.

```
1  Print Monoid.type. (* Monoid.type  := { sort : Type;  ... }                    *)
2  Check @add.        (* add          :  forall M : Monoid.type, M -> M -> M      *)
3  Check @addNr.      (* addNr        :  forall R : Ring.type, left_inverse zero opp add *)
4
5  Lemma addrC {R : Ring.type} : commutative (@add R).
6  Proof. (* Proof by Hankel 1867, in Appendix B *) Qed.
7
8  Definition Z_Monoid_axioms : Monoid_of_Type Z :=
9    Monoid_of_Type.Build Z 0%Z Z.add Z.add_assoc Z.add_0_l Z.add_0_r.
10
11 HB.instance Z Z_Monoid_axioms.
12
13 Definition Z_Ring_axioms : Ring_of_Monoid Z :=
14   Ring_of_Monoid.Build Z 1%Z Z.opp Z.mul
15     Z.add_opp_diag_l Z.add_opp_diag_r Z.mul_assoc Z.mul_1_l Z.mul_1_r
16     Z.mul_add_distr_r Z.mul_add_distr_l.
17
18 HB.instance Z Z_Ring_axioms.
19
20 Lemma exercise (m n : Z) : (n + m) - n * 1 = m.
21 Proof. by rewrite mulr1 (addrC n) -(addrA m) addrN addr0. Qed.
```

We can print the type for monoids as forged by $\mathcal{HB}$ (line 1). It packs a carrier, called sort, and the collection of operation and properties that we omit for brevity. We can also look at the type of two constants synthesized by $\mathcal{HB}$ out of the hierarchy declaration. Remark that while the names of the constants come from the names of mixin fields, their types differ. In particular they are quantified over a `Monoid.type` or `Ring.type`, and not a simple type as in the mixins. Moreover we evince that the `sort` projection is declared as an implicit coercion [24] and is automatically inserted in order to make `M -> M -> M` a meaningful type for binary operations on the carrier of `M`. Last, we see that properties are quantified on (hence apply to) the structure they belong to but use, in their statements, operations belonging to simpler structures. For example `addNr` is a property of a ring but its statement mentions `add`, the operation of the underlying monoid.

We then follow the proof of Hankel [5] to show that ring axioms imply the commutativity of the underlying monoid (line 5). This simple example shows we can populate the theory of abstract rings with new results.

We use the `Monoid_of_Type.Build` *abbreviation* (line 8) in order to build an instance of the `Monoid` structure for binary integers `Z`. We then register that monoid instance as the canonical one on `Z` (line 11) via the command `HB.instance`. We can similarly declare that `Z` forms a ring by using the `Ring_of_Monoid.Build` abbreviation (lines 13 and 18). Note that the `Ring_of_Monoid.Build` abbreviation is not a plain record constructor for `Ring_of_Monoid`, since that would require more arguments, namely the monoid ones (see the _ at line 13). The abbreviation synthesized by $\mathcal{HB}$ infers them automatically (as in [17, Section 7]) thanks to the `HB.instance` declaration given just above.

From now on the axioms as well as the abstract theory of rings apply to integers, as shown in lemma `exercise`. The details of the proof do not matter here, what is worth pointing out is that in a single statement we mix monoid (e.g. `+`) and ring (e.g. `-`) operations and in the proof we use monoid axioms (e.g. `addrA`), ring axioms (e.g. `addrN`) and ring lemmas (e.g. `addrC`), all seamlessly.

2.1 Evolution of the hierarchy

We proceed by accommodating the intermediate structure of Abelian groups.

```
HB.mixin Record Monoid_of_Type M := { ... (* unchanged *) ... }.
HB.structure Definition Monoid := { M of Monoid_of_Type M }.

HB.mixin Record AbelianGroup_of_Monoid A of Monoid A := {
  opp : A -> A;
  addrC : commutative (add : A -> A -> A);
  addNr : left_inverse zero opp add;
}.
HB.structure Definition AbelianGroup := { A of Monoid A & AbelianGroup_of_Monoid A }.

HB.mixin Record Ring_of_AbelianGroup R of AbelianGroup R := {
  one : R;
  mul : R -> R -> R;
  mulrA : associative mul;
  mul1r : left_id one mul;              mulr1 : right_id one mul;
  mulrDl : left_distributive mul add;   mulrDr : right_distributive mul add;
}.
HB.structure Definition Ring := { R of AbelianGroup R & Ring_of_AbelianGroup R }.

Lemma addrN {A : AbelianGroup.type} : right_inverse zero opp add.
Proof. by move=> x; rewrite addrC addNr. Qed.
```

Some operations and properties were moved from the old mixin for rings into a newborn mixin `AbelianGroup_of_Monoid` that gathers the axioms needed to turn a monoid into an Abelian group. Consequently the mixin for rings is now called `Ring_of_AbelianGroup` (instead of `Ring_of_Monoid`) since it expects the type `R` to be already an Abelian group and hence gathers fewer axioms. While operations moved from one structure to another, some properties undergo a deep change in their status. The lemma `addrC` part of the abstract theory of rings is now an axiom of Abelian groups, while `addrN` is no more an axiom of rings, but rather a theorem of the abstract theory of Abelian groups proved at line 20.

It is worth clarifying here what axioms and lemmas are by looking at the two distinct use cases for a hierarchy: 1) develop the abstract theory of some structure; 2) apply the abstract theory on some concrete example. In the first case all the axioms of the structure are (assumed to be) theorems so the distinction between axiom and theorem does not matter. This is what happens in the proof of `addNr` by taking `R` to be of type `AbelianGroup.type` as an assumption. In the second case, in order to show Coq that a data type and some operations forms a structure, one must pick the axioms of the structure an prove them for that specific type and operations. For example the `Ring_of_Monoid.Build` abbreviation is used at page 3 to package the set of proofs that make `Z` a ring (proviso `Z` is already a monoid).

With this new version of the hierarchy, that we name V2, the axiomatic that was previously exposed to user changed, and indeed code written for version V1 breaks. For example the declaration of the canonical ring over the integers fails, if only because we do not have a `Ring_of_Monoid.Build` abbreviation anymore.

Our objective is to obtain a version of the hierarchy, that we name V3, that does not only feature Abelian groups but that is also backward compatible with V1.

2.2 The missing puzzle piece

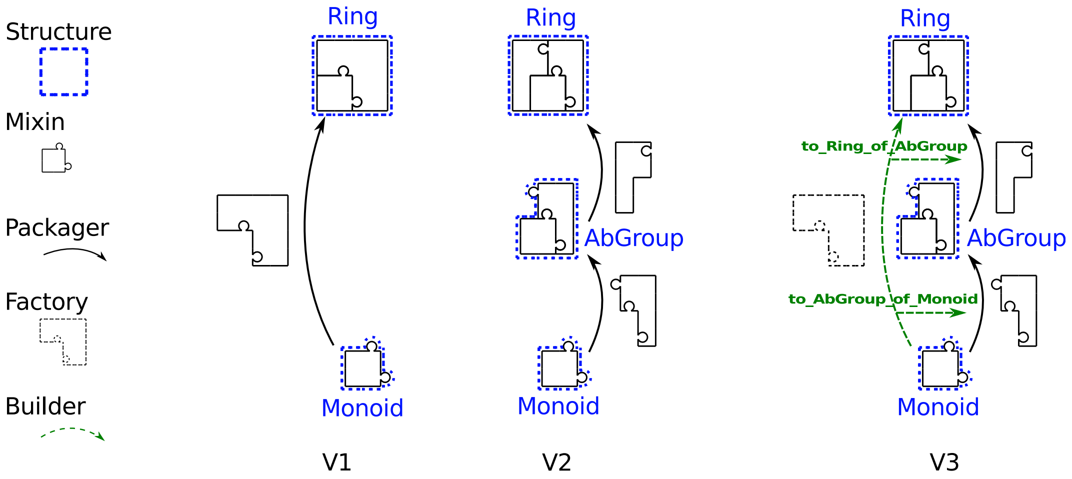


Figure 1 The evolution of the hierarchy. V3 is backward compatible with V1, while V2 is not.

The key to make a hierarchy evolve without breaking user code is the full fledged notion of *factory* (the mixins seen so far are degenerate, trivial, factories). Factories, like mixins, are packages for operations and properties but are not directly used in the definition of structures. Instead a factory is equipped with *builders*: user provided pieces of code that extract from the factory the contents of mixins, so that existing abbreviations can be used.

As depicted in Figure 1 we change again the hierarchy by declaring a `Ring_of_Monoid` factory, that, from the user point of view, will look indistinguishable from the old `Ring_of_Monoid` mixin and hence grant backward compatibility between version V3 and version V1.

```
1  HB.factory Record Ring_of_Monoid R of Monoid R := { ... (* unchanged *) ...}.
2
3  HB.builders Context R (f : Ring_of_Monoid R).
4
5    Lemma addrC : commutative add. Proof. (* The same proof as before *) Qed.
6
7    Definition to_AbelianGroup_of_Monoid :=
8      AbelianGroup_of_Monoid.Build R opp addrC addNr. (* addrN unused *)
9    HB.instance R to_AbelianGroup_of_Monoid.
10
11   Definition to_Ring_of_AbelianGroup := Ring_of_AbelianGroup.Build R one mul
12     mulrA mul1r mulr1 mulrDl mulrDr.
13   HB.instance R to_Ring_of_AbelianGroup.
14
15 HB.end.
```

The record `Ring_of_Monoid` is the same we declared as a mixin in version V1. In order to make a factory out of it we equip it with two definitions that embody the *builders*. The first is `to_AbelianGroup_of_Monoid` and explains how to build an `AbelianGroup` structure out of the factory axioms (named `f`, line 3). This construction is also registered as canonical for `R`, so that the next construction `to_Ring_of_AbelianGroup` can call the `Ring_of_AbelianGroup.Build` abbreviation that requires `R` to be an `AbelianGroup`. It is worth pointing out that the proof of `addrC` we had in V1 is now required in order to write the builder for Abelian groups, while the `addrN` field is not used (the same statement is already part of the theory of Abelian groups, see line 20 of the previous code snippet).

Thanks to this factory we can now declare `Z` to be an instance of a ring using the `Ring_of_Monoid.Build` abbreviation. The associated builders generate, behind the scenes, instances of the `Ring_of_AbelianGroup` and `AbelianGroup_of_Monoid` mixins that in turn are used to build instances of the `AbelianGroup` and `Ring` structures. Indeed, when used in the context of the hierarchy version V3, the command `HB.instance Z Z_Ring_axioms` makes `Z` an instance of *both* structures, and not just the `Ring` one as in version V1. Thanks to that the proof of `example` can use the theory of both structures, for example `addrC` holds on Abelian groups, `addrA` holds on monoids, while `addr1` holds on rings. As a result the very same proof works on both version V1 and V3.

Last, it is worth pointing out that the new factory makes the following two lines equivalent (the former declares rings in V1) since they both describe the same set of mixins.

```
HB.structure Definition Ring := { R of Monoid R & Ring_of_Monoid R }.
HB.structure Definition Ring := { R of AbelianGroup R & Ring_of_AbelianGroup R }.
```

This is another example of client code that would not break: the client of the hierarchy is allowed to declare new structures on top of existing ones.

2.3 $\mathcal{HB}$ in a nutshell

By using $\mathcal{HB}$ the hierarchy designer has the following freedoms and advantages.

- Operations and properties (axioms) are made available to the user as soon as they are used in a structure. The hierarchy designer is free to move them from one to another and replace an axiom by a lemma and vice versa.
- Structures cannot disappear but the way they are built may change. The hierarchy designer is free to split structures into smaller ones in order to better factor and reuse parts of the hierarchy and the library that follows it.
- Mixins cannot disappear but can change considerably in their implementation. A mixin can become a full fledged factory equipped with builders to ensure backward compatibility.

- $\mathcal{HB}$ high level commands compile to the discipline of Packed Classes (Coq modules, records, coercions, implicit arguments, canonical structure instances, notations, abbreviations). This process lifts a considerable burden from the shoulders of the hierarchy designer and the final user who are no longer required to master the details of Packed Classes.

3 The $\mathcal{HB}$ language

The Coq terms handled by $\mathcal{HB}$ are subdivided in five categories, the mixins $\mathcal{M}$, the factories $\mathcal{F}$, builders $\mathcal{B}$, the classes $\mathcal{C}$ and the structures $\mathcal{S}$. Mixins, factories and instances are tagged by the user, through the commands `HB.mixin`, `HB.factory` and `HB.instance`, and the user may rely on their implementation. However structures and classes are generated with the command `HB.structure` and builders are generated when using `HB.instance` while declaring a factory, and the user may only refer to structure types, but should never rely on their implementation, neither should they rely on explicit builders.

3.1 Mixins, factories and instances

In this section, we call "distinguished" a Coq definition or record that the developer of a library has labeled "mixin", "factory", "builder", "class" or "structure".

▶ **Definition 1** ($\mathcal{M}$, mixins). *A* mixin $m \in \mathcal{M}$ *is a distinguished Coq record with one or more parameters. The first parameter must be a* `(T : Type)`*, while the other parameters are mixins* $(m_i)_{i\in\{1,\ldots,n\}}$*, each of which is applied to* `T` *and possibly previous mixin variables. I.e.*

$$\texttt{Record}\ m\ \texttt{(T : Type)}\ \overrightarrow{(p : m\ T\ p_\sigma)}^n\ \texttt{: Type := \{ .. \}.}$$

where $\overrightarrow{(p : m\ T\ p_\sigma)}^n = (p_1 : m_1\ T) \ldots (p_n : m_n\ T\ p_{\sigma(n,1)}\ \ldots\ p_{\sigma(n,q_n)})$ *and where for all* $i \in \{1, \ldots, n\}$ *we have* $q_i \in \mathbb{N}$ *and the arguments of* $m_i \in \mathcal{M}$ *consist in* q_i *of the previously quantified mixin parameters* p_k*, i.e. for all* $k \in \{1, \ldots, q_i\}$*, we have* $\sigma(i, k) \in \{1, \ldots, i - 1\}$*.*

▶ **Definition 2** (dep, mixin dependencies). *Given* $m \in \mathcal{M}$*, we define* $dep(m) \in \wp(\mathcal{M})$ *as the set of all mixins that occur as parameters of* m*, i.e.* $dep(m) = \{m_1, \ldots, m_n\}$*.*

▶ Remark 3. For all $i \in \{1, \ldots, n\}$, we have $dep(m_i) = \left\{m_{\sigma(i,1)}, \ldots, m_{\sigma(i,q_i)}\right\}$.

▶ **Definition 4.** *If* $f : \mathcal{X} \to \wp(\mathcal{Y})$*, then* $f^\star : \wp(\mathcal{X}) \to \wp(\mathcal{Y})$ *is defined as* $f^\star(X) = \bigcup_{x\in X} f(x)$*.*

▶ **Proposition 5** (dep is transitively closed and describes a DAG).

$$\forall m \in \mathcal{M}, \qquad dep^\star(dep^\star(m)) \subseteq dep^\star(m) \quad \text{and} \quad m \notin dep(m).$$

Proof. Indeed dep is transitively closed because records are well typed in the empty context and describes a DAG since Coq does not admit circular definitions. ◀

▶ **Definition 6** (factories $\mathcal{F}$). *A* factory $f \in \mathcal{F}$ *is a distinguished Coq record or definition with one or more parameters. The first parameter must be a* `(T : Type)`*, while the other parameters are* n *mixins, applied to* `T` *and previous mixin variables. I.e.*

$$\texttt{Record (* or Definition *)}\ f\ \texttt{(T : Type)}\ \overrightarrow{(p : m\ T\ p_\sigma)}^n\ \texttt{: Type := ...}$$

▶ **Definition 7** ($requires$, factory requirements.). *Given* $f \in \mathcal{F}$*, we define* $requires(f) \in \wp(\mathcal{M})$ *as the set of all mixins that occur as parameters of* f*, i.e.* $dep(f) = \{m_1, \ldots, m_n\}$*.*

The following property holds,

▶ **Proposition 8** (*requires* is closed under *dep*)**.** $\forall f \in \mathcal{F}$, $\mathit{dep}^\star(\mathit{requires}(f)) \subseteq \mathit{requires}(f)$.

Proof. Because Coq records and global definitions are well typed in the empty context. ◀

▶ **Definition 9** (builders $\mathcal{B}$)**.** *A* builder $\mu \in \mathcal{B}$ *is a distinguished function whose return type is a mixin* $m_{n+1} \in \mathcal{M}$ *and whose parameters are the carrier type* `(T : Type)`*, mixins* $\{m_1, \ldots, m_n\} \in \wp(\mathcal{M})$ *and a* factory $f \in \mathcal{F}$*, such that* $\mathit{requires}(f) = \{m_1, \ldots, m_n\}$*. I.e.*

```
Definition $\mu$ (T : Type) $\overrightarrow{(p : m\ T\ p_\sigma)}^n$ : $f\ T\ p_1 \ldots p_n$ -> $m_{n+1}\ T\ p_{\sigma(n+1,1)}$ ... $p_{\sigma(n+1,q_{n+1})}$ := ...
```

Note that the builders of a given factory have the same set of dependencies.

▶ **Definition 10** (from)**.** *We define* $\mathsf{from}(f, m_{n+1}) = \mu$ *to be the (unique) builder for* m_{n+1} *from the factory* f*, when it exists.*

Note that from is not a total function, and that $\mathsf{from}(f, m)$ is defined if and only if there is a declared builder that shows how to build m from f. In this case, we say f provides m.

▶ **Definition 11** (*provides*)**.** $\mathit{provides}(f) = \{m \mid \mathsf{from}(f, m)$ *is defined*$\} \in \wp(\mathcal{M})$ *the set of mixins that a factory* $f \in \mathcal{F}$ *provides, through its builders.*

Mixins are declared by the user as the fundamental building blocks of a hierarchy. As the next proposition shows they shall not be regarded as different from regular factories, since they are a special case.

▶ **Proposition 12** ($\mathcal{M} \subseteq \mathcal{F}$)**.** *There is a way to see mixins as factories.*

Proof. For all $m \in \mathcal{M}$ we have $\mathit{requires}(m) = \mathit{dep}(m)$, $\mathit{provides}(m) = \{m\}$ and $\mathsf{from}(m, m) = \left(\texttt{fun T } \overrightarrow{(p : m\ T\ p_\sigma)}^n \texttt{ (x : } m \texttt{ T } p_1 \ \ldots p_n\texttt{) => x}\right) \in \mathcal{B}$. ◀

▶ **Coq command** to declare a new mixin:

```
HB.mixin Record M T of $f_1$ ... $f_n$ := { .. }.
```

Declares the record `axioms` inside a module `M`. This record `M.axioms` depends on the mixins $\mathit{requires}(\texttt{M.axioms}) = \mathit{dep}(\texttt{M.axioms}) = \mathit{provides}^\star(\{f_1, \ldots, f_n\})$ and is registered both as a mixin and a factory. Finally It exports an abbreviation `M.Build` and a notation `M` standing for `M.axioms`, so that the module name can be used to denote the `axioms` record it contains.

▶ **Coq command** to declare an instance: `HB.instance` $X\ b_1 \ldots b_k$, synthesizes terms corresponding to all the mixins that can be built from the b_i. Indeed if $b_i : f_i\ T\ \ldots$, then this command creates elements of types $\mathit{provides}^\star(\{f_1, \ldots, f_k\})$. This command also generates unification hints as described in Section 4.

▶ **Coq commands** to declare a new factory and generate new builders:

```
HB.factory Record F T of $f_1$ ... $f_n$ := { .. }.
HB.builders Context T (a : F T).

  Definition $b_{n+1}$ : $f_{n+1}$ T .. := ...
  HB.instance T $b_{n+1}$.
  ..
  Definition $b_{n+k}$ : $f_{n+k}$ T .. := ...
  HB.instance T $b_{n+k}$.
HB.end.
```

Declares the record `axioms` inside a module `F`. This record `F.axioms` depends on the mixins *requires*(`F.axioms`) $= \mathit{provides}^\star(\{f_1, \ldots, f_n\})$ and is registered as a factory. It exports an abbreviation `F.Build` and a notation `F` standing for `F.axioms`, so that the module name can be used to denote the `axioms` record it contains. Finally it uses the factory instances $b_{n+1}, \ldots, b_{n+k}$ provided by the user in order to derive builders, so that *provides*(`F`) $= \mathit{provides}^\star(\{f_{n+1}, \ldots, f_{n+k}\})$.
It is thus necessary that $\mathit{requires}^\star(\{f_{n+1}, \ldots, f_{n+k}\}) \subseteq \mathit{provides}^\star(\{f_1, \ldots, f_n\})$.

Note that the b_i are not builders since their return types are not necessarily mixins, but could be factories. However, since all factories provide mixins through builders, we obtain builders out of each b_i by function composition.

3.2 Classes and structures

▶ **Definition 13** ($\mathcal{C}$, class)**.** *A class* `c` $\in \mathcal{C}$ *is a distinguished Coq record with one parameter* `(T : Type)`*. The type of each field is a mixin in* $\mathcal{M}$ *applied to* `T` *and, if needed, any number of other fields:*

```
Record c (T : Type) := { p_1 : m_1 T; .. ; p_i : m_i T p_σ(i,1) ...p_σ(i,q_i); ..}.
```

▶ **Definition 14** (*def*, class definition)**.** *We call* $\mathit{def}(c) \in \wp(\mathcal{M})$ *the set of mixins mentioned in the fields of the class, i.e.* $\{m_1, \ldots, m_n\}$*. Given that class records are well typed in the empty context the set of mixin records is closed transitively. The implementation enforces that no two class records contains the same set of mixins (disregarding the order of the fields), i.e. def is injective.*

▶ **Programming invariant for** *def* For all $f \in \mathcal{F}$:

1. $\exists c \in \mathcal{C},\ \mathit{def}(c) = \mathit{requires}^\star(f) \cup \mathit{provides}^\star(f)$,
2. $\exists C \subseteq \mathcal{C},\ \mathit{def}^\star(C) = \mathit{requires}^\star(f)$.

Classes could also be seen as factories.

▶ **Proposition 15** ($\mathcal{C} \subseteq \mathcal{F}$)**.** *For all* $m \in \mathit{def}(c)$ *we have* $\mathsf{from}(c, m_i) = p_i$*, and it follows that* $\mathit{requires}(c) = \emptyset$*,* $\mathit{provides}(c) = \mathit{def}(c)$*.*

However since classes are generated, their implementation may change, thus users should not rely on constructors of classes. Hence the only way users may refer to a class is as an argument of `HB.mixin`, `HB.factory` or `HB.structure`.

▶ **Definition 16** ($\mathcal{S}$ Structure)**.** *A structure* `s` $\in \mathcal{S}$ *is a distinguished dependent pair: a Coq record where the value of the first field occurs in the type of the second. The first field is* `(sort : Type)` *and the second field is* `(class : c sort)` *for some* `c` $\in \mathcal{C}$*. As a consequence structures are in bijection with classes.*

▶ **Coq command** to declare a class and structure: `HB.structure Definition M := { A of` $f_1 \ldots f_n$ `}` crafts a class `M.class_of` $\in \mathcal{C}$ where $\mathit{def}(c) = \mathit{provides}^\star(\{f_1, \ldots, f_n\})$ and the corresponding structure `M.type` $\in \mathcal{S}$, together with unification hints as described in Section 4.

▶ **Definition 17** ($\leq\ \in \mathcal{C} \times \mathcal{C}$, subclass)**.** $c1 \leq c2$ *iff* $\mathit{def}(c2) \subseteq \mathit{def}(c1)$

3.3 Automatic inference of mixins

Since mixins may change but factories stay the same, $\mathcal{HB}$ commands must never rely on a particular set of mixins as arguments, and factory arguments must never be given explicitly by the user. As described in Sections 3.1 and 3.2, $\mathcal{HB}$ commands take a list of factories as arguments, which they expand into lists of mixins behind the scene. However factory types and constructors have mixin arguments that must be inferred automatically when used. To this end, the commands `HB.mixin` and `HB.factory` generate abbreviations for the user to replace uses of constructors of factories. These commands first create a record f_{aux} with a constructor F_{aux} and then create abbreviations f and F that automatically fill the mixin arguments of f_{aux} and F_{aux} respectively. See [17, Section 7] for a detailed description of how to implement these abbreviations in Coq.

```
Record $f_{aux}$ T $\overrightarrow{(p : m\ T\ p_\sigma)}^n$ := $F_{aux}$ { ... }.
Notation $f$ T := ($f_{aux}$ T ...(* $p_1 \dots p_n$ inferred when $T$ is known *)).
Notation $F$ T  $x_1$ ... $x_k$ := ($F_{aux}$ T ...(* $p_1 \dots p_n$ inferred when $T$ is known *) $x_1$ ... $x_k$).
Definition b : $f$ T := $F$ T $x_1$ ... $x_n$.
```

4 The target language: Coq with Packed, flat, Classes

The language of Packed Classes [13] is used directly to describe the algebraic hierarchy of the Mathematical Components library. It is based on a disciplined use of records and projections and on the possibility of extending the elaborator of Coq via the declaration of Canonical Structures [17] instances. In this section, we describe the *flat* variant of Packed Classes, the target language of `hierarchy-builder`, through the hierarchy V3 extended with semirings.

The `hierarchy-builder` addon can generate all the Coq declarations in this section automatically, but some details are omitted for brevity in this section.

4.1 Describing structures with records and projections

We describe mathematical structures with three kinds of dependent records: mixins, classes, and structures. As shown in Section 2, a mixin gathers operators and axioms newly introduced by a structure. As in [13, Section 2.4][26, Section 2], a class record is parametrized by the carrier type (`T : Type`) and gathers all the operators and axioms of a structure by assembling mixins, and a `Structure type` (a record) bundles a carrier and its class instance, as follows.

```
Module Monoid.
Record axioms (M : Type) : Type :=
  Class { Monoid_of_Type_mixin : Monoid_of_Type M; }.
Structure type : Type := Pack { sort : Type; class : axioms sort; }.
End Monoid.
```

The `Monoid` module plays the role of a name space and forces us to write qualified names such as `Monoid.type`; as a consequence we can reuse the same unqualified names for other structures, i.e., class and structure record can always be named as `axioms` and `type` respectively.

Mixins and classes are internal definitions to structures; in contrast, `Monoid.type` is part of the interface of the monoid structure. As seen in section 2, we declare `Monoid.sort` as an implicit coercion and lift monoid operators and axioms from projections for the monoid mixin to definitions and lemmas for `Monoid.type` as follows.

```
Coercion Monoid.sort : Monoid.type >-> Sortclass.

Definition zero {M : Monoid.type} : M :=
  Monoid_of_Type.zero M (Monoid.Monoid_of_Type_mixin M (Monoid.class M)).
Definition add {M : Monoid.type} : M -> M -> M :=
  Monoid_of_Type.add M (Monoid.Monoid_of_Type_mixin M (Monoid.class M)).
Lemma addrA {M : Monoid.type} : associative (@add M).
(* Two monoid axioms `add0r` and `addr0` are omitted. *)
```

Next we define Abelian groups. Since the monoid structure is the bottom of this hierarchy its class record `Monoid` consists of just one mixin. In contrast the class record of Abelian groups consists of two mixins where the second one depends on the former (since Abelian groups inherit from monoids).

```
Module AbelianGroup.
Record axioms (A : Type) : Type := Class {
  Monoid_of_Type_mixin : Monoid_of_Type A;
  AbelianGroup_of_Monoid_mixin : AbelianGroup_of_Monoid A Monoid_of_Type_mixin; }.
Structure type : Type := Pack { sort : Type; class : axioms sort }.
End AbelianGroup.
```

In the flat variant of Packed Classes, a class record gathers mixins as its fields directly as in above. In contrast, a class record of the non-flat variant packs a class instance of one of the superclasses with remaining mixins, which reduces the amount of code to implement inheritance significantly. Since we do not need to care about the amount of code in automated generation, `hierarchy-builder` uses the flat variant of Packed Classes as its target language.

As in the monoid structure, we declare `AbelianGroup.sort` as an implicit coercion and lift additive inverse `opp` and Abelian group axioms. Since Abelian groups inherit from monoids, we also declare an implicit coercion from Abelian groups to monoids. A coercion between structures can be defined in two steps: first a coercion between the class record of the superclass to the class record of the subclass (line 2); and a coercion between structure records (line 4) relying on the first one.

```
1  Coercion AbelianGroup.sort : AbelianGroup.type >-> Sortclass.
2  Coercion AbelianGroup_class_to_Monoid_class (A : Type) (c : AbelianGroup A) :
3    Monoid A := Monoid.Class A (AbelianGroup.Monoid_of_Type_mixin A c).
4  Coercion AbelianGroup_to_Monoid (A : AbelianGroup.type) : Monoid.type :=
5    Monoid.Pack (AbelianGroup.sort A) (AbelianGroup.class A).
6
7  Definition opp {A : AbelianGroup.type} : A -> A := ... .
8  (* Two Abelian group axioms `addrC` and `addNr` are omitted. *)
```

Generally, a coercion from a structure X to another structure Y can (and should) have the following form, thanks to the corresponding coercion between classes `X_class_to_Y_class`.

```
Coercion X_to_Y (T : X.type) : Y.type :=
  Y.Pack (X.sort T) ((* X_class_to_Y_class _ *) (X.class T)).
```

4.2 Multiple inheritance

In order to introduce multiple inheritance, we extend the hierarchy described in Section 2.2 with the structure of semirings as depicted in Figure 2, and name it V4. Semirings introduce the binary multiplication operator `mul` and its identity element `one`. For the sake of completeness, the full code of V4 is available in Appendix C.

```
HB.mixin Record SemiRing_of_Monoid S of Monoid S := {
  one : S;
  mul : S -> S -> S;
  (* 7 axioms are omitted. *)
}.

HB.factory Record Ring_of_AbelianGroup R of AbelianGroup R := {
  (* 2 operators and 5 axioms are omitted. *)
}.
HB.builders Context R (f : Ring_of_AbelianGroup R).
..
HB.end.
```

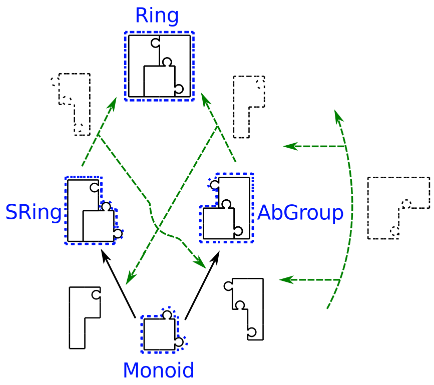


Figure 2 V4 introduces multiple inheritance. For brevity we omit the factories/builders for the upper arrows.

Since semirings and Abelian groups do not inherit from each other, the definition of semirings and Abelian groups are quite similar:

```
Module SemiRing.
Record axioms (S : Type) : Type := Class {
  Monoid_of_Type_mixin : Monoid_of_Type S;
  SemiRing_of_Monoid_mixin : SemiRing_of_Monoid S Monoid_of_Type_mixin;
}.
Structure type : Type := Pack { sort : Type; class : axioms sort; }.
End SemiRing.
```

We define implicit coercions from the semiring structure to the carrier and the monoid structure, and we lift semiring operators and axioms as follows:

```
Coercion SemiRing.sort : SemiRing.type >-> Sortclass.
Coercion SemiRing_to_Monoid : SemiRing.type >-> Monoid.type.

Definition one {S : SemiRing.type} : S := ... .
Definition mul {S : SemiRing.type} : S -> S -> S := ... .
(* 7 semiring axioms are omitted. *)
```

The class record is defined by gathering the monoid, Abelian group, and semiring mixins. Since the rings inherit from monoids, semirings, and Abelian groups, we define implicit coercions from the ring structure to those three structures.

```
Module Ring.
Record axioms (R : Type) : Type := Class {
  Monoid_of_Type_mixin : Monoid_of_Type R;
  SemiRing_of_Monoid_mixin : SemiRing_of_Monoid R Monoid_of_Type_mixin;
  AbelianGroup_of_Monoid_mixin : AbelianGroup_of_Monoid R Monoid_of_Type_mixin; }.
Structure type : Type := Pack { sort : Type; class : axioms sort; }.
End Ring.

Coercion Ring.sort : Ring.type >-> Sortclass.
Coercion Ring_to_Monoid : Ring.type >-> Monoid.type.
Coercion Ring_to_SemiRing : Ring.type >-> SemiRing.type.
Coercion Ring_to_AbelianGroup : Ring.type >-> AbelianGroup.type.
```

4.3 Linking structures and instances via Canonical Structures

We recall here how to use the Canonical Structures [17, 25] mechanism, which lets the user extend the elaborator of Coq, in order to handle inheritance and inference of structure [1, 26, 13]. This software component takes as input a term as written by the user and has to infer all the missing information that is necessary in order to make the term well typed.

A first example of elaboration that requires canonical structures is $0 + 1$. After removing all syntactic facilities, the underlying Coq term is `(add _ (zero _)) (one _)` where `_` stands for an implicit piece of information to be inferred and the constants `add` and `zero` come from the monoid structure while `one` comes from semirings. When the term is type checked each `_` is replaced by a unification variable written $?_v$. Respectively, the head and the argument of the top application can be typed as follows:

```
  add ?_M (zero ?_M) : Monoid.sort ?_M -> Monoid.sort ?_M
  one ?_SR : SemiRing.sort ?_SR
```

where $?_{\mathrm{M}}$ `: Monoid.type` and $?_{\mathrm{SR}}$ `: SemiRing.type`. In order to type check the application Coq has to unify `Monoid.sort` $?_{\mathrm{M}}$ with `SemiRing.sort` $?_{\mathrm{SR}}$, which is not trivial: it amounts at finding a structure that is both a monoid and a semiring, possibly the smallest one [26, Sect. 4]. This piece of information can be inferred from the hierarchy and its inheritance relation (Definition 17) and we can tell Coq to exploit it by declaring `SemiRing_to_Monoid : SemiRing.type -> Monoid.type` as canonical. With that hint Coq will pick $?_{\mathrm{M}}$ to be `SemiRing_to_Monoid` $?_{\mathrm{SR}}$ as the canonical solution for this unification problem. In general, all the coercions between structures must be declared as canonical.

Another example of elaboration problem is -1, which hides the term `opp _ (one _)`. Here `opp` and `one` are respectively from Abelian groups and semirings, which do not inherit each other but whose smallest common substructure is rings; thus we have to extend the unifier to solve a unification problem `AbelianGroup.sort` $?_{\mathrm{AbG}}$ = `SemiRing.sort` $?_{\mathrm{SR}}$ by instantiating $?_{\mathrm{AbG}}$ and $?_{\mathrm{SR}}$ with `Ring_to_AbelianGroup` $?_{\mathrm{R}}$ and `Ring_to_SemiRing` $?_{\mathrm{R}}$ respectively where $?_{\mathrm{R}}$ is a fresh unification variable of type `Ring.type`. This hint can be given as follows:

```
Canonical AbelianGroup_to_SemiRing (S : Ring.type) :=
  SemiRing.Pack (AbelianGroup.sort (Ring_to_AbelianGroup S)) (Ring.class S)).
```

Similarly, one can apply an algebraic theory to an instance (an example) of that structure, e.g., 2×3 where 2 and 3 have type $\mathbb{Z}$. The same mechanism of canonical structures let us extend the unifier to solve `SemiRing.type` $? = \mathbb{Z}$.

5 The implementation of $\mathcal{HB}$ in Coq-Elpi

The implementation is based on the Elpi extension language for Coq. In this section we introduce the features of the programming language that came in handy in the development of $\mathcal{HB}$ and comment a few code snippets.

Coq-Elpi [28] is a Coq plugin embedding Elpi and providing an extensive, high level, API to access and script the Coq system at the vernacular level. This API lives in the `coq` namespace and lets one easily declare records, coercions, canonical structures, modules, implicit arguments, etc. The most basic Coq data type exposed to Elpi is the one of references to global declarations:

```
kind gref  type.                 % The data type of references to global terms
type indt  inductive -> gref.    % eg: Coq.Init.Datatypes.nat
type indc  constructor -> gref.  % eg: Coq.Init.Datatypes.O
type const constant -> gref.     % eg: Coq.Init.Peano.plus
```

The arguments of the three constructors are opaque to Elpi, that can only use values of these types via dedicated APIs. For example the API for declaring an inductive type will generate a value of type `inductive` that is printed as, for example, «nat».

Elpi [11] is a dialect of λProlog [19], an higher order logic programming language that makes it easy to manipulate abstract syntax tree with binders. Coq-Elpi takes full advantage of this capability by representing Coq terms in Higher Order Abstract Syntax [20] style, reusing the binder of the programming language in order to represent Coq's ones. Here is an excerpt of the data type of Coq terms:

```
kind term type.                             % The data type of Coq terms
type global gref -> term.                   % eg: nat, O, S, plus, ...
type fun    term -> (term -> term) -> term. % eg: fun x : t => b(x)
type app    list term -> term.              % eg: app [hd|args]
... % all other term constructors are omitted for brevity
```

Note that the `fun` constructor holds a λProlog function. In this syntax the Coq term `(fun x : nat => x x)` becomes `(fun (global (indt «nat»)) x\ app[x, x])` where `x\` binds `x` in the body of the function. Substitution of a bound variable for a term can be computed by applying a term (of function type) to an argument.

