6th International Conference on Formal Structures for Computation and Deduction

FSCD 2021, July 17–24, 2021, Buenos Aires, Argentina (Virtual Conference)

Edited by

Naoki Kobayashi

LIPICS

Editors

Naoki Kobayashi
The University of Tokyo, Japan
koba@is.s.u-tokyo.ac.jp

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 978-3-95977-191-7

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-191-7.

Publication date
July, 2021

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.2021.0

LIPIcs – Leibniz International Proceedings in Informatics

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

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

This volume contains the proceedings of the 6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021). The conference was held during July 17 – July 24, 2021 as a virtual conference. It was initially planned to be held in Buenos Aires, Argentina, but was actually held as a virtual conference due to the COVID-19 pandemic. The conference (`https://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 Zena M. Ariola (University of Oregon, USA), Nao Hirokawa (JAIST, Japan), Elaine Pimentel (UFRN, Brazil), and Sam Staton (University of Oxford, UK). FSCD 2021 received 72 submissions with contributing authors from 22 countries. The program committee consisted of 31 members from 18 countries. Each submitted paper has been reviewed by at least three PC members with the help of 130 external reviewers. The reviewing process, which included a rebuttal phase, took place over nine weeks. A total of 28 regular research papers were accepted for publication and are included in these proceedings. The Program Committee awarded the FSCD 2021 Best Paper Award by Junior Researchers to Tom de Jong and Martín Hötzel Escardó for their paper "Predicative Aspects of Order Theory in Univalent Foundations".

In addition to the main program, 7 FSCD-associated workshops were held, also virtually:

- HoTT/UF: 6th Workshop on Homotopy Type Theory/Univalent Foundations
- ITRS: 10th Workshop on Intersection Types and Related Systems
- WPTE: 7th International Workshop on Rewriting Techniques for Program Transformations and Evaluation
- UNIF: 35th International Workshop on Unification
- LSFA: 16th Logical and Semantics Frameworks with Applications
- IWC: 10th International Workshop on Confluence
- IFIP WG 1.6: 24th meeting of the IFIP Working Group 1.6: Rewriting

This volume of FSCD 2021 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 for his prompt replies to any questions regarding the production of these proceedings.

Many people have helped to make FSCD 2021 a successful meeting. On behalf of the Program Committee, I thank the authors of submitted papers for considering FSCD as a venue for their work and the invited speakers who have agreed to speak at this meeting. The Program Committee and the external reviewers deserve special thanks for their careful review and evaluation of the submitted papers. The EasyChair conference management system has been a useful tool in all phases of the work of the Program Committee.

The associated workshops have made a big contribution to the lively scientific atmosphere of this virtual meeting and I thank the workshop organizers and workshop chairs Mauricio Ayala-Rincón and Carlos López Pombo for their efforts and enthusiasm in making sure that workshops continued to be an important element of FSCD. Alejandro Díaz-Caro, the Conference Chair, and the organising committee members of FSCD 2021 deserve special appreciation for the overall organization of the conference; although the conference was held virtually at the end, they pursued various possibilities, including a hybrid conference. Carsten Fuhs, as Publicity Chair, made a significant contribution in advertising the conference. The steering committee provided excellent guidance in setting up this meeting and in ensuring that FSCD will have a bright and enduring future. I would like to especially thank Delia Kesner, the steering committee chair, for her numerous pieces of advice in managing the conference.

FSCD 2021 was held in-cooperation with ACM SIGLOG and ACM SIGPLAN. It was supported by Universidad de Buenos Aires, Universidad Nacional de Quilmes, CONICET (Grant RD2256), Ministerio de Ciencia, Tecnología e Innovación (Grant RC-RPI-2020-00004), Onapsis, IRIF (Institut de Recherche en Informatique Fondamentale), FUNDACEN (Fundación Ciencias Exactas y Naturales), the CNRS/CONICET International Research Project SINFIN, and CertiSur. Finally, I thank all of the participants of the virtual conference for contributing to the success of the event.

Naoki Kobayashi
Program Chair of FSCD 2021

Committees

PROGRAM COMMITTEE

Mauricio Ayala-Rincón	Universidade de Brasília
Stefano Berardi	University of Torino
Frédéric Blanqui	INRIA
Eduardo Bonelli	Stevens Institute of Technology
Évelyne Contejean	CNRS, Université Paris-Saclay
Thierry Coquand	University of Gothenburg
Thomas Ehrhard	CNRS, Université de Paris
Santiago Escobar	Univ. Politècnica de València
José Espírito Santo	University of Minho
Claudia Faggian	CNRS, Université de Paris
Amy Felty	University of Ottawa
Santiago Figueira	Universidad de Buenos Aires
Marcelo Fiore	University of Cambridge
Marco Gaboardi	Boston University
Silvia Ghilezan	University of Novi Sad & Mathematical Institute SASA
Ichiro Hasuo	National Institute of Informatics
Delia Kesner	Université de Paris
Naoki Kobayashi (chair)	The University of Tokyo
Robbert Krebbers	Radboud University Nijmegen
Temur Kutsia	Johannes Kepler University Linz
Barbara König	University of Duisburg-Essen
Marina Lenisa	University of Udine
Naoki Nishida	Nagoya University
Luke Ong	University of Oxford
Paweł Parys	University of Warsaw
Jakob Rehof	TU Dortmund University
Camilo Rocha	Pontificia Univ. Javeriana Cali
Alexandra Silva	University College London
Alwen Tiu	Australian National University
Sarah Winkler	University of Bozen-Bolzano
Hongseok Yang	KAIST, South Korea

CONFERENCE CHAIR

Alejandro Díaz-Caro	Quilmes Univ. & ICC/CONICET

WORKSHOP CHAIRS

Mauricio Ayala-Rincón	Universidade de Brasília
Carlos López Pombo	Universidad de Buenos Aires

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

ORGANISING COMMITTEE

Local Organisers

Alejandro Díaz-Caro	Universidad Nacional de Quilmes & ICC (UBA/CONICET)
Santiago Figueira	Universidad de Buenos Aires & ICC (UBA/CONICET)
Carlos López Pombo	Universidad de Buenos Aires & ICC (UBA/CONICET)
Ricardo Rodríguez	Universidad de Buenos Aires & ICC (UBA/CONICET)

Collaborators

Mauricio Ayala-Rincón	Universidade de Brasília
Mauro Jaskelioff	Universidad Nacional de Rosario & CIFASIS (UNR/CONICET)
Nora Szasz	Universidad ORT Uruguay
Beta Ziliani	Universidad Nacional de Córdoba & CONICET

PUBLICITY CHAIR

Carsten Fuhs	Birkbeck, University of London

STEERING COMMITTEE

Zena M. Ariola	University of Oregon
Mauricio Ayala-Rincón	University of Brasilia
Carsten Fuhs	Birkbeck, University of London
Herman Geuvers	Radboud University & Eindhoven University of Technology
Silvia Ghilezan	University of Novi Sad & Mathematical Institute SASA
Stefano Guerrini	CNRS, Université Sorbonne Paris Nord
Delia Kesner (Chair)	Université de Paris
Hélène Kirchner	Inria
Cynthia Kop	Radboud University
Damiano Mazza	CNRS, Université Sorbonne Paris Nord
Luke Ong	Oxford University
Jakob Rehof	TU Dortmund
Jamie Vicary	University of Cambridge

External Reviewers

Andreas Abel
Beniamino Accattoli
Benedikt Ahrens
Andrea Aler Tubella
Fabio Alessi
Takahito Aoto
Thaynara Arielly de Lima
Kazuyuki Asada
Martin Avanzini
Arthur Azevedo de Amorim
David Baelde
Patrick Baillot
Demis Ballis
Pablo Barenbaum
Chris Barrett
Yves Bertot
Jan Bessai
Marc Bezem
Siddharth Bhaskar
Filippo Bonchi
Flavien Breuvart
Guillaume Burel
Marco Carbone
Davide Castelnovo
Horatiu Cirstea
Mario Coppo
Łukasz Czajka
Mariangiola Dezani-Ciancaglini
Pietro Di Gianantonio
Amina Doumane
Gilles Dowek
Paul Downen
Andrej Dudenhefner
Peter Dybjer
Raul Fervari
Mathias Fleury
Yannick Forster
Soichiro Fujii
Nicola Gambino
Richard Garner
Thomas Genet
Armaël Guéneau
Giulio Guerrieri
Walter Guttmann
Claudio Hermida
Tom Hirschowitz
Cédric Ho Thanh
Furio Honsell
Ross Horne
Naohiko Hoshino
Atsushi Igarashi
Farzad Jafarrahmani
Ohad Kammar
Shin-Ya Katsumata
Ken-Ichi Kawarabayashi
Kei Kimura
Daisuke Kimura
Oleg Kiselyov
Aleks Kissinger
Yuichi Komorida
Cynthia Kop
Roman Kuznets
Ambroise Lafont
Rodolphe Lepigre
Paul Blain Levy
Jean-Jacques Lévy
Ugo de'Liguoro
Tim Lyon
Philippe Malbos
Sonia Marin
Cristina Matache
Marek Materzok
Ralph Matthes
Dylan McDermott
Doriana Medic
Paul-André Melliès
Thiago Mendonça Ferreira Ramos
Joshua Moerman
Rasmus Ejlers Møgelberg
Ike Mulder
Keisuke Nakano
Koji Nakazawa
Daniele Nantes-Sobrinho
Sara Negri
Satoru Niki
Carlos Olarte
Eugenio Orlandelli
Yota Otachi
Edi Pavlovic
Luiz Carlos Pereira

Marco Peressotti
Francesca Poggiolesi
Damien Pous
Thomas Powell
Matija Pretnar
Jakub Radoszewski
Steven Ramsay
Martin Riener
Adrian Riesco
Simona Ronchi Della Rocca
Reuben Rowe
Antonino Salibra
Tetsuya Sato
Ivan Scagnetto
Alceste Scalas
Hiroyuki Seki
Michael Shulman
Sonja Smets
Paweł Sobociński
Simon Spies
Giannos Stamoulis
Dario Stein
Sorin Stratulat
Matias Toro
Riccardo Treglia
Takeshi Tsukada
Taichi Uemura
Hiroshi Unno
Paweł Urzyczyn
John van de Wetering
Benno van den Berg
Niels van der Weide
Gerco van Heerdt
Daniel Ventura
Alicia Villanueva
Andrés Ezequiel Viso
Masaki Waga
Marcin Wrochna
Ahmed Younes
Krzysztof Ziemiański

List of Authors

Zena M. Ariola (1)
Department of Computer & Information Science,
University of Oregon, Eugene, OR, USA

Nathanael Arkor (30)
University of Cambridge, UK

Kenichi Asai (12, 16)
Ochanomizu University, Tokyo, Japan

Thibaut Balabonski (9)
Université Paris-Saclay, CNRS,
ENS Paris-Saclay, LMF,
Gif-sur-Yvette, 91190, France

Frédéric Blanqui (20)
Université Paris-Saclay, ENS Paris-Saclay, LMF,
CNRS, Inria, France

Lison Blondeau-Patissier (17)
Université Lyon, EnsL, UCBL, CNRS, LIP,
F-69342, Lyon Cedex 07, France

Pierre Clairambault (17)
Université Lyon, EnsL, UCBL, CNRS, LIP,
F-69342, Lyon Cedex 07, France

Youyou Cong (12)
Tokyo Institute of Technology, Japan

Ugo Dal Lago (23)
University of Bologna, Italy;
INRIA Sophia Antipolis, France

Anupam Das (14, 29)
University of Birmingham, UK

Tom de Jong (8)
University of Birmingham, UK

Hans-Peter Deifel (28)
Friedrich-Alexander-Universität
Erlangen-Nürnberg, Germany

Gilles Dowek (20)
Université Paris-Saclay, ENS Paris-Saclay, LMF,
CNRS, Inria, France

Paul Downen (1)
Department of Computer & Information Science,
University of Oregon, Eugene, OR, USA

Martín Hötzel Escardó (8)
University of Birmingham, UK

Maika Fujii (16)
Ochanomizu University, Tokyo, Japan

Zeinab Galal (10)
IRIF, Université de Paris, France

Didier Galmiche (13)
Université de Lorraine, CNRS, LORIA,
Nancy, France

Francesco Gavazzo (23)
University of Bologna, Italy;
INRIA Sophia Antipolis, France

Marta Gawek (13)
Université de Lorraine, CNRS, LORIA,
Nancy, France

Émilie Grienenberger (20)
Université Paris-Saclay, ENS Paris-Saclay, LMF,
CNRS, Inria, France

Nao Hirokawa (2)
Japan Advanced Institute of Science and
Technology, Ishikawa, Japan

Pieter Hofstra (26)
Dept. of Mathematics & Statistics,
University of Ottawa, Canada

Kaho Honda (12)
Ochanomizu University, Tokyo, Japan

Gabriel Hondet (20)
Université Paris-Saclay, ENS Paris-Saclay, LMF,
CNRS, Inria, France

Chiaki Ishio (12)
Ochanomizu University, Tokyo,Japan

Deepak Kapur (15)
Department of Computer Science,
University of New Mexico,
Albuquerque, NM, USA

Axel Kerinec (7)
Laboratoire LIPN, CNRS UMR 7030,
Université Sorbonne Paris-Nord, France

Dohan Kim (19)
Clarkson University, Potsdam, NY, USA

Daisuke Kimura (11)
Toho University, Japan

Cynthia Kop (31)
Department of Software Science,
Radboud University Nijmegen, The Netherlands

Antoine Lanco (9)
Université Paris-Saclay, CNRS,
ENS Paris-Saclay, Inria, LMF,
Gif-sur-Yvette, 91190, France

Dominique Larchey-Wendling (18)
Université de Lorraine, CNRS, LORIA,
Vandœuvre-lès-Nancy, France

Christopher Lynch (19)
Clarkson University, Potsdam, NY, USA

Giulio Manzonetto (7)
Laboratoire LIPN, CNRS UMR 7030,
Université Sorbonne Paris-Nord, France

Cristina Matache (25)
University of Oxford, UK

Dylan McDermott (30)
Reykjavik University, Iceland

Guillaume Melquiond (9)
Université Paris-Saclay, CNRS,
ENS Paris-Saclay, Inria, LMF,
Gif-sur-Yvette, 91190, France

Stefan Milius (28)
Friedrich-Alexander-Universität
Erlangen-Nürnberg, Germany

Sean Moss (25)
University of Oxford, UK

Daniel Méry (13)
Université de Lorraine, CNRS, LORIA,
Nancy, France

Koji Nakazawa (11)
Nagoya University, Japan

Daniele Nantes-Sobrinho (21)
University of Brasília, Brazil

Vivek Nigam (3)
Huawei Munich Research Center, Germany

Carlos Olarte (3)
School of Science and Technology,
Federal University of Rio Grande Do Norte,
Natal, Brazil

Jason Parker (26)
Department of Mathematics & Computer
Science, Brandon University, Canada

Joseph W. N. Paulus (21)
University of Groningen, The Netherlands

Elaine Pimentel (3)
Department of Mathematics,
Federal University of Rio Grande Do Norte,
Natal, Brazil

Paolo Pistone (27)
University of Bologna, Italy

Jorge A. Pérez (21)
University of Groningen, The Netherlands;
CWI, Amsterdam, The Netherlands

Alex A. Rice (14)
University of Cambridge, UK

Simona Ronchi Della Rocca (7)
Computer Science Department,
University of Torino, Italy

Ken Sakayori (32)
The University of Tokyo, Japan

Kenji Saotome (11)
Nagoya University, Japan

Philip J. Scott (26)
Dept. of Mathematics & Statistics,
University of Ottawa, Canada

Jakob Grue Simonsen (5)
Department of Computer Science,
University of Copenhagen, Denmark

Rick Statman (6)
Carnegie Mellon University,
Pittsburgh, PA, USA

Sam Staton (4, 25)
University of Oxford, UK

François Thiré (20)
Nomadic Labs, Paris, France

Luca Tranchini (27)
Eberhard Karls Universität Tübingen, Germany

Takeshi Tsukada (32)
Chiba University, Japan

Deivid Vale (31)
Department of Software Science,
Radboud University Nijmegen,
The Netherlands

Vincent van Oostrom (24)
Universität Innsbruck, Austria

Niccolò Veltri (22)
Department of Software Science, Tallinn
University of Technology, Estonia

Thorsten Wißmann (28)
Radboud University Nijmegen, The Netherlands

Duality in Action

Paul Downen
Department of Computer & Information Science, University of Oregon, Eugene, OR, USA

Zena M. Ariola
Department of Computer & Information Science, University of Oregon, Eugene, OR, USA

Abstract

The duality between "true" and "false" is a hallmark feature of logic. We show how this duality can be put to use in the theory and practice of programming languages and their implementations, too. Starting from a foundation of constructive logic as dialogues, we illustrate how it describes a symmetric language for computation, and survey several applications of the dualities found therein.

2012 ACM Subject Classification Theory of computation → Logic

Keywords and phrases Duality, Logic, Curry-Howard, Sequent Calculus, Rewriting, Compilation

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.1

Category Invited Talk

Funding This work is supported by the NSF under Grants No. 1719158 and No. 1423617.

1 Introduction

Mathematical logic, through the Curry-Howard correspondence [25], has undoubtably proved its usefulness in the theory of computation and programming languages. It gave us tools to reason effectively about the behavior of programs, and serves as the backbone for proof assistants that let us formally specify and verify program correctness. We've found that the same correspondence with logic provides a valuable inspiration for the implementation of programming languages, too. The entire computer industry is based on the difference between the *ability to know something* versus *actually knowing it*, and the fact that real resources are needed to go from one to the other. In other words, the cost of an answer is just as important as its correctness. Thankfully, logic provides solutions for both.

We start with a story on the nature of "truth" (Section 2), and investigate different logical foundations with increasing nuance. The *classical* view of ultimate truth is quite different from constructive truth, embodied by intuitionistic logic, requiring that proofs be backed with evidence. However, the intuitionistic view of truth sadly discards many of the pleasant dualities of classical logic. Instead, we can preserve duality in constructivity by re-imagining logic not as a solitary exercise, but as a dialogue between two disagreeing characters: the optimistic Sage who argues in favor, and the doubtful Skeptic who argues against. Symmetry is restored – still backed by evidence – when both sides can enter the debate.

This dialogic notion of constructive classical logic can be seen as a symmetric language for describing computation (Section 3). The Sage and Skeptic correspond to producers and consumers of information; their debate corresponds to interaction in a program. The two-sided viewpoint brings up many dualities that are otherwise hidden implicitly in today's programming languages: questions versus answers, programs versus contexts, construction versus destruction, and so on. But more than this, the symmetric calculus allows us to express more types – and more relationships between them – than possible in the conventional programming languages used today.

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi; Article No. 1; pp. 1:1–1:32
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

From there, we survey several applications of computational duality (Section 4) across both theoretical and practical concerns. The theory of the untyped λ-calculus can be improved by viewing functions as codata (Section 4.1). Duality can help us design and analyze different forms of loops found in programs and proofs (Section 4.2). Compilers use intermediate languages to help generate code and perform optimizations, and logic can be put to action at this middle stage in the life of a program (Section 4.3). To bring it all together, a general-purpose method based on orthogonality provides a framework for developing models of safety that let us prove that well-typed programs do what we want (Section 4.4).

2 Logic as Dialogues

One of the most iconic principles of classical logic is the *law of the excluded middle*, $A \vee \neg A$: everything is either true or false. This principle conjures ideas of an omniscient notion of truth. That once all is said and done, every claim must fall within one of these two cases. While undoubtedly useful for proving theorems, the issue with the law of the excluded middle is that we as mortals are not omniscient: we cannot decide for everything, *a priori*, which case it is. As a consequence, reckless use of the excluded middle means that even if we know something must be true, we might not know exactly *why* it is true.

Consider this classic proof about irrational power [20].

▶ **Theorem 1.** *There exist two irrational numbers, x and y, such that x^y is rational.*

Proof. Since $\sqrt{2}$ is irrational, consider $\sqrt{2}^{\sqrt{2}}$. *This exponent is either rational or not.*

- If $\sqrt{2}^{\sqrt{2}}$ rational, then $x = y = \sqrt{2}$ are two irrational numbers (coincidentally the same) whose exponent is rational (by assumption).
- Otherwise, $\sqrt{2}^{\sqrt{2}}$ must be irrational. In this case, observe that the exponent $(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}}$ simplifies down to just 2, because $\sqrt{2}^2 = 2$, like so: $(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}^2} = \sqrt{2}^2 = 2$. Therefore, the two chosen irrational numbers are $x = \sqrt{2}^{\sqrt{2}}$ and $y = \sqrt{2}$ whose exponent is the rational number 2. ◀

On the one hand, this proof shows Theorem 1 is true in the sense that appropriate values for x and y cannot fail to exist. On the other hand, this proof fails to actually demonstrate *which* values of x and y satisfy the required conditions; it only presents two options without definitively concluding which one is correct. The root problem is in the assertion that the "exponent is either rational or not." If we had an effective procedure to decide which of the two options is correct, we could simply choose the correct branch to pursue. But alas, we do not. Depending on an undecidable choice results in a failure to provide a concrete example verifying the truth of the theorem. Can we do better?

2.1 Constructive truth

In contrast to the proof of Theorem 1, constructive logic demands that proofs construct real evidence to back up the truth of a claim. The most popular constructive logic is *intuitionistic logic*, wherein a proposition A is only considered true when a proof produces specific evidence that verifies the truth of A [3, 24]. As such, the basic logical connectives are interpreted intuitionistically in terms of the shape of the evidence needed to verify them.

Conjunction Evidence for $A \wedge B$ consists of both evidence for A and evidence for B.
Disjunction Evidence for $A \vee B$ can be either evidence for A or evidence for B.

Existence Evidence for $\exists x{:}D.P(x)$ consists of a specific example value $n \in D$ (*e.g.,* a concrete number when the domain of objects D is $\mathbb{N}$) along with evidence for $P(n)$.

Universal Evidence for $\forall x{:}D.P(x)$ is an algorithm that, applied to any possible value n in the domain D, provides evidence for $P(n)$.

Negation Evidence for $\neg A$ is a demonstration that evidence for A generates a contradiction.

The most iconic form of evidence is for the existential quantifier $\exists x{:}D.P(x)$. Intuitionistically, we must provide a real example for x such that $P(x)$ holds. Instead, classically we are not obligated to provide any example, but only need to demonstrate that one cannot fail to exist, as in Theorem 1. This is why intuitionistic logic rejects the law of the excluded middle as a principle that holds uniformly for every proposition. Without knowing more about the details of A, we have no way to know how to construct evidence for A or for $\neg A$. But still, $A \vee \neg A$ is never false; intuitionistic logic admits there may be things not yet known.

Intuitionistic logic is famous for its connection with computation, the λ-calculus, and functional programming [25]. Constructivity also gives us a more nuanced lens to study logics. For example, one way of understanding and comparing different logics is through the propositions they prove true. In this sense, intuitionistic and classical logic are different because classical logic accepts that $A \vee \neg A$ is true in general for any A, but intuitionistic logic does not. But this reduces logics to be merely nothing more than the set of their true propositions, irrespective of the reason *why* they are true. In a world in which we care about evidence, this reductive view ignores all evidence. Instead, we can go a step further to also compare the informational content of evidence provided by different logics.

In this sense, intuitionistic logic does very well in describing why propositions are true, especially compared to classical logic. The evidence supporting the truth of different connectives (like conjunction and disjunction) and quantifiers (like existential and universal) are tailor-made to fit the situation. But the evidence demonstrating falsehood is another story. Indeed, intuitionistic logic does *not* speak directly about what it means to be false. Rather, it instead says indirectly that "not A is true," *i.e.,* $\neg A$. In this case, the evidence of falsehood is rather poor, and always cast in the same form as a hypothetical: truth would be contradictory. For example, concrete evidence that $\forall x{:}\mathbb{N}.\ x + 1 \neq 3$ is false should be a specific counterexample for which the property fails; the same informational content as the evidence needed to prove $\exists x{:}\mathbb{N}.x + 1 = 3$ is true. For example, choosing 2 for x leads to $2 + 1 \neq 3$, which is obviously wrong. Yet, an intuitionistic proof of $\neg\forall x{:}\mathbb{N}.x + 1 \neq 3$ is under no such obligation to provide a specific counterexample, it only needs to show that a counterexample cannot fail to exist. The intuitionistic treatment of falsehood sounds awfully similar to the noncommittal vagueness of classical truth. Can we do better?

2.2 Constructive dialogues

The famous asymmetry of intuitionism is reflected by its biased treatment of the two basic truth values: it demands concretely constructed evidence of truth, but leaves falsehood as the mere shadow left behind from the absence of truth. This models the scenario of a solitary Sage building evidence to support a grand theorem. When the wise Sage delivers a claim we can be sure it is true – and verify the evidence for ourselves – but what if the Sage is silent? Is that passive evidence of falsehood, or just merely an artifact that work takes time? What is missing is a devil's advocate to actively argue the other side.

In reality, the uncharted frontier on the edge of current knowledge is occupied by contentious debate. Before something is fully known, there is a space where multiple people can honestly hold different, conflicting claims, even though they are all ultimately interested in discovering the same shared truth. There is no need to be confined to the isolated work of cloistered ivory towers. Instead, there can be a dialogue between disagreeing parties, who influence one another and poke holes in questionable lines of reasoning. The search for truth is then found inside the dialogue of debate, of (at least) two sides exchanging probing questions and rebutting answers, where the victorious side defeats their opponent by eventually constructing the complete body of evidence that finally proves their position.

To keep things simple, let's assume the proposition A is under dispute by only two people: the Sage and the Skeptic. Whereas the Sage is optimistically trying to prove A is true, as before, the Skeptic is doubtful and asserts A is false. The dispute over A is resolved by the process of dialogue between the Sage and the Skeptic. But who is responsible for providing the first piece of evidence supporting their claim? Whoever has the *burden of proof.*

A *positive burden of proof* is when the Sage must provide evidence supporting that A is true. The shape of evidence for A's truth follows the shape of the disputed proposition A, and shares similarities with the evidence of truth for the same intuitionistic logical concepts.

Conjunction Evidence for $A \otimes B$ is both evidence for A and evidence for B.
Disjunction Evidence for $A \oplus B$ is either evidence for A or evidence for B.
Existence Evidence for $\exists x{:}D.P(x)$ is an example value $n \in D$ along with evidence for $P(n)$.
Negation Evidence for $\ominus A$ is the same as evidence against A.

Notice that new symbols are used for the connectives, and the evidence for negation is completely different. Both changes are due to the fact that there are other logical concepts that demand evidence of falsehood, rather than truth. These involve a *negative burden of proof*, where the Skeptic must provide evidence supporting that A is false. Just like the positive burden of proof (and contrary to intuitionistic logic), the shape of the evidence against A depends on the shape of A.

Conjunction Evidence against $A \mathbin{\&} B$ is either evidence against A or evidence against B.
Disjunction Evidence against $A \parr B$ is both evidence against A and evidence against B.
Universal Evidence against $\forall x{:}D.P(x)$ is a counterexample value $n \in D$ (*e.g.,* a concrete number when D is $\mathbb{N}$) along with evidence against $P(n)$.
Negation Evidence against $\neg A$ is the same as evidence for A.

Now we can see that the new symbols for conjunction and disjunction disambiguate between the positive and negative burdens of proof, which carry complementary forms of evidence. In contrast, the two quantifiers $\exists$ and $\forall$ are not duplicated, but rather arranged to prioritize "finite" evidence (one specific example or counter example in the domain) instead of "infinite" hypothetical evidence (a general algorithm for generating evidence based on any object in the domain). Furthermore, there are two different notions of negation, the positive $\ominus A$ and negative $\neg A$, internalizing the duality between evidence for and against. The construction of evidence for or against each connectives is captured by these inference rules with two judgments: A **true** directly verifies A's truth and A **false** directly refutes it.

$$
\frac{A\ \textbf{true} \quad B\ \textbf{true}}{A \otimes B\ \textbf{true}}
\qquad
\frac{A\ \textbf{true}}{A \oplus B\ \textbf{true}}
\qquad
\frac{B\ \textbf{true}}{A \oplus B\ \textbf{true}}
\qquad
\frac{n \in D \quad P(n)\ \textbf{true}}{\exists x{:}D.P(x)\ \textbf{true}}
\qquad
\frac{A\ \textbf{false}}{\ominus A\ \textbf{true}}
$$

$$
\frac{A\ \textbf{false} \quad B\ \textbf{false}}{A \parr B\ \textbf{false}}
\qquad
\frac{A\ \textbf{false}}{A \mathbin{\&} B\ \textbf{false}}
\qquad
\frac{B\ \textbf{false}}{A \mathbin{\&} B\ \textbf{false}}
\qquad
\frac{n \in D \quad P(n)\ \textbf{false}}{\forall x{:}D.P(x)\ \textbf{false}}
\qquad
\frac{A\ \textbf{true}}{\neg A\ \textbf{false}}
$$

What does the other party without the burden of proof do? While they can wait to rebut the specific evidence they are given, it may take a long time (perhaps forever) for that evidence to be constructed. And absence of evidence does not imply the evidence of absence. For example, the Skeptic may doubt a universal conjecture, but cannot come up with a counterexample that shows it false yet; this alone does not prove the conjecture true. Instead, in the face of negative burden of proof, the Sage can prove truth with a hypothetical argument that no such evidence against exists: systematically consider all possible evidence for the falsehood of A and show that each one leads to a contradiction. Dually, the Skeptic – waiting for the positive burden of proof to be fulfilled – can prove falsehood by hypothetically refuting all evidence of truth, showing all possible evidence for the truth of A leads to a contradiction. These proofs by contradiction are captured by the following inference rules for a proposition A (having positive burden of truth) and B (having negative burden of proof) using a third and final judgment **contra** representing a logical contradiction.

$$\dfrac{\begin{array}{c}\overline{A\ \mathbf{true}}\\ \vdots \\ \mathbf{contra}\end{array}}{A\ \mathbf{false}} \qquad \dfrac{\begin{array}{c}\overline{B\ \mathbf{false}}\\ \vdots \\ \mathbf{contra}\end{array}}{B\ \mathbf{true}}$$

We can now see that the evidence for $\neg A$'s truth hasn't changed from Section 2.1. To show $\neg A$ true via proof by contradiction, we assume evidence that $\neg A$ is false – the same as assuming evidence A is true – and derive a contradiction. In contrast, $\ominus A$ is entirely new.

2.3 The duality of constructive evidence

Viewing logic as a dialogue between an advocate and adversary – rather than just a lone advocate building constructions by themself – already improves the evidence of falsehood by giving the adversary a voice. Moreover, it improves some pleasant symmetries of truth with a more nuanced library of logical connectives expressing the full range of burden of proof.

For example, consider the classical law of double-negation elimination, $\neg\neg A \implies A$ (where $\implies$ stands for implication): if A cannot be untrue, then A is true. Intuitionists reject this law because the evidence for $\neg\neg A$ is much weaker than for A. For example, the evidence for $\neg\neg\exists x{:}\mathbb{N}.\exists y{:}\mathbb{N}.\ x^2 = y$ is a hypothetical argument that only says that it is contradictory for $\exists x{:}\mathbb{N}.\exists y{:}\mathbb{N}.\ x^2 = y$ to lead to a contradiction. In contrast, one example of direct evidence for $\exists x{:}\mathbb{N}.\exists y{:}\mathbb{N}.\ x^2 = y$ is the witness that for $x = 3$ and $y = 9$, we have $3^2 = 9$. One possible conclusion, taken by intuitionists, is that double-negation elimination is just incompatible with constructive evidence. But another conclusion is that the wrong negation has been used. Instead, consider the shape evidence for $\ominus\neg\exists x{:}\mathbb{N}.\exists y{:}\mathbb{N}.\ x^2 = y$ given by the more refined, dual definitions of $\ominus$ and $\neg$ in Section 2.2: evidence proving $\ominus\neg\exists x{:}\mathbb{N}.\exists y{:}\mathbb{N}.\ x^2 = y$ true consists of evidence proving $\neg\exists x{:}\mathbb{N}.\exists y{:}\mathbb{N}.\ x^2 = y$ false, which in turn is the same as just evidence proving $\exists x{:}\mathbb{N}.\exists y{:}\mathbb{N}.\ x^2 = y$ true. So while $\neg\neg A \implies A$ for a generic A might not be considered constructive, $\ominus\neg A \implies A$ definitively is.

More generally, we can look at how negation interacts with the other logical connectives. In classical logic, the de Morgan laws describe how negation distributes over dual connectives, converting between conjunction ($\wedge$) and disjunction ($\vee$) as well as existential ($\exists$) and universal ($\forall$) quantifiers, like so (where $\iff$ means "if and only if"):

$$\begin{aligned}
\neg(A \vee B) &\iff (\neg A) \wedge (\neg B) & \qquad \neg(\exists x{:}D.P(x)) &\iff \forall x{:}D.\neg P(x)\\
\neg(A \wedge B) &\iff (\neg A) \vee (\neg B) & \qquad \neg(\forall x{:}D.P(x)) &\iff \exists x{:}D.\neg P(x)
\end{aligned}$$

However, not all of these laws hold intuitionistically. In particular, $\neg(A \wedge B) \not\Longrightarrow (\neg A) \vee (\neg B)$ because knowing that the combination of A and B is contradictory is not enough to show definitively which of A or B is contradictory. Likewise, $\neg(\forall x{:}D.P(x)) \not\Longrightarrow \exists x{:}D.\neg P(x)$ because, as we have seen before, knowing that it is contradictory for $P(x)$ to be universally true does not point out the specific element of D where P fails.

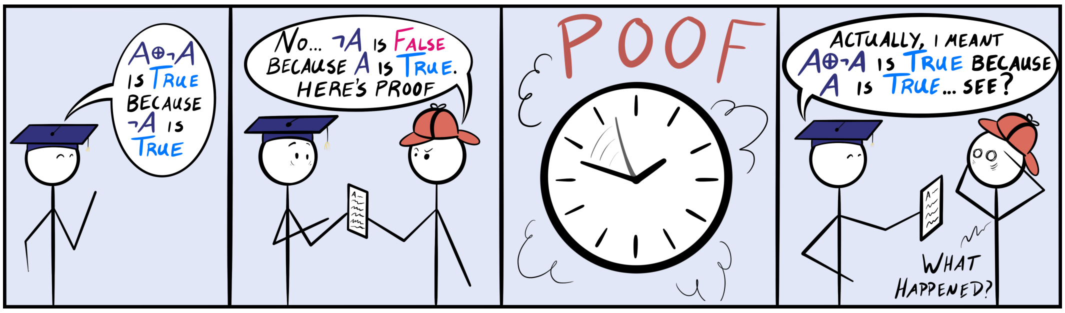


Figure 1 Law of excluded middle $A \oplus \neg A$ as a miraculous feat of time travel.

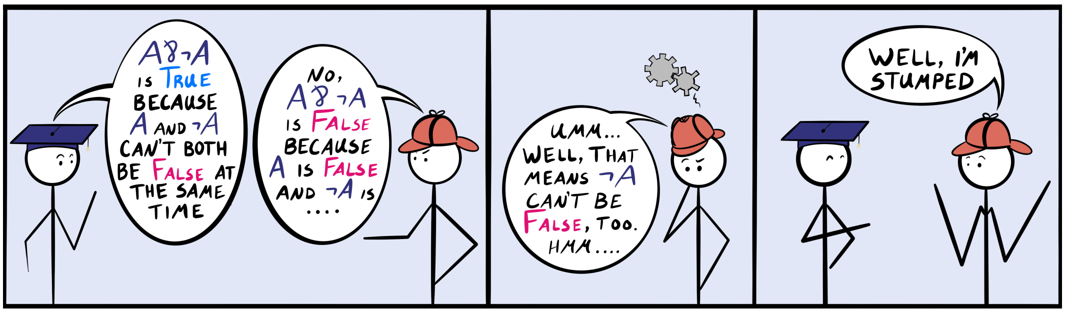


Figure 2 Law of excluded middle $A \parr \neg A$ as a mundane contradiction of falsehood.

Again, this problem with the asymmetry of the De Morgan laws can be seen as the classical logician being too vague about the burden of proof in their connectives. Rephrasing, we get the following symmetric versions of the De Morgan laws in terms of $\neg$ and $\ominus$ that are nonetheless constructive:

$$\begin{aligned} \neg(A \oplus B) &\iff (\neg A) \mathbin{\&} (\neg B) & \qquad \ominus(A \mathbin{\&} B) &\iff (\ominus A) \oplus (\ominus B) \\ \neg(A \otimes B) &\iff (\neg A) \parr (\neg B) & \ominus(A \parr B) &\iff (\ominus A) \otimes (\ominus B) \\ \neg(\exists x{:}D.P(x)) &\iff \forall x{:}D.\neg P(x) & \ominus(\forall x{:}D.P(x)) &\iff \exists x{:}D. \ominus P(x) \end{aligned}$$

Note the new meanings of the previously offensive directions. On the one hand, evidence for $\ominus(A \mathbin{\&} B)$ consists of evidence against $A \mathbin{\&} B$ that boils down to either evidence against A or evidence against B; exactly the same as the evidence for $(\ominus A) \oplus (\ominus B)$. On the other hand, evidence against $\neg(A \otimes B)$ is the same as evidence for $A \otimes B$ which consists of evidence for both A and B simultaneously; exactly the same as the evidence against $(\neg A) \parr (\neg B)$. Similarly, evidence for $\ominus(\forall x{:}D.P(x))$ is a specific counterexample n in D such that $P(n)$ is false, which is exactly the same evidence needed to prove $\exists x{:}D. \ominus P(x)$ true.

Finally, let's return to the troublesome law of the excluded middle, $A \vee \neg A$ that we started with. Now equipped with two different versions of disjunction, we can understand this law constructively in two very different ways. The first understanding is based on the connection of classical logic with control [23], which represents the excluded middle as the seemingly impossible choice $A \oplus \neg A$. This proposition is true through a cunning act of bait and switch as shown in Figure 1. First, the Sage (in the blue academic square cap) baselessly asserts that $\neg A$ is true hoping that this is ignored. Later the Skeptic (in the Sherlock Holmesian brown deerstalker) can call the Sage's bluff by providing evidence that A is in fact true. In response, the Sage miraculously turns back the clock and changes their claim, instead asserting that A is true by using the Skeptic's own evidence against them. Now, the use of time travel to change answers might seem a bit excessive, but luckily there is a much more mundane understanding based on the more modest $A \parr \neg A$. This proposition is true, almost trivially, as a basic contradiction shown in Figure 2, based on the fact that evidence for A is identical to evidence against $\neg A$. Here, the Sage merely asserts that A cannot be both true and false at the same time, to which the Skeptic has no retort. Thus, restoring the balance between true and false does a better job of explaining the constructive evidence of both classical and intuitionistic logic.

3 Computing with Duality

What does a calculus for writing logical dialogues look like? In order to prepare for representing hypothetical arguments, we will use a logical device called a *sequent* written:

$$A_1, A_2, \ldots, A_n \vdash B_1, B_2, \ldots, B_m$$

that groups together multiple propositions into a single package revolving around a central *entailment* denoted by the turnstyle ($\vdash$). This sequent can be read as "if A_1, A_2, ..., A_n are all true, then something among B_1, B_2, ..., B_m must be true," or more simply "the conjunction of the left $(A_1, \ldots, A_n)$ implies the disjunction of the right $(B_1, \ldots, B_m)$." In order to understand the practical meaning of the compound sequent, it can help to look at special cases where it contains at most one proposition, forcing either the left or the right side of entailment to be empty (denoted by $\bullet$).

True The sequent $\bullet \vdash A$ means that A is true. The assumption is trivial because the conjunction of nothing is true (asserting everything in an empty set passes some test is a vacuously true statement). Since A is the only option on the right, A must be true.

False The sequent $A \vdash \bullet$ means that A is false. The conclusion is impossible because the disjunction of nothing is false (asserting that a true element is found among an empty set is immediately false). Since assuming A is true implies falsehood, A must be false.

Contradiction The sequent $\bullet \vdash \bullet$ denotes a contradiction. Following the reasoning above, $\bullet \vdash \bullet$ means "true implies false," which is just plainly impossible.

Thus far, this is just rephrasing the basic judgments we had discussed in Section 2.2 (therein written A **true**, A **false**, and **contra**, respectively). What is more interesting is how these forms of logical judgments can be reinterpreted as analogous forms of expressions in a calculus for representing computation as interaction.

Production The typing judgment $\bullet \vdash v : A \mid$ means that the *term v produces* information of type A. By analogy with Section 2.2, v represents the Sage who is trying to prove that A is true, and the value returned by v represents the evidence (of type A) that verifies the veracity of their claim.

Consumption The typing judgment $\mid e : A \vdash \bullet$ means that the *coterm* (*a.k.a* continuation) e *consumes* information of type A. The coterm e is analogous to the Skeptic who is trying to prove that A is false. In this sense, the covalue returned by e represents the evidence of a counter argument (of type A), which refutes values of type A.

Computation The typing judgment $c : (\bullet \vdash \bullet)$ means that the *command* c is an *executable statement*. Commands are the computational unit of the language where all reductions happen; each step of reduction corresponds to the back-and-forth dialogue between the Sage and the Skeptic. The fundamental form of commands is an interaction $\langle v \| e \rangle$ between a term v and a coterm e. The command $\langle v \| e \rangle$ means that the value returned by v is given to e as input, or dually the covalue constructed by e inspects v's output.

Note that, whereas terms $\bullet \vdash v : A \mid$ produce output (*i.e.,* provide answers) and coterms $\mid e : A \vdash \bullet$ consume input (*i.e.,* ask questions), the command $c : (\bullet \vdash \bullet)$ does not produce or consume anything itself, and acts as an isolated computation. To interact with a command, it is necessary to provide for free *variables* x which stand for places to read inputs and free *covariables* α standing for places to send outputs. Open commands with free (co)variables have the more general typing judgment

$$c : (x_1 : A_1, x_2 : A_2, \dots, x_n : A_n \vdash \alpha_1 : B_1, \alpha_2 : B_2, \dots, \alpha_m : B_m)$$

As shorthand, we use Γ to denote a list of inputs $x_1 : A_1, \dots, x_n : A_n$ and Δ to denote a list of outputs $\alpha_1 : B_1, \dots, \alpha_m : B_m$. Similar to open commands of type $c : (\Gamma \vdash \Delta)$, we also have open terms $\Gamma \vdash v : A \mid \Delta$ and open coterms $\Gamma \mid e : A \vdash \Delta$ which might also use free (co)variables in Γ and Δ. Reference to these free (co)variables looks like this:[1]

$$\frac{}{\Gamma, x : A \vdash x : A \mid \Delta}\ VarR \qquad\qquad \frac{}{\Gamma \mid \alpha : A \vdash \alpha : A, \Delta}\ VarL$$

As another example, the typing rule for safe interactions in a command $\langle v \| e \rangle$ corresponds to the *Cut* rule, which only connects together a producer and consumer that agree on a shared type A of information being exchanged:

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \mid e : A \vdash \Delta}{\langle v \| e \rangle : (\Gamma \vdash \Delta)}\ Cut$$

The exciting part of this language is the way it renders the many dualities in logic directly in its syntax. We know that true is dual to false, and for the same reason things on the left of a sequent (*i.e.,* to the left of $\vdash$) are dual to things on the right. In this sense, the turnstyle $\vdash$ serves as an axis of duality in logic. The same axis exists in the form of commands $\langle v \| e \rangle$, where the left and right components are dual to one another. The most direct way to see this duality is in the exchange of answers and questions between the two sides of a command.

$$\begin{array}{c} \xRightarrow{\text{Answers}} \\ \langle v \| e \rangle \\ \xLeftarrow[\text{Questions}]{} \end{array}$$

However, there are many other dualities besides the answer-question dichotomy to explore along this same axis. While we imagine that information flows left-to-right, it turns out that control flows right-to-left. There is the construction-destruction dynamic between the creation of concrete evidence and the inspection of it, which can be arranged in either direction. Likewise, abstraction over types and hidden information gives rise to dual notions of generics (à la parametric polymorphism in functional languages and Java generics) which hide information in the consumer/client and modules (à la the SML module system) which hide information in the producer/server. So now let's consider how each of these computational dualities manifest themselves in the logical foundation of this language.

[1] The rules are named with an R and L because their conclusion below the horizontal line of inference introduces a new term on the *Right* of the turnstyle ($\vdash$) and a new coterm on the *Left*, respectively. This naming convention comes from the sequent calculus, which we will follow throughout the paper.

3.1 Positive burden of proof as data

In the constructive dialogues of Section 2.2, consider the case where the Sage has the positive burden of truth, and is responsible for constructing a concrete piece of evidence that backs up their claim that some proposition is true. The shape of the Sage's evidence depends on the proposition in question, and will contain enough information to fully justify truth in a way the Skeptic can examine. In computational terms, constructing this positive form of evidence corresponds to *constructing values* of a data type. In this sense, the Sage constructing evidence of A's truth is analogous to a producer v which constructs a value of type A.

For example, consider the basic cases for positive evidence of conjunction ($A \otimes B$) and disjunction ($A \oplus B$). The evidence of the conjunction $A \otimes B$ is made up of a combination of evidence v of A along with evidence w of B. In other words, it is a pair (v, w) of the tuple type $A \otimes B$. In contrast, the evidence of the disjunction $A \oplus B$ is a choice of either evidence v for A or evidence w for B. In other words, it is one of the two tagged values $\iota_1 v$ or $\iota_2 w$ of the sum type $A \oplus B$. These constructions are captured by the following typing rules, which resemble the inference rules for $A \otimes B$ **true** and $A \oplus B$ **true** in Section 2.2:

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \vdash w : B \mid \Delta}{\Gamma \vdash (v, w) : A \otimes B \mid \Delta} \otimes R \qquad \frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \vdash \iota_1 v : A \oplus B \mid \Delta} \oplus R_1 \qquad \frac{\Gamma \vdash w : B \mid \Delta}{\Gamma \vdash \iota_2 w : A \oplus B \mid \Delta} \oplus R_2$$

How, then, might the Skeptic respond to the evidence contained in these values? In general, the Skeptic is only obligated to show that evidence following these rules cannot be constructed, because their existence would lead to a contradiction. This corresponds to *pattern matching* or *deconstructing* on the shape of all possible values of a data type. A rebuttal of $A \otimes B$ is a process demonstrating a contradiction c given any generic pair $(x, y) : A \otimes B$, *i.e.,* in the context of two generic values $x : A$ and $y : B$. Similarly, a rebuttal of $A \oplus B$ is a process that demonstrates two different contradictions: c_1 which responds to a tagged value $\iota_1 x : A \oplus B$ (*i.e.,* in the context of a generic value $x : A$) and c_2 which responds to a tagged value $\iota_2 y : A \oplus B$ (*i.e.,* in the context of $y : B$). The two rebuttals are captured by the deconstructing consumers $\tilde{\mu}(x, y).c$ and $\tilde{\mu}[\iota_1 x.c_1 \mid \iota_2 y.c_2]$ given by these typing rules:

$$\frac{c : (\Gamma, x : A, y : B \vdash \Delta)}{\Gamma \mid \tilde{\mu}(x, y).c : A \otimes B \vdash \Delta} \otimes L \qquad \frac{c_1 : (\Gamma, x : A \vdash \Delta) \quad c_2 : (\Gamma, y : B \vdash \Delta)}{\Gamma \mid \tilde{\mu}[\iota_1 x.c_1 \mid \iota_2 y.c_2] : A \oplus B \vdash \Delta} \oplus L$$

Although more intricate, the evidence for or against an existential follows this same pattern of constructing values in the term and deconstructing them in the coterm. For simplicity, assume that the quantifiers' domain ranges over other types. $\exists X.B$ describes values of type B, which might reference a hidden type X. This kind of information hiding corresponds to modules in a program where the code implementing the module is written with full knowledge of a specific type X, but the client code using the module does not know which type was used for X. To be explicit about the module's hidden choice for X, we can use the (Sage's) constructor form (A, v) which means to produce the value v whose type depends on A. The client (Skeptic) side can unpack a generic value (evidence) of the form (X, y) to run a command (demonstrate a contradiction), which looks like $\tilde{\mu}(X, y).c$. This pair of construction-deconstruction looks like:[2]

[2] The $\exists L$ rule has the additional side condition $X \notin FV(\Gamma \vdash \Delta)$, meaning the type variable X is not found among the free variables of environments Γ and Δ. The side condition makes sure that X stands for a truly generic type parameter, which would be ruined if Γ and Δ constrained X with additional assumptions about it. Similar side conditions weren't needed in $\otimes L$ and $\oplus L$ because ordinary variables x, y cannot be referenced by types in Γ and Δ without dependent types. Alternatively, we could have also introduced yet another environment $\Theta = X, Y, Z, \ldots$ for keeping track of the free type variables in the sequent, as is often done in the type systems in polymorphic languages like System F [22].

$$\frac{\Gamma \vdash v : B\{A/x\} \mid \Delta}{\Gamma \vdash (A, v) : \exists X.B \mid \Delta}\ \exists R \qquad \frac{c : (\Gamma, y : B \vdash \Delta) \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma \mid \tilde{\mu}(X, y).c : \exists X.B \vdash \Delta}\ \exists L$$

3.2 Negative burden of proof as codata

If the positive burden of truth corresponds to constructing values of a data type, then what is the computational interpretation of the negative burden of proof? Applying syntactic duality of our symmetric calculus – that is, flipping the roles of producers v and consumers e in the command $\langle v \| e \rangle$ to get the analogue of $\langle e \| v \rangle$ – leaves us only one answer: *constructing covalues* of a codata type, which are defined in terms of observations rather than values. This corresponds to the evidence constructed by the Skeptic within a negative burden of proof, which has a different shape depending on the proposition A being argued against. Thus, the Skeptic's evidence can be represented by a consumer e of type A.

Consider the basic cases for negative evidence against conjunctions ($A \,\&\, B$) and disjunctions ($A \parr B$). Contrary to before, the evidence against a conjunction comes in one of two forms: either evidence e against A or evidence f against B. In other words, it is a first projection $\pi_1 e$ or second projection $\pi_2 f$ out of a product type $A \,\&\, B$. The evidence against a disjunction instead has just one form, containing both evidence e against A and evidence f against B. Taken together, this is a pair $[e, f]$ – dual to a tuple of values – of the type $A \parr B$. These constructions of consumers are captured by the following typing rules, which resemble the inference rules for $A \,\&\, B$ **false** and $A \parr B$ **false** from Section 2.2:

$$\frac{\Gamma \mid e : A \vdash \Delta}{\Gamma \mid \pi_1 e : A \,\&\, B \vdash \Delta}\ \&L_1 \qquad \frac{\Gamma \mid f : B \vdash \Delta}{\Gamma \mid \pi_2 f : A \,\&\, B \vdash \Delta}\ \&L_2 \qquad \frac{\Gamma \mid e : A \vdash \Delta \quad \Gamma \mid f : B \vdash \Delta}{\Gamma \mid [e, f] : A \parr B \vdash \Delta}\ \parr L$$

If the Skeptic is now constructing concrete evidence, then the Sage must be the one responding to it in some way. This proof of truth involves arguing that the Skeptic cannot possibly argue against the proposition: every potential piece of negative evidence that might be constructed leads to a contradiction. The computational interpretation of the Sage's response corresponds to an object that defines a reaction to every possible observation on it, which can be written via *copattern matching* [1] which *deconstructs* the shape of its observer.

A rebuttal in favor of $A \,\&\, B$ is a process that demonstrates two different contradictions: c_1 which responds to a generic first projection $\pi_1 \alpha : A \,\&\, B$, and c_2 which responds to a generic second projection $\pi_2 \beta : A \,\&\, B$. Instead, a rebuttal in favor of $A \parr B$ responds with just one contradiction c, given a generic $[\alpha, \beta] : A \parr B$ that combines both pieces of negative evidence (α against A and β against B). The two rebuttals in favor of $A \,\&\, B$ and $A \parr B$ are captured by the copattern-matching producers $\mu(\pi_1 \alpha.c_1 \mid \pi_2 \beta.c_2)$ and $\mu[\alpha, \beta].c$, respectively, given by these two typing rules:

$$\frac{c_1 : (\Gamma \vdash \alpha : A, \Delta) \quad c_2 : (\Gamma \vdash \beta : B, \Delta)}{\Gamma \vdash \mu(\pi_1 \alpha.c_1 \mid \pi_2 \beta.c_2) : A \,\&\, B \mid \Delta}\ \&R \qquad \frac{c : (\Gamma \vdash \alpha : A, \beta : B, \Delta)}{\Gamma \vdash \mu[\alpha, \beta].c : A \parr B \mid \Delta}\ \parr R$$

Universal quantification can be derived mechanically as the dual of existential quantification, where the roles of information hiding have been flipped between the implementor and client. With the polymorphic type $\forall X.B$ – describing values of type B that are generic in type X – it is now the clients using values of type $\forall X.B$ that get to choose X. For example, consider the polymorphic function $\forall X.X \to X$: the callers of this function get to choose the specific type for X – it could be integers, booleans, lists, *etc.*– before passing an argument of that type to receive a returned value of the same type. The implementor which produces a value of type $\forall X.B$ must instead be generic in X: it cannot know which X was chosen because different clients might all choose different specializations for X. Thus, the implementation (Sage) side can unpack a generic covalue (evidence) of the form $[X, \beta]$ to run a command (demonstrate a contradiction), which looks like $\mu[X, \beta].c$ corresponding to System F's $\Lambda X.v$ [22]. These (de)constructors follow rules dual to $\exists R$ and $\exists L$:

$$\frac{\Gamma \mid e : B\{A/X\} \vdash \Delta}{\Gamma \mid [A, e] : \forall X.B \vdash \Delta}\ \forall L \qquad \frac{c : (\Gamma \vdash \beta : B, \Delta) \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma \vdash \mu[A, \beta].c : \forall X.B \mid \Delta}\ \forall R$$

3.3 The two dual negations

Now that we have introduced the computational content of both the positive and negative burden of proof, we can finally examine the nature of negation which reverses these two roles. In Section 2.2, we had two different forms of negation: $\ominus A$ is described by positive evidence in favor of it, whereas $\neg A$ is described by negative evidence against it. Following our analogy, $\ominus A$ corresponds to a data type: the Sage's evidence in favor of $\ominus A$, written (e), contains *specific* evidence e against A. The Skeptic then responds by showing why *any* construction of the form $(\alpha) : \ominus A$ leads to a contradiction c, as expressed by these typing rules:

$$\frac{\Gamma \mid e : A \vdash \Delta}{\Gamma \vdash (e) : \ominus A \mid \Delta}\ \ominus R \qquad \frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \mid \tilde{\mu}(\alpha).c : \ominus A \vdash \Delta}\ \ominus L$$

The other negation $\neg A$ is its dual codata type: the Skeptic's evidence against $\neg A$, written $[v]$, contains *specific* evidence v in favor (*i.e.,* producing a value) of A. The Sage then responds by showing why *any* construction of the form $[x] : \neg A$ leads to a contradiction c, as in:

$$\frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \mid [v] : \neg A \vdash \Delta}\ \neg L \qquad \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \vdash \mu[x].c : \neg A \mid \Delta}\ \neg R$$

3.4 Proof by contradiction as control

We have talked about many different indirect proofs and (co)terms: those that show how potential constructions lead to a contradiction (*i.e.,* command), rather than giving a concrete construction itself. These include all the coterms which pattern-match on specific values of data types, as well as all the terms which copattern-match on the specific covalues of codata types. But in practical programming languages, we aren't forced to always match on the shape of a value. We can also just give any value a name, as in the expression $\mathbf{let}\, z = v\, \mathbf{in}\, w$ found in many functional languages. What does this look like in our symmetric language? We could generalize coterms like $\tilde{\mu}(x, y).c$ to just the generic $\tilde{\mu}z.c$ which names their input before running a command c (just like $\mathbf{let}\, z = v\, \mathbf{in}\, w$ names v before running w). The dual of the generic $\tilde{\mu}$ is a generic μ: the term $\mu\alpha.c$ names its output before running a command c.[3] The typing rules for these two dual abstractions correspond to the two forms of proof by contradiction from Section 2.2: if assuming A **true** leads to a contradiction, then A **false**; and dually if assuming A **false** leads to a contradiction, then A **true**.

$$\frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta}\ \mathit{ActL} \qquad \frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\ \mathit{ActR}$$

Notice how these two rules can be seen as simplifications of matching rules on the left ($\otimes L$, $\oplus L$, $\exists L$) and right ($\& R$, $⅋ R$, $\forall R$) to not depend on the structure of the abstracted type.

[3] The term $\mu\alpha.c$ gets the simpler name because it came first in Parigot's $\lambda\mu$-calculus [31] for classical logic. The dual coterm $\tilde{\mu}x.c$ was derived after in the sequent calculus [4] for call-by-value computation.

Although generic μ and $\tilde{\mu}$ might seem innocuous, they can have a serious impact on computational power. Whereas $\tilde{\mu}$ corresponds to the pervasive (and relatively innocent) feature of value-naming as expressed by basic let-bindings, μ corresponds to a notion of *control effect* equivalent to Scheme's `call/cc` operator [7]. In terms of a logic, μ can also increase the propositions that can be proven true.

For example, consider the two different interpretations of the law of the excluded middle from Section 2.3. The negative version, $A \parr \neg A$ corresponds to the term $\mu[\alpha, [x]].\langle x \| \alpha \rangle$ written in terms of nested copatterns. Intuitively, this term is isomorphic to the identity function, $\lambda x.x : A \to A$, and it's typing derivation (*i.e.,* proof) is given like so:

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{$VarR$}
\UnaryInfC{$x : A \vdash x : A \mid \alpha : A, \beta : \neg A$}
\AxiomC{}
\RightLabel{$VarL$}
\UnaryInfC{$x : A \mid \alpha : A \vdash \alpha : A, \beta : \neg A$}
\RightLabel{Cut}
\BinaryInfC{$\langle x \| \alpha \rangle : (x : A \vdash \alpha : A, \beta : \neg A)$}
\RightLabel{$\neg R$}
\UnaryInfC{$\vdash \mu[x].\langle x \| \alpha \rangle : \neg A \mid \alpha : A, \beta : \neg A$}
\AxiomC{}
\RightLabel{$VarL$}
\UnaryInfC{$\mid \beta : \neg A \vdash \alpha : A, \beta : \neg A$}
\RightLabel{Cut}
\BinaryInfC{$\langle \mu[x].\langle x \| \alpha \rangle \| \beta \rangle : (\vdash \alpha : A, \beta : \neg A)$}
\RightLabel{$\parr R$}
\UnaryInfC{$\vdash \mu[\alpha, \beta].\langle \mu[x].\langle x \| \alpha \rangle \| \beta \rangle : A \parr \neg A \mid$}
\end{prooftree}
$$

Notice how – in addition to the core Cut and Var rules – we only use the type-specific matching rules for $\parr$ and $\neg$ here. There is no need to resort to the generic $ActR$ or $ActL$.

In contrast, the positive law of the excluded middle, $A \oplus \neg A$, corresponds to the term $\mu\alpha.\langle \iota_2 \mu[x].\langle \iota_1 x \| \alpha \rangle \| \alpha \rangle$. Notice the use of the generic $\mu\alpha \ldots$, requiring the $ActR$ rule in its typing derivation (omitting the names for Var and Cut rules):

$$
\begin{prooftree}
\AxiomC{}
\UnaryInfC{$x : A \vdash x : A \mid \alpha : A \oplus \neg A$}
\RightLabel{$\oplus R_1$}
\UnaryInfC{$x : A \vdash \iota_1 x : A \oplus \neg A \mid \alpha : A \oplus \neg A$}
\AxiomC{}
\UnaryInfC{$x : A \mid \alpha : A \oplus \neg A \vdash \alpha : A \oplus \neg A$}
\BinaryInfC{$\langle \iota_1 x \| \alpha \rangle : (x : A \vdash \alpha : A \oplus \neg A)$}
\RightLabel{$\neg R$}
\UnaryInfC{$\vdash \mu[x].\langle \iota_1 x \| \alpha \rangle : \neg A \mid \alpha : A \oplus \neg A$}
\RightLabel{$\oplus R_2$}
\UnaryInfC{$\vdash \iota_2 \mu[x].\langle \iota_1 x \| \alpha \rangle : A \oplus \neg A \mid \alpha : A \oplus \neg A$}
\AxiomC{}
\UnaryInfC{$\mid \alpha : A \oplus \neg A \vdash \alpha : A \oplus \neg A$}
\BinaryInfC{$\langle \iota_2 \mu[x].\langle \iota_1 x \| \alpha \rangle \| \alpha \rangle : (\vdash \alpha : A \oplus \neg A)$}
\RightLabel{$ActR$}
\UnaryInfC{$\vdash \mu\alpha.\langle \iota_2 \mu[x].\langle \iota_1 x \| \alpha \rangle \| \alpha \rangle : A \oplus \neg A \mid$}
\end{prooftree}
$$

Whereas $A \parr \neg A$ is like the simple identity function, the term of type $A \oplus \neg A$ invokes a serious manipulation of control flow. Intuitively, this term corresponds to the Scheme expression:

```
(call/cc (lambda (alpha)
           (cons 2 (lambda (x) (alpha (cons 1 x))))))
```

Here, the "time travel" needed to implement the positive law of the excluded middle is expressed by the control operator `call/cc`. Before doing anything else, the current continuation is saved (in `alpha`), just in case we need to change our answer. Then, we first return the second option (represented by a numerically-labeled cons-cell `(cons 2 ...)`) containing a function. If that function is ever called with a value `x` of type A, then we invoke the continuation `alpha` which rolls back the clock and lets us change our answer to the first option `(cons 1 x)`: deftly giving back the value we were just given.

3.5 A symmetric system of computation

Thus far, we have only discussed how to build objects (producers and consumers) following this two-sided method of interaction. That alone does not tell us how to compute; we also need to know how the interaction unfolds over time.

$$
\begin{array}{llllr}
(\beta_{\otimes}) & \langle (v,w) \| \tilde{\mu}(x,y).c \rangle = \langle v \| \tilde{\mu}x.\langle w \| \tilde{\mu}y.c \rangle \rangle & (\eta_{\otimes}) & \tilde{\mu}(x,y).\langle (x,y) \| \alpha \rangle = \alpha & (\alpha : A \otimes B) \\
(\beta_{\oplus}) & \langle \iota_i v \| \tilde{\mu}[\iota_i x_i.c_i] \rangle = \langle v \| \tilde{\mu}x_i.c_i \rangle & (\eta_{\oplus}) & \tilde{\mu}[\iota_i x_i.\langle \iota_i x_i \| \alpha \rangle] = \alpha & (\alpha : A \oplus B) \\
(\beta_{\exists}) & \langle (A,v) \| \tilde{\mu}(X,y).c \rangle = \langle v \| \tilde{\mu}y.c\{A/X\} \rangle & (\eta_{\exists}) & \tilde{\mu}(X,y).\langle (X,y) \| \alpha \rangle = \alpha & (\alpha : \exists X.B) \\
(\beta_{\ominus}) & \langle (e) \| \tilde{\mu}(\alpha).c \rangle = \langle \mu\alpha.c \| e \rangle & (\eta_{\ominus}) & \tilde{\mu}(\beta).\langle (\beta) \| \alpha \rangle = \alpha & (\alpha : \ominus A) \\
(\beta_{\&}) & \langle \mu(\pi_i \alpha_i.c_i) \| \pi_i e \rangle = \langle \mu\alpha_i.c_i \| e \rangle & (\eta_{\&}) & \mu(\pi_i\alpha_i.\langle x \| \pi_i \alpha_i \rangle) = x & (x : A \,\&\, B) \\
(\beta_{⅋}) & \langle \mu[\alpha,\beta].c \| [e,f] \rangle = \langle \mu\alpha.\langle \mu\beta.c \| f \rangle \| e \rangle & (\eta_{⅋}) & \mu[\alpha,\beta].\langle x \| [\alpha,\beta] \rangle = x & (x : A ⅋ B) \\
(\beta_{\forall}) & \langle \mu[X,\beta].c \| [A,e] \rangle = \langle \mu\beta.c\{A/x\} \| e \rangle & (\eta_{\forall}) & \mu[X,\beta].\langle x \| [X,\beta] \rangle = x & (x : \forall X.B) \\
(\beta_{\neg}) & \langle \mu[x].c \| [v] \rangle = \langle v \| \tilde{\mu}x.c \rangle & (\eta_{\neg}) & \mu[y].\langle x \| [y] \rangle = x & (x : \neg A)
\end{array}
$$

Plus compatibility, symmetry, reflexivity, and transitivity.

Figure 3 Equational reasoning for (co)pattern matching in the dual core sequent calculus.

One of the simplest ways of viewing the computation of interaction is through the axioms which characterize the equality of expressions. These axioms, given in Figure 3, come in two main forms. The β family of laws say what happens when a matching term and coterm of a type meet up in a command. For example, when the tuple construction (v, w) meets up with a tuple deconstruction $\tilde{\mu}(x, y).c$, the interaction can be simplified with $\beta_{\otimes}$ by matching the structure of (v, w) with the pattern (x, y), and bind v to x and w to y (with the help of the generic $\tilde{\mu}$). When there is a choice like in the sum type $A \oplus B$, then the appropriate response is selected by $\beta_{\oplus}$. When the right construction $\iota_2 v$ meets up with the sum deconstruction $\tilde{\mu}[\iota_1 x.c_1 \mid \iota_2 y.c_2]$, then the result is c_2 with v bound to y from the matching pattern $\iota_2 y$. The same kind of matching happens for the codata types, but with the roles reversed. Instead, it is the coterm side that is constructed, like the second projection $\pi_2 e$ of a product type $A \,\&\, B$, and the term side selects a response, like the term $\mu(\pi_1\alpha.c_1 \mid \pi_2\beta.c_2)$ which matches with $\pi_2 e$ by binding e to β and running c_2 as per $\beta_{\&}$. Note that the β rules for both negations ($\ominus$ and $\neg$) end up swapping the two sides of a command.

The other family of laws are the η axioms, which give us a notion of *extensionality*. In each case, the η axioms say that deconstructing a structure and reconstructing it exactly as it was before does nothing. The side where this simplification applies depends on the type of the structure in question. For data types, the consumer does the deconstructing, so the $\eta_{\otimes}$, $\eta_{\oplus}$, $\eta_{\exists}$, and $\eta_{\ominus}$ axioms apply to a generic unknown coterm – represented by the covariable α – waiting to receive its input. Whereas for codata types, the producer does the deconstructing, so the $\eta_{\&}$, $\eta_{⅋}$, $\eta_{\forall}$, and $\eta_{\neg}$ axioms apply to a generic unknown term – represented by the variable x – waiting to receive an output request.

But equational axioms are quite far from a real implementation in a machine. They give the ultimate freedom of choice on where the rules can apply (in any context, due to compatibility) and in which direction (due to symmetry). In reality, a machine implementation will make a (deterministic) choice on the next step to take, and always move forward. This is modeled by the operational semantics given in Figure 4, where each step $c \mapsto c'$ applies *exactly* to the top of the command itself. The happy coincidence of a dual calculus based on the sequent calculus is that its operational semantics *is* an abstract machine [10], since there is never a search for the next redex which is always found at the top. Thus, this style of calculus is a good framework for studying the low-level details of computation needed to implement languages in real machines.

Call-by-value definition of values (V_+) and covalues (E_+):

$$
\begin{aligned}
\mathit{Value}_+ \ni V_+, W_+ &::= x \mid (V_+, W_+) \mid \iota_1 V_+ \mid \iota_2 V_+ \mid (A, V_+) \mid (E_+) \\
&\mid \mu(\pi_1 \alpha.c_1 \mid \pi_2 \beta.c_2) \mid \mu[\alpha, \beta].c \mid \mu[X, \beta].c \mid \mu[x].c \\
\mathit{CoValue}_+ \ni E_+, F_+ &::= e
\end{aligned}
$$

Call-by-name definition of values (V_-) and covalues (E_-):

$$
\begin{aligned}
\mathit{Value}_- \ni V_-, W_- &::= v \\
\mathit{CoValue}_- \ni E_-, F_- &::= \alpha \mid [E_-, F_-] \mid \pi_1 E_- \mid \pi_2 E_- \mid [A, E_-] \mid [V_-] \\
&\mid \tilde{\mu}[\iota_1 x.c_1 \mid \iota_2 y.c_2] \mid \tilde{\mu}[x, y].c \mid \tilde{\mu}(X, y).c \mid \tilde{\mu}(\alpha).c
\end{aligned}
$$

Reduction rules for call-by-value ($s = +$) and call-by-name ($s = -$) evaluation.

$$
\begin{array}{lrcl}
(\beta^s_\otimes) & \langle (V_s, W_s) \| \tilde{\mu}(x, y).c \rangle & \mapsto & c\{V_s/x, W_s/y\} \\
(\beta^s_\oplus) & \langle \iota_i V_s \| \tilde{\mu}[\iota_i x_i.c_i] \rangle & \mapsto & c_i\{V_s/x_i\} \\
(\beta^s_\exists) & \langle (A, V_s) \| \tilde{\mu}(X, y).c \rangle & \mapsto & c\{A/X, V_s/y\} \\
(\beta^s_\ominus) & \langle (E_s) \| \tilde{\mu}(\alpha).c \rangle & \mapsto & c\{E_s/\alpha\} \\
(\beta^s_\&) & \langle \mu(\pi_i \alpha_i.c_i) \| \pi_i E_s \rangle & \mapsto & c_i\{E_s/\alpha_i\} \\
(\beta^s_⅋) & \langle \mu[\alpha, \beta].c \| [E_s, F_s] \rangle & \mapsto & c\{E_s/\alpha, F_s/\beta\} \\
(\beta^s_\forall) & \langle \mu[X, \beta].c \| [A, E_s] \rangle & \mapsto & c\{A/x, E_s/\beta\} \\
(\beta^s_\neg) & \langle \mu[x].c \| [V_s] \rangle & \mapsto & c\{V_s/x\}
\end{array}
\qquad
\begin{array}{lrcl}
(\varsigma^s_\otimes) & \langle (v, w) \| E_s \rangle & \mapsto & \langle v \| \tilde{\mu}x.\langle w \| \tilde{\mu}y.\langle (x, y) \| E_s \rangle \rangle \rangle \\
(\varsigma^s_\oplus) & \langle \iota_i v \| E_s \rangle & \mapsto & \langle v \| \tilde{\mu}x.\langle \iota_i x \| E_s \rangle \rangle \\
(\varsigma^s_\exists) & \langle (A, v) \| E_s \rangle & \mapsto & \langle v \| \tilde{\mu}x.\langle (A, x) \| E_s \rangle \rangle \\
(\varsigma^s_\ominus) & \langle (e) \| E_s \rangle & \mapsto & \langle \mu\alpha.\langle (\alpha) \| E_s \rangle \| e \rangle \\
(\varsigma^s_\&) & \langle V_s \| \pi_i e \rangle & \mapsto & \langle \mu\alpha.\langle V_s \| \pi_i \alpha \rangle \| e \rangle \\
(\varsigma^s_⅋) & \langle V_s \| [e, f] \rangle & \mapsto & \langle \mu\alpha.\langle \mu\beta.\langle V_s \| [\alpha, \beta] \rangle \| f \rangle \| e \rangle \\
(\varsigma^s_\forall) & \langle V_s \| [A, e] \rangle & \mapsto & \langle \mu\alpha.\langle V_s \| [A, \alpha] \rangle \| e \rangle \\
(\varsigma^s_\neg) & \langle V_s \| [v] \rangle & \mapsto & \langle v \| \tilde{\mu}x.\langle V_s \| [x] \rangle \rangle
\end{array}
$$

In each of the ς^s rules, assume that (v, w), $\iota_i v$, (A, v), and (e) are not in Value_s, respectively, and $\pi_i e$, $[e, f]$, $[A, e]$, and $[v]$ are not in $\mathit{CoValue}_s$, respectively.

Figure 4 Operational semantics for (co)pattern matching in the dual core sequent calculus.

The difference between the β rules in the operational semantics (Figure 4) from the ones in the equational theory (Figure 3) is that the operational rules completely resolve the matching in one step. Rather than forming new bindings with generic μs and $\tilde{\mu}$s, the components of the construction (on either side) are substituted directly for the (co)pattern variables. To do so, we need to use a notion of *evaluation* strategy which informs us which terms can be substituted for variables (we call these *values*) and which coterms can be substituted for covariables (we call these *covalues*, which represent *evaluation contexts*).

Call-by-value evaluation simplifies terms first before substituting them for variables, so it has a quite restrictive notion of value (V_+) for constructed values like (V_+, W_+) and $\iota_i V_+$, but all coterms represent call-by-value evaluation contexts (hence every e is substitutable). Dually, call-by-name evaluation will substitute any term for a variable (hence a value V_- could be any v), but only certain coterms represent evaluation contexts: for example, the projection $\pi_1 E_-$ only represents an evaluation context because E_- does, but $\pi_1 e$ does not when e does not need its input yet.

The other cases of reduction are handled by the ς rules, which say what to do when a construction isn't a (co)value yet. In a call-by-value language like OCaml, the term `(1+2, 3+4)` first evaluates the two components before returning the pair value `(3, 7)`. This scenario is handled by the $\varsigma^+_\otimes$ step, which lifts the two computations in the tuple to the top of the command, replacing $\langle (1+2, 3+4) \| \alpha \rangle$ with $\langle 1+2 \| \tilde{\mu}x.\langle 3+4 \| \tilde{\mu}y.\langle (x, y) \| \alpha \rangle \rangle \rangle$; now we know that the next step is to simplify $1+2$ before binding it to x.

Equational axioms for $\mu\tilde{\mu}$ in both call-by-value ($s = +$) and call-by-name ($s = -$) reduction:

$$\begin{array}{llllll}
(\beta^s_\mu) & \langle \mu\alpha.c \| E_s\rangle = c\{E_s/\alpha\} & (\eta_\mu) & \mu\alpha\langle v\|\alpha\rangle = v & (\alpha \notin FV(v)) \\
(\beta^s_{\tilde{\mu}}) & \langle V_s \| \tilde{\mu}x.c\rangle = c\{V_s/x\} & (\eta_{\tilde{\mu}}) & \tilde{\mu}x.\langle x\|e\rangle = e & (x \notin FV(e))
\end{array}$$

Operational semantics for $\mu\tilde{\mu}$ in both call-by-value ($s = +$) and call-by-name ($s = -$):

$$(\beta^s_\mu) \quad \langle \mu\alpha.c \| E_s\rangle \mapsto c\{E_s/\alpha\} \qquad (\beta^s_{\tilde{\mu}}) \quad \langle V_s \| \tilde{\mu}x.c\rangle \mapsto c\{V_s/x\}$$

Figure 5 Rules for data flow and control flow in the dual core sequent calculus.

$$\begin{array}{llll}
\textbf{data } A \oplus B \textbf{ where} & \textbf{data } A \otimes B \textbf{ where} & \textbf{data } \ominus A \textbf{ where} & \textbf{data } \exists F \textbf{ where} \\
\quad \iota_1 : A \vdash A \oplus B \mid & \quad (_,_) : A, B \vdash A \otimes B \mid & \quad (_) : \vdash \ominus A \mid A & \quad (_,_) : F\ A \vdash \exists F \mid \\
\quad \iota_2 : B \vdash A \oplus B \mid & & & \\
\textbf{codata } A \,\&\, B \textbf{ where} & \textbf{codata } A \parr B \textbf{ where} & \textbf{codata } \neg A \textbf{ where} & \textbf{codata } \forall F \textbf{ where} \\
\quad \pi_1 : \mid A \,\&\, B \vdash A & \quad [_,_] : \mid A \parr B \vdash A, B & \quad [_] : A \mid \neg A \vdash & \quad [_,_] : \mid \forall F \vdash F\ A \\
\quad \pi_2 : \mid A \,\&\, B \vdash B & & &
\end{array}$$

Figure 6 (Co)Data declarations of the core connectives and quantifiers.

The last piece of the puzzle is what to do with the generic μs and $\tilde{\mu}$s. Fortunately, these are simpler than the individual rules for the various connectives and quantifiers. A coterm $\tilde{\mu}x.c$ binds its partner to x wholesale, without inspecting it further, and likewise $\mu\alpha.c$ binds its entire partner to α. These two actions are captured by $\beta^s_{\tilde{\mu}}$ and β^s_μ in Figure 5 which, like the rules in Figure 4, are careful to only substitute values and covalues. This careful consideration of substitutability prevents the fundamental critical pair between μ and $\tilde{\mu}$:

$$c_1\{\tilde{\mu}x.c_2/\alpha\} \leftarrow\!\!\shortmid_{\beta^+_\mu} \langle \mu\alpha.c_1 \| \tilde{\mu}x.c_2\rangle \mapsto_{\beta^-_{\tilde{\mu}}} c_2\{\mu\alpha.c_1/x\}$$

This restriction is necessary for both the equational axioms as well as the operational reduction steps (which are identical in name and result). These restrictions ensure that the operational semantics is *deterministic* and the equational theory is *consistent* (*i.e.,* not all commands are equated). Similarly, the η axioms for μ and $\tilde{\mu}$ say that binding a (co)variable just to use it immediately does nothing. While the η laws in Figures 3 and 5 are not themselves necessary for computation, they do give us a hint on how to keep going when we might get stuck. Specifically, the ς rules from Figure 4 can be derived from $\beta\eta$ equality, showing that these two families of axioms are *complete* for specifying computation [8].

▶ **Observation 2.** *If $c \mapsto_{\beta\varsigma} c'$ then $c =_{\beta\eta} c'$.*

3.6 (Co)Data in the wild

The connectives from Sections 3.1 and 3.2 originally arose from the field of logic, but that doesn't mean they are disconnected from programming. Indeed, the concept of data and codata they embody can be found to some degree in programming languages that are already in wide use today, although not in their full generality.

First, we can imagine a mechanism for declaring new connectives as (co)data types which list their patterns of construction. For example, all the connectives we have seen so far are given declarations in Figure 6. Each (term or coterm) constructor is given a type signature in the form of a sequent: input parameters are to the left of $\vdash$, and output parameters are to the right. For data types, constructors build a value returned as output, whose type is given in a special position to the right of the turnstyle between it and the vertical bar (*i.e.,* the A in $\dots \vdash A \mid \dots$). Dually for codata types, constructors build a covalue that takes an input, whose type is given in the special position on the left between the turnstyle and the vertical bar (*i.e.,* the A in $\dots \mid A \vdash \dots$).

This notion of data type corresponds to *algebraic data types* in typed functional languages. For example, the declarations for $A \oplus B$ and $A \otimes B$ correspond to the following Haskell declarations for sum (`Either`) and pair (`Both`) types:

```
data Either a b where
  Left  :: a -> Either a b
  Right :: b -> Either a b
```

```
data Both a b where
  Pair :: a -> b -> Both a b
```

Even the existential quantifier corresponds to a Haskell data type, whose constructor introduces a new generic type variable `a` not found in the return type `Exists f`.

```
data Exists f where Pack :: f a -> Exists f
```

However, the negation $\ominus A$ does not correspond to any data type in Haskell. That's because $\ominus A$'s constructor requires *two outputs* (notice the two types to the right of the turnstyle: the main $\ominus A$ plus the additional output parameter A). This requires some form of continuations or control effects, which is not available in a pure functional language like Haskell.

The dual notion of codata type corresponds to *interfaces* in typed object-oriented languages. For example, the declaration for $A \,\&\, B$ corresponds to the following Java interface for a generic `Product`:

```
interface Product<A,B> { A first(); B second(); }
```

Java's type system is not strong enough to capture quantifiers.[4] However, if its type system were extended so that generic types could range over other parameterized generic types, we could declare a `Forall` interface corresponding to the $\forall$ quantifier:

```
interface Forall<F> { F<A> specialize<A>(); }
```

Unfortunately, the types $A \parr B$ and $\neg A$ suffer the same fate as $\ominus A$; their constructors require a number of outputs different from 1: $[\alpha, \beta]$ has two outputs (both α and β), and $[x]$ has no outputs (x is an input, not an output). So they cannot be represented in Java without added support for juggling multiple continuations.

The possibilities for modeling additional information in the constructions of the type – representing *pre-* and *post-conditions* in a program – become much more interesting when we look at indexed (co)data types. For a long time, functional languages have been using *generalized algebraic data types* (GADTs), also known as *indexed data types*, that allow each constructor return a value with a more constrained version of that type. The classic example

[4] Unlike Haskell, Java does not support generic type variables with higher kinds. The Haskell declaration of `Exists f` relies on the fact that the type variable `f` has the kind `* -> *`, *i.e.,* `f` stands for a *function* that turns one type into another.

of indexed data types is representing expression trees with additional typing information. For example, here is a data type representing a simple expression language with literal values, plus and minus operations on numbers, an "is zero" test, and an if-then-else expression:

$$
\begin{array}{l}
\mathbf{data}\,\mathtt{Expr}\,X\ \mathbf{where} \\
\quad \begin{array}{rr}
\mathtt{Literal} : & X \vdash \mathtt{Expr}\,X \quad \mid \\
\mathtt{Plus} : & \mathtt{Expr}\,\mathtt{Int},\ \mathtt{Expr}\,\mathtt{Int} \vdash \mathtt{Expr}\,\mathtt{Int} \ \mid \\
\mathtt{Minus} : & \mathtt{Expr}\,\mathtt{Int},\ \mathtt{Expr}\,\mathtt{Int} \vdash \mathtt{Expr}\,\mathtt{Int} \ \mid \\
\mathtt{IsZero} : & \mathtt{Expr}\,\mathtt{Int} \vdash \mathtt{Expr}\,\mathtt{Bool} \mid \\
\mathtt{IfThenElse} : & \mathtt{Expr}\,\mathtt{Bool},\ \mathtt{Expr}\,X,\ \mathtt{Expr}\,X \vdash \mathtt{Expr}\,X \quad \mid
\end{array}
\end{array}
$$

The type parameter X acts as an index, and it lets us constrain the types of values an expression can represent. For example, `IsZero` expects an integer and returns a boolean. This lets us write a typed evaluation function $\mathit{eval} : \mathtt{Expr}\,X \to X$, and not worry about mistyped edge cases because the type system rules out poorly-constructed expressions.

The dual of indexed data types are *indexed codata types*, which let us constrain each observation of the codata type to only accept certain inputs which model another form of pre- and post-conditions [18]. For example, we can embed a basic socket protocol – for sending and receiving information along an address – inside this indexed codata type:

$$
\begin{array}{l}
\mathbf{codata}\,\mathtt{Socket}\,X\ \mathbf{where} \\
\quad \begin{array}{rl}
\mathtt{Bind} : \mathtt{String} & \mid \mathtt{Socket}\,\mathtt{Raw} \vdash \mathtt{Socket}\,\mathtt{Bound} \\
\mathtt{Connect} : & \mid \mathtt{Socket}\,\mathtt{Bound} \vdash \mathtt{Socket}\,\mathtt{Live} \\
\mathtt{Send} : \mathtt{String} & \mid \mathtt{Socket}\,\mathtt{Live} \vdash () \\
\mathtt{Receive} : & \mid \mathtt{Socket}\,\mathtt{Live} \vdash \mathtt{String} \\
\mathtt{Close} : & \mid \mathtt{Socket}\,\mathtt{Live} \vdash ()
\end{array}
\end{array}
$$

A new `Socket` starts out as `Raw`. We can `Bind` a `Socket Raw` to an address, after which it is `Bound` and can be `Connect`ed to make it `Live`. A `Socket Live` represents a connection we can use to `Send` and `Receive` messages, and is discarded by a `Close`.[5]

4 Applications of Duality

So a constructive view of symmetric classical logic gives us a dual language for expressing computation as interaction. Does this form of duality have any application in the broader scope of programs? Yes! Let's look at a few examples where computational duality can be put into action for solving problems in programming.

4.1 Functions as Codata

There is a delicate trilemma in the theory of the untyped λ-calculus: one cannot combine non-strict weak-head reduction, function extensionality, and computational effects. The specific reduction we are referring to follows two properties: "non-strict" means that functions are called without evaluating their arguments first, and "weak-head" means that evaluation stops at a λ-abstraction. Function extensionality is captured by the η law – $\lambda x.\ f\ x = f$ – from the foundation of the λ-calculus. And finally effects could be anything – from mutable

[5] This interface can be further improved by linear types, which ensure that outdated states of the `Socket` cannot be used, and forces the programmer to properly `Close` a `Socket` instead of leaving it hanging.

state to exceptions – but for our purposes, non-termination introduced by general recursion is enough. That is to say, the infinite loop $\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$ already expressible in the "pure" untyped λ-calculus counts as an effect.

So what is the problem when all three are combined in the same calculus? The conflict arises when we observe a λ-abstraction as the final result of evaluation. Because of weak-head reduction, any λ-abstraction counts as a final result, including $\lambda x.\Omega x$. Because of extensionality, the η law says that $\lambda x.\Omega x$ is equivalent to Ω. Taken together, this means that a program that ends immediately is the same as one that loops forever: an inconsistency.

4.1.1 Efficient head reduction

One way to fix the trilemma is to change from weak-head reduction to *head reduction.* With head reduction, evaluation no longer stops at a λ-abstraction. Instead, head reduction looks inside of λs to keep going until a head-normal form of the shape $\lambda x_1 \ldots \lambda x_n.x_i\ M_1 \ldots M_m$ is found. But going inside λs means that evaluation has to deal with open terms, *i.e.,* terms with free variables in them. How can we perform head reduction efficiently, when virtually all efficient implementations assume that evaluation only handles closed terms?

Our idea is to look at functions as yet another form of *codata*, just like $A \parr B$ and $A \,\&\, B$. Following the other declarations in Figure 6, the type of functions can be defined as:

$$\textbf{codata}\, A \to B\, \textbf{where}\ \_ \cdot \_ : A \mid A \to B \vdash B$$

This says that the coterm which observes a function of type $A \to B$ has the form of a *call stack* $v \cdot e$, where v is the argument (of type A), and e represents a kind of "return pointer" (expecting the returned B). The stack-like nature can be seen in the way a chain of function arrows requires a stack of arguments; for instance a coterm of type $Int \to Int \to Int \to Int$ has the stack shape $1 \cdot 2 \cdot 3 \cdot \alpha$, where α is a place to put the result.

Rather than the usual λ-abstraction, the codata view suggests that we can instead write functions in terms of copattern matching: $\mu[x \cdot \beta].c$ is a function of type $A \to B$ where c is the command to run in the scope of the (co)variables $x : A$ and $\beta : B$. Both forms of writing functions are equivalent to one another (via general μ):

$$\mu[x \cdot \beta].c = \lambda x.\mu\beta.c \qquad\qquad \lambda x.v = \mu[x \cdot \beta].\langle v \| \beta\rangle \qquad\qquad (\beta \notin FV(v))$$

This way, the main rule for reducing a call-by-name function call is to match on the structure of a call stack (recall from Section 3.5 that call-by-name covalues are restricted to E_-, so covalue call stacks have the form $v \cdot E_-$ in call-by-name) like so:

$$\langle \mu[x \cdot \beta].c \| v \cdot E_-\rangle \mapsto c\{v/x, E_-/\beta\}$$

But what happens when we encounter a function at the top-level? This is represented by the command $\langle \mu[x \cdot \beta].c \| \mathtt{tp}\rangle$ where $\mathtt{tp}$ is a constant standing in for the empty, top-level context. Normally, we would be stuck, so instead lets look at functions from the other side. A call stack $v \cdot E_-$ is similar to a pair (v, w). In some programming languages, we access a pair by matching on its structure (analogous to $\tilde{\mu}(x, y).c$). But in other languages, we are given primitive projections for accessing its fields. We can make the same change with functions: rather than matching on the structure of a call (with $\mu[x \cdot \beta].c$ or $\lambda x.v$), we can instead *project out of a call stack* [26]. The projection $\mathtt{arg}[v \cdot E_-]$ gives us the argument v and $\mathtt{ret}[v \cdot E_-]$ gives us the return pointer E_-. These two projections let us keep going when a function reaches the top level, by projecting the argument and return pointer out of $\mathtt{tp}$:

$$\langle \mu[x \cdot \beta].c \| \mathtt{tp}\rangle \mapsto c\{\mathtt{arg}\,\mathtt{tp}\,/x, \mathtt{ret}\,\mathtt{tp}\,/\beta\}$$

This goes "inside" the function, and yet there are no free variables in sight. Instead, the would-be free x is replaced with the placeholder $\texttt{arg}\,\texttt{tp}$, and we get a *new* "top-level" context $\texttt{ret}\,\texttt{tp}$, which stands for the context expecting the result of an implicit call with $\texttt{arg}\,\texttt{tp}$.

As we keep going, we may return another function to $\texttt{ret}\,\texttt{tp}$, and the process continues with the new placeholder argument $\texttt{arg}[\texttt{ret}\,\texttt{tp}]$ and the next top-level $\texttt{ret}[\texttt{ret}\,\texttt{tp}]$. Rewriting these rules in terms of the more familiar λ-abstractions, we get the following small abstract machine for *closed* head reduction, which says what to do when a function is called (with $w \cdot E_-$) or returned to any of the series of "top-level" contexts ($\texttt{ret}^n\,E_-$):

$$\begin{aligned}
\langle v\ w \| E_- \rangle &\mapsto \langle v \| w \cdot E_- \rangle \\
\langle \lambda x.v \| w \cdot E_- \rangle &\mapsto \langle v\{w/x\} \| E_- \rangle \\
\langle \lambda x.v \| \texttt{ret}^n\,\texttt{tp} \rangle &\mapsto \langle v\{\texttt{arg}[\texttt{ret}^n\,\texttt{tp}]/x\} \| \texttt{ret}^{n+1}\,\texttt{tp} \rangle
\end{aligned}$$

For example, the η-expansion of the infinite loop Ω also loops forever, instead of stopping:

$$\langle \lambda x.\Omega x \| \texttt{tp} \rangle \mapsto \langle \Omega\ (\texttt{arg}\,\texttt{tp}) \| \texttt{ret}\,\texttt{tp} \rangle \mapsto \langle \Omega \| \texttt{arg}\,\texttt{tp} \cdot \texttt{ret}\,\texttt{tp} \rangle \mapsto\!\!\!\!\to \langle \Omega \| \texttt{arg}\,\texttt{tp} \cdot \texttt{ret}\,\texttt{tp} \rangle \dots$$

4.1.2 Effective confluence

A similar issue arises when we consider confluence of the reduction theory. In particular, the call-by-name version of η for functions can be expressed as simplifying the deconstruction-reconstruction detour $\mu[x \cdot \beta].\langle v \| x \cdot \beta \rangle \to_{\eta^-_\to} v$, similar to Figure 3.[6] We might expect that $\beta\eta$ reduction is now confluent like it is in the λ-calculus. Unfortunately, it is not, due to a critical pair between function extensionality and a general μ (_ stands for an unused (co)variable):[7]

$$\mu\_.c \leftarrow_{\eta^-_\to} \mu[x \cdot \beta].\langle \mu\_.c \| x \cdot \beta \rangle \to_{\beta^-_\mu} \mu[x \cdot \beta].c$$

Can we restore confluence of function extensionality in the face of control effects? Yes! The key to eliminating this critical pair is to replace the $\eta^-_\to$ rule with an alternative extensionality rule provided by viewing functions as codata types, equipped with projections out of their constructed call stacks. Under this view, every function is equivalent to a $\mu\alpha.c$, where $\texttt{arg}\,\alpha$ replaces the argument, and $\texttt{ret}\,\alpha$ replaces the return continuation. Written as a reduction that replaces copatterns with projections, we have:

$$\mu[x \cdot \beta].c \to_{\mu_\to} \mu\alpha.c\{\texttt{arg}\,\alpha/x, \texttt{ret}\,\alpha/\beta\}$$

Analogously, the $\mu_\to$ rule can be understood in terms of the ordinary λ-abstraction as $\lambda x.v \to \mu\alpha.\langle v\{\texttt{arg}\,\alpha/x\} \| \texttt{ret}\,\alpha \rangle$. If all functions immediately reduce to a general μ, then how can we execute function calls? The steps of separating the argument and the result are done by the rules for projection, which have their own form of β-reduction along with a different extensionality rule $surj_\to$ capturing the *surjective pair* property of call stacks:

$$\texttt{arg}[v \cdot E_-] \to_{\beta_{\texttt{arg}}} v \qquad \texttt{ret}[v \cdot E_-] \to_{\beta_{\texttt{ret}}} E_- \qquad [\texttt{arg}\,E_-] \cdot [\texttt{ret}\,E_-] \to_{surj_\to} E_-$$

The advantage of these rules is that they are confluent in the presence control effects [27]. Even though surjective pairs can cause non-confluence troubles in general [28], the coarse distinction between terms and coterms is enough to resolve the problem for call-by-name call stacks. Moreover, these rules are strong enough to simulate the λ-calculus' $\beta\eta$ laws:

---

[6] This is the call-by-name version of $\mu[x \cdot \beta].\langle y \| x \cdot \beta \rangle \to_{\eta_\to} y$ because we have substituted a call-by-name value $v \in \mathit{Value}_-$ for the variable y.

[7] Note, this is not just a problem with copatterns; the same issue arises in Parigot's $\lambda\mu$-calculus with ordinary λ-abstractions and η law: $\mu\_.c \leftarrow \lambda x.(\mu\_.c)\ x \to \lambda x.\mu\_.c$.

$$(\lambda x.v)\ w = \mu\alpha.\langle \mu[x \cdot \beta].\langle v \| \beta \rangle \| w \cdot \alpha \rangle \to_{\mu_\to} \mu\alpha.\langle \mu\gamma.\langle v\{\texttt{arg}\,\gamma/x\} \| \texttt{ret}\,\gamma \rangle \| w \cdot \alpha \rangle$$

$$\to_{\beta_\mu^-} \mu\alpha.\langle v\{\texttt{arg}[w \cdot \alpha]/x\} \| \texttt{ret}[w \cdot \alpha] \rangle \twoheadrightarrow_{\beta_{\texttt{ret}}\beta_{\texttt{arg}}} \mu\alpha.\langle v\{w/x\} \| \alpha \rangle \to_{\eta_\mu} v\{w/x\}$$

$$\lambda x.(v\ x) = \mu[x \cdot \beta].\langle v \| x \cdot \beta \rangle \to_{\mu_\to} \mu\alpha.\langle v \| [\texttt{arg}\,\alpha] \cdot [\texttt{ret}\,\alpha] \rangle \to_{surj_\to} \mu\alpha.\langle v \| \alpha \rangle \to_{\eta_\mu} v$$

4.2 Loops in Types, Programs, and Proofs

Thus far, we've only talked about *finite* types of information: putting together a fixed number of things. However, real programs are full of loops. Many useful types are self-referential, letting them model information whose size is bounded but arbitrarily large (like lists and trees), or whose size is completely unbounded (like infinite streams). Programs using these types need to be able to loop over arbitrarily large data sets, and generate infinite objects in streams. Once those loops are introduced, reasoning about programs becomes much harder. Let's look at how duality can help us understand the least understood loops in types, programs, and proofs.

4.2.1 (Co)Recursion

Lists and trees – which cover structures that could be any size, as long as they're finite – are modeled by the familiar concept of *inductive data types* found in all mainstream, typed functional programming languages. The dual of these are *coinductive codata types*, which is a relatively newer feature that is finding its way into more practical languages for programming and proving. We already saw instances of both of these as `Expr` and `Socket` from Section 3.6. The canonical examples of (co)inductive (co)data are the types for natural numbers and infinite streams, which are defined like so:

$$\begin{array}{ll} \textbf{data}\,\texttt{Nat}\,\textbf{where} & \textbf{codata}\,\texttt{Stream}\,X\,\textbf{where} \\ \quad \texttt{Zero} : \qquad \vdash \texttt{Nat} \mid & \quad \texttt{Head} : \mid \texttt{Stream}\,X \vdash X \\ \quad \texttt{Succ} : \texttt{Nat} \vdash \texttt{Nat} \mid & \quad \texttt{Tail} : \mid \texttt{Stream}\,X \vdash \texttt{Stream}\,X \end{array}$$

The recursive nature of these two types are in the fact that they have constructors that take parameters of the type being declared: `Succ` takes a `Nat` as input before building a new `Nat`, whereas `Tail` consumes a $\texttt{Stream}\,X$ to produce a new $\texttt{Stream}\,X$ as output.

To program with inductive types, functional languages allow programmers to write recursive functions that match on the structure of its argument. For example, here is a definition of the addition function *plus*:

$$plus\ \texttt{Zero}\ x = x \qquad\qquad plus\ (\texttt{Succ}\,y)\ x = plus\ y\ (\texttt{Succ}\,x)$$

We know that this function is well-founded – that is, it always terminates on any input – because it's structurally recursive: the first argument shown in red shrinks on each recursive call, where $\texttt{Succ}\,y$ is replaced with the smaller y. The second argument in blue doesn't matter; it can grow from x to $\texttt{Succ}\,x$ since we already have a termination condition.

Translating this example into the dual language reveals that the same notion of structural recursion covers both induction *and* coinduction [16]. Instead of defining *plus* as matching on just its arguments, we can define it as matching on the structure of its entire call stack α in the command $\langle plus \| \alpha \rangle$. Generalizing to the entire call stack lets us write coinductive definitions using the same technique. For example, here is the definition of *plus* in the dual language alongside *count* which corecursively produces a stream of numbers from a given starting point (*i.e.,* $count\ x = x, x+1, x+2, x+3, \dots$):

$$\langle plus \| \mathtt{Zero} \cdot x \cdot \alpha \rangle = \langle x \| \alpha \rangle \qquad \langle plus \| \mathtt{Succ}\, y \cdot x \cdot \alpha \rangle = \langle plus \| y \cdot \mathtt{Succ}\, x \cdot \alpha \rangle$$
$$\langle count \| x \cdot \mathtt{Head}\, \alpha \rangle = \langle x \| \alpha \rangle \qquad \langle count \| x \cdot \mathtt{Tail}\, \alpha \rangle = \langle count \| \mathtt{Succ}\, x \cdot \alpha \rangle$$

Both definitions are well-founded because they are structurally recursive, but the difference is the structure they are focused on within the call stack. Whereas the *value* $\mathtt{Succ}\, y$ shrinks to y in the recursive call to *plus*, it's the *covalue* $\mathtt{Tail}\, \alpha$ that shrinks to α in the corecursive call to *count*. In both, the growth in blue doesn't matter, since the red always shrinks.

Here are two more streams defined by structural recursion on the shape of the stream projection $\mathtt{Head}\, \alpha$ or $\mathtt{Tail}\, \alpha$. *iterate* repeats the same function over and over on some starting value (*i.e.*, $iterate\ f\ x = x, f\ x, f(f\ x), f(f(f\ x)), \dots$) and *maps* modifies an infinite stream by applying a function to every element (*i.e.*, $maps\ f\ (x_1, x_2, x_3 \dots) = f\ x_1, f\ x_2, f\ x_3, \dots$):

$$\langle iterate \| f \cdot x \cdot \mathtt{Head}\, \alpha \rangle = \langle f \| x \cdot \alpha \rangle$$
$$\langle iterate \| f \cdot x \cdot \mathtt{Tail}\, \alpha \rangle = \langle iterate \| f \cdot \mu\beta. \langle f \| x \cdot \beta \rangle \cdot \alpha \rangle$$
$$\langle maps \| f \cdot xs \cdot \mathtt{Head}\, \alpha \rangle = \langle f \| \mu\beta. \langle xs \| \mathtt{Head}\, \beta \rangle \cdot \alpha \rangle$$
$$\langle maps \| f \cdot xs \cdot \mathtt{Tail}\, \alpha \rangle = \langle maps \| f \cdot \mu\beta. \langle xs \| \mathtt{Tail}\, \beta \rangle \cdot \alpha \rangle$$

4.2.2 (Co)Induction

Examining the structure of (co)values isn't just good for programming; it's good for proving, too. For example, if we want to prove some property Φ about values of type $A \oplus B$, it's enough to show it holds for the (exhaustive) cases of $\iota_1 x_1 : A \oplus B$ and $\iota_1 x_2 : A \oplus B$ like so:

$$\frac{\Phi(\iota_1 x_1) : (\Gamma, x_1 : A \vdash \Delta) \quad \Phi(\iota_2 x_2) : (\Gamma, x_2 : B \vdash \Delta)}{\Phi(x) : (\Gamma, x : A \oplus B \vdash \Delta)} \oplus \mathit{Induction}$$

Exhaustiveness is key to ensure that all cases are covered and no possible value was left out. This becomes difficult to do directly for recursive types like $\mathtt{Nat}$, because it represents an infinite number of cases $(0, 1, 2, 3, \dots)$. Instead, we can prove a property Φ indirectly through the familiar notion of *structural induction*: prove $\Phi(\mathtt{Zero})$ specifically and prove that the inductive hypothesis $\Phi(y)$ implies $\Phi(\mathtt{Succ}\, y)$ as expressed by this inference rule

$$\frac{\Phi(\mathtt{Zero}) : (\Gamma \vdash \Delta) \quad \begin{array}{c} \dfrac{}{\Phi(y) : (\Gamma, y : \mathtt{Nat} \vdash \Delta)} IH \\ \vdots \\ \Phi(\mathtt{Succ}\, y) : (\Gamma, y : \mathtt{Nat} \vdash \Delta) \end{array}}{\Phi(x) : (\Gamma, x : \mathtt{Nat} \vdash \Delta)} \mathtt{Nat}\mathit{Induction}$$

But how can we deal with coinductive codata types? There are also an infinite number of cases to consider, but the values don't follow the same, predictable patterns. Here is a conventional but questionable form of coinduction that takes the entire goal $\Phi(x)$ to be the coinductive hypothesis, as in:

$$\frac{\begin{array}{c} \dfrac{}{\Phi(x) : (\Gamma, x : \mathtt{Stream}\, A \vdash \Delta)} \mathit{CoIH} \\ \vdots\ ??? \\ \Phi(x) : (\Gamma, x : \mathtt{Stream}\, A \vdash \Delta) \end{array}}{\Phi(x) : (\Gamma, x : \mathtt{Stream}\, A \vdash \Delta)} \mathit{Questionable\ CoInduction}$$

But this rule obviously has serious problems: $CoIH$ could just be used immediately, leading to a viciously circular proof. To combat this clear flaw, other secondary, external checks and guards have to be put into place that go beyond the rule itself, and instead analyze the *context* in which $CoIH$ is used to prevent circular proofs. As a result, a prover can build a coinductive proof that follows all the rules, but run into a nasty surprise in the end when the proof is rejected because it fails some implicit guard. Can we do better?

Just like the way structural induction looks at the shape of values, structural coinduction looks at the shape of covalues which represent *contexts* [12]. For example, here is the coinductive rule dual to $\oplus$*Induction* for concluding that a property Φ holds for any output of $A \,\&\, B$ by checking the (exhaustive) cases $\pi_1\alpha_1 : A \,\&\, B$ and $\pi_2\alpha_2 : A \,\&\, B$:

$$\frac{\Phi(\pi_1\alpha_1) : (\Gamma \vdash \alpha_1 : A, \Delta) \quad \Phi(\pi_2\alpha_2) : (\Gamma \vdash \alpha_2 : A, \Delta)}{\Phi(\alpha) : (\Gamma \vdash \alpha : A \,\&\, B, \Delta)} \,\&\mathit{CoInduction}$$

Just like `Nat`, streams have too many cases ($\texttt{Head}\,\beta, \texttt{Tail}[\texttt{Head}\,\beta], \texttt{Tail}[\texttt{Tail}[\texttt{Head}\,\beta]], \dots$) to exhaustively check directly. So instead, here is the dual form of proof as `Nat`*Induction* for proving Φ for any observation α of type $\texttt{Stream}\,A$: it proves the base case $\Phi(\texttt{Head}\,\beta)$ directly, and then shows that the *coinductive hypothesis* $\Phi(\gamma)$ implies the next step $\Phi(\texttt{Tail}\,\gamma)$, like so:

$$\frac{\Phi(\texttt{Head}\,\beta) : (\Gamma \vdash \beta : A, \Delta) \quad \begin{array}{c}\dfrac{}{\Phi(\gamma) : (\Gamma \vdash \gamma : \texttt{Stream}\,A, \Delta)}\,\mathit{CoIH} \\ \vdots \\ \Phi(\texttt{Tail}\,\gamma) : (\Gamma \vdash \gamma : \texttt{Stream}\,A, \Delta)\end{array}}{\Phi(\alpha) : (\Gamma \vdash \alpha : \texttt{Stream}\,A, \Delta)}\,\texttt{Stream}\mathit{CoInduction}$$

Notice the similarities between this rule and the one for `Nat` induction. In the latter, even though the inductive hypothesis $\Phi(y)$ is assumed for a generic y, then there is no need for external checks because we are forced to provide $\Phi(\texttt{Succ}\,y)$ *for the very same* y. The information flow between the introduction of y in IH and its use in the final conclusion of $\Phi(\texttt{Succ}\,y)$ prevents viciously circular proofs. In the same way, the coinductive rule here assumes $\Phi(\gamma)$ for a generic γ, but we are forced to prove $\Phi(\texttt{Tail}\,\gamma)$ *for the very same* γ. In this case, there is an implicit control flow between the introduction of γ in $CoIH$ and its use in the final conclusion $\Phi(\texttt{Tail}\,\gamma)$. Thus, $CoIH$ can be used in any place it fits, without any secondary guards or checks after the proof is built; `Stream`*CoInduction* is sound as-is.

How can this form of coinduction be used to reason about corecursive programs? Consider this interaction between *maps* and *iterate*: *maps* f (*iterate* f x) = *iterate* f (f x). Written in the dual language, this property translates to an equality between commands: $\langle \mathit{maps} \| f \cdot \mu\beta.\langle \mathit{iterate} \| f \cdot x \cdot \beta\rangle \cdot \alpha\rangle = \langle \mathit{iterate} \| f \cdot \mu\beta.\langle f \| x \cdot \beta\rangle \cdot \alpha\rangle$. We can prove this property (for any starting value x) using coinduction with these two cases:

$\alpha = \texttt{Head}\,\alpha'$. The base case follows by direct calculation with the definitions.

$$\begin{aligned}
\langle \mathit{maps} \| f \cdot \mu\beta.\langle \mathit{iterate} \| f \cdot x \cdot \beta\rangle \cdot \texttt{Head}\,\alpha'\rangle &= \langle f \| \mu\beta.\langle \mathit{iterate} \| f \cdot x \cdot \texttt{Head}\,\beta\rangle \cdot \alpha'\rangle && (\mathit{maps}, \beta_\mu) \\
&= \langle f \| \mu\beta.\langle f \| x \cdot \beta\rangle \cdot \alpha'\rangle && (\mathit{iterate}) \\
&= \langle \mathit{iterate} \| f \cdot \mu\beta.\langle f \| x \cdot \beta\rangle \cdot \texttt{Head}\,\alpha'\rangle && (\mathit{iterate})
\end{aligned}$$

$\alpha = \texttt{Tail}\,\alpha'$. First, assume the coinductive hypothesis ($CoIH$) which is generic in the value of the initial x: for all x, $\langle \mathit{maps} \| f \cdot \mu\beta.\langle \mathit{iterate} \| f \cdot x \cdot \beta\rangle \cdot \alpha'\rangle = \langle \mathit{iterate} \| f \cdot \mu\beta.\langle f \| x \cdot \beta\rangle \cdot \alpha'\rangle$. The two sides are equated by applying $CoIH$ with an updated value for x:

$$
\begin{aligned}
&\langle \mathit{maps} \| f \cdot \mu\beta.\langle \mathit{iterate} \| f \cdot x \cdot \beta\rangle \cdot \mathtt{Tail}\,\alpha'\rangle \\
&= \langle \mathit{maps} \| f \cdot \mu\beta.\langle \mathit{iterate} \| f \cdot x \cdot \mathtt{Tail}\,\beta\rangle \cdot \alpha'\rangle && (\mathit{maps}, \beta_\mu) \\
&= \langle \mathit{maps} \| f \cdot \mu\beta.\langle \mathit{iterate} \| f \cdot \mu\gamma.\langle f \| x \cdot \gamma\rangle \cdot \beta\rangle \cdot \alpha'\rangle && (\mathit{iterate}) \\
&= \langle \mathit{iterate} \| f \cdot \mu\beta.\langle f \| \mu\gamma.\langle f \| x \cdot \gamma\rangle \cdot \beta\rangle \cdot \alpha'\rangle && (\mathit{CoIH}\{\mu\gamma.\langle f \| x \cdot \gamma\rangle / x\}) \\
&= \langle \mathit{iterate} \| f \cdot \mu\gamma.\langle f \| x \cdot \gamma\rangle \cdot \mathtt{Tail}\,\alpha'\rangle && (\mathit{iterate})
\end{aligned}
$$

4.3 Compilation and Intermediate Languages

In Section 3, we saw how a symmetric language based on the sequent calculus closely resembles the structure of an abstract machine, which helps to reveal the details of how programs are really implemented. This resemblance raises the question: does a language based on the sequent calculus be a good *intermediate language* (IL) used to compile programs to machine code? The λ-calculus' syntax structure buries the most relevant part of an expression. For example, applying f to four arguments is written as $((((f\ 1)\ 2)\ 3)\ 4)$; we are forced to search for the next step – $f\ 1$ – found at the bottom of the tree. Instead, the syntax of the dual calculus raises up the next step of a program to the top; the same application is written as $\langle f \| 1 \cdot (2 \cdot (3 \cdot (4 \cdot \alpha)))\rangle$, where calling f with 1 is the first part of the command.

We have found that the sequent calculus can in fact be used as an intermediate language of a compiler [17]. The feature of bringing out the most relevant expression to the top of a program is shared by other commonly-used representations like *continuation-passing style* (CPS) [2] and *static single assignment* (SSA) [5]. However, the sequent calculus is uniquely flexible. Unlike SSA which is an inherently imperative representation, the sequent calculus is a good fit for both purely functional and effectful languages. And unlike CPS, the sequent calculus preserves enough of the original structure of the program to enable high-level rewrite rules expressed in terms of the source, as done by the Glasgow Haskell Compiler (GHC). Besides these advantages, our experience with a sequent calculus IL has led the following new techniques, which apply more broadly to other compiler ILs, too.

4.3.1 Join points in control flow

Join points are places where separate lines of control flow come back together. They are as pervasive as the branching structures in a program. For example, the statement

```
if x > 100: print "x is large"
else:       print "x is small"
print "goodbye"
```

splits off in two different directions to print a different statement depending on the value of `x`. But in either case, both branches of control flow will rejoin at the shared third line to print `"goodbye"`. Compilers need to represent these join points for code generation and optimization, in a way that is efficient in both time and space. Ideally, we want to generate code to jump to the join point in as few instructions as possible. And it's not acceptable to copy the common code into each branch; this leads to a space inefficiency that can cause an exponential blowup in the size of the generated code.

In the past, GHC represented these join points as ordinary functions bound by a let-expression. For example, the function j in $\mathbf{let}\, j\ x = \ldots x \ldots \mathbf{in\ if}\, z\, \mathbf{then}\, j\ 10\, \mathbf{else}\, j\ 20$ serves as the join point for both branches of the if-expression. Of course, this is space efficient, since it avoids duplicating code of j. But a full-fledged function call is much less efficient than a simple jump. Fortunately, the local function j has some special properties: it is always used in tail-call position and never escapes the scope of the **let**. These properties let GHC compile the calls $j\ 10$ and $j\ 20$ as efficient jumps. Unfortunately, the necessary properties for optimization aren't stable under *other* useful optimizations. For example, it usually helps to push (strict) evaluation contexts inside of an if-then-else or case-expression. While semantically correct, this can break the tail-call property of join points like here:

$$\begin{array}{l} 3 + \mathbf{let}\, j\ y = 10 + (y + y) \\ \quad\ \ \mathbf{in\ case}\, x\, \mathbf{of} \\ \quad\quad \iota_1 z_1 \to j\ z_1 \\ \quad\quad \iota_2 z_2 \to j\ (-z_2) \end{array} \quad \to \quad \begin{array}{l} \mathbf{let}\, j\ y = 10 + (y + y) \\ \mathbf{in\ case}\, x\, \mathbf{of} \\ \quad \iota_1 z_1 \to 3 + (j\ z_1) \\ \quad \iota_2 z_2 \to 3 + (j\ (-z_2)) \end{array}$$

Before, j could be compiled as a join point, but after it is used in non-tail-call positions $3 + (j\ z_1)$ and $3 + (j\ (-z_2))$. To combat this issue, we developed a λ-calculus with purely-functional join points [29]. While this calculus ostensibly contains labels and jumps – which are indeed compiled to jumps into assembly code – from the outside there is no observable effect. Instead, this calculus gives rules for optimizing around join points while ensuring they are still compiled efficiently. The example above is rewritten like so, where the context $3 + \square$ is now pushed into the code of the join point, rather than inside of the case-expression:

$$\begin{array}{l} 3 + \mathbf{join}\, j\ y = 10 + (y + y) \\ \quad\ \ \mathbf{in\ case}\, x\, \mathbf{of} \\ \quad\quad \iota_1 z_1 \to \mathbf{jump}\, j\ z_1 \\ \quad\quad \iota_2 z_2 \to \mathbf{jump}\, j\ (-z_2) \end{array} \to \begin{array}{l} \mathbf{join}\, j\ y = 3 + 10 + (y + y) \\ \mathbf{in\ case}\, x\, \mathbf{of} \\ \quad \iota_1 z_1 \to \mathbf{jump}\, j\ z_1 \\ \quad \iota_2 z_2 \to \mathbf{jump}\, j\ (-z_2) \end{array} \to \begin{array}{l} \mathbf{join}\, j\ y = 13 + (y + y) \\ \mathbf{in\ case}\, x\, \mathbf{of} \\ \quad \iota_1 z_1 \to \mathbf{jump}\, j\ z_1 \\ \quad \iota_2 z_2 \to \mathbf{jump}\, j\ (-z_2) \end{array}$$

Besides preserving the efficiency of j itself, this new form of code movement enables new optimizations. In this case, we can perform some additional constant folding of $3 + 10$, and other optimizations such as loop fusion can be expressed in this way as well.