Data types with binders are also used as input to high level APIs that build terms behind the scenes. For example a Coq record is just an inductive type and the API to declare one must allow the type of a field to depend on the fields that comes before it. Note that the `field` constructor takes a coercion flag, the name of the field, its type and binds a term in the remaining record declaration.

```
kind indt-decl type.             % The type of an inductive type declaration
type record      string -> term -> string -> record-decl -> indt-decl.
type field       bool -> string -> term -> (term -> record-decl) -> record-decl.
type end-record  record-decl.
... % constructors for non-record inductive types are omitted for brevity

external pred coq.env.add-indt i:indt-decl, o:inductive.
external pred coq.CS.canonical-projections i:inductive, o:list (option constant).
```

The `pred` keyword documents types and modes (input or output) of the arguments of a predicate, while `external` signals that the predicate is a builtin (in other words it is implemented in OCaml rather than λProlog).

We comment these two builtin predicates and the `indt-decl` type while looking at the code of `declare-structure` that is in charge of scripting the following Coq code:

```
Structure type : Type := Pack { sort : Type; class : axioms sort }.
```

The following Elpi code builds the declaration, type checks it, adds it to the Coq environment and finally returns the projections for the sort and the class fields.

```
pred declare-structure i:gref, o:term, o:term, o:term.

declare-structure ClassName Structure SortProjection ClassProjection :- std.do! [
  StructureDeclaration =
    record "type" {{ Type }} "Pack" (
      field tt "sort" {{ Type }} s\
      field ff "class" (app [global ClassName, s]) _\
    end-record),
  coq.typecheck-indt-decl StructureDeclaration,
  coq.env.add-indt StructureDeclaration StructureName,
  coq.CS.canonical-projections StructureName [some SortP, some ClassP],
  Structure = global (indt StructureName),
  SortProjection = global (const SortP),
  ClassProjection = global (const ClassP),
].
```

Note that the binder `s\` at line 6 lets one mention the first field in the type of the second. The syntax `{{ Type }}` is a quotation: it lets one use the syntax of Coq to write an Elpi expression of type `term`. The API `coq.CS.canonical-projections` lets us find the projections automatically generated by Coq for a given record. The last detail worth mentioning is that this program makes no use of backtracking: the `std.do!` combinator signals that.

In the simple case of the structure record, the number of fields, and hence the number of binders, is fixed. On the contrary the class record has one field per mixin and each of them can depend on the previous ones. In order to synthesize terms with binders in an inductive fashion (using a recursive predicate) λProlog lets one postulate fresh nominal constants using the `pi` operator and attach to the nominal some knowledge in the form of a clause via the `=>` operator. This process is called binder mobility: the binder is moved from the data (that we are building) to the program (the context of the current computation). This feature is key to the following code that synthesizes the declaration of the fields of the class record.

```
1  pred synthesize-fields.field-for-mixin i:mixinname, o:term.
2  pred synthesize-fields i:list mixinname, i:term, o:record-decl.
3
4  synthesize-fields [] _ Decl :- Decl = end-record.
5  synthesize-fields [M|ML] T Decl :- std.do! [
6    get-mixin-modname M ModName, Name is ModName ^ "_mixin",
7    dep1 M Deps,
8    std.map Deps synthesize-fields.field-for-mixin Args,
9    Type = app[ global M, T | Args ],
10   Decl = (field ff Name Type f\ Fields f),
11   pi m\
12     synthesize-fields.field-for-mixin M m =>
13     synthesize-fields ML T (Fields m)
14 ].
```

The first predicate `synthesize-fields.field-for-mixin` is used to link a mixin to a nominal that corresponds to the record field for that mixin. It has no clauses in the base program but some clauses are added dynamically by `synthesize-fields`.

The second predicate proceeds by recursion on the (topologically sorted) list of mixins, and terminates when the list is empty. If the list contains a mixin `M` then it crafts a `Name` for it (line 6), fetches its dependencies (line 7) and finds the (previously declared) record fields holding these mixins (line 8). The `(std.map L1 P L2)` predicate relates the two lists `L1` and `L2` point wise using the predicate `P`.

Line 9 builds the type of the field: the mixin name applied to the type (sort) and all its dependencies. Note that the `Fields` variable, representing the declaration of the next fields, is under the binder for `f` (the current field). In order to make the recursive call under that binder (line 13) and recursively process `ML` we postulate a nominal `m` (line 11) that is a term satisfying any future dependency on the current mixin (line 12) and we replace `f` by `m` in `Fields` by writing `(Fields m)`.

6 Related work

The most closely related work is the one about Packed Classes [13] on which we build. The main differences are that $\mathcal{HB}$ is a higher level language that is compiled down to (flat) Packed Classes. The systematic use of factories makes the user interface of a hierarchy stable under the insertion of structures, a property that Packed Classes lacks. Finally many of the intricacies of Packed Classes are hidden to the user by the compilation step, in particular creating all the necessary coercions and canonical structure declarations, especially in the case of diamonds or when merging several libraries, which used to cause the need for *a posteriori* validation of a hierarchy design [26]. It also opens the way to automatically detect and solve problems tied to non judgmental commutative diagrams when forgetting structure [1].

In [7] Carette and O'Connor describe the language of Theory Presentation Combinators that can be used to describe a hierarchy of algebraic structures. They focus on the categorical semantics of the language that is built upon the category of context. They do not describe any actual compilation to the language of a mainstream interactive prover, indeed they claim their language to be mostly type theory independent. We know they considered targeting type theory and the language of unification hints [4] (a superset of the one of Canonical Structures), but we could not find any written trace of that. Language-wise they provide keywords such as `combine` and `over` to share (reuse) a property when defining a new structure. For example in order to avoid restating the commutativity property when defining Abelian groups they combine a commutative monoid and a group forcing the subjacent monoid to coincide: `CommutativeGroup := combine CommutativeMonoid , Group over Monoid.` In our language $\mathcal{HB}$ the same role is played by *mixins*. A mixin lets one write once and for all a property and abstract it over types and operations so that it can be reused in all the structures that need it. One operation $\mathcal{HB}$ allows for but that does not seem to be possible in the setting of Presentation Combinators is the one of replacing an axiom with a lemma and vice versa. As shown in subsection 2.1 $\mathcal{HB}$ supports that.

The MMT system [22] provides a framework to describe formal languages in a logical framework, providing good support for binders and notations. It also provides an expressive module system to organize theories and express relations among them in the form of functors. At the time of writing it provides limited support for elaborating user input taking systematic advantage of the contents of the theories. The elaborator can be extended by the user writing Scala code, and in principle use the contents of the libraries to make sense of an incomplete expressions, but no higher level language or mechanism is provided.

The library of the Mizar system features a hierarchy of algebraic structures [15]. In spite of lacking dependent types, Mizar provides the concept of attributed types and adjectives that can be used to describe the signature of structures as one would do with a dependently typed record and their properties as one would define a conjunctive predicate. The Mizar language also provides the notion of cluster that is used to link structures: by showing that property P implies property Q one can inform automation that structures characterized by P are instances of structures characterized by Q. The foundational theory of Mizar features an extensional notion of equality that makes it easy to share the signature or the properties of structures by just requiring a proof of their equivalence that is in turn used by automation to treat equivalent structures as equal.

The concepts of factory and builder presented here is akin to the `AbstractFactory` and `Builder` pattern from "the Gang Of Four" design patterns [12] in the sense that factories are used to build an arbitrary number of objects (here mixin instances).

7 Conclusion

In this paper we design and implement $\mathcal{HB}$, a high level language to describe hierarchies of algebraic structures. The implementation of $\mathcal{HB}$ is based on the Elpi extension language for the Coq system and is available at `https://github.com/math-comp/hierarchy-builder`. The implementation amounts to approximately a thousand lines of (commented) Elpi code and tree hundred lines of Coq vernacular. It took less than one month to implement $\mathcal{HB}$, but its design took several years of attempts and fruitful discussions with the people we thank in the front page of this paper.

The $\mathcal{HB}$ language is only loosely tied to Coq or even Type Theory. We believe it could be adopted with no major change to other tools. Indeed the properties and invariants that link factories, mixins and classes are key to rule out meaningless or ambiguous sentences and are not specific to our logic setting. Moreover most logics feature packing construction similar to records.

The compilation scheme we present in section 4 is tied to dependent records, that are available in Coq but also in other provers based on Type Theory such a Matita [3] and Lean [10]. We chose the flat variant of Packed Classes as the target for $\mathcal{HB}$ because the Mathematical Components library uses Packed Classes as well: As of today Packed Classes offer the best compromise between flexibility and performance [17, Section 8] in Coq. Of course one could imagine a future were other approaches such as telescopes [21] or unbundled classes [27] would offer equal or better performances in Coq, or in another system. It is a virtue of our work to provide a user language that is separate from the implementation one. We believe it would take a minor coding effort to retarget $\mathcal{HB}$ to another bundling approach.

We leave to future work extending $\mathcal{HB}$ to support structures parametrized by structures such as the one of module over a ring. We also leave to future work the automatic synthesis of the notion of morphism between structures of the hierarchy.

References

1 Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, and Kazuhiko Sakaguchi. Competing inheritance paths in dependent type theory: a case study in functional analysis. accepted in the proceedings of IJCAR 2020, available at `https://hal.inria.fr/hal-02463336`, 2020.

2 Reynald Affeldt, David Nowak, and Takafumi Saikawa. A hierarchy of monadic effects for program verification using equational reasoning. In *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings*, pages 226–254, 2019. `doi:10.1007/978-3-030-33636-3_9`.

3 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita interactive theorem prover. In *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, pages 64–69, 2011. `doi:10.1007/978-3-642-22438-6_7`.

4 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 84–98, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

5 Gerhard Betsch. On the beginnings and development of near-ring theory. In *Near-rings and near-fields. Proceedings of the conference held in Fredericton, New Brunswick, July 18-24, 1993*, pages 1–11. Mathematics and its Applications, 336. Kluwer Academic Publishers Group, Dordrecht, 1995.

6 Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, 2015. `doi:10.1007/s11786-014-0181-1`.

7 Jacques Carette and Russell O'Connor. Theory presentation combinators. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics*, pages 202–215, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

8 Cyril Cohen. *Formalized algebraic numbers: construction and first-order theory.* PhD thesis, Ecole Polytechnique X, 2012. URL: `https://pastel.archives-ouvertes.fr/pastel-00780446`.

9 The Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.11*, October 2020. URL: `http://coq.inria.fr`.

10 Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 378–388, 2015. `doi:10.1007/978-3-319-21401-6_26`.

11 Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast, embeddable, λProlog interpreter. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, pages 460–468, 2015. doi:10.1007/978-3-662-48899-7_32.

12 Erich Gamma. *Design patterns: elements of reusable object-oriented software.* Pearson Education India, 1995.

13 François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 327–342, 2009. doi:10.1007/978-3-642-03359-9_23.

14 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 *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 163–179, 2013. doi:10.1007/978-3-642-39634-2_14.

15 Adam Grabowski, Artur Korniłowicz, and Christoph Schwarzweller. On algebraic hierarchies in mathematical repository of Mizar. In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 363–371, September 2016.

16 Johannes Hölzl, Fabian Immler, and Brian Huffman. Type classes and filters for mathematical analysis in Isabelle/HOL. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 279–294. Springer, 2013. doi:10.1007/978-3-642-39634-2_21.

17 Assia Mahboubi and Enrico Tassi. Canonical structures for the working coq user. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 19–34, 2013. doi:10.1007/978-3-642-39634-2_5.

18 The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 367—381, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372885.3373824.

19 Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic.* Cambridge University Press, New York, NY, USA, 1st edition, 2012.

20 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, page 199–208, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/53990.54010.

21 Robert Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*, 13:386–402, 2002. doi:10.1007/s001650200018.

22 Florian Rabe and Michael Kohlhase. A scalable module system. *Information and Computation*, 230:1–54, 2013. doi:10.1016/j.ic.2013.06.001.

23 Damien Rouhling. *Formalisation Tools for Classical Analysis – A Case Study in Control Theory.* PhD thesis, Université Côte d'Azur, 2019. URL: https://hal.inria.fr/tel-02333396.

24 Amokrane Saïbi. Typing algorithm in type theory with inheritance. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1997), Paris, France, 15–17 January 1997*, pages 292–301. ACM, 1997.

25 Amokrane Saïbi. *Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types. Application à la Théorie des Catégories. (Formalization of Mathematics in Type Theory. Generic tools of Modelisation and Demonstration. Application to Category Theory).* PhD thesis, Pierre and Marie Curie University, Paris, France, 1999. URL: https://tel.archives-ouvertes.fr/tel-00523810.

26 Kazuhiko Sakaguchi. Validating mathematical structures. accepted in the proceedings of IJCAR 2020, available at https://arxiv.org/abs/2002.00620, 2020.

27 Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011. doi:10.1017/S0960129511000119.

28 Enrico Tassi. Coq-Elpi, Coq plugin embedding Elpi. https://github.com/LPCIC/coq-elpi, 2020.

A Coq reference

```
Section OperationProperties.

Variables (T : Type) (e : T) (inv : T -> T) (op : T -> T -> T) (add : T -> T -> T).

Definition left_id  := forall x, op e x = x.
Definition right_id := forall x, op x e = x.

Definition left_inverse := forall x, op (inv x) x = e.
Definition right_inverse := forall x, op x (inv x) = e.

Definition commutative := forall x y, op x y = op y x.
Definition associative := forall x y z, op x (op y z) = op (op x y) z.

Definition left_distributive  := forall x y z, op (add x y) z = add (op x z) (op y z).
Definition right_distributive := forall x y z, op x (add y z) = add (op x y) (op x z).

End OperationProperties.
```

B Proof of addrC

```
Lemma addrC {R : Ring.type} : commutative (@add R).
Proof.
have innerC (a b : R) : a + b + (a + b) = a + a + (b + b).
  by rewrite -[a+b]mul1r -mulrDl mulrDr !mulrDl !mul1r.
have addKl (a b c : R) : a + b = a + c -> b = c.
  apply: can_inj (add a) (add (opp a)) _ _ _.
  by move=> x; rewrite addrA addNr add0r.
have addKr (a b c : R) : b + a = c + a -> b = c.
  apply: can_inj (add ^~ a) (add ^~ (opp a)) _ _ _.
  by move=> x; rewrite /= -addrA addrN addr0.
move=> x y; apply: addKl (x) _ _ _; apply: addKr (y) _ _ _.
by rewrite -!addrA [in RHS]addrA innerC !addrA.
Qed.
```

C Full V4 code

```
From Coq Require Import ssreflect ssrfun ZArith.
From HB Require Import structures.

Declare Scope hb_scope.
Delimit Scope hb_scope with G.
Open Scope hb_scope.

(* Bottom mixin in Fig. 2. *)
HB.mixin Record Monoid_of_Type M := {
  zero : M;
  add : M -> M -> M;
  addrA : associative add;
  add0r : left_id zero add;
  addr0 : right_id zero add;
}.
HB.structure Definition Monoid := { M of Monoid_of_Type M }.
Notation "0" := zero : hb_scope.
Infix "+" := (@add _) : hb_scope.
```

```

(* Bottom right mixin in Fig. 2. *)
HB.mixin Record AbelianGroup_of_Monoid A of Monoid A := {
  opp : A -> A;
  addrC : commutative (add : A -> A -> A);
  addNr : left_inverse zero opp add;
}.
HB.structure Definition AbelianGroup := { A of Monoid A & AbelianGroup_of_Monoid A }.
Notation "- x" := (@opp _ x) : hb_scope.
Notation "x - y" := (x + - y) : hb_scope.

(* Bottom left mixin in Fig. 2. *)
HB.mixin Record SemiRing_of_Monoid S of Monoid S := {
  one : S;
  mul : S -> S -> S;
  mulrA : associative mul;
  mul1r : left_id one mul;
  mulr1 : right_id one mul;
  mulrDl : left_distributive mul add;
  mulrDr : right_distributive mul add;
  mul0r : left_zero zero mul;
  mulr0 : right_zero zero mul;
}.
HB.structure Definition SemiRing := { S of Monoid S & SemiRing_of_Monoid S }.
Notation "1" := one : hb_scope.
Infix "*" := (@mul _) : hb_scope.

Lemma addrN {A : AbelianGroup.type} : right_inverse (zero : A) opp add.
Proof. by move=> x; rewrite addrC addNr. Qed.

(* Top right factory in Fig. 2. *)
HB.factory Record Ring_of_AbelianGroup R of AbelianGroup R := {
  one : R;
  mul : R -> R -> R;
  mulrA : associative mul;
  mul1r : left_id one mul;
  mulr1 : right_id one mul;
  mulrDl : left_distributive mul add;
  mulrDr : right_distributive mul add;
}.

(* Builder arrow from top right to bottom left in Fig. 2. *)
HB.builders Context (A : Type) (f : Ring_of_AbelianGroup A).

  Fact mul0r : left_zero zero mul.
  Proof.
  move=> x; rewrite -[LHS]add0r addrC.
  rewrite -{2}(addNr (mul x x)) (addrC (opp _)) addrA.
  by rewrite -mulrDl add0r addrC addNr.
  Qed.

  Fact mulr0 : right_zero zero mul.
  Proof.
  move=> x; rewrite -[LHS]add0r addrC.
  rewrite -{2}(addNr (mul x x)) (addrC (opp _)) addrA.
  by rewrite -mulrDr add0r addrC addNr.
  Qed.

  Definition to_SemiRing_of_Monoid := SemiRing_of_Monoid.Build A _ mul mulrA
     mul1r mulr1 mulrDl mulrDr mul0r mulr0.
  HB.instance A to_SemiRing_of_Monoid.

HB.end.
HB.structure Definition Ring := { A of AbelianGroup A & Ring_of_AbelianGroup A }.
```

```
(* Top left factory in Fig. 2. *)
(* It is an exact copy of the bottom right mixin. *)
HB.factory Definition Ring_of_SemiRing R of SemiRing R := AbelianGroup_of_Monoid R.
(* The corresponding builder is the identity. *)
HB.builders Context (R : Type) (f : Ring_of_SemiRing R).
  Definition to_AbelianGroup_of_Monoid : AbelianGroup_of_Monoid R := f.
  HB.instance R to_AbelianGroup_of_Monoid.
HB.end.

(* Right-most factory in Fig. 2. *)
HB.factory Record Ring_of_Monoid R of Monoid R := {
  one : R;
  opp : R -> R;
  mul : R -> R -> R;
  addNr : left_inverse zero opp add;
  addrN : right_inverse zero opp add;
  mulrA : associative mul;
  mul1r : left_id one mul;
  mulr1 : right_id one mul;
  mulrDl : left_distributive mul add;
  mulrDr : right_distributive mul add;
}.

HB.builders Context (R : Type) (f : Ring_of_Monoid R).

  Lemma addrC : commutative (add : R -> R -> R).
  Proof. (* Exactly the same as in Appendix B. *)
  have innerC (a b : R) : a + b + (a + b) = a + a + (b + b).
    by rewrite -[a+b]mul1r -mulrDl mulrDr !mulrDl !mul1r.
  have addKl (a b c : R) : a + b = a + c -> b = c.
    apply: can_inj (add a) (add (opp a)) _ _ _.
    by move=> x; rewrite addrA addNr add0r.
  have addKr (a b c : R) : b + a = c + a -> b = c.
    apply: can_inj (add ^~ a) (add ^~ (opp a)) _ _ _.
    by move=> x; rewrite /= -addrA addrN addr0.
  move=> x y; apply: addKl (x) _ _ _; apply: addKr (y) _ _ _.
  by rewrite -!addrA [in RHS]addrA innerC !addrA.
  Qed.

  (* Builder to the bottom right mixin. *)
  Definition to_AbelianGroup_of_Monoid :=
    AbelianGroup_of_Monoid.Build R opp addrC addNr.
  HB.instance R to_AbelianGroup_of_Monoid.

  (* Builder to the top right factory, which is compiled to the bottom left mixin. *)
  Definition to_Ring_of_AbelianGroup := Ring_of_AbelianGroup.Build R one mul
    mulrA mul1r mulr1 mulrDl mulrDr.
  HB.instance R to_Ring_of_AbelianGroup.

HB.end.
```

The New Rewriting Engine of Dedukti

Gabriel Hondet

Université Paris-Saclay, ENS Paris-Saclay, CNRS, Inria, Laboratoire Spécification et Vérification, Gif-sur-Yvette, France
http://www.lsv.fr/~hondet/

Frédéric Blanqui

Université Paris-Saclay, ENS Paris-Saclay, CNRS, Inria, Laboratoire Spécification et Vérification, Gif-sur-Yvette, France
http://rewriting.gforge.inria.fr/

Abstract

Dedukti is a type-checker for the λΠ-calculus modulo rewriting, an extension of Edinburgh's logical framework LF where functions and type symbols can be defined by rewrite rules. It therefore contains an engine for rewriting LF terms and types according to the rewrite rules given by the user. A key component of this engine is the matching algorithm to find which rules can be fired. In this paper, we describe the class of rewrite rules supported by Dedukti and the new implementation of the matching algorithm. Dedukti supports non-linear rewrite rules on terms with binders using higher-order pattern-matching as in Combinatory Reduction Systems (CRS). The new matching algorithm extends the technique of decision trees introduced by Luc Maranget in the OCaml compiler to this more general context.

2012 ACM Subject Classification Theory of computation → Equational logic and rewriting; Theory of computation → Operational semantics

Keywords and phrases rewriting, higher-order pattern-matching, decision trees

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.35

Category System Description

Related Version https://hal.inria.fr/hal-02317471

Supplementary Material https://github.com/deducteam/lambdapi.git

Acknowledgements The authors thank Bruno Barras and Rodolphe Lepigre for their help in developing the new rewriting engine of Dedukti.

1 Introduction

Dedukti is primarily a type-checker for the so-called λΠ-calculus modulo rewriting, $\lambda\Pi/R$, an extension of Edinburgh's logical framework LF [9] where function and type symbols can be defined by rewrite rules. This means that Dedukti takes as input type declarations and rewrite rules, and check that expressions are well typed modulo these rewrite rules and the β-reduction of λ-calculus.

The λΠ-calculus is the simplest type system on top of the pure untyped λ-calculus combining both the usual simple types of (functional) programming (*e.g.* the type $\mathbb{N} \to \mathbb{N}$ of functions from natural numbers to natural numbers) with value-dependent types (*e.g.* the type $\Pi n : \mathbb{N}, V(n)$ of vectors of some given dimension). In fact, a simple type $A \to B$ is just a particular case of dependent type $\Pi x : A, B$ where x does not occur in B. Syntactically, this means that types are not defined prior to terms as usual, but that terms and types are mutually defined.

Moreover, in $\lambda\Pi/R$, a term of type A is also seen as a term of type B if A and B are equivalent not only modulo β-reduction but also modulo some user-defined rewrite rules R. Therefore, to check that a term t is of type A, one has to be able to check when two expressions are equivalent modulo β-reduction and rewrite rules. This is why there is a rewriting engine in Dedukti.

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 35; pp. 35:1–35:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Thanks to the Curry-Howard correspondence between λ-terms and proofs on the one hand, and (dependent) types and formulas on the other hand, Dedukti can be used as a proof checker. Hence, in recent years, many satellite tools have been developed in order to translate to Dedukti proofs generated by automated or interactive theorem provers: Krajono for Matita, Coqine for Coq, Holide of OpenTheory (HOL Light, HOL4), Focalide for Focalize, Isabelle, Zenon, iProverModulo, ArchSAT, etc. [1].

By unplugging its type verification engine to only retain its rewriting engine, Dedukti can also be used as a programming language. Thanks to its rewriting capabilities, Dedukti can be used to apply transformation rules on terms and formulas with binders [18, 4].

A rewrite rule is nothing but an oriented equation [2]. Rewriting consists in applying some set of rewrite rules R (and β-reduction) as long as possible so as to get a term in (weak head) normal form. At every step it is therefore necessary to check whether a term matches some left-hand side of a rule of R. It is therefore important to have an efficient algorithm to know whether a rule is applicable and select one:

▶ **Example 1.** Consider the following rules in the new Dedukti syntax (pattern variables are prefixed by `$` to avoid name clashes with other symbols):

```
rule f (c (c $x)) a ↪ $x
with f        $x    b ↪ $x
```

To select the correct rule to rewrite a term, the naive algorithm matches the term against each rule left-hand side from the top rule to the bottom one. Let us apply the algorithm on the matching of the term `t = f (c (c e)) b`.

The first argument of `t` is matched against the first argument of the first left-hand side `c (c $x)`. As `c (c e)` matches `c (c $x)`, it succeeds. However, when we pass to the second argument, `b` does not match the pattern `a`. So the second rule is tried. Pattern `$x` filters successfully `c (c e)`, and `b` matches `b`, so it succeeds.

Yet, matching `c (c e)` against `c (c $x)` can be avoided. Indeed, if we start by matching the second argument of `f`, then the first rule is rejected in one comparison. The only remaining work is to match `c (c e)` against `$x`.

In [14], Maranget introduces a domain-specific language of so-called decision trees for describing matching algorithms, and a procedure for compiling some set of rewrite rules into this language. But his language and compilation procedure handle rewrite systems whose left-hand sides are linear constructor patterns only. In Dedukti, as we are going to see it soon, we use a more general class of patterns containing defined symbols and λ-abstractions. They can also be non-linear and contain variable-occurrence conditions as in Klop's Combinartory Reduction Systems (CRS) [11].

In this paper, we describe an extension of Maranget's work to this more general setting, and present some benchmark.

Outline of the paper. In Section 2, we start by giving examples of the kind of rewrite rules that can be handled by Dedukti, before giving a more formal definition. In Section 3, we present our extension of Maranget's decision trees, their syntax and semantics, and how to compile a set of rewrite rules into this language. In Section 4, we compare this new implementation with previous ones and other tools implementing rewriting. Finally, in Section 5, we discuss some related work and conclude.

2 Rewriting in Dedukti

We will start by providing the reader with various examples of rewrite rules accepted by Dedukti before giving a formal definition. To this end, we will use the new Dedukti syntax (see `https://github.com/Deducteam/lambdapi`). In this new syntax, one can use Unicode characters, some function symbols can be written in infix positions and, in rewrite rules, pattern variables need to be prefixed by `$` to avoid name clashes with function symbols. Note however that, for the sake of simplicity, we may omit some declarations.

Dedukti can of course handle the "Hello world!" example of first-order rewriting, the addition on unary natural numbers, as follows:

```
symbol N: TYPE      symbol 0: N      symbol s: N → N

symbol +: N → N → N

rule      0   + $m ↪ $m
with (s $n) + $m ↪ s ($n + $m)
```

More interestingly is the fact that, in constrast to functional programming languages like OCaml, Haskell or Coq, rule left-hand sides can overlap each other. Consequently, in Dedukti, addition on unary numbers can be more interestingly defined as follows:

```
rule       0 + $m      ↪ $m
with (s $n) + $m      ↪ s ($n + $m)
with      $m + 0       ↪ $m
with      $m + (s $n) ↪ s ($m + $n)
```

With the first definition, one has $0+t$ equivalent to t modulo rewriting, written $0+t\simeq t$, for all terms t (of type $\mathbb{N}$), but not $t+0\simeq t$. Hence, the interest of the second definition.

It is also possible to match on defined symbols and not just on constructors like in usual functional programming languages. Hence, for instance, one can add the following associativity rule on addition:

```
rule ($x + $y) + $z ↪ $x + ($y + $z)
```

Moreover, one can use non-linear patterns, that is, require the equality of some subterms to fire a rule like in:

```
rule $x + (- $x) ↪ 0
```

Therefore, Dedukti can handle any first-order rewriting system [2]. But it can also handle higher-order rewriting in the style of Combinatory Reduction Systems (CRS) [11].

The simplest example of higher-order rewriting is given by the `map` function on lists, which applies an argument function to every element of a list:

```
symbol map: (N → N) → List → List

rule map $f (cons $x $l) ↪ cons ($f $x) (map $f $l)
```

Unlike first-order rewriting, function symbols can be partially applied, including in patterns. Hence, in Dedukti, one can write the following:

```
symbol id: N → N
rule id $x ↪ $x
rule plus 0 ↪ id with plus (s $n) $m ↪ s (plus $n $m)
rule map id $l ↪ $l
```

It is also possible to match λ-abstractions as follows:

```
symbol cos: R → R
symbol sin: R → R
symbol *: (R → R) → (R → R) → (R → R)
symbol diff: (R → R) → (R → R)

rule diff(λx,sin($v[x])) ↪ diff(λx,$v[x]) * cos
```

Following the definition of CRS, in a rule left-hand side, a higher-order pattern variable can only be applied to distinct bound variables (this condition could be slightly relaxed though [13]). A similar condition appears in λProlog [15]. It ensures the decidability of matching.

It can also be used to check variable-occurrence conditions. The differential of a constant function can thus be simply defined as follows in Dedukti:

```
rule diff(λx,$v) ↪ λx,0
```

While in the rule for `sin`, we had `$v`$[x]$, meaning that the term matching `$v`$[x]$ may depend on `x`, here we have `$v` applied to no bound variables, meaning that the term matching `$v` cannot depend on x.

2.1 Terms, Patterns, Rewrite Rules and Matching, Formally

We now define more formally terms, patterns, rewrite rules and rewriting. Following [3], the terms of the $\lambda\Pi$-calculus are inductively defined as follows:

$$t, u ::= \mathtt{TYPE} \mid \mathtt{KIND} \mid x \mid f \mid tu \mid \lambda x : t,\, u \mid \Pi x : t,\, u$$

where x is a term variable, f is a function symbol, tu is the application of the function t to the term u, $\lambda x : t,\, u$ is the function mapping x of type t to u, which type is the dependent product $\Pi x : t,\, u$. The simple type $t \to u$ is syntactic sugar for $\Pi x : t,\, u$ where x is any fresh term variable not occurring in u.

In $\lambda x : t,\, u$ and $\Pi x : t,\, u$, the occurrences of x in u are bound, and terms equivalent modulo renaming of their bound variables are identified, as usual. In Dedukti, this is implemented by using the Bindlib library [12].

A (possibly empty) ordered sequence of terms $t_1, \ldots, t_n$ is written $\vec{t}$ for short.

Patterns are inductively defined as follows:

$$p ::= \$x\,[\vec{y}] \mid f\vec{p} \mid \lambda y,\, p$$

where $\$x$ is a pattern variable and $\vec{y}$ is a sequence of distinct bound variables.

A rewrite rule is a pair of terms, written $\ell \to r$, such that ℓ is a pattern of the form $f\vec{p}$ and every pattern variable occurring in r also occurs in ℓ.

In the following, we will assume given a set of user-defined rewrite rules R.

Matching a term t against a pattern p whose bound variables are in the set V, written $p \preceq_V t$ is inductively defined as follows:

$$\begin{array}{rll}
\$x\,[\vec{y}] \preceq_V t & \text{iff } \mathrm{FV}(t) \cap V \subseteq \{\vec{y}\} & \text{(MatchFv)} \\
\mathtt{f}\,p_1 \ldots p_n \preceq_V \mathtt{f}\,t_1 \ldots t_n & \text{iff } (p_1 \ldots p_n) \preceq_V (t_1 \ldots t_n) & \text{(MatchSymb)} \\
\lambda y, p \preceq_V \lambda y : A, t & \text{iff } p \preceq_{V \uplus \{y\}} t & \text{(MatchAbst)} \\
(p_1 \ \ldots \ p_n) \preceq_V (t_1 \ \ldots \ t_n) & \text{iff } \forall i, p_i \preceq_V t_i \wedge \forall j, p_i = p_j \Rightarrow t_i = t_j & \text{(MatchTuple)}
\end{array}$$

and we say that the term t *matches* the pattern p or that the pattern p *filters* the term t.

The indexing set V of variables is used to record which binders have been traversed, which is necessary to perform variable-occurrence tests.

The condition in the (MatchTuple) rule translates non-linearity conditions: if a variable occurs twice in a pattern, then the matching values must be equal.

3 Implementing Matching With Decision Trees

The rewriting engine described in this paper is based on the work of Maranget [14]. Maranget introduces a domain-specific language for matching and an algorithm to transform a (ordered) list of first-order linear constructor patterns into a program in this language. In this section, we explain how we extend Maranget's language and compilation procedure to our more general setting with non-linear higher-order patterns, partially applied function symbols, and no order on patterns.

We start by defining the language of decision trees D and switch case lists L:

$$\begin{aligned}
D, E &::= \mathtt{Leaf}(r) \mid \mathtt{Fail} \mid \mathtt{Swap}_i(D) \mid \mathtt{Store}(D) \mid \mathtt{Switch}(L) \\
&\quad\mid \mathtt{BinNl}(D, \{i, j\}, E) \mid \mathtt{BinCl}(D, (n, X), E) \\
L &::= (s, D)::L \mid (\lambda, D)::L_\lambda \mid T \\
L_\lambda &::= (s, D)::L_\lambda \mid T \\
T &::= (*, D)::\mathtt{nil} \mid \mathtt{nil}
\end{aligned}$$

where r is a rule right-hand side, i, j and n are integers, X is a finite set of variables. For case lists, s is a function symbol annotated with the number of arguments it is applied to, :: is the cons operator on lists and `nil` is the empty list.

An element of a switch case list is a pair mapping:

- a function symbol s to a tree for matching its arguments,
- a λ to a tree for matching the body of an abstraction,
- a default case $*$ to a tree for matching the other arguments.

Note that a list L_λ has no element (λ, D) and, in a list L, there is at most one element of the form (λ, D). Finally, in both cases, there is at most one element of the form $(*, D)$ and, if so, it is the last one (default case).

Semantics. Decision trees are evaluated along with a stack of terms $\vec{v}$ to filter and an array $\vec{s}$ used by the decision tree to store elements. Informally, the semantics of each tree constructor is as follows:

$\mathtt{Leaf}(r)$ matching succeeds and yields right-hand side r.

$\mathtt{Fail}$ matching fails.

$\mathtt{Swap}_i(D)$ moves the ith element of $\vec{v}$ to the top of $\vec{v}$ and carries on with D.

$\mathtt{Store}(D)$ stores the top of the stack into $\vec{s}$ and continues with D.

$\mathtt{Switch}(L)$ branches on a tree in L depending on the term on top of $\vec{v}$.

$\mathtt{BinNl}(D, \{i, j\}, E)$ checks whether s_i and s_j are equal and continues with D if this is the case, and E otherwise.

$\mathtt{BinCl}(D, (n, X), E)$ checks whether $\mathrm{FV}(s_n) \subseteq X$ and continues with D if this is the case, and with E otherwise.

▶ **Example 2.** The matching algorithm described in Example 1 can be represented by the following decision tree:

$$\begin{aligned}
&\mathtt{Swap}_2(\mathtt{Switch}([(\mathrm{a}_0, \mathtt{Switch}([(\mathrm{c}_0, \mathtt{Switch}([(\mathrm{c}_0, \mathtt{Leaf}(\$x))]))]));\\
&\qquad\qquad\qquad (\mathrm{b}_0, \mathtt{Leaf}(\$x))]))
\end{aligned}$$

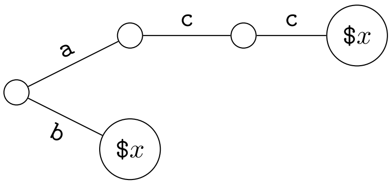


Figure 1 Graphical representation of the decision tree of Example 2.

which can be graphically represented as follows: where leaves are the right-hand sides of the rewrite rules and a path from the root to a leaf is a successful matching. The tree of Figure 1 can be used to rewrite any term of the form f $\vec{t}$. The sequence of operations to filter the term f (f a) b can be read from the tree. The initial vector $\vec{v}$ is $\vec{v} = (\texttt{f a}, \texttt{b})$ and the array $\vec{s}$ won't be necessary here.

1. The $\texttt{Swap}_2$ transforms $\vec{v}$ into $(\texttt{b}, \texttt{f a})$, so the next operations will be carried out on b.
2. The Switch node with the case list $[(\texttt{a}_0, D), (\texttt{b}_0, E)]$ allows to branch on D or E depending on the term on top of $\vec{v}$, that is, b. Since b is applied to no argument, it matches $\texttt{b}_0$ and filtering continues on E. The stack is now $\vec{v} = (\texttt{f a})$.
3. Node E is in fact a Leaf and so the matching succeeds.

Note that the top symbol f is not matched. Top symbols are analysed prior to filtering as they are needed to get the appropriate decision tree to filter the arguments.

The formal semantics is given in Figure 2. An evaluation is written as a judgement $\vec{v}, \vec{s}, V \vdash D \rightsquigarrow r$ which can be read: "stack $\vec{v}$, store $\vec{s}$ and abstracted variables V yield the term r when matched against tree D". We overload the comma notation, using it for the cons $(\texttt{s}, \vec{v})$ and the concatenation $(\vec{v}, \vec{w})$. The $|$ is used as the alternative.

Matching succeeds with the Match rule. Terms are memorised on the stack $\vec{s}$ using the Store rule. Matching on a symbol is performed with the SwitchSymb rule. If the stack has a term f applied on top and the switch-case list L contains an element $(\texttt{f}, D)$, then the symbol f can be removed, and matching continues using sub-tree D. The rule SwitchDefault allows to match on any symbol or abstraction, provided that the switch-case list L has a default case (and that we can apply neither rule SwitchSymb nor SwitchAbst). The binary constraint rules guide the matching depending on failure or success of the constraints. The last three rules allow to search for a symbol in a switch-case list. A judgement $s \vdash L \rightsquigarrow p$ reads "looking for symbol s in list L yields pair p". Cont skips a cell of the list, Default returns unconditionally the default cell of the list (which is the last by construction) and Found returns the cell that matches the symbol looked for.

3.1 Matrix Representation of Rewrite Systems

In order to compile a set of rewrite rules into this language, it is convenient to represent rewrite systems as tuples containing a matrix and three vectors. The matrix contains the patterns and can have lines of different lengths because function symbols can be partially applied. The vectors contain the right-hand side of the rewriting system and the constraints. Hence, a rewrite system for a function symbol f, that is, a set of rewrite rules $f\vec{p}^1 \rightarrow r^1, \dots, f\vec{p}^m \rightarrow r^m$ is represented by:

$$\left(\begin{bmatrix} p_1^1 & \cdots & p_{n_1}^1 \\ p_1^2 & \cdots & p_{n_2}^2 \\ & \vdots & \\ p_1^m & \cdots & p_{n_m}^m \end{bmatrix}, \begin{bmatrix} N_1 \\ N_2 \\ \vdots \\ N_m \end{bmatrix}, \begin{bmatrix} C_1 \\ C_2 \\ \vdots \\ C_m \end{bmatrix}, \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_m \end{bmatrix}\right)$$

$$\textsc{Match}\ \frac{}{\vec{v};\vec{s};V \vdash \texttt{Leaf}(k) \rightsquigarrow k} \qquad \textsc{Swap}\ \frac{(v_i,\dots,v_1,\dots,v_n);\vec{s};V \vdash D \rightsquigarrow k}{(v_1,\dots,v_i,\dots v_n);\vec{s};V \vdash \texttt{Swap}_i(D) \rightsquigarrow k}$$

$$\textsc{Store}\ \frac{\vec{v};\vec{s}\,v_1;V \vdash D \rightsquigarrow k}{\vec{v};\vec{s};V \vdash \texttt{Store}(D) \rightsquigarrow k} \qquad \textsc{SwitchSymb}\ \frac{\mathtt{f} \vdash L \rightsquigarrow (\mathtt{f},D) \qquad (\vec{w},\vec{v});\vec{s};V \vdash D \rightsquigarrow k}{(\mathtt{f}\ \vec{w},\vec{v});\vec{s};V \vdash \texttt{Switch}(L) \rightsquigarrow k}$$

$$\textsc{SwitchDefault}\ \frac{(\mathtt{s}_\ell \mid \lambda) \vdash L \rightsquigarrow (*,D) \qquad \vec{v};\vec{s};V \vdash D \rightsquigarrow k}{((\mathtt{s}\,w_1 \cdots w_\ell \mid \lambda x,\,w),\vec{v});\vec{s};V \vdash \texttt{Switch}(L) \rightsquigarrow k}$$

$$\textsc{SwitchAbst}\ \frac{\lambda \vdash L \rightsquigarrow (\lambda,D) \qquad (w,\vec{v});\vec{s};V \cup \{x\} \vdash D \rightsquigarrow k}{(\lambda x,\,w,\vec{v});\vec{s};V \vdash \texttt{Switch}(L) \rightsquigarrow k}$$

$$\textsc{BinClSucc}\ \frac{\mathrm{FV}(s_i) \cap X \subseteq V \qquad \vec{v};\vec{s};V \vdash D \rightsquigarrow k}{\vec{v};\vec{s};V \vdash \texttt{BinCl}(D,\{i,j\},E) \rightsquigarrow k} \qquad \textsc{BinClFail}\ \frac{\mathrm{FV}(s_i) \cap X \not\subseteq V \qquad \vec{v};\vec{s};V \vdash E \rightsquigarrow k}{\vec{v};\vec{s};V \vdash \texttt{BinCl}(D,(i,X),E) \rightsquigarrow k}$$

$$\textsc{BinNlSucc}\ \frac{s_j = s_j \qquad \vec{v};\vec{s};V \vdash D \rightsquigarrow k}{\vec{v};\vec{s};V \vdash \texttt{BinNl}(D,(i,X),E) \rightsquigarrow k} \qquad \textsc{BinNlFail}\ \frac{s_i \neq s_j \qquad \vec{v};\vec{s};V \vdash E \rightsquigarrow k}{\vec{v};\vec{s};V \vdash \texttt{BinNl}(D,(i,X),E) \rightsquigarrow k}$$

$$\textsc{Found}\ \frac{}{s \vdash (s,D) :: L \rightsquigarrow (s,D)} \qquad \textsc{Default}\ \frac{}{s \vdash (*,D) \rightsquigarrow (*,D)} \qquad \textsc{Cont}\ \frac{s \neq s' \qquad s \vdash L \rightsquigarrow (s|*,D)}{s \vdash (s',D) :: L \rightsquigarrow (s|*,D)}$$

Figure 2 Evaluation of decision trees.

where C_i encodes the variable-occurrence constraints in $\vec{p^i}$ and N_i encodes non-linearity constraints in $\vec{p^i}$.

A variable-occurrence constraint given by a pattern variable $\$x\,[\vec{y}]$ is encoded as a pair $(a,\vec{y})$ where a is the position of the variable in the main term.

Non-linearity constraints between two terms at positions a and b are encoded by the unordered pair $\{a,b\}$.

In the above matrix, we can then replace a pattern of the form $\$x\,[\vec{y}]$ or $\$x$ by $\_$.

For the sake of completeness, we recall the definition of positions:

▶ **Definition 3** (Positions in a term)**.** *The set of* positions *of a term t is the set of words over the alphabet of positive integers inductively defined as follows:*

- $\mathcal{P}os(x) \triangleq \{\epsilon\}$
- $\mathcal{P}os(\mathtt{f}\ t_1\ \cdots\ t_n) \triangleq \{\epsilon\} \cup \bigcup_{i=1}^{n}\{ia \mid a \in \mathcal{P}os(t_i)\}$
- $\mathcal{P}os(\lambda x,\,t) \triangleq \{\epsilon\} \cup \{1a \mid a \in \mathcal{P}os(t)\}$

The position ϵ is called the root position *of the term t and the symbol at this position is called the* root symbol *of t.*

For $a \in \mathcal{P}os(t)$, the subterm of t at position a*, denoted by $t|_a$, is defined by induction on the length of a, $t|_\epsilon \triangleq t$ and $\mathtt{f}\ t_1\ \cdots\ t_n|_{ia} \triangleq t_i|_a$*

The notion of position is extended to sequences of terms by taking $\vec{t}|_{ia} \triangleq t_i|_a$.

▶ **Example 4.** The rewrite system

```
rule f a  (λx,λy,$g[x]) ↪ 0
with f $x $x            ↪ 1
with f a  b             ↪ 2
```

is represented by the following matrix:

$$\left(\begin{bmatrix} \texttt{a} & \lambda x,\, \lambda y,\, \_ \\ \_ & \_ \\ \texttt{a} & \texttt{b} \end{bmatrix}, \begin{bmatrix} \emptyset \\ \{\{1,2\}\} \\ \emptyset \end{bmatrix}, \begin{bmatrix} \{(211,(x))\} \\ \emptyset \\ \emptyset \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}\right).$$

The variable-occurence constraint of the first rule is encoded by $(211,(x))$ since only the variable x is authorised in $\$g\,[x]$. The non-linearity constraint `f &x &x` is translated by $\{1,2\}$, hence the constraints set $\{\{1,2\}\}$.

3.2 Compiling Rewrite Systems to Decision Trees

We will describe the compilation process as a non-deterministic recursively defined relation $\rhd$ between matrices and decision trees.

To this end, we use the transformations on matrices defined in Table 1.

- $\mathsf{Spec}\left(\texttt{f}, a, \left(P, \vec{N}, \vec{C}, \vec{r}\right)\right)$ keeps rows whose first pattern filters the application of function `f` a arguments:

  ▶ **Example 5.** Let $P = \begin{bmatrix} \texttt{r}\ \$x & \texttt{q} \\ \texttt{r} & \texttt{f}\ \$x \\ \$x & \texttt{r} \\ \lambda x,\ \$x & \lambda x,\ \texttt{r} \end{bmatrix}$. Then,

  $$\mathsf{Spec}\left(\texttt{r}, 1, \left(P, \vec{N}, \vec{C}, \vec{r}\right)\right) = \left(\begin{bmatrix} \$x & \texttt{q} \\ \_ & \texttt{r} \end{bmatrix}, \vec{N}, \vec{C}, \begin{bmatrix} r_1 \\ r_3 \end{bmatrix}\right)$$

- $\mathsf{Spec}_\lambda\left(\left(P, \vec{N}, \vec{C}, \vec{r}\right)\right)$ keeps rows whose first pattern filters a λ-abstraction:

  ▶ **Example 6.** Let P be the same as in Example 5.

  $$\mathsf{Spec}_\lambda\left(P, \vec{N}, \vec{C}, \vec{r}\right) = \left(\begin{bmatrix} \_ & \texttt{r} \\ \$x\,[x] & \lambda x,\ \texttt{r} \end{bmatrix}, \vec{N}, \vec{C}, \begin{bmatrix} r_3 \\ r_4 \end{bmatrix}\right)$$

- $\mathsf{Def}\left(P, \vec{N}, \vec{C}, \vec{r}\right)$ keeps rows whose first pattern is a pattern variable:

  ▶ **Example 7.** Let P be the same as in Example 5.

  $$\mathsf{Def}\left(P, \vec{N}, \vec{C}, \vec{r}\right) = \left([\texttt{r}], \vec{N}, \vec{C}, [r_3]\right)$$

To sum up, given a pattern matrix P, a simplification function removes rows of P that are not compatible with some assumption on the form of the first pattern.

The same idea is used for constraints. Note that we will abuse set notations and write $k \in N$ or $N \backslash \{k\}$ even if N is not a set of elements of the type of k. In that case $k \in N$ is false and $N \backslash \{k\}$ is N.

- $\mathsf{csucc}(k, (P, \vec{N}, \vec{C}, \vec{r}))$ keeps all the rows and simplify the constraint sets

  $$\mathsf{csucc}(k, (\vec{p}, N, C, r)) \triangleq (\vec{p}, N\backslash\{k\}, C\backslash\{k\}, r)$$

Table 1 Decomposition operators.

Pattern p_1^j	Rows of $\mathsf{Spec}(\mathtt{f}, a, P \to A)$	Rows of $\mathsf{Spec}_\lambda(P \to A)$	Rows of $\mathsf{Def}(P \to A)$
$\mathtt{f}\ q_1\ \cdots\ q_a$	$q_1\ \cdots\ q_a\ p_2^j\ \cdots p_n^j$	No row	No row
$\mathtt{f}\ q_1\ \cdots\ q_b$	No row if $a \neq b$	No row	No row
$\mathtt{f}\ q_1\ \cdots\ q_b$	No row	No row	No row
$\lambda x,\ q$	No row	$q\ p_2^j\ \cdots p_n^j$	No row
$\_$	$\overbrace{\_\ \cdots\ \_}^{\times a}$	$\_\ p_2^j\ \cdots\ p_n^j$	$p_2^j\ \cdots\ p_n^j$

- $\mathsf{cfail}(k, (P, \vec{N}, \vec{C}, \vec{r}))$ keeps rows that don't have k in their constraint sets

$$\mathsf{cfail}(k, (\vec{p}, N, C, r)) \triangleq \begin{cases} \text{No row} & \text{if } k \in N \text{ or } k \in C \\ (\vec{p}, N, C, r) & \text{if } k \notin N \text{ and } k \notin C \end{cases}$$

A compilation process consists in reducing the matrix step by step, compiling the sub-matrices and aggregating the sub-trees obtained using the node that corresponds to the computed sub-matrices (e.g. a `Switch` if the Spec, Def and Spec_λ sub-matrices have been computed).

To say that the matrix $\left(P, \vec{N}, \vec{C}, \vec{r}\right)$ compiles to the decision tree D, we write

$$\left(\vec{\rho}, \left(P, \vec{N}, \vec{C}, \vec{r}\right), n, \mathcal{E}\right) \rhd D$$

where:

- $\vec{\rho}$ are the positions in the term that will be matched against during evaluation.
- $\mathcal{E}$ is a map from positions to integers such that $\mathcal{E}(\rho)$ is the index in $\vec{s}$ of the subterm at position ρ used during the evaluation of decision trees. The empty map is denoted $\emptyset$.
- n is the size of the store, which is incremented each time an element is added.

We now describe the compilation process implemented in DEDUKTI:

▶ **Definition 8** (Compilation). **1.** *If the matrix P has no row ($m = 0$), then matching always fails, since there is no rule to match,*

$$\vec{\rho}, \left(\emptyset, \vec{N}, \vec{C}, \vec{r}\right), n, \mathcal{E} \rhd \mathtt{Fail} \tag{2}$$

2. *If there is a row k in P composed of unconstrained variables, matching succeeds and yields right-hand side r*

$$\left(\vec{\rho}, \begin{pmatrix} p_1^1 & \cdots & p_{n_1}^1 & N_1 & C_1 & \to & r_1 \\ & & & \vdots & & & \\ \_ & \cdots & \_ & \emptyset & \emptyset & \to & r_k \\ & & & \vdots & & & \\ p_1^m & \cdots & p_{n_m}^m & N_m & C_m & \to & r_m \end{pmatrix}, n, \mathcal{E}\right) \rhd \mathtt{Leaf}(r_k) \tag{3}$$

3. *Otherwise, there is at least one row with either a symbol or a constraint or an abstraction. We can choose to either specialise on a column or solve a constraint.*

a. *Consider a specialisation on the first column, assuming it contains at least a symbol or an abstraction.*
If ρ_1 is constrained in some N_i or C_i, then define $n' = n + 1$ and $\mathcal{E}' = \mathcal{E} \cup \{\rho_1 \mapsto n\}$. Otherwise, let $n' = n$ and $\mathcal{E}' = \mathcal{E}$.
Let Σ be the set of root symbols of the terms of the first column and k the number of arguments f is applied to. Then for each $f \in \Sigma$, we compile

$$\left((\rho_1|_1 \ \cdots \ \rho_1|_k \ \rho_2 \ \cdots \ \rho_n), \mathsf{Spec}\left(f, k, \left(P, \vec{N}, \vec{C}, \vec{r}\right)\right), n', \mathcal{E}'\right) \rhd D_{f_k}$$

Let L be the switch case list defined as (we use the bracket notation for list comprehension as the order is not important here)

$$L \triangleq [(f, D_{f_k})|f \in \Sigma]$$

If there is an abstraction in the column, the Spec_λ *sub-matrix is computed and compiled to D_λ, and an abstraction case is added to the mapping*

$$\left((\rho_1|_1 \ \rho_2 \ \cdots \ \rho_n), \mathsf{Spec}_\lambda\left(\left(P, \vec{N}, \vec{C}, \vec{r}\right)\right), n', \mathcal{E}'\right) \rhd D_\lambda$$
$$L \triangleq [(s, D_{f_k}) \mid f \in \Sigma]::(\lambda, D_\lambda)::\mathtt{nil}$$

Similarly, if the column contains a variable, the Def *sub-matrix is computed and compiled to D_*, and the mapping is completed with a default case, (the abstraction case may or may not be present)*

$$\left((\rho_2 \ \cdots \ \rho_n), \mathsf{Def}\left(\left(P, \vec{N}, \vec{C}, \vec{r}\right)\right), n', \mathcal{E}'\right) \rhd D_*$$
$$L \triangleq [(f_k, D_{f_k})|s \in \Sigma]::(\lambda, D_\lambda)::(*, D_*)::\mathtt{nil}$$

Now that the switch case list L is complete (all the symbols, the abstractions and the pattern variables are handled) and the sub-trees are defined and related to their pattern matrix, we can create the top node $\mathtt{Switch}(L)$.
Furthermore, if ρ_1 is constrained, the term must be saved during evaluation. In that case, we add a `Store` *node,*

$$\left(\vec{\rho}, \left(P, \vec{N}, \vec{C}, \vec{r}\right), n, \mathcal{E}\right) \rhd \mathtt{Store}(\mathtt{Switch}(L)).$$

Otherwise,

$$\left(\vec{\rho}, \left(P, \vec{N}, \vec{C}, \vec{r}\right), n, \mathcal{E}\right) \rhd \mathtt{Switch}(L).$$

b. *If a term has been stored and is subject to a closedness constraint, then this constraint can be checked.*
That is, for any position μ such that $\mathcal{E}(\mu)$ is defined and there is a constraint set C_i and a variable set V such that $(\mu, V) \in C_i$, we compute the sub-matrices csucc *and* cfail *and we compile them to D_s and D_f,*

$$\left(\vec{\rho}, \mathsf{csucc}\left((\mu, V), \left(P, \vec{N}, \vec{C}, \vec{r}\right)\right), n, \mathcal{E}\right) \rhd D_s$$

$$\left(\vec{\rho}, \mathsf{cfail}\left((\mu, V), \left(P, \vec{N}, \vec{C}, \vec{r}\right)\right), n, \mathcal{E}\right) \rhd D_f$$

with $(\mu, X) \in F^j$ for some row number j and we finally define

$$\left(\vec{\rho}, \left(P, \vec{N}, \vec{C}, \vec{r}\right), n, \mathcal{E}\right) \rhd \mathtt{BinCl}(D_s, (\mathcal{E}(\mu), X), D_f)$$

c. *A non linearity constraints can be enforced when the two terms involved in the constraint have been stored, that is, when there is a couple $\{\mu, \nu\}$ ($\mu \neq \nu$) such that $\mathcal{E}(\mu)$ and $\mathcal{E}(\nu)$ are defined and there is a row j such that $\{\mu, \nu\} \in N_j$. If it is the case, then compute* csucc, cfail *and compile them,*

$$\left(\vec{\rho}, \mathsf{csucc}\left(\{\mu, \nu\}, \left(P, \vec{N}, \vec{C}, \vec{r}\right)\right), n, \mathcal{E}\right) \rhd D_s$$

$$\left(\vec{\rho}, \mathsf{cfail}\left(\{\mu, \nu\}, \left(P, \vec{N}, \vec{C}, \vec{r}\right)\right), n, \mathcal{E}\right) \rhd D_f$$

and define

$$\left(\vec{\rho}, \left(P, \vec{N}, \vec{C}, \vec{r}\right), n, \mathcal{E}\right) \rhd \mathtt{BinNl}(D_s, \{\mathcal{E}(i), \mathcal{E}(j)\}, D_f)$$

4. *If column i contain either a symbol, an abstraction or a constraint, and each pattern vector of P is at least of length i, then compile $\left(\vec{\mu}, \left(P', \vec{N}, \vec{F}, \vec{r}\right), n, \mathcal{E}\right) \rhd D'$ where $\vec{\mu} = (\rho_i\, \rho_1\, \ldots\, \rho_n)$ and P' is P with column i moved to the front; to build*

$$\left(\vec{\rho}, \left(P, \vec{N}, \vec{C}, \vec{r}\right), n, \mathcal{E}\right) \rhd \mathtt{Swap}_i(D') \tag{4}$$

▶ **Example 9** (Example 1, 2 continued)**.** We consider again the rewriting system used in Example 1. We start by computing the matrices:

$$(P, \emptyset, \emptyset, \vec{r}) \triangleq \left(\begin{bmatrix} \mathtt{c}\ (\mathtt{c}\ \_) & \mathtt{a} \\ \_ & \mathtt{b} \end{bmatrix}, \begin{bmatrix} \emptyset \\ \emptyset \end{bmatrix}, \begin{bmatrix} \emptyset \\ \emptyset \end{bmatrix}, \begin{bmatrix} \$x \\ \$x \end{bmatrix}\right).$$

1. We saw that it is better to start examining the second argument, so we start by swapping columns of the matrix, $P' = \begin{bmatrix} \mathtt{a} & \mathtt{c}\ (\mathtt{c}\ \_) \\ \mathtt{b} & \_ \end{bmatrix}$, define D such that $(2\,1), (P', \emptyset, \emptyset, \vec{r}), 0, \emptyset \rhd D$. and we thus have

$$((1\,2), (P, \emptyset, \emptyset, \vec{r}), 0, \emptyset) \rhd \mathtt{Swap}_2(D).$$

2. To continue and compute D, we can match on the symbols of the first column of P' with a `Switch` node. For this, we compute
   - $P_{\mathtt{a}} = \mathsf{Spec}(\mathtt{a}, 0, P') = \begin{bmatrix} \mathtt{c}\ (\mathtt{c}\ \_) \end{bmatrix}$, and
   - $P_{\mathtt{b}} = \mathsf{Spec}(\mathtt{b}, 0, P') = \begin{bmatrix} \_ \end{bmatrix}$.

   Then we compute $D_{\mathtt{a}}$ and $D_{\mathtt{b}}$ such that $(2, (P_{\mathtt{a}}, \emptyset, \emptyset, \vec{r}), 0, \emptyset) \rhd D_{\mathtt{a}}$ and $(2, (P_{\mathtt{b}}, \emptyset, \emptyset, \vec{r}), 0, \emptyset) \rhd D_{\mathtt{b}}$. The switch case list $L \triangleq [(\mathtt{a}_0, D_{\mathtt{a}}), (\mathtt{b}_0, D_{\mathtt{b}})]$ can be defined and so the compilation step produces

$$((2\,1), (P', \emptyset, \emptyset, \vec{r}), 0, \emptyset) \rhd \mathtt{Switch}(L).$$

3. Since $P_{\mathtt{b}}$ contains only unconstrained variables, we are in the case item 2 and so we have $(1, (P_{\mathtt{b}}, \emptyset, \emptyset, \$x), 0, \emptyset) \rhd \mathtt{Leaf}(\$x)$.
4. A specialisation on $P_{\mathtt{a}}$ with respect to `c` can be performed, let $Q_{\mathtt{a}} \triangleq \mathsf{Spec}(\mathtt{c}, 0, P_{\mathtt{a}}) = \begin{bmatrix} \mathtt{c}\ \_ \end{bmatrix}$ and define E such that $(1, (Q_{\mathtt{a}}, \emptyset, \emptyset, \$x), 0, \emptyset) \rhd E$. The compilation step produces

$$(1, (P_{\mathtt{a}}, \emptyset, \emptyset, \$x), 0, \emptyset) \rhd \mathtt{Switch}([(\mathtt{c}, E)]).$$

5. Similarly, we can specialise $Q_{\mathtt{a}}$ on `c` yielding the matrix $\begin{bmatrix} \_ \end{bmatrix}$ which compiles to `Leaf`. We thus have,

$$(1, (Q_{\mathtt{a}}, \emptyset, \emptyset, \$x), 0, \emptyset) \rhd \mathtt{Switch}([(\mathtt{c}, \mathtt{Leaf}(\$x))]).$$

Table 2 Time needed to solve Sudoku and SAT formulae in seconds.

	Sudoku			DPLL-SAT	
	easy	med	hard	`2_ex`	`ok_50x80`
Dedukti2.6	0.7	7.7	8 min 43	2	10
Lambdapi1.0	1.2	13	16 min 2	1.6	10
Dedukti3.0	0.5	5.2	5 min 15	0.2	2

The soundness and completeness proofs for this compilation process can be found in [10].

We have seen that at each compilation step, several options are possible. The stack can be swapped with `Swap` to orient the filtering. If a constraint can be solved, either is is solved with a `BinNl` or `BinCl` node, or a `Switch` can be performed. These possibilities make the compilation process undeterministic. Therefore, a given matrix can be compiled to several decision trees. Maranget compares different heuristics based on the shape of patterns as well as some more complex ones. In Dedukti, since verifying constraints can involve non trivial operations (non-linearity and variable occurrence tests), constraint checking is postponed as much as possible. Regarding `Swap`, we process in priority columns that have many constructors and few constraints.

4 Results

This section compares the performance of the new rewriting engine with previous implementations of Dedukti, and other tools as well.

4.1 Hand-written examples

We consider 3 different implementations of Dedukti:

- Dedukti2.6 is the latest official release of Dedukti available on `opam`. Its matching algorithm also implements decision trees but non-linearity and variable-occurrence constraints are not integrated in decision trees. Its implementation, primarily due to Ronan Saillard [17], is available on `https://github.com/Deducteam/dedukti`.
- Lambdapi1.0 is an alternative implementation of Dedukti due to Rodolphe Lepigre [12]. It implements a naive algorithm for matching. It is available on `https://github.com/rlepigre/lambdapi/tree/fix_ho`.
- Dedukti3.0 is our new implementation of Dedukti. It adds to Lambdapi1.0 the decision trees described in this paper. It is available on `https://github.com/Deducteam/lambdapi`.

The git repository `https://github.com/deducteam/libraries` contains several handwritten Dedukti examples, including a Sudoku solver with 3 examples labelled easy, medium and hard respectively, and a DPLL-based SAT solver to decide the satisfiability of propositional logic formulae in conjunctive normal form with two example files:

- `2_ex` contains a function that when given a integer n, produces n literals named v_n and the formula $p(0) = v_0 \wedge \bigwedge_{k=1}^{n} (p(k) = p(k-1) \wedge (v_{k-1} \neq v_k))$
- `ok_50x80` contains a formula with 50 literals and 80 clauses of the form $\neg x \vee \neg y \vee \neg z$.

Because of the nature of the problems, they require a substantial amount of rewriting steps to be solved.

Table 2 shows the performance of each tool on these examples. Using decision trees increases significantly performance on Sudoku since LAMBDAPI1.0 is twice as slow as DEDUKTI2.6 which is slower than DEDUKTI3.0. SAT problems confirm that DEDUKTI3 is more efficient than LAMBDAPI1.0 and DEDUKTI2.

More benchmarks are described in [10].

4.2 Rewriting Engine Competition (REC)

The Rewriting Engine Competition[1] (REC), first organized in 2009, aims to compare rewriting engines. F. Duràn and H. Garavel revived the competition in 2018, and another study has been done in 2019 [5]. There are 14 rewriting engines tested, among which Haskell's GHC and OCAML.

REC problems are written in a specific REC syntax which is then translated into one of the 14 target languages with AWK scripts. To use REC benchmarks with DEDUKTI, a translation from HASKELL benchmarks to DEDUKTI has been implemented[2].

Our rewriting engine[3] has been compared on the problems that do not use conditional rewriting with OCAML and HASKELL. For each language, we have measured both the interpretation time with `ocaml` and `runghc`, and the compiling and running time with `ocamlopt` and `ghc`. The results are in Table 3.

We can divide our observations on classes of problems. There are 43 problems, among which 22 are solved in less that one second by at least two other solvers than DEDUKTI (the first group of the table). On these problems, our rewriting engine is in average 4 times faster than the median of the other rewriting engines. The second group contains problems on which no other tool than DEDUKTI needs more than ten seconds. On this group, DEDUKTI is in average 10 times slower than the median of the other tools. However, DEDUKTI performs better than interpreted OCAML on `add8` and better than compiled OCAML on `benchtree10`. On the last group, DEDUKTI is in average 60 times slower than other engines. Interestingly `ocamlopt` has more memory overflows than DEDUKTI (7 against 4).

5 Conclusion & Related Work

This article describes the implementation of the new rewriting engine of DEDUKTI. It extends Maranget's techniques of decision trees used in the OCaml compiler [14] to the class of non-linear higher-order patterns used in Combinatory Reduction Systems (CRS) [11]. We define the language of decision trees and how to compile a set of rewrite rules into a decision tree. We finally present some benchmarks showing good performances.

A similar algorithm had been implemented in DEDUKTI2.6 by Ronan Saillard [17]. However, Saillard's rewriting engine used decision trees for first-order linear matching and handled non-linearity and variable-occurrence constraints afterwards in a naive way. In the new implementation, these constraints are fully integrated in decision trees.

Other rewriting engine uses decision trees as well like CRSX, which is a rewrite engine for an extension of Combinatory Reduction Systems [16], and Maude under certain conditions [8], but Maude considers first-order terms only.

[1] `http://rec.gforge.inria.fr`

[2] `https://raw.githubusercontent.com/Deducteam/lambdapi/master/tools/rec_to_lp/rec_hs_to_lp.awk`

[3] with commit `a0009fdaa53b607c53ebb8d1ee3a58b8d4bb8bc1`

Table 3 Performance on REC benchmark in seconds. N/A is for out of memory. T/O is for timeout (30 minutes). The last line indicates that on the `langton7` problem, Dedukti ran out of memory, the command `runghc langton7.hs` took 533.2 seconds to finish, `ocaml langton7.ml` took 101.7 seconds, `ghc langton7.hs && ./langton7` took 66 seconds and `ocamlopt langton7.ml && ./a.out` took 39.6 seconds.

	Dedukti	runghc	ocaml	ghc	ocamlopt
revelt	0.026	0.517	0.065	0.271	0.168
check1	0.030	0.417	0.065	0.270	0.170
calls	0.017	0.416	0.065	0.271	0.168
check2	0.033	0.466	0.065	0.271	0.168
garbagecollection	0.033	0.416	0.115	0.271	0.170
fibonacci05	0.033	0.416	0.065	0.271	0.168
soundnessofparallelengines	0.033	0.467	0.065	0.271	0.168
factorial5	0.033	0.416	0.066	0.271	0.168
empty	0.033	0.416	0.065	0.271	0.170
revnat100	0.064	0.516	0.115	0.276	0.170
factorial6	0.065	0.467	0.066	0.271	0.169
tautologyhard	0.065	0.716	0.165	0.271	0.221
fibonacci18	0.115	0.466	0.065	0.275	0.170
fibonacci21	0.166	0.566	0.068	0.178	0.174
benchexpr10	0.165	0.667	0.266	0.275	0.221
benchsym10	0.165	0.667	0.266	0.275	0.221
natlist	0.165	0.868	0.266	0.275	0.220
fibonacci19	0.168	0.517	0.065	0.174	0.170
factorial7	0.215	0.467	0.065	0.275	0.170
fibonacci20	0.265	0.567	0.065	0.283	0.174
permutations6	0.366	0.868	0.115	0.299	0.182
factorial8	1.467	0.817	0.115	0.299	0.183
revnat1000	3.300	3.578	0.266	0.482	0.281
benchtree10	3.476	0.667	126.027	0.275	11.546
factorial9	N/A	2.974	N/A	0.532	0.282
permutations7	6.376	3.276	0.416	0.531	0.331
add8	10.899	N/A	12.605	0.232	N/A
benchsym20	14.335	6.581	1.067	0.734	0.381
benchexpr20	14.787	6.786	1.417	0.932	0.983
mul16	T/O	11.154	6.640	0.735	N/A
add32	T/O	19.714	8.640	0.331	N/A
benchtree20	N/A	21.333	T/O	3.301	T/O
mul32	T/O	27.562	10.444	1.638	N/A
benchsym22	54.491	23.534	2.727	1.435	0.833
benchexpr22	52.995	24.492	3.979	2.092	2.337
add16	79.472	22.697	7.140	0.303	N/A
mul8	167.500	4.479	3.771	0.331	N/A
omul32	T/O	49.863	25.251	1.885	N/A
benchtree22	N/A	101.176	T/O	10.047	T/O
revnat10000	400	244.459	6.987	19.241	4.092
omul8	797.406	5.430	3.971	0.381	N/A
langton6	N/A	377.208	75.709	38.437	24.463
langton7	N/A	533.197	101.695	66.093	39.640

Other pattern-matching algorithm are also possible, in particular using backtracking automata instead of trees, which allow to have smaller data structures. The interested reader can look at Prolog implementations or Egison (see [6] and, more particularly on the question of pattern matching, [7]).

Further useful extensions would be interesting: conditional rewrite rules (the REC database contains many files with conditional rewrite rules) and matching modulo associativity and commutativity (AC). Conditional rewriting could be implemented without too much difficulty since it would consist in extending the constraints mechanism which is modular. A prototype implementation of matching modulo AC has already been developed for Dedukti2.6 by Gaspard Férey[4] but performances are not very good yet. This new implementation could provide a better basis to implement matching modulo AC.

References

1 A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. Dedukti: a logical framework based on the λΠ-calculus modulo theory, 2019. Draft.

2 F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

3 H. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of logic in computer science. Volume 2. Background: computational structures*, pages 117–309. Oxford University Press, 1992.

4 R. Cauderlier. Tactics and Certificates in Meta Dedukti. In *Proceedings of the 9th International Conference on Interactive Theorem Proving*, Lecture Notes in Computer Science 10895, 2018.

5 Francisco Durán and Hubert Garavel. The rewrite engines competitions: A rectrospective. In *TACAS*, 2019.

6 Satoshi Egi. Egison: Non-linear pattern-matching against non-free data types. *ArXiv*, abs/1506.04498, 2015.

7 Satoshi Egi and Yuichi Nishiwaki. Non-linear pattern matching with backtracking for non-free data types. *ArXiv*, abs/1808.10603, 2018.

8 S. Eker. Fast matching in combinations of regular equational theories. *Electronic Notes in Theoretical Computer Science*, 4:90–109, 1996. RWLW96, First International Workshop on Rewriting Logic and its Applications. doi:10.1016/S1571-0661(04)00035-0.

9 R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

10 G. Hondet. Efficient rewriting using decision trees. Master's thesis, ENAC and Université Paul Sabatier, 2019.

11 J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.

12 R. Lepigre and C. Raffalli. Abstract Representation of Binders in OCaml using the Bindlib Library. In *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, Electronic Proceedings in Theoretical Computer Science 274, 2018.

13 T. Libal and D. Miller. Functions-as-constructors Higher-order Unification. In *Proceedings of the 1st International Conference on Formal Structures for Computation and Deduction*, Leibniz International Proceedings in Informatics 52, 2016.

14 L. Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the ACM SIGPLAN Workshop on ML*, 2008.

15 D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[4] https://github.com/Deducteam/Dedukti/tree/acu-state

16 Kristoffer H. Rose. CRSX - Combinatory Reduction Systems with Extensions. In Manfred Schmidt-Schauß, editor, *22nd International Conference on Rewriting Techniques and Applications (RTA'11)*, volume 10 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 81–90, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.RTA.2011.81.

17 R. Saillard. *Type checking in the Lambda-Pi-calculus modulo: theory and practice*. PhD thesis, Mines ParisTech, France, 2015.

18 F. Thiré. Sharing a Library between Proof Assistants: Reaching out to the HOL Family. In *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, Electronic Proceedings in Theoretical Computer Science 274, 2018.

WANDA – a Higher Order Termination Tool

Cynthia Kop
Radboud University, The Netherlands
https://www.cs.ru.nl/~cynthiakop/
c.kop@cs.ru.nl

Abstract

Wanda is a fully automatic termination analysis tool for higher-order term rewriting. In this paper, we will discuss the methodology used in Wanda. Most pertinently, this includes a higher-order dependency pair framework and a variation of the higher-order recursive path ordering, as well as some non-termination analysis techniques and delegation to a first-order tool. Additionally, we will discuss Wanda's internal rewriting formalism, and how to use Wanda in practice for systems in two different formalisms. We also present experimental results that consider both formalisms.

2012 ACM Subject Classification Theory of computation → Equational logic and rewriting

Keywords and phrases higher-order term rewriting, termination, automatic analysis, dependency pair framework, higher-order recursive path ordering

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.36

Category System Description

Supplementary Material A detailed experimental evaluation and the snapshot of Wanda used in this paper are available from: https://www.cs.ru.nl/~cynthiakop/experiments/fscd20.

Funding The author is supported by the NWO TOP project "ICHOR", NWO 612.001.803/7571.

Acknowledgements Thanks go to Carsten Fuhs both for proof-reading and for creating a customised version of AProVE which gives an explicit example term for non-termination; to Julian Nagele for using CSI^ho to translate the pattern HRSs in COPS to AFSMs; and to the anonymous reviewers of FSCD 2020 whose thorough feedback helped to improve the paper.

1 Introduction

Termination of term rewriting systems has been an area of active research for several decades. This concerns not only the analysis of pure term rewriting, but also many variants, such as context-sensitive [51], conditional [40] and higher-order [7] term rewriting. Since the introduction of the annual *International Termination Competition* [12], automated techniques in particular have flourished, with many strong provers competing against each other.

Compared to the core area of first-order term rewriting, *higher-order* term rewriting provides some unique challenges, for example due to bound variables. Nevertheless, several tools have participated in the higher-order category of the termination competition (Hot [4], THOR [8], Sol [28], SizeChangeTool [23], Wanda), each using different methods; these include both extensions of first-order techniques like recursive and semantic path orderings [30, 14, 29, 9] and dependency pairs [3, 38, 37], and also dedicated methods such as sized types [5].

Wanda, a tool built primarily around dependency pairs, has participated in this category since 2010 and won most years, including 2019. Wanda was also used as a termination back-end in the higher-order category of the 2019 *International Confluence Competition* [11], with both participants (ACPH [44] and CSI^ho [42]) delegating termination analysis to Wanda.

Despite this history, Wanda is not well-documented: no tool description has ever been formally published. Implementation choices *are* outlined in the author's PhD thesis [35] alongside termination techniques, but are not easily accessible as understanding these parts requires an understanding of the whole document. This has led to problems, as critical

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 36; pp. 36:1–36:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

details – such as the rewriting formalism Wanda employs, what Wanda actually does and how to use Wanda in different configurations or for different styles of rewriting – are hard to find. In addition, there have been substantial updates in recent years.

The present work will address this issue by presenting the usage of and the most important techniques used in Wanda. To start, the formalism of higher-order rewriting Wanda uses, AFSMs, is explained in §2, as well as its relation to other popular higher-order formalisms. Then we will discuss the non-termination and termination techniques used in Wanda (§3–5), The paper ends with experimental results, practical information and conclusions (§6–8).