4.3.2 Polarized primitive types

Another key feature found in the duality of logic is the *polarization* of different propositions. In terms of computation [33, 30], polarization is the combination of an "ideal" evaluation strategy based on the structure of types. Consider the η laws expressing extensionality of the various types in Figure 3. All the η laws for data types (*e.g.,* built with $\otimes$, $\oplus$, $\ominus$, and $\exists$) are about expanding covalues α. These laws are the strongest in the call-by-value strategy, which maximizes the number of covalues. Dually, the η laws for codata types (*e.g.,* built with $\&$, $⅋$, $\neg$, and $\forall$) are about expanding values x. These are the strongest in call-by-name.

Usually, we think of picking *one* evaluation strategy for a language. But this means that in either case, we are necessarily weakening extensionality of data or codata types (or both, if we choose something other than call-by-value or call-by-name). Instead, we can use a *polarized* language which improves η laws for *all* types by combining both strategies. This involves separating types into two different camps – the positive Type_+ and the negative Type_- – following our analogy of the burden of proof from Section 2.2 like so:

$$\begin{aligned} \mathit{Sign} \ni \quad & s ::= + \mid - \\ \mathit{Type}_+ \ni A_+, B_+ & ::= X_+ \mid A_+ \oplus B_+ \mid A_+ \otimes B_+ \mid \exists X_s.A_+ \mid \ominus A_- \mid {\downarrow} A_- \\ \mathit{Type}_- \ni A_-, B_- & ::= X_- \mid A_- \,\&\, B_- \mid A_- ⅋ B_- \mid \forall X_s.A_- \mid \neg A_+ \mid {\uparrow} A_+ \end{aligned}$$

By separating types in two, we also have to add the *polarity shifts* ${\downarrow} A_-$ and ${\uparrow} A_+$ so they can still refer to one another. For example, the plain $A \oplus (B \,\&\, C)$ becomes $A_+ \oplus {\downarrow}(B_- \,\&\, C_-)$.

Once this separation of types has occurred, we can bring them back together and intermingle both within a single language. The distinction can be made explicit in a refined *Cut* rule, which is the only rule which creates computation, so that the type (and its sign) becomes part of the program:

$$\frac{\Gamma \vdash v : A \mid \Delta \quad A : s \quad \Gamma \mid e : A \vdash \Delta}{\langle v|A{:}s|e\rangle : (\Gamma \vdash \Delta)}\ \mathit{Cut}$$

Since there is no longer one global evaluation strategy, we instead use *types* to determine the order. The additional annotation in commands let us drive computation with more nuance, referring to the sign s of the command to determine the priorities of μ and $\tilde{\mu}$ computations:

$$(\beta^s_\mu) \qquad \langle \mu\alpha.c|A{:}s|E_s\rangle = c\{E_s/\alpha\} \qquad\qquad (\beta^s_{\tilde{\mu}}) \qquad \langle V_s|A{:}s|\tilde{\mu}x.c\rangle = c\{V_s/x\}$$

The advantage of this more nuanced form of computation is that the types of the language express the nice properties that usually only hold up in an idealized, pure theory; however, now they hold up in the pragmatic practice that combines all manner of computational effects like control flow, state, and general recursion. For example, we might think that curried and uncurried functions – $A \to (B \to C)$ versus $(A \otimes B) \to C$ – are exactly the same. In both Haskell and OCaml, they are not, due to interactions with non-termination or side effects. But in a polarized language, they are the same, even with side effects.

These ideal properties of polarized types let us encode a vast array of user-defined data and codata types into a small number of basic primitives. We can choose a perfectly symmetric basis of connectives found in Section 3 [11] or an asymmetric alternative that is suited for purely functional programs [9]. The ideal properties provided by polarization can be understood in terms of the dualities of evidence in Section 2.3. For example, the equivalence between the propositions $\ominus\neg A$ and A corresponds to an *isomorphism* between the polarized types $\ominus\neg A_+$ and A_+ (and dually $\neg \ominus A_-$ and A_-). Intuitively, the only (closed) values of type $\ominus\neg A$ have exactly the form $([V_v])$, which is in bijection with the plain values V_v. And coterms of those two types are also in bijection due to the optimized η laws. All the de Morgan equivalences in Section 2.3 correspond to type isomorphisms, too. For example, the only (closed) values of $\ominus\forall X_s.B_-$ have the form $([A_s, E_-])$, which is in bijection with $\exists X_s. \ominus B_-$'s (closed) values of the form $(A_s, (E_-))$. In contrast, the other negation $\neg{\downarrow}\forall X_s.B_-$ includes abstract values of the form $\mu[x].c$, which are *not* isomorphic to the more concrete values $(A_s, \mu[x].c)$ of $\exists X_s.\neg{\downarrow}B_-$ that witness their chosen A_s. Thus, constructivity, computation, and full de Morgan symmetry depend on both polarized negations.

Polarization itself only accounts for call-by-value and call-by-name evaluation. However, other evaluation strategies are sometimes used in practice for pragmatic reasons. For example, implementations of Haskell use call-by-need evaluation, which can lead to better asymptotic performance than call-by-name. How do other evaluation strategies fit? We can add additional signs – besides $-$ and $+$ – that stand in for other strategies like call-by-need. But do we need to duplicate the basic primitives? No! We only need additional shifts that convert between the new sign(s) with the original $+$ and $-$, four in total:

$$\begin{array}{ll}
\mathbf{data}\ {\downarrow^s}(X : s) : +\ \mathbf{where} & \mathbf{data}\ {\Uparrow^s}(X : +) : s\ \mathbf{where} \\
\quad \mathtt{Box}_s : \ X : s \ \vdash\ {\downarrow^s}X : + \ \mid & \quad \mathtt{Return}_s : \ X : + \ \vdash\ {\Uparrow^s}X : s \ \mid \\
\mathbf{codata}\ {\uparrow_s}(X : s) : -\ \mathbf{where} & \mathbf{codata}\ {\Downarrow_s}(X : -) : s\ \mathbf{where} \\
\quad \mathtt{Eval}_s : \ \mid\ {\uparrow_s}X : - \ \vdash\ X : s & \quad \mathtt{Enter}_s : \ \mid\ {\Downarrow_s}X : s \ \vdash\ X : -
\end{array}$$

4.3.3 Static calling conventions

Systems languages like C give the programmer fine-grained control over low-level representations and calling conventions. When defining a structure, the programmer can choose if values are stored directly or indirectly (*i.e., boxed*) as a pointer into the heap. When calling a function, the programmer can choose how many arguments are passed at once, and if they are passed directly in the call stack, or indirectly by reference. High-level functional languages save programmers from these details, but at the cost of using less efficient – but more uniform – representations and calling conventions. Is there a way to reconcile both high-level ease and low-level control?

It turns out that polarization also provides a logical foundation for efficient representations and calling conventions, too. Decades ago [32], Haskell implementors designed a way to add unboxed representations into the compiler IL, making it possible to more efficiently pass values directly in registers. However, doing so required extending the language, because unboxed values *must* be call-by-value, and the types of unboxed values are different from the other, ordinary Haskell types. This sounds awfully similar to polarization: unboxed values correspond to positive data types, which have a different polarity from Haskell's types.

With this inspiration, we considered the dual problem: what do negative types correspond to? If an unboxed pair (V_+, W_+) is described by the positive type $A_+ \otimes B_+$, then does an *unboxed call stack* $V_+ \cdot E_-$ correspond to the *negative function type* $A_+ \to B_-$? In [19], we found that negative functions correspond to a more primitive type of functions found in the machine, where the power of the polarized η law lets us express the arity of functions statically in the type. Static arity is important for optimizing higher-order functions. In

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
zipWith f _      _      = []
```

we cannot compile `f a b` as a simple binary function call even though `f`'s type suggests so. It might be that `f` only expects one argument, then computes a closure expecting the next. Instead, using negative functions, which are fully extensional, lets us statically optimize `zipWith` to pass both arguments to `f` at once.

However, this approach runs into some snags in practice, due to *polymorphism*. In order to be able to statically compile code, we sometimes need to know the representation of a type (to move its values around) or the calling convention of a type (to jump to its code in the correct environment). But if a type is unknown – because it's some polymorphic type variable – then that runtime information is unknown at compile time. A solution to this problem is given in [21], which introduces the idea of storing the runtime representation of values in the *kind* of their type. So even when a type is not known statically, their kind is. Following this idea, we combined the kind-based approach with function arity by storing both representations and *calling conventions* in kinds [14].

This can be seen as a refinement on the course-grained polarization from Section 4.3.2. Rather than just a basic sign – such as $-$ or $+$ – types are described by a pair of both a representation and a calling convention. Positive types like $A \otimes B$ can have interesting representations (their values can be tuples, tagged unions, or machine primitives) but have a plain convention (their terms are always just evaluated to get the resulting value). In contrast, negative types like $A \to B$ can have interesting conventions (they can be called with several arguments, which can have their own representations by value or reference) but have a plain representation (they are just stored as pointers). This approach lets us integrate efficient calling conventions in a higher-level language with polymorphism, and also lets us be polymorphic in representations and calling conventions *themselves*, introducing new forms of statically-compilable code re-use.

4.4 Orthogonal Models of Safety

We've looked at several applications based on the dual calculus in Section 3, but how do we know the calculus is *safe*? That is, what sorts of safety properties do the typing rules provide? For example, in certain applications, we might want to know for sure that well-typed programs, like the ones in Section 4.2, always terminate. We also might want a guarantee that the $\beta\eta$ equational theory in Section 3.5 is actually consistent. To reason about the impact of types, we must identify the safety property we're interested in. This is done with a chosen set of commands $\perp\!\!\!\perp$ called the *pole* which contains only those commands we deem as "safe." Despite being tailor-made to classify different notions of safety, there are shockingly few requirements of $\perp\!\!\!\perp$. In fact, the only requirement is that the pole must be *closed under expansion*: $c \mapsto c' \in \perp\!\!\!\perp$ implies $c \in \perp\!\!\!\perp$. *Any* set of commands closed under expansion can be used for $\perp\!\!\!\perp$. This gives the general framework for modeling type safety a large amount of flexibility to capture different properties, types, and language features. So in the following, assume only that $\perp\!\!\!\perp$ is an arbitrary set closed under expansion, and the sign s can stand for either $+$ (call-by-value) or $-$ (call-by-name) throughout.

4.4.1 Orthogonality and intuitionistic negation

The central concept in these family of models is *orthogonality* given in terms of the chosen pole $\perp\!\!\!\perp$. At an individual level, a term and coterm are orthogonal to one another, written $v \perp\!\!\!\perp e$, if they form a command in the pole: $\langle v \| e \rangle \in \perp\!\!\!\perp$. Generalizing to groups, a set of terms $\mathbb{A}^+$ and a set of coterms $\mathbb{A}^-$ are orthogonal, written $\mathbb{A}^+ \perp\!\!\!\perp \mathbb{A}^-$, if every combination drawn from the two sets is orthogonal: $v \perp\!\!\!\perp e$ for all $v \in \mathbb{A}^+$ and $e \in \mathbb{A}^-$. Working with sets has the benefit that we can always find the *biggest* set orthogonal to another. That is, for any set of terms $\mathbb{A}^+$, there is a largest set of coterms called $\mathbb{A}^{+\perp\!\!\!\perp}$ such that $\mathbb{A}^+ \perp\!\!\!\perp \mathbb{A}^{+\perp\!\!\!\perp}$ (and vice versa for any coterm set $\mathbb{A}^-$, there is a largest $\mathbb{A}^{-\perp\!\!\!\perp} \perp\!\!\!\perp \mathbb{A}^-$), defined as:

$$e \in \mathbb{A}^{+\perp\!\!\!\perp} \iff \forall v \in \mathbb{A}^+ . \langle v \| e \rangle \in \perp\!\!\!\perp \qquad\qquad v \in \mathbb{A}^{-\perp\!\!\!\perp} \iff \forall e \in \mathbb{A}^- . \langle v \| e \rangle \in \perp\!\!\!\perp$$

The fascinating thing about this notion of orthogonality is that – despite the fact that it was designed for symmetric and classical systems – it so closely mimics the properties of negation from the asymmetric *intuitionistic* logic. For example, it enjoys the properties analogous to *double negation introduction* ($A \implies \neg\neg A$) and *triple negation elimination* ($\neg\neg\neg A \iff A$) where $\mathbb{A}^{\pm\perp\!\!\!\perp}$ corresponds to the negation of $\mathbb{A}^{\pm}$ (which could be either a set of terms or a set of coterms) and set inclusion $\mathbb{A}^{\pm} \subseteq \mathbb{B}^{\pm}$ corresponds to implication.

▶ **Lemma 3** (Orthogonal Introduction/Elimination). *$\mathbb{A}^{\pm} \subseteq \mathbb{A}^{\pm\perp\!\!\!\perp\perp\!\!\!\perp}$ and $\mathbb{A}^{\pm\perp\!\!\!\perp\perp\!\!\!\perp\perp\!\!\!\perp} = \mathbb{A}^{\pm\perp\!\!\!\perp}$.*

However, the classical principle of *double negation elimination* ($\neg\neg A \implies A$) does *not* hold for orthogonality: in general, $\mathbb{A}^{\pm\perp\!\!\!\perp\perp\!\!\!\perp} \not\subseteq \mathbb{A}^{\pm}$. This connection is not just a single coincidence. Orthogonality also has properties corresponding to the contrapositive ($A \implies B$ implies $\neg B \implies \neg A$) as well as all the intuitionistic directions of the de Morgan laws from Section 2.3 – where set union ($\mathbb{A}^{\pm} \cup \mathbb{B}^{\pm}$) denotes disjunction and intersection ($\mathbb{A}^{\pm} \cap \mathbb{B}^{\pm}$) denotes conjunction – but, again, *not* the classical-only directions like $\neg(A \wedge B) \implies (\neg A) \vee (\neg B)$.

Where does $\perp\!\!\!\perp$'s closure under expansion come into play? It lets us reason about sets of the form $\mathbb{A}^{\pm\perp\!\!\!\perp}$, and argue that they must contain certain elements by virtue of the way they *behave* with elements of the underlying $\mathbb{A}^{\pm}$, rather than the way they were built. For example, we can show that general μs and $\tilde{\mu}$s belong to orthogonally-defined sets, as long as their commands are safe under any possible substitution.

▶ **Observation 4.** *For any set of* values $\mathbb{A}^+$*, if* $c\{V_s/x\} \in \perp\!\!\!\perp$ *for all* $V_s \in \mathbb{A}^+$ *then* $\tilde{\mu}x.c \in \mathbb{A}^{+\perp\!\!\!\perp}$*. For any set of* covalues $\mathbb{A}^-$*, if* $c\{E_s/\alpha\} \in \perp\!\!\!\perp$ *for all* $E_s \in \mathbb{A}^-$ *then* $\mu\alpha.c \in \mathbb{A}^{-\perp\!\!\!\perp}$*.*

Proof. For all values, $V_s \in \mathbb{A}^+$, observe that $\langle V_s \| \tilde{\mu}x.c\rangle \mapsto_{\beta^s_{\tilde{\mu}}} c\{V_s/x\} \in \perp\!\!\!\perp$. Thus, $\langle V_s \| \tilde{\mu}x.c\rangle \in \perp\!\!\!\perp$ by closure under expansion, so $\tilde{\mu}x.c \in \mathbb{A}^{+\perp\!\!\!\perp}$ by definition. The other case is dual. ◁

Note the fact that Observation 4 starts with only a set of *values* or *covalues*, rather than general (co)terms. This (co)value restriction is necessary to ensure that the $\beta^s_{\tilde{\mu}}$ and β^s_{μ} rules can fire, which triggers the closure-under-expansion result. Formally, we write this restriction as $\mathbb{A}^{\pm\mathbb{V}}$ to denote the subset of $\mathbb{A}^{\pm}$ containing only (co)values, which is built into the very notion of candidates that model safety of individual types.

▶ **Definition 5** (Candidates)**.** *A* reducibility candidate*,* $\mathbb{A} \in \mathcal{RC}$*, is a pair* $\mathbb{A} = (\mathbb{A}^+, \mathbb{A}^-)$ *of a set of terms (*$\mathbb{A}^+$*) and set of coterms (*$\mathbb{A}^-$*) that is:*
Sound *For all* $v \in \mathbb{A}^+$ *and* $e \in \mathbb{A}^-$*,* $\langle v\|e\rangle \in \perp\!\!\!\perp$ *(*i.e., $\mathbb{A}^+ \perp\!\!\!\perp \mathbb{A}^-$*).*
Complete *If* $\langle v\|E_s\rangle \in \perp\!\!\!\perp$ *for all* covalues $E_s \in \mathbb{A}^-$ *then* $v \in \mathbb{A}^+$ *(*i.e., $\mathbb{A}^{-\mathbb{V}\perp\!\!\!\perp} \subseteq \mathbb{A}^+$*).*
If $\langle V_s\|e\rangle \in \perp\!\!\!\perp$ *for all* values $V_s \in \mathbb{A}^+$*, then* $e \in \mathbb{A}^-$ *(*i.e., $\mathbb{A}^{+\mathbb{V}\perp\!\!\!\perp} \subseteq \mathbb{A}^-$*).*
We write $v \in \mathbb{A}$ *as shorthand for* $v \in \mathbb{A}^+$ *and* $e \in \mathbb{A}$ *for* $e \in \mathbb{A}^-$*.*

There are two distinct ways of defining specific reducibility candidates. We could begin with a set $\mathbb{A}^+$ of terms, and build the rest of the candidate around the values of $\mathbb{A}^+$, or we can start with a set $\mathbb{A}^-$ of coterms, and build the rest around the covalues of $\mathbb{A}^-$. These are the *positive* ($\mathrm{Pos}(\mathbb{A}^+)$) and *negative* ($\mathrm{Neg}(\mathbb{A}^-)$) construction of candidates, defined as:

$$\mathrm{Pos}(\mathbb{A}^+) = (\mathbb{A}^{+\mathbb{V}\perp\!\!\!\perp\mathbb{V}\perp\!\!\!\perp}, \mathbb{A}^{+\mathbb{V}\perp\!\!\!\perp\mathbb{V}\perp\!\!\!\perp\mathbb{V}\perp\!\!\!\perp}) \qquad \mathrm{Neg}(\mathbb{A}^-) = (\mathbb{A}^{-\mathbb{V}\perp\!\!\!\perp\mathbb{V}\perp\!\!\!\perp\mathbb{V}\perp\!\!\!\perp}, \mathbb{A}^{-\mathbb{V}\perp\!\!\!\perp\mathbb{V}\perp\!\!\!\perp})$$

Importantly, these constructions are indeed reducibility candidates, meaning they are both sound and complete. But why are three applications of orthogonality needed instead of just two (like some other models in this family)? The extra orthogonality is needed because of the *(co)value restriction* $\mathbb{A}^{\pm\mathbb{V}}$ interleaved with orthogonality $\mathbb{A}^{\pm\perp\!\!\!\perp}$. Taken together, (co)value-restricted orthogonality has similar introduction and elimination properties as the general one (Lemma 3), but restricted to just (co)values rather than general (co)terms.

▶ **Lemma 6.** $\mathbb{A}^{\pm\mathbb{V}} \subseteq \mathbb{A}^{\pm\mathbb{V}\perp\!\!\!\perp\mathbb{V}\perp\!\!\!\perp\mathbb{V}}$ *and* $\mathbb{A}^{\pm\mathbb{V}\perp\!\!\!\perp\mathbb{V}\perp\!\!\!\perp\mathbb{V}\perp\!\!\!\perp\mathbb{V}} = \mathbb{A}^{\pm\mathbb{V}\perp\!\!\!\perp\mathbb{V}}$*.*

Thus, the final application of orthogonality takes these (co)values and soundly completes the rest of the candidate.[8]

4.4.2 An orthogonal view of types

With the positive and negative construction of candidates, we can define operations that are analogous to the positive and negative burden of proof from Section 2.2. Here, terms represent evidence of truth, and coterms represent evidence of falsehood, so the various connectives are built like so:

$$\begin{aligned}
\mathbb{A} \otimes \mathbb{B} &= \mathrm{Pos}\{(v, w) \mid v \in \mathbb{A}, w \in \mathbb{B}\} & \mathbb{A} \parr \mathbb{B} &= \mathrm{Neg}\{[e, v] \mid e \in \mathbb{A}, f \in \mathbb{B}\} \\
\mathbb{A} \oplus \mathbb{B} &= \mathrm{Pos}(\{\iota_1 v \mid v \in \mathbb{A}\} \cup \{\iota_2 w \mid w \in \mathbb{B}\}) & \mathbb{A} \,\&\, \mathbb{B} &= \mathrm{Neg}(\{\pi_1 e \mid e \in \mathbb{A}\} \cup \{\pi_2 f \mid f \in \mathbb{B}\}) \\
\ominus\mathbb{A} &= \mathrm{Pos}\{(e) \mid e \in \mathbb{A}\} & \neg\mathbb{A} &= \mathrm{Neg}\{[v] \mid v \in \mathbb{A}\}
\end{aligned}$$

[8] In fact, the simpler double-orthogonal constructions are valid, but only in certain evaluation strategies. In call-by-value, where $\mathbb{A}^{-\mathbb{V}} = \mathbb{A}^-$ because every coterm is a covalue, the positive construction simplifies to just the usual $\mathrm{Pos}(\mathbb{A}^+) = (\mathbb{A}^{+\perp\!\!\!\perp\perp\!\!\!\perp}, \mathbb{A}^{+\perp\!\!\!\perp})$ when $\mathbb{A}^+$ contains only values. Dually in call-by-name, the negative construction simplifies to just $\mathrm{Neg}(\mathbb{A}^-) = (\mathbb{A}^{-\perp\!\!\!\perp}, \mathbb{A}^{-\perp\!\!\!\perp\perp\!\!\!\perp})$ when $\mathbb{A}^-$ contains only covalues.

Similarly, evidence for or against the existential and universal quantifiers can be defined as operations taking a function $\mathbb{F} : \mathcal{RC} \to \mathcal{RC}$ over reducibility candidates, and producing a specific reducibility candidate that quantifies over *all* possible instances of $\mathbb{F}(\mathbb{B})$.[9]

$$\exists\mathbb{F} = \mathrm{Pos}\{(A, v) \mid \mathbb{B} \in \mathcal{RC}, v \in \mathbb{F}(\mathbb{B})\} \qquad \forall\mathbb{F} = \mathrm{Neg}\{[A, e] \mid \mathbb{B} \in \mathcal{RC}, e \in \mathbb{F}(\mathbb{B})\}$$

With a semantic version of the connectives, we have a direct way to translate each syntactic type to a reducibility candidate. The translation $[\![A]\!]\theta$ is aided by a map θ from type variables to reducibility candidates, and the cases of translation are now by the numbers:

$$[\![X]\!]\theta = \theta(X) \qquad [\![A \otimes B]\!]\theta = [\![A]\!]\theta \otimes [\![B]\!]\theta \qquad \dots \qquad [\![\forall X.B]\!]\theta = \forall(\lambda\mathbb{A}{:}\mathcal{RC}.[\![B]\!]\theta\{\mathbb{A}/X\})$$

Going further, we can translate typing judgments to logical statements.

$$\begin{aligned}
[\![c : (\Gamma \vdash \Delta)]\!]\theta &= \forall\sigma \in [\![\Gamma \vdash \Delta]\!]\theta.\ c\{\sigma\} \in {\perp\!\!\!\perp} \\
[\![\Gamma \vdash v : A \mid \Delta]\!]\theta &= \forall\sigma \in [\![\Gamma \vdash \Delta]\!]\theta.\ v\{\sigma\} \in [\![A]\!]\theta \\
[\![\Gamma \mid e : A \vdash \Delta]\!]\theta &= \forall\sigma \in [\![\Gamma \vdash \Delta]\!]\theta.\ e\{\sigma\} \in [\![A]\!]\theta
\end{aligned}$$

Each judgment is based on a translation of the environment, $\sigma \in [\![\Gamma \vdash \Delta]\!]\theta$, which says that σ is a syntactic substitution of (co)values for (co)variables such that $x\{\sigma\} \in [\![A]\!]\theta$ if $x : A$ is in Γ, and similarly for $\alpha : A$ in Δ. The main result is that typing derivations imply the truth of their concluding judgment, which follows by induction on the derivation.

▶ **Theorem 7** (Adequacy). *$c : (\Gamma \vdash \Delta)$ implies $[\![c : (\Gamma \vdash \Delta)]\!]\theta$ (and similar for (co)terms).*

4.4.3 Applications of adequacy

Adequacy (Theorem 7) may not seem like a special property, but the generality of the model means that it has many serious implications. We get different results by plugging in a different notion of safety for ${\perp\!\!\!\perp}$. The most basic corollary of adequacy is given by the most trivial pole: ${\perp\!\!\!\perp} = \{\}$ is vacuously closed under expansion since it is empty to start with. By instantiating adequacy with ${\perp\!\!\!\perp} = \{\}$, we get a notion of logical consistency, there is no derivation of a closed contradiction $c : (\bullet \vdash \bullet)$ since $[\![c : (\bullet \vdash \bullet)]\!]$ means that $c \in \{\}$.

▶ **Corollary 8** (Logical Consistency). *There is no well-typed $c : (\bullet \vdash \bullet)$.*

However, the most interesting results come from instances where ${\perp\!\!\!\perp}$ is not empty. For example, the set of terminating commands, $\{c \mid c \mapsto\!\!\!\!\rightarrow c' \not\mapsto\}$, is also closed under expansion. Defining ${\perp\!\!\!\perp}$ as this set ensures that all well-typed commands are terminating.

▶ **Corollary 9** (Termination). *If $c : (\Gamma \vdash \Delta)$ then $c \mapsto\!\!\!\!\rightarrow_{\beta\varsigma} c' \not\mapsto$.*

But perhaps the most relevant application to discuss here is how *constructivity* from Section 2 is reconciled with computation in Section 3. The notion of positive constructive evidence of $A \oplus B$ (Section 2.2) corresponds directly with the two value constructors: we have $\iota_1 V_1 : A_1 \oplus A_2$ and $\iota_2 V_2 : A_1 \oplus A_2$ for any value $V_i : A_i$. Similarly, the evidence in favor of $\exists X.B$ corresponds directly with the constructed value $(A, V) : \exists X.B$ where $V : B[A/X]$.

[9] Note that there is no connection between the syntactic type A used in (A, v) and $[A, e]$ and the actual reducibility candidate used in $\mathbb{F}(\mathbb{B})$ that classifies v and e. Just like System F's model of impredicativity [22], we can get away with this bald-faced lie because of *parametricity* of $\forall$ and $\exists$: the (co)term that unpacks (A, v) or $[A, e]$ is not allowed to react any differently based on the choice for A.

But both of these types also have the general μ abstractions $\mu\alpha.c : A \oplus B$ and $\mu\beta.c' : \exists X.B$, which do not directly correspond with either. How do we know that both of these μs will compute and eventually produce the required evidence? We can instantiate $\perp\!\!\!\perp$ with only the commands that do so. For $A \oplus B$ we can set $\perp\!\!\!\perp = \{c \mid c \mapsto\!\!\!\!\rightarrow \langle \iota_i V \| \alpha \rangle\}$, and for $\exists X.B$ we can set $\perp\!\!\!\perp = \{c \mid c \mapsto\!\!\!\!\rightarrow \langle (A, V) \| \alpha \rangle\}$; both of these definitions are closed under expansion, which is all we need to apply adequacy to compute the construction matching the type.

▶ **Corollary 10** (Positive Evidence)**.** *If* $\bullet \vdash v : A_1 \oplus A_2 \mid$ *then* $\langle v \| \alpha \rangle \mapsto\!\!\!\!\rightarrow_{\beta^s\varsigma^s} \langle \iota_i V_s \| \alpha \rangle$ *such that* $V_s \in [\![A_i]\!]$*. If* $\bullet \vdash v : \exists X.B \mid$ *then* $\langle v \| \alpha \rangle \mapsto\!\!\!\!\rightarrow_{\beta^s\varsigma^s} \langle (A, V_s) \| \alpha \rangle$ *such that* $V_s \in [\![B]\!]\{[\![A]\!]/X\}$*.*

Dually, we can design similar poles which characterize the computation of negative evidence. For example, types like $A_1 \,\&\, A_2$ and $\forall X.B$ include general $\tilde{\mu}$ abstractions of the form $\tilde{\mu}x.c$ in addition to the constructed covalues $\pi_1 E_1 : A_1$, $\pi_2 E_2 : A_2$, and $[A, E] : \forall X.B$ that correspond to the negative evidence of these connectives. Luckily, we can set the global $\perp\!\!\!\perp$ to either $\{c \mid c \mapsto\!\!\!\!\rightarrow \langle x \| \pi_i E \rangle\}$ or $\{c \mid c \mapsto\!\!\!\!\rightarrow \langle x \| [A, E] \rangle\}$ to ensure that general $\tilde{\mu}$s compute the correct concrete evidence for these negative types.

▶ **Corollary 11** (Negative Evidence)**.** *If* $\mid e : A_1 \,\&\, A_2 \vdash \bullet$ *then* $\langle x \| e \rangle \mapsto\!\!\!\!\rightarrow_{\beta^s\varsigma^s} \langle x \| \pi_i E_s \rangle$ *such that* $E_s \in [\![A_i]\!]$*. If* $\mid e : \forall X.B \vdash \bullet$ *then* $\langle x \| e \rangle \mapsto\!\!\!\!\rightarrow_{\beta^s\varsigma^s} \langle x \| [A, E_s] \rangle$ *such that* $E_s \in [\![B]\!]\{[\![A]\!]/X\}$*.*

This model is extensible with other language features, too, without fundamentally changing the shape of adequacy (Theorem 7). For example, because reducibility candidates are two-sided objects, there are two different ways to order them:

$$\mathbb{A} \sqsubseteq \mathbb{B} \iff \mathbb{A}^+ \subseteq \mathbb{B}^+ \text{ and } \mathbb{A}^- \subseteq \mathbb{B}^- \qquad \mathbb{A} \leq \mathbb{B} \iff \mathbb{A}^+ \subseteq \mathbb{B}^+ \text{ and } \mathbb{A}^- \supseteq \mathbb{B}^-$$

The first order $\mathbb{A} \sqsubseteq \mathbb{B}$ where both sides are in the same direction is analogous to ordinary set containment. However, the second order $\mathbb{A} \leq \mathbb{B}$ where the two sides are opposite instead denotes *subtyping* [15]. Besides modeling subtyping as a language feature itself, this idea is the backbone of several other type features, including (co)inductive types [12], intersection and union types [13], and indexed (co)data types [16]. It also lets us model non-determinism [15], where the critical pair between μ and $\tilde{\mu}$ is allowed.

We can also generalize the form of our model, to capture properties that are binary relations rather than unary predicates. This only requires that we make each of the fundamental components binary, without changing their overall structure. For example, the pole $\perp\!\!\!\perp$ is generalized from a *set* to a *relation* between commands that is closed under expansion: $c_1 \mapsto\!\!\!\!\rightarrow c'_1 \perp\!\!\!\perp c'_2 \leftarrow\!\!\!\!\leftarrow\!\!\!| \, c_2$ implies $c_1 \perp\!\!\!\perp c_2$. From there, reducibility candidates become a pair of term relation $v \;\mathbb{A}^+\; v$ and coterm relation $e \;\mathbb{A}^-\; e'$, where soundness and completeness can be derived from the generalized notion of orthogonality between relations:

$$\mathbb{A}^+ \perp\!\!\!\perp \mathbb{A}^- \iff \forall (v \;\mathbb{A}^+\; v'),\ (e \;\mathbb{A}^-\; e').\ \langle v \| e \rangle \perp\!\!\!\perp \langle v' \| e' \rangle$$

This lets us represent equalities between commands and (co)terms in the orthogonality model, and prove that the equational theory is consistent with contextual equivalence [6], *i.e.,* equal expressions produce the same result in any context. As a consequence, (co)values built with distinct constructors – such as ι_1 and ι_2 or π_1 and π_2 – are never equal.

▶ **Corollary 12** (Equational Consistency)**.** *The equalities* $\Gamma \vdash \iota_1 V_s = \iota_2 V'_s : A \oplus B \mid \Delta$ *and* $\Gamma \mid \pi_1 E_s = \pi_2 E'_s : A \,\&\, B \vdash \Delta$ *are not derivable.*

5 Conclusion

Duality is not just an important idea in logic; it is also a useful tool to study and implement programs. By re-imagining constructive logic as a fair debate between multiple competing viewpoints, we derive a symmetric calculus that lets us transfer the logical idea of duality to computation. This modest idea has serious ramifications, and leads to several applications in both the theory and practice of programming languages. Moreover, it reveals new ideas and new relationships that are not expressible in today's languages. We hope the next generation of programming languages puts the full force of duality into programmers' hands.

References

1 Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming infinite structures by observations. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 27–38, 2013.

2 Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

3 L. E. J. Brouwer. *Over de Grondslagen der Wiskunde*. PhD thesis, University of Amsterdam, 1907.

4 Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 233–243. ACM, 2000.

5 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

6 Paul Downen. *Sequent Calculus: A Logic and a Language for Computation and Duality*. PhD thesis, University of Oregon, 2017.

7 Paul Downen and Zena M. Ariola. Compositional semantics for composable continuations: From abortive to delimited control. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 109–122. ACM, 2014.

8 Paul Downen and Zena M. Ariola. The duality of construction. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 249–269. Springer Berlin Heidelberg, 2014.

9 Paul Downen and Zena M. Ariola. Beyond polarity: Towards a multi-discipline intermediate language with sharing. In *27th EACSL Annual Conference on Computer Science Logic, CSL 2018, September 4-7, 2018, Birmingham, UK*, LIPIcs, pages 21:1–21:23. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

10 Paul Downen and Zena M. Ariola. A tutorial on computational classical logic and the sequent calculus. *Journal of Functional Programming*, 28:e3, 2018.

11 Paul Downen and Zena M. Ariola. Compiling with classical connectives. *Logical Methods in Computer Science*, 16:13:1–13:57, 2020. `arXiv:1907.13227`.

12 Paul Downen and Zena M. Ariola. A computational understanding of classical (co)recursion. In *22nd International Symposium on Principles and Practice of Declarative Programming*, PPDP '20. Association for Computing Machinery, 2020.

13 Paul Downen, Zena M. Ariola, and Silvia Ghilezan. The duality of classical intersection and union types. *Fundamenta Informaticae*, 170:1–54, 2019.

14 Paul Downen, Zena M. Ariola, Simon Peyton Jones, and Richard A. Eisenberg. Kinds are calling conventions. *Proceedings of the ACM on Programming Languages*, 4(ICFP), 2020.

15 Paul Downen, Philip Johnson-Freyd, and Zena Ariola. Abstracting models of strong normalization for classical calculi. *Journal of Logical and Algebraic Methods in Programming*, 111, 2019.

16 Paul Downen, Philip Johnson-Freyd, and Zena M. Ariola. Structures for structural recursion. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 127–139. ACM, 2015.

17 Paul Downen, Luke Maurer, Zena M. Ariola, and Simon Peyton Jones. Sequent calculus as a compiler intermediate language. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 74–88. ACM, 2016.

18 Paul Downen, Zachary Sullivan, Zena M. Ariola, and Simon Peyton Jones. Codata in action. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 119–146. Springer International Publishing, 2019.

19 Paul Downen, Zachary Sullivan, Zena M. Ariola, and Simon Peyton Jones. Making a faster curry with extensional types. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, pages 58–70. ACM, 2019.

20 M. Dummett and R. Minio. *Elements of Intuitionism.* Oxford University Press, 1977.

21 Richard A. Eisenberg and Simon Peyton Jones. Levity polymorphism. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 525–539. Association for Computing Machinery, 2017.

22 Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the 2nd Scandinavian Logic Symposium*, pages 63–92. North-Holland, 1971.

23 Timothy G. Griffin. A formulae-as-types notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 47–58. ACM, 1990.

24 Arend Heyting. Die formalen regeln der intuitionistischen logik. *Sitzungsbericht PreuBische Akademie der Wissenschaften*, pages 42–56, 1930.

25 William Alvin Howard. The formulae-as-types notion of construction. In Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Academic Press, 1980.

26 Philip Johnson-Freyd, Paul Downen, and Zena M. Ariola. First class call stacks: Exploring head reduction. In *Workshop on Continuations*, volume 212 of *WOC*, 2015.

27 Philip Johnson-Freyd, Paul Downen, and Zena M. Ariola. Call-by-name extensionality and confluence. *Journal of Functional Programming*, 27:e12, 2017.

28 Jan W. Klop and Roel C. de Vrijer. Unique normal forms for lambda calculus with surjective pairing. *Information and Computation*, 80(2):97–113, 1989.

29 Luke Maurer, Paul Downen, Zena M. Ariola, and Simon Peyton Jones. Compiling without continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 482–494. ACM, 2017.

30 Guillaume Munch-Maccagnoni. *Syntax and Models of a non-Associative Composition of Programs and Proofs.* PhD thesis, Université Paris Diderot, 2013.

31 Michel Parigot. Lambda-my-calculus: An algorithmic interpretation of classical natural deduction. In *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, LPAR '92, pages 190–201. Springer-Verlag, 1992.

32 Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, volume 523 of *Lecture Notes in Computer Science*, pages 636–666. Springer-Verlag, 1991.

33 Noam Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching.* PhD thesis, Carnegie Mellon University, 2009.

Completion and Reduction Orders

Nao Hirokawa
Japan Advanced Institute of Science and Technology, Ishikawa, Japan

Abstract

We present three techniques for improving the Knuth–Bendix completion procedure and its variants: An order extension by semantic labeling, a new confluence criterion for terminating term rewrite systems, and inter-reduction for maximal completion.

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

Keywords and phrases term rewriting, completion, reduction order

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.2

Category Invited Talk

Funding *Nao Hirokawa*: JSPS KAKENHI Grant Numbers 17K00011.

1 Introduction

Completion [11] is a procedure that takes an equational system and a reduction order to construct a conversion-equivalent complete (terminating and confluent) term rewrite system.

Consider the equational system for commuting group endomorphisms (CGE_2):

$$
\begin{aligned}
\mathsf{e} + x &\approx x & \mathsf{f}(x+y) &\approx \mathsf{f}(x) + \mathsf{f}(y)\\
\mathsf{i}(x) + x &\approx \mathsf{e} & \mathsf{g}(x+y) &\approx \mathsf{g}(x) + \mathsf{g}(y)\\
(x+y)+z &\approx x+(y+z) & \mathsf{f}(x)+\mathsf{g}(y) &\approx \mathsf{g}(y)+\mathsf{f}(x)
\end{aligned}
$$

This system is known as a challenging completion problem. Stump and Löchner [16] showed that it admits the following complete TRS consisting of 20 rewrite rules:

$$
\begin{aligned}
\mathsf{e}+x &\to x & \mathsf{f}(\mathsf{e}) &\to \mathsf{e} & \mathsf{i}(x+y) &\to \mathsf{i}(y)+\mathsf{i}(x)\\
x+\mathsf{e} &\to x & \mathsf{g}(\mathsf{e}) &\to \mathsf{e} & \mathsf{f}(x+y) &\to \mathsf{f}(x)+\mathsf{f}(y)\\
\mathsf{i}(x)+x &\to \mathsf{e} & \mathsf{i}(\mathsf{e}) &\to \mathsf{e} & \mathsf{g}(x+y) &\to \mathsf{g}(x)+\mathsf{g}(y)\\
x+\mathsf{i}(x) &\to \mathsf{e} & \mathsf{i}(\mathsf{i}(x)) &\to x & \mathsf{f}(x)+\mathsf{g}(y) &\to \mathsf{g}(y)+\mathsf{f}(x)\\
x+(\mathsf{i}(x)+y) &\to y & \mathsf{i}(\mathsf{f}(x)) &\to \mathsf{f}(\mathsf{i}(x)) & \mathsf{f}(x)+(\mathsf{f}(y)+z) &\to \mathsf{f}(x+y)+z\\
\mathsf{i}(x)+(x+y) &\to y & \mathsf{i}(\mathsf{g}(x)) &\to \mathsf{g}(\mathsf{i}(x)) & \mathsf{g}(x)+(\mathsf{g}(y)+z) &\to \mathsf{g}(x+y)+z\\
(x+y)+z &\to x+(y+z) & & & \mathsf{g}(x)+(\mathsf{f}(y)+z) &\to \mathsf{f}(y)+(\mathsf{g}(x)+z)
\end{aligned}
$$

The main difficulty is that termination of the complete TRS cannot be shown by standard reduction orders such as the Knuth–Bendix order (KBO) [11] and the lexicographic path order (LPO) [8]. Therefore, existing completion tools capable of handling such a system either employ termination tools or adopts the dependency pair method [1, 5], giving up direct termination proofs by reduction orders. Instances of the former are [18, 15, 21], and an instance of the latter is [14].

In this note we present another approach to the problem. The idea is easy. We simply develop powerful reduction orders to use them for (maximal) completion. To this end, we reformulate Zantema's semantic labeling [22] as an order extension method for reduction orders (in Section 3). In order to perform completion with powerful orders effectively, we introduce a new variant of maximal completion [10, 14] that integrates the feature of rule simplification [7, 3], known as inter-reduction (in Section 5). In addition to them, we show that confluence of terminating systems can be characterized by rewrite strategies (in Section 4). This results in a new critical pair criterion.

2 Preliminaries

We assume familiarity with the basic notions of term rewriting and completion [2, 17]. Here we shortly recapitulate terminology and notation that we use in this note.

An *abstract rewrite system* ARS $\mathcal{A}$ is a pair of a set A and a binary relation $\to_{\mathcal{A}}$ on the set A. An ARS $\mathcal{A} = (A, \to_{\mathcal{A}})$ is *terminating* if there exists no infinite rewrite sequence $a_1 \to_{\mathcal{A}} a_2 \to_{\mathcal{A}} t_3 \to_{\mathcal{A}} \cdots$. An ARS $\mathcal{A}$ is *confluent* if ${}^{*}_{\mathcal{A}}{\leftarrow} \cdot \to^{*}_{\mathcal{A}} \subseteq {\downarrow_{\mathcal{A}}}$ holds. Here $\downarrow_{\mathcal{A}}$ stands for the *joinability* relation $\to^{*}_{\mathcal{A}} \cdot {}^{*}_{\mathcal{A}}{\leftarrow}$. An element a is a *normal form* of $\mathcal{A}$ if there is no element b with $a \to_{\mathcal{A}} b$. The set of all normal forms is denoted by $\mathsf{NF}(\mathcal{A})$. When an ARS $\mathcal{A}$ is terminating, an arbitrary element a admits a normal form b such that $a \to^{*}_{\mathcal{A}} b$. By $a{\downarrow_{\mathcal{A}}}$ we denote some fixed normal form of a.

Terms are built from a signature $\mathcal{F}$ and a countable set $\mathcal{V}$ of variables. An *equational system* over $\mathcal{F}$ is a set of equations. Here we assume that *equations* are ordered pairs of terms over $\mathcal{F}$. We write $s \approx t$ for the equation (s, t). An equation $s \approx t$ is called a *rewrite rule*, denoted by $s \to t$, if s is a non-variable term and $\mathcal{V}\mathsf{ar}(t) \subseteq \mathcal{V}\mathsf{ar}(s)$ holds. A *term rewrite system* (TRS) over $\mathcal{F}$ is an equational system consisting of rewrite rules over $\mathcal{F}$. The rewrite step $\to_{\mathcal{R}}$ of a TRS $\mathcal{R}$ is defined as follows: $s \to_{\mathcal{R}} t$ if there exist a rule $\ell \to r \in \mathcal{R}$, a position p of s, and a substitution σ such that $s|_p = \ell\sigma$ and $t = s[r\sigma]_p$. Any TRS $\mathcal{R}$ can be regarded as the ARS comprising the set of terms and the rewrite relation $\to_{\mathcal{R}}$.

A TRS is *complete* if it is terminating and confluent. A complete TRS $\mathcal{R}$ is called *canonical* if for every rule $\ell \to r \in \mathcal{R}$ we have $r \in \mathsf{NF}(\mathcal{R})$ and $\ell \in \mathsf{NF}(\mathcal{R}')$, where $\mathcal{R}'$ consists of $\mathcal{R}$-rules that are not a variant of $\ell \to r$. We say that $\mathcal{R}$ is a TRS *for* an equational system $\mathcal{E}$ if they are conversion-equivalent, namely, $\leftrightarrow^{*}_{\mathcal{R}} = \leftrightarrow^{*}_{\mathcal{E}}$. The aim of completion procedures is to find a complete (or canonical) TRS for a given equational system. Let $\mathcal{R}$ be a terminating TRS and $\mathcal{E}$ a set of equations. Notation $\mathcal{E}{\downarrow_{\mathcal{R}}}$ stands for the set $\{s{\downarrow_{\mathcal{R}}} \approx t{\downarrow_{\mathcal{R}}} \mid s \approx t \in \mathcal{E} \text{ and } s{\downarrow_{\mathcal{R}}} \neq t{\downarrow_{\mathcal{R}}}\}$.

Reduction orders are well-founded orders on terms that are closed under contexts and substitutions. LPO and KBO are instances of reduction orders. We denote them by $\succ_{\mathsf{lpo}}$ and $\succ_{\mathsf{kbo}}$, respectively.

▶ **Theorem 1.** *A TRS $\mathcal{R}$ is terminating if $\mathcal{R} \subseteq {\succ}$ holds for some reduction order $\succ$.*

Confluence of terminating TRSs is characterized by the notion of critical pair.

▶ **Definition 2** ([6]). *Let $\mathcal{R}$ be a TRS. A tuple $(\ell_1 \to r_1, p, \ell_2 \to r_2)_\sigma$ is an* overlap *of $\mathcal{R}$ if*

- *$\ell_1 \to r_1$ and $\ell_2 \to r_2$ are variants of rules in $\mathcal{R}$ with $\mathcal{V}\mathsf{ar}(\ell_1) \cap \mathcal{V}\mathsf{ar}(\ell_2) = \varnothing$,*
- *p is a function position of ℓ_2,*
- *σ is a most general unifier of ℓ_1 and $\ell_2|_p$, and*
- *$p \neq \epsilon$ or $\ell_1 \to r_1$ is not a variant of $\ell_2 \to r_2$.*

Such an overlap induces the critical peak $(\ell_2\sigma)[r_1\sigma]_p \;{}_{\mathcal{R}}{\leftarrow}\; (\ell_2\sigma)[\ell_1\sigma]_p = \ell_2\sigma \xrightarrow{\epsilon}_{\mathcal{R}} r_2\sigma$, *and the equation $(\ell_2\sigma)[r_1\sigma]_p \approx r_2\sigma$ is called a* critical pair *of $\mathcal{R}$. We write $t \;{}_{\mathcal{R}}{\leftarrow}{\rtimes}{\to}_{\mathcal{R}}\; u$ for critical pair (t, u).*

▶ **Theorem 3** ([11]). *A terminating TRS $\mathcal{R}$ is confluent if and only if ${}_{\mathcal{R}}{\leftarrow}{\rtimes}{\to}_{\mathcal{R}} \subseteq {\downarrow_{\mathcal{R}}}$ holds.*

Finally, we define terminologies for algebras. An $\mathcal{F}$-*algebra* $\mathcal{M}$ is a pair of a set A and the set of interpretations $f_{\mathcal{M}} : A^n \to A$ for each $f \in \mathcal{F}$, where n is the arity of f. Mappings from $\mathcal{V}$ to A are called *assignments*. Let $\mathcal{M} = (A, \{f_{\mathcal{M}}\}_{f \in \mathcal{F}})$ be an $\mathcal{F}$-algebra and α an assignment from $\mathcal{V}$ to A. The valuation $[\alpha]_{\mathcal{M}}(t)$ of a term t under α is inductively defined as follows:

$$[\alpha]_{\mathcal{M}}(t) = \begin{cases} \alpha(t) & \text{if } t \text{ is a variable} \\ f_{\mathcal{M}}([\alpha]_{\mathcal{M}}(t_1), \ldots, [\alpha]_{\mathcal{M}}(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

Suppose that A is non-empty and equipped with a well-founded order $>$. If every interpretation $f_{\mathcal{A}}$ is weakly monotone, $\mathcal{M}$ is said to be a *weakly monotone well-founded* algebra.

3 Reduction Orders Extended by Semantic Labeling

Semantic labeling introduced by Zantema [22] is a powerful transformation technique for proving termination of term rewrite systems. In this section we reformulate it as an order extension for reduction orders. This is technically trivial but it is useful for completion.

Semantic labeling employs a labeling function for terms. Let $\mathcal{F}$ be a signature. To each n-ary function symbol $f \in \mathcal{F}$ we assign a fresh n-ary function symbol $f^\sharp$. The union of $\mathcal{F}$ and $\{f^\sharp \mid f \in \mathcal{F}\}$ is denoted by $\mathcal{F}^\sharp$.

▶ **Definition 4.** *Let $\mathcal{F}$ and $\mathcal{G}$ be signatures with $\mathcal{F} \subseteq \mathcal{G} \subseteq \mathcal{F}^\sharp$, and let $\mathcal{M} = (A, \{f_{\mathcal{M}}\}_{f \in \mathcal{G}})$ be a $\mathcal{G}$-algebra. Given a term t over $\mathcal{F}$ and an assignment $\alpha : \mathcal{V} \to A$, the* labeled term *$\mathsf{lab}_{\mathcal{M}}(t, \alpha)$ is inductively defined as follows:*

$$\mathsf{lab}_{\mathcal{M}}(t, \alpha) = \begin{cases} t & \text{if } t \text{ is a variable} \\ f_a(\mathsf{lab}_{\mathcal{M}}(t_1, \alpha), \ldots, \mathsf{lab}_{\mathcal{M}}(t_n, \alpha)) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } f^\sharp \in \mathcal{G} \\ f(\mathsf{lab}_{\mathcal{M}}(t_1, \alpha), \ldots, \mathsf{lab}_{\mathcal{M}}(t_n, \alpha)) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } f^\sharp \notin \mathcal{G} \end{cases}$$

where, $a = [\alpha]_{\mathcal{M}}(f^\sharp(t_1, \ldots, t_n))$. Note that labeled terms are terms over the signature $\mathcal{F}_{\mathsf{lab}} := \mathcal{F} \cup \{f_a \mid f^\sharp \in \mathcal{G} \setminus \mathcal{F} \text{ and } a \in A\}$.

▶ **Example 5.** Consider the algebra $\mathcal{M} = (\mathbb{N}, \{\mathsf{g}_{\mathcal{M}}, \mathsf{f}_{\mathcal{M}}, \mathsf{f}^\sharp_{\mathcal{M}}\})$ with $\mathsf{g}_{\mathcal{M}}(x) = 0$, $\mathsf{f}_{\mathcal{M}}(x) = 1$, and $\mathsf{f}^\sharp_{\mathcal{M}}(x) = x$, and the assignment α defined by $\alpha(x) = 2$. Then, we have $\mathsf{lab}_{\mathcal{M}}(\mathsf{f}(\mathsf{g}(\mathsf{f}(x))), \alpha) = \mathsf{f}_0(\mathsf{g}(\mathsf{f}_2(x)))$. Here labels 0 and 2 are determined by $[\alpha]_{\mathcal{M}}(\mathsf{f}^\sharp(\mathsf{g}(\mathsf{f}(x)))) = 0$ and $[\alpha]_{\mathcal{M}}(\mathsf{f}^\sharp(x)) = 2$.

We now present an order extension by semantic labeling.

▶ **Definition 6.** *Suppose $\mathcal{F} \subseteq \mathcal{G} \subseteq \mathcal{F}^\sharp$. Let $(\mathcal{M}, >)$ be a weakly monotone well-founded $\mathcal{G}$-algebra, and $\succ$ a strict order on terms over $\mathcal{F}_{\mathsf{lab}}$. We define the binary relation $\succ^{\mathcal{M}}$ on terms over $\mathcal{F}$ as follows: $s \succ^{\mathcal{M}} t$ if for every assignment α the following inequalities hold:*

$$[\alpha]_{\mathcal{M}}(s) \geqslant [\alpha]_{\mathcal{M}}(t) \qquad \mathsf{lab}_{\mathcal{M}}(s, \alpha) \succ \mathsf{lab}_{\mathcal{M}}(t, \alpha)$$

Moreover, we define the TRS $\mathcal{D}\mathsf{ec}(\mathcal{M}, >)$ as follows:

$$\mathcal{D}\mathsf{ec}(\mathcal{M}, >) = \{f_a(x_1, \ldots, x_n) \to f_b(x_1, \ldots, x_n) \mid f^\sharp \in \mathcal{G} \setminus \mathcal{F} \text{ and } a > b\}$$

where, $x_1, \ldots, x_n$ are pairwise distinct variables and n is the arity of f.

▶ **Theorem 7.** *Suppose $\mathcal{F} \subseteq \mathcal{G} \subseteq \mathcal{F}^\sharp$. Let $(\mathcal{M}, >)$ be a weakly monotone well-founded $\mathcal{G}$-algebra, and $\succ$ a reduction order on terms over $\mathcal{F}_{\mathsf{lab}}$. If $\mathcal{D}\mathsf{ec}(\mathcal{M}, >) \subseteq \succ$ holds, $\succ^{\mathcal{M}}$ is a reduction order on terms over $\mathcal{F}$.*

Proof. Immediate from [22, Theorem 8]. ◀

In the remaining part of the paper, the extended versions of KBO and LPO ($\succ_{\mathsf{kbo}}^{\mathcal{M}}$ and $\succ_{\mathsf{lpo}}^{\mathcal{M}}$) are referred to as EKBO and ELPO, respectively. We illustrate the use of EKBO by examples.

▶ **Example 8.** Consider the one-rule TRS $\mathcal{R}$:

$$\mathsf{f}(\mathsf{f}(x)) \to \mathsf{f}(\mathsf{g}(\mathsf{f}(x)))$$

Let $\mathcal{M} = (\mathbb{N}, \{\mathsf{g}_{\mathcal{M}}, \mathsf{f}_{\mathcal{M}}, \mathsf{f}^{\sharp}_{\mathcal{M}}\})$ be the weakly monotone well-founded algebra given by the interpretations $\mathsf{g}_{\mathcal{M}}(x) = 0$, $\mathsf{f}_{\mathcal{M}}(x) = 1$, and $\mathsf{f}^{\sharp}_{\mathcal{M}}(x) = x$. The KBO $\succ_{\mathsf{kbo}}$ with the weight function given by

$$w(\mathsf{g}) = 0 \qquad w(\mathsf{f}_a) = 1 \quad \text{for all } a \in \mathbb{N} \qquad w(x) = 1 \quad \text{for all variables } x$$

and the well-founded precedence $\mathsf{g} \succ \cdots \succ \mathsf{f}_2 \succ \mathsf{f}_1 \succ \mathsf{f}_0$ satisfies the inclusion:

$$\mathcal{D}\mathsf{ec}(\mathcal{M}, >) = \{\mathsf{f}_a(x) \to \mathsf{f}_b(x) \mid a > b\} \subseteq \succ_{\mathsf{kbo}}$$

Thus, the EKBO $\succ_{\mathsf{kbo}}^{\mathcal{M}}$ is a reduction order. Let $\ell \to r$ denote the rule of the TRS. We have the inequalities $[\alpha]_{\mathcal{M}}(\ell) = 1 \geqslant 1 = [\alpha]_{\mathcal{M}}(r)$ and $\mathsf{lab}_{\mathcal{M}}(\ell, \alpha) = \mathsf{f}_1(\mathsf{f}_{\alpha(x)}(x)) \succ_{\mathsf{kbo}} \mathsf{f}_0(\mathsf{g}(\mathsf{f}_{\alpha(x)}(x)))) = \mathsf{lab}_{\mathcal{M}}(r, \alpha)$ for all assignments α. Therefore, $\ell \succ_{\mathsf{kbo}}^{\mathcal{M}} r$ holds. Hence, $\mathcal{R}$ is terminating.

▶ **Example 9.** We show termination of the complete TRS $\mathcal{R}$ for CGE_2 in the introduction. Let $\mathcal{M} = (\mathbb{N}, \{\mathsf{e}_{\mathcal{M}}, \mathsf{f}_{\mathcal{M}}, \mathsf{g}_{\mathcal{M}}, \mathsf{i}_{\mathcal{M}}, +_{\mathcal{M}}, +^{\sharp}_{\mathcal{M}}\})$ be the weakly monotone algebra with the interpretations:

$$\mathsf{e}_{\mathcal{M}} = 0 \quad \mathsf{f}_{\mathcal{M}}(x) = 0 \quad \mathsf{g}_{\mathcal{M}}(x) = 1 \quad \mathsf{i}_{\mathcal{M}}(x) = x \quad x +_{\mathcal{M}} y = x + y \quad x +^{\sharp}_{\mathcal{M}} y = x$$

The KBO $\succ_{\mathsf{kbo}}$ comprising the weight function

$$\begin{array}{lll} w(\mathsf{i}) = 0 & w(+_a) = 0 & \text{for all } a \in \mathbb{N} \\ w(\mathsf{g}) = w(\mathsf{f}) = w(\mathsf{e}) = 1 & w(x) = 1 & \text{for all variables } x \end{array}$$

and the well-founded precedence $\mathsf{i} \succ \mathsf{g} \succ \cdots \succ +_2 \succ +_1 \succ +_0 \succ \mathsf{e} \succ \mathsf{f}$ satisfies the inclusion:

$$\mathcal{D}\mathsf{ec}(\mathcal{M}, >) = \{x +_a y \to x +_b y \mid a > b\} \subseteq \succ_{\mathsf{kbo}}$$

Thus, $\succ_{\mathsf{kbo}}^{\mathcal{M}}$ is a reduction order. It is easy to verify that $[\alpha]_{\mathcal{M}}(\ell) \geqslant [\alpha]_{\mathcal{M}}(r)$ holds for every rules $\ell \to r \in \mathcal{R}$ and assignment α. The inequality $\mathsf{lab}_{\mathcal{M}}(\ell, \alpha) \succ_{\mathsf{kbo}} \mathsf{lab}_{\mathcal{M}}(r, \alpha)$ holds too:

$$\begin{array}{rll}
\mathsf{e} +_0 x \succ_{\mathsf{kbo}} x & \mathsf{f}(\mathsf{e}) \succ_{\mathsf{kbo}} \mathsf{e} & \mathsf{i}(x +_a y) \succ_{\mathsf{kbo}} \mathsf{i}(y) +_b \mathsf{i}(x) \\
x +_a \mathsf{e} \succ_{\mathsf{kbo}} x & \mathsf{g}(\mathsf{e}) \succ_{\mathsf{kbo}} \mathsf{e} & \mathsf{f}(x +_a y) \succ_{\mathsf{kbo}} \mathsf{f}(x) +_0 \mathsf{f}(y) \\
\mathsf{i}(x) +_a x \succ_{\mathsf{kbo}} \mathsf{e} & \mathsf{i}(\mathsf{e}) \succ_{\mathsf{kbo}} \mathsf{e} & \mathsf{g}(x +_a y) \succ_{\mathsf{kbo}} \mathsf{g}(x) +_1 \mathsf{g}(y) \\
x +_a \mathsf{i}(x) \succ_{\mathsf{kbo}} \mathsf{e} & \mathsf{i}(\mathsf{i}(x)) \succ_{\mathsf{kbo}} x & \mathsf{f}(x) +_0 \mathsf{g}(y) \succ_{\mathsf{kbo}} \mathsf{g}(y) +_1 \mathsf{f}(x) \\
x +_a (\mathsf{i}(x) +_a y) \succ_{\mathsf{kbo}} y & \mathsf{i}(\mathsf{f}(x)) \succ_{\mathsf{kbo}} \mathsf{f}(\mathsf{i}(x)) & \mathsf{f}(x) +_0 (\mathsf{f}(y) +_0 z) \succ_{\mathsf{kbo}} \mathsf{f}(x +_a y) + z \\
\mathsf{i}(x) +_a (x +_a y) \succ_{\mathsf{kbo}} y & \mathsf{i}(\mathsf{g}(x)) \succ_{\mathsf{kbo}} \mathsf{g}(\mathsf{i}(x)) & \mathsf{g}(x) +_1 (\mathsf{g}(y) +_1 z) \succ_{\mathsf{kbo}} \mathsf{g}(x +_a y) + z \\
(x +_a y) +_{a+b} z \succ_{\mathsf{kbo}} x +_a (y +_b z) & & \mathsf{g}(x) +_1 (\mathsf{f}(y) +_0 z)) \succ_{\mathsf{kbo}} \mathsf{f}(y) +_0 (\mathsf{g}(x) +_1 z)
\end{array}$$

where, $a = \alpha(x)$ and $b = \alpha(y)$. Therefore, $\mathcal{R} \subseteq \succ_{\mathsf{kbo}}^{\mathcal{M}}$ holds. Hence, $\mathcal{R}$ is terminating.

By using SAT/SMT solvers one can easily implement a program to find suitable parameters for EKBOs and ELPOs. See [12] for SAT/SMT encoding technique. As a final remark in the section, ELPO is almost same as the lexicographic version of the semantic path order (SPO) [8]; see [22] for discussions on the relation between semantic labeling and SPO.

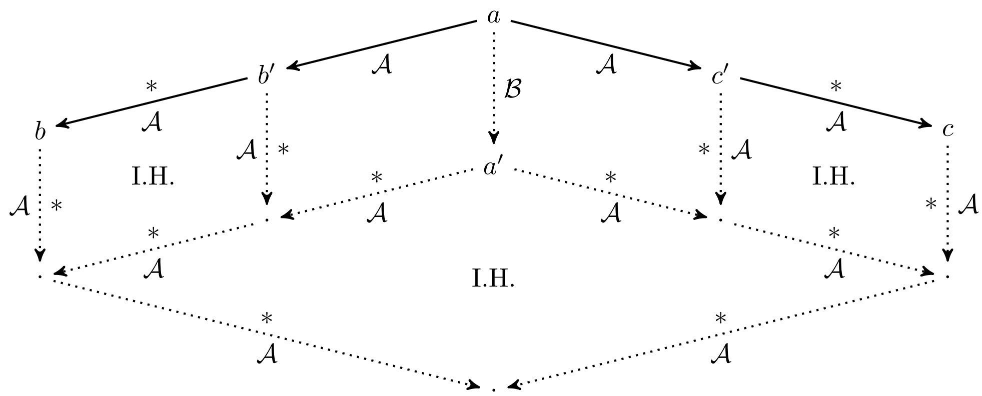


Figure 1 Proof of Theorem 11.

4 Confluence via Rewrite Strategies

In this section we present a new confluence criterion based on *rewrite strategies*.

▶ **Definition 10** ([17, Section 9.1]). *Let $\mathcal{A} = (A, \to_{\mathcal{A}})$ be an ARS. We say that an ARS $\mathcal{B} = (A, \to_{\mathcal{B}})$ is a rewrite strategy if $\to_{\mathcal{B}} \subseteq \to^{+}_{\mathcal{A}}$ and $\mathsf{NF}(\mathcal{A}) = \mathsf{NF}(\mathcal{B})$.*

▶ **Theorem 11.** *A terminating ARS $\mathcal{A}$ is confluent if and only if the inclusion ${}_{\mathcal{B}}{\leftarrow} \cdot \to_{\mathcal{A}} \subseteq {\downarrow_{\mathcal{A}}}$ holds for some rewrite strategy $\mathcal{B}$ of $\mathcal{A}$.*

Proof. The "only if"-direction is trivial as we can take $\mathcal{B} = \mathcal{A}$. We show the "if"-direction. Let $\mathcal{A}$ be a terminating ARS and $\mathcal{B}$ a rewrite strategy for $\mathcal{A}$ with ${}_{\mathcal{B}}{\leftarrow} \cdot \to_{\mathcal{A}} \subseteq {\downarrow_{\mathcal{A}}}$. Suppose $b \;{}^{*}_{\mathcal{A}}{\leftarrow}\; a \to^{*}_{\mathcal{A}} c$. As $\mathcal{A}$ is terminating, $\to^{+}_{\mathcal{A}}$ is a well-founded order. So we perform well-founded induction on a with respect to $\to^{+}_{\mathcal{A}}$ to show $b \downarrow_{\mathcal{A}} c$. If $b = a$ then $b \to^{*}_{\mathcal{A}} c$. Thus, $b \downarrow_{\mathcal{A}} c$ holds. Similarly, if $a = c$ then $b \;{}^{*}_{\mathcal{A}}{\leftarrow}\; c$. Thus, $b \downarrow_{\mathcal{A}} c$ holds. Otherwise, there exist b' and c' such that $b \;{}^{*}_{\mathcal{A}}{\leftarrow}\; b' \;{}_{\mathcal{A}}{\leftarrow}\; a \to_{\mathcal{A}} c' \to^{*}_{\mathcal{A}} c$ holds. Because $\mathcal{B}$ is a rewrite strategy, $a \notin \mathsf{NF}(\mathcal{A}) = \mathsf{NF}(\mathcal{B})$. Thus, there exists an element a' with $a \to_{\mathcal{B}} a'$. Since a', b', and c' are smaller than a with respect to $\to^{+}_{\mathcal{A}}$, the corresponding induction hypotheses and the assumption ${}_{\mathcal{B}}{\leftarrow} \cdot \to_{\mathcal{A}} \subseteq {\downarrow_{\mathcal{A}}}$ yield the diagram indicated in Figure 1. ◀

Using this characterization, we develop a new critical pair criterion. Let $\xrightarrow{\alpha}_{\mathcal{R}}$ be a rewrite strategy for a TRS $\mathcal{R}$. We say that a critical peak $t \;{}_{\mathcal{R}}{\leftarrow}\; s \xrightarrow{\epsilon}_{\mathcal{R}} u$ is an α-critical peak if $s \xrightarrow{\alpha}_{\mathcal{R}} t$. The corresponding critical pair (t, u) is denoted by $t \;{}_{\mathcal{R}}{\xleftarrow{\alpha}}{\rtimes}{\to}_{\mathcal{R}}\; u$. For instance, the innermost strategy $\xrightarrow{\mathsf{i}}_{\mathcal{R}}$ is a rewrite strategy. Innermost critical pairs ${}_{\mathcal{R}}{\xleftarrow{\mathsf{i}}}{\rtimes}{\to}_{\mathcal{R}}$ correspond to *prime* critical pairs.[1]

▶ **Corollary 12** ([9]). *A terminating TRS $\mathcal{R}$ is confluent if and only if ${}_{\mathcal{R}}{\xleftarrow{\mathsf{i}}}{\rtimes}{\to}_{\mathcal{R}} \subseteq {\downarrow_{\mathcal{R}}}$ holds.*

This result can be refined by adopting the leftmost innermost strategy $\xrightarrow{\mathsf{li}}_{\mathcal{R}}$. Since $\xrightarrow{\mathsf{li}}_{\mathcal{R}}$ is a subrelation of $\xrightarrow{\mathsf{i}}_{\mathcal{R}}$, the inclusion ${}_{\mathcal{R}}{\xleftarrow{\mathsf{li}}}{\rtimes}{\to}_{\mathcal{R}} \subseteq {}_{\mathcal{R}}{\xleftarrow{\mathsf{i}}}{\rtimes}{\to}_{\mathcal{R}}$ holds in general.

▶ **Corollary 13.** *A terminating TRS $\mathcal{R}$ is confluent if and only if ${}_{\mathcal{R}}{\xleftarrow{\mathsf{li}}}{\rtimes}{\to}_{\mathcal{R}} \subseteq {\downarrow_{\mathcal{R}}}$ holds.*

Proof. Since ${}_{\mathcal{R}}{\xleftarrow{\mathsf{li}}}{\rtimes}{\to}_{\mathcal{R}} \subseteq {\downarrow_{\mathcal{R}}}$ and ${}_{\mathcal{R}}{\xleftarrow{\mathsf{li}}} \cdot \to_{\mathcal{R}} \subseteq {\downarrow_{\mathcal{R}}}$ are equivalent, Theorem 11 applies. ◀

[1] This was pointed out by Masahiko Sakai (personal communication).

delete	$(\mathcal{E} \uplus \{s \approx s\}, \mathcal{R}) \vdash_{\succ} (\mathcal{E}, \mathcal{R})$	
orient$_1$	$(\mathcal{E} \uplus \{s \approx t\}, \mathcal{R}) \vdash_{\succ} (\mathcal{E}, \mathcal{R} \cup \{s \to t\})$	if $s \succ t$
orient$_2$	$(\mathcal{E} \uplus \{s \approx t\}, \mathcal{R}) \vdash_{\succ} (\mathcal{E}, \mathcal{R} \cup \{t \to s\})$	if $t \succ s$
simplify$_1$	$(\mathcal{E} \uplus \{s \approx t\}, \mathcal{R}) \vdash_{\succ} (\mathcal{E} \cup \{u \approx t\}, \mathcal{R})$	if $s \to_{\mathcal{R}} u$
simplify$_2$	$(\mathcal{E} \uplus \{s \approx t\}, \mathcal{R}) \vdash_{\succ} (\mathcal{E} \cup \{s \approx u\}, \mathcal{R})$	if $t \to_{\mathcal{R}} u$
collapse	$(\mathcal{E}, \mathcal{R} \uplus \{t \to s\}) \vdash_{\succ} (\mathcal{E} \cup \{u \approx s\}, \mathcal{R})$	if $t \to_{\mathcal{R}} u$
compose	$(\mathcal{E}, \mathcal{R} \uplus \{s \to t\}) \vdash_{\succ} (\mathcal{E}, \mathcal{R} \cup \{s \to u\})$	if $t \to_{\mathcal{R}} u$

Figure 2 Inference rules of abstract completion except deduce.

▶ **Example 14.** Consider the terminating TRS $\mathcal{R}$:

$$-0 \to 0 \qquad x + 0 \to x \qquad (-x) + x \to 0 \qquad (-x) + (-x) \to 0$$

The TRS admits five overlaps and they form the five critical peaks (a–e):

$$\begin{tikzcd}[column sep=small] & \underline{\underline{(-0)}+0} \arrow[dl, "1"'] \arrow[dr, "\epsilon"] & \\ 0+0 & \approx & 0 \end{tikzcd} \quad \begin{tikzcd}[column sep=small] & \underline{\underline{(-0)+0}} \arrow[dl, "\epsilon"'] \arrow[dr, "\epsilon"] & \\ -0 & \approx & 0 \end{tikzcd} \quad \begin{tikzcd}[column sep=small] & \underline{\underline{(-0)+0}} \arrow[dl, "\epsilon"'] \arrow[dr, "\epsilon"] & \\ 0 & \approx & -0 \end{tikzcd} \quad \begin{tikzcd}[column sep=small] & \underline{\underline{(-0)}+(-0)} \arrow[dl, "1"'] \arrow[dr, "\epsilon"] & \\ 0+(-0) & \approx & 0 \end{tikzcd} \quad \begin{tikzcd}[column sep=small] & \underline{(-0)+\underline{(-0)}} \arrow[dl, "2"'] \arrow[dr, "\epsilon"] & \\ (-0)+0 & \approx & 0 \end{tikzcd}$$

(a) (b) (c) (d) (e)

Out of the five, only (a) and (d) are leftmost innermost critical pairs (${}_{\mathcal{R}}\xleftarrow{\mathsf{li}}\!\rtimes\!\rightarrow_{\mathcal{R}}$), and they are joinable: $0 + 0 \downarrow_{\mathcal{R}} 0$ and $0 + (-0) \downarrow_{\mathcal{R}} 0$. Hence, confluence of the TRS $\mathcal{R}$ is concluded. Note that ${}_{\mathcal{R}}\xleftarrow{\mathsf{i}}\!\rtimes\!\rightarrow_{\mathcal{R}}$ contains one more critical pair (e).

▶ **Example 15.** The complete TRS for CGE$_2$ in the introduction admits 115 overlaps. Out of them, 18 overlaps are discarded by the condition of leftmost innermost critical pairs ($\xleftarrow{\mathsf{li}}\!\rtimes\!\rightarrow$). For this rewrite system $\xleftarrow{\mathsf{i}}\!\rtimes\!\rightarrow$ and $\xleftarrow{\mathsf{li}}\!\rtimes\!\rightarrow$ coincide.

Unfortunately, the outermost strategy $\xrightarrow{\mathsf{o}}_{\mathcal{R}}$ cannot be used for discarding critical pairs. The culprit is that ${}_{\mathcal{R}}\xleftarrow{\mathsf{o}}\!\rtimes\!\rightarrow_{\mathcal{R}} \subseteq \downarrow_{\mathcal{R}}$ does not imply ${}_{\mathcal{R}}\xleftarrow{\mathsf{o}} \cdot \rightarrow_{\mathcal{R}} \subseteq \downarrow_{\mathcal{R}}$ in general.

▶ **Example 16.** Consider the terminating TRS $\mathcal{R} = \{\mathsf{f}(\mathsf{f}(x)) \to \mathsf{a}\}$. Since the only critical peak

$$\mathsf{f}(\mathsf{a}) \;{}_{\mathcal{R}}\!\leftarrow \mathsf{f}(\mathsf{f}(\mathsf{f}(x))) \xrightarrow{\epsilon}_{\mathcal{R}} \mathsf{a}$$

is not an outermost critical peak, the inclusion ${}_{\mathcal{R}}\xleftarrow{\mathsf{o}}\!\rtimes\!\rightarrow_{\mathcal{R}} = \varnothing \subseteq \downarrow_{\mathcal{R}}$ holds. However, $\mathcal{R}$ is not confluent, as $\mathsf{f}(\mathsf{a})$ and a are not joinable.

5 Maximal Completion with Inter-reduction

In this section we present a new variant of maximal completion [10, 14], which incorporates inter-reduction of standard completion [7]. Figure 2 shows a subset of the inference rules of abstract completion [3], where the deduce rule is excluded. Inter-reduction corresponds to collapse and compose. Due to absence of deduce, the derivation relation $\vdash_{\succ}$ fulfils the termination property. So for any finite equational system $\mathcal{E}$ the pair $(\mathcal{E}, \varnothing)$ has a normal form with respect to $\vdash_{\succ}$. We denote its arbitrary but fixed normal form by $\psi(\mathcal{E}, \succ)$.

We now formalize our procedure. Let $\mathcal{O}$ be a mapping from an equational system to a finite set of reduction order, and S a mapping from an equational system $\mathcal{E}$ to a set of equations $s \approx t$ satisfying $s \leftrightarrow^*_{\mathcal{E}} t$.

▶ **Definition 17.** *For an equational system $\mathcal{E}$ the partial function $\varphi(\mathcal{E})$ is defined as follows:*

$$\varphi(\mathcal{E}) = \begin{cases} \mathcal{R} & \text{if } \mathcal{E}' = \varnothing \text{ and } {}_{\mathcal{R}}\overset{\mathsf{li}}{\leftarrow}\rtimes\rightarrow_{\mathcal{R}} \subseteq {\downarrow_{\mathcal{R}}} \text{ for some } {\succ} \in \mathcal{O}(\mathcal{E}) \\ \varphi(\mathcal{E} \cup \mathsf{S}(\mathcal{E})) & \text{otherwise} \end{cases}$$

where $(\mathcal{E}', \mathcal{R}) = \psi(\mathcal{E}, \succ)$.

▶ **Theorem 18.** *If $\varphi(\mathcal{E})$ is defined then it is a complete presentation of $\mathcal{E}$.*

Proof. Immediate from $\leftrightarrow^*_{\mathcal{E}} = \leftrightarrow^*_{\mathcal{E}'\cup\mathcal{R}}$ and $\leftrightarrow^*_{\mathcal{E}} = \leftrightarrow^*_{\mathcal{E}\cup\mathsf{S}(\mathcal{E})}$. ◀

The procedure $\varphi(\mathcal{E})$ runs as follows: (1) $\mathcal{O}(\mathcal{E})$ generates reduction orders; (2) for each of them $\psi(\mathcal{E}, \succ)$ runs standard completion without the deduce rule; (3) if one of them results in a confluent TRS $\mathcal{R}$, the procedure returns $\mathcal{R}$; (4) otherwise $\mathcal{E}$ is extended by $\mathsf{S}(\mathcal{E})$. The second step ψ is a new ingredient to maximal completion [10, 14, 19].

In order to evaluate effectiveness of the presented framework we implemented it on the top of the completion tool Maxcomp [10].[2] In the implementation $\mathsf{S}(\mathcal{E})$ selects 21 smallest equations from the set:

$$\bigcup_{\succ\in\mathcal{O}(\mathcal{E})} (\mathcal{E}_{\succ} \cup \mathcal{R}_{\succ} \cup \mathsf{CP}_{\mathsf{li}}(\mathcal{R}_{\succ}){\downarrow_{\mathcal{R}_{\succ}}}) \setminus \mathcal{E}$$

where, $(\mathcal{E}_{\succ}, \mathcal{R}_{\succ}) = \psi(\mathcal{E}, \succ)$ and $\mathsf{CP}_{\mathsf{li}}(\mathcal{R})$ stands for ${}_{\mathcal{R}}\overset{\mathsf{li}}{\leftarrow}\rtimes\rightarrow_{\mathcal{R}}$. The definition of $\mathcal{O}$ is based on Sato and Winkler's heuristic method [14, 19]. The method aims to find *canonical* TRSs $\mathcal{P}$ for $\mathcal{E}$ such that $\mathcal{P} \subseteq \mathcal{E} \cup \mathcal{E}^{-1}$. Assume that we want to find k orders from a designated class $\mathcal{RO}$ of reduction orders. We define $\mathcal{RO}(\mathcal{E}, k)$ as $\mathcal{RO}(\mathcal{E}, 0) = \varnothing$ and $\mathcal{RO}(\mathcal{E}, k+1) = \mathcal{RO}(\mathcal{E}, k) \cup \{(\mathcal{P}, \succ)\}$. Here $\mathcal{P}$ is a TRS and $\succ$ is a reduction order in $\mathcal{RO}$ that minimizes the cardinality of $\mathcal{P}$ subject to the three constraints: The inclusion

$$\mathcal{P} \subseteq \{s \to t \in \mathcal{E} \cup \mathcal{E}^{-1} \mid s \succ t\}$$

holds, all non-trivial equations in $\mathcal{E}$ are $\mathcal{P}$-reducible, and $\mathcal{P} \neq \mathcal{P}'$ for all $(\mathcal{P}', \succ') \in \mathcal{RO}(\mathcal{E}, k)$. Our tool employs $\mathcal{O}$ defined by $\mathcal{O}(\mathcal{E}) = \{\succ \mid (\mathcal{P}, \succ) \in \mathcal{RO}(\mathcal{E}, 2)\}$.

▶ **Example 19.** Let $\mathcal{RO}$ be the class of EKBOs. Following our procedure, we complete the next equational system:

$$1\colon\ \mathsf{s}(\mathsf{p}(x)) \approx x \qquad 2\colon\ \mathsf{p}(\mathsf{s}(x)) \approx x \qquad 3\colon\ \mathsf{s}(x) + y \approx \mathsf{s}(x+y)$$

The run of φ proceeds as follows: $\varphi(\{1,2,3\}) = \varphi(\{1,2,3,4,5\}) = \varphi(\{1,2,\ldots,8\})$, where:

$$\begin{array}{lll} 4\colon\ \mathsf{p}(\mathsf{s}(x)+y) \approx x+y & 6\colon\ \mathsf{s}((\mathsf{p}(x)+y)+z) \approx (x+y)+z & 8\colon\ \mathsf{p}(x)+y \approx \mathsf{p}(x+y) \\ 5\colon\ \mathsf{s}(\mathsf{p}(x)+y) \approx x+y & 7\colon\ \mathsf{p}((\mathsf{s}(x)+y)+z) \approx (x+y)+z & \end{array}$$