Wanda is open-source, and is available at: `http://wandahot.sourceforge.net`. The snapshot that was used in the present paper, including all back-ends, is available from: `https://www.cs.ru.nl/~cynthiakop/experiments/fscd20/wanda2020.zip` .

Theoretical contribution. Although the focus is on Wanda, this paper also presents some theoretical results that were previously only published in the author's PhD thesis:

- a transformation from pattern HRSs [43] to Wanda's internal format, AFSMs;
- two simple non-termination techniques (§3.1–3.2);
- a new variation of the higher-order recursive path ordering suited to AFSMs (§4.2).

In addition, the results of §2.3 and §4.1, and the "dynamic" part of §5, were previously presented for a more restricted formalism and are here generalised to AFSMs. The remaining results in this paper connect and discuss existing work, and explain how it is used in Wanda.

2 Higher-order term rewriting using AFSMs

There is no single, unified approach to higher-order term rewriting; rather, there are several similar but not fully compatible systems. This is a problem, since users of various kinds of higher-order TRSs may be interested in termination, and it would be frustrating to adapt techniques and write different tools for each style. Therefore, Wanda uses a custom format, *AFSMs*, which several popular kinds of rewriting systems can be translated into. AFSMs (Algebraic Functional Systems with Meta-variables) are essentially simply-typed CRSs [32] and also largely correspond to the formalism in [6]. AFSMs are fully presented in [22].

2.1 Preliminaries: the AFSM formalism

Wanda operates on typed expressions, defined by Definitions 1 and 2.

▶ **Definition 1** (Simple types). *We fix a set $\mathcal{S}$ of* sorts. *All sorts are simple types, and if σ, τ are simple types, then so is $\sigma \rightarrow \tau$. Here, $\rightarrow$ is right-associative.*

Denoting ι, κ for a sort, all types have a unique form $\sigma_1 \rightarrow \ldots \rightarrow \sigma_m \rightarrow \iota$.

This definition does not have type variables, which occur in polymorphic styles of rewriting. Wanda *does* allow them as input, but since the implementation of most termination techniques does not support polymorphism, we will here consider only the simple types above.

▶ **Definition 2** (Terms and meta-terms). *We fix disjoint sets $\mathcal{F}$ of* function symbols, *$\mathcal{V}$ of* variables *and $\mathcal{M}$ of* meta-variables, *each symbol equipped with a type. Each meta-variable is additionally equipped with a natural number (its arity). We assume that both $\mathcal{V}$ and $\mathcal{M}$ contain infinitely many symbols of all types. The set $\mathcal{T}(\mathcal{F}, \mathcal{V})$ of* terms *over $\mathcal{F}, \mathcal{V}$ consists of expressions s where $s : \sigma$ can be derived for some type σ by the following clauses:*

$$\begin{array}{llll@{\qquad}llll} (\mathsf{V}) & x : \sigma & \text{if} & x : \sigma \in \mathcal{V} & (@) & s\ t : \tau & \text{if} & s : \sigma \rightarrow \tau \text{ and } t : \sigma \\ (\mathsf{F}) & \mathtt{f} : \sigma & \text{if} & \mathtt{f} : \sigma \in \mathcal{F} & (\Lambda) & \lambda x.s : \sigma \rightarrow \tau & \text{if} & x : \sigma \in \mathcal{V} \text{ and } s : \tau \end{array}$$

Meta-terms *are expressions whose type can be derived by the clauses above along with:*

(M) $Z[s_1,\dots,s_k] : \sigma_{k+1} \to \dots \to \sigma_m \to \iota$
if $Z : (\sigma_1 \to \dots \to \sigma_k \to \dots \to \sigma_m \to \iota,\ k) \in \mathcal{M}$ *and* $s_1 : \sigma_1,\dots,s_k : \sigma_k$

The λ binds variables as in the λ-calculus; unbound variables are called free, *and $FV(s)$ is the set of free variables in s. Meta-variables cannot be bound; we write $FMV(s)$ for the set of meta-variables occurring in s. A meta-term s is called* closed *if $FV(s) = \emptyset$ (even if $FMV(s) \neq \emptyset$). Meta-terms are considered modulo α-conversion. Application (@) is left-associative; abstractions (Λ) extend as far to the right as possible. A meta-term s* has type *σ if $s : \sigma$; it* has base type *if $\sigma \in \mathcal{S}$. Let* $\mathsf{head}(s) = \mathsf{head}(s_1)$ *if $s = s_1\ s_2$; otherwise* $\mathsf{head}(s) = s$.

A (meta-)term s has a sub-(meta-)term *t, notation $s \trianglerighteq t$, if either $s = t$ or $s \triangleright t$, where $s \triangleright t$ if (a) $s = \lambda x.s'$ and $s' \trianglerighteq t$, (b) $s = s_1\ s_2$ and $s_2 \trianglerighteq t$ or (c) $s = s_1\ s_2$ and $s_1 \trianglerighteq t$.*

Note that every term s has a form $t\ s_1 \cdots s_n$ with $n \geq 0$ and $t = \mathsf{head}(s)$ a variable, function symbol, or abstraction; in meta-terms t may also be a meta-variable application $Z[s_1,\dots,s_k]$. *Terms* are the objects that we will rewrite; *meta-terms* are used to define rewrite rules. Note that all our terms (and meta-terms) are, by definition, well-typed. An example of a meta-term is $\lambda x.\lambda y.\mathtt{sin}\ Z[x]$. In the left-hand side of a rule, this meta-term stands for an arbitrary term of the form $\lambda x.\lambda y.\mathtt{sin}\ t$ where t may contain the bound variable x, but not the bound variable y. This is more fully defined in Definitions 4 and 5.

For rewriting, we will additionally employ *patterns*:

▶ **Definition 3** (Patterns). *A meta-term is a* pattern *if it has one of the forms $Z[x_1,\dots,x_k]$ with all x_i distinct variables and $Z : (\sigma,k) \in \mathcal{M}$ for some σ; $\lambda x.\ell$ with $x \in \mathcal{V}$ and ℓ a pattern; or $a\ \ell_1 \cdots \ell_n$ with $a \in \mathcal{F} \cup \mathcal{V}$ and all ℓ_i patterns ($n \geq 0$).*

In rewrite rules, meta-variables are used for *matching* and variables are only used with *binders*. In terms, variables can occur both free and bound, and meta-variables cannot occur. Meta-variables originate in early forms of higher-order rewriting (e.g., [1, 32]), but have also been used in later formalisms (e.g., [6]). They strike a balance between matching modulo β and syntactic matching. By using meta-variables, we obtain the same expressive power as with Miller patterns [41], but without including a reversed β-reduction as part of matching.

In Wanda, function symbols are identified by their name, and variables and meta-variables by an integer index; using integers makes it very easy to allocate fresh variables when needed. The indexes are not shown to the user; instead a unique name is generated for printing.

▶ **Definition 4** (Substitution). *A* substitution *γ is a type-preserving mapping from a subset of $\mathcal{V} \cup \mathcal{M}$ (the* domain *of γ) to terms, typically denoted in a form $\gamma = [b_1 := s_1,\dots,b_n := s_n]$ (here, the domain is $\{b_1,\dots,b_n\}$). Substitutions may have infinite domain, but – denoting $\mathtt{dom}(\gamma)$ for the domain of γ – we require that there are infinitely many variables x of all types such that (a) $x \notin \mathtt{dom}(\gamma)$ and (b) for all $b \in \mathtt{dom}(\gamma)$: $x \notin FV(\gamma(b))$.*

A substitution is extended to a function from meta-terms to meta-terms as follows:

$$\begin{array}{rcll}
x\gamma & = & \gamma(x) & \textit{if}\quad x \in \mathcal{V} \cap \mathtt{dom}(\gamma) \\
x\gamma & = & x & \textit{if}\quad x \in \mathcal{V} \setminus \mathtt{dom}(\gamma) \\
\mathtt{f}\gamma & = & \mathtt{f} & \textit{if}\quad \mathtt{f} \in \mathcal{F} \\
(s\ t)\gamma & = & (s\gamma)\ (t\gamma) & \\
(\lambda x.s)\gamma & = & \lambda x.(s\gamma) & \textit{if}\quad x \notin \mathtt{dom}(\gamma) \wedge x \notin \\
 & & & \quad \bigcup_{y \in \mathtt{dom}(\gamma)} FV(\gamma(y)) \\
Z[s_1,\dots,s_k]\gamma & = & Z[s_1\gamma,\dots,s_k\gamma] & \textit{if}\quad Z \notin \mathtt{dom}(\gamma) \\
Z[s_1,\dots,s_k]\gamma & = & t[x_1 := s_1\gamma,\dots,x_k := s_k\gamma] & \textit{if}\quad \gamma(Z) = \lambda x_1 \dots x_k.t \\
Z[s_1,\dots,s_k]\gamma & = & t[x_1 := s_1\gamma,\dots,x_n := s_n\gamma]\ \ (s_{n+1}\gamma)\cdots(s_k\gamma) & \textit{if}\quad \gamma(Z) = \lambda x_1 \dots x_n.t \\
 & & & \quad \wedge\ n < k
\end{array}$$

Note that substituting an abstraction is fully defined due to α-conversion and the requirement that there are infinitely many variables not occurring in the domain or range of γ. Moreover, for fixed k, any meta-term $\gamma(Z)$ can be written in the form $\lambda x_1 \dots x_n.t$ with either $n < k$ and t not an abstraction, or $n = k$ (and t unrestricted). Thus, this is well-defined.

Essentially, applying a substitution with meta-variables in its domain combines a substitution with a β-development. For example, $\mathtt{deriv}\ (\lambda x.\mathtt{sin}\ (F[x]))[F := \lambda y.\mathtt{plus}\ y\ x]$ equals $\mathtt{deriv}\ (\lambda z.\mathtt{sin}\ (\mathtt{plus}\ z\ x))$, and $X[\mathtt{0},\mathtt{nil}][X := \lambda x.\mathtt{map}\ (\lambda y.x)]$ equals $\mathtt{map}\ (\lambda y.\mathtt{0})\ \mathtt{nil}$. If $\mathtt{dom}(\gamma)$ contains all meta-variables in $FMV(s)$, then $s\gamma$ is a term.

▶ **Definition 5** (Rules and Rewriting). *A* rule *is a pair $\ell \Rightarrow r$ of closed meta-terms of the same type, where ℓ is a pattern of the form $\mathtt{f}\ \ell_1 \cdots \ell_n$ with $\mathtt{f} \in \mathcal{F}$, and $FMV(r) \subseteq FMV(\ell)$. For a set of rules $\mathcal{R}$, reduction is the smallest monotonic relation $\Rightarrow_{\mathcal{R}}$ on terms that includes:*

(Rule)	$\ell\gamma$	$\Rightarrow_{\mathcal{R}}$	$r\gamma$	*for $\ell \Rightarrow r \in \mathcal{R}$, and γ a substitution with $\mathtt{dom}(\gamma) = FMV(\ell)$*
(Beta)	$(\lambda x.s)\ t$	$\Rightarrow_{\mathcal{R}}$	$s[x := t]$	

Note that we can reduce at any position of a term, even below a λ. We write $s \Rightarrow_\beta t$ if $s \Rightarrow_{\mathcal{R}} t$ is derived using (Beta). A term s is terminating *under $\mathcal{R}$ if there is no infinite reduction $s = s_0 \Rightarrow_{\mathcal{R}} s_1 \Rightarrow_{\mathcal{R}} \dots$,* is in normal form *if there is no t with $s \Rightarrow_{\mathcal{R}} t$, and* is β-normal *if there is no t with $s \Rightarrow_\beta t$. The relation $\Rightarrow_{\mathcal{R}}$ is terminating if all terms are terminating.*

Although the theory in [35] allows for $\mathcal{R}$ to be infinite (mostly with an eye on polymorphism), Wanda does not fully support this yet, so we will here limit interest to finite $\mathcal{R}$.

▶ **Example 6.** Let $\mathcal{F} \supseteq \{\mathtt{0} : \mathtt{nat},\ \mathtt{s} : \mathtt{nat} \to \mathtt{nat},\ \mathtt{nil} : \mathtt{list}, \mathtt{cons} : \mathtt{nat} \to \mathtt{list} \to \mathtt{list},\ \mathtt{map} : (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{list} \to \mathtt{list}\}$ and consider the following rules $\mathcal{R}_1$:

$$\begin{array}{rcl} \mathtt{map}\ (\lambda x.Z[x])\ \mathtt{nil} & \Rightarrow & \mathtt{nil} \\ \mathtt{map}\ (\lambda x.Z[x])\ (\mathtt{cons}\ H\ T) & \Rightarrow & \mathtt{cons}\ Z[H]\ (\mathtt{map}\ (\lambda x.Z[x])\ T) \end{array}$$

Then $\mathtt{map}\ (\lambda y.\mathtt{0})\ (\mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ \mathtt{nil}) \Rightarrow_{\mathcal{R}_1} \mathtt{cons}\ \mathtt{0}\ (\mathtt{map}\ (\lambda y.\mathtt{0})\ \mathtt{nil}) \Rightarrow_{\mathcal{R}_1} \mathtt{cons}\ \mathtt{0}\ \mathtt{nil}$. Note that the bound variable y does not need to occur in the body of $\lambda y.\mathtt{0}$ to be matched by $\lambda x.Z[x]$. However, note also that a term like $\mathtt{map}\ \mathtt{s}\ (\mathtt{cons}\ \mathtt{0}\ \mathtt{nil})$ *cannot* be reduced, because $\mathtt{s}$ does not match $\lambda x.Z[x]$. We could alternatively consider the rules $\mathcal{R}_2$:

$$\begin{array}{rcl} \mathtt{map}\ Z\ \mathtt{nil} & \Rightarrow & \mathtt{nil} \\ \mathtt{map}\ Z\ (\mathtt{cons}\ H\ T) & \Rightarrow & \mathtt{cons}\ (Z\ H)\ (\mathtt{map}\ Z\ T) \end{array}$$

In the previous example, we had $Z : (\mathtt{nat} \to \mathtt{nat}, 1) \in \mathcal{M}$; here, we have $Z : (\mathtt{nat} \to \mathtt{nat}, 0) \in \mathcal{M}$ (we will typically leave this implicit since the arity of meta-variables can be read off from the left-hand sides of the rules). Instead of meta-variable application $Z[x]$, we use explicit application $Z\ x$. Now we do have $\mathtt{map}\ \mathtt{s}\ (\mathtt{cons}\ \mathtt{0}\ \mathtt{nil}) \Rightarrow_{\mathcal{R}_2} \mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ (\mathtt{map}\ \mathtt{s}\ \mathtt{nil})$. However, now we will often need explicit β-reductions; e.g., $\mathtt{map}\ (\lambda y.\mathtt{0})\ (\mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ \mathtt{nil}) \Rightarrow_{\mathcal{R}_2} \mathtt{cons}\ ((\lambda y.\mathtt{0})\ (\mathtt{s}\ \mathtt{0}))\ (\mathtt{map}\ (\lambda y.\mathtt{0})\ \mathtt{nil}) \Rightarrow_\beta \mathtt{cons}\ \mathtt{0}\ (\mathtt{map}\ (\lambda y.\mathtt{0})\ \mathtt{nil})$.

Thus, AFSMs allow us to define essentially the same rules in multiple ways. This flexibility may seem redundant, but is necessary to enable the analysis of different styles of higher-order term rewriting, as we will see in §2.2. An AFSM is a pair $(\mathcal{T}(\mathcal{F}, \mathcal{V}), \Rightarrow_{\mathcal{R}})$ of a set of terms and a reduction relation on that set. To define an AFSM, it suffices to supply $\mathcal{F}$ and $\mathcal{R}$; types of (meta-)variables can be derived from context. This is what Wanda takes as input.

▶ **Example 7.** The first `map` rules from Example 6 can be given to Wanda, in a file `map.afsm`, which provides first the signature and then the rules:

```
nil : list
cons : nat -> list -> list
map : (nat -> nat) -> list -> list

map (/\x.Z[x]) nil => nil
map (/\x.Z[x]) (cons H T) => cons Z[H] (map (/\x.Z[x]) T)
```

Note: all identifiers (function symbols, variables and meta-variables) in `.afsm` files are expected to be alphanumeric. Characters such as +, - and _ are not allowed in names. The only exceptions are the exclamation mark symbol ('!') and the percentage symbol ('%'); the latter may only be used at the start of (meta-)variables.

2.2 Transformations

AFSMs are not meant to be interesting in their own right. Rather, they are defined to support termination proofs in multiple formalisms. Let us consider the two most relevant.

Higher-order Rewriting Systems (HRSs) [43] are one of the oldest styles of higher-order term rewriting. Here, rewriting is *modulo* $\Rightarrow_\beta$: for terms s, t in η-long β-normal form we have $s \Rightarrow_\mathcal{R} t$ if there exist a rule $\ell \Rightarrow r$ and a substitution γ such that $\ell\gamma \Rightarrow_\beta^* s$ and $r\gamma \Rightarrow_\beta^* t$. All terms are presented in η-long β-normal form, and rules are pairs of such terms (there are no meta-variables). The η-long form of a term s is obtained by repeatedly applying the step "$s \Rightarrow_\eta \lambda x.(s\ x)$" on all subterms of s where this can be done without creating a β-redex.

In general, the reduction relation $\Rightarrow_\mathcal{R}$ in an HRS is not computable, but practical examples typically consider *pattern HRSs* (PRSs), where for all rules $\ell \Rightarrow r$ and for all subterms $x\ \ell_1 \cdots \ell_m$ of the left-hand side with x a variable: each ℓ_i is the η-long form of a distinct bound variable. Pattern HRSs are translated to AFSMs in a natural way, by replacing free variables in the rules by meta-variables, and their applications by meta-applications.

▶ **Example 8.** Let us consider an example of a pattern HRS:

$$\begin{array}{rcl} \mathtt{bind}\ (\mathtt{return}\ x)\ (\lambda y.f\ y) & \Rightarrow & f\ x \\ \mathtt{bind}\ x\ (\lambda y.\mathtt{return}\ y) & \Rightarrow & x \\ \mathtt{bind}\ (\mathtt{bind}\ x\ (\lambda y.f\ y))\ (\lambda z.g\ z) & \Rightarrow & \mathtt{bind}\ x\ (\lambda u.\mathtt{bind}\ (f\ u)\ (\lambda v.g\ v)) \end{array}$$

It is translated to the following AFSM (meta-variables are indicated with capitals):

$$\begin{array}{rcl} \mathtt{bind}\ (\mathtt{return}\ X)\ (\lambda y.F[y]) & \Rightarrow & F[X] \\ \mathtt{bind}\ X\ (\lambda y.\mathtt{return}\ y) & \Rightarrow & X \\ \mathtt{bind}\ (\mathtt{bind}\ X\ (\lambda y.F[y]))\ (\lambda z.G[z]) & \Rightarrow & \mathtt{bind}\ X\ (\lambda u.\mathtt{bind}\ (F[u])\ (\lambda v.G[v])) \end{array}$$

This translated system has very similar behaviour to the original PRS, but there is a critical difference: the PRS is a relation on η-long β-normal terms, while the AFSM is generally considered as a relation on all terms. It turns out that the restriction to η-long terms does not affect termination, but the β-normalisation does ([35, Theorem 3.5]):

▶ **Lemma 9.** *The original PRS $(\mathcal{F}, \mathcal{R})$ is terminating if and only if the translated AFSM $(\mathcal{F}, \mathcal{R}')$ is terminating using a reduction strategy where $\Rightarrow_\beta$ is preferred to other steps.*

That is, we need to β-normalise terms after every reduction step. Wanda can test the property of termination with a $\Rightarrow_\beta$-first strategy by being invoked with a runtime argument `--betafirst` (e.g., `./wanda.exe --betafirst system.afsm`). As a side note, however, the examples where this requirement makes a difference are rare and typically artificial.

▶ **Example 10.** Let $\mathcal{F} = \{\mathtt{a} : \mathtt{o},\ \mathtt{f} : \mathtt{o} \to \mathtt{o},\ \mathtt{g} : ((\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}) \to \mathtt{o}\}$ and $\mathcal{R}$ given by:

$$\mathtt{f}\ \mathtt{a} \ \Rightarrow\ \mathtt{g}\ (\lambda x.\lambda y.x\ (\mathtt{f}\ y)) \qquad \mathtt{g}\ (\lambda x.\lambda y.h\ (\lambda z.x\ z)\ y) \ \Rightarrow\ h\ (\lambda z.\mathtt{a})\ (\mathtt{a})$$

This PRS is translated to an AFSM with the following rules $\mathcal{R}'$:

$$\mathtt{f}\ \mathtt{a} \ \Rightarrow\ \mathtt{g}\ (\lambda x.\lambda y.x\ (\mathtt{f}\ y)) \qquad \mathtt{g}\ (\lambda x.\lambda y.H[x,y]) \ \Rightarrow\ H[\lambda z.\mathtt{a},\ \mathtt{a}]$$

While the original PRS is terminating, the same does not hold for the translated AFSM: we have $\mathtt{f}\ \mathtt{a} \Rightarrow_{\mathcal{R}'} \mathtt{g}\ (\lambda x.\lambda y.x\ (\mathtt{f}\ y)) \Rightarrow_{\mathcal{R}'} (\lambda z.\mathtt{a})\ (\mathtt{f}\ \mathtt{a})$, where the last term has $\mathtt{f}\ \mathtt{a}$ as a subterm. In an AFSM, it is not mandatory to reduce the β-redex. Wanda concludes non-termination normally, but cannot find a proof or disproof if the `--betafirst` argument is provided.

Remark: Wanda has not been optimised for HRSs, and does not take advantage of the `--betafirst` argument other than avoiding false claims of non-termination. This is primarily due to a lack of motivating examples: the annual Termination Competition does not consider HRSs. However, since the *International Confluence Competition* [11] *does* consider HRSs, and comes with its own benchmark set, this situation is likely to change in the future.

Algebraic Functional Systems (AFSs) [29] are higher-order term rewriting systems with $\Rightarrow_\beta$ as a separate step (i.e., $\Rightarrow_\beta \subseteq \Rightarrow_{\mathcal{R}}$; unlike HRSs, β-steps are not implicitly done as part of other steps); this is the format used in the higher-order category of the International Termination Competition [12]. Rules in an AFS are pairs of *terms*, not *meta-terms*, and there is no pattern Another difference with AFSMs is that AFSMs use applicative (curried) notation while AFSs use a mixture of functional and applicative term formation; however, this difference is not significant, since – following [34, 35] – currying does not affect termination.

Using variables rather than meta-variables for matching is not important either: just replace all free variables by meta-variables. This gives rules like the "alternative" rules $\mathcal{R}_2$ in Example 6. However, the lack of a pattern restriction is very significant.

▶ **Example 11.** Let us consider an example of an AFS that cannot be naturally translated without violating pattern restrictions. We let $\mathcal{F} = \{\mathtt{new} : (\mathtt{N} \to \mathtt{A}) \to \mathtt{A}\}$ and $\mathcal{R}$ consist of:

$$\mathtt{new}\ (\lambda x.y) \ \Rightarrow\ y \qquad \mathtt{new}\ (\lambda x.\mathtt{new}\ (\lambda y.f\ x\ y)) \ \Rightarrow\ \mathtt{new}\ (\lambda x.\mathtt{new}\ (\lambda y.f\ y\ x))$$

Now, the left-hand sides *look* like patterns. Indeed, they satisfy the requirements for an HRS-pattern: the free variable f in the second rule is only applied to distinct bound variables. So if this was an HRS, we could translate it to the following AFSM:

$$\mathtt{new}\ (\lambda x.Y) \ \Rightarrow\ Y \qquad \mathtt{new}\ (\lambda x.\mathtt{new}\ (\lambda y.F[x,y])) \ \Rightarrow\ \mathtt{new}\ (\lambda x.\mathtt{new}\ (\lambda y.F[y,x]))$$

However, since the original system was an AFS, this is *not* equivalent. Unlike in HRSs, matching in AFSs is not modulo beta: like in AFSMs, s rewrites to t by rule $\ell \Rightarrow r$ if there exists a substitution γ such that $s = \ell\gamma$ and $t = r\gamma$. So, in the AFS, the subterm $f\ x\ y$ can *only* be instantiated by terms of the form $s\ x\ y$. An accurate translation of the second rule to AFSMs would simply replace $f\ x\ y$ by $F[]\ x\ y$, resulting in a non-pattern.

This is important because the AFSM above is non-terminating: $\mathtt{new}\ (\lambda x.\mathtt{new}\ (\lambda y.z))$ reduces to itself in one step because the meta-variable F can be instantiated by a substitution $\lambda x.\lambda y.z$. On the other hand, the original AFS is terminating, as we will see below.

A final difference is that, following [29], AFSs use polymorphic types. Wanda limits interest to simply-typed AFSs, which is what the Termination Competition uses. Polymorphic AFSs can be translated to polymorphic AFSMs, but this is not yet well-supported in Wanda.

Wanda accepts AFSs as input directly (using the xml format of the Termination Competition or a custom human-readable format). *Most* AFSs can be naturally translated into AFSMs just by replacing free variables by meta-variables; typically counterexamples look like they were meant as HRSs, but translated poorly into AFSs. For the examples that cannot be naturally translated, Wanda first applies the transformations in [34] to create patterns. This involves introducing fresh symbols $\mathtt{app}_i$ to replace some of the applications $s\ t$ by terms of the form $\mathtt{app}_i\ s\ t$. New rules may also be introduced, as for instance $\mathtt{f}\ (X\ Y)\ \mathtt{a} \Rightarrow \mathtt{f}\ (X\ \mathtt{b})\ Y$ is replaced by not only $\mathtt{f}\ (\mathtt{app}_i\ X\ Y)\ \mathtt{a} \Rightarrow \mathtt{f}\ (\mathtt{app}_i\ X\ \mathtt{b})\ Y$, but in addition potentially many rules of the form $\mathtt{f}\ (\mathtt{g}\ X_1 \cdots X_n\ Y)\ \mathtt{a} \Rightarrow \mathtt{f}\ (\mathtt{g}\ X_1 \cdots X_n\ \mathtt{a})\ Y$. This is exacerbated when the AFS is presented in curried (applicative) form rather than functional notation.

▶ **Example 12.** The AFS of Example 11 is translated to an AFSM with the following rules:

$$\begin{array}{rcl} \mathtt{new}\ (\lambda x.Y) & \Rightarrow & Y \\ \mathtt{new}\ (\lambda x.\mathtt{new}\ (\lambda y.\mathtt{app}\ F\ x\ y)) & \Rightarrow & \mathtt{new}\ (\lambda x.\mathtt{new}\ (\lambda y.\mathtt{app}\ F\ u\ z)) \\ \mathtt{app}\ F\ X & \Rightarrow & F\ X \end{array}$$

This AFSM can be proved terminating by Wanda's recursive path ordering (§4.2) in combination with dependency pairs (§5).

It is worth noting that the transformations needed to translate an AFS to an AFSM with equivalent behaviour can sometimes cause the system to become much more difficult to analyse, both due to the inclusion of explicit "application" symbols in the rules and the addition of potentially many new rules. For this reason, Wanda uses the following approach:

- create both an accurate translation and an *overestimation* of the AFS (so that termination of the overestimation implies termination of the original system, but not the reverse); this results in translations like those given in Example 11;
- try to prove non-termination using the accurate translation;
- try to prove termination using the overestimation;
- if this fails, try to prove termination using the accurate translation.

2.3 Uncurrying

Following [35, §2.3.1] and [34, §7], *uncurrying* does not affect termination provided the rules are (essentially) unchanged. That is, we can denote both rules and terms in a functional notation, but only if the number of arguments is respected in each rule. To be exact:

▶ **Lemma 13.** *Let $(\mathcal{F},\mathcal{R})$ be an AFSM, and let $minar(\mathtt{f})$ denote the largest number k such that (1) the type of $\mathtt{f}$ allows $\mathtt{f}$ to be applied to at least k arguments, and (2) every occurrence of $\mathtt{f}$ in $\mathcal{R}$ is applied to at least k arguments. Then $\Rightarrow_{\mathcal{R}}$ is non-terminating if and only if there is an infinite reduction $s_1 \Rightarrow_{\mathcal{R}} s_2 \Rightarrow_{\mathcal{R}} \dots$ where, in every term s_i, each symbol $\mathtt{f}$ always occurs with at least $minar(\mathtt{f})$ arguments.*

For example, in Example 6, $minar(\mathtt{s}) = 1$ and $minar(\mathtt{cons}) = minar(\mathtt{map}) = 2$; thus, we do not need to consider terms such as $\mathtt{map}\ \mathtt{s}\ (\mathtt{cons}\ \mathtt{0}\ \mathtt{nil})$ or $\mathtt{map}\ (\lambda x.\mathtt{s}\ x)$ for termination. Wanda indicates this by showing terms in functional notation; e.g., $\mathtt{map}(\lambda x.\mathtt{s}(x), \mathtt{cons}(\mathtt{0}, \mathtt{nil}))$.

▶ **Example 14.** Consider the toy system with $\mathcal{F} = \{\mathtt{a}, \mathtt{b} : \mathtt{o},\ \mathtt{f} : \mathtt{o} \to \mathtt{o} \to \mathtt{o},\ \mathtt{g} : \mathtt{o} \to (\mathtt{o} \to \mathtt{o}) \to \mathtt{o}\}$ and $\mathcal{R} = \{\ \mathtt{f}\ \mathtt{a}\ X \Rightarrow \mathtt{g}\ X\ (\mathtt{f}\ \mathtt{a}),\ \mathtt{g}\ \mathtt{a}\ F \Rightarrow F\ \mathtt{b}\}$. Then $minar(\mathtt{a}) = minar(\mathtt{b}) = 0$, $minar(\mathtt{g}) = 2$ and $minar(\mathtt{f}) = 1$ (since $\mathtt{f}$ occurs both with 1 or 2 arguments, we must choose the smaller value). Wanda prints these rules as $\mathtt{f}(\mathtt{a})\ X \Rightarrow \mathtt{g}(X,\ \mathtt{f}(\mathtt{a}))$ and $\mathtt{g}(\mathtt{a}, F) \Rightarrow F\ \mathtt{b}$.

We do *not* η-expand as part of uncurrying. To illustrate why not, note that the above system is terminating, but its η-long variant, which has a rule $\mathtt{f}\ \mathtt{a}\ X \Rightarrow \mathtt{g}\ (\lambda z.\mathtt{f}\ \mathtt{a}\ z)$, is not.

3 Non-termination

As Wanda's focus is on proving termination, the available non-termination techniques are currently quite minimal. There are three methods. The first two are very quick, and are applied at the start of the analysis, before termination is considered. The last one is employed when dependency pairs are initiated, as it is combined with the simplification given in §5.2.

3.1 Detecting obvious loops

An AFSM is clearly non-terminating if there is a reduction $s \Rightarrow_{\mathcal{R}}^{*} t$ such that $t \trianglerighteq s\gamma$ for some γ. To discover such loops, Wanda takes the left-hand side of a rule, replaces meta-variable applications $Z[x_1, \dots, x_k]$ by variable applications $y\ x_1 \cdots x_k$, and performs a breadth-first search on reducts to see whether any instances of the original term appear, not going beyond the first 1000. If the `betafirst` runtime argument is given, then reducts are β-normalised before this test is done. This simple method will not find any sophisticated counterexamples for termination, but is quick and easy, and often catches mistakes in a recursive call.

In the future, it would be natural to extend this module to use semi-unification [31] instead of matching, as done for first-order rewriting in [26]. However, this would require the design of a higher-order semi-unification algorithm. Similarly, Wanda could be strengthened by creating higher-order variants of existing first-order non-termination techniques (e.g., [17, 45, 46]), but this would require substantial new work to develop the theory.

3.2 The $\omega\omega$ counterexample

Wanda also has one truly higher-order non-termination technique, which does not build on first-order methods. This technique recognises a particular kind of rule that leads to non-termination in a non-obvious way. The idea is to build a variation of the λ-term $\omega\omega$ in the untyped λ-calculus, where $\omega = \lambda x.xx$. Note that $\omega\omega$ reduces to itself in one $\Rightarrow_\beta$-step.

Let a *context* be a meta-term $\underline{C}[\Box_1, \dots, \Box_n]$ containing n typed holes $\Box_i$, and denote $\underline{C}[s_1, \dots, s_n]$ for the same meta-term with each $\Box_i$ replaced by s_i. Wanda identifies rules $\ell \Rightarrow r$ where ℓ has the form $\underline{C}[\underline{D}[Z], X]$ such that:

- $Z : \sigma_1 \to \dots \to \sigma_n \to \tau \in \mathcal{M}$, where τ is the type of ℓ;
- there is some i with $X : (\sigma_i, 0) \in \mathcal{M}$ and also $\underline{D}[Z]$ has type σ_i;
- r can be written as $\underline{E}[Z\ s_1 \cdots s_{i-1}\ X\ s_{i+1} \cdots s_n]$
- X and Z do not appear at other positions in $\underline{C}$ or $\underline{D}$.

If this is satisfied, Wanda concludes non-termination with the following justification. Let γ be the substitution that maps each $Y : \pi_1 \to \dots \to \pi_m \to \iota$ in the rule, aside from Z and X, to a term $\lambda x_1 \dots x_m.y\ x_1 \cdots x_m$ (with y a variable), and let $\omega := \underline{D}\gamma[\lambda x_1 \dots x_n.\underline{C}\gamma[x_i, x_i]]$. Let $\delta := \gamma \cup [X := \omega, Z := \lambda x_1 \dots x_n.\underline{C}\gamma[x_i, x_i]]$. Then $\underline{C}\gamma[\omega, \omega] = \ell\delta \Rightarrow_{\mathcal{R}} \underline{E}[(\lambda x_1 \dots x_n.\underline{C}\gamma[x_i, x_i])\ s_1 \cdots \omega \cdots s_n)]\delta \Rightarrow_{\beta}^{*} \underline{E}[\underline{C}\gamma[\omega, \omega]]\delta \trianglerighteq \underline{C}\gamma[\omega, \omega]$, a loop.

The method above is specialised for AFSMs that originate from AFSs (as used in the Termination Competition): it is designed for meta-variables that do not take any arguments. If meta-variables do take arguments, and for instance $\lambda x_1 \dots x_n.Z[x_1, \dots, x_n]$ is used instead of Z, we *probably* have a similar counter-example – depending on how Z and X are used in $\underline{E}$ (it is possible that $\underline{E}[]\delta$ does not contain any copies of $\Box_1$). Wanda tries to recognise such variations of the meta-variables, and tests whether the counterexample still applies.

3.3 Using a first-order tool

Finally, it is clear that an AFSM $(\mathcal{F}, \mathcal{R})$ is non-terminating if there is a subset $\mathcal{R}' \subseteq \mathcal{R}$ such that $\Rightarrow_{\mathcal{R}'}$ is non-terminating. An interesting subset is the set of rules that can be viewed as first-order (i.e., rules that do not use λ, that only use function symbols with a type declaration $\iota_1 \rightarrow \ldots \rightarrow \iota_m \rightarrow \iota_0$ with all $\iota_i \in \mathcal{S}$, and where function symbols only occur fully applied). This subset is easier to analyse, as known methods for first-order rewriting apply.

Thus, Wanda extracts this first-order part, to pass it to a dedicated first-order (non-) termination tool. The main problem with this approach is that existing tools do not consider types. This can make a difference, as shown by an example due to Toyama [50]:

▶ **Example 15.** Let $\mathcal{F} = \{\mathtt{0} : \mathtt{a},\ \mathtt{1} : \mathtt{a}, \mathtt{f} : \mathtt{a} \rightarrow \mathtt{a} \rightarrow \mathtt{a} \rightarrow \mathtt{a},\ \mathtt{g} : \mathtt{b} \rightarrow \mathtt{b} \rightarrow \mathtt{b}\}$ and $\mathcal{R} = \{\mathtt{f}(X, X, X) \Rightarrow \mathtt{f}(\mathtt{0}, \mathtt{1}, X),\ \mathtt{g}(X, Y) \Rightarrow X,\ \mathtt{g}(X, Y) \Rightarrow Y\}$. This system is terminating, because there is no term of type a that reduces to both 0 and 1. However, there is an *untypable* term that loops by these rules: $\mathtt{f}(\mathtt{0},\ \mathtt{1},\ \mathtt{g}(\mathtt{0}, \mathtt{1})) \Rightarrow_{\mathcal{R}} \mathtt{f}(\mathtt{g}(\mathtt{0}, \mathtt{1}),\ \mathtt{g}(\mathtt{0}, \mathtt{1}),\ \mathtt{g}(\mathtt{0}, \mathtt{1})) \Rightarrow_{\mathcal{R}} \mathtt{f}(\mathtt{0},\ \mathtt{g}(\mathtt{0}, \mathtt{1}),\ \mathtt{g}(\mathtt{0}, \mathtt{1})) \Rightarrow_{\mathcal{R}} \mathtt{f}(\mathtt{0},\ \mathtt{1},\ \mathtt{g}(\mathtt{0}, \mathtt{1}))$. Thus, a first-order termination tool (which does not consider types) would conclude non-termination.