Let $\mathcal{E} = \{1, 2, \ldots, 8\}$. The function $\mathcal{RO}(\mathcal{E}, 2)$ yields $\{(\mathcal{P}_1, \succ_1), (\mathcal{P}_2, \succ_2)\}$, which pinpoints canonical TRSs for $\mathcal{E}$:

$$\mathcal{P}_1 = \{\ \mathsf{s}(\mathsf{p}(x)) \to x,\ \ \mathsf{p}(\mathsf{s}(x)) \to x,\ \ \mathsf{s}(x)+y \to \mathsf{s}(x+y),\ \ \mathsf{p}(x)+y \to \mathsf{p}(x+y)\ \}$$
$$\mathcal{P}_2 = \{\ \mathsf{s}(\mathsf{p}(x)) \to x,\ \ \mathsf{p}(\mathsf{s}(x)) \to x,\ \ \mathsf{s}(x+y) \to \mathsf{s}(x)+y,\ \ \mathsf{p}(x+y) \to \mathsf{p}(x)+y\ \}$$

Although they are ignored by $\mathcal{O}$, uniqueness of canonical TRSs [13] ensures that ψ reproduces the same TRSs: $\psi(\mathcal{E}, \succ_i) = (\varnothing, \mathcal{P}_i)$. Thus, $\varphi(\mathcal{E})$ returns one of them. Note that the EKBOs $\succ_1$ and $\succ_2$ employ algebras like $\mathsf{s}_{\mathcal{M}}(x) = \mathsf{p}_{\mathcal{M}}(x) = 0$ and $x +_{\mathcal{M}} y = 1$ to avoid unnecessary orientations for 4–7.

[2] `https://www.jaist.ac.jp/project/maxcomp/`

Table 1 Experimental results on 115 equational systems.

	LPO	ELPO	KBO	EKBO	ELPO + EKBO	KBCV	MaxcompDP
# of completed systems	81	89	82	85	96	86	97

▶ **Example 20.** Recall the equational system $\mathcal{E}$ of CGE$_2$. The procedure $\varphi(\mathcal{E})$ with the united class of ELPOs and EKBOs results in the same complete TRS in the introduction. At the last step φ maintains 120 equations. Sato and Winkler's method automatically constructs an EKBO like Example 9 to produce the 20-rule complete TRS $\mathcal{R}$ indicated in the introduction (or a symmetric variant that employs the right-associative rule $x + (y + z) \to (x + y) + z$).

Table 1 summaries experimental results on the standard set of completion problems.[3] The tests were single-threaded run on a system equipped with an Intel Core i7-1065G7 CPU with 1.3 GHz and 32 GB of RAM using a timeout of 600 seconds. We used SMT solver Z3[4] for computing $\mathcal{RO}(\mathcal{E}, k)$. See [10, 14] for the employed encoding techniques. Note that $k = 2$ is used in the implementation.

The first five columns indicate the results of our completion procedure with the classes of reduction orders LPO, ELPO, KBO, EKBO, and the union of ELPO and EKBO, respectively. Linear interpretations on natural numbers with $0, 1$-coefficients were employed for ELPO and EKBO. The union of ELPO and EKBO is the most powerful and subsumes all results of the other classes. The use of ordinary critical pairs did not change any number. For the comparison sake, we also included in the table the results of completion tools KBCV version 2.1.0.6 [15] and MaxcompDP [14].

6 Conclusion

We have presented an order extension by semantic labeling and maximal completion with inter-reduction as well as a confluence criterion based on rewrite strategies. Our primary future work is to evaluate these methods in the setting of (maximal) ordered completion [4, 20].

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
2. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
3. L. Bachmair, N. Dershowitz, and J. Hsiang. Orderings for equational proofs. In *Proc. 1st Symposium on Logic in Computer Science*, pages 346–357, 1986.
4. L. Bachmair, N. Dershowitz, and D. A. Plaisted. *Resolution of Equations in Algebraic Structures: Completion without Failure*, volume 2, pages 1–30. Academic Press, 1989.
5. J. Giesl, T. Arts, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *Journal of Symbolic Computation*, 34(1):21–58, 2002.
6. G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.
7. G. Huet. A complete proof of correctness of the Knuth-Bendix completion algorithm. *Journal of Computer and System Sciences*, 21(1):11–21, 1981.
8. S. Kamin and J.J. Lévy. Two generalizations of the recursive path ordering. Technical report, University of Illinois, 1980. Unpublished manuscript.

[3] The problem set and detailed data are available from: `http://www.jaist.ac.jp/project/maxcomp/`
[4] `https://github.com/Z3Prover/`

9 D. Kapur, D.R. Musser, and P. Narendran. Only prime superpositions need be considered in the Knuth-Bendix completion procedure. *Journal of Symbolic Computation*, 6(1):19–36, 1988.

10 D. Klein and N. Hirokawa. Maximal completion. In *Proc. 22nd International Conference on Rewriting Techniques and Applications*, volume 10 of *LIPIcs*, pages 71–80, 2011.

11 D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

12 A. Koprowski and A. Middeldorp. Predictive labeling with dependency pairs using SAT. In *Proc. 21st International Conference on Automated Deduction*, volume 4603 of *LNCS (LNAI)*, pages 410–425, 2007.

13 Y. Métivier. About the rewriting systems produced by the Knuth-Bendix completion algorithm. *Information Processing Letter*, 16(1):31–34, 1983.

14 H. Sato and S. Winkler. Encoding DP techniques and control strategies for maximal completion. In *Proc. 25th International Conference on Automated Deduction*, volume 9195 of *LNCS*, pages 152–162, 2015.

15 T. Sternagel and H. Zankl. KBCV – Knuth–Bendix completion visualizer. In *Proc. 6th International Joint Conference on Automated Reasoning*, volume 7364 of *LNCS (LNAI)*, pages 530–536, 2012.

16 A. Stump and B. Löchner. Knuth–Bendix completion of theories of commuting group endomorphisms. *Information Processing Letter*, 98(6):195–198, 2006.

17 Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

18 I. Wehrman, A. Stump, and E. M. Westbrook. Slothrop: Knuth-Bendix completion with a modern termination checker. In *Proc. 17th International Conference on Rewriting Techniques and Applications*, volume 4098 of *LNCS*, pages 287–296, 2006.

19 S. Winkler. Extending maximal completion. In *Proc. 4th International Conference on Formal Structures on Computation and Deduction*, volume 131 of *LIPIcs*, pages 3:1–3:15, 2019.

20 S. Winkler and G. Moser. MædMax: A maximal ordered completion tool. In *Proc. 9th International Joint Conference on Automated Reasoning*, volume 10900 of *LNCS*, pages 472–480, 2018.

21 S. Winkler, H. Sato, M. Kurihara, and A. Middeldorp. Multi-completion with termination tools (system description). *Journal of Automated Reasoning*, 50:317–354, 2013.

22 H. Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24:89–105, 1995.

Process-As-Formula Interpretation: A Substructural Multimodal View

Elaine Pimentel[1]
Department of Mathematics, Federal University of Rio Grande Do Norte, Natal, Brazil

Carlos Olarte
School of Science and Technology, Federal University of Rio Grande Do Norte, Natal, Brazil

Vivek Nigam
Huawei Munich Research Center, Germany

Abstract

In this survey, we show how the processes-as-formulas interpretation, where computations and proof-search are strongly connected, can be used to specify different concurrent behaviors as logical theories. The proposed interpretation is parametric and modular, and it faithfully captures behaviors such as: Linear and spatial computations, epistemic state of agents, and preferences in concurrent systems. The key for this modularity is the incorporation of multimodalities in a resource aware logic, together with the ability of quantifying on such modalities. We achieve tight adequacy theorems by relying on a focusing discipline that allows for controlling the proof search process.

2012 ACM Subject Classification Theory of computation → Proof theory; Theory of computation → Process calculi

Keywords and phrases Linear logic, proof theory, process calculi

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.3

Category Invited Talk

Funding *Elaine Pimentel*: Partially funded by the project MOSAIC (MSCA RISE 101007627).
Carlos Olarte: Partially funded by CNPq.

1 Introduction

Computational logic research has produced deep and fruitful cross-fertilizations between programming languages and proof theory. Arguably, the most well-known one is the *Curry-Howard correspondence* (also known as types-as-formulas) where (functional) programs correspond to formal proofs and their execution to cut-elimination. A second type of correspondence, *processes-as-formulas* (also known as computation-as-proof-search), was initiated by Miller [21] where, instead, (logic) programs correspond to formulas and their execution to proof search. These two foundational correspondences have been exploited to propose new programming language paradigms as well as greatly extend the expressiveness of existing ones.

When processes or programs are specified as formulas, one has to be careful with the level of adequacy obtained. In particular, it is expected that logical steps in derivations correspond to steps of computations in programs. However, different from computational systems, where *one step of computation* is rigidly determined by the operation semantics, *one step of logical reasoning* depends strongly on the logical framework chosen. Also, the logic should capture, in a natural way, the behavior of programs. For instance, intuitionistic logic (IL) is not adequate to specify systems that may consume information (substructural behavior), execute

[1] Corresponding author.

processes in different locations (spatial modalities) or time instances (timed reasoning), or when the information shared by processes is subject to quantitative information (such as preferences or costs).

Hence the need for a more expressive logic (such as *multimodal and resource aware logics*) and an appropriate notion of normal proofs as the logical counterpart of the processes-as-formulas correspondence. This paper surveys one of such choices: focused linear logic with subexponentials (SELLF) [28]. We present different mechanisms previously explored by the authors to both: extend SELLF with quantification over subexponentials; and give adequate characterizations of existing concurrent languages. This fruitful collaboration between the two areas has been useful to provide reasoning techniques for process calculi with the motto *reachability as entailment*, and also to propose declarative extensions of concurrent languages with solid logical grounds.

The *focusing discipline* [1] determines an alternating mechanism on proofs (between *focused* and *unfocused* phases), which controls the non-determinism during proof search, producing normal form proofs. Such normalization of proofs leads to a practical approach to identify logical steps: a *focused step* is a block determined by a focused phase followed by an unfocused one, in a (bottom-up) focused proof.

In Section 2 we recall the proof theory of focused intuitionistic linear logic (ILLF), which will be the base logical language for the processes-as-formulas correspondences addressed in this paper. Section 3 then introduces the base computational counterpart of the correspondence, Concurrent Constraint Programming (CCP) [42], a declarative model for concurrency. We show how to adequately capture the behavior of CCP processes in ILLF.

The *level of adequacy* attained in such interpretations will be important in order to justify the choice of the underlying logic: the closer the two systems are, the easier is to prove the correspondence. Also, a strong adequacy allows for the use of the logical system for proving properties of the computational system, or reconstructing counter-examples from failing derivations. Following [29], we classify the level of adequacy into two classes:

- **FCP** (*full completeness of proofs*) claims that processes outputting an observable are in 1-1 correspondence with the corresponding completed proofs.
- **FCD** (*full completeness of derivations*) claims that one step of computation should correspond to one step of logical reasoning.

In the first case, even though the outputs of a program are characterized by proofs in the underlying logic, it may be the case that there are steps in the logical reasoning that do not correspond to computational steps and vice-versa. In the second case, computational and (in our case, focused) logical steps are in one-to-one correspondence. We present a careful discussion about these different levels of adequacy regarding CCP and ILLF in Section 3.2, and indicate throughout the text, in each result, its level of adequacy.

Even though (focused, intuitionistic) linear logic is suitable for the encoding of (vanilla) CCP, the situation changes when *modalities* are added to concurrent systems: For that, linear logic *subexponentials* are needed. In Section 4 we present SELLF, which shares with ILL all its connectives except the exponential: instead of having a single !, it may contain as many subexponentials as one needs (written $!^a$). Such labels are organized in a pre-order, and different organizations give rise to different CCP flavors. Section 5 is then devoted to show how to add such structures *parametrically* to SELLF, obtaining strongly adequate specifications. In this way, processes may be executed and add/query constraints in different *locations*, where the meaning of such locations may vary, for example: Spaces of computation, the epistemic state of agents, time units, levels of preferences, etc. *Modularity* is guaranteed by the fact that the underline interpretation is the same: Locations in CCP become labels in SELLF. Finally, Section 6 concludes the paper.

2 Focused intuitionistic linear logic

Linear logic (LL) is a substructural logic proposed by Girard [13] as a refinement of classical and intuitionistic logics, joining the dualities of the former with many of the constructive properties of the latter.

In this paper, we will concentrate in the *intuitionistic* version of linear logic (ILL) [13], with formulas built from the following grammar

$$F, G ::= A \mid 1 \mid 0 \mid \top \mid F \otimes G \mid F \mathbin{\&} G \mid F \oplus G \mid F \multimap G \mid {!}\,F \mid \forall x.F \mid \exists x.F$$

Here, A denotes an atomic formula; $\multimap$, $\otimes$, 1 represent the *multiplicative* implication, conjunction and true, respectively; $\&$, $\top$, $\oplus$, 0 are the *additive* conjunction, true, disjunction, and false, respectively; $!$ is the *exponential*; and $\exists, \forall$ represent the existential and universal quantifiers, respectively.[2]

These connectives can be separated into two classes, the *negative*: $\multimap, \&, \top, \forall$ and the *positive*: $\otimes, \oplus, !, 1, 0, \exists$. The polarity of non-atomic formulas is inherited from its outermost connective (e.g., $F \otimes G$ is a positive formula) and any bias can be assigned to atomic formulas.[3] This partition induces an alternating mechanism on proofs, known as *focusing*, which aims at reducing the non-determinism during proof search. In this sense, focused proofs can be interpreted as *normal form* proofs.

The focusing discipline [1] is determined by the alternation of *focused* and *unfocused* phases in the proof construction. In the unfocused phase, inference rules can be applied eagerly and no backtracking is necessary; in the focused phase, on the other hand, either context restrictions apply, or choices within inference rules can lead to failures for which one may need to backtrack. These phases are totally determined by the polarities of formulas: provability is preserved when applying right/left rules for negative/positive formulas respectively, but not necessarily in other cases.

The focused intuitionistic linear logic system (ILLF) is depicted in Figure 1.

There are three contexts on the left side of ILLF sequents: the set Θ denotes the *unbounded* context, containing only formulas with a banged scope; Γ is a *linear* context containing only negative or atomic formulas; and Δ is the general linear context. Observe that formulas in the context Θ behave as in classical logic: they can be weakened (erased) or contracted (duplicated). Formulas in the other contexts are linear, and are consumed when used.

The phase distinction is reflected in the design of sequents in ILLF: the presence of "$\Uparrow$" indicates unfocused sequents, while "$\Downarrow$" marks the formula under focus in focused sequents. Sequents in ILLF have one of the following shapes:

- **i.** $\Theta; \Gamma \Uparrow \Delta \vdash F \Uparrow$ is an unfocused sequent.
- **ii.** $\Theta; \Gamma \Uparrow \cdot \vdash \cdot \Uparrow F$ is an unfocused sequent representing the end of an unfocused phase.
- **iii.** $\Theta; \Gamma \vdash F \Downarrow$ is a sequent focused on the right.
- **iv.** $\Theta; \Gamma \Downarrow F \vdash R$ is a sequent focused on the left.

The swing between focused and unfocused phases is described below.

- At the beginning of an unfocused phase, sequents have the shape (i) and: non-atomic negative formulas appearing in the right context, and positive non-atomic formulas appearing in Δ are eagerly introduced; atomic/negative left formulas are stored in Γ using the store rule S_l; atomic/positive right formulas are stored in the outermost right context using the store rule S_r.

  When this phase ends, sequents have the form (ii).

[2] Observe that the multiplicative false $\bot$ could be added to ILL's syntax. However, this would break the nice feature of having *exactly* one formula on succedent of sequents.

[3] Although the bias assigned to atoms does not interfere with provability, it changes *considerably* the shape of proofs (see, e.g., [19]).

Unfocused introduction rules

$$\frac{\Theta;\Gamma \Uparrow F,\Delta \vdash G \Uparrow}{\Theta;\Gamma \Uparrow \Delta \vdash F \multimap G \Uparrow}\ \multimap_r \qquad \frac{\Theta;\Gamma \Uparrow F,G,\Delta \vdash \mathcal{R}}{\Theta;\Gamma \Uparrow F \otimes G,\Delta \vdash \mathcal{R}}\ \otimes_l \qquad \frac{F,\Theta;\Gamma \Uparrow \Delta \vdash \mathcal{R}}{\Theta;\Gamma \Uparrow\, !F,\Delta \vdash \mathcal{R}}\ !_l$$

$$\frac{\Theta;\Gamma \Uparrow \Delta \vdash F \Uparrow \quad \Theta;\Gamma \Uparrow \Delta \vdash G \Uparrow}{\Theta;\Gamma \Uparrow \Delta \vdash F \,\&\, G \Uparrow}\ \&_r \qquad \frac{\Theta;\Gamma \Uparrow F,\Delta \vdash \mathcal{R} \quad \Theta;\Gamma \Uparrow G,\Delta \vdash \mathcal{R}}{\Theta;\Gamma \Uparrow F \oplus G,\Delta \vdash \mathcal{R}}\ \oplus_l$$

$$\frac{\Theta;\Gamma \Uparrow \Delta \vdash F[y/x] \Uparrow}{\Theta;\Gamma \Uparrow \Delta \vdash \forall x.F \Uparrow}\ \forall_r \qquad \frac{\Theta;\Gamma \Uparrow F[y/x],\Delta \vdash \mathcal{R}}{\Theta;\Gamma \Uparrow \exists x.F,\Delta \vdash \mathcal{R}}\ \exists_l$$

$$\frac{}{\Theta;\Gamma \Uparrow \Delta \vdash \top \Uparrow}\ \top_r \qquad \frac{\Theta;\Gamma \Uparrow \Delta \vdash \mathcal{R}}{\Theta;\Gamma \Uparrow 1,\Delta \vdash \mathcal{R}}\ 1_l \qquad \frac{}{\Theta;\Gamma \Uparrow 0,\Delta \vdash \mathcal{R}}\ 0_l$$

Focused introduction rules

$$\frac{\Theta;\Gamma_1 \vdash F \Downarrow \quad \Theta;\Gamma_2 \Downarrow G \vdash R}{\Theta;\Gamma_1,\Gamma_2 \Downarrow F \multimap G \vdash R}\ \multimap_l \qquad \frac{\Theta;\Gamma \vdash F_i \Downarrow}{\Theta;\Gamma \vdash F_1 \oplus F_2 \Downarrow}\ \oplus_{r_i} \qquad \frac{\Theta;\Gamma \Downarrow F_i \vdash R}{\Theta;\Gamma \Downarrow F_1 \,\&\, F_2 \vdash R}\ \&_{l_i}$$

$$\frac{\Theta;\Gamma_1 \vdash F \Downarrow \quad \Theta;\Gamma_2 \vdash G \Downarrow}{\Theta;\Gamma_1,\Gamma_2 \vdash F \otimes G \Downarrow}\ \otimes_r \qquad \frac{\Theta;\cdot \Uparrow \cdot \vdash F \Uparrow}{\Theta;\cdot \vdash\, !F \Downarrow}\ !_r$$

$$\frac{\Theta;\Gamma \Downarrow F[t/x] \vdash R}{\Theta;\Gamma \Downarrow \forall x.F \vdash R}\ \forall_l \qquad \frac{\Theta;\Gamma \vdash F[t/x] \Downarrow}{\Theta;\Gamma \vdash \exists x.F \Downarrow}\ \exists_r \qquad \frac{}{\Theta;\cdot \vdash 1 \Downarrow}\ 1_r$$

Structural and identity rules

$$\frac{\Theta;\Gamma \Downarrow N \vdash R}{\Theta;\Gamma,N \Uparrow \cdot \vdash \cdot \Uparrow R}\ \mathsf{D}_l \qquad \frac{\Theta,F;\Gamma \Downarrow F \vdash R}{\Theta,F;\Gamma \Uparrow \cdot \vdash \cdot \Uparrow R}\ \mathsf{Du} \qquad \frac{\Theta;\Gamma \vdash P \Downarrow}{\Theta;\Gamma \Uparrow \cdot \vdash \cdot \Uparrow P}\ \mathsf{D}_r$$

$$\frac{\Theta;\Gamma \Uparrow P \vdash \cdot \Uparrow R}{\Theta;\Gamma \Downarrow P \vdash R}\ \mathsf{R}_l \qquad \frac{\Theta;\Gamma \Uparrow \cdot \vdash N \Uparrow}{\Theta;\Gamma \vdash N \Downarrow}\ \mathsf{R}_r \qquad \frac{\Theta;C,\Gamma \Uparrow \Delta \vdash \mathcal{R}}{\Theta;\Gamma \Uparrow C,\Delta \vdash \mathcal{R}}\ \mathsf{S}_l \qquad \frac{\Theta;\Gamma \Uparrow \cdot \vdash \cdot \Uparrow D}{\Theta;\Gamma \Uparrow \cdot \vdash D \Uparrow}\ \mathsf{S}_r$$

$$\frac{}{\Theta;A \vdash A \Downarrow}\ \mathsf{I} \qquad \frac{}{\Theta,A;\cdot \vdash A \Downarrow}\ \mathsf{I}_c$$

Here, P is positive, N is negative, C is a negative formula or positive atom, D a positive formula or negative atom, and A is a positive atom. Other formulas are arbitrary. $\mathcal{R}$ denotes $\Delta_1 \Uparrow \Delta_2$ where the union of Δ_1 and Δ_2 contains exactly one formula. In the rules $\forall_r$ and $\exists_l$ the eigenvariable y does not occur free in any formula of the conclusion.

Figure 1 The focused intuitionistic linear sequent calculus ILLF.

- The focused phase begins by choosing, via one of the decide rules D_l, Du or D_r, a formula to be focused on, enabling sequents of the forms (iii) or (iv). Rules are then applied on the focused formula until either: an axiom is reached (in which case the proof ends); the right promotion rule $!_r$ is applied; or a negative formula on the right or a positive formula on the left is derived. At this point, focusing will be lost, and the proof switches to the unfocused phase again.

We will call a *focused step* a focused phase followed by an unfocused one, in a (bottom-up) focused proof.

Observe that the design of the axioms I and I_c in ILLF induces a *positive* polarity to atoms. As it will become clear in Section 3.2, this is necessary for guaranteeing the higher level of adequacy on encodings.

Sequents in ILL will be denoted by $\Gamma \vdash A$. Rules for ILL are the same as in ILLF, only not considering focusing, and the structural rules being substituted by the usual bang left rules: dereliction (D), weakening (W) and contraction (C):

$$\dfrac{\Gamma, F \vdash G}{\Gamma, !F \vdash G}\ \mathsf{D} \qquad \dfrac{\Gamma \vdash G}{\Gamma, !F \vdash G}\ \mathsf{W} \qquad \dfrac{\Gamma, !F, !F \vdash G}{\Gamma, !F \vdash G}\ \mathsf{C}$$

Note that, in ILLF, dereliction is embedded into the bang left ($!_l$) and unbounded decide (Du) rules.

3 Concurrent Constraint Processes as LL Formulas

In this section we shall see how the process-as-formula interpretation can be used for both, providing verification techniques for a process calculus and characterizing different semantics for it in a uniform way. We start by describing the model of computation of *Concurrent Constraint Programming* (CCP) to later show that ILLF provides a suitable framework for interpreting CCP processes.

Concurrent Constraint Programming (CCP) [41, 42, 43, 37] is a model for concurrency based upon the shared-variables communication model. CCP traces its origins back to the ideas of *computing with constraints* [25], *Concurrent Logic Programming* [45] and *Constraint Logic Programming* (CLP) [15]. Different from other models for concurrency, based on point-to-point communication as in CCS [23], the π-calculus [24], CSP [14] among several others, the CCP model focuses on the concept of *partial information*, traditionally referred to as *constraints*. Under this paradigm, the conception of *store as valuation* in the von Neumann model is replaced by the notion of *store as constraint*, and processes are seen as *information transducers*.

The model of concurrency in CCP is quite simple: concurrent agents (or processes) interact with each other and their environment by posting and asking information (i.e., constraints) in a medium, a so-called *store*. As we shall see, CCP processes can be seen as both computing processes (behavioral style) and as formulas in logic (logical declarative style). In particular, we shall see a strong connection between ILL and CCP originally developed in [11] and later refined in [34].

3.1 Constraint system and processes

We start by defining the language of processes and constraints. The type of constraints processes may act on is not fixed but parametric in a constraint system. Such systems can be formalized as Scott information systems [44] as in [40], or they can be built upon a suitable fragment of logic e.g. as in [46, 11, 26]. Here we shall follow the second approach. More precisely, a constraint system is a tuple $\mathbf{C} = (\mathcal{C}, \models_\Delta)$ where the set of constraints $\mathcal{C}$ is built from a first-order signature and the grammar

$$F ::= \mathtt{true} \mid A \mid F \wedge F \mid \exists \overline{x}.F$$

where A is an atomic formula. We shall use c, c', d, d', etc, to denote elements in $\mathcal{C}$. The entailment relation $\models_\Delta$ is parametric on a set of non-logical axioms Δ of the form $\forall \overline{x}.[c \supset c']$ where all free variables in c and c' are in $\overline{x}$. We say that d *entails* c, written as $d \models_\Delta c$, iff the sequent $\Delta, d \vdash c$ is provable in intuitionistic logic (IL). Intuitively, the entailment relation specifies inter-dependencies between constraints: $c \models_\Delta d$ means that the information d can be deduced from the information represented by c, e.g. $x > 42 \models_\Delta x > 0$.

The constraint store, shared by processes, is a conjunction of constraints and `true` denotes the empty store. The existential quantifier is used to specify variable hiding.

Processes are built from constraint as follows:

$$P, Q ::= \mathbf{tell}(c) \mid \sum_{i \in I} \mathbf{ask}\ c_i\ \mathbf{then}\ P_i \mid P \parallel Q \mid (\mathbf{local}\, x)\, P \mid p(\overline{x})$$

A process $\mathbf{tell}(c)$ adds the constraint c to the store, thus incrementing the information in it. The guarded choice $\sum_{i \in I} \mathbf{ask}\ c_i\ \mathbf{then}\ P_i$, where I is a finite set of indexes, chooses non-deterministically one of the processes P_j whose guard c_j can be deduced from the current store. If none of the guards can be deduced, this process remains blocked until more information is added. Hence, ask agents implement a synchronization mechanism based on entailment of constraints. The interleaved parallel composition of P and Q is denoted as $P \parallel Q$. The agent $(\mathbf{local}\, x)\, P$ behaves as P and binds the variable x to be local to it. Finally, given a possibly recursive process definition $p(\overline{y}) \stackrel{\Delta}{=} P$, where all free variables of P are in the set of pairwise distinct variables $\overline{y}$, the process $p(\overline{x})$ evolves into $P[\overline{x}/\overline{y}]$.

The operational semantics of CCP is given by the transition relation $\gamma \longrightarrow \gamma'$ satisfying the rules in Figure 2. Here we follow the semantics in [11] and a *configuration* γ is a triple of the form $(X; \Gamma; c)$, where c is a constraint specifying the store, Γ is a multiset of processes, and X is the set of hidden (local) variables of c and Γ. The multiset $\Gamma = P_1, P_2, \ldots, P_n$ represents the process $P_1 \parallel P_2 ... \parallel P_n$. We shall indistinguishably use both notations to denote parallel composition of processes.

Processes are quotiented by a structural congruence relation $\cong$ satisfying: (1) $P \cong Q$ if they differ only by a renaming of bound variables (alpha-conversion); (2) $P \parallel Q \cong Q \parallel P$; and (3) $P \parallel (Q \parallel R) \cong (P \parallel Q) \parallel R$. Furthermore, $\Gamma = \{P_1, ..., P_n\} \cong \{P_1', ..., P_n'\} = \Gamma'$ iff $P_i \cong P_i'$ for all $1 \leq i \leq n$. Finally, $(X; \Gamma; c) \cong (X'; \Gamma'; c')$ iff $X = X'$, $\Gamma \cong \Gamma'$ and $c \equiv_\Delta c'$ (i.e., $c \models_\Delta c'$ and $c' \models_\Delta c$).

Rules $\mathrm{R_T}$ and $\mathrm{R_C}$ are self-explanatory. Rule $\mathrm{R_{EQUIV}}$ says that structurally congruent processes have the same transitions. Rule $\mathrm{R_L}$ adds the variable x to the set of variables X when it is fresh (otherwise, Rule $\mathrm{R_{EQUIV}}$ can be used to apply alpha conversion). The rule $\mathrm{R_A}$ says that the process $\sum_{i \in I} \mathbf{ask}\ c_i\ \mathbf{then}\ P_i$ evolves into P_j if the current store d entails c_j.

▶ **Definition 1** (Observables). *Let $\longrightarrow^*$ be the reflexive and transitive closure of $\longrightarrow$. If $(X; \Gamma; d) \longrightarrow^* (X'; \Gamma'; d')$ and $\exists X'.d' \models_\Delta c$ we write $(X; \Gamma; d) \Downarrow_c$. If $X = \emptyset$ and $d =$ `true` we simply write $\Gamma \Downarrow_c$.*

Intuitively, if P is a process then $P \Downarrow_c$ says that P outputs c under input `true`.

3.2 Interpretation and adequacy

We shall present different encodings for processes ($\mathcal{P}[\![\cdot]\!]$) and constraints ($\mathcal{C}[\![\cdot]\!]$) as formulas in ILL. Our goal is to show that the outputs of a process P can be characterized by proofs in ILLF. More precisely, we shall show that P outputs c iff a sequent of the form $\mathcal{P}[\![\Psi]\!], \mathcal{C}[\![\Delta]\!] \ : \ \cdot \Uparrow \mathcal{P}[\![P]\!] \vdash \mathcal{C}[\![c]\!] \otimes \top \Uparrow$ is provable in ILLF, where Ψ is a set of process definitions and Δ is the set of non-logical axioms in the constraint system. Note the use of $\top$: we shall erase the formulas corresponding to processes that were not executed. Below, we will see how to tune the process interpretation to get the highest level of adequacy possible.

$$\frac{(X;\Gamma;c)\cong(X';\Gamma';c')\longrightarrow(Y';\Delta';d')\cong(Y;\Delta;d)}{(X;\Gamma;c)\longrightarrow(Y;\Delta;d)}\ \mathrm{R_{EQUIV}}$$

$$\frac{}{(X;\mathbf{tell}(c),\Gamma;d)\longrightarrow(X;\Gamma;c\wedge d)}\ \mathrm{R_T}\qquad\frac{d\models_\Delta c_j}{\langle X,\sum_{i\in I}\mathbf{ask}\ c_i\ \mathbf{then}\ P_i,\Gamma,d\rangle\longrightarrow\langle X,P_j,\Gamma,d\rangle}\ \mathrm{R_A}$$

$$\frac{}{(X;(\mathbf{local}\,x)\,P,\Gamma;d)\longrightarrow(X\cup\{x\};P,\Gamma;d)}\ \mathrm{R_L}\qquad\frac{p(\overline{x})\stackrel{\Delta}{=}P}{(X;p(\overline{y}),\Gamma;d)\longrightarrow(X;P[\overline{y}/\overline{x}],\Gamma;d)}\ \mathrm{R_C}$$

Figure 2 Operational semantics of CCP. In $\mathrm{R_L}$, $x\notin X$ and it does not occur free in Γ nor in d.

▶ **Definition 2.** *Constraints and axioms in CCP are encoded in* ILL *as follows:*

$$\begin{array}{lclclclclcl}
\mathcal{C}[\![\mathit{true}]\!] & = & 1 & & \mathcal{C}[\![A]\!] & = & !\,A & & \mathcal{C}[\![F_1\wedge F_2]\!] & = & \mathcal{C}[\![F_1]\!]\otimes\mathcal{C}[\![F_2]\!]\\
\mathcal{C}[\![\exists x.F]\!] & = & \exists x.\mathcal{C}[\![F]\!] & & \mathcal{C}[\![\forall\overline{x}.(c\supset d)]\!] & = & \forall\overline{x}.(\mathcal{C}[\![c]\!]\multimap\mathcal{C}[\![d]\!]) & & & &
\end{array}$$

For the processes and process definition, the interpretation is the following:

$$\begin{array}{lcllcl}
\mathcal{P}[\![\mathbf{tell}(c)]\!]] & = & \mathcal{C}[\![c]\!] & \mathcal{P}[\![P\parallel Q]\!] & = & \mathcal{P}[\![P]\!]\otimes\mathcal{P}[\![Q]\!]\\
\mathcal{P}[\![\sum_{i\in I}\mathbf{ask}\ c_i\ \mathbf{then}\ P_i]\!] & = & \operatorname*{\&}_{i\in I}(\mathcal{C}[\![c_i]\!]\multimap\mathcal{P}[\![P_i]\!]) & \mathcal{P}[\![(\mathbf{local}\,x)\,P]\!] & = & \exists x.\mathcal{P}[\![P]\!]\\
\mathcal{P}[\![p(\overline{y})]\!] & = & p(\overline{y}) & \mathcal{P}[\![p(\overline{x})\stackrel{\Delta}{=}P]\!] & = & \forall\overline{x}.(p(\overline{x})\multimap\mathcal{P}[\![P]\!])
\end{array}$$

Since the store in CCP is monotonic, i.e., constraints cannot be removed, we mark atomic formulas with a bang (to be stored in the unbounded context). Parallel composition is identified with multiplicative conjunction and the act of choosing one of the branches in a non-deterministic choice is specified with additive conjunction. The action of querying the store in ask agents is specified with a linear implication. Similarly, the unfolding of a process definition is guarded by the atomic proposition $p(\vec{y})$ (denoting the call).

If Γ is a set of constraints, or axioms of the form $\forall\overline{x}.[c\supset c']$, we write $\mathcal{C}[\![\Gamma]\!]$ to denote the set $\{\mathcal{C}[\![d]\!]\mid d\in\Gamma\}$. A similar convention applies for $\mathcal{P}[\![\cdot]\!]$. Moreover, $!\Gamma=\{!F\mid F\in\Gamma\}$.

▶ **Theorem 3** (Adequacy – ILL [11]). *Let $(\mathcal{C},\models_\Delta)$ be a constraint system, P be a process and Ψ be a set of process definitions. Then, for any constraint c, $P\Downarrow_c$ iff there is a proof of the sequent $!\,\mathcal{P}[\![\Psi]\!],!\,\mathcal{C}[\![\Delta]\!],\mathcal{P}[\![P]\!]\vdash\mathcal{C}[\![c]\!]\otimes\top$ in* ILL*. The level of adequacy is* **FCP***.*

Without focusing (as originally done in [11]), the proof of this theorem is not straightforward and a low level of adequacy is obtained: there may be logical steps not corresponding to any operational step and vice-versa. Let us focus first in the case where logical steps do not correspond to the operational ones. We will come back to the other direction later.

Consider the two derivations bellow.

$$\frac{\begin{array}{c}\pi_1\\ \Gamma,c_1\multimap F_1\vdash d\end{array}}{\Gamma,(c_1\multimap F_1)\,\&\,(c_2\multimap F_2)\vdash d}\ \&_l\qquad\qquad\frac{\begin{array}{c}\pi_2\\ \Gamma_1,F_1\vdash d\end{array}\quad\begin{array}{c}\pi_3\\ \Gamma_2\vdash c_1\end{array}}{\Gamma_1,\Gamma_2,c_1\multimap F_1\vdash d}\ \multimap_l\qquad(1)$$

In the first, one of the branches is chosen but, in π_1, it could be the case that c_1 is never proved (and F_1 is never added to the context). This is not the intended meaning in Rule $\mathrm{R_A}$, that first checks the entailment of c_j to *immediately* add the corresponding process P_j to the context. In the second example, π_3 could contain sub-derivations that have nothing to do with the proof of the guard c_1. For instance, process definitions could be unfolded or other processes could be executed. This would correspond, operationally, to the act of triggering an ask process **ask** c **then** P with no guarantee that its guard c will be derivable only from the set of non-logical axioms Δ and the current store. For instance, it may be the case, in π_3, that c_1 will be later produced by a process Q such that $\mathcal{P}[\![Q]\!]\in\Gamma_2$. This is clearly not allowed by the operational semantics.

Let's now put focusing into play. An inspection in the encoding reveals that the fragment of ILL used is restricted to the following grammar:

$$
\begin{array}{lcll}
G & := & 1 \mid\ !\,A \mid G \otimes G \mid \exists x.G & \textit{Guards and Goals} \\
P & := & G \mid P \otimes P \mid P \,\&\, P \mid G \multimap P \mid \exists x.P \mid p(\bar{t}) & \textit{Processes} \\
PD & := & \forall \overline{x}.p(\overline{x}) \multimap P. & \textit{Process Definitions}
\end{array}
$$

where A is an atomic formula (constraint) in $\mathcal{C}$ and p (a process identifier) is also atomic but $p \notin \mathcal{C}$. In any derivation, the only formulas that can appear on the right are guards/goals G and heads p. The other formulas, including processes, process definitions and axioms, appear on the left. Hence, only instances of the unfocused rules $1_l, \otimes_l, \exists_l, !_l, \top_r$ and the focused rules $\otimes_r, \multimap_l, \exists_r, !_r, \&_l, \forall_l$ are used.

Observe that formulas G, p are *strictly positive*. Thus, focusing on such a formula on the right either forces finishing the proof, or the formula will be *entirely* decomposed into formulas of the shape 1 or $!\,A$. This means that a proof of A can use only the theory Δ, the encoding of constraints and process definitions (since all of them are unbounded). In fact, we can show that the encoding of process definitions can be weakened (since calls of the form $p(\vec{y})$ are necessarily stored in the linear context). Hence, when a goal is focused on, it must be completely decomposed, and the atomic constraints must be proved only from the current store and the non-logical axioms.

Formulas occurring on the left of sequents can be positive or negative. *Positive formulas* on the left (that cannot be focused on) come from the interpretation of *tell*, *parallel composition* and *locality* that do not need any interaction with the context. Note, for instance, that the formula $\exists x.\,!\,G_1 \otimes !\,G_2$, resulting from the encoding of $\mathbf{tell}(\exists x.G_1 \wedge G_2)$, can be entirely decomposed in an unfocused phase using the rules $\otimes_l$, $\exists_l$ and $!_l$. On the other hand, *negative formulas* on the left (that can be chosen for focusing) come from the encodings of *guarded choices* and *process definitions*. They *do need* to interact with the environment, either for choosing a path to follow (in non-deterministic choices), or waiting for a guard to be available (in asks or procedure calls).

Due to completeness of focusing [1], Theorem 3 trivially holds if we replace in it ILL with ILLF. But using directly the focused system, the proof of the theorem becomes simpler. For instance, it is a routine exercise to show that non-logical axioms permute up, and it is always possible to apply them at the top of proofs. Moreover, situations as the ones described after the derivations in Equation (1) are not longer valid in the focused system: focusing over $c_1 \multimap F_1$ implies immediately proving c_1 (from the logical axioms and accumulated constraints), thus reflecting *exactly* the operational semantics of CCP.

▶ **Example 4.** Consider a *community coffee machine*, which is triggered by the insertion of a coin, always available at the side of the machine. When the user inserts the coin, the machine delivers a coffee and returns the coin, which will be available for the next user. This machine can be specified as the CCP process

$$P = \mathbf{tell}(\textit{coin}) \parallel \texttt{m()} \text{ where } \texttt{m()} \stackrel{\Delta}{=} \mathbf{ask}\ \textit{coin}\ \mathbf{then}\ (\mathbf{tell}(\textit{coffee}) \parallel \texttt{m()})$$

Hence, $P \Downarrow_c$, where $c = \textit{coin} \wedge \textit{coffee}$:

$$\langle \emptyset, P, \texttt{true}\rangle \longrightarrow \langle \emptyset, \texttt{m()}, \textit{coin}\rangle \longrightarrow \langle \emptyset, \mathbf{tell}(\textit{coffee}) \parallel \texttt{m()}, \textit{coin}\rangle \longrightarrow \langle \emptyset, \texttt{m()}, \textit{coin} \wedge \textit{coffee}\rangle$$

On the other hand, the sequent $\mathcal{P}[\![P]\!] \vdash \mathcal{C}[\![c]\!] \otimes \top$ has the following focused proof

$$
\dfrac{
  \dfrac{}{\mathit{coin}, \mathcal{P}[\![\mathtt{m()}]\!]; \cdot \vdash\, !\,\mathit{coin} \Downarrow}\;!_r, \mathsf{S}_r, \mathsf{D}_r, \mathsf{I}
  \qquad
  \dfrac{
    \dfrac{
      \dfrac{}{\mathit{coffee}, \mathit{coin}, \mathcal{P}[\![\mathtt{m()}]\!]; \cdot \vdash\, !\,\mathit{coffee} \Downarrow}\;!_r, \mathsf{S}_r, \mathsf{D}_r, \mathsf{I}
      \quad
      \dfrac{}{\mathit{coffee}, \mathit{coin}, \mathcal{P}[\![\mathtt{m()}]\!]; \cdot \vdash\, !\,\mathit{coin} \Downarrow}\;!_r, \mathsf{S}_r, \mathsf{D}_r, \mathsf{I}
    }{\mathit{coffee}, \mathit{coin}, \mathcal{P}[\![\mathtt{m()}]\!]; \mathtt{m()} \vdash (!\,\mathit{coffee} \otimes !\,\mathit{coin}) \otimes \top \Downarrow}\;\otimes_r, \top_r
  }{\mathit{coin}, \mathcal{P}[\![\mathtt{m()}]\!]; \cdot \Downarrow !\,\mathit{coffee} \otimes \mathtt{m()} \vdash (!\,\mathit{coffee} \otimes !\,\mathit{coin}) \otimes \top}\;\mathsf{R}_l, \otimes_l, !_l, \mathsf{D}_r
}{
  \dfrac{
    \dfrac{\mathit{coin}, \mathcal{P}[\![\mathtt{m()}]\!]; \mathtt{m()} \Downarrow \mathtt{m()} \multimap (!\,\mathit{coin} \multimap (!\,\mathit{coffee} \otimes \mathtt{m()})) \vdash (!\,\mathit{coffee} \otimes !\,\mathit{coin}) \otimes \top}{\mathit{coin}, \mathcal{P}[\![\mathtt{m()}]\!]; \mathtt{m()} \Uparrow \cdot \vdash \cdot \Uparrow (!\,\mathit{coffee} \otimes !\,\mathit{coin}) \otimes \top}\;\mathsf{D}_l
  }{\mathcal{P}[\![\mathtt{m()}]\!]; \cdot \Uparrow !\,\mathit{coin} \otimes \mathtt{m()} \vdash \cdot \Uparrow (!\,\mathit{coffee} \otimes !\,\mathit{coin}) \otimes \top}\;\otimes_l, !_l, \mathsf{S}_l
}\;\multimap_l, \mathsf{I}
$$

Bottom up, we introduce the tell process in the unfocused phase. Then, after focusing on the encoding of the ask agents, the guard *coin* is deduced (left-most derivation), and the token *coin* is stored into the classical context, thus reflecting the final configuration in the execution of the process.

Unfortunately, even with focusing, the adequacy level continues to be **FCP**. In fact, the focusing discipline causes that some CCP computations do not have a corresponding proof in ILLF. To see that, consider the following process

$$
\begin{array}{rcl}
P & = & \textbf{tell}(a \wedge b) \parallel \textbf{ask}\ a\ \textbf{then ask}\ b\ \textbf{then tell}(\mathtt{ok}) \parallel \\
 & & \qquad\qquad \textbf{ask}\ b\ \textbf{then ask}\ a\ \textbf{then tell}(\mathtt{ok}')
\end{array}
$$

We denote the two external ask agents in P as Q_1 and Q_2 respectively. The operational semantics dictates that there are three possible transitions leading to the final store $d = a \wedge b \wedge \mathtt{ok} \wedge \mathtt{ok}'$. All such transitions start by executing $\textbf{tell}(a \wedge b)$:

$$
\begin{array}{llll}
\mathtt{Trace\ 1:} & \langle \emptyset, P, \mathtt{true} \rangle & \longrightarrow & \langle \emptyset, Q_1 \parallel Q_2, a \wedge b \rangle \longrightarrow \langle \emptyset, \textbf{ask}\ b\ \textbf{then tell}(\mathtt{ok}) \parallel Q_2, a \wedge b \rangle \\
 & & \longrightarrow & \langle \emptyset, \textbf{tell}(\mathtt{ok}) \parallel Q_2, a \wedge b \rangle \longrightarrow \langle \emptyset, Q_2, a \wedge b \wedge \mathtt{ok} \rangle \longrightarrow^* \langle \emptyset, \cdot, d \rangle \not\longrightarrow \\
\mathtt{Trace\ 2:} & \langle \emptyset, P, \mathtt{true} \rangle & \longrightarrow & \langle \emptyset, Q_1 \parallel Q_2, a \wedge b \rangle \longrightarrow \langle \emptyset, Q_1 \parallel \textbf{ask}\ a\ \textbf{then tell}(\mathtt{ok}'), a \wedge b \rangle \\
 & & \longrightarrow & \langle \emptyset, Q_1 \parallel \textbf{tell}(\mathtt{ok}'), a \wedge b \rangle \longrightarrow \langle \emptyset, Q_1, a \wedge b \wedge \mathtt{ok}' \rangle \longrightarrow^* \langle \emptyset, \cdot, d \rangle \not\longrightarrow \\
\mathtt{Trace\ 3:} & \langle \emptyset; P; \mathtt{true} \rangle & \longrightarrow & \langle \emptyset; Q_1 \parallel Q_2; a \wedge b \rangle \longrightarrow \langle \emptyset; \textbf{ask}\ b\ \textbf{then tell}(\mathtt{ok}) \parallel Q_2; a \wedge b \rangle \\
 & & \longrightarrow & \langle \emptyset, \textbf{ask}\ b\ \textbf{then tell}(\mathtt{ok}) \parallel \textbf{ask}\ a\ \textbf{then tell}(\mathtt{ok}'), a \wedge b \rangle \\
 & & \longrightarrow & \langle \emptyset, \textbf{tell}(\mathtt{ok}) \parallel \textbf{ask}\ a\ \textbf{then tell}(\mathtt{ok}'), a \wedge b \rangle \longrightarrow \langle \emptyset, \textbf{tell}(\mathtt{ok}) \parallel \textbf{tell}(\mathtt{ok}'), a \wedge b \rangle \\
 & & \longrightarrow^* & \langle \emptyset, \cdot, d \rangle
\end{array}
$$

`Trace 1` and `Trace 2` correspond exactly to a different focused proof of the sequent $\mathcal{P}[\![P]\!] \vdash \mathcal{C}[\![d]\!]$: one focusing first on $\mathcal{P}[\![Q_1]\!]$ and the other focusing first on $\mathcal{P}[\![Q_2]\!]$. On the other hand, `Trace 3` corresponds to an *interleaved* execution of Q_1 and Q_2. We note that such a trace does not have any correspondent derivation in ILLF. In fact, since $\multimap$ is a negative connective, focusing on $\mathcal{C}[\![Q_1]\!]$ will decompose the formula $!a \multimap !b \multimap !\mathtt{ok}$ producing the focused formula $!b \multimap !\mathtt{ok}$, which is still negative. Hence focusing cannot be lost and the inner ask has to be triggered.

This example shows something interesting: although the formulas $F \otimes G \multimap H$ and $F \multimap G \multimap H$ are logically equivalent, they are operationally different when *concurrent computations* are considered. In fact, if we allow processes to consume constraints as the linear version of CCP in [11], an interleaving execution as the one in `Trace 3` may not output the constraint `ok`, since the two agents are competing for the same resources.

In order to recover interleaving executions as the one in `Trace 3`, *logical delays* [28] can be introduced.

▶ **Definition 5.** *The positive and negative delay operators $\delta^+(\cdot), \delta^-(\cdot)$ are defined as $\delta^+(F) = F \otimes 1$ and $\delta^-(F) = 1 \multimap F$ respectively.*

Observe that $\delta^+(F) \equiv \delta^-(F) \equiv F$, hence delays can be used in order to replace a formula with a provably equivalent formula of a given polarity.

We define the encoding $\mathcal{P}[\![\cdot]\!]_+$ as $\mathcal{P}[\![\cdot]\!]$ but replacing the following cases:

$$\begin{array}{lcl} \mathcal{P}[\![\sum_{i\in I} \textbf{ask } c_i \textbf{ then } P_i]\!]_+ & = & \mathop{\&}_{i\in I} (\mathcal{C}[\![c_i]\!] \multimap \delta^+(\mathcal{P}[\![P_i]\!]_+)) \\ \mathcal{P}[\![p(\overline{x}) \stackrel{\Delta}{=} P]\!]_+ & = & \forall \overline{x}.p(\overline{x}) \multimap \delta^+(\mathcal{P}[\![P]\!]_+) \end{array}$$

The use of delays forces the focused phase to end, e.g., once the guard of the ask agent is entailed. In this encoding, we can prove a stronger adequacy theorem.

▶ **Theorem 6** (Strong adequacy [34]). *Let $(\mathcal{C}, \models_\Delta)$ be a constraint system, P be a process and Ψ be a set of process definitions. Then, for any constraint c,*

$$P \Downarrow_c \text{ iff there is a proof of the sequent } \mathcal{P}[\![\Psi]\!]_+, \mathcal{C}[\![\Delta]\!]; \cdot \Uparrow \mathcal{P}[\![P]\!]_+ \vdash \cdot \Uparrow \mathcal{C}[\![c]\!] \otimes \top$$

in ILLF. *The adequacy level is* **FCD**.

Now derivations in logic have a one-to-one correspondence with traces of a computation in a CCP program.

It is possible to modify the encoding to introduce negative actions (*tell*, *parallel* and *local*) during a focused phase (thus counting them as a focused step). For that, it suffices to introduce, in the encoding, negative delays $\delta^-(F)$. By using a multi-focusing systems [38], maximal parallelism semantics [9] - where all the enabled agents must all proceed in one step - can be also captured. Finally, if recursive definitions are interpreted as fixed points, more interesting properties of infinite computations can be specified and proved. See [34] for further details.

4 LL with multi-modalities

A careful analysis of the rules for the exponential ! in Figure 1 reveals that this connective has a differentiated behavior w.r.t. the other ones. In fact, ! is the only operator having a positive/negative behavior: the application of the right rule ($!_r$) immediately breaks focusing. Also, this is the only rule in ILLF that is *context dependent*, in the sense that it demands the linear context Γ to be empty in order to be applied.

This distinguished character of the exponential in linear logic is akin to the behavior found in *modal connectives*. In particular, the connective ! is not *canonical*, in the sense that, if we label ! with different colors, say b (for blue – $!^b$) and r (for red – $!^r$), but with the same introduction rules, then it is not possible to prove, in the resulting proof system, the equivalence $!^r A \equiv !^b A$ for an arbitrary formula A, where $H \equiv G$ denotes the formula $(H \multimap G) \,\&\, (G \multimap H)$. Not surprisingly, this exercise would have a different outcome for any other linear logic connective. For instance, if we construct a proof system with two labeled connectives, e.g., $\otimes^r$ and $\otimes^b$, together with their introduction rules, then it would be possible to prove $A \otimes^b B \equiv A \otimes^r B$ for any A and B. This opens the possibility of defining new connectives: the colored exponentials, known as *subexponentials* [8].

4.1 Linear logic with subexponentials

Linear logic with subexponentials (SELL)[4] shares with intuitionistic linear logic all its connectives except the exponential: instead of having a single !, SELL may contain as many subexponentials, written $!^a$ for a label (or color) a, as one needs.

[4] Although in this paper we are mostly interested in the intuitionistic version of SELL, it was proven in [3] that classical and intuitionistic subexponential logics are equally expressive. Hence we will abuse the notation and use SELL for intuitionistic linear logic system with subexponentials.

Such labels are organized in a pre-order, giving rise to a *subexponential signature* $\Sigma = \langle I, \preceq, U \rangle$, where I is a set of labels, $U \subseteq I$ is a set specifying which subexponentials behave classically (i.e., those labels that allow for weakening and contraction), and $\preceq$ is a pre-order among the elements of I. We shall use $a, b, \ldots$ to range over elements in I, and we will assume that $\preceq$ is upwardly closed with respect to U, i.e., if $a \in U$ and $a \preceq b$, then $b \in U$.

The division of *unbounded* ($a \in U$) and linear or *bounded* ($a \notin U$) subexponentials induces also a partition of the subexponential context Θ, which is split into two: a set Θ^u and a multiset Θ^b of labeled formulas, having the form

$$\Theta^u = \{a_1 : \Theta^u_1, \ldots, a_n : \Theta^u_n\} \qquad \Theta^b = \{b_1 : \Theta^b_1, \ldots, b_m : \Theta^b_m\}$$

The formulas in Θ^u_i are under the scope of the unbounded subexponential $!^{a_i}$, and formulas in Θ^b_j are under the scope of the bounded subexponential $!^{b_j}$. The linear context Γ continues containing only negative or atomic formulas, as in ILLF.

The focused proof system SELLF [28] is constructed by adding all the rules for the intuitionistic linear logic connectives as shown in Figure 1,[5] except for the exponentials. The rules for subexponentials are the following:

- A formula F under the scope of $!^a$ is stored in the exponential context Θ accordingly: if a is unbounded/bounded, then F is added to the set/multiset Θ_a, which is created if it does not exist. This action is represented by $\Theta \uplus \{a : F\}$.

$$\frac{\Theta \uplus \{a : F\}; \Gamma \Uparrow \Delta \vdash \mathcal{R}}{\Theta; \Gamma \Uparrow !^a F, \Delta \vdash \mathcal{R}}\ !^a{}_l$$

- The unbounded decide rule in ILLF is split into bounded and unbounded versions, depending of the nature of the subexponential.

$$\frac{\Theta^u, \Theta^b; \Gamma \Downarrow F \vdash R}{\Theta^u, \Theta^b \uplus \{a : F\}; \Gamma \Uparrow \cdot \vdash \cdot \Uparrow R}\ \mathsf{Db} \qquad \frac{\Theta^u \uplus \{a : F\}, \Theta^b; \Gamma \Downarrow F \vdash R}{\Theta^u \uplus \{a : F\}, \Theta^b; \Gamma \Uparrow \cdot \vdash \cdot \Uparrow R}\ \mathsf{Du}$$

- The promotion rule has the form

$$\frac{\Theta^u_{\succeq a}, \Theta^b; \cdot \Uparrow \cdot \vdash F \Uparrow}{\Theta^u, \Theta^b; \cdot \vdash !^a F \Downarrow}\ !^a{}_r$$

  with the proviso that, for all $b_j : \Theta^b_j$ in Θ^b, it must be the case that $a \preceq b_j$. In the premise of the rule, $\Theta^u_{\succeq a} \subseteq \Theta^u$ contains only elements of the form $a_i : \Theta^u_i$ where $a \preceq a_i$ (the other contexts are weakened). That is, $!^a F$ is provable only if F can be proved in the presence of subexponentials greater than a.

It is known that subexponentials greatly increase the expressiveness of the system when compared to linear logic. For instance, subexponentials can be used to represent contexts of proof systems [32], to mark the epistemic state of agents [27], or to specify locations in sequential computations [28]. The key difference is that, while linear logic has only seven logically distinct prefixes of bangs and question-marks (? is the dual of !), SELL allows for an unbounded number of such prefixes, e.g., $!^i$, or $!^i ?^j$. As we show later, by using different prefixes, we can interpret subexponentials in more creative ways, such as linear constraints, epistemic modalities or preferences. The interested reader can also check in [30, 35, 31] the interpretation of subexponentials as temporal units, and the study of dynamical subexponentials in distributed systems.

[5] Taking the extra-care of splitting the bounded context Θ^b for the multiplicative rules $\multimap_l$ and $\otimes_r$.

The organization of subexponentials in pre-orders brings at least two interesting aspects that can be further investigated: what kind of refinements of the proof system can be obtained by adopting richer algebraic structures for subexponentials (Section 4.2 below); and what is the proof-theoretic notion of quantification over modalities (Section 4.3 below).

Being able to quantify over subexponentials is important, e.g., for specifying properties that are valid in an unbounded number of locations or agents. It is also crucial for establishing a certain notion of *mobility*, or *permissibility* of resources, that can be available, e.g., iff they are marked with a label of some specific sort. But one has to be careful here: the pre-order structure is a minimal requirement in subexponential signatures in order to guarantee the *cut-elimination* property [8]. Since, in the presence of quantifiers, proving cut-elimination requires *substitution lemmas*, a naive approach of exchanging labels could invalidate such results (see [31] for an extensive discussion on the topic).

On the other hand, if we move above the pre-order minimality and consider, e.g., $\wedge$-semi-lattices as subexponential structures, then the side condition in the promotion rule, $a \preceq a_i$ for all $a_i \in \Theta_{\geq a}$, is equivalent to $a \preceq \bigwedge_i a_i$. And this reflects certain kinds of preferences, as explained next.

4.2 Richer subexponential signatures

We now explore a refinement of SELLF, where richer structures are considered as subexponential signatures. For that, we shall use an algebraic structure that defines a mean to compare ($\preceq$) and accumulate ($\bullet$) values.

More precisely, a complete lattice monoid [12] is a tuple $CLM = \langle \mathcal{D}, \preceq, \bullet \rangle$ such that $\langle \mathcal{D}, \preceq \rangle$ is a complete lattice, $\perp$ and $\top$ are, respectively, the least and the greatest elements of $\mathcal{D}$ and $\{\mathcal{D}, \bullet, \top\}$ is a abelian monoid. Moreover, $\bullet$ distributes over lubs, i.e., for all $v \in \mathcal{D}$ and $X \subseteq \mathcal{D}$, $v \bullet \sqcup X = \sqcup\{v \bullet x \mid x \in X\}$. Due to distributivity, $\bullet$ is monotone and decreasing: $a \bullet b \preceq a$.

Observe that, if the SELL signature structure is a lattice, then $a \preceq \{b, c\}$ is equivalent to $a \preceq glb(b, c)$. Moreover, in the presence of $\bullet$, promotion can be refined so to consider the combination of values as follows.

Given a SELL signature $\Sigma = \langle \mathcal{D}, \preceq, U \rangle$ with $\langle \mathcal{D}, \preceq, \bullet \rangle$ a CLM, the promotion rule $!^a{}_{r\bullet}$ is defined as:

$$\frac{\Theta^u_{\geq a}, \Theta^b; \cdot \Uparrow \cdot \vdash F \Uparrow}{\Theta^u, \Theta^b; \cdot \vdash !^a F \Downarrow} \; !^a{}_{r\bullet}, \text{ provided } a \preceq \bullet\{a_i, b_j\}$$

Note that, if the CLM is $\bullet$-*idempotent* (i.e. $a \bullet a = a$), then $glb(a, b) = a \bullet b$, and the above rule coincides with SELLF's promotion rule.

▶ **Example 7.** Consider the signature $\Sigma = \langle \mathcal{D}, \preceq, \mathcal{D} \rangle$, with the following instances of CLM.

- $\langle \{\texttt{pub}, \texttt{sec}\}, \preceq, \wedge \rangle$, where `pub` and `sec` represent public and private information, respectively. The ordering is $\texttt{pub} \prec \texttt{sec}$ and $a \wedge b = \texttt{sec}$ iff $a = b = \texttt{sec}$. Hence, any proof of $\Theta; \cdot \vdash !^{\mathsf{sec}} F \Downarrow$ does not make use of any *public* information.
- $\langle [0, 1], \leq_{\mathbb{R}}, \min \rangle$ (fuzzy), where $[0, 1] \subset \mathbb{R}$, and $\leq_{\mathbb{R}}$ is the usual order in $\mathbb{R}$. In this case, we can interpret $!^{0.2} c$ as "c is believed with preference 0.2". Note that the sequent $!^{0.2} c \otimes !^{0.7} d \vdash !^a (c \otimes d)$ is provable only if $a \leq_{\mathbb{R}} 0.2$.
- $\langle [0, 1], \leq_{\mathbb{R}}, \times \rangle$ (probabilistic), where $\times$ is the multiplication operator in $\mathbb{R}$. This is a non-idempotent CLM, and the sequent $!^{0.2} c \otimes !^{0.7} d \vdash !^a (c \otimes d)$ is provable only if $a \leq_{\mathbb{R}} 0.14$.

In [39] we have showed that this new version of the promotion rule is not at all ad-hoc. The resulting system, SELLS, is a smooth extension of ILLF and it is a closed subsystem of SELLF, which is strict when non-idempotent CLMs are considered. Hence SELLS inherits all SELLF good properties such as cut-elimination.

The SELLS system has inspired the development of new CCP-based calculi where processes can tell and ask soft constraints, understood as formulas of the form $!^a c$ where a is an element of a given CLM [39]. Also, since the underlying logic is the same, it is possible to obtain adequate interpretations of processes as formulas as the ones in Section 3.2. More interestingly, it is also possible to combine, in a uniform way, different modalities [35], all of them grounded on linear logic principles. Some of these modalities will be explored in Section 5.

4.3 Subexponential Quantifiers

This section introduces the focused system $\mathsf{SELLF}^{\Cap}$, containing two novel connectives $\Cap$ and $\Cup$, representing, respectively, a universal and existential quantifiers over *subexponentials*.[6]

As mentioned in Section 4.1, in order to guarantee cut-elimination of the resulting system, the substitution of subexponentials in the rules for quantification should be done carefully. As showed in [31], it is enough to require that labels are substituted, bottom-up, for *smaller* ones. Also, the possibility of creating new labels dynamically implies that there should be two sorts of labels: constants and variables. This justifies the next definition.

▶ **Definition 8.** *Given a pre-order $(I, \preceq)$ and $a \in I$, the* ideal generated by a *is the set* $\downarrow a = \{b \in I \mid b \preceq a\}$.

The subexponential signature *of* $\mathsf{SELL}^{\Cap}$ *is the triple $\Sigma = \langle I, \preceq, U\rangle$, where I is a set of* subexponential constants*, $\preceq$ is a pre-order over I and $U \subseteq I$ is the upwardly closed set of* unbounded constants.

The sets of typed subexponential constants *and* typed subexponential variables *are denoted respectively by*

$$\mathcal{T}_\Sigma = \{b : a \mid b \in \downarrow a\} \qquad \mathcal{T}_x = \{l_{x_1} : a_1, \ldots, l_{x_n} : a_n\}$$

where $\{l_{x_1}, \ldots, l_{x_n}\}$ is a disjoint set of subexponential variables*, and $\{a_1, \ldots, a_n\} \subseteq I$ are subexponential constants.*

Formally, only these subexponential constants and variables may appear free in an index of subexponential bangs and question marks.

Sequents in $\mathsf{SELLF}^{\Cap}$ have the same form as in SELLF, with the difference that there is an extra context $\mathcal{T} = \mathcal{T}_\Sigma \cup \mathcal{T}_x$.

The rules for for $\Cap$ and $\Cup$ are the novelty with respect to the focused proof system for SELLF. They behave exactly as the first-order quantifiers: the $\Cap_r$ and $\Cup_l$ belong to the negative phase because they are invertible, while $\Cap_l$ and $\Cup_r$ are positive since they are not invertible.

$$\frac{\mathcal{T} \cup \{l_e : a\}; \Theta; \Gamma \Uparrow \Delta \vdash F[l_e/l_x] \Uparrow}{\mathcal{T}; \Theta; \Gamma \Uparrow \Delta \vdash \Cap l_x : a.F \Uparrow} \ \Cap_r \qquad \frac{\mathcal{T} \cup \{l_e : a\}; \Theta; \Gamma \Uparrow \Delta, F[l_e/l_x] \vdash \mathcal{R}}{\mathcal{T}; \Theta; \Gamma \Uparrow \Delta, \Cup l_x : a.F \vdash \mathcal{R}} \ \Cup_l$$

$$\frac{\mathcal{T}; \Theta; \Gamma \Downarrow F[l/l_x] \vdash R}{\mathcal{T}; \Theta; \Gamma \Downarrow \Cap l_x : a.F \vdash R} \ \Cap_l \qquad \frac{\mathcal{T}; \Theta; \Gamma \vdash F[l/l_x] \Downarrow}{\mathcal{T}; \Theta; \Gamma \vdash \Cup l_x : a.F \Downarrow} \ \Cup_r$$

[6] Some motivation for the symbols $\Cap$ and $\Cup$. The former resembles the symbol for intersection, which is the usual semantics assigned to for all quantifiers, namely, the intersection of all models, while the latter is same for exists and union.

In the left rule of $\Cap$ and the right rule of $\Cup$, l_x is substituted with a subexponential of the right type: $l : b \in \mathcal{T}$, $b \in\downarrow a$. In the rules $\Cap_r$ and $\Cup_l$, a fresh variable l_e of type a is created and added to the context $\mathcal{T}$.

Next, we shall see that the quantifiers allows for encoding, in a modular way, systems dealing with an unbounded number of modalities.

5 Parametric interpretations

This section illustrates how focusing, subexponentials and quantifiers in $\mathsf{SELLF}^{\Cap}$ can be used to give adequate interpretations to CCP calculi featuring different modalities. The interpretation is *modular*: there is only one base logic – $\mathsf{SELL}^{\Cap}$; and *parametric*: each modal flavor of CCP is specified by a signature in SELL having a particular algebraic structure. In this way, processes may be executed and add/query constraints in different *locations*, where the meaning of such locations may vary, for example: spaces of computation, the epistemic state of agents, time units, levels of preferences, etc. But the underline interpretation is the same: locations in CCP become labels in SELL.

Another modular aspect of our process-as-formula interpretation is the organization of the encodings of constraints, processes and process definitions, into non-comparable *families* of subexponentials, so that focusing on an element of a family forces all elements of the other families to be erased during proof search. This ensures the discipline necessary for guaranteeing the highest level of adequacy (**FCD**).

Formally, let M be an underlying set of labels, with least and greatest elements represented by *nil* and ∞ respectively, ordered with a pre-order $\preceq_M$. The families of subexponentials are built with marked copies of elements of M: $\mathfrak{c}(\cdot)$ for constraints, $\mathfrak{p}(\cdot)$ for processes, and $\mathfrak{d}(\cdot)$ for process definitions. The subexponential signature $\Sigma = \langle I, \preceq, U\rangle$ is built from M in the following way:

- The set of labels is: $I = \{l, \mathfrak{c}(l), \mathfrak{p}(l), \mathfrak{d}(l) \mid l \in M\}$; that is, besides the elements in M, we consider three additional distinct copies of the labels, each of them marked with the appropriate family.
- The subexponential pre-order is: $l \preceq l'$ iff $l \preceq_M l'$ and $\mathfrak{f}(l) \preceq \mathfrak{f}(l')$ iff $l \preceq_M l'$ where $\mathfrak{f} \in \{\mathfrak{c}, \mathfrak{p}, \mathfrak{d}\}$; note that subexponentials pertaining to different families are not related.
- The set U of unbounded subexponentials will vary depending on the encoded system.

Constraints and CCP processes are encoded into $\mathsf{SELLF}^{\Cap}$ by using the functions $\mathcal{C}[\![\cdot]\!]_l$ and $\mathcal{P}[\![\cdot]\!]_l$ as in Definition 2, now parametric w.r.t. subexponentials $l \in M$ as follows.[7]

▶ **Definition 9** (General Encoding). *Constraints and axioms of the constraint system are encoded in* $\mathsf{SELL}^{\Cap}$ *as:*

$$\mathcal{C}[\![\mathit{true}]\!]_l = 1 \qquad \mathcal{C}[\![A]\!]_l = !^{\mathfrak{c}(l)} A \qquad \mathcal{C}[\![c_1 \wedge c_2]\!]_l = \mathcal{C}[\![c_1]\!]_l \otimes \mathcal{C}[\![c_2]\!]_l$$

$$\mathcal{C}[\![\exists \overline{x}.c]\!]_l = \exists \overline{x}.\mathcal{C}[\![c]\!]_l \qquad \mathcal{C}[\![\forall \overline{x}.(d \supset c)]\!] = \Cap l_x : \infty.\forall \overline{x}.(\mathcal{C}[\![d]\!]_{l_x} \multimap \mathcal{C}[\![c]\!]_{l_x})$$

[7] We observe that, technically, the encoding functions should also consider subexponential variables. However, the encoded processes/axioms are stored on left contexts, and the left introduction rule for universal quantifiers does not create fresh variables.

The encoding of processes and process definitions is:

$$
\begin{array}{lcl}
\mathcal{P}[\![\mathbf{tell}(c)]\!]_l & = & !^{\mathfrak{p}(l)}[\Cap l_x : l.(\mathcal{C}[\![c]\!]_{l_x})] \\
\mathcal{P}[\![\sum\limits_{i\in I} \mathbf{ask}\ c\ \mathbf{then}\ P]\!]_l & = & !^{\mathfrak{p}(l)}[\Cap l_x : l.(\mathop{\&}\limits_{i\in I}[\mathcal{C}[\![c_i]\!]_{l_x} \multimap \mathcal{P}[\![P_i]\!]_{l_x}])] \\
\mathcal{P}[\![(\mathbf{local}\,\overline{x})\,P]\!]_l & = & !^{\mathfrak{p}(l)}[\Cap l_x : l.\exists\overline{x}.(\mathcal{P}[\![P]\!]_{l_x})] \\
\mathcal{P}[\![P \parallel Q]\!]_l & = & \mathcal{P}[\![P]\!]_l \otimes \mathcal{P}[\![Q]\!]_l \\
\mathcal{P}[\![p(\overline{x})]\!]_l & = & !^{\mathfrak{d}(l)}p(\overline{x}) \\
\mathcal{P}[\![p(\vec{x}) \stackrel{\Delta}{=} P]\!] & = & \Cap l_x : \infty.\forall\overline{x}.(!^{\mathfrak{d}(l_x)}p(\overline{x}) \multimap \mathcal{P}[\![P]\!]_{l_x})
\end{array}
$$

The main difference between the encodings in $\mathsf{SELL}^{\Cap}$ and ILL is the presence of *mobility* of processes, given by the universal quantifier $\Cap$ over subexponentials. This enables the specification of systems to govern an unbounded number of modalities.

Intuitively, when (left) focusing over a quantified clause of the form $\Cap l_x : l.!^{\mathfrak{f}(l_x)}F$, a location $a \in\downarrow l$ is chosen, and F becomes available in the location a, inside a family $\mathfrak{f}$, which is totally determined by the nature of the encoded object: $\mathfrak{c}$ for constraints, $\mathfrak{p}$ for processes, $\mathfrak{d}$ for process definitions. In the special case of $l = \infty$, F can be allocated *anywhere* inside the family. This is the case for example, of axioms and process definitions.

Let us now illustrate how the use of subexponentials and quantifiers allow for attaining the highest level of adequacy. The first thing to note is that, due to the shape of the encoding, the subexponential context can be divided into 3 zones: $\mathcal{C}, \mathcal{D}$ and $\mathcal{P}$, containing the formulas marked, respectively, with subexponentials of the form $\mathfrak{c}(\cdot), \mathfrak{d}(\cdot)$ and $\mathfrak{p}(\cdot)$.

Using simple logical equivalences, we can rewrite the encoding of a constraint $\mathcal{C}[\![c]\!]_l$ so that it has the following shape $\exists\overline{x}.\left(!^{\mathfrak{c}(l_1)}A_1 \otimes \cdots \otimes !^{\mathfrak{c}(l_n)}A_n\right)$, where $A_1, \ldots, A_n$ are atomic (positive) formulas. Whenever such a formula appears in the left-hand side, it is completely decomposed and stored in the $\mathcal{C}$ context:

$$
\dfrac{\dfrac{\mathcal{C} \uplus \{\mathfrak{c}(l_1) : A_1, \cdots, \mathfrak{c}(l_n) : A_n\}, \mathcal{D}, \mathcal{P}; \cdot \Uparrow \Delta \vdash \mathcal{R}}{\mathcal{C}, \mathcal{D}, \mathcal{P}; \cdot \Uparrow !^{\mathfrak{c}(l_1)}A_1, \cdots, !^{\mathfrak{c}(l_n)}A_n, \Delta \vdash \mathcal{R}}\ !^a{}_l}{\mathcal{C}, \mathcal{D}, \mathcal{P}; \cdot \Uparrow !^{\mathfrak{c}(l_1)}A_1 \otimes \cdots \otimes !^{\mathfrak{c}(l_n)}A_n, \Delta \vdash \mathcal{R}}\ \exists_l, \otimes_l
$$

That is, in the negative phase, the atomic formulas $A_1, \ldots, A_n$ appearing in the premise of this derivation are moved to the contexts $\mathcal{C}$.

Consider now a derivation that focuses on the encoding of a process. For instance, let $Q = \mathbf{ask}\ c\ \mathbf{then}\ P$, and $\mathcal{P}[\![Q]\!]_l = !^{\mathfrak{p}(l)}F$, with $F = \Cap l_x : a.(\mathcal{C}[\![c]\!]_{l_x} \multimap \mathcal{P}[\![P]\!]_{l_x})$. Focusing on F results necessarily in a focused derivation of the following shape:

$$
\dfrac{\dfrac{\dfrac{\pi}{\mathcal{C}'; \cdot \vdash \mathcal{C}[\![c]\!]_{l'} \Downarrow} \quad \dfrac{\mathcal{C}'', \mathcal{D}, \mathcal{P}' \uplus \{\mathfrak{p}(l') : F_P\}; \cdot \Uparrow \cdot \vdash G \Uparrow}{\mathcal{C}'', \mathcal{D}, \mathcal{P}'; \cdot \Downarrow \mathcal{P}[\![P]\!]_{l'} \vdash G}\ \mathsf{R}_l, !^a{}_l}{\mathcal{C}, \mathcal{D}, \mathcal{P}'; \cdot \Downarrow \Cap l_x : a(\mathcal{C}[\![c]\!]_{l_x} \multimap \mathcal{P}[\![P]\!]_{l_x}) \vdash G}\ \Cap_l, \multimap_l}{\mathcal{C}, \mathcal{D}, \mathcal{P} \uplus \{\mathfrak{p}(l) : F\}; \cdot \Uparrow \cdot \vdash \cdot \Uparrow G}\ \mathsf{Du/Db}
$$

If $\mathfrak{p}(l) \in U$ (resp. $\mathfrak{p}(l) \notin U$) the rule Du (resp. Db is applied) and $\mathcal{P}' = \mathcal{P} \uplus \{\mathfrak{p}(l) : F\}$ (resp. $\mathcal{P}' = \mathcal{P}$). Since $\mathcal{C}[\![c]\!]_{l'}$ contains only positive formulas, it will be totally decomposed, and every exponential context in π will be a $\mathcal{C}$ context. That is, only constraints and axioms from the constraint system can be used in the proof π.

A similar analysis can be done when a process definition is selected: only the context $\mathcal{D}$, storing all the calls, can be used to entail the needed guard.

In the following, we instantiate the general definition of the encoding for different flavors of CCP. The adequacy we obtain, in each case is at the **FCD** level.

Classical and linear CCP

For encoding the language in Section 3, the set of modalities is the simplest one: $M = \{nil, \infty\}$. All the subexponentials but $\mathfrak{p}(nil)$ and $\mathfrak{d}(\cdot)$ are unbounded.

▶ **Theorem 10.** *Let $(\mathcal{C}, \models_\Delta)$ be a constraint system, P be a CCP process and Ψ be a set of process definitions. Then, for any constraint c,*
$P \Downarrow_c$ *iff* $\cdot \Uparrow !^{\mathfrak{c}(\infty)}\mathcal{C}[\![\Delta]\!], !^{\mathfrak{p}(\infty)}[\![\Psi]\!], \mathcal{P}[\![P]\!]_{nil} \vdash \mathcal{C}[\![c]\!]_{nil} \otimes \top \Uparrow$

It is worth noticing that all the processes remain in the location nil (denoting "without modality") and then, the universal quantification in the encoding is always forced to instantiate l_x with nil.

Linear CCP. As we already know, the store in CCP increases monotonically: once a constraint is added, it cannot be removed from the store. This can be problematic for the specification of systems where resources can be consumed. In linear CCP (`lcc`) [11], constraints are built from formulas in the following fragment of ILL:

$F ::= A \mid 1 \mid F \otimes F \mid \exists x.F \mid !F$

In this setting, the empty store is 1 and constraints are accumulated using $\otimes$. The extra case $!F$, as expected, is used to denote persistent constraints.

▶ **Example 11.** The *vending coffee machine* has the same CCP specification as the community coffee machine presented in Example 4. However, as expected, linear asks consume constraints when querying the store and the coin does not come back after delivering the coffee:

$$\langle \emptyset, P, 1\rangle \quad\longrightarrow\quad \langle \emptyset, \mathtt{m()}, \mathit{coin}\rangle \longrightarrow \langle \emptyset, \mathbf{tell}(\mathit{coffee}) \parallel \mathtt{m()}, 1\rangle \longrightarrow \langle \emptyset, \mathtt{m()}, \mathit{coffee}\rangle$$

In order to characterize the semantics of `lcc`, we configure the encoding in Definition 9 as follows. We declare $\mathfrak{c}(nil) \notin U$ (i.e., constraints can be consumed) and $\mathfrak{c}(\infty) \in U$. Moreover, the encoding is extended for the case of unbounded constraints: $\mathcal{C}[\![!\,c]\!]_l = \mathcal{C}[\![c]\!]_\infty$. In this way, we obtain an adequacy theorem as the one in Theorem 10, also at the **FCD** level, in contrast to the weakest level of adequacy (**FCP**) obtained originally in [11] (for linear logic and without focusing).

It is important to note that the characterization in Theorem 6, that uses (vanilla) linear logic, does not work for `lcc` at the **FCD** level. Take for instance the process $Q = \mathbf{ask}\ c \otimes d\ \mathbf{then}\ P$ being executed in the store $!(c \otimes d)$. Clearly, Q reduces to P and the store remains unchanged. If we were to use the encoding in Theorem 6, before focusing on $\mathcal{P}[\![Q]\!]$, we have to do an intermediary step without an operational counterpart: focus on $c \otimes d$, stored in the classical context, to produce a copy of c and d in the linear context. Only after that, the implication in $\mathcal{P}[\![Q]\!]$ is able to entail the guard $c \otimes d$. In the encoding of the present section, proving the query of Q results in focusing on $!^{\mathfrak{c}(nil)}c \otimes !^{\mathfrak{c}(nil)}d$. After decomposing the tensor, focusing is lost and only linear $\mathfrak{c}(nil)$ and replicated ($\mathfrak{c}(\infty)$) constraints and the axioms of the constraint systems can be used to deduce the atoms c and d. This adequately reflects the semantics of linear asks.

Epistemic CCP

Now let us consider a richer system where different modalities will play a fundamental role. Epistemic CCP (`eccp`) [16] is a CCP-based language where systems of agents are considered for distributed and epistemic reasoning. In `eccp`, the constraint system is extended to consider space of agents, denoted as $s_a(c)$, and meaning "c holds in the space –store– of agent a." The function $s_a(\cdot)$ satisfies certain conditions to reflect epistemic behaviors:

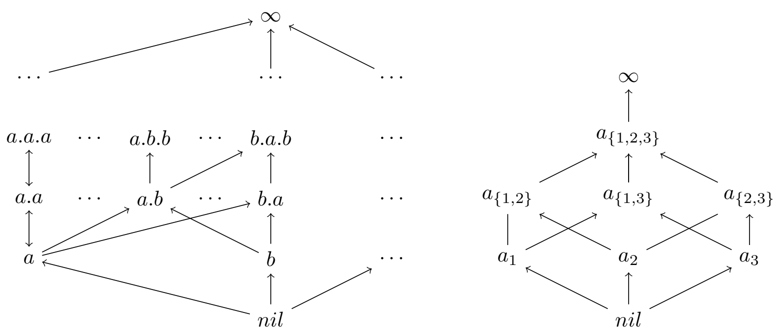


Figure 3 Subexponential signature for `eccp`.

$$\frac{(X;P,\Gamma;c) \longrightarrow (X';P',\Gamma;d)}{(X;[P]_a,\Gamma;c) \longrightarrow (X';[P]_a,P',\Gamma;d)}\ \mathrm{R_E} \qquad \frac{(X;P,\Gamma;d^a) \longrightarrow (X';P',\Gamma;d')}{(X;[P]_a,\Gamma;d) \longrightarrow (X';[P']_a,\Gamma;d\wedge s_a(d'))}\ \mathrm{R_S}$$

Figure 4 Operational rules for `eccp` and `sccp`.

1. $s_a(1) = 1$ (bottom preserving)
2. $s_a(c \wedge d) = s_a(c) \wedge s_a(d)$ (lub preserving)
3. If $d \vdash_{\Delta_e} c$ then $s_a(d) \vdash_{\Delta_e} s_a(c)$ (monotonicity)
4. $s_a(c) \vdash_{\Delta_e} c$ (believes are facts –extensiveness–)
5. $s_a(s_a(c)) = s_a(c)$ (idempotence)

In `eccp`, the language of processes is extended with the constructor $[P]_a$ that represents P running in the space of the agent a. The operational rules for $[P]_a$ are specified in Figure 4. In epistemic systems, agents are trustful, i.e., if an agent a knows some information c, then c is necessarily true. Furthermore, if b knows that a knows c, then b also knows c. For example, given a hierarchy of agents as in $[[P]_a]_b$, it should be possible to propagate the information produced by P in the space a to the outermost space b. This is captured exactly by the rule $\mathrm{R_E}$, which allows a process P in $[P]_a$ to run also outside the space of agent a. Notice that the process P is contracted in this rule. The rule $\mathrm{R_S}$, on the other hand, allows us to observe the evolution of processes inside the space of an agent. There, the constraint d^a represents the information the agent a may see or have of d, i.e., $d^a = \bigwedge\{c \mid d \vdash_{\Delta_e} s_a(c)\}$. For instance, a *sees* c from the store $s_a(c) \wedge s_b(c')$ but it does not see c'.

We now configure the encoding in Definition 9 so to capture the behavior of `eccp` processes. We consider a possibly infinite set of agents $\mathcal{A} = \{a_1, a_2, ...\}$ and the set of locations/modalities M, besides nil and ∞, contains the set $\mathcal{A}^+$ of non-empty strings of elements in $\mathcal{A}$; for example, if $a, b \in \mathcal{A}$, then $a, b, a.a, b.a, a.b.a, \ldots \in \mathcal{A}^+$. We use $\overline{a}, \overline{b}, etc$ to denote elements in $\mathcal{A}^+$ and nil will denote the empty string. The only linear subexponentials are $\mathfrak{d}(nil)$ and $\mathfrak{p}(nil)$. This reflects the fact that both constraints and processes in the space of an agent are unbounded, as specified by rule $\mathrm{R_E}$. Intuitively, $!^{\mathfrak{p}(1.2.3)}$ specifies a process in the structure $[[[\cdot]_3]_2]_1$, denoting "agent 1 knows that agent 2 knows that agent 3 knows" expressions. The connective $!^{\mathfrak{c}(1.2.3)}$, on the other hand, specifies a constraint of the form $s_1(s_2(s_3(\cdot)))$. We thus extend the encoding accordingly: $\mathcal{C}[\![s_i(c)]\!]_{\overline{l}} = \mathcal{C}[\![c]\!]_{\overline{l}.i}$ and $\mathcal{P}[\![[P]_i]\!]_{\overline{l}} = \mathcal{P}[\![P]\!]_{\overline{l}.i}$.

The pre-order $\preceq$ is as depicted in Figure 3 on the left. Note that for every two different agent names a and b in $\mathcal{A}$, the subexponentials a and b are unrelated. Moreover, $a \approx a.\overline{a}$ and $b_1.b_2.\ldots.b_n \preceq \overline{a}_1.b_1.\overline{a}_2.b_2.\ldots.\overline{a}_n.b_n.\overline{a}_{n+1}$ where each $\overline{a}_i$ is a possible empty string of elements in $\mathcal{A}$. The shape of the pre-order is key for our encoding. For instance, the formula

$\Cap l_x : a.b.b.\mathcal{P}[\![P]\!]_{l_x}$ on the left, allows us to place P on the (outer) location $a.b$ and b as required by $\mathrm{R_E}$. In fact, we can show that the sequent $\mathcal{P}[\![P]\!]_{\bar{l}.i} \vdash \mathcal{P}[\![P]\!]_{\bar{l}}$ is provable in $\mathsf{SELL}^{\Cap}$ for any process P and subexponentials $\bar{l}$ and i. We can also show that the encoding of constraints satisfy the axioms of an epistemic constraint system. For instance, the sequent $\mathcal{C}[\![s_i(c)]\!]_{\bar{l}} \vdash \mathcal{C}[\![c]\!]_{nil}$ is provable, showing that believes are facts. Hence, a tailored version of Theorem 10 applies for this language, with the same level of adequacy.

As an interesting example of epistemic behavior, it is possible to specify common knowledge by extending the subexponential signature as in Figure 3 on the right, where for all $\mathcal{S} \subseteq \mathcal{A}$, $\overline{a} \preceq a_{\mathcal{S}}$ for any string $\overline{a} \in \mathcal{S}^+$. Then, the announcement of c on the group of agents $\mathcal{S}$ can be represented by $!^{\mathfrak{c}(a_{\mathcal{S}})}c$. Notice that the sequent $!^{\mathfrak{c}(a_{\mathcal{S}})}c \vdash !^{\mathfrak{c}(\overline{a})}c \otimes \top$ can be proved for any $\overline{a} \in \mathcal{S}^+$. For instance, if $\mathcal{S} = \{a_i, a_j\}$, from $!^{\mathfrak{c}(a_{\mathcal{S}})}c$ one can prove that a_i knows that a_j knows that a_i knows that a_i knows ... c, i.e., c is common knowledge between a_i and a_j.

Spatial CCP

Inconsistent information in CCP arises when considering theories containing axioms such as $c \wedge d \vdash_\Delta 0$. Unlike epistemic scenarios, in spatial computations, a space can be locally inconsistent and it does not imply the inconsistency of the other spaces (i.e., $s_a(0)$ does not imply $s_b(0)$). Moreover, the information produced by a process in a space is not propagated to the outermost spaces (i.e., $s_a(s_b(c))$ does not imply $s_a(c)$).

In [16], spatial computations are specified in spatial CCP (`sccp`) by considering processes of the form $[P]_a$ as in the epistemic case, but excluding the rule $\mathrm{R_E}$ in the system shown in Figure 4. Furthermore, some additional requirements are imposed on the representation of agents' spaces $s_a(\cdot)$. In particular, $s_a(\cdot)$ must satisfy false containment, i.e., if $c \wedge d \models_\Delta 0$, it does not necessarily imply that $s_a(c) \wedge s_b(d) \models_\Delta 0$ if $a \neq b$.

We build the subexponential signature as we did in the epistemic case but the pre-order is much simpler: for any $\overline{a} \in \mathcal{A}^+$, $\overline{a} \preceq \infty$. That is, two different elements of $\mathcal{A}^+$ are unrelated. Moreover, since `sccp` does not contain the $\mathrm{R_E}$ rule, processes in spaces are again treated linearly. Thus: $U = \{\mathfrak{c}(a) \mid a \in I\} \cup \{\mathfrak{p}(\infty)\}$.

By modifying the pre-order we partially capture the behavior of spatial systems. However, it is not enough to confine inconsistencies. In particular, note that $!^a 0 \vdash G$ for any a and G. The solution for information confinement, as shown in [31], is to consider combinations of bangs and question marks (the dual of bang). In this case, $!^a ?^a 0 \vdash !^a ?^a G$ but $!^a ?^a 0 \not\vdash !^b ?^b G$ for a, b not related. Hence, the encoding remains the same, but for the base cases: atomic propositions are encoded as $!^{\mathfrak{c}(l)} ?^{\mathfrak{c}(l)} A$, and procedure calls as $!^{\mathfrak{d}(l)} ?^{\mathfrak{d}(l)} p(\vec{x})$.

6 Conclusion and future work

We have shown that the process-as-formula interpretation can provide useful reasoning techniques for process calculi, by faithfully capturing the behavior of processes. The interpretations we have achieved are *modular* and *parametric*, and they can capture different modal behaviors as Table 1 summarizes.

Other examples of processes-as-formulas interpretations, relating computation and proof search, include linear logic-based models for the π-calculus [22], abstract transition systems and operational semantics [20], CCS [10], Bigraphs [5], P-systems [33] and concurrent object oriented programming languages [36]. Also, in [4] we have tailored the notion of fixed points in linear logic [2] to the system $\mathsf{SELL}^{\Cap}$, and this allowed the encoding of CTL (Computational Tree Logic) formulas as SELL theories, thus opening the possibility of specifying and proving temporal properties inside the same logical framework.

Table 1 Encoding of CCP modalities in $\mathsf{SELL}^{\Cap}$.

General Encoding	
Connective	Meaning
$\nabla_s = !^s$	$!^s P$ is **located** at s.
$\nabla_s = !^s ?^s$	$!^s ?^s P$ is **confined** to s.
$\Cap l : a\ P$	P can **move** to locations below (outside) a
Epistemic Modalities	
Pre-order	Meaning
$a.a \sim a$	Modalities are **idempotent**: $[[P]_a]_a \sim [P]_a$
$a \preceq a.b$	Processes can **move outside** $[[P]_b]_a \longrightarrow [P \parallel [P]_b]_a$
Spatial Modalities	
Pre-order	Meaning
$a \not\preceq b$	P does not communicate with Q in $[P]_a \parallel [Q]_b$
$a.a \not\sim a$	Modalities **are not** necessarily idempotent.
$a \not\preceq a.b$	Processes are confined: $[[P]_b]_a \not\sim [P \parallel [P]_b]_a$

Regarding future work, in [17] we have shown how to incorporate other modal behaviors (besides the structural ones of weakening and contraction) in linear logic, thus extending the multiplicative and additive fragment of LL with *simply dependent* multi-modalities. The interpretations we have presented here have inspired new CCP-based calculi [35]. We foresee that the finer control of modalities given in [17], as well as the extensions with *non-normal modalities* [6, 18, 7], may contribute with other declarative models of concurrency with strong logical foundations.

References

1. Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3):297–347, 1992.
2. David Baelde. Least and greatest fixed points in linear logic. *ACM Trans. Comput. Log.*, 13(1):2:1–2:44, 2012.
3. Kaustuv Chaudhuri. Classical and intuitionistic subexponential logics are equally expressive. In Anuj Dawar and Helmut Veith, editors, *CSL 2010*, volume 6247 of *LNCS*, pages 185–199. Springer, 2010.
4. Kaustuv Chaudhuri, Joëlle Despeyroux, Carlos Olarte, and Elaine Pimentel. Hybrid linear logic, revisited. *Math. Struct. Comput. Sci.*, 29(8):1151–1176, 2019.
5. Kaustuv Chaudhuri and Giselle Reis. An adequate compositional encoding of bigraph structure in linear logic with subexponentials. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *LPAR-20*, volume 9450 of *LNCS*, pages 146–161. Springer, 2015.
6. Brian F. Chellas. *Modal Logic.* Cambridge University Press, 1980. `doi:10.1017/CBO9780511621192`.
7. Tiziano Dalmonte, Björn Lellmann, Nicola Olivetti, and Elaine Pimentel. Hypersequent calculi for non-normal modal and deontic logics: countermodels and optimal complexity. *J. Log. Comput.*, 31(1):67–111, 2021. `doi:10.1093/logcom/exaa072`.
8. Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. The structure of exponentials: Uncovering the dynamics of linear logic proofs. In Georg Gottlob, Alexander Leitsch, and Daniele Mundici, editors, *Kurt Gödel Colloquium*, volume 713 of *LNCS*, pages 159–171. Springer, 1993.
9. Frank S. de Boer, Maurizio Gabbrielli, and Maria Chiara Meo. Proving correctness of timed concurrent constraint programs. *ACM Trans. Comput. Log.*, 5(4):706–731, 2004.

10 Yuxin Deng, Robert J. Simmons, and Iliano Cervesato. Relating reasoning methodologies in linear logic and process algebra. *Math. Struct. Comput. Sci.*, 26(5):868–906, 2016.

11 François Fages, Paul Ruet, and Sylvain Soliman. Linear concurrent constraint programming: Operational and phase semantics. *Information and Computation*, 165(1):14–41, 2001.

12 Fabio Gadducci, Francesco Santini, Luis Fernando Pino, and Frank D. Valencia. Observational and behavioural equivalences for soft concurrent constraint programming. *J. Log. Algebr. Meth. Program.*, 92:45–63, 2017.

13 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

14 C. A. R. Hoare. *Communications Sequential Processes.* Prentice-Hall, Englewood Cliffs (NJ), USA, 1985.

15 Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.

16 Sophia Knight, Catuscia Palamidessi, Prakash Panangaden, and Frank D. Valencia. Spatial and epistemic modalities in constraint-based process calculi. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR*, volume 7454 of *LNCS*, pages 317–332. Springer, 2012.

17 Björn Lellmann, Carlos Olarte, and Elaine Pimentel. A uniform framework for substructural logics with modalities. In Thomas Eiter and David Sands, editors, *LPAR-21*, volume 46 of *EPiC Series in Computing*, pages 435–455. EasyChair, 2017.

18 Björn Lellmann and Elaine Pimentel. Modularisation of sequent calculi for normal and non-normal modalities. *ACM Trans. Comput. Log.*, 20(2):7:1–7:46, 2019. `doi:10.1145/3288757`.

19 Chuck Liang and Dale Miller. Focusing and polarization in intuitionistic logic. In J. Duparc and T. A. Henzinger, editors, *CSL 2007: Computer Science Logic*, volume 4646 of *LNCS*, pages 451–465. Springer, 2007.

20 Raymond McDowell, Dale Miller, and Catuscia Palamidessi. Encoding transition systems in sequent calculus. *Theor. Comput. Sci.*, 294(3):411–437, 2003.

21 Dale Miller. Hereditary harrop formulas and logic programming. In *Proceedings of the VIII International Congress of Logic, Methodology, and Philosophy of Science*, pages 153–156, Moscow, 1987.

22 Dale Miller. The pi-calculus as a theory in linear logic: Preliminary results. In Evelina Lamma and Paola Mello, editors, *ELP'92*, volume 660 of *LNCS*, pages 242–264. Springer, 1992.

23 Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

24 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

25 Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Inf. Sci.*, 7:95–132, 1974.

26 M. Nielsen, C. Palamidessi, and F. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nordic Journal of Computing*, 9(1):145–188, 2002.

27 Vivek Nigam. On the complexity of linear authorization logics. In *LICS*, pages 511–520. IEEE, 2012.

28 Vivek Nigam and Dale Miller. Algorithmic specifications in linear logic with subexponentials. In António Porto and Francisco Javier López-Fraguas, editors, *Proc. of PPDP'09*, pages 129–140. ACM, 2009.

29 Vivek Nigam and Dale Miller. A framework for proof systems. *J. Autom. Reasoning*, 45(2):157–188, 2010.

30 Vivek Nigam, Carlos Olarte, and Elaine Pimentel. A general proof system for modalities in concurrent constraint programming. In Pedro R. D'Argenio and Hernán C. Melgratti, editors, *CONCUR*, volume 8052 of *Lecture Notes in Computer Science*, pages 410–424. Springer, 2013.

31 Vivek Nigam, Carlos Olarte, and Elaine Pimentel. On subexponentials, focusing and modalities in concurrent systems. *Theor. Comput. Sci.*, 693:35–58, 2017.

32 Vivek Nigam, Elaine Pimentel, and Giselle Reis. An extended framework for specifying and reasoning about proof systems. *J. Log. Comput.*, 26(2):539–576, 2016.

33 Carlos Olarte, Davide Chiarugi, Moreno Falaschi, and Diana Hermith. A proof theoretic view of spatial and temporal dependencies in biochemical systems. *Theor. Comput. Sci.*, 641:25–42, 2016.

34 Carlos Olarte and Elaine Pimentel. On concurrent behaviors and focusing in linear logic. *Theor. Comput. Sci.*, 685:46–64, 2017.

35 Carlos Olarte, Elaine Pimentel, and Vivek Nigam. Subexponential concurrent constraint programming. *Theor. Comput. Sci.*, 606:98–120, 2015.

36 Carlos Olarte, Elaine Pimentel, and Camilo Rueda. A concurrent constraint programming interpretation of access permissions. *Theory Pract. Log. Program.*, 18(2):252–295, 2018.

37 Carlos Olarte, Camilo Rueda, and Frank D. Valencia. Models and emerging trends of concurrent constraint programming. *Constraints*, 18(4):535–578, 2013.

38 Elaine Pimentel, Vivek Nigam, and João Neto. Multi-focused proofs with different polarity assignments. In Mario R. F. Benevides and René Thiemann, editors, *Proc. of LSFA'15*, volume 323 of *ENTCS*, pages 163–179. Elsevier, 2015.

39 Elaine Pimentel, Carlos Olarte, and Vivek Nigam. A proof theoretic study of soft concurrent constraint programming. *Theory Pract. Log. Program.*, 14(4-5):649–663, 2014.

40 V. Saraswat and Martin Rinard. Concurrent constraint programming. In *17th ACM Symp. on Principles of Programming Languages*, pages 232–245, San Francisco, CA, 1990.

41 Vijay A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

42 Vijay A. Saraswat and Martin C. Rinard. Concurrent constraint programming. In Frances E. Allen, editor, *POPL*, pages 232–245. ACM Press, 1990.

43 Vijay A. Saraswat, Martin C. Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In David S. Wise, editor, *POPL*, pages 333–352. ACM Press, 1991.

44 Dana S. Scott. Domains for denotational semantics. In Mogens Nielsen and Erik Meineche Schmidt, editors, *ICALP*, volume 140 of *LNCS*, pages 577–613. Springer, 1982.

45 Ehud Shapiro. The family of concurrent logic programming languages. *ACM Comput. Surv.*, 21(3), 1989.

46 Gert Smolka. A foundation for higher-order concurrent constraint programming. In J.-P. Jouannaud, editor, *Proceedings of Constraints in Computational Logics*, volume 845 of *LNCS*, pages 50–72. Springer-Verlag, 1994.

Some Formal Structures in Probability

Sam Staton
University of Oxford, UK

Abstract

This invited talk will discuss how developments in the Formal Structures for Computation and Deduction can also suggest new directions for the foundations of probability theory. I plan to focus on two aspects: abstraction, and laziness. I plan to highlight two challenges: higher-order random functions, and stochastic memoization.

2012 ACM Subject Classification Theory of computation → Program semantics; Mathematics of computing → Nonparametric statistics

Keywords and phrases Probabilistic programming

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.4

Category Invited Talk

Funding *Sam Staton*: Research supported by a Royal Society University Research Fellowship and the ERC BLAST grant.

Summary

Probabilistic programming is a popular tool for statistics and machine learning. The idea is to describe a probabilistic model as a program with random choices. The program might be a simulation of some system, such as a physics model, a model of viral spread, or a model of electoral behaviour. We can now carry out statistical inference over the system by running a Monte Carlo simulation – running the simulation 100,000's of times. The key observation of probabilistic programming is that we can actually run this same probabilistic program with different advanced simulation methods, instead of a naive Monte Carlo simulation, such as a Hamiltonian Monte Carlo simulation or Variational Inference, without changing the program. See [20] for an overview.

Part of the *practical* appeal of probabilistic programming is this separation between probabilistic models and inference algorithms. But this also has a *foundational* appeal: if we can understand probabilistic models as programs, then the foundations of probability and statistics can be discussed in terms of program semantics. This might take the lead of denotational semantics, by interpreting programs in terms of traditional measure theory (e.g. [17, 18]). But there is also a chance of new foundational perspectives on probability by following other semantic methods, such as equational reasoning, rewriting, or categorical axiomatics (see also [1, 4]).

This programming-based foundation for probability is attractive because there are some intuitively simple probabilistic scenarios which have an easy programming implementation but for which a plain measure-theoretic interpretation seems impossible. I now highlight some issues in abstraction and laziness.

Abstraction. Abstraction is a crucial concept in probability: statistics arise by abstracting away information. At a higher level, we have argued that de Finetti's theorem, a fundamental theorem in probability, can be understood in terms of abstract data types [15], and so too generalizations [8, 16].

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi; Article No. 4; pp. 4:1–4:4
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Function types are a key abstraction in programming theory, but are less well understood in probability. For example, write $\mathrm{Pr}(X)$ for the space of probability distributions on X, and consider the functional

$$\mathsf{piecewise} : \mathrm{Pr}(\mathbb{R} \to \mathbb{R}) \to \mathrm{Pr}(\mathbb{R} \to \mathbb{R})$$

which converts a random function to a random piecewise version of it (see Figure 1). It is easy to define in a few lines of code: $\mathsf{piecewise}(f)$ will draw a random partition of the x-axis, draw random functions from f for each part, and splice them together. But although statistics and probability make plenty of use of random piecewise linear functions, random piecewise constant functions, and so on, the piecewise functional itself has no direct interpretation in traditional measure theory. This has led to some recent semantic developments (e.g. [6, 3, 13, 2, 7, 14]).

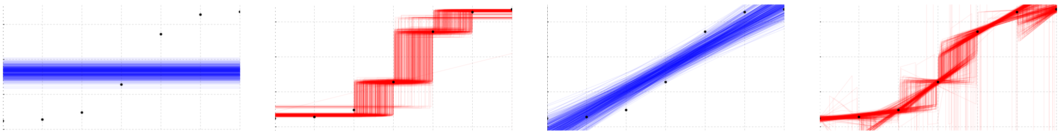
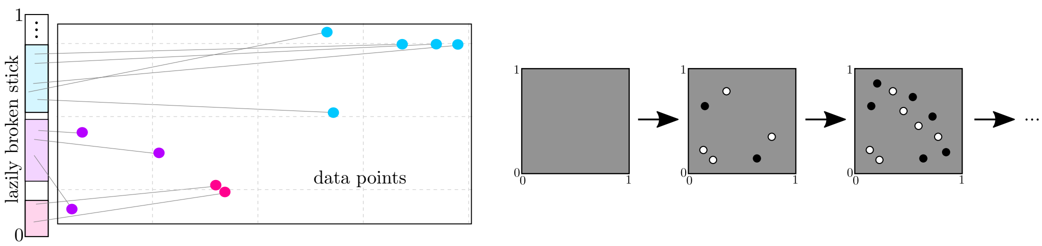


Figure 1 Bayesian regression for the data set indicated by black dots. The regression was done using constant, piecewise constant, linear, and piecewise linear functions respectively. The piecewise functional was used to program the random piecewise functions.

Laziness. Laziness in programming is a counterpart to the notion of "process" which is fundamental in probability. This has long been understood [5, 9, 10], and I recently explored more aspects of laziness in the prototype LazyPPL [19]. For example, a "stick breaking" process randomly divides the unit interval into an infinite number of parts, each part representing a different cluster of some data. If this is computed lazily, it always terminates, because the data is finite (Figure 2(a)).

One outstanding problem is a semantic interpretation of stochastic memoization. In the non-probabilistic setting, memoization is a program optimization, where we are lazy about re-evaluating a function at a given argument, by caching or tabling. But in the probabilistic setting it gives new semantic possibilities. Stochastic memoization is a functional

$$\mathsf{memoize} : (X \to \mathrm{Pr}(Y)) \to \mathrm{Pr}(X \to Y)$$



(a) (b)

Figure 2 (a) Stick-breaking: To group the data points (right) into an unknown number of clusters, we randomly divide the unit interval "stick" into an infinite partition (left), and then assign a cluster to each data point by randomly picking a number in $[0, 1]$ for each point (lines from the data points to the stick). In practice, this is done lazily. (b) Lazily building the adjacency matrix of an uncountable random graph, as a memoized random function $[0, 1]^2 \to \mathsf{bool}$.

which converts a family $f : X \to \Pr(Y)$ of probability distributions into a distribution on functions $X \to Y$, by sampling $f(x)$ once for every x. When X is infinite, this is impossible to do eagerly, but it is no problem lazily. For example, consider a function $g : [0,1]^2 \to \Pr(\mathsf{bool})$ where $g(x, y)$ is the Bernoulli distribution (a coin flip); then $\mathsf{memoize}(g) : \Pr([0,1]^2 \to \mathsf{bool})$ is the random adjacency matrix of a random uncountable graph (Figure 2(b)). More generally, g is a "graphon" (e.g. [12]). This also generalizes clustering by stick-breaking, because clusters can be regarded as connected components of graphs.

This memoize functional is easy to implement. It appears in several languages [5, 11, 19], and is practically useful in random graphs, probabilistic logic [11], clustering [5, §2.1], and natural language modelling [21]. But there remains a big open problem:

▶ **Open problem.** *To find a denotational model for a language with stochastic memoization.*

I will discuss some recent progress on this problem, based on ongoing work with Swaraj Dash, Younesse Kaddar, Hugo Paquet, and others.

References

1 Kenta Cho and B. Jacobs. Disintegration and Bayesian inversion via string diagrams. *Math. Struct. Comput. Sci.*, 29:938–971, 2019.

2 Fredrik Dahlqvist and Dexter Kozen. Semantics of higher-order probabilistic programs with conditioning. In *Proc. POPL 2020*, 2020.

3 Thomas Ehrhard, Michele Pagani, and Christine Tasson. Measurable cones and stable, measurable functions: a model for probabilistic higher-order programming. In *Proc. POPL 2018*, 2018.

4 T. Fritz. A synthetic approach to Markov kernels, conditional independence and theorems on sufficient statistics. *Adv. Math.*, 370, 2020.

5 N. D. Goodman, V. K. Mansinghka, et al. Church: a language for generative models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, 2008.

6 Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. A convenient category for higher-order probability theory. In *Proc. LICS 2017*, 2017.

7 Xiaodong Jia, Bert Lindenhovius, Michael W. Mislove, and Vladimir Zamdzhiev. Commutative monads for probabilistic programming languages. In *Proc. LICS 2021*, 2021.

8 P. Jung, J. Lee, S. Staton, and H. Yang. A generalization of hierarchical exchangeability on trees to directed acyclic graphs. *Annales Henri Lebesgue*, 4, 2021.

9 Oleg Kiselyov and Chung-chieh Shan. Embedded probabilistic programming. In *Proc. DSL 2009*, 2009.

10 Daphne Koller, David McAllester, and Avi Pfeffer. Effective Bayesian inference for stochastic programs. In *Proc. AAAI 1997*, 1997.

11 Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG: Probabilistic models with unknown objects. In *Introduction to Statistical Relational Learning*. MIT Press, 2007.

12 Peter Orbanz and Daniel M. Roy. Bayesian models of graphs, arrays and other exchangeable random structures. *IEEE Trans. Pattern Anal. Mach. Intell.*, 37(2):437–461, 2015.

13 Hugo Paquet and Glynn Winskel. Continuous probability distributions in concurrent games. In *Proc. MFPS 2018*, pages 321–344, 2018.

14 Marcin Sabok, Sam Staton, Dario Stein, and Michael Wolman. Probabilistic programming semantics for name generation. In *Proc. POPL 2021*, 2021.

15 S. Staton, D. Stein, H. Yang, L. Ackerman, C. E. Freer, and D. M. Roy. The Beta-Bernoulli process and algebraic effects. In *Proceedings of the 45th International Colloquium on Automata, Languages, and Programming (ICALP)*, 2018.

16 S. Staton, H. Yang, N. L.. Ackerman, C. Freer, and D. Roy. Exchangeable random process and data abstraction. In *PPS 2017*, 2017.

17 Sam Staton. Commutative semantics for probabilistic programming. In *Proc. ESOP 2017*, 2017.
18 Sam Staton. Probabilistic programs as measures. In *Foundations of Probabilistic Programming*. CUP, 2020.
19 Sam Staton. LazyPPL, 2021. URL: `https://bitbucket.org/samstaton/lazyppl/src/`.
20 Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An introduction to probabilistic programming, 2018. `arXiv:1809.10756`.
21 Frank D. Wood, Cédric Archambeau, Jan Gasthaus, Lancelot James, and Yee Whye Teh. A stochastic memoizer for sequence data. In *Proc. ICML 2009*, 2009.

The Expressive Power of One Variable Used Once: The Chomsky Hierarchy and First-Order Monadic Constructor Rewriting

Jakob Grue Simonsen ✉
Department of Computer Science, University of Copenhagen, Denmark

Abstract

We study the implicit computational complexity of constructor term rewriting systems where every function and constructor symbol is unary or nullary. Surprisingly, adding simple and natural constraints to rule formation yields classes of systems that accept exactly the four classes of languages in the Chomsky hierarchy.

2012 ACM Subject Classification Theory of computation → Grammars and context-free languages; Theory of computation → Equational logic and rewriting; Theory of computation → Computability

Keywords and phrases Constructor term rewriting, Chomsky Hierarchy, Implicit Complexity

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.5

Acknowledgements I wish to thank the anonymous referees for diligent comments that have helped improve the presentation of the paper.

1 Introduction

A natural means of studying the expressive power of declarative programming languages is via constructor term rewriting systems; In these, the set of symbols are partitioned into *defined symbols* and *constructor symbols*, the former representing function names, and the latter representing data constructors.

The study of *implicit complexity* for a class of rewrite systems is, roughly, the study of the set of problems that can be accepted, decided, or otherwise characterized by the class. Implicit complexity has been studied extensively in functional programming (see – amongst many others – [4, 18, 22]), and in term rewriting [3, 2, 9, 19, 8].

In this paper, we study the implicit complexity of constructor term rewriting systems where all function and constructor symbols are restricted to have arity at most one (*monadic* systems); the rewriting systems are characterized according to the computational complexity of the constructor terms they accept. Unsurprisingly, the most general class of monadic systems accept the entire class of recursively enumerable sets. However, imposing simple and natural restrictions leads to exact characterization of the three other classes in the Chomsky hierarchy [7]: Context-sensitive, context-free, and regular languages. The results hold for the *unrestricted* rewriting relation, that is, we impose no evaluation order, and no typing beyond partitioning into sets of defined symbols and constructor symbols.

The restrictions we impose echo the usual intuition about classes in the Chomsky hierarchy: R.e. languages are accepted by Turing machines (finite state + two stacks), context-free languages by PDAs (finite state + one stack), regular languages by DFAs (finite state + no stacks), and context-sensitive languages by LBAs (finite-state + two stacks with a boundedness condition). The novel bits are that (i) we do not enforce machine-like restrictions on the rewrite relation (e.g., rewriting is not required to be innermost), and (ii) that both the encoding of the stacks and the behaviour of tail vs. general recursion have to be done with some finesse.

Classes:

Type	Restriction on rules $l \to r$	Example(s) of rule(s)	Expressive power
Unrestricted	None	$f(\mathtt{c}(x)) \to g(h((\mathtt{a}(\mathtt{d}(x)))))$	RE
Non-length-increasing	$\|r\| \leq \|l\|$	$f(\mathtt{c}(\mathtt{d}(x))) \to g(h(\mathtt{a}(x)))$	CSL
(Strongly) cons-free	No constructor symbols in r	$f(\mathtt{c}(x)) \to g(h(x))$, $f(\mathtt{c}(\mathtt{d}(x))) \to x$	CFL
(Strongly) cons-free & tail recursive	cons-free, and the order of defined symbols in r respect a certain preorder	$f(\mathtt{c}(x)) \to g(x)$, $f(x) \to f(g(h(x))$ (with f not appearing below g or h in the rhs of any rule with g or h in the lhs)	REG

Figure 1 Classes of monadic constructor TRSs and the classes of sets they accept.

Figure 1 gives an overview of the four classes of systems we consider and their relation to the language classes in the Chomsky hierarchy.

Related work

For characterizing context-free and regular languages, we disallow constructors in the right-hand side of rules; this idea stems from Jones' work on the expressive power of higher-order types in functional programming [18] where a number of complexity classes were characterized in programs with call-by-value semantics and where functions may have arbitrary arity. Similar ideas have since been used in rewriting with less strict constraints on the evaluation order [9, 19], but for symbols with arbitrarily high finite arity. Correspondences between context-free languages and so-called *monadic recursion schemes* – essentially function declarations where all functions and data constructors are unary – were investigated some 40 years ago [14, 11, 10, 12]; the research focused mostly on decidability results, but close correspondences between monadic programs with very limited data construction abilites and context-free languages, was established there. Caron [6] proved undecidability of termination for non-length-increasing TRSs by encoding a certain class of linear bounded automata; we use a very similar approach to show that non-length-increasing constructor TRSs precisely accept the context-sensitive languages. Implicit complexity for term rewriting systems has been investigated in a number of papers; see the references above. Finally, the restriction to unary and nullary symbols means that all results in the paper can be viewed as concerning an especially well-behaved class of *string rewriting*; we refer the reader to [27, 28] for overviews of the correspondence between string rewriting and rewriting with unary symbols.

2 Preliminaries

We assume a non-empty alphabet, A, of *characters* and consider languages $L \subseteq A^+$ where A^+ is the set of non-empty strings of characters from A. The *empty* string over any alphabet will be denoted ϵ. We presuppose general familiarity with the Chomsky hierarchy, including the four classes of *recursively enumerable languages* (RE, type-0), *context-sensitive languages* (CSL, type-1), *context-free languages* (CFL, type-2), and *regular languages* (REG, type-3).

Ample introductions can be found in [15, 26]. For (constructor) term rewriting, we refer to [27] for basic definitions; we very briefly recapitulate the most pertinent notions in the below definition.

▶ **Definition 1.** *We assume a denumerably infinite set* Var *of* variables*; given a signature* Σ *of symbols with non-negative integer arities, we define the set of terms* $\mathrm{Ter}(\Sigma, \mathrm{Var})$ *over* Σ *and* Var *inductively as usual:* $\mathrm{Var} \subseteq \mathrm{Ter}(\Sigma, \mathrm{Var})$ *and if* $s_1, \ldots, s_n \in \mathrm{Ter}(\Sigma, \mathrm{Var})$ *and* $f \in \Sigma$ *has arity* n*, then* $f(s_1, \ldots, s_n) \in \mathrm{Ter}(\Sigma, \mathrm{Var})$*.*

A rule *is a pair of terms, written* $l \to r$ *such that* l *and* r *are terms with* $l \notin \mathrm{Var}$ *and such that every variable occurring in* r *occurs in* l*. A* term rewriting system *(abbreviated* TRS*) is a set of rules.*

Let $\Sigma = \mathcal{F} \cup \mathcal{C}$ *where* $\mathcal{F}$ *and* $\mathcal{C}$ *are disjoint sets of* defined symbols *and* constructor *symbols, respectively. A* constructor TRS *is a TRS where each rule* $l \to r$ *satisfies* $l = f(t_1, \ldots, t_n)$ *where* $f \in \mathcal{F}$ *and* $t_1, \ldots, t_n \in \mathrm{Ter}(\mathcal{C}, \mathrm{Var})$*.*

A TRS is said to be monadic *if the arity of all function and constructor symbols is at most* 1*. If* R *is monadic and* $l \to r$ *is a rule of* R*, we occasionally write* $l(x) \to r(x)$ *where* x *is the unique variable occurring in* l *and* r *(and we extend the notation to the case where there are no variables in* l *or* r *– in which case the choice of* x *does not matter).*

A substitution *is a partial map* $\theta : \mathrm{Var} \longrightarrow \mathrm{Ter}(\Sigma, \mathrm{Var})$*. In monadic systems, each term* s *contains at most one variable, and we shall write* $s\theta$ *for the term obtained by replacing the variable* x *in* s *by* $\theta(x)$ *(if* $x \in \mathrm{dom}(\theta)$*).*

A context *in a monadic TRS is a term over the variable set* $\mathrm{Var} \cup \{\Box\}$ *where* $\Box \notin \Sigma \cup \mathrm{Var}$*; if* C *is a context and* w *is a term, we denote by* $C[w]$ *the term obtained by replacing the (unique!)* $\Box$ *in* C *by* w*. For* $s, t \in \mathrm{Ter}(\Sigma, \mathrm{Var})$*, we write* $s \to t$ *if there is a context* C*, a rule* $l \to r$*, and a substitution* θ *such that* $s = C[l\theta]$ *and* $t = C[r\theta]$*, and we call* $(C, l \to r, \theta)$ *a* redex *in* s*; The redex is said to be* contracted *in the* step $s \to t$*. The* position *of a redex is* 1^k *where* k *is the number of symbols in* C *(we set* $1^0 = \epsilon$*); we say that the rule* $l \to r$ *is* applied to s at position p*. We write* $\to^*$ *for the reflexive, transitive closure of* $\to$ *and* $\to^+$ *as the transitive closure. We call* $s \to^* t$ *a* reduction *or* rewrite sequence*.*

Two redexes $(C, l \to r, \theta)$ *and* $(C', l' \to r', \theta')$ *in* $s = C[l\theta] = C'[l'\theta']$ overlap *if a symbol in* l *and a symbol in* l' *share the same position in* $C[l\theta] = C'[l'\theta']$*.*

A redex v *at position* p *in* s *is* innermost *if, for any redex* w *at position* $p' > p$*,* w *overlaps* v *(intuitively:* v *is innermost if no other redex occurs "to the right of* v*"). The* size *of a term* s *in a monadic TRS is defined by induction as:* $|s| = 1$ *if* s *is a variable or a nullary function symbol, and* $|s| = 1 + |s'|$ *if* $s = g(s')$ *where* $g \in \Sigma$*.*

Throughout the paper, we assume that all rewrite systems have a *finite* set of rules.

▶ **Definition 2.** *Let* A *be an alphabet and* $\rhd$ *a nullary constructor symbol. For every* $a \in A$*, we associate a unary constructor symbol* $\widetilde{a}$*, and we define* $\widetilde{A} = \cup_{a \in A}\{\widetilde{a}\}$*. For any string* $\alpha = a_1 \cdots a_n \in A^+$*, we associate the constructor term* $\widetilde{\alpha} = \widetilde{a}_1(\cdots \widetilde{a}_n(\rhd))$*, and set* $\widetilde{\epsilon} = \rhd$*.*

▶ Remark 3. Throughout the paper, *every* term is built from unary or nullary symbols. Hence, there is a natural correspondence between terms and strings: If $f_1, \ldots, f_m$ are unary symbols and b is nullary, then $f_1(f_2(\cdots f_m(b)))$ corresponds to the string of symbols $f_1 f_2 \cdots f_m b$.

▶ **Definition 4.** *Let* A *be an alphabet and let* R *be a constructor TRS with* $\Sigma = \mathcal{F} \cup \mathcal{C}$*, such that* $(\widetilde{A} \cup \{\rhd\}) \subseteq \mathcal{C}$ *where* $\rhd$ *is a nullary symbol.* R *is said to* accept $L \subseteq A^+$ *if there is a defined symbol* $f_0 \in \mathcal{F}$ *– the "start function" – such that for every* $\alpha \in A^+$*, there is a reduction* $f_0(\widetilde{\alpha}) \to^* \rhd$ *iff* $\alpha \in L$*.*

▶ Remark 5. The use of $\rhd$ in $f_0(\widetilde{\alpha}) \to^* \rhd$ can be replaced by a fresh constructor (instead of the "nil" constructor $\rhd$), or a nullary defined symbol when characterizing the classes RE or CSL. For CFL and RE, we consider systems where rules cannot contain any constructors in the right-hand side; there, acceptance by $\rhd$–the last constructor in the representation $\tilde{\alpha}$ of any string $\alpha \in A^+$–is completely natural (it could wlog. be replaced by introducing rules of the form $f(\rhd) \to h$ with h a nullary symbol in $\mathcal{F}$, but there seems to be no good reason to do so).

▶ **Definition 6.** *Let R be a monadic constructor TRS with alphabet $\Sigma = \mathcal{F} \cup \mathcal{C}$. R is said to be* tail recursive *if there is a preorder $\leq$ on $\mathcal{F}$ such that for every rule $f(w) \to r$ in R and every occurrence of a defined symbol $g \in \mathcal{F}$ in r, either (i) $f > g$, or (ii) $f \geq g$ and the occurrence of g is at position ϵ.*

The reason for requiring $\leq$ to be (only) a preorder as in [8] (rather than a partial order as in, e.g. [18]) is that recursion should be limited to tail calls (so, in rewriting terms, at the root of the rhs), but that the tail call does not need to be the same defined symbol as in the left-hand-side, merely a symbol having the same rank in the $\leq$-order.

3 Recursively enumerable languages: General monadic systems

For each of the class of languages we consider, we first remind the reader of their associated class of accepting machine; for recursively enumerable languages, these are Turing machines.

▶ **Definition 7.** *A (one-tape, non-deterministic)* Turing machine *is a tuple $(Q, A, \Gamma, \delta, q_0, q_h)$ where Q is a set of states, A is the input alphabet (which does not contain blanks), Γ is the tape alphabet (with $A \subseteq \Gamma$ and a designated symbol $\Box \in \Gamma$ representing "blank"), $\delta : Q \times \Gamma \longrightarrow \mathcal{P}(Q \times (\Gamma \cup \{\epsilon\}) \times \{L, R\})$ is the transition function, $q_0 \in Q$ is the start state, and $q_h \in Q$ is the accept state.*

We write $\delta(q, a) \to (q', b, H)$ if $(q', b, H) \in \delta(a, b)$; note that several such transition rules may exist for each (q, a). On a transition rule $\delta(q, a) \to (q', b, H)$, the machine is said to *transition*, when reading symbol a in state q, to state q', writing symbol b (or not writing anything when $b = \epsilon$), and moving either left or right on the tape, according to whether $H = L$ or $H = R$.

As usual, we define Turing machine configurations as a (tape contents, tape head position, state)-triple:

▶ **Definition 8.** *A* configuration *of a machine $M = (Q, \Gamma, A, \delta, q_0, q_h)$ is a triple (T, n, q) where $T \in \Gamma^+$ is the current content of the tape (disregarding the infinite strings of blanks to the left and right of the portion of the tape that contains the input and the set of cells scanned by M so far; T is assumed to have length at least 1, possibly consisting of a single blank symbol), n is an integer where $1 \leq n \leq |T|$ (the position of the tape head in T), and $q \in Q$. M* transitions in one step *on configuration (T, n, q) to configuration (T', n', q')* on transition *$\delta(q, b) \to (q', b', H)$ if the nth element of T is b and (T', n', q') represents the machine state after the corresponding move. We say that "A transitions to B" if A reduces to B by a sequence of ≥ 0 steps. M* accepts *input α if it transitions from $(\alpha, 1, q_0)$ to a configuration (T, n, q_h) – we also say that M transitions to q_h on input α. A language $L \subseteq A^+$ is* recursively enumerable *if there is a Turing machine that, for each $\alpha \in A^+$, accepts α iff $\alpha \in L$.*

Huet and Lankford proved that monadic TRSs can simulate Turing machines [16]. For completeness, we re-prove Huet and Lankford's result in a new setting, giving a new proof of simulation of Turing machines by constructor TRSs with unary function and constructor

$$M = (Q, A, \Gamma, \delta, q_0, q_h)$$

Sets of function and constructor symbols:

$$\mathcal{F} = \{f_q : q \in Q\} \cup \{f_b : b \in \Gamma\} \cup \{f_0, f_\triangleleft\}$$
$$\mathcal{C} = \{\mathtt{b} : b \in \Gamma\} \cup \{\mathtt{c}_q : q \in Q\} \cup \{\triangleright\}$$