Now, if the first-order subset is *orthogonal*, then it is terminating if and only if it is terminating without regarding types as observed in [20] (using a combination of results in [19] and [27]). Thus, in this case Wanda can use an arbitrary first-order tool without inhibitions. The same is true if the set of first-order rules uses only one sort. If neither of those cases holds, Wanda investigates the output of the first-order tool to see whether a non-terminating term is given, and if so, tests whether it is well-sorted.

Comment: unfortunately, the standard output format for the Termination Competition does not require tools to output a non-terminating term if NO is answered. Thus, any common first-order tool can be used if $\mathcal{R}'$ is orthogonal or has only one sort, but otherwise a specialised tool with the right output format is needed. For this, Wanda uses a custom adaptation of AProVE [24]. As AProVE is currently not open-source, this is not included in Wanda's release.

4 Orderings

At the heart of Wanda's termination techniques are *reduction pairs*. These are orderings on terms – generated by an ordering on meta-terms – which can be used both as part of the dependency pair framework (§5) and on their own to simplify a termination proof.

▶ **Definition 16.** *A* reduction pair *is a pair* $(\succsim, \succ)$ *of a quasi-ordering and a well-founded ordering on meta-terms of the same type, such that:*

- $\succsim$ *and* $\succ$ *are* compatible*:* $\succ \cdot \succsim$ *is included in* $\succ$*;*
- $\succsim$ *and* $\succ$ *are* meta-stable*: if* $s \succsim t$ *and* γ *is a substitution on domain* $FMV(s) \cup FMV(t)$*, then* $s\gamma \succsim t\gamma$ *(and similar for* $\succ$*);*
- $\succsim$ *is* monotonic*: if* $s \succsim t$*, then* $s\ u \succsim t\ u$ *and* $u\ s \succsim u\ t$ *and* $\lambda x.s \succsim \lambda x.t$
- $\succsim$ contains beta*:* $(\lambda x.s)\ t \succsim s[x := t]$ *if* s *and* t *are terms.*

A reduction pair is strongly monotonic *if moreover* $\succ$ *is monotonic.*

Strongly monotonic reduction pairs can be used in *rule removal*: if $\ell \succsim r$ for some rules, and $\ell \succ r$ for the remainder, then the rules in the remainder cannot occur infinitely often in a reduction sequence, and thus can be "removed" (they no longer need to be considered for the termination argument). Reduction pairs are also used – without the strong monotonicity requirement – in the dependency pair framework. It *would* be possible to also include rule removal with strongly monotonic reduction pairs in the framework rather than using it as a separate step; however, using it as a separate step often gives simpler termination proofs, and makes it possible to assess the strength of these reduction pairs in isolation.

Wanda has two ways to generate reduction pairs: *weakly monotonic interpretations* and *recursive path orderings*. Both ideas extend first-order methods, and use *functional notation*. This is an extension of uncurrying, where the remaining applications are replaced by function application, as follows: in every rule, every subterm of the left- or right-hand side of the form $s\ t$ is replaced by $@^{\sigma,\tau}(s,t)$, where $s : \sigma \to \tau$. The set of all symbols $@^{\sigma,\tau}$ that are used in the rules is added to $\mathcal{F}$, and the corresponding rules $@^{\sigma,\tau}(\lambda x.Z[x],Y) \Rightarrow Z[Y]$ are added to $\mathcal{R}$.

▶ **Example 17.** The AFSM of Example 14 is functionalised by replacing $\mathtt{f}(\mathtt{a})\ X$ in the uncurried rules by $@^{o,o}(\mathtt{f}(\mathtt{a}),X)$ and $F\ \mathtt{b}$ by $@^{o,o}(F,\mathtt{b})$. Thus, we obtain the rules:

$$\begin{array}{rcl@{\qquad}rcl} @^{o,o}(\mathtt{f}(\mathtt{a}),X) & \Rightarrow & \mathtt{g}(X,\mathtt{f}(\mathtt{a})) & @^{o,o}(\lambda x.Z[x],Y) & \Rightarrow & Z[Y] \\ \mathtt{g}(\mathtt{a},F) & \Rightarrow & @^{o,o}(F,\mathtt{b}) & & & \end{array}$$

4.1 Weakly monotonic algebras

The idea of van de Pol's *weakly monotonic algebras* [47] is to assign valuations which map all function symbols $\mathtt{f}$ of type σ to a *weakly monotonic functional* $\mathcal{J}_\mathtt{f}$: an element of $\llbracket\sigma\rrbracket$, where $\llbracket\iota\rrbracket$ is the set of natural numbers for a sort ι and $\llbracket\sigma \to \tau\rrbracket$ is the set of those functions from $\llbracket\sigma\rrbracket$ to $\llbracket\tau\rrbracket$ that are weakly monotonic (i.e., if $a,b \in \llbracket\sigma\rrbracket$ and $a \geq b$, then $f(a) \geq f(b)$ for $f \in \llbracket\sigma \to \tau\rrbracket$, where $\geq$ is a point-wise comparison). This induces a value on closed terms, which can be extended to a reduction pair, as explained below.

Given a meta-term s in functional notation and a function α which maps each variable $x : \sigma$ occurring freely in s to an element of $\llbracket\sigma\rrbracket$ and each meta-variable $Z : (\sigma_1 \to \dots \to \sigma_k \to \tau, k)$ to an element of $\llbracket\sigma_1 \to \dots \to \sigma_k \to \tau\rrbracket$, we let $[s]^\mathcal{J}_\alpha$ be recursively defined as follows:

$$\begin{array}{rcl@{\qquad}rcl} [x]^\mathcal{J}_\alpha & = & \alpha(x) & [\mathtt{f}(s_1,\dots,s_k)]^\mathcal{J}_\alpha & = & \mathcal{J}_\mathtt{f}([s_1]^\mathcal{J}_\alpha,\dots,[s_k]^\mathcal{J}_\alpha) \\ [\lambda x.s]^\mathcal{J}_\alpha & = & u \mapsto [s]^\mathcal{J}_{\alpha\cup[x:=u]} & [Z[s_1,\dots,s_k]]^\mathcal{J}_\alpha & = & \alpha(Z)([s_1]^\mathcal{J}_\alpha,\dots,[s_k]^\mathcal{J}_\alpha) \end{array}$$

(This follows the definition of $[\cdot]^\mathcal{J}_\alpha$ for functionalised AFSs in [21], but extends it with a case for meta-variable applications.) For closed meta-terms ℓ, r, let $\ell \succ r$ if $[\ell]^\mathcal{J}_\alpha > [r]^\mathcal{J}_\alpha$ for all α, and $\ell \succsim r$ if $[\ell]^\mathcal{J}_\alpha \geq [r]^\mathcal{J}_\alpha$ for all α. Then $(\succsim,\succ)$ is a reduction pair if the valuations $\mathcal{J}_{@\langle\sigma,\tau\rangle}$ are chosen to have $\mathcal{J}_{@\langle\sigma,\tau\rangle}(F,X) \geq F(X)$. It is a strongly monotonic pair if each function $\mathcal{J}_\mathtt{f}$ (including each $\mathcal{J}_{@\langle\sigma,\tau\rangle}$) is monotonic over $>$ in the first $\mathit{minar}(\mathtt{f})$ arguments.

In [21], a strategy is discussed to find interpretations based on *higher-order polynomials* for AFSs, and an automation using encodings of the ordering requirements into SAT. Wanda implements this methodology, only slightly adapted to take meta-variables into account.

▶ **Example 18.** We consider $\mathcal{R}_2$ in Example 6. Let $\mathcal{J}_\mathtt{nil} = 0$ and $\mathcal{J}_\mathtt{cons} = (n,m) \mapsto n+m+1$ and $\mathcal{J}_\mathtt{map} = (f,n) \mapsto nf(n) + 2n + f(0)$ and $\mathcal{J}_{@^{\mathtt{nat},\mathtt{nat}}} = (f,n) \mapsto f(n) + n$. Then, writing $F := \alpha(Z),\ n := \alpha(H),\ m := \alpha(T)$, we have:

- $[\mathtt{map}(Z,\mathtt{nil})]^\mathcal{J}_\alpha = F(0) \geq 0 = [\mathtt{nil}]^\mathcal{J}_\alpha$
- $[\mathtt{map}(Z,\mathtt{cons}(H,T))]^\mathcal{J}_\alpha = (n+m+1)\cdot F(n+m+1) + 2\cdot(n+m+1) + F(0) > (F(n) + n) + (m \cdot F(m) + 2\cdot m + F(0)) + 1 = [\mathtt{cons}(@^{\langle\mathtt{nat},\mathtt{nat}\rangle}(Z,H),\mathtt{map}(Z,T))]^\mathcal{J}_\alpha$
- $[@^{\mathtt{nat},\mathtt{nat}}(\lambda x.Z[x],H)]^\mathcal{J}_\alpha = F(H) + H \geq F(H) = [F[H]]^\mathcal{J}_\alpha$.

4.2 StarHorpo

The recursive path ordering [14] is a syntactic method to extend an ordering on function symbols to an ordering on first-order terms. There are various extensions (e.g. [18, 30]) including several higher-order variations (e.g. [7, 29]). However, these are mostly designed for rewriting with plain matching, and adapting them to work well with meta-variables is non-trivial. Instead, Wanda uses a specialised definition, built using the same ideas as [29] but using *iterative* path orderings [33, 36] as a starting point. This is discussed in detail in [35, Ch. 5]; here, we note only the end result: a reduction pair that can be used on functionalised AFSMs and (unlike other higher-order recursive path orderings) is natively transitive.

Following [33, 36], `StarHorpo` employs a star mark $\star$ to indicate an *intent to decrease*; practically, $\mathtt{f}^\star_\sigma(s_1, \dots, s_k)$ should be seen as an upper bound for all functional meta-terms of type σ which are *strictly smaller* than $\mathtt{f}(s_1, \dots, s_k)$. Let $s^\star$ denote $\lambda x_1 \dots x_n.\mathtt{f}^\star_\sigma(s_1, \dots, s_k)$ if $s = \lambda x_1 \dots x_n.\mathtt{f}(s_1, \dots, s_k)$. If s has any other form, then $s^\star$ is undefined.

`StarHorpo` assumes given a *precedence* $\underline{\blacktriangleright}$: a quasi-ordering on all symbols, whose strict part $\blacktriangleright$ is well-founded; we let $\approx$ denote the equivalence relation $\underline{\blacktriangleright} \cap \underline{\blacktriangleleft}$. We assume that there is a special symbol $\bot_\sigma$ for each type σ, which is minimal for $\underline{\blacktriangleright}$ (i.e., $\mathtt{f} \,\underline{\blacktriangleright}\, \bot_\sigma$ for all $\mathtt{f}$); $\bot^\star_\sigma$ is undefined. All symbols are assigned a *status* in $\{Lex, Mul\}$, such that $status(\mathtt{f}) = status(\mathtt{g})$ whenever $\mathtt{f} \approx \mathtt{g}$. Let $\succ^{\mathtt{f}}_\star$ denote either the lexicographic or multiset extension of $\succ_\star$, depending on the status of $\mathtt{f}$. Now the reduction pair $(\succeq_\star, \succ_\star)$ is given by the rules in Figure 1.

$$
\begin{array}{llllll}
(\succ) & s & \succ_\star & t & \text{if} & s^\star \succeq_\star t \\
(\text{Var}) & x & \succeq_\star & x & \text{if} & x \in \mathcal{V} \\
(\text{Abs}) & \lambda x.s & \succeq_\star & \lambda x.t & \text{if} & s \succeq_\star t \\
(\text{Meta}) & Z[s_1, \dots, s_k] & \succeq_\star & Z[t_1, \dots, t_k] & \text{if} & \text{each } s_i \succeq_\star t_i \\
(\text{Fun}) & \mathtt{f}(s_1, \dots, s_n) & \succeq_\star & \mathtt{g}(t_1, \dots, t_k) & \text{if} & \mathtt{f} \approx \mathtt{g} \text{ and } [s_1, \dots, s_n] \succeq^{\mathtt{f}}_\star [t_1, \dots, t_k] \\
(\text{Put}) & \mathtt{f}(s_1, \dots, s_n) & \succeq_\star & t & \text{if} & \mathtt{f}^\star_\sigma(s_1, \dots, s_n) \succeq_\star t \text{ (for } \mathtt{f}(\vec{s}) : \sigma) \\
(\text{Select}) & \mathtt{f}^\star_\sigma(s_1, \dots, s_n) & \succeq_\star & t & \text{if} & s_i \langle \mathtt{f}^\star_{\tau_1}(\vec{s}), \dots, \mathtt{f}^\star_{\tau_j}(\vec{s}) \rangle \succeq_\star t \text{ (**)} \qquad (**) \\
 & & & & & \text{where } s_i : \tau_1 \to \dots \to \tau_j \to \sigma \\
(\text{FAbs}) & \mathtt{f}^\star_{\sigma \to \tau}(s_1, \dots, s_n) & \succeq_\star & \lambda x.t & \text{if} & \mathtt{f}^\star_\tau(s_1, \dots, s_n, x) \succeq_\star t \\
(\text{Copy}) & \mathtt{f}^\star_\sigma(s_1, \dots, s_n) & \succeq_\star & \mathtt{g}(t_1, \dots, t_k) & \text{if} & \mathtt{f} \blacktriangleright \mathtt{g} \text{ and } \mathtt{f}^\star_{\tau_i}(\vec{s}) \succeq_\star t_i \text{ for } 1 \leq i \leq k \\
(\text{Stat}) & \mathtt{f}^\star_\sigma(s_1, \dots, s_n) & \succeq_\star & \mathtt{g}(t_1, \dots, t_k) & \text{if} & \mathtt{f} \approx \mathtt{g} \text{ and } \mathtt{f}^\star_{\tau_i}(\vec{s}) \succeq_\star t_i \text{ for } 1 \leq i \leq k \\
 & & & & & \text{and } [s_1, \dots, s_n] \succ^{\mathtt{f}}_\star [t_1, \dots, t_k] \\
(\text{Bot}) & s & \succeq_\star & \bot_\sigma & \text{if} & s : \sigma
\end{array}
$$

The notation $s\langle t_1, \dots, t_n \rangle$ applies s to $t_1, \dots, t_n$ in the following sense: $s\langle\rangle = s$ and $(\lambda x.s)\langle t, \vec{u} \rangle = s[x := t]\langle \vec{u} \rangle$ and $\mathtt{f}(\vec{s})\langle t, \vec{u} \rangle = \mathtt{f}^\star_\tau(\vec{s}, t)\langle \vec{u} \rangle$ and also $\mathtt{f}^\star_{\sigma \to \tau}(\vec{s})\langle t, \vec{u} \rangle = \mathtt{f}^\star_\tau(\vec{s}, t)\langle \vec{u} \rangle$.

Figure 1 Rules of `StarHorpo`.

Note that $\succeq_\star$ and $\succ_\star$ only compare terms of the same type, and that marked symbols $\mathtt{f}^\star$ may occur with different types (indicated as subscripts) within a term. Symbols $\mathtt{f}^\star$ may also have varying numbers of arguments, but must always have at least $minar(\mathtt{f})$.

▶ **Example 19.** Given a function symbol $@ : (\sigma \to \tau) \to \sigma \to \tau$ (with σ and τ arbitrary types), we can prove $@(\lambda x.Z[x], Y) \succ Z[Y]$ as follows:

- by $(\succ)$, because $@^\star_\tau(\lambda x.Z[x], Y) \succeq_\star Z[Y]$
- by (Select), because $Z[@^\star_\sigma(\lambda x.Z[x], Y)] \succeq_\star Z[Y]$
- by (Meta), because $@^\star_\sigma(\lambda x.Z[x], Y) \succeq_\star Y$
- by (Select) because $Y \succeq_\star Y$ by (Meta).

Wanda uses `StarHorpo` in combination with *argument functions*: each function symbol $\mathtt{f}$ with $minar(\mathtt{f}) = k$ is mapped to a functionalised term $\lambda x_1 \dots x_k.s$, and in a given functionalised meta-term, all occurrences of $\mathtt{f}(t_1, \dots, t_k)$ are replaced by $s[x_1 := t_1, \dots, x_k := t_k]$. If the reduction pair is required to be strongly monotonic (as is the case for rule removal), then $FV(s)$ must be $\{x_1, \dots, x_k\}$. Argument functions are a generalisation of *argument filterings* [39], and were introduced in [37]. In Wanda, they are not restricted to being used with dependency pairs (unlike [39, 37]), and s is limited to one of three forms: (1) x_i, (2) $\mathtt{f}'(x_{i_1}, \dots, x_{i_n})$ (with $n \leq k$, all x_{i_j} distinct), or (3) $\perp_\sigma$. This effectively extends argument filterings with argument permutations and a mapping to one of the minimal constants $\perp_\sigma$.

Wanda combines the search for a suitable precedence and status function with the search for an argument function, using a SAT encoding following [35, Chapter 8.6].

▶ **Example 20.** Consider the (first-order) AFSM with just one sort $\mathtt{o}$ and the following rules:

$$\mathtt{f}\ X\ (\mathtt{s}\ Y) \Rightarrow \mathtt{g}\ Y\ (\mathtt{s}\ (\mathtt{s}\ X)) \qquad \mathtt{f}\ X\ Y \Rightarrow \mathtt{g}\ \mathtt{a}\ \mathtt{b} \qquad \mathtt{g}\ X\ (\mathtt{s}\ Y) \Rightarrow \mathtt{f}\ Y\ X$$

Then $minar(\mathtt{f}) = minar(\mathtt{g}) = 2$. We use the argument functions $\pi(\mathtt{f}) = \lambda x.\lambda y.\mathtt{f}'(y, x)$ and $\pi(\mathtt{g}) = \lambda x.\lambda y.\mathtt{g}'(x, y)$ and $\pi(\mathtt{s}) = \lambda x.\mathtt{s}'(x)$ and $\pi(\mathtt{a}) = \pi(\mathtt{b}) = \perp_{\mathtt{nat}}$ to get the requirements:

$$\mathtt{f}'(\mathtt{s}'(Y), X) \succ \mathtt{g}'(Y, \mathtt{s}'(\mathtt{s}'(X))) \qquad \begin{array}{rcl} \mathtt{f}'(Y, X) & \succsim & \mathtt{g}'(\perp_{\mathtt{nat}}, \perp_{\mathtt{nat}}) \\ \mathtt{g}'(X, \mathtt{s}'(Y)) & \succ & \mathtt{f}'(X, Y) \end{array}$$

This is easily handled with $\mathtt{f}' \approx \mathtt{g}' \blacktriangleright \mathtt{s}'$, and $status(\mathtt{f}') = status(\mathtt{g}') = Lex$. This example relies on $\mathtt{a}$ and $\mathtt{b}$ being mapped to $\perp_{\mathtt{nat}}$. Such use of a minimal constant originates in [48].

5 Dependency Pairs

After trying to prove non-termination using the methods in §3.1–3.2, and removing as many rules as possible with strongly monotonic reduction pairs, control is passed to the *dependency pair (DP) framework*. Like the first-order DP framework [25], this is an extendable framework for termination (and non-termination), which new termination methods can easily be plugged into in the form of "processors". This framework encompasses all remaining termination techniques, but does not currently contain any processors for non-termination. The DP framework is detailed in [37, 22] and [35, Ch. 6–7]. Let us here consider a high-level overview.

5.1 The DP framework

The relatively simple form of the DP framework in Wanda operates on pairs $(\mathcal{P}, \mathcal{R})$ called *DP problems*. For a given AFSM, an initial pair is generated, which must be proved "finite" (also called "non-looping" in [2, 37]). If this property applies, then the AFSM is terminating.

Now, a *processor* is a function that maps a DP problem ρ to a finite set of DP problems. Wanda has a list of processors M such that ρ is finite if and only if all elements of $M(\rho)$ are finite; moreover, either $M(\rho) = \{\rho\}$, or all elements of $M(\rho)$ are strictly smaller than ρ (counting the number of elements in $\mathcal{P}$ and $\mathcal{R}$). Wanda then applies the following algorithm:

1. Let A be the set containing just the initial DP problem $(\mathcal{P}, \mathcal{R})$.
2. If $A = \emptyset$ then return `YES`.
3. Otherwise, choose an arbitrary element $\rho \in A$.
4. Find the first processor M in the list of processors such that $M(\rho) \neq \{\rho\}$.
5. If such a processor cannot be found, then the process has failed; return `MAYBE`.
6. Otherwise, let $A := (A \setminus \{\rho\}) \cup M(\rho)$, and go back to Item 2.

Note that, throughout the process, we retain the following property: the original AFSM is terminating if the initial DP problem is finite, which holds if and only if all elements in A are finite. This is why the conclusion in Item 2 is correct.

The processors used are, in order: the dependency graph, the subterm criterion, the computable subterm criterion, formative rules, and reduction pairs with usable rules (first polynomial interpretations, then `StarHorpo`). All processors are explained in [35, 22].

5.2 Delegation to a first-order prover

Following [20], the framework starts (as part of Item 1 in §5.1) by identifying the *first-order* rules in the AFSM. These are functionalised and passed to an external first-order termination tool; if the full AFSM is not orthogonal then additionally all rules in $\mathcal{C}_\epsilon = \{\mathtt{c}_\iota(X,Y) \Rightarrow X,\ \mathtt{c}_\iota(X,Y) \Rightarrow Y \mid \iota \in \mathcal{S}\}$ are added (with $\mathtt{c}_\iota : \iota \rightarrow \iota \rightarrow \iota$ fresh function symbols).[1]

If the tool detects termination, then this is stored, as it allows all dependency pairs for these first-order rules to be omitted from the set of generated DPs. If the tool returns `NO` and no $\mathcal{C}_\epsilon$ rules were added, then non-termination is concluded as explained in §3.3. Otherwise, the remaining cases of §3.3 are tested with a dedicated non-termination prover.

5.3 Static and Dynamic DPs

To complete item 1 – so to generate the initial DP problem $(\mathcal{P},\mathcal{R})$ – there are two different approaches, originating from distinct lines of work around the same period [37, 38]. In both cases, an AFSM $(\mathcal{F},\mathcal{R})$ gives an initial DP problem $(\bigcup\{\mathit{DP}(\rho) \mid \rho \in \mathcal{R}\},\ \mathcal{R} \cup \mathit{OptionalExtra})$, where the set $\mathit{DP}(\rho)$ of dependency pairs generated for a given rule varies between the two approaches. In both cases, the elements of $\mathit{DP}(\rho)$ with ρ a first-order rule may be omitted if the first-order part was proved terminating following §5.2. Unlike the name suggests (as this differs from the first-order definition), these dependency pairs are actually *triples* of a pattern of the form $\mathtt{f}\ \ell_1 \dots \ell_n$, a meta-term r and a set; this is discussed in more detail in [35, 22].

In the *dynamic* approach, each $\mathit{DDP}(\rho)$ contains triples whose second component r has a form $\mathtt{g}\ r_1 \cdots r_m$ or $Z[r_1,\dots,r_m]$; the latter kind is called a "collapsing" DP. In the *static* approach, $\mathit{SDP}(\rho)$ contains no collapsing DPs, but may have DPs where $\mathit{FMV}(r) \not\subseteq \mathit{FMV}(\ell)$. Both fresh meta-variables in r and collapsing DPs are complications not present in the first-order setting, which make some of the processors weaker. The static approach for generating DPs can only be used if some restrictions on the AFSM are satisfied, but when applicable often gives an easier termination proof than the dynamic one.

The notion of a finite problem and the processors used in Wanda can all be defined generally enough to apply for both the static and dynamic approach. Hence, once the initial DP problem is generated, the same DP framework can be used for both. Wanda tries dynamic DPs first, and if this fails, falls back to static DPs. However, if $\bigcup\{\mathit{SDP}(\rho) \mid \rho \in \mathcal{R}\} \subseteq \bigcup\{\mathit{DDP}(\rho) \mid \rho \in \mathcal{R}\}$, this first step is omitted and only the static approach is tried.

▶ **Example 21.** For $\mathcal{R}_1$ in Example 6, the dynamic approach generates $(\{(1),(2)\},\mathcal{R}_1)$ with:

$$\begin{array}{lllll} (1) & \mathtt{map}^\sharp\ (\lambda x.Z[x])\ (\mathtt{cons}\ H\ T) & \Rrightarrow & \mathtt{map}^\sharp\ (\lambda x.Z[x])\ T & (\emptyset) \\ (2) & \mathtt{map}^\sharp\ (\lambda x.Z[x])\ (\mathtt{cons}\ H\ T) & \Rrightarrow & Z[H] & (\emptyset) \end{array}$$

The static approach generates $(\{(1)\},\mathcal{R}_1)$. Thus, Wanda does not try the dynamic approach.

6 Experimental results

To test the power of both Wanda as a whole, and individual techniques, various configurations of Wanda were tested on two data sets: (1) the "higher order union beta" benchmarks in the Termination Problem DataBase [13] (which are used in the International Termination

[1] These rules allow for the construction of a term that can be reduced to all elements of an arbitrary finite set of terms with the same type. They are trivially discarded by many termination techniques, but may complicate analysis because they turn the system non-confluent.

	YES	NO	MAYBE	TIMEOUT	Avg. time
Full	188	16	25	32	1.14
Only rule removal	123	0	118	20	1.13
Only `StarHorpo`	111	0	141	9	0.24
Only interpretations	59	0	156	46	0.07
Only dependency pairs	186	0	42	33	1.02
only static DPs	152	0	86	23	0.55
only dynamic DPs	167	0	58	36	1.30
no first-order tool	183	9	47	22	0.90
no overestimation	155	16	25	65	0.75

Figure 2 Experimental results on the TPDB (261 benchmarks).

Competition [12]), and (2) the pattern HRSs in the COPS (Confluence Problems) database [10] (which are used in the International Confluence Competition [11]), most of which were translated to AFSMs by the tool CSI^ho [42]. Wanda was executed with a timeout of 60 seconds, on a Lenovo Thinkpad T420, using AProVE [24] as a first-order termination prover, and MiniSAT [16] as a SAT-solver. The results are discussed below. Note that the average time only takes `YES` and `NO` results into account; in particular, `TIMEOUT`s are not considered.

An evaluation page with detailed results is available at:

`https://www.cs.ru.nl/~cynthiakop/experiments/fscd20/`

6.1 Benchmarks from the TPDB

The results on the termination problem database are given in Figure 2. The first test is Wanda's default behaviour, the next three use only rule removal (with both techniques or only one), and the next three use only the DP framework (either full or with only one way of generating the initial DP problem). The final tests disable specific features in the full version: using a first-order termination tool, and overestimating AFSs as described in §2.2. The longest successful evaluation is 20.46 seconds, so not close to the 60 second timeout.

The tests show that rule removal is not as effective as dependency pairs, but does help a little: when it is disabled, Wanda loses two benchmarks (and does not gain any). This could be avoided by implementing rule removal as a processor in the DP framework, but this has thus far not been done (the implementation is not entirely straightforward due to the different requirements imposed by the DP framework). The effect of rule removal on speed is variable: rule removal often *succeeds* fast, but may take a long time to *fail*. Thus, when both are tried, the solution speed could go either way. Within rule removal, `StarHorpo` is much more powerful than polynomial interpretations, but the techniques are incomparable: there are 12 benchmarks that can be handled by interpretations but not `StarHorpo`.

Also the two styles of dependency pairs are incomparable: the dynamic approach seems to give a bit more power, but there are benchmarks that can be handled with static DPs and not with dynamic ones. Moreover, the static approach is significantly faster.

Worth noting is that there are 16 benchmarks Wanda can prove non-terminating, of which 7 are found by AProVE. Of the remainder, manual checking shows that 7 have obvious loops, and 2 admit the $\omega\omega$ example. For termination, using AProVE gives a modest gain (five benchmarks). The last row deserves some further discussion. Due to unclear documentation on the competition's format, the 85 newest benchmarks in this database are all "fake HRS": like the system in Example 11, the left-hand sides often have subterms such as $F\ x\ y$ where

	YES	NO	MAYBE	TIMEOUT	Avg. time
Full	43	30	19	1	0.09
Only rule removal	37	0	56	0	0.11
Only `StarHorpo`	33	0	60	0	0.17
Only interpretations	21	0	72	0	0.01
Only dependency pairs	43	0	49	1	0.51
only static DPs	37	0	56	0	0.26
only dynamic DPs	40	0	52	1	0.77
no first-order tool	43	30	19	1	0.07

Figure 3 Experimental results on the COPS database (93 benchmarks).

F is a free variable. Wanda spends more time on these benchmarks than others, since not only the true translation to AFSM is considered, but also an overestimation that is often easier to handle. When overestimating is disabled, Wanda is faster, but significantly weaker.

The first-order tool. It is worth noting that more than fifty benchmarks in this database are actually first-order systems with one or two (typically trivial) higher-order rules. Indeed, about 25 of Wanda's `TIMEOUT`s are due to AProVE timing out on a complicated first-order fragment. This raises the question whether the choice of first-order tool is significant.

The answer is ambiguous. For *non-termination*, Wanda relies on an explicit counterexample, which only the customised version of AProVE provides; without it, Wanda loses 7 `NO`s. For *termination*, comparing Wanda's performance when instead coupled with NaTT or MU-TERM, we found that NaTT outperforms both AProVE and MU-TERM by 13 benchmarks. However, this advantage is local: the "higher-order union beta" category of the TPDB has seven sub-directories, each representing a batch of (often similar) benchmarks that were added at the same time. On six of those seven, Wanda performs almost identically whichever first-order tool is used: MU-TERM and AProVE give one benchmark that NaTT fails, and all other answers are the same. In the seventh, NaTT wins 14 benchmarks over the others.

Looking at all benchmarks, we observe: the *only* cases where using a first-order tool helps, are combinations of a challenging first-order TRS and a quite simple higher-order part: it can be handled with static DPs and one of the subterm criterion processors [22]. Which first-order tool is the best for the job depends only on the form of the first-order part.

6.2 Benchmarks from COPS

Figure 3 shows the experimental results on AFSMs translated from the Confluence Problems database (COPS) [10]. Here, unlike the benchmarks from the TPDB, meta-variables are used with arguments. Even so, the comparative results between rule removal and full Wanda, and between static, dynamic and full dependency pairs, are similar to the TPDB results. There are relatively far more `NO` answers, which seems to be because COPS contains more non-terminating systems (and quite a few trivially so). This is explained by the purpose of the database: confluence is harder to prove for non-terminating than terminating systems.

7 Practical use

Wanda is designed to run on a Linux terminal, and is invoked by supplying one or more input files, and zero or more runtime parameters that customise the behaviour. Runtime parameters range from purely aesthetical commands (e.g., to indicate that Wanda should use coloured output), to commands that make Wanda output properties of the given system (e.g., to indicate whether a system has η-long form) or that modify the termination checking behaviour (e.g., the previously mentioned `betafirst` parameter). Wanda has a fixed strategy – that is, techniques are always applied in the same order – but certain techniques can be disabled for practical experiments; this was done in §6. The full range of parameters is documented in the `README.txt` file included with the distribution. Some pertinent commands are:

- `-d` ⟨`methods`⟩ disables the given methods; for example, use `./wanda.exe -d nt,poly,dp` to disable non-termination analysis, algebra interpretations and dependency pairs; this forces Wanda to generate a proof using `StarHorpo`, if one can be found;
- `-i` ⟨`tool`⟩ tells Wanda to use the given first-order termination tool as back-end, which must be located in the `resources/` sub-directory. If not given, Wanda uses the file "firstorder-prover". Similarly, `-n` ⟨`tool`⟩ tells Wanda to use the given tool for *non-termination* analysis when this is done in a separate step.

In standard usage, Wanda takes an input file describing an AFSM or AFS, performs an analysis following §3–5 and then prints `YES` (a termination proof was found), `NO` (a non-termination proof was found) or `MAYBE` (neither could be proved). In the first two cases, this is followed by a human-readable proof. If more than one input file is supplied, Wanda prints the name of each file, followed by the answer and possibly proof. A timeout may be supplied (following the standard for the termination competition) but is ignored.

8 Conclusions and directions for future work

This paper has discussed the various techniques used in Wanda, and how they are applied. Wanda is only one of several higher-order tools, and interestingly, *incomparable* to others: there are benchmarks that Wanda can handle and other tools cannot, and vice versa. This is because all tools that have participated in the Termination Competition have focused on different techniques. For Wanda, the main termination approach is the DP framework.

There are many directions for improvement. Most pertinently, due to the presence of a large database of termination benchmarks in the competition format [13], Wanda has been optimised for AFSs and is decidedly weak in the presence of meta-variables with arguments. Moreover, non-termination analysis is very limited and does not take advantage of the DP framework. Other improvements could be to further extend first-order termination techniques, to build on primarily higher-order techniques like sized types [5], and to support AFSMs with polymorphic types. Automatic certification as has been done for first-order rewriting [49] would be a highly interesting direction to pursue, but would require a vast amount of work to build up the formalisation library. Finally, Wanda's usability could be substantially improved by the addition of a web interface, for example using the EasyInterface toolkit [15].

A complete discussion of most techniques in Wanda and the technology behind automating them is available in the author's PhD thesis [35]. Wanda is open-source and available from `http://wandahot.sourceforge.net/`. The snapshot that was used in the present paper (including both open- and closed-source back-ends) is available from the evaluation pages:

`https://www.cs.ru.nl/~cynthiakop/experiments/fscd20/`

References

1 P. Aczel. A general Church-Rosser theorem, 1978. Unpublished Manuscript, University of Manchester. http://www.ens-lyon.fr/LIP/REWRITING/MISC/AGeneralChurch-RosserTheorem.pdf.

2 T. Aoto and Y. Yamada. Argument filterings and usable rules for simply typed dependency pairs. In *Proceedings of FroCoS*, volume 5749 of *LNAI*, pages 117–132. Springer, 2009.

3 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

4 F. Blanqui. HOT – an automated termination prover for higher-order rewriting. URL: http://rewriting.gforge.inria.fr/hot.html.

5 F. Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In *Proceedings of RTA*, volume 3091 of *LNCS*, pages 24–39. Springer, 2004.

6 F. Blanqui, J. Jouannaud, and M. Okada. Inductive-data-type systems. *Theoretical Computer Science*, 272(1-2):41–68, 2002.

7 F. Blanqui, J. Jouannaud, and A. Rubio. The computability path ordering: The end of a quest. In *Proceedings of CSL*, volume 5213 of *LNCS*, pages 1–14. Springer, 2008.

8 C. Borralleras and A. Rubio. THOR – an automatic tool for proving termination of higher-order rewriting. URL: https://www.cs.upc.edu/~albert/term.html.

9 C. Borralleras and A. Rubio. A monotonic higher-order semantic path ordering. In *Proceedings of LPAR*, volume 2250 of *LNAI*, pages 531–547. Springer, 2001.

10 Community. Confluence Problems (COPS). URL: https://cops.uibk.ac.at/?q=prs.

11 Community. The international Confluence Competition (CoCo). URL: http://coco.nue.riec.tohoku.ac.jp/.

12 Community. Termination Portal. URL: http://www.termination-portal.org/wiki/Termination_Competition.

13 Community. Termination Problem DataBase (TPDB). URL: http://termination-portal.org/wiki/TPDB.

14 N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

15 J. Doménech, S. Genaim, E.B. Johnsen, and R. Schlatte. EasyInterface: A toolkit for rapid development of GUIs for research prototype tools. In *Proceedings of FASE*, volume 10202 of *LNCS*, pages 379–383. Springer, 2017.

16 N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03)*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004. See also http://minisat.se/.

17 F. Emmes, T. Enger, and J. Giesl. Proving non-looping non-termination automatically. In *Proceedings of IJCAR*, volume 7364 of *LNAI*, pages 225–240. Springer, 2012.

18 M. Ferreira and H. Zantema. Syntactical analysis of total termination. In *Proceedings of ALP*, volume 850 of *LNCS*, pages 204–222. Springer, 1994.

19 C. Fuhs, J. Giesl, M. Parting, P. Schneider-Kamp, and S. Swiderski. Proving termination by dependency pairs and inductive theorem proving. *Journal of Automated Reasoning*, 47(2):133–160, 2011.

20 C. Fuhs and C. Kop. Harnessing first order termination provers using higher order dependency pairs. In *Proceedings of FroCoS*, volume 6989 of *LNAI*, pages 147–162. Springer, 2011.

21 C. Fuhs and C. Kop. Polynomial interpretations for higher-order rewriting. In *Proceedings of RTA*, volume 15 of *LIPIcs*, pages 176–192. Dagstuhl, 2012.

22 C. Fuhs and C. Kop. A static higher-order dependency pair framework. In *Proceedings of ESOP*, volume 11423 of *LNCS*, pages 752–782, 2019.

23 G. Genestier. SizeChangeTool: A termination checker for rewriting dependent types. In *Proceedings of HOR*, pages 14–19, 2019. URL: https://hal.archives-ouvertes.fr/hal-02442465/document.

24 J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning*, 58(1):3–31, 2017.

25 J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proceedings of LPAR*, volume 3452 of *LNAI*, pages 301–331. Springer, 2005.

26 J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proceedings of FroCoS*, volume 3717 of *LNAI*, pages 216–231. Springer, 2005.

27 B. Gramlich. Abstract relations between restricted termination and confluence properties of rewrite systems. *Fundamenta Informaticae*, 24(1-2):3–23, 1995.

28 M. Hamana. PolySOL – an automatic tool for confluence and termination of polymorphic second-order systems. URL: `http://www.cs.gunma-u.ac.jp/hamana/polysol/`.