Note: $f_\Box \in \mathcal{F}$ and $\Box \in \mathcal{C}$. If $b \in A$, then $\mathtt{b} = \tilde{b}$. The constructors $\mathtt{c}_q$ are fresh symbols designed to "store" the state of M on left-moves.

Rewrite rules induced by transition rules of M:

(L/R)-move	rewrite rules (for each $q \in Q$, $b \in \Gamma$)
$\delta(q,b) \to (q', b', R)$	$f_q(\mathtt{b}(x)) \to f_{b'}(f_{q'}(x))$
$\delta(q,b) \to (q', \epsilon, R)$	$f_q(\mathtt{b}(x)) \to f_b(f_{q'}(x)))$
$\delta(q,b) \to (q', b', L)$	$f_q(\mathtt{b}(x)) \to \mathtt{c}_{q'}(\mathtt{b}'(x))$
$\delta(q,b) \to (q', \epsilon, L)$	$f_q(\mathtt{b}(x)) \to \mathtt{c}_{q'}(\mathtt{b}(x))$

Housekeeping rules for endmarkers:

rewrite rules (for each $q \in Q \setminus \{q_h\}$)
$f_q(\triangleright) \to f_q(\Box(\triangleright))$
$f_\triangleleft(\mathtt{c}_q(x)) \to f_\triangleleft(f_q(\Box(x))))$

Propagation and start rules:

rewrite rules (for each $q \in Q$, $b \in \Gamma$)
$f_{q_h}(x) \to \mathtt{c}_{q_h}(x)$
$f_\triangleleft(\mathtt{c}_{q_h}(x)) \to \triangleright$
$f_b(\mathtt{c}_q(x)) \to f_q(\mathtt{b}(x))$
$f_0(x) \to f_\triangleleft(f_{q_0}(x))$

Figure 2 Basic encoding $\Delta^1(M)$ of a Turing machine M: The part of the tape to the left of the single read/write head is represented by a string of defined symbols f, and the part to the right by a string of constructor symbols $\mathtt{c}$.

symbols. The constructor TRS simulating a given Turing machine is given in Figure 2; the construction is similar to that given in [27], but uses only unary function and unary and nullary constructor symbols. The simulation serves as illustration of an observation we shall exploit throughout the paper: A term $f_1(\cdots f_m(\mathtt{c}_1(\cdots \mathtt{c}_n(\triangleright))))$ may be viewed as composed of a "call stack" of defined symbols and a "memory stack" of constructor symbols; intuitively, we thus obtain the expressive power of the (Turing-complete) class of 2-counter machines [23].

The following is tedious, but not hard, to prove:

▶ **Lemma 9.** *Let $\alpha \in A^+$. Then, $f_0(\widetilde{\alpha}) \to^* \triangleright$ iff M transitions to q_h on input α.*

We then have:

▶ **Theorem 10.** *Let $L \subseteq A^+$. The following are equivalent: (i) L is recursively enumerable, (ii) L is accepted by a monadic constructor TRS.*

Proof. If R is a monadic constructor TRS accepting L, we may construct a (non-deterministic) Turing machine accepting L by encoding the finitely many rules of R and non-deterministically applying the rules to input α. If this simulation reaches $\triangleright$, the machine halts. It therefore suffices to prove that every recursively enumerable language is accepted by a monadic constructor TRS. By standard results (see e.g. [25, Thm. 17.2]), L is recursively enumerable iff it is accepted by a non-deterministic Turing machine. Lemma 9 then furnishes that, for each $\alpha \in A^+$, we have $f_0(\widetilde{\alpha}) \to^* \triangleright$ iff M accepts α, as desired. ◀

4 Context-sensitive languages: Non-length-increasing rules

The class of context-sensitive languages is characterized by its class of acceptors: the linearly bounded automata. The following definition is standard.

▶ **Definition 11.** *A non-deterministic Turing (multi-tape) machine M accepts $L \subseteq A^+$ in non-deterministic linear space if there is a k such that all computation branches halts on all inputs and any computation scans at most $k \cdot |x|$ distinct cells on each of its tapes. The set of languages acceptable by such machines is called* NLINSPACE.

▶ **Proposition 12.** *Let $L \subseteq A^+$ be a language accepted by an m-tape Turing machine in space $O(n)$. Then there is a* one-*tape Turing machine with input alphabet A (but possibly a much larger tape alphabet) that accepts L in space $\leq n$ on all inputs.*

Proof. Standard exercise in linear space reduction, see e.g. [17, Prop. 21.1.5] for the reduction to one-tape machines (at the cost of an input-independent constant factor of more space use), and [24] for the technique of getting rid of constant space factors on one-tape machines. ◀

By Proposition 12 we may restrict attention to machines that accept their input using no more space than that originally allocated to the input: The linear bounded automata. To ensure that linear bounded automata do not exceed their tape allowance, we make the provision that inputs are always bookended by special *stoppers* ◀ and ▶. For example, if $A = \{0, 1\}$ the string 10010 will be fed to the automaton as ◀ 10010 ▶.

▶ **Definition 13.** *A* linear bounded automaton *(LBA) over alphabet A is a one-tape Turing machine $(Q, A, \Gamma, \delta, q_0, q_h)$ with input alphabet $A' = A \cup \{◀, ▶\}$ and where ◀ and ▶ are the* left *and* right *stoppers, respectively, and such that: (i) $◀, ▶ \notin \Gamma$, (ii) for every rule $\delta(q, b) \to (q', b', H)$, we have $b' \notin \{◀, ▶\}$ (i.e. stoppers are not written on the tape), (iii) for every rule $\delta(q, ◀) \to (q', b', H)$, we have $b' = \epsilon$ and $H = R$ (i.e. the left stopper is not overwritten, and the tape head cannot move left of the left stopper), (iv) for every rule $\delta(q, ▶) \to (q', b', H)$, we have $b' = \epsilon$ and $H = L$ (the right stopper is not overwritten and the tape head cannot move to the right of the stopper endmarker). An LBA is said to* accept *input $\alpha \in A^+$ if its underlying Turing machine accepts the string ◀ α ▶.*

Thus: A linear bounded automaton can *only* use the space that its input originally occupies: Space *exactly* n where n is the size of the input. The following proposition makes this precise:

▶ **Proposition 14.** *Let M be an LBA. If $(◀\, b_1 \cdots b_m\, ▶, n, q)$ is a configuration of M and $b_1, \ldots, b_m \in \Gamma \backslash \{◀, ▶\}$, and M transitions to configuration (T', n', q'), then $T' = ◀\, b'_1 \cdots b'_m\, ▶$ for $b'_1, \ldots, b'_m \in \Gamma \setminus \{◀, ▶\}$.*

Proof. By the assumptions on the form of the rules of the LBA in Definition 13, neither of the symbols ◀ and ▶ can be overwritten by M, nor can any symbol be overwritten by ◀ or ▶. By the same assumptions on the form of rules, M cannot move to the left of a ◀, nor to the right of a ▶. ◀

▶ **Theorem 15.** *Let $L \subseteq A^+$. The following are equivalent: (i) L is accepted by an LBA, (ii) L is context-sensitive, (iii) $L \in$* NLINSPACE.

Proof. Standard textbook exercise, see e.g. [21, Exerc. 6.29], or [25, Thm. 24.3]. For the original proof, see [20]. ◀

For every $(a,b,d) \in A^3$:

$M_{abd} = (Q_{abd}, A \cup \{\blacktriangleleft, \blacktriangleright\}, \Gamma, \delta_{abd}, q_0^{\mathrm{abd}}, q_h)$

(where M_{abd} is given by Proposition 16)

$\mathcal{F}_{\mathrm{abd}} = \{f_q : q \in Q_{abd}\} \cup \{f_b : b \in \Gamma\} \cup \{f_\triangleleft\}$

$\mathcal{C}_{\mathrm{abd}} = \{\mathtt{b} : b \in \Gamma\} \cup \{\mathtt{c}_q : q \in Q_{\mathrm{abd}}\} \cup \{\triangleright\}$

Note that as $A \cup \{\blacktriangleleft, \blacktriangleright\} \subseteq \Gamma$, we have $f_\blacktriangleleft, f_\blacktriangleright \in \mathcal{F}_{\mathrm{abd}}$, and $\blacktriangleleft, \blacktriangleright \in \mathcal{C}_{\mathrm{abd}}$.

Rules induced by transition rules of M_{abd}:

(L/R)-move	rewrite rules ($q \in Q_{\mathrm{abd}}$, $b \in \Gamma$)
$\delta(q,b) \to (q', b', R)$	$f_q(\mathtt{b}(x)) \to f_{b'}(f_{q'}(x))$
$\delta(q,b) \to (q', \epsilon, R)$	$f_q(\mathtt{b}(x)) \to f_b(f_{q'}(x)))$
$\delta(q,b) \to (q', b', L)$	$f_q(\mathtt{b}(x)) \to \mathtt{c}_{q'}(\mathtt{b}'(x))$
$\delta(q,b) \to (q', \epsilon, L)$	$f_q(\mathtt{b}(x)) \to \mathtt{c}_{q'}(\mathtt{b}(x))$

Propagation rules:

rewrite rules ($q \in Q_{\mathrm{abd}}$, $b \in \Gamma$)
$f_{q_h}(x) \to \mathtt{c}_{q_h}(x)$
$f_\triangleleft(\mathtt{c}_{q_h}(x)) \to \triangleright$
$f_b(\mathtt{c}_q(x)) \to f_q(\mathtt{b}(x))$

Figure 3 Non-length-increasing constructor TRS defined from an LBA M_{abd}. Observe that $f_0 \notin \mathcal{F}_{\mathrm{abd}}$ and that the constructor TRS will not accept any strings on its own. .

Due to our convention that constructor TRSs must start their computations on terms on the form $f_0(\widetilde{\alpha})$, we encounter the problem that non-length-increasingness prevents us from setting up the simulation of the LBA tape and state: We would need a rule of the form $f_0(x) \to f_\triangleleft(f_{q_0}(\cdots))$. The problem is solved by the following proposition:

▶ **Proposition 16.** *Let LBA M accept the language $L \subseteq A^+$ and let $(a,b,d) \in A^3$. Then there exists an LBA M_{abd} with input and tape alphabets identical to those of M that accepts the language $L' = \{\beta \in A^* : abd \cdot \beta \in L\}$.*

Proof. If the input to M has size $n \geq 3$, we may encode all possible configurations of the leftmost 3 cells of the tape of M in $|\Gamma|^3$ states. If M has $|Q|$ states, we may construct an LBA M_{abd} with $(4|\Gamma|^3) \times |Q|$ states that encodes any changes to the leftmost 3 cells in its states (the factor 4 is used by M_{abd} to keep track of where the tape head is (either of the first three "cells" encoded by the states, or to their right), and only uses $n-3$ tape cells (where it simply simulates M) . ◀

For each LBA M and $(a,b,d) \in A^3$, we define a non-length-increasing constructor TRS $\Delta_{LBA}^{\mathrm{abd}}(M)$ by the translation given in Figure 3 – effectively the same translation as that in Figure 2, except for the absence of a start rule and the addition of rules for stopper fitting. For each LBA M, we define a corresponding non-length-increasing system $\Delta_{LBA}(M)$ by taking the union of all rules from all of the $|\Gamma|^3$ LBAs M_{abd} and adding rules to start the computation. The resulting constructor TRS is shown in Figure 4.

▶ **Proposition 17.** *If M is an LBA, then $\Delta_{LBA}(M)$ is a non-length-increasing monadic constructor TRS.*

Proof. Observe that every rule of M is translated by $\Delta_{LBA}(\cdot)$, whence $\Delta_{LBA}(M)$ is defined for all M. Furthermore, every rule of $\Delta_{LBA}(M)$ is non-length-increasing, and the general result follows. ◀

$$\mathcal{F} = \left(\bigcup_{(a,b,d) \in A^3} \mathcal{F}_{\mathrm{abd}} \right) \cup \{f_0\} \quad \mathcal{C} = \bigcup_{(a,b,d) \in A^3} \mathcal{C}_{\mathrm{abd}}$$

The rules of $\Delta_{LBA}(M)$ are the union of $\bigcup_{(a,b,c) \in A^3} R_{\mathrm{abd}}$ with the set of rules below:

Start rules and stopper rules:

rewrite rules (for each $a, b, d \in A$, not necessarily distinct)	
$f_0(\widetilde{a}(\rhd)) \to \rhd$	if M accepts a
$f_0(\widetilde{a}(\widetilde{b}(\rhd))) \to \rhd$	if M accepts ab
$f_0(\widetilde{a}(\widetilde{b}(\widetilde{d}(x)))) \to f_{\lhd}(f_{q_0^{\mathrm{abd}}}(\blacktriangleleft (e(x))))$	
$e(\widetilde{a}(x)) \to \widetilde{a}(e(x))$	
$e(\rhd) \to \blacktriangleright (\rhd)$	

Figure 4 Encoding Δ_{LBA}(M) of an LBA M as a non-length-increasing system.

Again, the following is tedious, but not hard, to prove:

▶ **Lemma 18.** *Let $\alpha \in A^+$. Then LBA M transitions to the halting state on input $\blacktriangleleft \alpha \blacktriangleright$ iff $f_0(\widetilde{\alpha}) \to^*_{\Delta_{LBA}(M)} \rhd$.*

We can now prove the main result of the section:

▶ **Theorem 19.** *Let $L \subseteq A^+$. The following are equivalent: (i) L is context-sensitive, (ii) L is accepted by a monadic non-length-increasing constructor TRS.*

Proof. If L is context-sensitive, it is accepted by an LBA M by Theorem 15. Then, Lemma 18 furnishes that $\Delta_{LBA}(M)$ accepts L, and by Proposition 17, $\Delta_{LBA}(M)$ is a monadic non-length-increasing constructor TRS. Conversely, if L is accepted by a monadic non-length-increasing constructor TRS R over alphabet Σ, we can define a non-deterministic Turing machine with tape alphabet Σ that runs in linear space and accepts L: In every rule $l \to r$, both l and r are terms over unary and nullary symbols, hence essentially strings. As $|l| \geq |r|$, a rewrite step corresponds to replacing a substring by a substring of at most the same size. Thus, we may simply encode the rules of R in the states M. The current state of the term $f_1(f_2(\cdots f_m(b)))$ is encoded in $m+1$ symbols $f_1 f_2 \cdots f_m b$ on the Turing machine's tape, and application of a rule is simply done by replacing the symbols on the relevant tape cells. Choosing what rule to apply and where to apply it is selected non-deterministically by M. As $|l| \geq |r|$, the number of tape cells used will never increase, whence the machine runs in linear space, and Theorem 15 furnishes that L is context-sensitive. ◀

5 Context-Free Languages: (Strongly) cons-free systems

We now treat context-free languages; we first need their corresponding notion of accepting machine.

▶ **Definition 20.** *A* pushdown automaton *(PDA) is a tuple $(Q, A, \Gamma, \delta, q_0, Z_0)$ where Q is a finite set of* states, *A is a finite set of* input symbols, *Γ is a finite* stack alphabet, *$q_0 \in Q$ is the* start state, *$Z_0 \in \Gamma$ is the* start stack symbol, *and δ is a relation consisting of a finite number of transition rules of the form $\delta(q, a, X) \to (p, \gamma)$ where $q \in Q$, $a \in A \cup \{\epsilon\}$, $X \in \Gamma$, $p \in Q$, and $\gamma \in \Gamma^*$.*

The definition of PDA above has no final states, and will thus accept by empty stack (and empty input), as is common in the literature [26]. We make the convention that the bottom of the stack is written to the left and the top to the right; hence, symbols are pushed and popped to the right.

As we shall only consider one-state PDAs in this paper; the below definition of acceptance has been specialized to that case (for the general case, see any standard textbook, e.g. [26]):

▶ **Definition 21.** *A one-state PDA is said to* accept *input* $\alpha \in A^+$ *if* $\alpha = a_1 \cdots a_m$ *where each* $a_i \in A \cup \{\epsilon\}$ *and there is a sequence of strings* $s_1, \ldots, s_m$ *from* Γ^* *such that: (i)* $s_0 = Z_0$*, (ii) for* $i = 0, \ldots, m-1$*, there is a rule* $\delta(a_{i+1}, Z) \to Z'$ *where* $s_i = tZ$ *and* $s_{i+1} = tZ'$ *for some* $Z, Z' \in \Gamma \cup \{\epsilon\}$ *with* $Z \neq \epsilon$*, and* $t \in \Gamma^*$ *(that is, the PDA moves according to the stack and next input symbol)*[1]*, (iii)* $a_m = \epsilon$ *and* $s_m = \epsilon$ *(that is, empty input and empty stack are reached at the end). Otherwise, the PDA is said to* reject *the input.*

The following proposition is standard; see for example [13] for a proof.

▶ **Proposition 22.** *If* $L \subseteq A^*$ *is accepted by a PDA, it is accepted by a one-state PDA* $(\{q_0\}, A, \Gamma', \delta, q_0, Z_0)$ *(where we assume acceptance by empty stack).*

The following theorem is standard (see e.g. [26, Thm. 2.12])

▶ **Theorem 23.** *A language* $L \subseteq A^+$ *is context-free iff it is accepted by a PDA with input alphabet* A*.*

By Proposition 22 and Theorem 23, a language is thus context-free iff it is accepted by a one-state PDA.

As with the language classes RE and CSL, we shall prove that a particular class of monadic rewrite systems corresponds to CFL; this class consists of the (strongly) cons-free systems:

▶ **Definition 24.** *A constructor TRS is said to be (strongly)* cons-free *if, for every rule* $l \to r$ *there are no constructor symbols in* r*.*

▶ Remark 25. Cons-freeness has been used for multiple characterizations of complexity classes (see, e.g., [18, 5, 19, 8]). The gist is that, during rewriting, no new constructor terms can be built; thus, the definition of cons-freeness is usually less restrictive than the *strong* cons-freeness of Definition 24 [19, 8][2], but we believe that the restriction to the very simple notion of strong cons-freeness is cleaner and simpler to work with here.

▶ Remark 26. As pointed out by a referee, there are likely simpler grammar-based proofs that the class of strongly cons-free constructor TRSs characterizes CFL compared to the one we give using PDAs. However, the proof via PDAs shed light on the intuition that rewriting in monadic constructor TRSs essentially consist of manipulation of up to two stacks – and that for (strongly) cons-free systems, the manipulation is essentially a single "general" stack and a "restricted" stack that can only be decremented, exactly as in a PDA.

The following proposition shows that we may disregard nullary defined symbols in the remainder of the paper:

---

[1] As the PDA has only a single state, we have suppressed the state in the notation of the rule $\delta(a_{i+1}, Z) \to Z'$.

[2] For example, cons-freeness of a rule $l \to r$ in [8] is defined as the requirement that every subterm of the form $\mathsf{c}(s)$ in r (where $\mathsf{c} \in \mathcal{C}$) either occurs in l, or is a ground constructor term.

$M = (\{q_0\}, A, \Gamma, \delta, q_0, Z_0)$

$\mathcal{F} = \{f_Z : Z \in \Gamma\} \cup \{f_0\}$ and $\mathcal{C} = \{\widetilde{a} : a \in A\} \cup \{\rhd\}$

Rewrite rules induced by transition rules in δ:

transition rule in δ	rule of R_m
$\delta(a, Z) \to Z_1 \cdots Z_m$	$f_Z(\widetilde{a}(x)) \to f_{Z_1}(\cdots f_{Z_m}(x))$
$\delta(a, Z) \to \epsilon$	$f_Z(\widetilde{a}(x)) \to x$
$\delta(\epsilon, Z) \to Z_1 \cdots Z_m$	$f_Z(x) \to f_{Z_1}(\cdots f_{Z_m}(x))$
$\delta(\epsilon, Z) \to \epsilon$	$f_Z(x) \to x$

Start rule:

rewrite rules
$f_0(x) \to f_{Z_0}(x)$

Figure 5 Rules of the cons-free system R_M induced by the PDA M. As M has only one state, the state argument has been omitted from δ.

▶ **Proposition 27.** *Let R be a cons-free, monadic constructor TRS that accepts $L \subseteq A^+$, and let R' be the monadic, cons-free constructor TRS obtained by omitting all rules in R that contain a nullary defined symbol. Then R' accepts L.*

To greatly simplify our proofs for context-free and regular languages, we introduce *normal* systems:

▶ **Definition 28.** *A rule $l \to r$ is* normal *if l contains at most one constructor symbol, and that constructor symbol is unary, that is either $l = f(\mathsf{c}(x))$, or $l = f(x)$ (for some $f \in \mathcal{F}$ and $\mathsf{c} \in \mathcal{C}$). A constructor TRS R is* normal *if every rule is normal.*

The following lemma shows that we can transform a set of rules with “large” left-hand sides into (a larger set of) normal rules that accept the same language:

▶ **Lemma 29.** *If $L \subseteq A^+$ is accepted by a monadic, cons-free constructor TRS R, then L is accepted by a monadic, cons-free, normal constructor TRS R' with $\mathcal{C} = \widetilde{A} \cup \{\rhd\}$. If R is tail recursive, then R' may be chosen to be tail recursive as well.*

For each one-state PDA, we define a cons-free constructor TRS R_M as given in Figure 5.

In Figure 5, the presence of transition rules of the form $\delta(\epsilon, Z) \to r$ force us to let R_M contain rules of the form $f(x) \to r'$. By the definition of TRSs, application of such a rule may occur anywhere in a term. However, as we want to simulate the PDA stack by a string of defined symbols, applying a rule $f(x) \to r'$ corresponds to removing a symbol in the middle of the stack rather than popping it off the top. Hence, we are forced to require that redexes in R_M are contracted only at places corresponding to the top of the stack – which is the case if the redexes are *innermost*. This is also sufficient, as we shall see shortly.

▶ **Definition 30.** *Let p be a non-negative integer. A ground term s has a* border position *at p if $s = f_1(\cdots(f_p(t))\cdots)$ where $p \geq 1$, $f_1, \ldots, f_p \in \mathcal{F}$ and t is a ground constructor term.*

The following proposition is proved by induction on the length of the involved rewrite sequence:

▶ **Proposition 31.** *Let R be a monadic, cons-free constructor TRS. If t is a ground term with a border position such that $t \longrightarrow^* \rhd$, then every term in the rewrite sequence, except the last, has a border position, and an innermost redex at the border position.*

Even if R contains overlapping redexes, innermost rewite steps can be retracted across non-innermost ones (and efficiently so, as monadic systems cannot make more than a single copy of each subterm):

▶ **Proposition 32.** *Let R be a monadic, cons-free constructor TRS, let s be a term containing a redex at a border position, and let $m \geq 0$. If $s \rightarrow^k t'$ by non-innermost steps, and $t' \rightarrow_{IM} t$, then there is a term s' such that $s \rightarrow_{IM} s' \rightarrow^k t$, where $\rightarrow_{IM}$ is innermost reduction.*

Proof. As R is a constructor TRS, every redex at a border position is innermost, whence s contains an innermost redex. As every non-innermost redex cannot overlap an innermost redex, all innermost redexes in s are preserved across any non-innermost reduction, and remain innermost. Consider the redex u contracted in the step $t' \rightarrow t$; as the innermost redex at border position in s is preserved across $s \rightarrow^k t'$, it overlaps with u. But as the left-hand sides of all redexes in R are of the form $f(w)$ where w is a constructor term, no redexes created in the reduction $s \rightarrow^k t'$ can overlap with the descendants of redexes at innermost position in s. Hence, u is the descendant of an innermost redex u' in s. Furthermore, contracting an innermost redex cannot destroy any redexes except those that overlap with it (and are thus, by definition, also innermost), and thus we may contract u' to obtain the step $s \rightarrow_{\text{IM}} s'$, followed by mimicking the steps in $s \rightarrow^k t'$ starting from the term s' (all of which can be performed, as u' does not overlap with any non-innermost redex). Thus, $s' \rightarrow^k t$, concluding the proof. ◀

▶ **Lemma 33.** *Let R be a monadic, cons-free, normal constructor TRS. If $s = f_0(\tilde{\alpha}) \rightarrow^* \rhd$, then $s \longrightarrow^*_{IM} \rhd$.*

Proof. By Proposition 31, every term in $s \longrightarrow^* \rhd$, except the last, contains an innermost redex at a border position. Divide $s \longrightarrow^* \rhd$ into subsequences, each of the form $s' \rightarrow^*_{\text{IM}} s'' \rightarrow^k t' \rightarrow^+_{\text{IM}} t''$ where $s'' \rightarrow^k t'$ consists solely of non-innermost steps for some $k \geq 1$. Observe that this is always possible because the last step of $s \longrightarrow^* \rhd$ must be innermost as R is cons-free and $\rhd$ is a constructor. By repeated application of Proposition 32, we obtain $s' \rightarrow^+_{\text{IM}} s''' \rightarrow^k t''$ for some term s'''. Hence, a straightforward induction on the length of $s \longrightarrow^* \rhd$ shows that all innermost steps can be retracted across non-innermost steps, resulting in a reduction $s \rightarrow^*_{\text{IM}} t''' \rightarrow^* \rhd$ of length no more than the original where $t''' \rightarrow^* \rhd$ contains no innermost steps. But as the last step of any reduction $s \rightarrow^* \rhd$ must be innermost, the length of $t''' \rightarrow^* \rhd$ is zero, and thus $s \rightarrow^*_{\text{IM}} \rhd$, as desired. ◀

As with our previous simulation results, the following result is tedious to prove, but not difficult:

▶ **Lemma 34.** *Let M be a one-state PDA accepting language $L \subseteq A^+$. Then R_M accepts L by innermost evaluation.*

We now show how to simulate any cons-free constructor TRS by a one-state PDA. We consider only normal systems, as this suffices by Lemma 29. For any normal, monadic, cons-free constructor TRS with $\mathcal{C} = \widetilde{A} \cup \{\rhd\}$, we define a one-state PDA as shown in Figure 6.

Again, the following is tedious, but fairly straightforward:

▶ **Lemma 35.** *Let $L \subseteq A^+$ be accepted by innermost reduction by a normal, cons-free, monadic constructor TRS R. Then* PDA_R *accepts L.*

We thus have:

▶ **Theorem 36.** *The following are equivalent for a language $L \subseteq A^+$: (i) L is context-free, (ii) L is accepted by a monadic cons-free constructor TRS.*

$\text{PDA}_R = (\{q_0\}, A, \{Z_f : f \in \mathcal{F}\}, \delta, q_0, Z_{f_0})$

rule of R	transition rule in δ
$f(\mathsf{c}(x)) \to f_1(\cdots f_m(x))$	$\delta(\mathsf{c}, Z_f) \to Z_{f_1} \cdots Z_{f_m}$
$f(x) \to f_1(\cdots f_m(x))$	$\delta(\epsilon, Z_f) \to Z_{f_1} \cdots Z_{f_m}$
$f(\mathsf{c}(x)) \to x$	$\delta(\mathsf{c}, Z_f) \to \epsilon$
$f(x) \to x$	$\delta(\epsilon, Z_f) \to \epsilon$

Figure 6 Definition of the pushdown automaton PDA_R from a normal, monadic, cons-free constructor TRS R with signature $\mathcal{F} \cup \mathcal{C} = \mathcal{F} \cup (\tilde{A} \cup \{\rhd\})$.

Proof. If L is context-free, Theorem 23 yields that L is accepted by a PDA M, which we may assume by Proposition 22, has exactly one state. Lemma 34 yields that R_M accepts L by innermost reduction, and Lemma 33 shows that the elements of A^* accepted by R_M are exactly those accepted by innermost reduction. Clearly, R_M is a monadic, cons-free constructor TRS.

Conversely, if L is accepted by a monadic, cons-free constructor TRS R, Lemma 33 yields that R accepts L by innermost reduction, and by Lemma 29 we may assume wlog. that R is normal. Lemma 35 now shows that PDA_R accepts L, whence L is context-free by Theorem 23. ◀

6 Regular languages: tail recursive cons-free systems

We shall now consider the class of *regular* languages. We assume the reader to be familiar with the fact that a language is regular iff it is accepted by an NFA iff it is accepted by a DFA. To fix notation, we give the following definition:

▶ **Definition 37.** *A* non-deterministic finite automaton *(NFA) is a tuple* $(Q, A, \delta, q_0, Q_{\text{accept}})$ *such that* Q *is a non-empty set of* states, A *is the* input alphabet, δ *is a set of transition rules on one of the forms* $\delta(q, a) \to q'$ *or* $\delta(q, \epsilon) \to q'$ *where* $q, q' \in Q$ *and* $a \in A$, $q_0 \in Q$ *is the start state, and* $Q_{\text{accept}} \subseteq Q$ *is the set of accept states. Furthermore, for any* $q \in Q$ *and any* $a \in A$*, there is at least one transition of the form* $\delta(q, a) \to q'$*. A* deterministic finite automaton *(DFA) is an NFA such that there are* no *transitions of the form* $\delta(q, \epsilon) \to q'$*, and if there is a transition of the form* $\delta(q, a) \to q'$*, then there is no transition* $\delta(q, a) \to q''$ *with* $q' \neq q''$*.*

The class REG is characterized by the monadic constructor TRSs that are both cons-free and *one-call* (see Definition 40).

In tail-recursive functional programming, the height of the call stack is bounded above by a constant; a similar result holds here for *innermost* reduction:

▶ **Proposition 38.** *Let* R *be a monadic, normal, cons-free, tail-recursive constructor TRS. Then there is a constant* c *such that for any* $\alpha \in A^*$ *and any innermost reduction* $f_0(\tilde{\alpha}) \to_{IM}^* \rhd$*, the number of defined symbols in any term of the reduction is at most* c*.*

Proof. By Proposition 31, any term in the reduction $f_0(\tilde{\alpha}) \to_{\text{IM}}^* \rhd$ contains an innermost redex at border position. Hence, the position of any rewrite step in a term t in the reduction will occur at the *rightmost* element of $\mathcal{F}$ in t. Thus, redex contraction in innermost reduction will always occur at the rightmost element of $\mathcal{F}$ in t. Let $f \in \mathcal{F}$ be such an element, and let $f(\mathsf{c}(x)) \to f_1(\cdots f_m(x))$ be the rule of a redex at that position (if there is no variable in the right-hand side of the rule, the supposition that R is cons-free entails that no future steps will be able to produce $\triangleright$, a contradiction). As R is tail recursive, we have $f > f_2, \ldots, f_m$, and $f \geq f_1$.

Let l be the maximum number of occurrences of symbols from $\mathcal{F}$ in any right-hand side among rules of R. Any totally ordered chain $f_1 > f_2 > \cdots > f_m$ in $\mathcal{F}$ has length at most $|\mathcal{F}|$, and thus, the maximal number of defined symbols in any term in $f_0(\tilde{\alpha}) \rightarrow^*_{\mathrm{IM}} \triangleright$ is at most $c \triangleq 1 + l \cdot |\mathcal{F}|$. ◀

▶ **Example 39.** The assumption that $f_0(\tilde{\alpha}) \rightarrow^*_{\mathrm{IM}} \triangleright$ in Proposition 38 cannot be omitted (that is, the presence of $\triangleright$ as the final term is crucial). Consider the following constructor TRS:

$$R = \left\{ \begin{array}{rcll} f_0(x) & \rightarrow & f_0(g(x)) & \\ f_0(x) & \rightarrow & f_0(h(x)) & \\ f_0(x) & \rightarrow & g(x) & \\ g(\tilde{a}(x)) & \rightarrow & x & \text{for all } a \in A \end{array} \right\}$$

Observe that R is tail recursive and accepts A^+ (because $f_0(\tilde{\alpha}) \rightarrow^* f_0(g^{|\alpha|-1}(\tilde{\alpha})) \rightarrow g^{|\alpha|}(\tilde{\alpha}) \rightarrow^* \triangleright$. But the number of elements of $\mathcal{F}$ in terms occurring in reductions starting from $f_0(\tilde{\alpha})$ is unbounded, as witnessed by $f_0(\tilde{\alpha}) \rightarrow f_0(g(\tilde{\alpha})) \rightarrow \cdots$ and $f_0(\tilde{\alpha}) \rightarrow f_0(h(\tilde{\alpha})) \rightarrow f_0(h(h(\tilde{\alpha}))) \rightarrow \cdots$; in particular, the latter reduction shows that there are infinite reductions with an innermost redex at the *root* of every term, and where the number of elements of $\mathcal{F}$ in the terms has no upper bound.

We now define *one-call* systems:

▶ **Definition 40.** *A monadic constructor TRS is said to be* one-call *if, for every rule $l \rightarrow r$, the right-hand side r contains at most one element of $\mathcal{F}$.*

The following lemma shows that instead of tail recursion, we could instead have considered one-call systems:

▶ **Lemma 41.** *Let R be a monadic, cons-free, tail-recursive constructor TRS accepting language $L \subseteq A^+$. Then, there is a one-call, normal, monadic, cons-free constructor TRS that accepts L.*

Proof. By Lemma 29, we may assume wlog. that R is normal. By Lemma 33, for every $\alpha \in A^+$, if $f_0(\tilde{\alpha}) \rightarrow^* \triangleright$, then $f_0(\tilde{\alpha}) \rightarrow^*_{\mathrm{IM}} \triangleright$. By Proposition 38, there is a constant c such that for every $\alpha \in A^+$, for every reduction of the form $f_0(\tilde{\alpha}) \rightarrow^*_{\mathrm{IM}} \triangleright$, the number of elements of $\mathcal{F}$ in any term of the reduction is at most c.

We now construct a one-call (and normal, monadic, cons-free) constructor TRS R' that accepts L. R' will have a new set of defined symbols $\mathcal{F}'$ and use the same set of constructors $\mathcal{C}$ as R. For every integer k with $0 < k \leq c$ and every $(f_1, \ldots, f_k) \in \mathcal{F}^k$, create a defined symbol $g_{f_1 \cdots f_k} \in \mathcal{F}'$. As R is normal and cons-free, every rule of R is on one of the forms $f(\mathsf{c}(x)) \rightarrow r$ or $f(x) \rightarrow r$. For each symbol $g_{f_1 \cdots f_k} \in \mathcal{F}'$, and each rule $l \rightarrow r$ of R such that the root symbol of l is f_k, create a rule of R' as follows:

- $g_{f_1 \cdots f_k}(\mathsf{c}(s)) \rightarrow g_{f_1 \cdots f_{k-1} h_1 \cdots h_m}(s)$ if $l \rightarrow r = f_k(\mathsf{c}(x)) \rightarrow h_1(\cdots h_m(s))$ (where $s = x$ or $s \in \mathcal{F}$).
- $g_{f_1 \cdots f_k}(x) \rightarrow g_{f_1 \cdots f_{k-1} h_1 \cdots h_m}(s)$ if $l \rightarrow r = f_k(x) \rightarrow h_1(\cdots h_m(s))$ (where $s = x$ or $s \in \mathcal{F}$).

Define S to be the resulting TRS. By construction, S is one-call, monadic, and cons-free.

$$\mathrm{NFA}_R = (Q, A, \delta, \{q_{f_0}\}, \{q_h\}) \quad \text{where} \quad Q = \{q_f : f \in \mathcal{F}\} \cup \{q_h\}$$

Transition rules in δ:

rule(s) of R	transition rule in δ
$f(\mathtt{c}(x)) \to g(x)$	$\delta(q_f, c) \to q_g$
$f(x) \to g(x)$	$\delta(q_f, \epsilon) \to q_g$
$f(\mathtt{c}(x)) \to x$	$\delta(q_f, c) \to q_h$
$f(x) \to x,$	$\delta(q_f, \epsilon) \to q_h$

Figure 7 The NFA–NFA_R–defined from a normal, monadic cons-free, one-call constructor TRS R.

We claim that, for each $\alpha \in A^+$, we have $f_0(\tilde{\alpha}) \to_R^* \rhd$ iff $f_0(\tilde{\alpha}) \to_S^* \rhd$.

If $f_0(\tilde{\alpha}) \to_{\mathrm{IM}}^* \rhd$, write the reduction as $t_0 = f_0(\tilde{\alpha}) \to_{\mathrm{IM}} t_1 \to_{\mathrm{IM}} \to \cdots \to \rhd = t_n$. Observe that each term $t_i = f_1(\cdots f_k(\mathtt{c}))$ (where $\mathtt{c}$ is a constructor term) in the reduction can be mimicked in S by a term of the form $g_{f_1 \cdots f_k}(\mathtt{c})$.

By Proposition 31, every term of $f_0(\tilde{\alpha}) \to_{R,\mathrm{IM}}^* \rhd$, except the last, contains at least one innermost redex at border position, and hence, the step $t_i \to t_{i+1}$ must be $f_1(\cdots f_k(\mathtt{c})) \to f_1(\cdots f_{k-1}(h_1(\cdots h_m(\cdots))))$ using some rule $f_k(\mathtt{c}(x)) \to h_1(\cdots h_m(s))$ or $f_k(x) \to h_1(\cdots h_m(s))$ (for some $m \geq 0$). Hence, the step can clearly be mimicked by application of a rule in S, and we have $f_0(\tilde{\alpha}) \to_S^* \rhd$.

Conversely, if $f_0(\tilde{\alpha}) \to_S^* \rhd$, by construction of S, every term in the reduction is of the form $g_{f_1 \cdots f_k}(\mathtt{c})$ for some constructor term $\mathtt{c}$. For each such term, there is a step $g_{f_1 \cdots f_k}(\mathtt{c}) \to g_{f_1 \cdots f_{k-1} h_1 \cdots h_m}(s'\{x \mapsto \mathtt{c}\})$ iff there is a rule $f_k(s) \to h_1(\cdots h_m(s'))$ in R, and hence $f_1(\cdots f_k(\mathtt{c})) \to_R f_1(\cdots f_{k-1}(h_1(\cdots h_m(s'\{x \mapsto \mathtt{c}\}))))$.

Thus, every step of $f_0(\tilde{\alpha}) \to_S^* \rhd$ can be mimicked by an innermost step in R, whence $f_0(\tilde{\alpha}) \to_R^* \rhd$, as desired. Hence, S accepts L, and by construction, S is normal, monadic, and one-call. ◀

▶ **Lemma 42.** *Let R be a normal, monadic, cons-free, one-call constructor TRS deciding language $L \subseteq A^+$. Then, the NFA NFA_R (see Fig. 7) accepts L.*

Proof. Recall from basic automata theory that we may wlog. assume that an NFA only accepts if it is in an accepting state when all of its input has been consumed. Denote by $L(\mathrm{NFA}_R)$ the language accepted by NFA_R. By construction of NFA_R, any run of NFA_R clearly mimicks reductions of R: every rewrite step is mimicked by exactly one transition in NFA_R, and conversely, any transition in NFA_R can be mimicked by a rewrite step in R. If $f_0(\tilde{\alpha}) \to^* \rhd$, there is in particular a run of NFA_R ending in q_h with the entire input α having been consumed in the run, and hence $L \subseteq L(\mathrm{NFA}_R)$. Conversely, if $\alpha \in L(\mathrm{NFA}_R$, there is a run of NFA_R on input α that (i) consumes all the input, and (ii) ends in q_h, and hence there is a rewrite sequence starting from $f_0(\tilde{\alpha})$ that ends with one of the two rewrite steps $f(\mathtt{c}(\rhd)) \to \rhd$ or $f(\rhd) \to \rhd$, whence $L(\mathrm{NFA}_R) \subseteq L$. ◀

By the equivalence of DFAs and NFAs, it suffices to simulate DFAs by rewriting systems. In Figure 8 we show how to obtain such a system.

▶ **Lemma 43.** *If $M = (Q, A, \delta, Q_{\mathrm{accept}})$ is a DFA accepting language $L \subseteq A^+$, then R_M^{DFA} (see Fig. 8) accepts L.*

$M = (Q, A, \delta, q_0, Q_{\text{accept}})$ $\quad \mathcal{F} = \{f_q : q \in Q\}$ $\quad \mathcal{C} = \widetilde{A} \cup \{\triangleright\}$

Rules:

transition in δ	rewrite rule
$\delta(q, a) \to q'$	$f_q(\tilde{a}(x)) \to f_{q'}(x)$

Accepting run:

Rule (for every $q \in Q_{\text{accept}}$)
$f_q(x) \to x$

(recall that DFAs do not have ϵ-transitions)

Figure 8 Monadic cons-free, tail recursive constructor TRS R_M^{DFA} induced by a DFA M.

Proof. As the DFA is deterministic, there are no ϵ-transitions, and for every $(q, a) \in Q \times A$, there is at most one transition $\delta(q, a) \to q'$. Thus, the constructor TRS R_M^{DFA} is monadic, cons-free and one-call. Furthermore, if q_0 is the start state, set $f_0 = f_{q_0}$. We claim that for any $\alpha \in A^+$, we have $f_0(\widetilde{\alpha}) \to^* \triangleright$ iff there is an accepting run of the automaton on input α starting in q_0. To see this, note that there is a transition on string $b_1 b_2 \cdots b_k$ from state q to state $q' \notin Q_{\text{accept}}$ iff there is a rule $\delta(q, b_1) \to q'$ iff $f_q(\widetilde{b}_1(\widetilde{b}_2 \cdots \widetilde{b}_k(\triangleright))) \to f_{q'}(\widetilde{b}_2 \cdots \widetilde{b}_k(\triangleright)))$. Thus, M reaches an accepting state after emptying the input iff $f_0(\widetilde{\alpha}) \to^* f_q(\triangleright)$ where $q \in Q_{\text{accept}}$; and $f_q(\triangleright) \to \triangleright$ iff $q \in Q_{\text{accept}}$. Hence, the DFA accepts string α iff the above system accepts string α, and the result follows. ◀

We thus have the final result of the paper:

▶ **Theorem 44.** *The following are equivalent for a language $L \subseteq A^+$: (i) L is regular, (ii) L is accepted by a one-call, monadic, cons-free constructor TRS, (iii) L is accepted by a tail recursive, monadic, cons-free constructor TRS.*

Proof. If L is regular, it is accepted by a DFA, hence by Lemma 43 accepted by a monadic, cons-free, one-call constructor TRS. Conversely, if L is accepted by a monadic cons-free, one-call constructor TRS, Lemma 29 shows that we may wlog. assume that R is normal and one-call. Lemma 42 then shows that there is an NFA accepting L, whence L is regular. Finally, observe that a one-call TRS is always tail-recursive (by relating all defined symbols in the weak component of the ordering), and that Lemma 41 shows that any language accepted by a tail-recursive monadic, cons-free constructor TRS is also accepted by a one-call monadic, cons-free constructor TRS. ◀

7 Conclusion and future work

While we have characterized the original 4 language classes in the Chomsky hierarchy, it is clear that similar characterizations *should* exist for other classes, e.g., the visibly pushdown languages [1], or for deterministic context-free languages (where it is natural to conjecture that *non-overlapping* (strongly) cons-free constructor TRSs suffice). However, the proofs of the correspondences asserted in this paper followed from intuition about the (set of) *stacks* maintained by the restricted computational models traditionally used to characterize the classes; it is unclear whether this intuition can be used for more esoteric classes of languages.

On a different note, while the restriction to monadic systems plays well with the Chomsky hierarchy, it seems to be less amenable to characterizations of the usual complexity classes of interest in implicit complexity theory, e.g. PTIME, and it would be interesting to find *natural* constraints on monadic systems that allowed characterization of these classes in a liberal rewriting setting (i.e., no typing beyond what is strictly necessary, and with no restrictions on the evaluation order).

Finally, it should be investigated whether *strong* cons-freeness can be relaxed to more lenient versions of cons-freeness, but for the reasons noted in the paper, this may not give as short and clean a characterization as for strongly cons-free systems.

References

1. R. Alur and P. Madhusudan. Visibly pushdown languages. In László Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 202–211. ACM, 2004.
2. M. Avanzini, N. Eguchi, and G. Moser. A path order for rewrite systems that compute exponential time functions. In Manfred Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011*, volume 10 of *LIPIcs*, pages 123–138. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011.
3. M. Avanzini, G. Moser, and A. Schnabl. Automated implicit computational complexity analysis. In *Proceedings of the 4th International Joint Conference on Automated Reasoning (IJCAR '08)*, volume 5195 of *Lecture Notes in Computer Science*, pages 132–138. Springer-Verlag, 2008.
4. S. Bellantoni and S.A. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
5. Guillaume Bonfante. Some programming languages for logspace and ptime. In Michael Johnson and Varmo Vene, editors, *Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006, Kuressaare, Estonia, July 5-8, 2006, Proceedings*, volume 4019 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2006.
6. A.-C. Caron. Linear bounded automata and rewrite systems: Influence of initial configurations on decision properties. In Samson Abramsky and T. S. E. Maibaum, editors, *TAPSOFT, Vol.1*, volume 493 of *Lecture Notes in Computer Science*, pages 74–89. Springer, 1991. `doi:10.1007/3-540-53982-4_5`.
7. N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.
8. L. Czajka. Term rewriting characterisation of LOGSPACE for finite and infinite data. In Hélène Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, volume 108 of *LIPIcs*, pages 13:1–13:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.
9. D. de Carvalho and J. G. Simonsen. An implicit characterization of the polynomial-time decidable sets by cons-free rewriting. In G. Dowek, editor, *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014*, volume 8560 of *Lecture Notes in Computer Science*, pages 179–193. Springer, 2014.
10. E. P. Friedman. Equivalence problems for deterministic context-free languages and monadic recursion schemes. *Journal of Computer and Systems Sciences*, 14(3):344–359, 1977.
11. E. P. Friedman. Simple context-free languages and free monadic recursion schemes. *Mathematical Systems Theory*, 11(1):9–28, 1977.
12. E. P. Friedman and Sheila A. Greibach. Monadic recursion schemes: The effect of constants. *Journal of Computer and Systems Sciences*, 18(3):254–266, 1979.
13. J. Goldstine, J. K. Price, and D. Wotschke. On reducing the number of states in a pda. *Mathematical Systems Theory*, 15(4):315–321, 1982.
14. S. A. Greibach. *Theory of Program Structures: Schemes, Semantics, Verification*, volume 36 of *Lecture Notes in Computer Science*. Springer-Verlag, 1975.
15. J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson Education International Inc., 2 edition, 2003.
16. G. Huet and D.S. Lankford. On the uniform halting problem for term rewriting systems. Rapport Laboria 283, IRIA, 1978.
17. N. D. Jones. *Computability and Complexity from a Programming Perspective*. The MIT Press, 1997.

18 N.D. Jones. The expressive power of higher-order types, or: Life without cons. *Journal of Functional Programming*, 11(1):55–94, 2001.

19 C. Kop and J. G. Simonsen. Complexity hierarchies and higher-order cons-free term rewriting. *Log. Methods Comput. Sci.*, 13(3), 2017.

20 S. Kuroda. Classes of languages and linear-bounded automata. *Information and Control*, 7(2):207–223, 1964.

21 H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981.

22 J.-Y. Marion. Analysing the implicit complexity of programs. *Information and Computation*, 183(1):2–18, 2003.

23 M. L. Minsky. Recursive unsolvability of Post's problem of "tag" and other topics in theory of Turing machines. *The Annals of Mathematics*, 74(3):437–455, 1961.

24 C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

25 E. Rich. *Automata, Computability and Complexity: Theory and Applications*. Pearson Prentice Hall, 2008.

26 M. Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, 2nd edition, 2006.

27 Terese, editor. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

28 R. Thiemann, H. Zantema, J. Giesl, and P. Schneider-Kamp. Adding constants to string rewriting. *Appl. Algebra Eng. Commun. Comput.*, 19(1):27–38, 2008. `doi:10.1007/s00200-008-0060-6`.

Church's Semigroup Is Sq-Universal

Rick Statman ✉
Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

We prove Church's lambda calculus semigroup is sq-universal.

2012 ACM Subject Classification Theory of computation → Lambda calculus

Keywords and phrases lambda calculus, Church's semigroup, sq-universal

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.6

1 Introduction

In 1937 ([2]) Church formulated lambda calculus as a semigroup. His ideas were pursued by Curry and Feys ([3]), and later by Bohm (Barendregt [1, 532]) and Dezani ([4]). If lambda terms in some way represent functions, then such a presentation based on composition is a quite natural complement to the presentation based on application. Of course, it is widely held that lambda calculus, therefore this semigroup, is an important part of the foundation of functional programming.

In 1968 Peter Neumann [6] introduced the notion of an sq-universal group. Many results in classical group theory can be interperted as saying that a particular group (or class of groups) is sq-universal. The notion of sq-universal makes perfectly good sense for semigroups as well as groups. A countable semigroup O is sq-universal if every countable semigroup is a subsemigroup of a homomorphic image (quotient) of O ("sq" stands for "sub … of quotient …").

We shall show that Church's semigroup is sq-universal. We shall also characterize lambda theories as special kinds of quotients of the semigroup (there are quotients which do not correspond to lambda theories) at least when $I = 1$ (eta).

2 Church's semigroup

Some notation will be useful. We adopt for the most part the notation and terminology of [1].

$$\begin{aligned}
I &:= \lambda x.\, x \\
1 &:= \lambda xy.\, xy \\
B &:= \lambda xyz.\, x(yz) \\
K &:= \lambda xy.\, x \\
C &:= \lambda xyz.\, xzy. \\
\sim &:= \text{ beta conversion} \\
\to &:= \text{ beta reduction} \\
\twoheadrightarrow &:= \text{ beta reduction multistep.}
\end{aligned}$$

Both Church and Curry observed that the combinators form a semigroup under multiplication B and beta conversion. The same is true for addition $\lambda xyuv.\, xu(yuv)$ and beta conversion. Since these satisfy the right distributive law

$$(\lambda xyz.\, x(yz))((\lambda xyuv.\, xu(yuv))ab)c \sim (\lambda xyuv.\, xu(yuv))((\lambda xyz.\, x(yz))ac)(((\lambda xyz.\, x(yz))bc)$$

they form a near semiring.

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi; Article No. 6; pp. 6:1–6:6
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Many years ago I noticed a generalization of this near semiring structure to a hierarchy of semigroups. Define

$$A_n := \lambda xyu_1 \cdots u_n v.\, xu_1 \cdots u_n(yu_1 \cdots u_n v)$$

so $A_0 := B$ and A_1 is Church's addition. Then combinators form a semigroup with multiplication A_n with beta conversion. Again the right distributive law holds. More precisely we have

$$\begin{array}{lll} \text{(associativity)} & A_m(A_n xy) \sim A_0(A_n x)(A_n y) & \text{if } m = n, \\ \text{(distributivity)} & A_m(A_n xy) \sim A_{n+1}(A_m x)(A_m y) & \text{if } m < n. \end{array}$$

and in addition,

(i) $A_m x \sim A_{m+1}(Kx)I$
(ii) $K(A_m xy) \sim A_{m+1}(Kx)(Ky)$.

Let O_n be the semigroup of all combinators with multiplication A_n. Let $J = \lambda x.\, xI$ (J is usually written $C{*}{*}$). Now we adopt the infix notation $*$ for the prefixing of B.

(iii) $J * K \sim I$
(iv) $J(A_{m+1}xy) \sim A_m(Jx)(Jy)$.

3 Homomorphisms

A homomorphism h of O_n induces a congruence relation H defined by $M \mathrel{\mathrm{H}} N$ iff $h(M) = h(N)$. Here we identify h with the map that takes M to its congruence class $\{N \mid M \mathrel{\mathrm{H}} N\}$, so h is a set valued map.

▶ **Example 1.** $h(M) := BM$ defines a homomorphism of O_0.

▶ **Definition 2.** *h is said to be "entire" if*
(a) $h(KM) = K(h(M))$
(b) $h(JM)$ *contains* $J(h(M))$
(c) $h(1M) = h(M)$

▶ **Example 3.** $h(M) :=$ the beta-eta congruence class of M is entire. For, if KM beta-eta converts to N there exists P s.t. N beta converts to KP. This follows from Church-Rosser and eta postponement.

Now if h is an entire homomorphism for O_n then h is a homomorphism for every O_m with $m < n$, for we have

$$\begin{array}{l} K(h(A_{n-1}xy)) \sim \\ h(K(A_{n-1}xy)) \sim \\ h(A_n(Kx)(Ky)) \sim \\ A_n(h(Kx))(h(Ky)) \sim \\ A_n(K(h(x)))(K(h(y))) \sim \\ K(A_{n-1}(h(x))(h(y))) \end{array}$$

so by (iii) $h(A_{n-1}xy) \sim A_{n-1}(h(x))(h(y))$.

Lambda theories are defined as in [1] 4.1.1. Each lambda theory T over beta conversion induces a homomorphism for each O_n where H is defined by $M \mathrm{~H~} N$ iff $T \vdash M = N$. Each lambda theory T over beta- eta conversion induces an entire homomorphism for each O_n where H is defined by $M \mathrm{~H~} N$ iff $T \vdash M = N$. Now there are O_0 homomorphisms which are not induced by theories. For example, the Rees factor monoid induced by the ideal $\{KM \mid \text{all } M\}$. However we shall show that this is essentially the only example.

▶ **Theorem 4.** *Let h be an entire homomorphism for O_1. Then $T = \{M = N \mid M \mathrm{~H~} N\}$ is closed under logical consequence over beta conversion.*

Proof. We suppose that $T \vdash M = N$ over beta conversion. For what follows we will use a theorem of Jacopini [5] in the form exposited and marginally improved in [8].

By Jacopini's theorem, there exist $M_i = N_i$ in T for $i = 1, \dots, n$ and closed terms $P_1, \dots, P_n$ such that

$$\begin{aligned} M &\sim P_1 M_1 N_1 \\ P_1 N_1 M_1 &\sim P_2 M_2 N_2 \\ P_2 N_2 M_2 &\sim P_3 M_3 N_3 \\ &\vdots \\ P_n N_n M_n &\sim N. \end{aligned}$$

Thus by Church's theorem ([1, 531]), which uses eta in one spot,

$$\begin{aligned} & M \mathrm{~H~} CIN_1 * CIM_1 * CIP_1 * B * B * CI \\ & CIM_1 * CIN_1 * CIP_1 * B * B * CI \mathrm{~H~} CIN_2 * CIM_2 * CIP_2 * B * B * CI \\ & CIM_2 * CIN_2 * CIP_2 * B * B * CI \mathrm{~H~} CIN_3 * CIM_3 * CIP_3 * B * B * CI \\ & \vdots \\ & CIM_n * CIN_n * CIP_n * B * B * CI \mathrm{~H~} N. \end{aligned}$$

Now let us write $\#$ for A_1 infixed. We have

$$\begin{aligned} & KM \mathrm{~H~} K(CIN_1) \# K(CIM_1) \# K(CIP_1) \# KB \# KB \# K(CI) \\ & K(CIM_1) \# K(CIN_1) \# K(CIP_1) \# KB \# KB \# K(CI) \mathrm{~H} \\ & \qquad K(CIN_2) \# K(CIM_2) \# K(CIP_2) \# KB \# KB \# K(CI) \\ & K(CIM_2) \# K(CIN_2) \# K(CIP_2) \# KB \# KB \# K(CI) \mathrm{~H} \\ & \qquad K(CIN_3) \# K(CIM_3) \# K(CIP_3) \# KB \# KB \# K(CI) \\ & \vdots \\ & K(CIM_n) \# K(CIN_n) \# K(CIP_n) \# KB \# KB \# K(CI) \mathrm{~H~} KN \end{aligned}$$

by (ii). Now $K(CIx) \sim CI * Kx$ so since h is entire

$$h(K(CIM_i)) = h(K(CIN_i)) \quad \text{for } i = 1, \dots, n.$$

Thus, since h is a $\#$ homomorphism, $h(KM) = h(KN)$. But h is entire so $h(M) = h(N)$. ◀

▶ **Corollary 5.** *Let h be an entire homomorphism for O_1. Then $T = \{M = N \mid M \mathrm{~H~} N\}$ is closed under logical consequence over beta-eta conversion.*

▶ **Corollary 6.** *If h is an entire homomorphism for O_1 then it is an entire homomorphism for all O_n.*

4 SQ universality

▶ **Definition 7.** *A set \$ of order zero lambda-I terms is said to be independent if for every member M of \$ no beta reduct of M contains a beta reduct of any member of \$ as a proper subterm.*

▶ **Example 8.** The set of terms $(\lambda x.\, xx)(\lambda x.\, xx)N$, where N is a non-zero Church numeral is independent.

Curiously, independent sets must exist for recursion theoretic reasons.

▶ **Lemma 9.** *There must be an infinite independent set.*

Proof. We construct an increasing sequence of finite independent sets by induction.

Basis: $\{(\lambda x.\, xx)(\lambda x.\, xx)\}$ is independent.

Induction step; we suppose that \$ is a finite independent set. Now the following sets of lambda-I terms are RE and closed under beta reduction

(i) the set of combinators with positive order

(ii) the set of combinators M s.t. there is a beta reduct of a member of $\$ \cup \{M\}$ which is a proper subterm of a beta reduct of M.

In addition, both of these sets have non-empty complements. Thus by Visser's theorem (as modified in [7] and adapted to lambda-I) the intersection of the complements of these two sets is infinite (modulo beta-conversion). Thus one element can be added to \$. ◀

▶ **Definition 10.** *The B polynomials over \$ are defined as follows. Any variable or member of \$ is a B polynomial. If F and G are B polynomials then so is $F * G$.*

▶ **Lemma 11.** *Let \$ be an independent set. Let $P, P_1, \ldots, P_k$ be products of the members of \$. Then if $P \sim MP_1 \cdots P_k$ there exists a B polynomial $F(x_1, \ldots, x_k)$ over \$ s.t. $Mx_1 \cdots x_k \sim F(x_1, \ldots, x_k)$.*

Proof. Wlog we can assume that $P = \lambda x.\, J_1(\cdots(J_l x)\cdots)$ for the J_i members of \$ where if $l = 1$ then $P = J_1$. When $l = 1$ consider a standard reduction of $MP_1 \cdots P_k$ to P. Now if one of the P_j comes to the head of the head reduction part of the standard reduction we have

$$P_j \twoheadrightarrow J_1$$

and $Mx_1 \cdots x_k \twoheadrightarrow x_j$. Otherwise since the members of \$ are independent $Mx_1 \cdots x_k \twoheadrightarrow J_1$. Let $l > 1$, and let

$$MP_1 \cdots P_k \twoheadrightarrow \lambda x.\, J_1(\cdots(J_l x)\cdots)$$

by a standard beta reduction. Now if one of the P_j comes to the head of the head reduction part let @ be the substitution $[P_1/x_1, \ldots, P_k/x_k]$. We have for some X, $P_j = \lambda x.\, J_1(\cdots(J_m x)\cdots)$ or J_1

$$\begin{aligned} \lambda x.\, P_j(@X) &\twoheadrightarrow P \\ (\lambda x_1 \cdots x_k x.\, X)P_1 \cdots P_k &\twoheadrightarrow \lambda x.\, J_{m+1}(\cdots(J_l x)\cdots). \end{aligned}$$

In this case the proposition follows by induction on l. If no P_j comes to the head then at the end of the head reduction we have a term

$$\lambda x.\ @((\lambda y.\, Y)Y_1 \cdots Y_m)$$

which reduces to P by internal reductions. Thus $m = 2$ and $@((\lambda y.\, Y)Y_1) \twoheadrightarrow J_1$ by internal reductions, and

$$(\lambda x_1 \cdots x_k x.\, Y_2)P_1 \cdots P_k \twoheadrightarrow \lambda x.\, J_2(\cdots(J_l x)\cdots).$$

Since the J_i are independent $(\lambda y.\, Y)Y_1 \twoheadrightarrow J_1$ and the case follows by induction. ◀

Now if T is any set of equations between products of members of the independent set \$ then the lambda theory generated by T is certainly consistent since all these terms are unsolvables. Now these equations can be thought of as the presentation of a semigroup on the alphabet \$. If $P = Q$ is an equation between products of members of \$ then we may have $T \vdash P = Q$ where T is a lambda calculus theory, or $T \vdash P = Q$ where T is thought of as the presentation of a semigroup. It will be convenient to use the terminology $T \vDash P = Q$ for the semigroup case. Clearly if $T \vDash P = Q$ then $T \vdash P = Q$.

▶ **Lemma 12.** *If $T \vdash P = Q$ then $T \vDash P = Q$.*

Proof. Suppose that $T \vdash P = Q$. By Jacopini's theorem ([5]) there exist $M_1, \ldots, M_m$, and $P_1 = Q_1, \ldots, P_m = Q_m$ in T s.t.

$$\begin{aligned} P &\sim M_1 P_1 Q_1 \\ M_1 Q_1 P_1 &\sim M_2 P_2 Q_2 \\ M_2 Q_2 P_2 &\sim M_3 P_3 Q_3 \\ &\vdots \\ M_m Q_m P_m &\sim Q, \end{aligned}$$

The proof is by induction on m. Wlog we can assume that $P = \lambda x.\, J_1(\cdots(J_l x)\cdots)$. By lemma 11 there exists a B polynomial $F(x_1, x_2)$ over \$ s.t.

$$M_1 x_1 x_2 \sim F(x_1, x_2)$$

so

$$\begin{aligned} P &\sim F(P_1, Q_1) \\ T &\vDash P = F(P_1, Q_1) \\ T &\vDash P = F(Q_1, P_1) \\ F(Q_1, P_1) &\sim M_2 P_2 Q_2 \end{aligned}$$

and we can apply the induction hypothesis to

$$\begin{aligned} F(Q_1, P_1) &\sim M_2 P_2 Q_2 \\ M_2 Q_2 P_2 &\sim M_3 P_3 Q_3 \\ &\vdots \\ M_m Q_m P_m &\sim Q. \end{aligned}$$

◀

▶ **Theorem 13.** *O_0 is sq-universal.*

Proof. Suppose that the countable semigroup S is given. We take a set of generators and a presentation of S on these generators. Using lemma 9, we construct an independent set, which we identify with these generators, and we construct a lambda theory T, which encodes the presentation of S. By lemma 12 $T \vdash P = Q$ if and only if $P = Q$ is true in S. But by section 2 there is a homomorphism h of O_0 s.t. $T \vdash P = Q$ if and only if $h(P) = h(Q)$. ◀

References

1 Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985.

2 Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

3 Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume 1. North Holland, 1958.

4 Mariangiola Dezani-Ciancaglini. Characterization of Normal Forms Possessing Inverse in the *lambda-beta-eta*-Calculus. *Theor. Comput. Sci.*, 2(3):323–337, 1976. doi:10.1016/0304-3975(76)90085-2.

5 Giuseppe Jacopini. A condition for identifying two elements of whatever model of combinatory logic. In Corrado Böhm, editor, *Lambda-Calculus and Computer Science Theory, Proceedings of the Symposium Held in Rome, Italy, March 25-27, 1975*, volume 37 of *Lecture Notes in Computer Science*, pages 213–219. Springer, 1975. doi:10.1007/BFb0029527.

6 P. Neumann. The SQ-universality of some finitely presented groups. *J. Austral. Math. Soc.*, 16:1–6, 1973.

7 Richard Statman. Morphisms and Partitions of V-Sets. In Georg Gottlob, Etienne Grandjean, and Katrin Seyr, editors, *Computer Science Logic, 12th International Workshop, CSL '98, Annual Conference of the EACSL, Brno, Czech Republic, August 24-28, 1998, Proceedings*, volume 1584 of *Lecture Notes in Computer Science*, pages 313–322. Springer, 1998. doi:10.1007/10703163_21.

8 Richard Statman. Consequences of Jacopini's Theorem: Consistent Equalities and Equations. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings*, volume 1581 of *Lecture Notes in Computer Science*, pages 355–364. Springer, 1999. doi:10.1007/3-540-48959-2_25.

Call-By-Value, Again!

Axel Kerinec
Laboratoire LIPN, CNRS UMR 7030, Université Sorbonne Paris-Nord, France

Giulio Manzonetto
Laboratoire LIPN, CNRS UMR 7030, Université Sorbonne Paris-Nord, France

Simona Ronchi Della Rocca
Computer Science Department, University of Torino, Italy

Abstract

The quest for a fully abstract model of the call-by-value λ-calculus remains crucial in programming language theory, and constitutes an ongoing line of research. While a model enjoying this property has not been found yet, this interesting problem acts as a powerful motivation for investigating classes of models, studying the associated theories and capturing operational properties semantically.