29 J. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *Proceedings of LICS*, IEEE, pages 402–411, 1999.

30 S. Kamin and J.-J. Lévy. Two generalizations of the recursive path ordering, 1980. Unpublished Manuscript, University of Illinois.

31 D. Kapur, P. Musser, D. Narendran, and J. Stillman. Semi-unification. In *Proceedings of FSTTCS*, volume 338 of *LNCS*, pages 435–454, 1988.

32 J.W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121(1-2):279–308, 1993.

33 J.W. Klop, V. van Oostrom, and R. de Vrijer. Iterative lexicographic path orders. In *Essays dedicated to Joseph A. Goguen on the Occasion of his 65th Birthday*, volume 4060 of *LNCS*, pages 541–554. Springer, 2006. Festschrift.

34 C. Kop. Simplifying algebraic functional systems. In *Proceedings of CAI*, volume 6742 of *LNCS*, pages 201–215. Springer, 2011.

35 C. Kop. *Higher Order Termination.* PhD thesis, VU University Amsterdam, 2012.

36 C. Kop and F. van Raamsdonk. A higher-order iterative path ordering. In *Proceedings of LPAR*, volume 5330 of *LNAI*, pages 697–711, 2008.

37 C. Kop and F. van Raamsdonk. Dynamic dependency pairs for algebraic functional systems. *Logical Methods in Computer Science*, 8(2):10:1–10:51, 2012. Special Issue for RTA '11.

38 K. Kusakari, Y. Isogai, M. Sakai, and F. Blanqui. Static dependency pair method based on strong computability for higher-order rewrite systems. *IEICE Transactions on Information and Systems*, 92(10):2007–2015, 2009.

39 K. Kusakari, M. Nakamura, and Y. Toyama. Argument filtering transformation. In *Proceedings of PPDP*, volume 1702 of *LNCS*, pages 47–61. Springer, 1999.

40 S. Lucas, C. Marché, and J. Meseguer. Operational termination of conditional term rewriting systems. *Information Processing letters*, 95(4):446–453, 2005.

41 D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

42 J. Nagele. CoCo 2016 participant: CSI^ho 0.2. ; tool webpage: `http://cl-informatik.uibk.ac.at/software/csi/ho/`. URL: `http://coco.nue.riec.tohoku.ac.jp/2016/papers/csiho.pdf`.

43 T. Nipkow. Higher-order critical pairs. In *Proceedings of LICS*, pages 342–349. IEEE, 1991.

44 K. Onozawa, K. Kikuchi, T. Aoto, and Y. Toyama. ACPH: System description for CoCo 2016. URL: `http://coco.nue.riec.tohoku.ac.jp/2016/papers/acph.pdf`.

45 É. Payet. Loop detection in term rewriting using the eliminating unfoldings. *Theoretical Computer Science*, 403(2-3):307–327, 2008.

46 É. Payet. Guided unfoldings for finding loops in standard term rewriting. In *Proceedings of LOPSTR*, volume 11408 of *LNCS*, pages 22–37, 2018.

47 J.C. van de Pol. *Termination of Higher-order Rewrite Systems*. PhD thesis, University of Utrecht, 1996.

48 R. Thiemann, G. Allais, and J. Nagele. On the formalization of termination techniques based on multiset orderings. In *Proceedings of RTA*, volume 15 of *LIPIcs*, pages 339–354. Dagstuhl, 2012.

49 R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proceedings of TPHOLs*, volume 5674 of *LNCS*, pages 452–468. Springer, 2009.

50 Y. Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25(1):141–143, 1987.

51 H. Zantema. Termination of context-sensitive rewriting. In *Proceedings of RTA*, volume 1232 of *LNCS*, pages 172–186, 1997.

A Type Checker for a Logical Framework with Union and Intersection Types

Claude Stolze
IRIF, Université de Paris, France
Claude.Stolze@irif.fr

Luigi Liquori
Université Côte d'Azur, Nice, France
Inria Sophia Antipolis - Méditerranée, France
Luigi.Liquori@inria.fr

Abstract

We present the syntax, semantics, typing, subtyping, unification, refinement, and REPL of Bull, a prototype theorem prover based on the Δ-Framework, i.e. a fully-typed Logical Framework à la Edinburgh LF decorated with union and intersection types, as described in previous papers by the authors. Bull also implements a subtyping algorithm for the Type Theory Ξ of Barbanera-Dezani-de'Liguoro. Bull has a command-line interface where the user can declare axioms, terms, and perform computations and some basic terminal-style features like error pretty-printing, subexpressions highlighting, and file loading. Moreover, it can typecheck a proof or normalize it. These terms can be incomplete, therefore the typechecking algorithm uses unification to try to construct the missing subterms. Bull uses the syntax of Berardi's Pure Type Systems to improve the compactness and the modularity of the kernel. Abstract and concrete syntax are mostly aligned and similar to the concrete syntax of Coq. Bull uses a higher-order unification algorithm for terms, while typechecking and partial type inference are done by a bidirectional refinement algorithm, similar to the one found in Matita and Beluga. The refinement can be split into two parts: the essence refinement and the typing refinement. Binders are implemented using commonly-used de Bruijn indices. We have defined a concrete language syntax that will allow user to write Δ-terms. We have defined the reduction rules and an evaluator. We have implemented from scratch a refiner which does partial typechecking and type reconstruction. We have experimented Bull with classical examples of the intersection and union literature, such as the ones formalized by Pfenning with his Refinement Types in LF and by Pierce. We hope that this research vein could be useful to experiment, in a proof theoretical setting, forms of polymorphism alternatives to Girard's parametric one.

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

Keywords and phrases Intersection types, Union types, Dependent types, Subtyping, Type checker, Refiner, Δ-Framework

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.37

Category System Description

Supplementary Material BULL: `https://github.com/cstolze/Bull`

Funding Work supported by the COST Action CA15123 EUTYPES "The European research network on types for programming and verification"

Acknowledgements This work could not be have be done without the many useful discussions with Furio Honsell, Ivan Scagnetto, Ugo de'Liguoro, Daniel Dougherty, and the Anonymous Reviewers.

1 Introduction

This paper provides a unifying framework for two hitherto unreconciled understandings of types: i.e. types-as-predicates à la Curry and types-as-propositions à la Church. The key to our unification consists in introducing, implementing and experimenting *strong proof-*

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 37; pp. 37:1–37:24
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

functional connectives [40, 2, 3] in a dependent type theory such as the Edinburgh Logical Framework (LF) [21]. Both Logical Frameworks and Proof-Functional Logic consider proofs as first-class citizens, albeit differently.

Strong proof-functional connectives take seriously into account the shape of logical proofs, thus allowing for polymorphic features of proofs to be made explicit in formulæ. Hence they provide a finer semantics than classical/intuitionistic connectives, where the meaning of a compound formula depends only on the *truth value* or the *provability* of its subformulæ. However, existing approaches to strong proof-functional connectives are all quite idiosyncratic in mentioning proofs. Existing Logical Frameworks, on the other hand, provide a uniform approach to proof terms in object logics, but they do not fully capitalize on subtyping.

This situation calls for a natural combination of the two understandings of types, which should benefit both worlds. On the side of Logical Frameworks, the expressive power of the metalanguage would be enhanced thus allowing for shallower encodings of logics, a more principled use of subtypes [36], and new possibilities for formal reasoning in existing interactive theorem provers. On the side of type disciplines for programming languages, a principled framework for proofs would be provided, thus supporting a uniform approach to "proof reuse" practices based on type theory [14, 38, 11, 20, 8].

Therefore, in [25] we extended LF with the connectives of *strong intersection* (corresponding to intersection types [4, 5]) and *strong union* (corresponding to union types [31, 2]) of Proof-Functional Logic [40]. We called this extension the Δ-Framework (LF$_\Delta$), since it builds on the Δ-calculus [28]. As such, LF$_\Delta$ subsumes many expressive type disciplines in the literature [36, 2, 3, 38, 11].

It is not immediate to extend the Curry-Howard isomorphism to logics supporting strong proof-functional connectives, since these connectives need to compare the shapes of derivations and do not just take into account the provability of propositions, i.e. the inhabitation of the corresponding type. In order to capture successfully strong logical connectives such as ∩ or ∪, we need to be able to express the rules:

$$\frac{\mathcal{D}_1 : A \quad \mathcal{D}_2 : B \quad \mathcal{D}_1 \equiv_{\mathcal{R}} \mathcal{D}_2}{A \cap B}\ (\cap I) \qquad \frac{\mathcal{D}_1 : A \supset C \quad \mathcal{D}_2 : B \supset C \quad A \cup B \quad \mathcal{D}_1 \equiv_{\mathcal{R}} \mathcal{D}_2}{C}\ (\cup E)$$

where $\equiv_{\mathcal{R}}$ is a suitable equivalence between logical proofs. Notice that the above rules suggest immediately intriguing applications in polymorphic constructions, i.e. the *same evidence* can be used as a proof for different statements.

Pottinger [40] was the first to study the strong connective ∩. He contrasted it to the intuitionistic connective ∧ as follows: *"The intuitive meaning of* ∩ *can be explained by saying that to assert* $A \cap B$ *is to assert that one has a reason for asserting* A *which is also a reason for asserting* B *[while] to assert* $A \wedge B$ *is to assert that one has a pair of reasons, the first of which is a reason for asserting* A *and the second of which is a reason for asserting* B*"*.

A logical theorem involving intuitionistic conjunction which does not hold for strong conjunction is $(A \supset A) \wedge (A \supset B \supset A)$, otherwise there should exist a closed λ-term having simultaneously both one and two abstractions. López-Escobar [29] and Mints [33] investigated extensively logics featuring both strong and intuitionistic connectives especially in the context of *realizability* interpretations.

Dually, it is in the ∪-elimination rule that proof equality needs to be checked. Following Pottinger, we could say that *asserting* $(A \cup B) \supset C$ *is to assert that one has a reason for* $(A \cup B) \supset C$*, which is also a reason to assert* $A \supset C$ *and* $B \supset C$. The two connectives differ since the intuitionistic theorem $((A \supset B) \vee B) \supset A \supset B$ is not derivable for ∪, otherwise there would exist a term which behaves both as **I** and as **K**.

Strong connectives arise naturally in investigating the propositions-as-types analogy for intersection and union type assignment systems. From a logical point of view, there are many proposals [33, 36, 47, 42, 34, 10, 9, 39, 19, 18] to find a suitable logic to fit intersection and union: we also refer to [15, 25, 43] for a detailed discussion.

The LF_Δ Logical Framework introduced in [25] extends [28] with union types and dependent types. The novelty of LF_Δ in the context of Logical Frameworks, lies in the full-fledged use of strong proof-functional connectives, which to our knowledge has never been explored before. Clearly, all Δ-terms have a computational counterpart.

Useful applications rely on using Proof-Functional Logics (that care about the proof-derivation), instead of Truth-Functional Logics (that care about the validity of the conclusion). In a nutshell: *i*) Intersection-types better catch the class of strongly normalizing terms than $F\omega$ (see, among others, [46]). *ii*) Union and intersection types allow for a little form of abstract interpretation (see Pierce $IsZero$ example [38, 2]) that cannot easily be encoded in $LF\omega$. *iii*) Proof-Functional Logics introduces a notion of "proof-synchronization", as showed in the operational semantics of the Δ-calculus, see Definition 4.3 and 5 of [28]: two proofs are "in-sync" iff their "untyped essences" are equals. This could be possibly exploited in a synchronized tactic in Bull. *iv*) Union types can capture a natural form of parallelism that could be conveniently put to use in formalizing reasoning on concurrent execution, as in proving correctness of concurrency control protocols or in branch-prediction techniques.

This paper presents the implementation of Bull [44, 43], an Interactive Theorem Prover (ITP) based on the Δ-Framework [45, 25]. The first author wrote this theorem prover from scratch for three years. Bull have a command-line interface program where the user can declare axioms, terms, and perform computations. These terms can be incomplete, therefore the typechecking algorithm uses unification to try to construct the missing subterms.

We have designed and implemented a novel subtyping algorithm [27] which extends the well-known algorithm for intersection types, designed by Hindley [23], with union types. Our subtyping algorithm has been mechanically proved correct in Coq and extracted in OCaml, extending the proof of a subtyping algorithm for intersection types of Bessai et al. [7].

We have implemented several features. A *Read-Eval-Print-Loop* (REPL) allows to define axioms and definitions, and performs some basic terminal-style features like error pretty-printing, subexpressions highlighting, and file loading. Moreover, it can typecheck a proof or normalize it. We use the Berardi's syntax of Pure Type Systems [6] to improve the compactness and the modularity of the kernel. Abstract and concrete syntax are mostly aligned: the concrete syntax is similar to the concrete syntax of Coq.

We have designed a *higher-order unification algorithm* for terms, while typechecking and partial type inference are done by our *bidirectional refinement algorithm*, similar to the one found in Matita [1]. The refinement can be split into two parts: the essence refinement and the typing refinement. The bidirectional refinement algorithm aims to have partial type inference, and to give as much information as possible to the unifier. For instance, if we want to find a $?y$ such that $\vdash_\Sigma \langle \lambda x{:}\sigma.x, \lambda x{:}\tau.?y\rangle : (\sigma \to \sigma) \cap (\tau \to \tau)$, then we can infer that $x{:}\tau \vdash ?y : \tau$ and that $\wr\, ?y \,\wr =_\beta x$.

This paper is organized as follows: in Section 2, we introduce the language we have implemented. In Section 3, we define the reduction rules and explain the evaluation process. In Section 4, we present the subtyping algorithm. In Section 5, we present the unifier. In Section 6, we present the refiner which does partial typechecking and type reconstruction. In Section 7, we present the REPL. In Section 8, we present possible enhancements of the type theory and of the ITP. Appendix contains interesting encodings that could be typechecked in Bull and could help the reader to understand the usefulness of adding ad hoc polymorphism and proof-functional operators to LF.

2 Syntax of terms

The syntax for the logical framework we have designed and implemented is as follows:

$$\begin{array}{rcll}
\Delta,\sigma & ::= & s \;,\; c \;,\; v \;,\; \_ \;,\; ?x[\Delta;...;\Delta] & \text{Sorts, Const, Vars, Placeholders and Metavars}\\
 & | & \textbf{let } x{:}\sigma := \Delta \textbf{ in } \Delta & \text{Local definition}\\
 & | & \Pi x{:}\sigma.\Delta \;,\; \lambda x{:}\sigma.\Delta \;,\; \Delta\, S & \Pi\text{- and } \lambda\text{-abstraction and application}\\
 & | & \sigma \cap \sigma \;,\; \sigma \cup \sigma & \text{Intersection and Union types}\\
 & | & \langle \Delta, \Delta \rangle \;,\; \mathsf{pr}_1\, \Delta \;,\; \mathsf{pr}_2\, \Delta & \text{Strong pair and Left/Right projections}\\
 & | & \textbf{smatch } \Delta \textbf{ return } \sigma \textbf{ with } [x{:}\sigma \Rightarrow \Delta \mid x{:}\sigma \Rightarrow \Delta] & \text{Strong sum}\\
 & | & \mathsf{in}_1\, \sigma\, \Delta \;,\; \mathsf{in}_2\, \sigma\, \Delta \;,\; \textbf{coe}\, \sigma\, \Delta & \text{Left/Right injections and Coercions}\\
S & ::= & () \mid (S;\Delta) & \text{Typed Spines}
\end{array}$$

By using a Pure Type System approach [6], all the terms are read through the same parser. The main differences with the Δ-Framework [25] are the additions of a placeholder and meta-variables, used by the refiner. We also added a **let** operator and changed the syntax of the strong sum **smatch** so it looks more like the concrete syntax used in the implementation. A meta-variable $?x[\Delta_1;...;\Delta_n]$ has the, so called, *suspended substitutions* $\Delta_1;...;\Delta_n$, which will be explained clearly in Subsection 2.4. Finally, following the Cervesato-Pfenning jargon [12], applications are in *spine form*, i.e. the arguments of a function are stored together in a list, exposing the head of the term separately. We also implemented a corresponding syntax for the untyped counterpart of the framework, called *essence* [28], where all pure λ-terms M and spines are defined as follows:

$$\begin{array}{rcll}
M,\varsigma & ::= & s \;,\; c \;,\; x \;,\; \_ \;,\; ?x[M;...;M] & \text{Sorts, Const, Vars, Placeholders and Metavars}\\
 & | & \textbf{let } x := M \textbf{ in } M & \text{Local definition}\\
 & | & \Pi x{:}\varsigma.\varsigma \;,\; \lambda x.M \;,\; M\, R & \Pi\text{- and } \lambda\text{-abstraction and application}\\
 & | & \varsigma \cap \varsigma \;,\; \varsigma \cup \varsigma & \text{Intersection and Union types}\\
R & ::= & () \mid (R;M) & \text{Untyped Spines}
\end{array}$$

Note that essences of types (ς) belongs to the same syntactical set as essences of terms.

2.1 Concrete syntax

The concrete syntax of the terms has been implemented with OCamllex and OCamlyacc. Its simplified syntax is as follows:

```
term ::= Type                              # type
| let ID [args] [: term] := term in term   # let
| ID                            # variables and constants
| forall args, term             # dependent product
| term -> term                  # non-dependent product
| fun args => term              # lambda-abstraction
| term term                     # application
| term & term                   # intersection of types
| term | term                   # union of types
| <term,term>                   # strong pair
| proj_l term                   # left  projection of a strong pair
| proj_r term                   # right projection of a strong pair
| smatch term [as ID] [return term] with ID [: term] => term, ID [: term] => term end
| inj_l term term               # left  injection of a strong sum        # strong sum
| inj_r term term               # right injection of a strong sum
| coe term term                 # coercion
| _                             # wildcard
```

Identifiers `ID` refers to any alphanumeric string (possibly with underscores and apostrophes). The non-terminal symbol `args` correspond to a non-empty sequence of arguments, where an argument is an identifier, and can be given with its type. In the latter case, you should parenthesize it, for instance `(x : A)`, and if you want to assign the same type to several consecutive arguments, you can e.g. write `(x y z : A)`. Strong sums have a complicated syntax. For instance, consider this term:

```
smatch foo as x return T with y : T1 ⇒ bar, z : T2 ⇒ baz end
```

The above term in the concrete syntax corresponds to
smatch foo **return** λx:_.T **with** [y:T1 ⇒ bar | z:T2 ⇒ baz]
in the abstract syntax. The concrete syntax thus guarantees that the returned type is a λ-abstraction, and it allows a simplified behaviour of the type reconstruction algorithm. The behaviour of the concrete syntax is intended to mimic Coq.

2.2 Implementation of the syntax

In the OCaml implementation, Δ-terms and their types along with essences and type essences are represented with a single type called `term`. It allows some functions (such as the normalization function) to be applied both on Δ-terms and on essences.

```
type term =
  | Sort of location * sort
  | Let of location * string * term * term * term (* let s : t1 := t2 in t3 *)
  | Prod of location * string * term * term (* forall s : t1, t2 *)
  | Abs of location * string * term * term (* fun s : t1 => t2 *)
  | App of location * term * term list (* t t1 t2 ... tn *)
  | Inter of location * term * term (* t1 & t2 *)
  | Union of location * term * term (* t1 | t2 *)
  | SPair of location * term * term (* < t1, t2 > *)
  | SPrLeft of location * term (* proj_l t1 *)
  | SPrRight of location * term (* proj_r t1 *)
  | SMatch of location * term * term * string * term * term * string * term * term
                    (* match t1 return t2 with s1 : t3 => t4 , s2 : t5 => t6 end *)
  | SInLeft of location * term * term (* inj_l t1 t2 *)
  | SInRight of location * term * term (* inj_r t1 t2 *)
  | Coercion of location * term * term (* coe t1 t2 *)
  | Var of location * int (* de Bruijn index *)
  | Const of location * string (* variable name *)
  | Underscore of location (* meta-variables before analysis *)
  | Meta of location * int * (term list) (* index and substitution *)
```

The constructors of `term` contain the location information retrieved by the parser that allows the typechecker to give the precise location of a subterm to the user, in case of error.

The `App` constructor takes as parameters the applied function and the list of all the arguments. The list of parameters is used as a stack, hence the rightmost argument is the head of the list, and can easily be removed in the OCaml recursive functions. The variables are referred to as strings in the `Const` constructor, and as de Bruijn indices in `Var` constructors.

The parser does not compute de Bruijn indices, it gives the variables as strings. The function `fix_index` replaces bound variables by de Bruijn indices. We still keep track of the string names of the variables, in case we have to print them back. Its converse function, `fix_id`, replaces the de Bruijn indices with the previous strings, possibly updating the string names in case of name clashes. For instance, the string printed to the user, showing the normalized form of `(fun (x y : nat) => x) y`, is `fun y0 : nat => y :` the bound variable `y` has been renamed `y0`. The meta-variables are generated by the typecheckers, and their identifier is an integer. We have defined several helper functions to ease the process of terms.

The generic function `visit_term f g h t` looks at the children of the term `t`, and: *i*) every child `t1` outside of a binder is replaced with `f t1`; *ii*) every child `t1` inside the binding of a variable whose name (a string) is `s` is replaced with `g s t1`, while `s` is replaced with `h s t1`.

The functions `g` and `h` take a string as an argument, for helping the implementation of the `fix_index` and `fix_id` functions.

The function `map_term` is a kind of mapping function: `map_term k f t` finds every variable `Var(l, n)` inside the term `t`, and replaces it by `f (k+offset) l n`, where `offset` is the number of extra bindings.

The `lift` and `map_term` functions allow us to define a substitution in a clean way:

```
(* update all indices greater than k by adding n to them *)
let lift k n = map_term k (fun k l m -> if m < k then Var (l, m) else Var (l, m+n))

(* Transform (lambda x. t1) t2 into t1[t2/x] *)
let beta_redex t1 t2 =
  let subst k l m =
    if m < k then Var (l, m) (* bound variable *)
    else if m = k then (* x *)
      lift 0 k t2
    else (* the enclosing lambda goes away *)
      Var (l, m-1)
  in map_term 0 subst t1
```

2.3 Environments

There are four kinds of environments, namely:

1. the *global environment* (noted Σ). The global environment holds constants which are fully typechecked: $\Sigma ::= \cdot \mid \Sigma, c{:}\varsigma@\sigma \mid \Sigma, c := M@\Delta : \varsigma@\sigma$. Intuitively, $c{:}\varsigma@\sigma$ is a declaration of a constant (or axiom), and $c := M@\Delta : \varsigma@\sigma$ corresponds to a global definition.
2. the *local environment* (noted Γ). It is used for the first step of typechecking, and looks like a standard environment: $\Gamma ::= \cdot \mid \Gamma, x{:}\sigma \mid \Gamma, x := \Delta : \sigma$. Intuitively, $x{:}\sigma$ is a variable introduced by a λ-abstraction, and $x := \Delta : \sigma$ is a local definition introduced by a **let**.
3. the *essence environment* (noted Ψ). It is used for the second step of typechecking, and holds the essence of the local variables: $\Psi ::= \cdot \mid \Psi, x \mid \Psi, x := M$. Intuitively, x is a variable introduced by a λ-abstraction, and $x := M$ is a local definition introduced by a **let**. Notice that the variable x in the BNF expression Ψ, x carries almost no information. However, since local variables are referred to by their de Bruijn indices, and these indices are actually their position in the environment, it follows that they have to appear in the environment, even when there is no additional information.
4. the *meta-environment* (noted Φ). It is used for unification, and records meta-variables and their instantiation whenever the unification algorithm has found a solution: $\Phi ::= \cdot \mid \Phi, \mathbf{sort}(?x) \mid \Phi, ?x := s \mid \Phi, (\Gamma \vdash ?x : \sigma) \mid \Phi, (\Gamma \vdash ?x := \Delta : \sigma) \mid \Phi, \Psi \vdash ?x \mid \Phi, \Psi \vdash ?x := M$. Intuitively, since there are some meta-variables for which we know they have to be sorts, it follows that $\mathbf{sort}(?x)$ declares a meta-variable $?x$ which correspond either to Type or Kind, and $?x := s$ is the instantiation of a sort $?x$. Also, $\Gamma \vdash ?x : \sigma$ is the declaration of a meta-variable $?x$ of type σ which appeared in a local environment Γ, and

$\Gamma \vdash ?x := \Delta : \sigma$ is the instantiation of the meta-variable $?x$. Concerning meta-variables inside essences, $\Psi \vdash ?x$ is the declaration of a meta-variable $?x$ in an essence environment Ψ, and $\Psi \vdash ?x := M$ is the instantiation of $?x$.

2.4 Suspended substitution

We shortly introduce suspended substitution, as presented in [1]. Let's consider the following example: if we want to unify $(\lambda x{:}\sigma.?y)\, c_1$ with c_1, we could unify $?y$ with c_1 or with x, the latter being the preferred solution. However, if we normalize $(\lambda x{:}\sigma.?y)\, c_1$, we should record the fact that c_1 can be substituted by any occurrence of x appearing in $?y$, even though the term which will replace $?y$ is currently unknown. That is the purpose of suspended substitution: the term is actually noted $(\lambda x{:}\sigma.?y[x])\, c_1$ and reduces to $?y[c_1]$, noting that c_1 has replaced x.

▶ **Definition 1** (Erase function and suspended substitution).

1. *The vector* $x_1; \ldots; x_n$ *is created using the erase function* $\bar{\cdot}$, *defined as* $\overline{x_1{:}\sigma_1; \ldots; x_n{:}\sigma_n} \overset{def}{=} x_1; \ldots; x_n$ *and* $\overline{x_1; \ldots; x_n} \overset{def}{=} x_1; \ldots; x_n$.
2. *When we want to create a new meta-variable in a local context* $\Gamma = x_1{:}\sigma_1; \ldots; x_n{:}\sigma_n$, *we create a meta-variable* $?y[\overline{\Gamma}] \equiv ?y[x_1; \ldots; x_n]$. *The vector* $\Delta_1; \ldots; \Delta_n$ *inside* $?y[\Delta_1; \ldots; \Delta_n]$ *is the suspended substitution of* $?y$. *Substitutions for meta-variables and their suspended substitution are propagated as follows:*

$$\begin{array}{rcl} ?y[\Delta_1; \ldots; \Delta_n][\Delta/x] & \overset{def}{=} & ?y[\Delta_1[\Delta/x]; \ldots; \Delta_n[\Delta/x]] \\ ?y[M_1; \ldots; M_n][N/x] & \overset{def}{=} & ?y[M_1[N/x]; \ldots; M_n[N/x]] \end{array}$$

3 The evaluator of Bull

The evaluator follows the applicative order strategy, which recursively normalizes all subterms from left to right (with the help of the `visit_term` function, see full code in [44]), then: if the resulting term is a redex, reduces it, then uses the same strategy again; or else, the resulting term is in normal form.

3.1 Reduction rules

The notions of reduction, from which we can define one-step reduction, multistep reduction, and equivalence relation, are defined below.

▶ **Definition 2** (Reductions).

1. *for* Δ*-terms:*

$$\begin{array}{rcll} (\lambda x{:}\sigma.\Delta_1)\, \Delta_2 & \mapsto_{\beta} & \Delta_1[\Delta_2/x] & \\ \lambda x{:}\sigma.\Delta\, x & \mapsto_{\eta} & \Delta & \textit{if } x \notin \mathsf{Fv}(\Delta) \\ \mathit{pr}_i\, \langle \Delta_1, \Delta_2 \rangle & \mapsto_{\mathit{pr}_i} & \Delta_i & \\ \multicolumn{4}{l}{\textbf{smatch } \mathit{in}_i\, \Delta_3 \textbf{ return } \rho \textbf{ with} [x{:}\sigma \Rightarrow \Delta_1 \mid x{:}\tau \Rightarrow \Delta_2]} \\ & \mapsto_{\mathit{in}_i} & \Delta_i[\Delta_3/x] & \\ \textbf{let } x{:}\sigma := \Delta_1 \textbf{ in } \Delta_2 & \mapsto_{\zeta} & \Delta_2[\Delta_1/x] & \\ c & \mapsto_{\delta\Sigma} & \Delta & \textit{if } (c := M @\Delta : \varsigma @\sigma) \in \Sigma \\ x & \mapsto_{\delta\Gamma} & \Delta & \textit{if } (x := \Delta : \sigma) \in \Gamma \\ ?x[\Delta_1; \ldots; \Delta_n] & \mapsto_{\delta\Phi} & \Delta\overrightarrow{[\Delta_i/\overline{\Gamma}]} & \textit{if } (\Gamma \vdash ?x := \Delta : \sigma) \in \Phi \\ ?x[\Delta_1; \ldots; \Delta_n] & \mapsto_{\delta\Phi} & s & \textit{if } ?x := s \in \Phi \end{array}$$

2. *for pure λ-terms:*

$$\begin{array}{rcll}
(\lambda x.M)\, N & \mapsto_\beta & M[N/x] & \\
\lambda x.M\, x & \mapsto_\eta & M & \textit{if } x \notin \mathsf{Fv}(M) \\
\mathbf{let}\ x := M\ \mathbf{in}\ N & \mapsto_\zeta & N[M/x] & \\
c & \mapsto_{\delta\Sigma} & M & \textit{if } (c := M @\Delta : \varsigma @\sigma) \in \Sigma \\
x & \mapsto_{\delta\Psi} & M & \textit{if } (x := M) \in \Psi \\
?x[M_1; \ldots; M_n] & \mapsto_{\delta\Phi} & N[\overrightarrow{M_i/\Psi}] & \textit{if } (\Psi \vdash ?x := M) \in \Phi \\
?x[M_1; \ldots; M_n] & \mapsto_{\delta\Phi} & s & \textit{if } ?x := s \in \Phi
\end{array}$$

3.2 Implementation

When the user inputs a term, the refiner creates meta-variables and tries to instantiate them, but this should remain as much as possible invisible to the user. Therefore the term returned by the refiner should be meta-variable free, even though not in normal form. Thus, terms in the global signature Σ are meta-variable free, and the $\delta\Phi$ reductions are only used by the unifier and the refiner.

The function `strongly_normalize` works on both Δ-terms and pure λ-terms, and supposes that the given term is meta-variable free. Note that reductions can create odd spines, for instance if you consider the term $(\lambda x{:}\sigma.x\, S_1)\,(\Delta\, S_2)$, a simple β-redex would give $\Delta\, S_2\, S_1$, therefore we merge S_2 and S_1 in a single spine.

```
let rec strongly_normalize is_essence env ctx t =
  let sn_children = visit_term (strongly_normalize is_essence env ctx)
                (fun _ → strongly_normalize is_essence
                         env (Env.add_var ctx (DefAxiom ("",nothing))))
                (fun id _ → id)
  in let sn = strongly_normalize is_essence env ctx in
  (* Normalize the children *)
  let t = sn_children t in
  match t with
  (* Spine fix *)
  | App(l, App(l',t1,t2), t3) →
     sn (App(l, t1, List.append t2 t3))
  (* Beta-redex *)
  | App (l, Abs (l', _,_, t1), t2 ::  [ ]) →
     sn (beta_redex t1 t2)
  | App (l, Abs (l', x,y, t1), t2 ::  t3)
    → sn @@ app l (sn (App(l,Abs (l',x,y, t1), t3))) t2
  | Let (l, _, t1, t2, t3) → sn (beta_redex t2 t1)
  (* Delta-redex *)
  | Var (l, n) → let (t1, _) = Env.find_var ctx n in
        (match t1 with
         | Var _ → t1
         | _ → sn t1)
  | Const (l, id) → let o = Env.find_const is_essence env id in
                (match o with
                 | None → Const(l, id)
                 | Some (Const (_,id') as t1,_) when id = id' → t1
                 | Some (t1,_) → sn t1)
  (* Eta-redex *)
  | Abs (l,_, _, App (l', t1, Var (_,0) ::  l2))
```

```
    → if is_eta (App (l’, t1, l2)) then
         let t1 = lift 0 (−1) t1 in
         match l2 with
         | [ ] → t1
         | _ → App (l’, t1, List.map (lift 0 (−1)) l2)
       else t
  (* Pair-redex *)
  | SPrLeft (l, SPair (l’, x,_)) → x
  | SPrRight (l, SPair (l’, _, x)) → x
  (* inj-reduction *)
  | SMatch (l, SInLeft(l’,_,t1), _, id1, _, t2, id2, _, _) →
     sn (beta_redex t2 t1)
  | SMatch (l, SInRight(l’,_,t1), _, id1, _, _, id2, _, t2) →
     sn (beta_redex t2 t1)
  | _ → t
```

4 The subtyping algorithm of Bull

The subtyping algorithm implemented in Bull is basically the algorithm $\mathcal{A}$ as described and Coq certified/extracted in [27] by the authors. The main judgment is $\Sigma;\Gamma \vdash \sigma \leqslant \tau$. The only difference is that the types are normalized before applying the algorithm. The auxiliary rewriting functions $\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3, \mathcal{R}_4$, described in [27], rewrite terms in normal forms as follows:

```
let rec anf a =
  let rec distr f a b =
    match (a,b) with
    | (Union(l,a1,a2),_) → Inter(l, distr f a1 b, distr f a2 b)
    | (_, Inter(l,b1,b2)) → Inter(l, distr f a b1, distr f a b2)
    | _ → f a b
  in
  match a with
  | Prod(l,id,a,b) → distr (fun a b → Prod(l,id,a,b)) (danf a) (canf b)
  | _ → a
and canf a =
  let rec distr a b =
    match (a,b) with
    | (Inter(l,a1,a2),_) → Inter(l, distr a1 b, distr a2 b)
    | (_,Inter(l,b1,b2)) → Inter(l, distr a b1, distr a b2)
    | _ → Union(dummy_loc,a,b)
  in
  match a with
  | Inter(l,a,b) → Inter(l, canf a, canf b)
  | Union(l,a,b) → distr (canf a) (canf b)
  | _ → anf a
and danf a =
  let rec distr a b =
    match (a,b) with
    | (Union(l,a1,a2),_) → Union(l, distr a1 b, distr a2 b)
    | (_,Union(l,b1,b2)) → Union(l, distr a b1, distr a b2)
    | _ → Inter(dummy_loc, a,b)
  in
  match a with
  | Inter(l,a,b) → distr (danf a) (danf b)
  | Union(l,a,b) → Union(l, danf a, danf b)
  | _ → anf a
```

It follows that, our subtyping function is quite simple:

```
let is_subtype env ctx a b =
  let a = danf @@ strongly_normalize false env ctx a in
  let b = canf @@ strongly_normalize false env ctx b in
  let rec foo env ctx a b =
  match (a, b) with
  | (Union(_,a1,a2),_) → foo env ctx a1 b && foo env ctx a2 b
  | (_,Inter(_,b1,b2)) → foo env ctx a b1 && foo env ctx a b2
  | (Inter(_,a1,a2),_) → foo env ctx a1 b || foo env ctx a2 b
  | (_,Union(_,b1,b2)) → foo env ctx a b1 || foo env ctx a b2
  | (Prod(_,_,a1,a2),Prod(_,_,b1,b2))
    → foo env ctx b1 a1 && foo env (Env.add_var ctx (DefAxiom("",nothing))) a2 b2
  | _ → same_term a b
  in foo env ctx a b
```

5 The unification algorithm of Bull

Higher-order unification of two terms Δ_1 and Δ_2 aims at finding a most general substitution for meta-variables such that Δ_1 and Δ_2 becomes convertible. The structural rules are given in Figure 1. Classical references are the work of Huet [26], and Dowek et al. [16].

Our higher-order unification algorithm is inspired by the Reed [41] and Ziliani-Sozeau [48] papers. In [48], conversion of terms is quite involved because of the complexity of Coq. For simplicity, our algorithm supposes the terms to be in normal form.

The unification algorithm takes as input a meta-environment Φ, a global environment Σ, a local environment Γ, the two terms to unify Δ_1 and Δ_2, and either fails or returns the updated meta-environment Φ. The rest of the unification algorithm implements *Higher-Order Pattern Unification* (HOPU) [41]. In a nutshell, HOPU takes as an argument a unification problem $?f\,S \overset{?}{=} N$, where all the terms in S are free variables and each variable occurs once. For instance, for the unification problem $?f\,y\,x\,z \overset{?}{=} x\,c\,y$, it creates the solution $?f := \lambda y{:}\sigma_2.\lambda x{:}\sigma_1.\lambda z{:}\sigma_3.x\,c\,y$. The expected type of x, y, and z can be found in the local environment, but capturing correctly the free variables x, y, and z is quite tricky because we have to permute their de Bruijn indices. If HOPU fails, we recursively unify every subterm.

6 The refinement algorithm of Bull

The Bull refinement algorithm is inspired by the work on the Matita ITP [1]. It is defined using *bi-directionality*, in the style of Harper-Licata [22]. The bi-directional technique is a mix of typechecking and type reconstruction, in order to trigger the unification algorithm as soon as possible. Moreover, it gives more precise error messages than standard type reconstruction. For instance, if `f : (bool -> nat -> bool) -> bool`, then `f (fun x y ⇒ y)` is ill-typed. With a simple type inference algorithm, we would type `f`, then `fun x y ⇒ y` which would be given some type `?x -> ?y -> ?y`, and finally we would try to unify `bool -> nat -> bool` with `?x -> ?y -> ?y`, which fails. However, the failure is localized on the application, whereas it would better be localized inside the argument. More precisely, we would have the following error message:

$$\frac{s_1 \equiv s_2}{\Phi; \Sigma; \Gamma \vdash s_1 \overset{?}{=} s_2 \overset{\mathcal{U}}{\leadsto} \Phi}\ (\mathit{Sort}) \qquad \frac{c_1 \equiv c_2}{\Phi; \Sigma; \Gamma \vdash c_1 \overset{?}{=} c_2 \overset{\mathcal{U}}{\leadsto} \Phi}\ (\mathit{Const}) \qquad \frac{x_1 \equiv x_2}{\Phi; \Sigma; \Gamma \vdash x_1 \overset{?}{=} x_2 \overset{\mathcal{U}}{\leadsto} \Phi}\ (\mathit{Var})$$

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \sigma_1 \overset{?}{=} \sigma_2 \overset{\mathcal{U}}{\leadsto} \Phi_2 \quad \Phi_2; \Sigma; \Gamma, x{:}\sigma_1 \vdash \Delta_1 \overset{?}{=} \Delta_2 \overset{\mathcal{U}}{\leadsto} \Phi_3}{\Phi_1; \Sigma; \Gamma \vdash \lambda x{:}\sigma_1.\Delta_1 \overset{?}{=} \lambda x{:}\sigma_2.\Delta_2 \overset{\mathcal{U}}{\leadsto} \Phi_3}\ (\mathit{Abs})$$

$$\frac{\begin{array}{l} x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n \vdash ?x : \Pi z_1{:}\tau_1 \ldots \Pi z_m{:}\tau_m.\tau \in \Phi_1 \qquad y_1 \ldots y_n, z_1 \ldots z_m \text{ distinct} \\ \Phi_2 \equiv \Phi_1, (x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n \vdash ?x := \lambda z_1{:}\tau_1. \ldots \lambda z_m{:}\tau_m.\Delta\overrightarrow{[x_i/y_i]} : \Pi z_1{:}\tau_1 \ldots \Pi z_m{:}\tau_m.\tau) \end{array}}{\Phi_1; \Sigma; \Gamma \vdash ?x[y_1; \ldots; y_n]\,(z_1; \ldots; z_m) \overset{?}{=} \Delta \overset{\mathcal{U}}{\leadsto} \Phi_2}\ (\mathit{App}_1)$$

$$\frac{\Phi_1 \vdash \Delta_1 \overset{?}{=} \Delta_2 \overset{\mathcal{U}}{\leadsto} \Phi_2}{\Phi_1; \Sigma; \Gamma \vdash \Delta_1\,() \overset{?}{=} \Delta_2 \overset{\mathcal{U}}{\leadsto} \Phi_2}\ (\mathit{App}_2) \quad \frac{\Phi_1 \vdash \Delta_1\,S_1 \overset{?}{=} \Delta_3\,S_2 \overset{\mathcal{U}}{\leadsto} \Phi_2 \quad \Phi_2 \vdash \Delta_2 \overset{?}{=} \Delta_4 \overset{\mathcal{U}}{\leadsto} \Phi_3}{\Phi_1; \Sigma; \Gamma \vdash \Delta_1\,(S_1; \Delta_2) \overset{?}{=} \Delta_3\,(S_2; \Delta_4) \overset{\mathcal{U}}{\leadsto} \Phi_3}\ (\mathit{App}_3)$$

$$\frac{\begin{array}{c} \Phi_1; \Sigma; \Gamma \vdash \sigma_1 \overset{?}{=} \sigma_2 \overset{\mathcal{U}}{\leadsto} \Phi_2 \\ \Phi_2; \Sigma; \Gamma \vdash \tau_1 \overset{?}{=} \tau_2 \overset{\mathcal{U}}{\leadsto} \Phi_3 \end{array}}{\Phi_1; \Sigma; \Gamma \vdash \sigma_1 \cap \tau_1 \overset{?}{=} \sigma_2 \cap \tau_2 \overset{\mathcal{U}}{\leadsto} \Phi_3}\ (\cap) \qquad \frac{\begin{array}{c} \Phi_1; \Sigma; \Gamma \vdash \sigma_1 \overset{?}{=} \sigma_2 \overset{\mathcal{U}}{\leadsto} \Phi_2 \\ \Phi_2; \Sigma; \Gamma \vdash \tau_1 \overset{?}{=} \tau_2 \overset{\mathcal{U}}{\leadsto} \Phi_3 \end{array}}{\Phi_1; \Sigma; \Gamma \vdash \sigma_1 \cup \tau_1 \overset{?}{=} \sigma_2 \cup \tau_2 \overset{\mathcal{U}}{\leadsto} \Phi_3}\ (\cup)$$

$$\frac{\begin{array}{c} \Phi_1; \Sigma; \Gamma \vdash \Delta_1 \overset{?}{=} \Delta_2 \overset{\mathcal{U}}{\leadsto} \Phi_2 \\ \Phi_2; \Sigma; \Gamma \vdash \Delta_3 \overset{?}{=} \Delta_4 \overset{\mathcal{U}}{\leadsto} \Phi_3 \end{array}}{\Phi_1; \Sigma; \Gamma \vdash \langle \Delta_1, \Delta_3 \rangle \overset{?}{=} \langle \Delta_2, \Delta_4 \rangle \overset{\mathcal{U}}{\leadsto} \Phi_3}\ (\mathit{Spair}) \qquad \frac{\Phi_1; \Sigma; \Gamma \vdash \Delta_1 \overset{?}{=} \Delta_2 \overset{\mathcal{U}}{\leadsto} \Phi_2}{\Phi_1; \Sigma; \Gamma \vdash \mathsf{pr}_i\,\Delta_1 \overset{?}{=} \mathsf{pr}_i\,\Delta_2 \overset{\mathcal{U}}{\leadsto} \Phi_2}\ (\mathit{Proj})$$

$$\frac{\begin{array}{c} \Phi_1; \Sigma; \Gamma \vdash \sigma_1 \overset{?}{=} \sigma_2 \overset{\mathcal{U}}{\leadsto} \Phi_2 \\ \Phi_2; \Sigma; \Gamma \vdash \Delta_1 \overset{?}{=} \Delta_2 \overset{\mathcal{U}}{\leadsto} \Phi_3 \end{array}}{\Phi_1; \Sigma; \Gamma \vdash \mathsf{in}_i\,\sigma_1 \Delta_1 \overset{?}{=} \mathsf{in}_i\,\sigma_2 \Delta_2 \overset{\mathcal{U}}{\leadsto} \Phi_3}\ (\mathit{Inj}) \qquad \frac{\begin{array}{c} \Phi_1; \Sigma; \Gamma \vdash \sigma_1 \overset{?}{=} \sigma_2 \overset{\mathcal{U}}{\leadsto} \Phi_2 \\ \Phi_2; \Sigma; \Gamma \vdash \Delta_1 \overset{?}{=} \Delta_2 \overset{\mathcal{U}}{\leadsto} \Phi_3 \end{array}}{\Phi_1; \Sigma; \Gamma \vdash \mathbf{coe}\,\sigma_1\,\Delta_1 \overset{?}{=} \mathbf{coe}\,\sigma_2\,\Delta_2 \overset{\mathcal{U}}{\leadsto} \Phi_3}\ (\mathit{Coe})$$

$$\frac{\begin{array}{lll} \Phi_1; \Sigma; \Gamma \vdash \Delta \overset{?}{=} \Delta' \overset{\mathcal{U}}{\leadsto} \Phi_2 & \Phi_2; \Sigma; \Gamma \vdash \tau \overset{?}{=} \tau' \overset{\mathcal{U}}{\leadsto} \Phi_3 & \Phi_3; \Sigma; \Gamma \vdash \sigma_1 \overset{?}{=} \sigma'_1 \overset{\mathcal{U}}{\leadsto} \Phi_4 \\ \Phi_4; \Sigma; \Gamma, x{:}\sigma_1 \vdash \Delta_1 \overset{?}{=} \Delta'_1 \overset{\mathcal{U}}{\leadsto} \Phi_5 & \Phi_5; \Sigma; \Gamma \vdash \sigma_2 \overset{?}{=} \sigma'_2 \overset{\mathcal{U}}{\leadsto} \Phi_6 & \Phi_6; \Sigma; \Gamma, x{:}\sigma_2 \vdash \Delta_2 \overset{?}{=} \Delta'_2 \overset{\mathcal{U}}{\leadsto} \Phi_7 \end{array}}{\begin{array}{l} \Phi_1; \Sigma; \Gamma \vdash \mathbf{smatch}\ \Delta\ \mathbf{return}\ \tau\ \mathbf{with}\,[x{:}\sigma_1 \Rightarrow \Delta_1 \mid x{:}\sigma_2 \Rightarrow \Delta_2] \\ \overset{?}{=} \mathbf{smatch}\ \Delta'\ \mathbf{return}\ \tau'\ \mathbf{with}\,[x{:}\sigma'_1 \Rightarrow \Delta_1 \mid x{:}\sigma'_2 \Rightarrow \Delta'_2] \overset{\mathcal{U}}{\leadsto} \Phi_7 \end{array}}\ (\mathit{Ssum})$$

$$\frac{\Phi_1; \Sigma; \Gamma, x{:}\sigma \vdash \Delta_1 \overset{?}{=} \Delta_2\,x \overset{\mathcal{U}}{\leadsto} \Phi_2}{\Phi_1; \Sigma; \Gamma \vdash \lambda x{:}\sigma.\Delta_1 \overset{?}{=} \Delta_2 \overset{\mathcal{U}}{\leadsto} \Phi_2}\ (\eta_l) \qquad \frac{\Phi_1; \Sigma; \Gamma, x{:}\sigma \vdash \Delta_1\,x \overset{?}{=} \Delta_2 \overset{\mathcal{U}}{\leadsto} \Phi_2}{\Phi_1; \Sigma; \Gamma \vdash \Delta_1 \overset{?}{=} \lambda x{:}\sigma.\Delta_2 \overset{\mathcal{U}}{\leadsto} \Phi_2}\ (\eta_r)$$

Figure 1 Structural rules of the unification algorithm.

```
f (fun x y ⇒ y)
             ^

Error: the term "y" has type "nat" while it is expected to have type "bool".
```

Our typechecker is also a *refiner*: intuitively, a refiner takes as input an incomplete term, and possibly an incomplete type, and tries to infer as much information as possible in order to reconstruct a well-typed term. For example, let's assume we have in the global environment the following constants:

```
(eq : nat -> nat -> Type), (eq_refl : forall x : nat, eq x x)
```

Then refining the term `eq_refl _ : eq _ 0` would create the following term:

```
eq_refl 0 : eq 0 0
```

$$\frac{(x{:}\sigma) \in \Gamma \text{ or } (x := \Delta : \sigma) \in \Gamma}{\Phi;\Sigma;\Gamma \vdash x \overset{\Uparrow}{\rightsquigarrow} x : \sigma;\Phi}\;(\mathit{Var}) \qquad \frac{(c{:}\sigma) \in \Sigma \text{ or } (c := \Delta : \sigma) \in \Sigma}{\Phi;\Sigma;\Gamma \vdash c \overset{\Uparrow}{\rightsquigarrow} c : \sigma;\Phi}\;(\mathit{Const})$$

$$\frac{\Phi_1;\Sigma;\Gamma \vdash \sigma \overset{\mathcal{F}}{\rightsquigarrow} \sigma' : s;\Phi_2 \quad \Phi_2;\Sigma;\Gamma \vdash \Delta_1 : \sigma' \overset{\Downarrow}{\rightsquigarrow} \Delta_1';\Phi_3 \quad \Phi_3;\Sigma;\Gamma, x := \Delta_1'{:}\sigma' \vdash \Delta_2 \overset{\Uparrow}{\rightsquigarrow} \Delta_2' : \tau;\Phi_4}{\Phi_1;\Sigma;\Gamma \vdash \mathbf{let}\ x{:}\sigma := \Delta_1\ \mathbf{in}\ \Delta_2 \overset{\Uparrow}{\rightsquigarrow} \mathbf{let}\ x{:}\sigma' := \Delta_1'\ \mathbf{in}\ \Delta_2' : \tau[\sigma'/x];\Phi_4}\;(\mathit{Let})$$

$$\frac{\Phi_1;\Sigma;\Gamma \vdash \sigma_1 \overset{\mathcal{F}}{\rightsquigarrow} \sigma_1' : s_1;\Phi_2 \quad \Phi_2;\Sigma;\Gamma \vdash \sigma_2 \overset{\mathcal{F}}{\rightsquigarrow} \sigma_2' : s_2;\Phi_3 \quad \Phi_3 \vdash (s_1,s_2) \in \mathbf{LF};\Phi_4}{\Phi_1;\Sigma;\Gamma \vdash \Pi x{:}\sigma_1.\sigma_2 \overset{\Uparrow}{\rightsquigarrow} \Pi x{:}\sigma_1'.\sigma_2' : s_2;\Phi_4}\;(\mathit{Prod})$$

$$\frac{\Phi_1;\Sigma;\Gamma \vdash \sigma \overset{\mathcal{F}}{\rightsquigarrow} \sigma' : s;\Phi_2 \quad \Phi_2;\Sigma;\Gamma, x{:}\sigma' \vdash \Delta \overset{\Uparrow}{\rightsquigarrow} \Delta' : \tau;\Phi_3 \quad \Phi_3;\Sigma;\Gamma \vdash \Pi x{:}\sigma'.\tau \overset{\mathcal{F}}{\rightsquigarrow} \rho : s;\Phi_4}{\Phi_1;\Sigma;\Gamma \vdash \lambda x{:}\sigma.\Delta \overset{\Uparrow}{\rightsquigarrow} \lambda x{:}\sigma'.\Delta' : \Pi x{:}\sigma'.\tau;\Phi_4}\;(\mathit{Abs})$$

$$\frac{\Phi_1;\Sigma;\Gamma \vdash \Delta \overset{\Uparrow}{\rightsquigarrow} \Delta' : \sigma;\Phi_2}{\Phi_1;\Sigma;\Gamma \vdash \Delta\,() \overset{\Uparrow}{\rightsquigarrow} \Delta' : \sigma;\Phi_2}\;(\mathit{App}_1) \qquad \frac{}{\Phi;\Sigma;\Gamma \vdash \mathsf{Type} \overset{\Uparrow}{\rightsquigarrow} \mathsf{Type} : \mathsf{Kind};\Phi}\;(T)$$

$$\frac{\Phi_1;\Sigma;\Gamma \vdash \Delta_1\,S \overset{\Uparrow}{\rightsquigarrow} \Delta_1' : \sigma;\Phi_2 \quad \Phi_2;\Sigma;\Gamma \vdash \sigma =_\beta \Pi x{:}\sigma_1.\sigma_2 \quad \Phi_2;\Sigma;\Gamma \vdash \Delta_2 : \sigma_1 \overset{\Downarrow}{\rightsquigarrow} \Delta_2';\Phi_3}{\Phi_1;\Sigma;\Gamma \vdash \Delta_1\,(S;\Delta_2) \overset{\Uparrow}{\rightsquigarrow} \Delta_1'\,\Delta_2' : \sigma_2[\Delta_2'/x];\Phi_3}\;(\mathit{App}_2)$$

$$\frac{\begin{array}{c}\Phi_1;\Sigma;\Gamma \vdash \Delta_1\,S \overset{\Uparrow}{\rightsquigarrow} \Delta_1' : \sigma;\Phi_2 \quad \Phi_2;\Sigma;\Gamma \vdash \Delta_2 \overset{\Uparrow}{\rightsquigarrow} \Delta_2' : \sigma_1;\Phi_3 \\ \Phi_3, \mathbf{sort}(?y), (\Gamma, x{:}\sigma_1 \vdash ?x :?y[\,]);\Sigma;\Gamma \vdash \sigma \overset{?}{=} \Pi x{:}\sigma_1.?x[\overline{\Gamma};x] \overset{\mathcal{U}}{\rightsquigarrow} \Phi_4\end{array}}{\Phi_1;\Sigma;\Gamma \vdash \Delta_1\,(S;\Delta_2) \overset{\Uparrow}{\rightsquigarrow} \Delta_1'\,\Delta_2' : ?x[\overline{\Gamma};x][\Delta_2'/x];\Phi_4}\;(\mathit{App}_3)$$

$$\frac{\begin{array}{c}\Phi_1;\Sigma;\Gamma \vdash \sigma_1 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \sigma_1';\Phi_2 \\ \Phi_2;\Sigma;\Gamma \vdash \sigma_2 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \sigma_2';\Phi_3\end{array}}{\Phi_1;\Sigma;\Gamma \vdash \sigma_1 \cap \sigma_2 \overset{\Uparrow}{\rightsquigarrow} \sigma_1' \cap \sigma_2' : \mathsf{Type};\Phi_3}\;(\cap) \qquad \frac{\begin{array}{c}\Phi_1;\Sigma;\Gamma \vdash \sigma_1 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \sigma_1';\Phi_2 \\ \Phi_2;\Sigma;\Gamma \vdash \sigma_2 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \sigma_2';\Phi_3\end{array}}{\Phi_1;\Sigma;\Gamma \vdash \sigma_1 \cup \sigma_2 \overset{\Uparrow}{\rightsquigarrow} \sigma_1' \cup \sigma_2' : \mathsf{Type};\Phi_3}\;(\cup)$$

Figure 2 Rules for $\overset{\Uparrow}{\rightsquigarrow}$ (1st part).

Refinement also enables untyped abstractions: the refiner may recover the type of bound variables, because untyped abstractions are incomplete terms. The typechecking is done in two steps: firstly the term is typechecked without caring about the essence, then we check the essence. The five typing judgments are defined as follows:

▶ **Definition 3** (Typing judgments). *We have five typing judgments, corresponding to five OCaml functions:*

1. *The function `reconstruct` takes as inputs a meta-environment Φ_1, a global environment Σ, a local environment Γ, and a term Δ_1. It either fails or fills the holes in Δ_1, which becomes Δ_2, and returns Δ_2 along with its type σ and the updated meta-environment Φ_2. The corresponding judgment is the following $\Phi_1;\Sigma;\Gamma \vdash \Delta_1 \overset{\Uparrow}{\rightsquigarrow} \Delta_2 : \sigma;\Phi_2$, and the most relevant rules are described in Figures 2 and 3;*
2. *The function `force_type` takes as inputs a meta-environment Φ_1, a global environment Σ, a local environment Γ, and a term σ_1. It either fails or fills the holes in σ_1, which becomes σ_2 while ensuring it is a type, i.e. its type is a sort s, and returns σ_2 along with s, and the updated meta-environment Φ_2. The corresponding judgment is the following $\Phi_1;\Sigma;\Gamma \vdash \sigma_1 \overset{\mathcal{F}}{\rightsquigarrow} \sigma_2 : \tau;\Phi_2$, and the rules are described in Figure 4. Intuitively, the function reconstructs the type τ of σ_1, then tries to unify τ with* Type *and* Kind*. If it can only do one unification, it keeps the successful one, if both unifications work, we choose unification with a sort meta-variable, so τ is convertible to a sort;*
3. *The function `reconstruct_with_type` takes as inputs a meta-environment Φ_1, a global environment Σ, a local environment Γ, a term Δ_1, and its expected type σ. It either fails or fills the holes in Δ_1, which becomes Δ_2 while ensuring its type is σ, and returns Δ_2 along the updated meta-environment Φ_2. The corresponding judgment is the following*

$$\dfrac{\begin{array}{l}\Phi_1;\Sigma;\Gamma\vdash\Delta_1 \overset{\Uparrow}{\rightsquigarrow} \Delta_1' : \sigma_1;\Phi_2\\ \Phi_2;\Sigma;\Gamma\vdash\Delta_2 \overset{\Uparrow}{\rightsquigarrow} \Delta_2' : \sigma_2;\Phi_3 \quad \Phi_3;\Sigma;\Gamma\vdash\sigma_1\cap\sigma_2 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \Phi_4\end{array}}{\Phi_1;\Sigma;\Gamma\vdash\langle\Delta_1,\Delta_2\rangle \overset{\Uparrow}{\rightsquigarrow} \langle\Delta_1',\Delta_2'\rangle : \sigma_1\cap\sigma_2;\Phi_4}\ (\mathit{Spair})$$

$$\dfrac{\Phi_1;\Sigma;\Gamma\vdash\Delta \overset{\Uparrow}{\rightsquigarrow} \Delta' : \sigma;\Phi_2 \quad \Phi_2;\Sigma;\Gamma\vdash\sigma =_\beta \sigma_1\cap\sigma_2}{\Phi_1;\Sigma;\Gamma\vdash \mathsf{pr}_i\,\Delta \overset{\Uparrow}{\rightsquigarrow} \mathsf{pr}_i\,\Delta' : \sigma_i;\Phi_2}\ (\mathit{Proj}_1)$$

$$\dfrac{\begin{array}{l}\Phi_1;\Sigma;\Gamma\vdash\Delta \overset{\Uparrow}{\rightsquigarrow} \Delta' : \sigma;\Phi_2\\ \Phi_2,(\Gamma\vdash ?x_1 : \mathsf{Type}),(\Gamma\vdash ?x_2 : \mathsf{Type});\Sigma;\Gamma\vdash\sigma \overset{?}{=} ?x_1[\overline{\Gamma}]\cap ?x_2[\overline{\Gamma}] \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3\end{array}}{\Phi_1;\Sigma;\Gamma\vdash \mathsf{pr}_i\,\Delta \overset{\Uparrow}{\rightsquigarrow} \mathsf{pr}_i\,\Delta' : ?x_i[\overline{\Gamma}];\Phi_3}\ (\mathit{Proj}_2)$$

$$\dfrac{\begin{array}{ll}\Phi_1;\Sigma;\Gamma\vdash\Delta \overset{\Uparrow}{\rightsquigarrow} \Delta' : \sigma';\Phi_2 & \Phi_2;\Sigma;\Gamma\vdash\lambda x{:}\tau_1.\tau_2 : \Pi x{:}\sigma.\mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \lambda x{:}\tau_1'.\tau_2';\Phi_3\\ \Phi_3;\Sigma;\Gamma\vdash\sigma_1 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \sigma_1';\Phi_4 & \Phi_4;\Sigma;\Gamma\vdash\sigma_2 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \sigma_2';\Phi_5\\ \Phi_5;\Sigma;\Gamma\vdash\sigma' \overset{?}{=} \sigma_1'\cup\sigma_2' \overset{\mathcal{U}}{\rightsquigarrow} \Phi_6 & \Phi_6;\Sigma;\Gamma,x{:}\sigma_1'\vdash\Delta_1 : \tau_2'[\mathsf{in}_1\,\sigma_2'\,x/x] \overset{\Downarrow}{\rightsquigarrow} \Delta_1';\Phi_7\\ \Phi_7;\Sigma;\Gamma,x{:}\sigma_2'\vdash\Delta_2 : \tau_2'[\mathsf{in}_2\,\sigma_1'\,x/x] \overset{\Downarrow}{\rightsquigarrow} \Delta_2';\Phi_8 & \end{array}}{\begin{array}{l}\Phi_1;\Sigma;\Gamma\vdash \textbf{smatch}\ \Delta\ \textbf{return}\ \lambda x{:}\tau_1.\tau_2\ \textbf{with}\,[x{:}\sigma_1\Rightarrow\Delta_1 \mid x{:}\sigma_2\Rightarrow\Delta_2] \overset{\Uparrow}{\rightsquigarrow}\\ \quad \textbf{smatch}\ \Delta'\ \textbf{return}\ \lambda x{:}\tau_1'.\tau_2'\ \textbf{with}\,[x{:}\sigma_1'\Rightarrow\Delta_1' \mid x{:}\sigma_2'\Rightarrow\Delta_2'] : \tau_2'[\Delta'/x];\Phi_8\end{array}}\ (\mathit{Ssum})$$

$$\dfrac{\Phi_1;\Sigma;\Gamma\vdash\sigma \overset{\mathcal{F}}{\rightsquigarrow} \sigma' : s;\Phi_2 \quad \Phi_2;\Sigma;\Gamma\vdash\Delta \overset{\Uparrow}{\rightsquigarrow} \Delta' : \tau;\Phi_3 \quad \Sigma;\Gamma\vdash\tau\leqslant\sigma'}{\Phi_1;\Sigma;\Gamma\vdash \textbf{coe}\,\sigma\,\Delta \overset{\Uparrow}{\rightsquigarrow} \textbf{coe}\,\sigma'\,\Delta' : \sigma';\Phi_3}\ (\mathit{Coe})$$

$$\dfrac{}{\Phi;\Sigma;\Gamma\vdash \_ \overset{\Uparrow}{\rightsquigarrow} ?x[\overline{\Gamma}] : ?y[\overline{\Gamma}];\Phi,\textbf{sort}(?z),(\Gamma\vdash ?y : ?z[\,]),(\Gamma\vdash ?x : ?y[\overline{\Gamma}])}\ (\mathit{Wildcard})$$

$$\dfrac{\begin{array}{l}(\Gamma'\vdash ?x : \sigma)\in\Phi \text{ or } (\Gamma'\vdash ?x := \Delta : \sigma)\in\Phi\\ \Gamma' = x_1{:}\sigma_1,\ldots,x_n{:}\sigma_n \quad \Phi_i;\Sigma;\Gamma\vdash\Delta_i : \sigma_i \overset{\Downarrow}{\rightsquigarrow} \Delta_i';\Phi_{i+1} \quad (i=1\ldots n)\end{array}}{\Phi_1;\Sigma;\Gamma\vdash ?x[\Delta_1;\ldots;\Delta_n] \overset{\Uparrow}{\rightsquigarrow} ?x[\Delta_1';\ldots;\Delta_n'] : \sigma[\overrightarrow{\Delta_i'/\overline{\Gamma'}}];\Phi_{n+1}}\ (\mathit{Meta\!-\!Var})$$

Figure 3 Rules for $\overset{\Uparrow}{\rightsquigarrow}$ (2nd part).

$\Phi_1;\Sigma;\Gamma\vdash\Delta_1 : \sigma \overset{\Downarrow}{\rightsquigarrow} \Delta_2;\Phi_2$, *and the rules are described in Figure 5. There is a rule* (*Default*) *which applies only if none of the other rules work. The acute reader could remark two subtle things:*

a. *we chose not to add any inference rule for coercions, because we believe it would make error messages clearer: more precisely, if we want to check that* $\textbf{coe}\,\sigma\,\Delta$ *has type* τ*, there could be two errors happening concurrently: it is possible that the type of* Δ *is not a subtype of* σ*, and at the same time* σ *is not unifiable with* τ*. We think that the error to be reported should be the first one, and in this case the* (*Default*) *rule is sufficient;*

b. *the management of de Bruijn indices for the* (*Let*) *is tricky: if we want to check that* $\textbf{let}\ x{:}\sigma := \Delta_1\ \textbf{in}\ \Delta_2$ *has type* τ *in some local context* Γ*, we recursively check that* Δ_2 *has type* τ *in the local context* $\Gamma, x := \Delta_1' : \sigma'$ *for some* Δ_1'*, but the de Bruijn indices for* τ *correspond to the position of the local variables in the local context, which has been updated. We therefore have to increment all the de Bruijn indices in* τ*, in order to report the fact that there is one extra element in the local context;*

4. *The function* `essence` *takes as inputs a meta-environment* Φ_1*, a global environment* Σ*, an essence environment* Ψ*, and a term* Δ*. It either fails or construct its essence* M*, and returns* M *along with the updated meta-environment* Φ_2*. The corresponding judgment is the following* $\Phi_1;\Sigma;\Psi\vdash\Delta \overset{\mathcal{E}\Uparrow}{\rightsquigarrow} M;\Phi_2$*, and the rules are described in Figure 6;*

$$\frac{\begin{array}{c}\Phi_1; \Sigma; \Gamma \vdash \sigma \overset{\Uparrow}{\rightsquigarrow} \sigma' : \tau; \Phi_2 \quad \Phi_2; \Sigma; \Gamma \vdash \tau \overset{?}{=} \mathsf{Type} \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3 \\ \Phi_2; \Sigma; \Gamma \vdash \tau \overset{?}{=} \mathsf{Kind} \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3' \quad \Phi_2, \mathbf{sort}(?x); \Sigma \vdash \tau \overset{?}{=} ?x[\,] \overset{\mathcal{U}}{\rightsquigarrow} \Phi_4\end{array}}{\Phi_1; \Sigma; \Gamma \vdash \sigma \overset{\mathcal{F}}{\rightsquigarrow} \sigma' : \tau; \Phi_4}\ (\mathit{Force}_1)$$

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \sigma \overset{\Uparrow}{\rightsquigarrow} \sigma' : \tau; \Phi_2 \quad \Phi_2; \Sigma; \Gamma \vdash \tau \overset{?}{=} \mathsf{Type} \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3 \quad \Phi_2; \Sigma; \Gamma \nvdash \tau \overset{?}{=} \mathsf{Kind} \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3'}{\Phi_1; \Sigma; \Gamma \vdash \sigma \overset{\mathcal{F}}{\rightsquigarrow} \sigma' : \tau; \Phi_3}\ (\mathit{Force}_2)$$

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \sigma \overset{\Uparrow}{\rightsquigarrow} \sigma' : \tau; \Phi_2 \quad \Phi_2; \Sigma; \Gamma \nvdash \tau \overset{?}{=} \mathsf{Type} \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3 \quad \Phi_2; \Sigma; \Gamma \vdash \tau \overset{?}{=} \mathsf{Kind} \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3'}{\Phi_1; \Sigma; \Gamma \vdash \sigma \overset{\mathcal{F}}{\rightsquigarrow} \sigma' : \tau; \Phi_3'}\ (\mathit{Force}_3)$$

Figure 4 Rules for $\overset{\mathcal{F}}{\rightsquigarrow}$.

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \Delta \overset{\Uparrow}{\rightsquigarrow} \Delta' : \sigma; \Phi_2 \quad \Phi_2; \Sigma; \Gamma \vdash \sigma \overset{?}{=} \tau \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3}{\Phi_1; \Sigma; \Gamma \vdash \Delta : \tau \overset{\Downarrow}{\rightsquigarrow} \Delta'; \Phi_3}\ (\mathit{Default})$$

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \sigma \overset{\mathcal{F}}{\rightsquigarrow} \sigma' : s; \Phi_2 \quad \Phi_2; \Sigma; \Gamma \vdash \Delta_1 : \sigma' \overset{\Downarrow}{\rightsquigarrow} \Delta_1'; \Phi_3 \quad \Phi_3; \Sigma; \Gamma, x := \Delta_1' : \sigma' \vdash \Delta_2 : \tau \overset{\Downarrow}{\rightsquigarrow} \Delta_2'; \Phi_4}{\Phi_1; \Sigma; \Gamma \vdash \mathbf{let}\ x{:}\sigma := \Delta_1\ \mathbf{in}\ \Delta_2 : \tau \overset{\Downarrow}{\rightsquigarrow} \mathbf{let}\ x{:}\sigma := \Delta_1'\ \mathbf{in}\ \Delta_2'; \Phi_4}\ (\mathit{Let})$$

$$\frac{\begin{array}{c}\Phi_1; \Sigma; \Gamma \vdash \tau =_\beta \Pi x{:}\tau_1.\tau_2 \quad \Phi_1; \Sigma; \Gamma \vdash \sigma \overset{\mathcal{F}}{\rightsquigarrow} \sigma'; \Phi_2 \\ \Phi_2; \Sigma; \Gamma \vdash \sigma' \overset{?}{=} \tau_1 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3 \quad \Phi_3; \Sigma; \Gamma, x{:}\sigma' \vdash \Delta : \tau_2 \overset{\Downarrow}{\rightsquigarrow} \Delta'; \Phi_4\end{array}}{\Phi_1; \Sigma; \Gamma \vdash \lambda x{:}\sigma.\Delta : \tau \overset{\Downarrow}{\rightsquigarrow} \lambda x{:}\sigma'.\Delta'; \Phi_4}\ (\mathit{Abs})$$

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \sigma =_\beta \sigma_1 \cap \sigma_2 \quad \Phi_1; \Sigma; \Gamma \vdash \Delta_1 : \sigma_1 \overset{\Downarrow}{\rightsquigarrow} \Delta_1'; \Phi_2 \quad \Phi_2; \Sigma; \Gamma \vdash \Delta_2 : \sigma_2 \overset{\Downarrow}{\rightsquigarrow} \Delta_2'; \Phi_3}{\Phi_1; \Sigma; \Gamma \vdash \langle \Delta_1, \Delta_2 \rangle : \sigma \overset{\Downarrow}{\rightsquigarrow} \langle \Delta_1', \Delta_2' \rangle; \Phi_3}\ (\mathit{Spair})$$

$$\frac{\Phi_1, (\Gamma \vdash ?x : \mathsf{Type}); \Sigma; \Gamma \vdash \sigma \cap ?x : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \tau; \Phi_2 \quad \Phi_2; \Sigma; \Gamma \vdash \Delta : \sigma \cap ?x \overset{\Downarrow}{\rightsquigarrow} \Delta'; \Phi_3}{\Phi_1; \Sigma; \Gamma \vdash \mathsf{pr}_1\, \Delta : \sigma \overset{\Downarrow}{\rightsquigarrow} \mathsf{pr}_1\, \Delta'; \Phi_3}\ (\mathit{Proj}_1)$$

$$\frac{\Phi_1, (\Gamma \vdash ?x : \mathsf{Type}); \Sigma; \Gamma \vdash ?x \cap \sigma : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \tau; \Phi_2 \quad \Phi_2; \Sigma; \Gamma \vdash \Delta : ?x \cap \sigma \overset{\Downarrow}{\rightsquigarrow} \Delta'; \Phi_3}{\Phi_1; \Sigma; \Gamma \vdash \mathsf{pr}_2\, \Delta : \sigma \overset{\Downarrow}{\rightsquigarrow} \mathsf{pr}_2\, \Delta'; \Phi_3}\ (\mathit{Proj}_2)$$

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \tau =_\beta \tau_1 \cup \tau_2 \quad \Phi_1; \Sigma; \Gamma \vdash \sigma : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \sigma'; \Phi_2 \quad \Phi_2; \Sigma; \Gamma \vdash \sigma' \overset{?}{=} \tau_i \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3}{\Phi_1; \Sigma; \Gamma \vdash \mathsf{in}_i\, \sigma \Delta : \tau \overset{\Downarrow}{\rightsquigarrow} \mathsf{in}_i\, \sigma' \Delta'; \Phi_3}\ (\mathit{Inj})$$

$$\frac{}{\Phi; \Sigma; \Gamma \vdash \_ : \sigma \overset{\Downarrow}{\rightsquigarrow} ?x[\overline{\Gamma}]; \Phi, (\Gamma \vdash ?x : \sigma)}\ (\mathit{Wildcard})$$

Figure 5 Rules for $\overset{\Downarrow}{\rightsquigarrow}$.

5. *The function* `essence_with_hint` *takes as inputs a meta-environment Φ_1, a global environment Σ, an essence environment Ψ, a term Δ, and its expected essence M. It either fails or succeeds by returning the updated meta-environment Φ_2. The corresponding judgment is the following $\Phi_1; \Sigma; \Psi \vdash M @ \Delta \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_2$, and the rules are described in Figure 7. There is a rule (Default) which applies only if none of the other rules work.*

$$\frac{\Phi_1;\Sigma;\Psi\vdash\Delta_1\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_2\quad\Phi_2;\Sigma;\Psi\vdash M@\Delta_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3}{\Phi_1;\Sigma;\Psi\vdash\langle\Delta_1,\Delta_2\rangle\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_3}\;(\mathit{Spair})\qquad\frac{\Phi_1;\Sigma;\Psi\vdash\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_2}{\Phi_1;\Sigma;\Psi\vdash\mathsf{pr}_i\,\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_2}\;(\mathit{Proj})$$

$$\frac{\begin{array}{ccc}\Phi_1;\Sigma;\Psi\vdash\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}N;\Phi_2 & \Phi_2;\Sigma;\Psi\vdash\sigma\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma;\Phi_3 & \Phi_3;\Sigma;\Psi\vdash\sigma_1\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma_1;\Phi_4\\ \Phi_4;\Sigma;\Psi,x\vdash\Delta_1\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_5 & \Phi_5;\Sigma;\Psi\vdash\sigma_2\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma_2;\Phi_6 & \Phi_6;\Sigma;\Psi,x\vdash M@\Delta_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_7\end{array}}{\Phi_1;\Sigma;\Psi\vdash\textbf{smatch}\;\Delta\;\textbf{return}\;\sigma\;\textbf{with}\,[x{:}\sigma_1\Rightarrow\Delta_1\mid x{:}\sigma_2\Rightarrow\Delta_2]\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}(\lambda x.M)\,N;\Phi_7}\;(\mathit{Ssum})$$

$$\frac{\Phi_1;\Sigma;\Psi\vdash\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_2}{\Phi_1;\Sigma;\Psi\vdash\mathsf{in}_i\,\sigma\;\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_2}\;(\mathit{Inj})\qquad\frac{\Phi_1;\Sigma;\Psi\vdash\sigma\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma;\Phi_2\quad\Phi_2;\Sigma;\Psi,x\vdash\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_3}{\Phi_1;\Sigma;\Psi\vdash\lambda x{:}\sigma.\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\lambda x.M;\Phi_3}\;(\mathit{Abs})$$

$$\frac{\Phi_1;\Sigma;\Psi\vdash\sigma_1\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma_1;\Phi_2\quad\Phi_2;\Sigma;\Psi,x\vdash\sigma_2\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma_2;\Phi_3}{\Phi_1;\Sigma;\Psi\vdash\Pi x{:}\sigma_1.\sigma_2\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\Pi x{:}\varsigma_1.\varsigma_2;\Phi_3}\;(\mathit{Prod})\qquad\frac{\Phi_1;\Sigma;\Psi\vdash\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_2}{\Phi_1;\Sigma;\Psi\vdash\textbf{coe}\,\sigma\,\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_2}\;(\mathit{Coe})$$

$$\frac{\Phi_1;\Sigma;\Psi\vdash\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_2}{\Phi_1;\Sigma;\Psi\vdash\Delta\,()\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_2}\;(\mathit{App}_1)\qquad\frac{\Phi_1;\Sigma;\Psi\vdash\Delta_1\,S\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_2\quad\Phi_2;\Sigma;\Psi\vdash\Delta_2\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}N;\Phi_3}{\Phi_1;\Sigma;\Psi\vdash\Delta_1\,(S;\Delta_2)\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M\,N;\Phi_3}\;(\mathit{App}_2)$$

$$\frac{\Phi_1;\Sigma;\Psi\vdash\sigma_1\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma_1;\Phi_2\;\;\Phi_2;\Sigma;\Psi\vdash\sigma_2\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma_2;\Phi_3}{\Phi_1;\Sigma;\Psi\vdash\sigma_1\cap\sigma_2\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma_1\cap\varsigma_2;\Phi_3}\,(\cap)\qquad\frac{\Phi_1;\Sigma;\Psi\vdash\sigma_1\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma_1;\Phi_2\;\;\Phi_2;\Sigma;\Psi\vdash\sigma_2\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma_2;\Phi_3}{\Phi_1;\Sigma;\Psi\vdash\sigma_1\cup\sigma_2\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma_1\cup\varsigma_2;\Phi_3}\,(\cup)$$

Figure 6 Rules for $\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}$.

$$\frac{\begin{array}{c}\Phi_1;\Sigma;\Psi\vdash\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M_2;\Phi_2\\ \Phi_2;\Sigma;\Psi\vdash M_1\overset{?}{=}M_2\overset{\mathcal{U}}{\rightsquigarrow}\Phi_3\end{array}}{\Phi_1;\Sigma;\Psi\vdash M_1@\Delta\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3}\;(\mathit{Default})\qquad\frac{\begin{array}{c}\Phi_1;\Sigma;\Psi\vdash M@\Delta_1\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_2\\ \Phi_2;\Sigma;\Psi\vdash M@\Delta_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3\end{array}}{\Phi_1;\Sigma;\Psi\vdash M@\langle\Delta_1,\Delta_2\rangle\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3}\;(\mathit{Spair})$$

$$\frac{\Phi_1;\Sigma;\Psi\vdash M@\Delta\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_2}{\Phi_1;\Sigma;\Psi\vdash M@\mathsf{pr}_i\,\Delta\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_2}\;(\mathit{Proj})\qquad\frac{\Phi_1;\Sigma;\Psi\vdash\sigma\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma;\Phi_2\quad\Phi_2;\Sigma;\Psi\vdash M@\Delta\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow};\Phi_3}{\Phi_1;\Sigma;\Psi\vdash M@\mathsf{in}_i\,\sigma\,\Delta\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3}\;(\mathit{Inj})$$

$$\frac{\Phi_1;\Sigma;\Psi\vdash\sigma\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma;\Phi_2\quad\Phi_2;\Sigma;\Psi\vdash\Delta_1\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M_1;\Phi_3\quad\Phi_3;\Sigma;\Psi,x:=M_1\vdash M@\Delta_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_4}{\Phi_1;\Sigma;\Psi\vdash M@\textbf{let}\;x{:}\sigma:=\Delta_1\;\textbf{in}\;\Delta_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_4}\;(\mathit{Let})$$

$$\frac{\Phi_1;\Sigma;\Psi\vdash\varsigma=_\beta\Pi x{:}\varsigma_1.\varsigma_2\quad\Phi_1;\Sigma;\Psi\vdash\varsigma_1@\sigma_1\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_2\quad\Phi_2;\Sigma;\Psi,x\vdash\varsigma_2@\sigma_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3}{\Phi_1;\Sigma;\Psi\vdash\varsigma@\Pi x{:}\sigma_1.\sigma_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3}\;(\mathit{Prod})$$

$$\frac{\Phi_1;\Sigma;\Psi\vdash M_1=_\beta\lambda x.M_2\quad\Phi_1;\Sigma;\Psi,x\vdash M_2@\Delta\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_2}{\Phi_1;\Sigma;\Psi\vdash M_1@\lambda x{:}\sigma.\Delta\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_2}\;(\mathit{Abs})$$

$$\frac{\Phi_1;\Sigma;\Psi\vdash\varsigma=_\beta\varsigma_1\cap\varsigma_2\quad\Phi_1;\Sigma;\Psi\vdash\varsigma_1@\sigma_1\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_2\quad\Phi_2;\Sigma;\Psi\vdash\varsigma_2@\sigma_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3}{\Phi_1;\Sigma;\Psi\vdash\varsigma@\sigma_1\cap\sigma_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3}\;(\cap)$$

$$\frac{\Phi_1;\Sigma;\Psi\vdash\varsigma=_\beta\varsigma_1\cup\varsigma_2\quad\Phi_1;\Sigma;\Psi\vdash\varsigma_1@\sigma_1\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_2\quad\Phi_2;\Sigma;\Psi\vdash\varsigma_2@\sigma_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3}{\Phi_1;\Sigma;\Psi\vdash\varsigma@\sigma_1\cup\sigma_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3}\;(\cup)$$

Figure 7 Rules for $\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}$.

7 The Read-Eval-Print-Loop of Bull

The *Read-Eval-Print-Loop* (REPL) reads a command which is given by the parser as a list of atomic commands. For instance, if the user writes:

```
Axiom (a b : Type) (f : a -> b).
```

The parser creates the following list of three atomic commands:

1. the command asking `a` to be an axiom of type `Type`;
2. the command asking `b` to be an axiom of type `Type`;
3. the command asking `f` to be an axiom of type `a -> b`.

The REPL tries to process the whole list. If there is a single failure while processing the list of atomic commands, it backtracks so the whole commands fails without changing the environment. These commands are similar to the vernacular Coq commands and are quite intuitive. Here is the list of the REPL commands, along with their description:

```
Help.                                show this list of commands
Load "file".                         for loading a script file
Axiom term : type.                   define a constant or an axiom
Definition name [: type] := term.    define a term
Print name.                          print the definition of name
Printall.                            print all the signature
                                     (axioms and definitions)
Compute name.                        normalize name and print the result
Quit.                                quit
```

8 Future work

The current version of Bull [44] lacks of the following features that we plan to implement in the next future.

1. *Inductive types* are the most important feature to add, in order to have a usable theorem prover. We plan to take inspiration from the works of Paulin-Mohring [35]. This should be reasonably feasible;
2. *Mixing subtyping and unification* is a difficult problem, especially with intersection and union types. The most extensive research which has been done in this domain is the work of Dudenhefner, Martens, and Rehof [17], where the authors study unification modulo subtyping with intersection types (but no union). It would be challenging to find a unification algorithm modulo subtyping for intersection and union types, but ideally it would allow us to do some implicit coercions. Take for example the famous Pierce code exploiting union and intersection types (full details in the Bull [44] distribution and in Appendix A): it would be interesting for the user to use implicit coercions in this way:

   ```
   Axiom (Neg Zero Pos T F : Type) (Test : Pos | Neg).
   Axiom Is_0 : (Neg -> F) & (Zero -> T) & (Pos -> F).
   Definition Is_0_Test : F := smatch Test with
                                  x => coe _ Is_0 x
                                , x => coe _ Is_0 x
                                end.
   ```

   The unification algorithm would then guess that the first wildcard should be replaced with `Pos -> F` and the second one should be replaced with `Neg -> F`, which does not seem feasible if the unification algorithm does not take subtyping into account;

3. *Relevant arrow*, as defined in [25], it could be useful to add more expressivity to our system. Relevant implication allows for a natural introduction of subtyping, in that $A \supset_{\mathsf{r}} B$ morally means $A \leqslant B$. Relevant implication amounts to a notion of "proof-reuse". Combining the remarks in [3, 2], minimal relevant implication, strong intersection and strong union correspond respectively to the implication, conjunction and disjunction operators of Meyer and Routley's Minimal Relevant Logic B^+ [32]. This could lead to some implementation problem, because deciding β-equality for the essences in this extended system would be undecidable;
4. A *Tactic language*, such as the one of Coq, should be useful. Currently Bull has no tactics: conceiving such a language should be feasible in the medium term.

References

1 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A bi-directional refinement algorithm for the calculus of (co)inductive constructions. *Logical Methods in Computer Science*, 8(1), 2012.

2 Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

3 Franco Barbanera and Simone Martini. Proof-functional connectives and realizability. *Archive for Mathematical Logic*, 33:189–211, 1994.

4 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

5 Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda calculus with types.* Cambridge University Press, 2013.

6 Stefano Berardi. *Type dependence and Constructive Mathematics.* PhD thesis, University of Turin, 1990.

7 Jan Bessai, Jakob Rehof, and Boris Düdder. Fast verified BCD subtyping. In *Models, Mindsets, Meta: The What, the How, and the Why Not? - Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*, volume 11200 of *Lecture Notes in Computer Science*, pages 356–371. Springer, 2018.

8 Olivier Boite. Proof reuse with extended inductive types. In *Theorem Proving in Higher Order Logics (TPHOLs)*, pages 50–65, 2004.

9 Viviana Bono, Betti Venneri, and Lorenzo Bettini. A typed lambda calculus with intersection types. *Theoretical Computer Science*, 398(1-3):95–113, 2008.

10 Beatrice Capitani, Michele Loreti, and Betti Venneri. Hyperformulae, Parallel Deductions and Intersection Types. *Electronic Notes in Theoretical Computer Science*, 50(2):180–198, 2001.

11 Joshua E. Caplan and Mehdi T. Harandi. A logical framework for software proof reuse. In *Symposium on Software Reusability (SSR)*, pages 106–113, 1995.

12 Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.

13 Joëlle Despeyroux, Amy P. Felty, and André Hirschowitz. Higher-order abstract syntax in coq. In Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, editors, *Typed Lambda Calculi and Applications, TLCA*, volume 902 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 1995.

14 Roberto Di Cosmo. *Isomorphisms of types: from λ-calculus to information retrieval and language design.* Birkhauser, 1995.

15 Daniel J. Dougherty, Ugo de'Liguoro, Luigi Liquori, and Claude Stolze. A realizability interpretation for intersection and union types. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 10017 of *Lecture Notes in Computer Science*, pages 187–205. Springer-Verlag, 2016.

16 Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher order unification via explicit substitutions. *Information and Computation*, 157(1-2):183–235, 2000.

17 Andrej Dudenhefner, Moritz Martens, and Jakob Rehof. The intersection type unification problem. In *Formal Structures for Computation and Deduction (FSCD)*, pages 19:1–19:16. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

18 Andrej Dudenhefner and Paweł Urzyczyn. Kripke semantics for intersection formulas. Tenth Workshop on Intersection Types and Related Systems, 2020.

19 Thomas Ehrhard. Non-idempotent intersection types in logical form. In *Foundations of Software Science and Computation Structures (FOSSACS/ETAPS)*, volume 12077 of *Lecture Notes in Computer Science*, pages 198–216. Springer, 2020.

20 Amy Felty and Douglas J. Howe. Generalization and reuse of tactic proofs. In *Logic Programming and Automated Reasoning (LPAR)*, pages 1–15, 1994.

21 Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

22 Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4–5):613–673, July 2007.

23 J. Roger Hindley. The simple semantics for Coppo-Dezani-Sallé types. In *International Symposium on Programming*, pages 212–226, 1982.

24 Furio Honsell, Marina Lenisa, Luigi Liquori, and Ivan Scagnetto. Implementing Cantor's paradise. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 229–250, 2016.

25 Furio Honsell, Luigi Liquori, Claude Stolze, and Ivan Scagnetto. The Delta-framework. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37:1–37:21, 2018.

26 Gérard Huet. A unification algorithm for typed lambda-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

27 Luigi Liquori and Claude Stolze. A decidable subtyping logic for intersection and union types. In *Topics In Theoretical Computer Science (TTCS)*, volume 10608 of *Lecture Notes in Computer Science*, pages 74–90. Springer-Verlag, 2017.

28 Luigi Liquori and Claude Stolze. The Delta-calculus: Syntax and types. In *Formal Structures for Computation and Deduction (FSCD)*, pages 28:1–28:20. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

29 Edgar G. K. López-Escobar. Proof functional connectives. In *Methods in Mathematical Logic*, volume 1130 of *Lecture Notes in Mathematics*, pages 208–221. Springer-Verlag, 1985.

30 William Lovas and Frank Pfenning. Refinement types for logical frameworks and their interpretation as proof irrelevance. *Logical Methods in Computer Science*, 6(4), 2010.

31 David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2):95–130, 1986.

32 Robert K. Meyer and Richard Routley. Algebraic analysis of entailment I. *Logique et Analyse*, 15:407–428, 1972.

33 Grigori Mints. The completeness of provable realizability. *Notre Dame Journal of Formal Logic*, 30(3):420–441, 1989.

34 Alexandre Miquel. The implicit calculus of constructions. In *Typed Lambda Calculi and Applications (TLCA)*, pages 344–359. Springer-Verlag, 2001.

35 Christine Paulin-Mohring. Inductive definitions in the system coq rules and properties. In *Typed Lambda Calculi and Applications (TLCA)*, pages 328–345. Springer, 1993.

36 Frank Pfenning. Refinement types for logical frameworks. In *TYPES*, pages 285–299, 1993.

37 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Notices*, volume 23(7), pages 199–208. ACM, 1988.

38 Benjamin C. Pierce. *Programming with intersection types, union types, and bounded polymorphism.* PhD thesis, Technical Report CMU-CS-91-205. Carnegie Mellon University, 1991.

39 Elaine Pimentel, Simona Ronchi Della Rocca, and Luca Roversi. Intersection types from a proof-theoretic perspective. *Fundamenta Informaticae*, 121(1-4):253–274, 2012.

40 Garrel Pottinger. A type assignment for the strongly normalizable λ-terms. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 561–577. Academic Press, 1980.

41 Jason Reed. Higher-order constraint simplification in dependent type theory. In *Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)*, pages 49–56. ACM, 2009.

42 Simona Ronchi Della Rocca and Luca Roversi. Intersection logic. In *Computer Science Logic (CSL)*, volume 2142 of *Lecture Notes in Computer Science*, pages 421–428. Springer-Verlag, 2001.

43 Claude Stolze. *Combining union, intersection and dependent types in an explicitly typed lambda-calculus*. PhD thesis, Université Côte d'Azur, Inria, 2019.

44 Claude Stolze. Bull. `https://github.com/cstolze/Bull`, 2020.

45 Claude Stolze, Luigi Liquori, Furio Honsell, and Ivan Scagnetto. Towards a logical framework with intersection and union types. In *Logical Frameworks and Meta-languages: Theory and Practice (LFMTP)*, pages 1–9, 2017.

46 Pawel Urzyczyn. Type reconstruction in $\mathrm{f}_{\mathrm{omega}}$. *Mathematical Structures in Computer Science*, 7(4):329–358, 1997.

47 Betti Venneri. Intersection types as logical formulae. *Journal of Logic and Computation*, 4(2):109–124, 1994.

48 Beta Ziliani and Matthieu Sozeau. A unification algorithm for Coq featuring universe polymorphism and overloading. In *ACM SIGPLAN Notices*, volume 50(9), pages 179–191. ACM, 2015.

A Examples

This Appendix presents some examples in LF_Δ along with their code in Bull, showing a uniform approach to the encoding of a plethora of type disciplines and systems which ultimately stem or can capitalize from strong proof-functional connectives and subtyping. The framework LF_Δ presented in [25], and its software incarnation Bull introduced in this paper, are the first to accommodate all the examples and counterexamples that have appeared in the literature. In what follows, we denote by λ^{BDdL} [2] the union and intersection type assignment system à la Curry. Type inference of λ^{BDdL} is, of course, undecidable. We start by showing the expressive power of LF_Δ in encoding classical features of typing disciplines with strong intersection and union. For these examples, we set $\Sigma \overset{def}{=} \sigma{:}\mathsf{Type}, \tau{:}\mathsf{Type}$.

Auto application. The judgment $\vdash_{\lambda^{\mathrm{BDdL}}} \lambda x.x\,x : \sigma \cap (\sigma \to \tau) \to \tau$ in λ^{BDdL}, is rendered in LF_Δ by the LF_Δ judgment $\vdash_\Sigma \lambda x{:}\sigma \cap (\sigma \to \tau).(\mathsf{pr}_2\, x)\,(\mathsf{pr}_1\, x) : \sigma \cap (\sigma \to \tau) \to \tau$.

Polymorphic identity. The judgment $\vdash_{\lambda^{\mathrm{BDdL}}} \lambda x.x : (\sigma \to \sigma) \cap (\tau \to \tau)$ in λ^{BDdL}, is rendered in LF_Δ by the judgment $\vdash_\Sigma \langle \lambda x{:}\sigma.x, \lambda x{:}\tau.x \rangle : (\sigma \to \sigma) \cap (\tau \to \tau)$.

Commutativity of union. The judgment $\vdash_\Sigma \lambda x.x : (\sigma \cup \tau) \to (\tau \cup \sigma)$ in λ^{BDdL}, is rendered in LF_Δ by the judgment $\vdash_\Sigma \lambda x{:}\sigma{\cup}\tau.[\lambda y{:}\sigma.\mathsf{in}_2^\tau\, y, \lambda y{:}\tau.\mathsf{in}_1^\sigma\, y]\, x : (\sigma \cup \tau) \to (\tau \cup \sigma)$.
The Bull code corresponding to these examples is the following:

```
Axiom (s t : Type).
Definition auto_application (x : s & (s -> t)) := (proj_r x) (proj_l x).
Definition poly_id : (s -> s) & (t -> t) := let id1 x := x in
                                         let id2 x := x in < id1, id2 >.
Definition commut_union (x : s | t) := smatch x with
                                          x : s ⇒ inj_r t x
                                        , x : t ⇒ inj_l s x
                                        end.
```

Atomic propositions, non-atomic goals and non-atomic programs: $\alpha, \gamma_0, \pi_0 : \mathsf{Type}$

Goals and programs: $\gamma = \alpha \cup \gamma_0 \qquad \pi = \alpha \cup \pi_0$

Constructors (implication, conjunction, disjunction).

$$\begin{array}{lcl}
\mathsf{impl} & : & (\pi \to \gamma \to \gamma_0) \cap (\gamma \to \pi \to \pi_0) \\
\mathsf{impl}_1 & = & \lambda x{:}\pi.\lambda y{:}\gamma.\mathsf{in}_2^{\alpha}\,(\mathsf{pr}_1\,\mathsf{impl}\,x\,y) \qquad \mathsf{impl}_2 = \lambda x{:}\gamma.\lambda y{:}\pi.\mathsf{in}_2^{\alpha}\,(\mathsf{pr}_2\,\mathsf{impl}\,x\,y) \\
\mathsf{and} & : & (\gamma \to \gamma \to \gamma_0) \cap (\pi \to \pi \to \pi_0) \\
\mathsf{and}_1 & = & \lambda x{:}\gamma.\lambda y{:}\gamma.\mathsf{in}_2^{\alpha}\,(\mathsf{pr}_1\,\mathsf{and}\,x\,y) \qquad \mathsf{and}_2 = \lambda x{:}\pi.\lambda y{:}\pi.\mathsf{in}_2^{\alpha}\,(\mathsf{pr}_2\,\mathsf{and}\,x\,y) \\
\mathsf{or} & : & (\gamma \to \gamma \to \gamma_0) \qquad \mathsf{or}_1 = \lambda x{:}\gamma.\lambda y{:}\gamma.\mathsf{in}_2^{\alpha}\,(\mathsf{or}\,x\,y)
\end{array}$$

$\mathsf{solve}\,p\,g$ indicates that the judgment $p \vdash g$ is valid.

$\mathsf{bchain}\,p\,a\,g$ indicates that, if $p \vdash g$ is valid, then $p \vdash a$ is valid.

$\mathsf{solve} \quad : \quad \pi \to \gamma \to \mathsf{Type} \qquad \mathsf{bchain} : \pi \to \alpha \to \gamma \to \mathsf{Type}$

Rules for solve:

$$\begin{array}{lcl}
- & : & \Pi(p{:}\pi)(g_1, g_2{:}\gamma).\mathsf{solve}\,p\,g_1 \to \mathsf{solve}\,p\,g_2 \to \mathsf{solve}\,p\,(\mathsf{and}_1\,g_1\,g_2) \\
- & : & \Pi(p{:}\pi)(g_1, g_2{:}\gamma).\mathsf{solve}\,p\,g_1 \to \mathsf{solve}\,p\,(\mathsf{or}_1\,g_1\,g_2) \\
- & : & \Pi(p{:}\pi)(g_1, g_2{:}\gamma).\mathsf{solve}\,p\,g_2 \to \mathsf{solve}\,p\,(\mathsf{or}_1\,g_1\,g_2) \\
- & : & \Pi(p_1, p_2{:}\pi)(g{:}\gamma).\mathsf{solve}\,(\mathsf{and}_2\,p_1\,p_2)\,g \to \mathsf{solve}\,p_1\,(\mathsf{impl}_1\,p_2\,g) \\
- & : & \Pi(p{:}\pi)(a{:}\alpha)(g{:}\gamma).\mathsf{bchain}\,p\,a\,g \to \mathsf{solve}\,p\,g \to \mathsf{solve}\,p\,(\mathsf{in}_1^{\gamma_0}\,a)
\end{array}$$

Rules for bchain:

$$\begin{array}{lcl}
- & : & \Pi(a{:}\alpha)(g{:}\gamma).\mathsf{bchain}\,(\mathsf{impl}_2\,g\,(\mathsf{in}_1^{\pi_0}\,a))\,a\,g \\
- & : & \Pi(p_1, p_2{:}\pi)(a{:}\alpha)(g{:}\gamma).\mathsf{bchain}\,p_1\,a\,g \to \mathsf{bchain}\,(\mathsf{and}_2\,p_1\,p_2)\,a\,g \\
- & : & \Pi(p_1, p_2{:}\pi)(a{:}\alpha)(g{:}\gamma).\mathsf{bchain}\,p_2\,a\,g \to \mathsf{bchain}\,(\mathsf{and}_2\,p_1\,p_2)\,a\,g \\
- & : & \Pi(p{:}\pi)(a{:}\alpha)(g, g_1, g_2{:}\gamma).\mathsf{bchain} \\
 & & (\mathsf{impl}_2\,(\mathsf{and}_1\,g_1\,g_2)\,p)\,a\,g \to \mathsf{bchain}\,(\mathsf{impl}_2\,g_1\,(\mathsf{impl}_2\,g_2\,p))\,a\,g \\
- & : & \Pi(p_1, p_2{:}\pi)(a{:}\alpha)(g, g_1{:}\gamma).\mathsf{bchain}\,(\mathsf{impl}_2\,g_1\,p_1)\,a\,g \to \mathsf{bchain} \\
 & & (\mathsf{impl}_2\,g_1\,(\mathsf{and}_2\,p_1\,p_2))\,a\,g \\
- & : & \Pi(p_1, p_2{:}\pi)(a{:}\alpha)(g, g_1{:}\gamma).\mathsf{bchain} \\
 & & (\mathsf{impl}_2\,g_1\,p_2)\,a\,g \to \mathsf{bchain}\,(\mathsf{impl}_2\,g_1\,(\mathsf{and}_2\,p_1\,p_2))\,a\,g
\end{array}$$

Figure 8 The LF_Δ encoding of Hereditary Harrop Formulæ.

Pierce's code [38]. It shows the great expressivity of union and intersection types:

$$\begin{array}{rcl}
\text{Test} & \overset{def}{=} & \text{if } b \text{ then } 1 \text{ else } -1 : Pos \cup Neg \\
\text{Is\_0} & : & (Neg \to F) \cap (Zero \to T) \cap (Pos \to F) \\
(\text{Is\_0 Test}) & : & F
\end{array}$$

The expressive power of union types highlighted by Pierce is rendered in LF_Δ by:

$$\begin{array}{rcl}
Neg & : & \mathsf{Type} \quad Zero : \mathsf{Type} \quad Pos : \mathsf{Type} \quad T : \mathsf{Type} \quad F : \mathsf{Type} \quad \text{Test} : Pos \cup Neg \\
\text{Is\_0} & : & (Neg \to F) \cap ((Zero \to T) \cap (Pos \to F)) \\
\text{Is\_0\_Test} & \overset{def}{=} & [\lambda x{:}Pos.(\mathsf{pr}_2\,\mathsf{pr}_2\,\text{Is\_0})\,x, \lambda x{:}Neg.(\mathsf{pr}_1\,\text{Is\_0})\,x]\,\text{Test}
\end{array}$$

The Bull code corresponding to this example is the following:

```
Axiom (Neg Zero Pos T F : Type) (Test : Pos | Neg).
Axiom Is_0 : (Neg -> F) & (Zero -> T) & (Pos -> F).
Definition Is_0_Test := smatch Test with
                          x ⇒ coe (Pos -> F) Is_0 x
                        , x ⇒ coe (Neg -> F) Is_0 x
                        end.
```

Hereditary Harrop formulæ. The encoding of Hereditary Harrop's Formulæ is one of the motivating examples given by Pfenning for introducing Refinement Types in LF [36, 30]. In LF_Δ it can be expressed as in Figure 8 and type checked in Bull, without any reference to intersection types, by a subtle use of union types. We add also rules for solving and backchaining. Hereditary Harrop formulæ can be recursively defined using two mutually recursive syntactical objects called programs (π) and goals (γ):

$$\gamma \;:=\; \alpha \mid \gamma \wedge \gamma \mid \pi \Rightarrow \gamma \mid \gamma \vee \gamma \qquad\qquad \pi \;:=\; \alpha \mid \pi \wedge \pi \mid \gamma \Rightarrow \pi$$

The Bull code is the following:

```
(* three base types: atomic propositions, non-atomic goals and non-atomic programs*)
Axiom atom : Type.
Axiom non_atomic_goal : Type.
Axiom non_atomic_prog : Type.

(* goals and programs are defined from the base types *)
Definition goal := atom | non_atomic_goal.
Definition prog := atom | non_atomic_prog.

(* constructors (implication, conjunction, disjunction) *)
Axiom impl : (prog -> goal -> non_atomic_goal) & (goal -> prog -> non_atomic_prog).
Definition impl_1 p g := inj_r atom (proj_l impl p g).
Definition impl_2 g p := inj_r atom (proj_r impl g p).
Axiom and : (goal -> goal -> non_atomic_goal) & (prog -> prog -> non_atomic_prog).
Definition and_1 g1 g2 := inj_r atom (proj_l and g1 g2).
Definition and_2 p1 p2 := inj_r atom (proj_r and p1 p2).
Axiom or : (goal -> goal -> non_atomic_goal).
Definition or_1 g1 g2 := inj_r atom (or g1 g2).

(* solve p g means: the judgment p |- g is valid *)
Axiom solve : prog -> goal -> Type.

(* backchain p a g means: if p |- g is valid, then p |- a is valid *)
Axiom backchain : prog -> atom -> goal -> Type.

(* rules for solve *)
Axiom solve_and : forall p g1 g2, solve p g1 -> solve p g2 -> solve p (and_1 g1 g2).
Axiom solve_or1 : forall p g1 g2, solve p g1 -> solve p (or_1 g1 g2).
Axiom solve_or2 : forall p g1 g2, solve p g2 -> solve p (or_1 g1 g2).
Axiom solve_impl : forall p1 p2 g, solve (and_2 p1 p2) g -> solve p1 (impl_1 p2 g).
Axiom solve_atom : forall p a g, backchain p a g -> solve p g ->
                                solve p (inj_l non_atomic_goal a).

(* rules for backchain *)
Axiom backchain_and1 :
 forall p1 p2 a g, backchain p1 a g -> backchain (and_2 p1 p2) a g.
Axiom backchain_and2 :
 forall p1 p2 a g, backchain p1 a g -> backchain (and_2 p1 p2) a g.
Axiom backchain_impl_atom :
 forall a g, backchain (impl_2 g (inj_l non_atomic_prog a)) a g.
Axiom backchain_impl_impl :
 forall p a g g1 g2,  backchain (impl_2 (and_1 g1 g2) p) a g ->
                      backchain (impl_2 g1 (impl_2 g2 p)) a g.
Axiom backchain_impl_and1 :
 forall p1 p2 a g g1, backchain (impl_2 g1 p1) a g ->
```

```
                         backchain (impl_2 g1 (and_2 p1 p2)) a g.
Axiom backchain_impl_and2 :
 forall p1 p2 a g g1, backchain (impl_2 g1 p2) a g ->
                         backchain (impl_2 g1 (and_2 p1 p2)) a g.
```

Natural deductions in normal form. The second motivating example for intersection types given in [36, 30] is *natural deductions in normal form.* A natural deduction is in normal form if there are no applications of elimination rules of a logical connective immediately following their corresponding introduction, in the main branch of a subderivation.

$$\supset_I : \Pi A, B{:}o.(\mathit{Elim}(A) \to \mathrm{Nf}(B)) \to \mathrm{Nf}^0(A \supset B) \qquad o : \mathsf{Type} \qquad \mathit{Elim}, \mathrm{Nf}^0 : o \to \mathsf{Type} \ \supset : o \to o \to o$$

$$\supset_E : \Pi A, B{:}o.\mathit{Elim}(A \supset B) \to \mathrm{Nf}^0(A) \to \mathit{Elim}(B) \qquad \mathrm{Nf} \equiv \lambda A{:}o.\mathrm{Nf}^0(A) \cup \mathit{Elim}(A)$$

The corresponding Bull code is the following:

```
Axiom (o : Type) (impl : o -> o -> o) (Elim Nf0 : o -> Type).
Definition Nf A := Nf0 A | Elim A.
Axiom impl_I : forall A B, (Elim A -> Nf B) -> (Nf0 (impl A B)).
Axiom impl_E : forall A B,  Elim (impl A B) -> Nf0 A -> Elim B.
```

The encoding we give in LF_Δ is a slightly improved version of the one in [36, 30]: as Pfenning, we restrict to the purely implicational fragment. As in the previous example, we use union types to define normal forms $\mathrm{Nf}(A)$ either as pure elimination-deductions from hypotheses $\mathit{Elim}(A)$ or normal form-deductions $\mathrm{Nf}^0(A)$. This example is interesting in itself, being the prototype of the encoding of type systems using canonical and atomic syntactic categories [22] and also of Fitch Set Theory [24].

Encoding of Edinburgh LF. A shallow encoding of LF [21] in LF_Δ making essential use of intersection types can be also type checked. Here we consider LF as defined with several syntactical categories :

$$\begin{array}{rcllrcll} M & ::= & c \mid x \mid \lambda x{:}\sigma.M \mid M\,M & \text{Objects} & K & ::= & \star \mid \Pi x{:}\sigma.K & \text{Kinds} \\ \sigma & ::= & a \mid \Pi x{:}\sigma.\sigma \mid \lambda x{:}\sigma.\sigma \mid \sigma\,M & \text{Families} & S & ::= & \Box & \text{Superkind} \end{array}$$

We encode LF using *Higher-Order Abstract Syntax* (HOAS) [37, 13]. Moreover, using intersection types, we can use the same axiom in order to encode both λ-abstractions on objects and λ-abstractions on families, as well as a single axiom to encode both application on objects and application on families, and a single axiom to encode both dependent products on families and dependent products on kinds.

The typing rules, defined as axioms, have similar essence, it could be interesting to investigate how to profit from these similarities for better encodings. We have decided to explore two different approaches:

- for the typing rules, we chose to define distinct axioms `of_1`, `of_2`, and `of_3` of different precise types. We have not found a way for these axioms to share the same essence, so we have to write different (but very similar) rules for each of these different typing judgment;
- for equality, we chose to define a single axiom `eq : (obj | fam) -> (obj | fam) -> Type`. The type of this axiom is not very precise (it implies we could compare objects and families), but we can factorize equality rules with the same shape, e.g. the rule `beta_eq` define equalities between a β-redex and its contractum, both on objects and on families.

The corresponding Bull code is the following:

```
Axiom obj' : Type.
Axiom fam' : Type.
Axiom knd' : Type.
Axiom sup' : Type.
Axiom same : (obj' & fam' & knd' & sup').
Axiom term : (obj' | fam' | knd' | sup') -> Type.

(* obj, fam, knd, and sup are different terms *)
(* but their essence is always (term same) *)
Definition obj := term (coe (obj' | fam' | knd' | sup') (coe obj' same)).
Definition fam := term (coe (obj' | fam' | knd' | sup') (coe fam' same)).
Definition knd := term (coe (obj' | fam' | knd' | sup') (coe knd' same)).
Definition sup := term (coe (obj' | fam' | knd' | sup') (coe sup' same)).

(* Syntax *)
Axiom star : knd.
Axiom sqre : sup.
Axiom lam : (fam -> (obj -> obj) -> obj) & (fam -> (obj -> fam) -> fam).
Definition lam_obj := coe (fam -> (obj -> obj) -> obj) lam.
Definition lam_fam := coe (fam -> (obj -> fam) -> fam) lam.
Axiom pi : (fam -> (obj -> fam) -> fam) & (fam -> (obj -> knd) -> knd).
Definition pi_fam := coe (fam -> (obj -> fam) -> fam) pi.
Definition pi_knd := coe (fam -> (obj -> knd) -> knd) pi.
Axiom app : (obj -> obj -> obj) & (fam -> obj -> fam).
Definition app_obj := coe (obj -> obj -> obj) app.
Definition app_fam := coe (fam -> obj -> fam) app.

(* Typing rules *)
Axiom of_1 : obj -> fam -> Type.
Axiom of_2 : fam -> knd -> Type.
Axiom of_3 : knd -> sup -> Type.
Axiom of_ax : of_3 star sqre.

(* Rules for lambda-abstraction are "essentially" the same *)
Definition of_lam_obj := forall t1 t2 t3, of_2 t1 star ->
      (forall x, of_1 x t1 -> of_1 (t2 x) (t3 x)) -> of_1 (lam_obj t1 t2) (pi_fam t1 t3).
Definition of_lam_fam := forall t1 t2 t3, of_2 t1 star ->
      (forall x, of_1 x t1 -> of_2 (t2 x) (t3 x)) -> of_2 (lam_fam t1 t2) (pi_knd t1 t3).
(* Rules for product are ''essentially'' the same *)
Definition of_pi_fam := forall t1 t2, of_2 t1 star ->
 (forall x, of_1 x t1 -> of_2 (t2 x) star) -> of_2 (pi_fam t1 t2) star.
Definition of_pi_knd := forall t1 t2, of_2 t1 star ->
 (forall x, of_1 x t1 -> of_3 (t2 x) sqre) -> of_3 (pi_knd t1 t2) sqre.
(* Rules for application are ''essentially'' the same *)
Definition of_app_obj := forall t1 t2 t3 t4, of_1 t1 (pi_fam t3 t4) ->
 of_1 t2 t3 -> of_1 (app_obj t1 t2) (t4 t2).
Definition of_app_fam := forall t1 t2 t3 t4, of_2 t1 (pi_knd t3 t4) ->
 of_1 t2 t3 -> of_2 (app_fam t1 t2) (t4 t2).

(* equality *)
Axiom eq : (obj | fam) -> (obj | fam) -> Type.
Definition c_obj (x : obj) := coe (obj | fam) x.
Definition c_fam (x : fam) := coe (obj | fam) x.
Axiom beta_eq : forall t1 f g,
  smatch f with
  (* object *)
     f ⇒ eq (c_obj (app_obj (lam_obj t1 f) g)) (c_obj (f g))
  (* family *)
   , f ⇒ eq (c_fam (app_fam (lam_fam t1 f) g)) (c_fam (f g))
   end.
Axiom lam_eq : forall t1 t2 f,
  eq (c_fam t1) (c_fam t2) ->
  smatch f with
  (* object *)
     f ⇒ forall g, (forall x, eq (c_obj (f x)) (c_obj (g x))) ->
           eq (c_obj (lam_obj t1 f)) (c_obj (lam_obj t2 f))
  (* family *)
   , f ⇒ forall g, (forall x, eq (c_fam (f x)) (c_fam (g x))) ->
           eq (c_fam (lam_fam t1 f)) (c_fam (lam_fam t2 f))
   end.
Axiom app_eq : forall n1 n2 m1,
  eq (c_obj n1) (c_obj n2) ->
  smatch m1 with
  (* object *)
    m1 ⇒ forall m2, eq (c_obj m1) (c_obj m2) ->
         eq (c_obj (app_obj m1 n1)) (c_obj (app_obj m2 n2))
  (* family *)
  , m1 ⇒ forall m2, eq (c_fam m1) (c_fam m2) ->
         eq (c_fam (app_fam m1 n1)) (c_fam (app_fam m2 n2))
  end.
Axiom pi_eq : forall m1 m2 n1 n2,
  eq (c_fam m1) (c_fam m2) ->
  (forall x, eq (c_fam (n1 x)) (c_fam (n2 x))) ->
  eq (c_fam (pi_fam m1 n1)) (c_fam (pi_fam m2 n2)).
```