We study a relational model presented as a relevant intersection type system, where intersection is in general non-idempotent, except for an idempotent element that is injected in the system. This model is adequate, equates many λ-terms that are indeed equivalent in the maximal observational theory, and satisfies an Approximation Theorem w.r.t. a system of approximants representing finite pieces of call-by-value Böhm trees. We show that these tools can be used for characterizing the most significant properties of the calculus – namely valuability, potential valuability and solvability – both semantically, through the notion of approximants, and logically, by means of the type assignment system. We mainly focus on the characterizations of solvability, as they constitute an original result. Finally, we prove the decidability of the inhabitation problem for our type system by exhibiting a non-deterministic algorithm, which is proven sound, correct and terminating.

2012 ACM Subject Classification Theory of computation → Denotational semantics; Theory of computation → Linear logic

Keywords and phrases λ-calculus, call-by-value, intersection types, solvability, inhabitation

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.7

Funding *Giulio Manzonetto*: This work is partly supported by ANR Project ANR-19-CE48-0014.

Acknowledgements We would like to thank Giulio Guerrieri for interesting discussions, as well as the anonymous reviewers for helpful suggestions.

Introduction

Despite the fact that the call-by-value (CbV) λ-calculus has been introduced by Plotkin several decades ago [22], the problem of finding a denotational model satisfactorily reflecting its operational semantics is not completely solved, yet. While a plethora of adequate models has been constructed, e.g., in the Scott continuous and stable semantics [13, 23, 18], none enjoys completeness and it is therefore fully abstract. Similarly, the theory of program approximations for the CbV λ-calculus remained for a longtime rather involuted compared to the one developed in the call-by-name (CbN) setting (see [5, Ch. 14]). As an example, in [26] the authors show that the continuous model built in [13] does satisfy an Approximation Theorem, but the considered notion of approximant turns out to be too weak for capturing any interesting operational property. The main problem one encounters when approximating CbV reductions is that certain redexes remain stuck along reductions for silly reasons, thus preventing the creation of other redexes and leading to premature CbV normal forms (see [2]). A possible solution has been proposed in [10] by introducing permutation reductions that allow to unblock such redexes without altering fundamental operational properties of the

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi; Article No. 7; pp. 7:1–7:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

calculus, like the capability of a program of reducing to a value or the notion of solvability (as shown in [17]). This breakthrough has renewed the interest in the CbV λ-calculus within the scientific community and led to a wealth of original results [2, 17, 3, 20, 19].

Inspired by the relational semantics of Linear Logic [15] and exploiting Girard's "boring" translation of intuitionistic arrow in linear logic, sending $A \to B$ into $!(A \multimap B)$, Ehrhard introduced a class of relational models for CbV λ-calculus [14]. Like filter models correspond to intersection type systems under the celebrated Stone duality [1], also relational models can be nicely presented in a similar fashion [25], except for the fact that the intersection becomes a non-idempotent operator. Thus, the type $\alpha_1 \wedge \cdots \wedge \alpha_n$ can be seen as a multiset $[\alpha_1, \dots, \alpha_n]$. The advantage of counting the multiplicities is that it allows to expose quantitative properties of programs, e.g., by extracting a bound to their head reduction sequences [12]. The disadvantage of using relational models is that they are extremely poor in terms of representable theories. In the CbN setting, it is clear from [7] that all non-extensional relational graph models induce the same theory, and we have reasons to believe that the same holds for the class of CbV models from [14]. Therefore, in order to obtain different theories, one needs to substantially modify the construction of the model. Now, in coherent spaces, it is possible construct CbV models by performing a "lifting" that injects a new point $\star$ (coherent with all existing points) [18], leading to a solution of the domain equation $\mathcal{D} \cong [\mathcal{D} \to_s \mathcal{D}] \oplus \{\star\}$, where $[\cdot \to_s \cdot]$ denotes the domain of stable functions [6]. Mimicking this construction in the relational semantics, a new class of relational models for the CbV λ-calculus was introduced in [20]. The main difference is that, in the associated type assignment systems, the intersection is still non-idempotent except for an idempotent element $[]$, which is available at will and can be used to type any value in the empty environment. The authors show that all models in this class satisfy adequacy, a property they share with Ehrhard's relational models, but induce different theories. More precisely, they equate many λ-terms that are indeed equal in the maximal observational equivalence, whence the induced theory is closer to full abstraction. A notable example is given by the λ-terms $(\lambda x.xx)M$ and MM that are observationally indistinguishable – even when M is not a value – but have distinct interpretations in Ehrhard's models [16]. Moreover it has been proved that all models in this class enjoy an Approximation Theorem.

In this paper we study a particular relational model $\mathcal{M}$ living in the class of [20], corresponding to a relevant intersection type system having countably many atoms and no additional equivalences among types (in particular, atoms are not equivalent to arrow types). We show that the model $\mathcal{M}$ satisfies an Approximation Theorem with respect to a refined notion of syntactic approximants, that take permutation rules into account and were successfully applied in [19] to introduce a CbV notion of Böhm trees. By exploiting the resource consciousness of the model, we are able to provide an easy inductive proof of this result (Theorem 23) and avoid the impredicative techniques based on reducibility candidates that are needed in the continuous and stable semantics (see, e.g., [4, Ch. 17] or [26, Thm. 11.1.19]). As a consequence of the Approximation Theorem we derive that the model $\mathcal{M}$ equates all λ-terms having the same CbV Böhm tree. The fact that $\mathcal{M}$ is sensitive to the amount of resources needed by a λ-term during its execution still breaks the full abstraction property (a counter-example is given in [20]).

Despite the lack of full abstraction, we show that the model $\mathcal{M}$ and the associated system of approximants allow to characterize nicely operational properties like *valuability*, *potential valuability*, and *solvability*. A λ-term is (potentially) valuable if it reduces to a value (under suitable substitutions), and solvable if it is capable of generating a completely defined result, like the identity, when plugged in a suitable context. The notion of solvability, inherited from the CbN λ-calculus, is particularly interesting since it identifies the "meaningful" programs. In CbN, solvability has been characterized operationally (via head reduction), logically (through typability) and semantically (by building models assigning non-trivial interpretations to solvable terms exclusively). The model $\mathcal{M}$ provides a logical and semantic characterization of CbV solvability. On the logical side, we show that a λ-term is solvable if and only if it is typable in $\mathcal{M}$ with types that are "proper", in the sense of Definition 31. On the semantic side, we prove that all solvables admit approximants having a particular shape. Both the logical and semantic characterizations are presented in Theorem 36 and, as a consequence, we obtain that $\mathcal{M}$ is not sensible, but semi-sensible. This means that the model is "meaningful" since it does not equate all unsolvables, but neither equates a solvable to an unsolvable.

Finally, since the model $\mathcal{M}$ is presented as an intersection type system, it feels natural to wonder whether the type inhabitation problem is decidable.

> The Inhabitation Problem (IHP): Given any type environment Γ and any type α, is there a λ-term M having type α in Γ?

Since Urzyczyn's work [27], it is known that IHP is undecidable for the CbN (idempotent) intersection type system presented in [11]. Van Bakel subsequently simplified the system using strict types [29], where intersection is only allowed on the left-hand side of an arrow, while maintaining the undecidability of inhabitation – even in its "relevant" version where type environments only contain the consumed premises (Urzyczyn's proof extends easily [28]). On the one hand, this shows that the decidability of inhabitation is not strictly connected with the relevance of the system, on the other hand IHP has been proven decidable for several non-idempotent intersection type systems [9]. In Section 4, we describe a non-deterministic algorithm taking an environment Γ and a type α as inputs, and generating as output all minimal approximants having type α in Γ. First, we show that the algorithm is terminating (Theorem 46), a result only possible because there are finitely many approximants satisfying the above criteria. Then we demonstrate the soundness and completeness properties of the algorithm (Theorem 48), from which the decidability of IHP in this setting follows. Although our inhabitation algorithm is clearly inspired by [9], the adaptation is non-trivial for two reasons: the presence in the CbV setting of normal inhabitants having the shape $(\lambda x.M)N$ of a β-redex, and the presence of an idempotent element in the type assignment system.

Some related works

Despite the existence of several models of the CbV λ-calculus, their theories have rarely been explored. An exception is [26], where the theory of the model from [13] has been extensively studied. A semantic characterization of solvability is given, but not completely satisfactory because of the weak notion of approximation employed. The first logical characterization of CbV solvability is in [21], through a particular class of (idempotent) intersection types – it is, in some sense, similar to ours, but it is not based on a semantic model. Two known attempts at characterizing this notion from an operational point of view are [21, 10], both based on *ad hoc* reduction rules that are however unsound for CbV semantics. This suggests that CbV languages still lack a satisfactory rewriting theory.

1 Preliminaries

For the syntax of λ-calculus we mainly follow Barendregt's first book [5], for its call-by-value version [26], and for its extension with permutation rules [10].

1.1 The call-by-value λ-calculus

We consider fixed a countable set $\mathbb{V}$ of variables. The set Λ of λ-*terms* and the set Val of *values* are defined inductively via the following grammar (where $x \in \mathbb{V}$):

$$(\Lambda) \quad M, N ::= MN \mid V \qquad\qquad (\mathrm{Val}) \quad V, U ::= x \mid \lambda x.M$$

Application is represented as juxtaposition. As usual, we assume that it associates to the left and has higher-precedence than abstraction. Given $M \in \Lambda$, we shorten $\lambda x_1.(\cdots(\lambda x_k.M)\cdots)$ as $\lambda x_1 \ldots x_k.M$ or even as $\lambda \vec{x}.M$. For example, $\lambda xyz.xyz$ stands for $\lambda x.(\lambda y.(\lambda z.(xy)z))$. Given $N_1, \ldots, N_n \in \Lambda$, we write $M\vec{N}$ for $MN_1 \cdots N_n$ and $MN_1^{\sim k}$ for $MN_1 \cdots N_1$ (k-times).

The set $\mathrm{FV}(M)$ of *free variables of* M and α-*conversion* are defined as usual [5, §2.1]. We say that a λ-term M is *closed*, or *a combinator*, whenever $\mathrm{FV}(M) = \emptyset$. We denote by Λ^o the set of all combinators. From now on, λ-terms are considered up to α-conversion.

Concerning specific combinators, we fix the following notations (for $n \in \mathbb{N}$):

$$\begin{array}{lllllllll} \mathbf{K} & = & \lambda xy.x & \mathbf{\Delta} & = & \lambda x.xx, & \mathbf{\Omega} & = & \mathbf{\Delta\Delta}, \qquad \mathbf{P}_n = \lambda x_0 \ldots x_n.x_n, \\ \mathbf{B} & = & \lambda fgx.f(gx), & \mathbf{K}^\star & = & \mathbf{ZK}, & \mathbf{Z} & = & \lambda f.(\lambda y.f(\lambda z.yyz))(\lambda y.f(\lambda z.yyz)), \end{array}$$

where $\mathbf{K}$ is the first projection, $\mathbf{\Delta}$ the self-application, $\mathbf{\Omega}$ the paradigmatic looping combinator, $\mathbf{P}_n$ erases n arguments, $\mathbf{B}$ is the composition, $\mathbf{K}^\star$ an ogre and $\mathbf{Z}$ a CbV recursion operator. Notice that $\mathbf{P}_0 = \lambda x_0.x_0$ is the identity, therefore we also use $\mathbf{I}$ as an alternative notation.

▶ **Definition 1.** *On Λ, we define the following notions of reduction (for $V \in \mathrm{Val}$):*

$$\begin{array}{llcll} (\beta_{\mathsf{v}}) & (\lambda x.M)V & \to & M[V/x], & \textit{where } [V/x] \textit{ denotes capture-free substitution,} \\ (\sigma_1) & (\lambda x.M)NP & \to & (\lambda x.MP)N, & \textit{with } x \notin \mathrm{FV}(P), \\ (\sigma_3) & V((\lambda x.M)N) & \to & (\lambda x.VM)N, & \textit{with } x \notin \mathrm{FV}(V). \end{array}$$

We also define $(\sigma) = (\sigma_1) \cup (\sigma_3)$ and $(\mathsf{v}) = (\beta_{\mathsf{v}}) \cup (\sigma)$. Each $\mathsf{R} \in \{\beta_{\mathsf{v}}, \sigma_1, \sigma_3, \sigma, \mathsf{v}\}$ induces a one-step *(resp.* multi-steps*)* reduction *relation $\to_{\mathsf{R}}$ ($\twoheadrightarrow_{\mathsf{R}}$), and a* conversion *relation $=_{\mathsf{R}}$. We say that a λ-term M* is in R-normal form *(R-nf, for short) if there is no $N \in \Lambda$ such that $M \to_{\mathsf{R}} N$. We say that M* has an R-nf *if $M \twoheadrightarrow_{\mathsf{R}} N$ for some λ-term N in R-nf.*

▶ **Fact 2.** *The set* Val *is closed under substitutions $\vartheta : \mathbb{V} \to \mathrm{Val}$ and v-reductions.*

Plotkin's original formulation of the CbV λ-calculus only considers the β_{v}-reduction [22]. The permutation rules (σ), introduced by Regnier in the CbN setting [24], have been extended by Carraro and Guerrieri to CbV in [10], where the following properties are shown.

▶ **Proposition 3.**

(i) *The reduction $\to_\sigma$ is strongly normalizing. More precisely, there exists a measure $\mathsf{s} : \Lambda \to \mathbb{N}$ such that $M \to_\sigma N$ entails $\mathsf{s}(N) < \mathsf{s}(M)$.*

(ii) *The reduction $\to_{\mathsf{v}}$ is confluent. In particular, the v-nf of $M \in \Lambda$ (if any) is unique.*

▶ **Example 4.**

(i) $\mathbf{\Omega} \to_{\beta_{\mathsf{v}}} \mathbf{\Omega}$, $\lambda x.\mathbf{\Omega} \to_{\beta_{\mathsf{v}}} \lambda x.\mathbf{\Omega}$ and $\mathbf{I}x \to_{\beta_{\mathsf{v}}} x$, while $\mathbf{I}(xy)$ is a v-nf.

(ii) $(\lambda y.\mathbf{\Delta})(x\mathbf{I})\mathbf{\Delta}$ is a β_{v}-nf, but $(\lambda y.\mathbf{\Delta})(x\mathbf{I})\mathbf{\Delta} \to_{\sigma_1} (\lambda y.\mathbf{\Omega})(x\mathbf{I}) \to_{\beta_{\mathsf{v}}} (\lambda y.\mathbf{\Omega})(x\mathbf{I}) \to_{\beta_{\mathsf{v}}} \cdots$

(iii) $\mathbf{I}(\mathbf{\Delta}(xx))$ is a β_{v}-nf, but contains a σ_3-redex, indeed $\mathbf{I}(\mathbf{\Delta}(xx)) \to_{\sigma_3} (\lambda z.\mathbf{I}(zz))(xx)$.

(iv) $\mathbf{Z}$ is called a recursion operator since $\mathbf{Z}V =_{\beta_{\mathsf{v}}} V(\lambda x.\mathbf{Z}Vx)$, for all $V \in \mathrm{Val}$ and x fresh.

(v) $\mathbf{K}^\star =_{\mathsf{v}} \mathbf{K}(\lambda y.\mathbf{K}^\star y) =_{\mathsf{v}} \lambda x_0x_1.\mathbf{K}^\star x_1 =_{\mathsf{v}} \lambda x_0x_1x_2.\mathbf{K}^\star x_2 =_{\mathsf{v}} \cdots =_{\mathsf{v}} \lambda x_0 \ldots x_n.\mathbf{K}^\star x_n$.

(vi) For all $\vec{V} \in \mathrm{Val}$, we have $\mathbf{K}^\star\vec{V} =_{\mathsf{v}} \mathbf{K}^\star$ and $\mathbf{P}_nV_1 \cdots V_m \twoheadrightarrow_{\mathsf{v}} \mathbf{P}_{n-m}$ provided $n \geq m$.

Lambda terms are classified into valuable, potentially valuable, solvable or unsolvable depending on their behavior and their capability of interaction with the environment.

▶ **Definition 5.** *A λ-term M is called:*

(i) valuable *if it reduces to a value, namely $M \twoheadrightarrow_{\mathsf{v}} V$ for some $V \in \mathrm{Val}$.*

(ii) potentially valuable *if there exists a substitution $\vartheta : \mathbb{V} \to \mathrm{Val}$ such that M^{ϑ} is valuable.*

(iii) solvable *if there exist sequences $\vec{x}, \vec{V} \in \mathrm{Val}$ such that $(\lambda\vec{x}.M)\vec{V} \twoheadrightarrow_{\mathsf{v}} \mathbf{I}$.*

(iv) unsolvable*, if it is not solvable.*

Notice that the notions of solvability and valuability are both stronger than potential valuability, but orthogonal with each other. We provide some discriminating examples.

▶ **Example 6.**

(i) $\mathbf{I}, \boldsymbol{\Delta}, \mathbf{P}_n, \boldsymbol{\Delta}(\mathbf{II}), \mathbf{P}_1(\lambda x.\boldsymbol{\Omega})$ are (potentially) valuable and solvable.

(ii) $\mathbf{P}_1 x(\lambda x.\boldsymbol{\Omega}), xy\mathbf{I}\boldsymbol{\Delta}$ and $\boldsymbol{\Delta}(xy)$ are not valuable, but potentially valuable and solvable.

(iii) $\lambda x.\boldsymbol{\Omega}, \mathbf{ZB}$ and $\mathbf{K}^\star$ are valuable, but unsolvable. The term $\mathbf{K}^\star$ is called an ogre because of its capability of eating any $\vec{V}$ while remaining valuable: $\mathbf{K}^\star\vec{V} \twoheadrightarrow_{\beta_{\mathsf{v}}} \lambda x.M \in \mathrm{Val}$.

(iv) $\boldsymbol{\Omega}, \boldsymbol{\Omega}(xy), (\lambda y.\boldsymbol{\Delta})(x\mathbf{I})\boldsymbol{\Delta}, \mathbf{I}\boldsymbol{\Omega}, \mathbf{ZI}$ are not potentially valuable nor solvable. The same holds for $\mathbf{Y}M$, where $\mathbf{Y}$ is a CbN fixed point operator and M is a λ-term.

▶ Remark 7. The original definitions of valuability, potential valuability and solvability are given in terms of β_{v}-reduction (see [22] and [26], respectively). In [10] and [17], it is shown that all these notions are preserved when considering the extended v-reduction. In particular, for all λ-terms M, we have that $M \twoheadrightarrow_{\beta_{\mathsf{v}}} \mathbf{I}$ holds exactly when $M \twoheadrightarrow_{\mathsf{v}} \mathbf{I}$ does.

▶ **Property 8.** *If $M = (\lambda x_1 \dots x_k.P)N_1 \cdots N_n \twoheadrightarrow_{\mathsf{v}} \mathbf{I}$ then each N_i is valuable, say $N_i \twoheadrightarrow_{\mathsf{v}} V_i$. Moreover, we must have $k \leq n+1$.*

Proof. By the above remark, $M \twoheadrightarrow_{\mathsf{v}} \mathbf{I}$ entails $M \twoheadrightarrow_{\beta_{\mathsf{v}}} \mathbf{I}$. Therefore, $k > n+1$ would imply $M \twoheadrightarrow_{\beta_{\mathsf{v}}} (\lambda\vec{x}.P)\vec{V} \twoheadrightarrow_{\beta_{\mathsf{v}}} \lambda x_{n+1} \dots x_k.P' \neq_{\beta_{\mathsf{v}}} \mathbf{I}$. ◀

For the model theory of CbV λ-calculus, we refer to [26]. Every model $\mathcal{S}$ comes equipped with an interpretation map $[\![-]\!]$ that allows to compute the *denotation* $[\![M]\!]$ of $M \in \Lambda$. We say that $\mathcal{S}$ *equates* $M, N \in \Lambda$ whenever $[\![M]\!] = [\![N]\!]$. The least requirement for a model $\mathcal{S}$ is that it equates all β_{v}-convertible λ-terms (*soundness*). A model $\mathcal{S}$ is called *consistent* if it does not equate all λ-terms; *inconsistent* if it is not consistent; *sensible* if it is consistent and equates all unsolvables; *semi-sensible* if it does not equate a solvable and an unsolvable.

2 A Call-by-Value Relational Model

We define a particular model $\mathcal{M}$ living in the class of relational models introduced in [20]. A model $\mathcal{S}$ in this class can be described as a type assignment system, where finite multisets of types appear at the left-hand side of an arrow. Such a model $\mathcal{S}$ is uniquely identified by a set $\mathbb{A}$ of atomic types and a congruence $\simeq$ on types, respecting the multiset cardinalities. The model $\mathcal{M}$ under consideration corresponds to the relational model having countably many atoms, and the trivial congruence relation on types (namely, $\simeq$ is the equality $=$).

2.1 The Type Assignment System $\mathcal{M}$

In order to define the type assignment system $\mathcal{M}$, we need to introduce some basic notions and notations concerning finite multisets. Given a set A, we represent a finite multiset over A as an unordered list $[\alpha_1, \dots, \alpha_n]$, possibly with repetitions, where $n \in \mathbb{N}$ and each $\alpha_i \in A$.

$$\frac{}{x:[\alpha]\vdash x:\alpha}\ (\mathrm{var}) \qquad \frac{\Gamma, x:\sigma\vdash M:\alpha}{\Gamma\vdash \lambda x.M:\sigma\to\alpha}\ (\mathrm{lam}) \qquad \frac{\Gamma_0\vdash M:\sigma\to\alpha \quad \Gamma_1\vdash N:\sigma}{\Gamma_0+\Gamma_1\vdash MN:\alpha}\ (\mathrm{app})$$

$$\frac{\Gamma_1\vdash M:\alpha_1 \quad \cdots \quad \Gamma_n\vdash M:\alpha_n \quad n>0}{\sum_{i=1}^n \Gamma_i\vdash M:[\alpha_1,\dots,\alpha_n]}\ (\mathrm{val}_{>0}) \qquad \frac{V\in\mathrm{Val}}{\vdash V:[]}\ (\mathrm{val}_0)$$

Figure 1 The inference rules of the type assignment system $\mathcal{M}$. In (lam) we assume $x\notin\mathrm{dom}(\Gamma)$.

The empty multiset will be denoted by $[]$. We write $\mathcal{M}_{\mathrm{f}}(A)$ for the set of all finite multisets over A. Given $\sigma,\tau\in\mathcal{M}_{\mathrm{f}}(A)$, we write $\sigma+\tau$ for their multiset union. The operator $+$ extends to the n-ary case $\sigma_1,\dots,\sigma_n\in\mathcal{M}_{\mathrm{f}}(A)$ in the obvious way, in symbols, $\sum_{i=1}^n\sigma_i\in\mathcal{M}_{\mathrm{f}}(A)$.

▶ **Definition 9.** *Let us fix a countable set $\mathbb{A}=\{\mathsf{a},\mathsf{b},\mathsf{c},\dots\}$ of constants called* atomic types*.*

(i) *The set $\mathbb{T}$ of* types *over $\mathbb{A}$ and the set $\mathbb{T}^!$ of* multiset types *are defined by (for $n\geq 0$):*

$$\begin{array}{lll} (\mathbb{T}) & \alpha,\beta & ::= \ \mathsf{a}\mid []\mid \sigma\to\alpha \\ (\mathbb{T}^!) & \sigma,\tau,\rho & ::= \ [\alpha_1,\dots,\alpha_n] \quad \text{with } \alpha_i\neq[], \text{ for all } i\ (1\leq i\leq n). \end{array}$$

The arrow is right associative, i.e., $\sigma_1\to\cdots\to\sigma_n\to\alpha=(\sigma_1\to(\cdots(\sigma_n\to\alpha)\cdots))$.

(ii) Type environments *are functions $\Gamma:\mathbb{V}\to\mathbb{T}^!$ having a finite domain, which is defined by $\mathrm{dom}(\Gamma)=\{x\mid\Gamma(x)\neq[]\}$. The multiset sum is extended to type environments Γ and Δ pointwisely, namely, by setting $(\Gamma+\Delta)(x)=\Gamma(x)+\Delta(x)$, for all $x\in\mathbb{V}$.*

(iii) *We denote by $x_1:\sigma_1,\dots,x_n:\sigma_n$ the type environment Γ defined by setting:*

$$\Gamma(x)=\begin{cases}\sigma_i, & \text{if } y=x_i \text{ for some } i\in\{1,\dots,n\},\\ [], & \text{otherwise.}\end{cases}$$

Intuitively, the multiset type $[\alpha_1,\dots,\alpha_n]\in\mathbb{T}^!=\mathcal{M}_{\mathrm{f}}(\mathbb{T}-\{[]\})$ represents an intersection type $\alpha_1\wedge\cdots\wedge\alpha_n$, where $\wedge$ enjoys associativity and commutativity, but not idempotency ($\alpha\wedge\alpha\neq\alpha$). The empty multiset $[]$ belongs both to $\mathbb{T}$ and $\mathbb{T}^!$, but with different meanings: $[]\in\mathbb{T}$ should be thought of as a special "idempotent" type atom which is available at will; morally, $[]\in\mathbb{T}^!$ is a multiset only containing an indeterminate amount of atoms $[]\in\mathbb{T}$.

▶ **Definition 10.**

(i) *A* typing judgement *has shape $\Gamma\vdash M:\xi$, where Γ is an environment, $M\in\Lambda$ and $\xi\in\mathbb{T}\cup\mathbb{T}^!$. The inference rules of the type system $\mathcal{M}$ are given in Figure 1.*

(ii) *We write $\Pi\ \triangleright\Gamma\vdash M:\xi$ to indicate that Π is a derivation of $\Gamma\vdash M:\xi$. Hereafter, when writing $\Gamma\vdash M:\xi$, we assume that $\Pi\ \triangleright\Gamma\vdash M:\xi$ holds for some derivation Π.*

The rules (var), (lam) and (app) are self-explanatory. In case $x\notin\mathrm{FV}(M)$, the rule (lam) assigns $\lambda x.M$ the type $[]\to\alpha$ in the environment Γ. The rule (val_0) can be used to type every value with $[]$ in the empty environment. The rule $(\mathrm{val}_{>0})$ allows to collect several types of M into a non-empty multiset type, by adding the corresponding environments together. The type system is relevant in the sense that $\Gamma\vdash M:\alpha$ entails $\mathrm{dom}(\Gamma)\subseteq\mathrm{FV}(M)$.

▶ **Example 11.** The following is a derivation Π in system $\mathcal{M}$ (setting $\Gamma=f:[[\mathsf{a},\mathsf{a}]\to\mathsf{a}]$):

$$\dfrac{\dfrac{}{\Gamma\vdash f:[\mathsf{a},\mathsf{a}]\to\mathsf{a}} \qquad \dfrac{\dfrac{\dfrac{x:[[]\to\mathsf{a}]\vdash x:[]\to\mathsf{a} \qquad \vdash y:[]}{x:[[]\to\mathsf{a}]\vdash xy:\mathsf{a}} \qquad \dfrac{x:[[\mathsf{b}]\to\mathsf{a}]\vdash x:[\mathsf{b}]\to\mathsf{a} \qquad y:[\mathsf{b}]\vdash y:\mathsf{b}}{x:[[\mathsf{b}]\to\mathsf{a}],y:[\mathsf{b}]\vdash xy:\mathsf{a}}}{x:[[]\to\mathsf{a},[\mathsf{b}]\to\mathsf{a}],y:[\mathsf{b}]\vdash xy:[\mathsf{a},\mathsf{a}]}}{}}{\Gamma+x:[[]\to\mathsf{a},[\mathsf{b}]\to\mathsf{a}],y:[\mathsf{b}]\vdash f(xy):\mathsf{a}}$$

Other derivable typing judgements are $\vdash\mathbf{I}:[\mathsf{a}]\to\mathsf{a}$, $\vdash\mathbf{I}x:[]$ and $x:[\mathsf{a}]\vdash(\lambda y.x)x:\mathsf{a}$.

Through the rule (val_0), it is possible to assign the type $[]$ to a value V without inspecting its shape and typing its subterms[1] – we say that such a V is not *fully typed*. Similarly, in a derivation of $\Gamma \vdash M : \alpha$, certain subterms of M might not be fully typed. E.g., in any derivation of $x : [\mathsf{a}] \vdash (\lambda y.x)x : \mathsf{a}$, the former occurrence of x must be fully typed, while the latter cannot be. To identify occurrences of a subterm and formalize this intuitive property, we introduce single-hole contexts. A *single-hole context* $C[]$ is a λ-term containing exactly one occurrence of a distinguished algebraic variable $[]$, traditionally called its *hole*. Given a single-hole context $C[]$ and $N \in \Lambda$, we write $C[N]$ for the λ-term obtained by substituting N for the occurrence of the hole $[]$ in $C[]$, possibly with capture of free variables. Every such context $C[]$ uniquely identifies one occurrence of a subterm N of M, as in $M = C[N]$.

▶ **Definition 12.** *Let $M \in \Lambda$, and $\Pi \triangleright \Gamma \vdash M : \xi$ for some context Γ and $\xi \in \mathbb{T} \cup \mathbb{T}^!$.*

(i) *The set* $\mathsf{fto}(\Pi)$ *of* fully typed occurrences of subterms of M in Π *is the set of single-hole contexts defined by structural induction on Π and by cases on its last applied rule:*

(var) $\mathsf{fto}(\Pi) = \{[]\}$.

(lam) $\mathsf{fto}(\Pi) = \{[]\} \cup \{\lambda x.C[] \mid C[] \in \mathsf{fto}(\Pi')\}$, *if $M = \lambda x.M'$ and Π' is the premise of Π.*

(app) $\mathsf{fto}(\Pi) = \{[]\} \cup \{(C[])Q \mid C[] \in \mathsf{fto}(\Pi_1)\} \cup \{P(C[]) \mid C[] \in \mathsf{fto}(\Pi_2)\}$, *where $M = PQ$ and Π_1, Π_2 are the major and minor premises of Π, respectively.*

(val_0) $\mathsf{fto}(\Pi) = \emptyset$.

($\mathrm{val}_{>0}$) $\mathsf{fto}(\Pi) = \bigcap_{1\le i\le n} \mathsf{fto}(\Pi'_i)$, *where $(\Pi'_i)_{1\le i\le n}$ are the premises of Π.*

(ii) *We say that* N is a typed subterm occurrence of M in Π *if $M = C[N]$ for $C[] \in \mathsf{fto}(\Pi)$.*

(iii) *On Π, define a measure $\mathsf{m}(\Pi) = \langle \mathsf{app}(\Pi), \mathsf{s}(M)\rangle \in \mathbb{N}^2$ (lexicographically ordered) where*
- $\mathsf{app}(\Pi)$ *is the number of* (app) *rules in Π, and*
- $\mathsf{s}(M)$ *is the measure from Proposition 3*(i)*, strictly decreasing along* (σ) *steps.*

▶ Remark 13. When the last rule of Π is ($\mathrm{val}_{>0}$), a subterm occurrence is typed if and only if it is typed in all subjects of the premises. For example, in the derivation Π of Example 11, the occurrence of y in $f(xy)$ is not fully typed since $\mathsf{fto}(\Pi) = \{[], [](xy), f([]y)\}$.

▶ **Proposition 14.** *Let $M, N \in \Lambda$ be such that $M \to_{\mathsf{v}} N$, Γ be an environment and $\alpha \in \mathbb{T}$.*

(i) *(Weighted Subject Reduction) If $\Pi \triangleright \Gamma \vdash M : \alpha$ then $\Pi' \triangleright \Gamma \vdash N : \alpha$ for some Π'. Moreover, if the redex occurrence contracted in M is fully typed in Π then $\mathsf{m}(\Pi') < \mathsf{m}(\Pi)$.*

(ii) *(Subject Expansion) If $\Gamma \vdash N : \alpha$ is derivable, then so is $\Gamma \vdash M : \alpha$.*

Proof. By Lemma 4.14 in [20]. ◀

From this proposition, the soundness of the model $\mathcal{M}$ follows easily.

▶ **Definition 15.** *The* interpretation of a λ-term M in the model $\mathcal{M}$ *is given by:*

$$[\![M]\!] = \{(\Gamma, \alpha) \mid \Gamma \vdash M : \alpha\}.$$

We write $\mathcal{M} \models M = N$ whenever $[\![M]\!] = [\![N]\!]$ holds.

▶ **Corollary 16** (Soundness). *For $M, N \in \Lambda$, $M =_{\mathsf{v}} N$ entails $\mathcal{M} \models M = N$.*

[1] This includes the case $\vdash x : []$, although x contains itself as a subterm and it is assigned a type. This is consistent with the fact that (val_0) uses the information that x is a value, without looking at its shape.

2.2 The Approximation Theory of $\mathcal{M}$

We now show that the model $\mathcal{M}$ is also well-suited to model the theory of program approximation introduced in [19] for defining Call-by-Value Böhm trees. In particular, we provide a quantitative proof of the Approximation Theorem in the spirit of [7, 9, 20].

▶ **Definition 17.**

(i) *Let $\Lambda_\perp$ be the set of λ-terms possibly containing occurrences of a constant $\perp$, and* $\mathrm{Val}_\perp = \perp \cup \mathbb{V} \cup \{\lambda x.M \mid M \in \Lambda_\perp\} \subseteq \Lambda_\perp$ *the set of* extended values.

(ii) *The set $\mathcal{A}$ of* (finite) approximants *is inductively defined by the grammar (for $n \geq 0$):*

$$\begin{array}{llll} (\mathcal{A}) & A & ::= & H \mid R \\ & H & ::= & \perp \mid x \mid \lambda x.A \mid xHA_1 \cdots A_n \\ & R & ::= & (\lambda x.A)(yHA_1 \cdots A_n) \end{array}$$

Terms of shape H are called head approximants *as they remind those used for building CbN Böhm trees, while approximants of shape R are called* redex-like *because they look like a β-redex. Let $\mathcal{H}$ (resp. $\mathcal{R}$) be the set of all head (resp. redex-like) approximants.*

(iii) *Define $\sqsubseteq_\perp \subseteq \Lambda_\perp^2$ as the least order relation compatible with abstraction and application, and including $\perp \sqsubseteq_\perp V$ for all $V \in \mathrm{Val}_\perp$. Given a set $\mathcal{X} \subseteq \Lambda_\perp$, we write $\uparrow \mathcal{X}$ if its elements are pairwise compatible, and in this case $\bigsqcup \mathcal{X}$ denotes their least upper bound.*

(iv) *For $M \in \Lambda$, define the* set $\mathcal{A}(M)$ of (finite) approximants of M *as follows*

$$\mathcal{A}(M) = \{A \in \mathcal{A} \mid \exists N \in \Lambda \,.\, M \twoheadrightarrow_{\mathsf{v}} N \text{ and } A \sqsubseteq_\perp N\}$$

We say that two λ-terms M, N have the same CbV Böhm tree *when $\mathcal{A}(M) = \mathcal{A}(N)$.*

▶ Remark 18.

(i) Although not formally needed, one could extend the v-reduction to terms in $\Lambda_\perp$ in the obvious way, and check that all approximants $A \in \mathcal{A}$ are in v-normal form. The subterm of shape H in $xHA_1 \cdots A_n$ is precisely needed to prevent a σ_3-redex.

(ii) The terminology "M and N have the same CbV Böhm tree" is consistent with [19], where the CbV Böhm tree of a λ-term M is defined as the possibly infinite tree $\bigsqcup \mathcal{A}(M)$. Indeed, it is easy to check that $\mathcal{A}(M) = \mathcal{A}(N)$ if and only if their suprema coincide.

▶ **Example 19.**

(i) $\mathcal{A}(\mathbf{\Omega}) = \mathcal{A}(\mathbf{\Omega}(xy)) = \mathcal{A}(\mathbf{ZI}) = \emptyset$.

(ii) $\mathcal{A}(\mathbf{\Delta}) = \{\perp, \lambda x.x\perp, \lambda x.xx\}$, $\mathcal{A}(\lambda x.\mathbf{\Omega}) = \{\perp\}$ and $\mathcal{A}(\mathbf{K}^*) = \{\lambda x_1 \ldots x_n.\perp \mid n \geq 0\}$.

(iii) $\mathcal{A}(\mathbf{Z}) = \bigcup_{n \in \mathbb{N}} \{\lambda f.f(\lambda z_0.f(\lambda z_1.f \cdots (\lambda z_n.f\perp Z_n) \cdots Z_1)Z_0) \mid \forall i \,.\, Z_i \in \{z_i, \perp\}\} \cup \{\perp\}$.

(iv) $\mathcal{A}(\mathbf{ZB}) = \{\perp, \lambda f_0.\perp\} \cup \{\lambda f_0 x_0.(\cdots(\lambda f_{n-1}x_{n-1}.(\lambda f_n.\perp)(f_{n-1}X_{n-1}))\cdots)(f_0 X_0) \mid$
$n > 0, \forall i \in \{1, \ldots, n\} \,.\, X_i \in \{x_i, \perp\}\}$.

▶ **Definition 20.**

(i) *The rules in Figure 1 and the interpretation $[\![-]\!]$ in Definition 15 are extended to $\Lambda_\perp$ in the obvious way. E.g.,* (val_0) *becomes* $\vdash V : []$, *for all* $V \in \mathrm{Val}_\perp$.

(ii) *We say that a derivation $\Pi \triangleright \Gamma \vdash M : \alpha$ is in* typed v-normal form *if, for all $C[] \in \mathsf{fto}(\Pi)$, $M = C[N]$ entails N is not a* v*-redex.*

(iii) *A derivation Π induces a term $M_\Pi \in \Lambda_\bot$ defined by induction on Π as follows:*

(var) $M_\Pi = x$, *if* $\Pi \triangleright \Gamma \vdash x : \alpha$.
(lam) $M_\Pi = \lambda x.M_{\Pi'}$, *if* $\Pi \triangleright \Gamma \vdash \lambda x.N : \alpha$ *and* Π' *is the premise of* Π.
(app) $M_\Pi = M_{\Pi_1} M_{\Pi_2}$, *where* Π_1, Π_2 *are the major and minor premises of* Π, *respectively.*
(val_0) $M_\Pi = \bot$.
($\text{val}_{>0}$) $M_\Pi = \bigsqcup\{M_{\Pi_i} \mid 1 \le i \le n\}$, *where* $(\Pi'_i)_{1\le i\le n}$ *are the premises of* Π.

In the case ($\text{val}_{>0}$)*, notice that* $\uparrow\{M_{\Pi_i} \mid 1 \le i \le n\}$*, whence its supremum is well-defined. Whenever* $M_\Pi \in \mathcal{A}$*, we rather call this term* A_Π *to stress the fact that it is an approximant.*

Intuitively, $\Pi \triangleright \Gamma \vdash M : \alpha$ is in typed v-nf if no redex occurrence in M is fully typed in Π.

▶ **Lemma 21.**
(i) *For all $A \in \mathcal{A}$, there exist $\alpha \in \mathbb{T}$ and Γ such that $\Gamma \vdash A : \alpha$.*
(ii) *For all $A \in \mathcal{A}$ and $N \in \Lambda$, $\Gamma \vdash A : \alpha$ and $A \sqsubseteq_\bot N$ entail $\Gamma \vdash N : \alpha$.*

Proof. Both items follow by a straightforward induction on the structure of A. ◀

▶ **Lemma 22.** *If $\Pi \triangleright \Gamma \vdash N : \alpha$ is in typed v-nf, then $M_\Pi \in \mathcal{A}(N)$ and $\Gamma \vdash M_\Pi : \alpha$.*

Proof. Straightforward induction on the structure of Π. ◀

▶ **Theorem 23** (Approximation Theorem)**.** *Let $M \in \Lambda$, $\alpha \in \mathbb{T}$ and Γ be an environment.*

$$\Gamma \vdash M : \alpha \iff \exists A \in \mathcal{A}(M)\,.\,\Gamma \vdash A : \alpha$$

Proof. ($\Rightarrow$) Assume $\Gamma \vdash M : \alpha$. By weighted subject reduction (Proposition 14(i)), $M \twoheadrightarrow_{\mathsf{v}} N$ for some $N \in \Lambda$ such that there exists $\Pi \triangleright \Gamma \vdash N : \alpha$ in typed v-nf. Conclude by Lemma 22.

($\Leftarrow$) Assume $\Gamma \vdash A : \alpha$ for some $A \in \mathcal{A}(M)$. By definition, $M \twoheadrightarrow_{\mathsf{v}} N$ for some N satisfying $A \sqsubseteq_\bot N$. By Lemma 21(ii), $\Gamma \vdash N : \alpha$. Conclude by subject expansion (Lemma 14(ii)). ◀

▶ **Corollary 24.** *If $M, N \in \Lambda$ have the same CbV Böhm trees then $\mathcal{M} \models M = N$.*

Proof. Assume $\mathcal{A}(M) = \mathcal{A}(N)$. By applying the Approximation Theorem 23, we get $[\![M]\!] = \bigcup_{A\in\mathcal{A}(M)}[\![A]\!] = \bigcup_{A\in\mathcal{A}(N)}[\![A]\!] = [\![N]\!]$. As a consequence, we conclude $\mathcal{M} \models M = N$. ◀

3 Characterizations of Operational Properties

We now provide two characterizations of the most significant properties of the calculus, namely valuability, potential valuability and solvability. The former is logical, through the type assignment system, the latter semantic, through the Approximation Theorem.

▶ **Theorem 25** (Characterizations of valuability and potential valuability)**.** *Let $M \in \Lambda$, then:*
1. *M is valuable* $\iff$ $\vdash M : []$ $\iff$ $\bot \in \mathcal{A}(M)$.
2. *M is potentially valuable* $\iff$ $\exists\Gamma, \alpha\,.\,\Gamma \vdash M : \alpha$ $\iff$ $\mathcal{A}(M) \neq \emptyset$.

Proof. See [20] for the logical characterizations, and [19] for the semantic ones. ◀

To characterize solvability, we need a deeper analysis of the structure of the approximants.

▶ **Definition 26.** *The subsets $\mathcal{S}$,$\mathcal{U} \subseteq \mathcal{A}$ are defined inductively by the grammars (for $n \geq 0$):*

$$
\begin{array}{lllllllll}
(\mathcal{S}) & S & ::= & H' \mid R' & \qquad & (\mathcal{U}) & U & ::= & \bot \mid \lambda x.U \\
 & H' & ::= & x \mid \lambda x.S \mid xHA_1 \cdots A_n & & & & \mid & (\lambda x.U)(yHA_1 \cdots A_n) \\
 & R' & ::= & (\lambda x.S)(yHA_1 \cdots A_n) & & & & &
\end{array}
$$

Note that $\{\mathcal{S},\mathcal{U}\}$ constitutes a partition of $\mathcal{A}$, namely $\mathcal{A} = \mathcal{S} \cup \mathcal{U}$ and $\mathcal{S} \cap \mathcal{U} = \emptyset$.

▶ **Example 27.**

(i) $x, \mathbf{I}, x\mathbf{K}\bot, \mathbf{I}(zz), \mathbf{\Delta}(zz), \mathbf{K}(y\mathbf{I}\bot), (\lambda x.(\mathbf{I}(yz)))(zy\bot) \in \mathcal{S}$.

(ii) $\bot, \lambda x_0 \ldots x_n.\bot, (\lambda x.\bot)(zz), (\lambda x.\bot)(y\mathbf{II}), (\lambda x.(\lambda y.\bot)(wz))(zw) \in \mathcal{U}$.

(iii) Finally, notice that $\mathcal{A}(\mathbf{\Omega}), \mathcal{A}(\mathbf{ZI}), \mathcal{A}(\lambda x.\mathbf{\Omega}), \mathcal{A}(\mathbf{K}^\star) \subseteq \mathcal{U}$.

We are going to show that the existence of an approximant $A \in \mathcal{A}(M)$ of shape S is enough to ensure the solvability of M. Conversely, when M is unsolvable, $\mathcal{A}(M)$ is only populated by approximants of shape U. We need a couple of technical lemmas.

▶ **Lemma 28** (Substitution Lemma). *Let $M \in \Lambda$, $\mathcal{A}(M) \neq \emptyset$ and $\vec{x} = \{x_1, \ldots, x_i\} \supseteq \mathrm{FV}(M)$. Then, for all $j \geq 0$ large enough and $n_1, \ldots, n_i \geq j$, we have*

$$M[\mathbf{P}_{n_1}/x_1, \ldots, \mathbf{P}_{n_i}/x_i] \twoheadrightarrow_{\mathsf{v}} V, \text{ for some } V \in \mathrm{Val} \cap \Lambda^o.$$

Moreover, if $x_m HA_1 \cdots A_n \in \mathcal{A}(M)$ then we can take $V = \mathbf{P}_\ell$, for $\ell = n_m - n - 1 \geq 0$.

Proof. If $A \in \mathcal{A}(M)$, then there is $N \in \Lambda$ such that $M \twoheadrightarrow_{\mathsf{v}} N$ and $A \sqsubseteq_\bot N$. By Fact 2, setting $\vartheta = [\mathbf{P}_{n_1}/x_1, \ldots, \mathbf{P}_{n_i}/x_i]$, we have $M^\vartheta \twoheadrightarrow_{\mathsf{v}} N^\vartheta \in \Lambda^o$. It suffices to check $N^\vartheta \twoheadrightarrow_{\mathsf{v}} V$.

By structural induction on A.

Case $A = x_m$ for some $m\,(1 \leq m \leq i)$. Then $N = x_m$, so $N^\vartheta = \mathbf{P}_{n_m}$ and we are done.

Case $A = \lambda y.A_0$. Then $N = \lambda y.N_0$ with $y \notin \vec{x}$ (wlog), whence $N^\vartheta = \lambda y.N_0^\vartheta \in \mathrm{Val}$.

Case $A = \bot$. Since $\bot \sqsubseteq_\bot N$ entails $N \in \mathrm{Val}$, we have either $N = x_m$ or $N = \lambda y.N_0$. Therefore, we proceed as above.

Case $A = x_m HA_1 \cdots A_n$ for $m\,(1 \leq m \leq i)$. Then $A \sqsubseteq_\bot N$ entails $N = x_m N_0 \cdots N_n$ with $H \in \mathcal{A}(N_0)$ and $A_r \in \mathcal{A}(N_r)$ for all $r\,(1 \leq r \leq n)$. Assuming $j > n$, we obtain

$$
\begin{array}{rcll}
N^\vartheta & = & \mathbf{P}_{n_m} N_0^\vartheta \cdots N_n^\vartheta, & \text{by definition of } \vartheta, \\
 & \twoheadrightarrow_{\mathsf{v}} & \mathbf{P}_{n_m} V_0 \cdots V_n, & \text{by I.H. (induction hypothesis)}, \\
 & \twoheadrightarrow_{\beta_{\mathsf{v}}} & \mathbf{P}_{n_m - n - 1}, & \text{with } n_m - n - 1 \geq 0, \text{ since } n_m \geq j > n.
\end{array}
$$

Case $A = (\lambda y.A_0)(xHA_1 \cdots A_n)$ with $x \in \vec{x}$ and, wlog, $y \notin \vec{x}$. From $A \sqsubseteq_\bot N$, we derive $N = (\lambda y.N_0)N_1$ where $A_0 \in \mathcal{A}(N_0)$ and $xHA_1 \cdots A_n \in \mathcal{A}(N_1)$. Easy calculations give:

$$
\begin{array}{rll}
N^\vartheta = & (\lambda y.N_0^\vartheta)N_1^\vartheta, & \text{since } y \notin \mathrm{dom}(\vartheta), \text{ then for some } \ell_1 \geq 0 \text{ we get:} \\
\twoheadrightarrow_{\mathsf{v}} & (\lambda y.N_0^\vartheta)\mathbf{P}_{\ell_1}, & \text{as the I.H. on } N_1 \text{ gives } N_1^\vartheta \twoheadrightarrow_{\mathsf{v}} \mathbf{P}_{\ell_1} \text{ since } xHA_1 \cdots A_n \in \mathcal{A}(N_1), \\
\to_{\mathsf{v}} & N_0^\vartheta[\mathbf{P}_{\ell_1}/y], & \text{by } (\beta_{\mathsf{v}}), \\
\twoheadrightarrow_{\mathsf{v}} & V, & \text{by applying the I.H. to } N_0 \text{ and } \vartheta \circ [\mathbf{P}_{\ell_1}/y].
\end{array}
$$
◀

▶ **Proposition 29** (Context Lemma). *Let $M \in \Lambda$ and $\{x_1, \ldots, x_i\} \supseteq \mathrm{FV}(M)$. If $A \in \mathcal{A}(M) \cap \mathcal{S}$ then, for all $j \geq 0$ large enough, there is $k \geq 0$ such that for all $n_1, \ldots, n_{i+k} \geq j$ we have*

$$M[\mathbf{P}_{n_1}/x_1, \ldots, \mathbf{P}_{n_i}/x_i]\mathbf{P}_{n_{i+1}} \cdots \mathbf{P}_{n_{i+k}} \twoheadrightarrow_{\mathsf{v}} \mathbf{P}_\ell, \text{ for some } \ell \geq 0.$$

Proof. Since $A \in \mathcal{A}(M)$, there exists $N \in \Lambda$ such that $M \twoheadrightarrow_{\mathsf{v}} N$ and $A \sqsubseteq_\bot N$. Now, setting $\vartheta = [\mathbf{P}_{n_1}/x_1, \ldots, \mathbf{P}_{n_i}/x_i]$, we have $M^\vartheta \twoheadrightarrow_{\mathsf{v}} N^\vartheta$. Proceed by structural induction on $A \in \mathcal{S}$.

Case $A = x$. Take $k = 0$ and proceed as in the proof of Lemma 28.

Case $A = xHA_1 \cdots A_n$. Again, take $k = 0$ and apply Lemma 28.

Case $A = \lambda y.S$. Then $N = \lambda y.N_0$ with $y \notin \vec{x}$ and $S \in \mathcal{A}(N_0)$. By induction hypothesis, there is $k' \geq 0$ such that $n_1, \ldots, n_{i+k'+1} \geq j$ entails $N_0^\vartheta[\mathbf{P}_{n_{i+1}}/y]\mathbf{P}_{n_{i+2}} \cdots \mathbf{P}_{n_{i+k'+1}} \twoheadrightarrow_{\mathsf{v}} \mathbf{P}_\ell$, for some $\ell \geq 0$. Taking $k = k'+1$, easy calculations give $(\lambda y.N_0)^\vartheta \mathbf{P}_{n_{i+1}} \cdots \mathbf{P}_{n_{i+k}} \twoheadrightarrow_{\mathsf{v}} \mathbf{P}_\ell$.

Case $A = (\lambda y.S)(x_m HA_1 \cdots A_n)$ with $1 \leq m \leq i$ and, wlog, $y \notin \vec{x}$. From $A \sqsubseteq_\perp N$, we obtain $N = (\lambda y.N_0)N_1$ with $S \in \mathcal{A}(N_0)$, $\mathrm{FV}(N_0) \subseteq \{\vec{x}, y\}$, and $x_m HA_1 \cdots A_n \in \mathcal{A}(N_1)$. By induction hypothesis, for all j' large enough, there is k' such that for all $h_1, \ldots, h_{i+k'+1} \geq j'$ we have $N_0[\mathbf{P}_{h_1}/x_1, \ldots, \mathbf{P}_{h_i}/x_i, \mathbf{P}_{h_{i+1}}/y]\mathbf{P}_{h_{i+2}} \cdots \mathbf{P}_{h_{i+k'+1}} \twoheadrightarrow_{\mathsf{v}} \mathbf{P}_\ell$, for some $\ell \geq 0$. Therefore, taking $k = k'+1$, we obtain, for all $j \geq j'+n+1$ and $n_1, \ldots, n_{i+k} \geq j$, the following:

$$\begin{array}{rcll}
N^\vartheta \mathbf{P}_{n_{i+1}} \cdots \mathbf{P}_{n_{i+k}} & = & (\lambda y.N_0^\vartheta)N_1^\vartheta \mathbf{P}_{n_{i+1}} \cdots \mathbf{P}_{n_{i+k}}, & \text{as } y \notin \mathrm{dom}(\vartheta),\\
& \twoheadrightarrow_{\mathsf{v}} & (\lambda y.N_0^\vartheta)\mathbf{P}_{n_m-n-1}\mathbf{P}_{n_{i+1}} \cdots \mathbf{P}_{n_{i+k}}, & \text{by Lemma 28},\\
& \to_{\mathsf{v}} & N_0^\vartheta[\mathbf{P}_{\ell'}/y]\mathbf{P}_{n_{i+1}} \cdots \mathbf{P}_{n_{i+k}}, & \text{setting } \ell' = n_m - n - 1,\\
& \twoheadrightarrow_{\mathsf{v}} & \mathbf{P}_\ell, & \text{by I.H. since } \ell' \geq j'.
\end{array}$$ ◀

▶ **Corollary 30.** *Let $M \in \Lambda$ and $A \in \mathcal{A}(M)$. If $A \in \mathcal{S}$ then M is solvable.*

Proof. Assume $A \in \mathcal{A}(M) \cap \mathcal{S}$ and $\mathrm{FV}(M) = \{\vec{x}\}$. By Proposition 29, there are $P_1, \ldots, P_k \in \Lambda^o$ such that $(\lambda \vec{x}.M)\vec{P} \twoheadrightarrow_{\mathsf{v}} \mathbf{P}_n$ for some $n \geq 0$. By applying the identity n times, we get $(\lambda \vec{x}.M)\vec{P}\,\mathbf{I}^{\sim n} \twoheadrightarrow_{\mathsf{v}} \mathbf{I}$. We conclude that M is solvable. ◀

▶ **Definition 31** (Proper type). *A type α is* trivial *if it has the following shape (for $n \geq 0$):*

$$\alpha = \sigma_1 \to \cdots \to \sigma_n \to []$$

The type α is called proper *if it is not trivial.*

▶ **Example 32.**

(i) Every atom $\mathsf{a} \in \mathbb{A}$ is proper.

(ii) The following types are proper: $[] \to \mathsf{a}$, $[\mathsf{a}] \to \mathsf{a}$, $[[] \to []] \to \mathsf{a}$ and $[\mathsf{a}, \mathsf{a}] \to \mathsf{a}$.

(iii) The following types are trivial: $[] \to []$, $[\mathsf{a}] \to []$, $[[] \to []] \to []$ and $[\mathsf{a}, \mathsf{a}] \to []$.

▶ Remark 33. If $\alpha \in \mathbb{T}$ is proper (resp. trivial), then so is $\sigma \to \alpha$ for all $\sigma \in \mathbb{T}^!$.

We show that solvable terms admit proper types in appropriate type environments. Conversely, unsolvables are either not typable or they only admit trivial types.

▶ **Lemma 34.** *Let $M \in \Lambda$. If M is solvable then there exist an environment Γ and a proper type α such that $\Gamma \vdash M : \alpha$ is derivable.*

Proof. Assume M solvable and let $\mathrm{FV}(M) = \{x_1, \ldots, x_k\}$. By definition of solvability, there exist $V_1, \ldots, V_n \in \mathrm{Val}$ such that $(\lambda \vec{x}.M)\vec{V} \twoheadrightarrow_{\mathsf{v}} \mathbf{I}$. Now, for β proper, we have $\vdash \mathbf{I} : [\beta] \to \beta$. By subject expansion (Proposition 14(ii)), there is a derivation $\Pi \triangleright \vdash (\lambda \vec{x}.M)\vec{V} : [\beta] \to \beta$. If $n = k = 0$ then $M \twoheadrightarrow_{\mathsf{v}} \mathbf{I}$ and we are done taking $\Gamma = \emptyset$ and $\alpha = \beta$. Otherwise, we split into cases depending on the values of n, k. By Property 8, only the following cases are possible.

- Subcase $k = n+1$. For some $\Gamma = x_1 : \sigma_1, \ldots, x_{k-1} : \sigma_{k-1}, x_k : [\beta]$, Π must have shape:

$$\dfrac{\dfrac{\dfrac{\Pi_0}{\Gamma \vdash M : \beta}}{\vdash \lambda \vec{x}.M : \sigma_1 \to \cdots \to \sigma_n \to [\beta] \to \beta}\,(\mathrm{lam}) \quad \dfrac{\Pi_1}{\vdash V_1 : \sigma_1} \cdots \dfrac{\Pi_n}{\vdash V_n : \sigma_n}}{\vdash (\lambda \vec{x}.M)V_1 \cdots V_n : [\beta] \to \beta}\,(\mathrm{app})$$

We found a derivation $\Pi_0 \triangleright \Gamma \vdash M : \beta$, so we conclude because β is proper.

- Case $k \le n > 0$. For some $\Gamma = x_1 : \sigma_1, \dots, x_k : \sigma_k$, Π must have the following shape:

$$\dfrac{\dfrac{\dfrac{\Pi_0}{\Gamma \vdash M : \sigma_{k+1} \to \cdots \to \sigma_n \to [\beta] \to \beta}}{\vdash \lambda\vec{x}.M : \sigma_1 \to \cdots \to \sigma_n \to [\beta] \to \beta}\,(\mathrm{lam}) \quad \dfrac{\Pi_1}{\vdash V_1 : \sigma_1} \quad \cdots \quad \dfrac{\Pi_n}{\vdash V_n : \sigma_n}}{\vdash (\lambda\vec{x}.M)V_1 \cdots V_n : [\beta] \to \beta}\,(\mathrm{app})$$

Thus, we can take $\alpha = \sigma_{k+1} \to \cdots \to \sigma_n \to [\beta] \to \beta$, which is proper by Remark 33. ◀

▶ **Lemma 35.** *For every $A \in \mathcal{A}$, we have:*
(i) $A \in \mathcal{S} \iff \exists \Gamma, \alpha \,.\, \Gamma \vdash A : \alpha$*, with α proper.*
(ii) $A \in \mathcal{U} \iff \forall \Gamma, \alpha \,.\, \Gamma \vdash A : \alpha$ *implies that α is trivial.*

Proof. It is enough to show that ($\Rightarrow$) holds for (i) and (ii). The converse implication follows taking the contrapositive and using the facts that $\mathcal{U} = \mathcal{A} - \mathcal{S}$ and $\mathcal{S} = \mathcal{A} - \mathcal{U}$, respectively.

(i) By induction on the structure of $A \in \mathcal{S}$ (following the grammar in Definition 26).

Case $A = x$. For every $\mathsf{a} \in \mathbb{A}$, which is a proper type, we have $x : [\mathsf{a}] \vdash x : \mathsf{a}$ by (var).

Case $A = \lambda x.S$. By I.H., there exist $\Gamma, x : \sigma$ and a proper type α such that $\Gamma, x : \sigma \vdash S : \alpha$. Thus $\Gamma \vdash \lambda x.S : \sigma \to \alpha$ is derivable by (lam), where $\sigma \to \alpha$ is a proper type by Remark 33.

Case $A = xHA_1 \cdots A_n$. In this case we can assign A any type β, in the appropriate Γ. By Lemma 21(i), there are environments $\Gamma_0, \dots, \Gamma_n$ and types $\alpha_0, \dots, \alpha_n$ such that $\Gamma_0 \vdash H : \alpha_0$ and $\Gamma_i \vdash A_i : \alpha_i$ for all $i\,(1 \le i \le n)$. Setting $\Gamma = \sum_i \Gamma_i + [x : [[\alpha_0] \to \cdots \to [\alpha_n] \to \beta]]$, we get $\Gamma \vdash xHA_1 \cdots A_n : \beta$ via ($\mathrm{val}_{>0}$) and (app). We conclude by taking, e.g., $\beta = \mathsf{a} \in \mathbb{A}$.

Case $A = (\lambda y.S)(xHA_1 \cdots A_n)$. By I.H., there exist Γ_0 and a proper type α such that $\Gamma_0 \vdash S : \alpha$. Let $\Gamma_0(y) = [\alpha_1, \dots, \alpha_k]$ with $k \ge 0$, then there are environments $\Gamma_1, \dots, \Gamma_k$ such that $\Gamma_i \vdash xHA_1 \cdots A_n : \alpha_i$ for all $i\,(1 \le i \le k)$, as we have seen above that such term can be assigned any type. Taking $\Gamma = \sum_{i=0}^{n} \Gamma_i$, we conclude $\Gamma \vdash A : \alpha$ where α is proper.

(ii) By induction on the structure of $A \in \mathcal{U}$ (following the grammar in Definition 26).

Case $A = \bot$. The only applicable rule is (val_0), namely $\vdash \bot : []$.

Case $A = \lambda x.U$. Assume that $\Gamma \vdash \lambda x.U : \sigma \to \alpha$ holds, then also $\Gamma, x : \sigma \vdash U : \alpha$ is derivable. By I.H. the type α is trivial, therefore $\sigma \to \alpha$ is also trivial by Remark 33.

Case $A = (\lambda y.U)(xHA_1 \cdots A_n)$. Assume that $\Gamma \vdash A : \alpha$ holds, then there exists a decomposition $\Gamma = \Gamma_0 + \Gamma_1$ and a $\sigma \in \mathbb{T}^!$ such that $\Gamma_0, y : \sigma \vdash U : \alpha$ and $\Gamma_1 \vdash xHA_1 \cdots A_n : \sigma$. By applying the I.H. on $\Gamma_0, y : \sigma \vdash U : \alpha$, we conclude that α is trivial. ◀

▶ **Theorem 36** (Characterizations of solvability)**.** *For $M \in \Lambda$, the following are equivalent:*
1. *M is solvable.*
2. *There exists a proper type α such that $\Gamma \vdash M : \alpha$, for some environment Γ.*
3. *There exists an approximant $A \in \mathcal{A}(M) \cap \mathcal{S}$.*

Proof. ($1 \Rightarrow 2$) By Lemma 34.

($2 \Rightarrow 3$) By the Approximation Theorem, there exists $A \in \mathcal{A}(M)$ such that $\Gamma \vdash M : \alpha$. By Lemma 35(i), we derive $A \in \mathcal{S}$.

($3 \Rightarrow 1$) By Corollary 30. ◀

▶ **Corollary 37.** *A λ-term M is unsolvable exactly when $\mathcal{A}(M) \subseteq \mathcal{U}$, equivalently, whenever $\Gamma \vdash M : \alpha$ entails that α is a trivial type.*

▶ **Corollary 38.** *The model $\mathcal{M}$ is not sensible, but semi-sensible.*

Proof. The model is not sensible as $[\![\mathbf{\Omega}]\!] = \emptyset$ and $[\![\lambda x.\mathbf{\Omega}]\!] = \{[]\}$, entail $\mathcal{M} \not\models \mathbf{\Omega} = \lambda x.\mathbf{\Omega}$. If M is solvable and N is unsolvable, by Theorem 36 there exist an environment Γ and a type α proper such that $(\Gamma, \alpha) \in [\![M]\!] - [\![N]\!]$, therefore the model is semi-sensible. ◀

$$\frac{}{\bot \in \mathsf{IM}(\emptyset;\, [])}\ (\mathsf{bot}^!) \qquad \frac{A_i \in \mathsf{IT}(\Gamma_i;\, \alpha_i) \quad \uparrow\{A_i\}_{i\in I} \quad A = \bigsqcup_{i\in I} A_i}{A \in \mathsf{IM}(\Sigma_{i\in I}\Gamma_i;\, [\alpha_i]_{i\in I})}\ (\mathsf{sup}^!)$$

$$\frac{}{\bot \in \mathsf{IT}(\emptyset;\, [])}\ (\mathsf{bot}) \qquad \frac{A \in \mathsf{IT}(\Gamma, x:\sigma;\, \alpha)}{\lambda x.A \in \mathsf{IT}(\Gamma;\, \sigma \to \alpha)}\ (\mathsf{abs}) \qquad \frac{}{x \in \mathsf{IT}(x:[\alpha];\, \alpha)}\ (\mathsf{head}_0)$$

$$\frac{A_j \in \mathsf{IM}(\Gamma_j;\, \sigma_j) \quad 0 \le j \le n \quad A_0 \in \mathcal{H}}{xA_0 \cdots A_n \in \mathsf{IT}(\sum_{j=0}^{n} \Gamma_j + x : [\sigma_0 \to \cdots \to \sigma_n \to \alpha];\, \alpha)}\ (\mathsf{head}_{>0})$$

$$\frac{A_j \in \mathsf{IM}(\Gamma_j;\, \sum_{i=0}^{m} \tau_j^i) \quad 0 \le j \le n \quad A_0 \in \mathcal{H} \quad A \in \mathsf{IT}(\Gamma_{n+1}, x : [\alpha_i]_{0\le i\le m};\, \alpha)}{(\lambda x.A)(yA_0 \cdots A_n) \in \mathsf{IT}(\sum_{j=0}^{n+1} \Gamma_j + y : [\tau_0^i \to \cdots \to \tau_n^i \to \alpha_i]_{0\le i\le m};\, \alpha)}\ (\mathsf{redlike})$$

Figure 2 The inhabitation algorithm for system $\mathcal{M}$. In (redlike), we assume $x \notin \mathrm{FV}(yA_0 \cdots A_n)$.

4 Decidability of the Inhabitation Problem

The inhabitation problem for system $\mathcal{M}$ requires to determine for every environment Γ and type α whether there is a λ-term M satisfying $\Gamma \vdash M : \alpha$. To show that this problem is decidable we describe an algorithm that takes (Γ, α) as input and returns as output the set of all approximants A satisfying $\Gamma \vdash A : \alpha$ as well as the following minimality condition.

▶ **Definition 39.** *Let Γ be an environment and $\xi \in \mathbb{T} \cup \mathbb{T}^!$. An $A \in \mathcal{A}$* is minimal for (Γ, ξ) *if $\Gamma \vdash A : \xi$ and, for all $A' \in \mathcal{A}$ compatible with A (i.e. $\uparrow\{A, A'\}$), $\Gamma \vdash A' : \xi$ entails $A \sqsubseteq_\bot A'$.*

Finding the minimal approximants inhabiting (Γ, α) is enough for solving the original inhabitation problem because $\Gamma \vdash M : \alpha$ holds exactly when there is an $A \in \mathcal{A}(M)$ minimal for (Γ, α). Following [9, 8], we present the inhabitation algorithm as a deductive system.

▶ **Definition 40.**

(i) *Let Γ be an environment and $\alpha \in \mathbb{T}$. The* inhabitation algorithm $\mathsf{IT}(\Gamma;\, \alpha)$ *for $\mathcal{M}$ is given in Figure 2, via an auxiliary predicate $\mathsf{IM}(\Gamma;\, \sigma)$, for $\sigma \in \mathbb{T}^!$. Note that the condition $A_0 \in \mathcal{H}$ occurring as a premise of the rules* (head$_{>0}$) *and* (redlike) *is decidable since $\mathcal{H}$ is generated by a context-free grammar (Definition 17*(ii)*).*

(ii) *A* run *of the algorithm is a deduction tree built bottom-up by applying the rules in Figure 2 in such a way that every node is an instance of a rule (as in Example 41). We say that* a run of the algorithm terminates *if such a tree is finite. The* algorithm terminates *if it needs to execute a finite number of different terminating runs.*

It is easy to check that $A \in \mathsf{IT}(\Gamma;\, \alpha)$ (resp. $A \in \mathsf{IM}(\Gamma;\, \sigma)$) implies $\mathrm{FV}(A) \subseteq \mathrm{dom}(\Gamma)$. We are going to prove that the inhabitation algorithm is terminating, sound and complete. Completeness is achieved by exploiting the non-determinism of the algorithm: indeed, when $\alpha = \sigma \to \beta$, the rules (abs), (redlike) and (head$_{-}$) might be applicable and in (redlike) and (head$_{>0}$), the environment Γ can be decomposed in countably many different ways (taking many $\Gamma_i = \emptyset$). By collecting all possible runs, we recover all minimal approximants for (Γ, α).

▶ **Example 41.** The following are examples of possible runs of the algorithm on $\mathsf{IT}(\Gamma;\, \alpha)$.

(i) Let $\Gamma = y : [[] \to \mathsf{a}]$ and $\alpha = \mathsf{a}$. There are two runs:

$$\frac{\dfrac{}{\bot \in \mathsf{IM}(\emptyset;\, [])}\,(\mathsf{bot}^!) \quad \bot \in \mathcal{H} \quad \dfrac{}{x \in \mathsf{IT}(x : [\mathsf{a}];\, \mathsf{a})}\,(\mathsf{head}_0)}{(\lambda x.x)(y\bot) \in \mathsf{IT}(y : [[] \to \mathsf{a}];\, \mathsf{a})}\ (\mathsf{redlike}) \qquad \frac{\dfrac{}{\bot \in \mathsf{IM}(\emptyset;\, [])}\,(\mathsf{bot}^!) \quad \bot \in \mathcal{H}}{y\bot \in \mathsf{IT}(y : [[] \to \mathsf{a}];\, \mathsf{a})}\ (\mathsf{head}_{>0})$$

(ii) Let $\Gamma = y : [[] \to []]$ and $\alpha = [\mathsf{a}] \to \mathsf{a}$. The only possible run is $\mathsf{redlike}(\mathsf{abs}(\mathsf{head}_0), \mathsf{bot}^!)$ which constructs the approximant $(\lambda x.\mathbf{I})(y\bot)$.

(iii) Let $\Gamma = \emptyset$ and $\alpha = [[\mathsf{a}] \to \mathsf{a}, [\mathsf{a}] \to \mathsf{a}] \to [\mathsf{a}] \to \mathsf{a}$. Also in this case, the only possible run is $\mathsf{abs}(\mathsf{abs}(\mathsf{head}_{>0}(\mathsf{sup}^!(\mathsf{head}_{>0}(\mathsf{sup}^!(\mathsf{head}_0))))))$, which constructs $\lambda xy.x(xy)$.

(iv) Let $\Gamma = \emptyset$ and $\alpha = [[\mathsf{a}] \to \mathsf{a}] \to [\mathsf{a}] \to \mathsf{a}$. The run $\mathsf{abs}(\mathsf{head}_0)$ constructs $\lambda x.x$, while the run $\mathsf{abs}(\mathsf{abs}(\mathsf{head}_{>0}(\mathsf{sup}^!(\mathsf{head}_0))))$ constructs $\lambda xy.xy$.

(v) Let $\Gamma = x : [[] \to [] \to \mathsf{a}]$ and $\alpha = \mathsf{a}$. There are two possible runs: $\mathsf{head}_{>0}(\mathsf{bot}^!, \mathsf{bot}^!)$, building $x\bot\bot$, and $\mathsf{redlike}(\mathsf{bot}^!, \mathsf{head}_0)$, building $(\lambda z.z)(x\bot\bot)$.

▶ **Definition 42.** *To show that the inhabitation algorithm terminates we define two* measures, $\#(\cdot)$ *on types, and* $(\cdot)^\bullet$ *on multiset types and type environments, as follows (for* $\mathsf{a} \in \mathbb{A}, n \geq 0$*):*

$$\begin{array}{lllllll} \#\mathsf{a} &=& \#[] &=& 1, & \#(\sigma \to \alpha) &=& \sigma^\bullet + \#\alpha + 3, \\ {[\alpha_1, \ldots, \alpha_n]}^\bullet &&&=& \sum_{i=1}^n \#\alpha_i, & \Gamma^\bullet &=& \sum_{x \in \mathrm{dom}(\Gamma)} \Gamma(x)^\bullet. \end{array}$$

Note that $\#\alpha \geq 1$*, while* $[]^\bullet = 0$*. If* $\sigma = \sigma_1 + \sigma_2$ *then* $\sigma^\bullet = \sigma_1^\bullet + \sigma_2^\bullet$*, thus* $\Gamma = \sum_{i \in I} \Gamma_i$ *entails* $\Gamma^\bullet = \sum_{i \in I} \Gamma_i^\bullet$*. The measure* $\#(\cdot)$ *is extended to judgements* $\mathsf{IT}(-;\, -)$ *and* $\mathsf{IM}(-;\, -)$ *by*

$$\#(\mathsf{IT}(\Gamma;\, \alpha)) = \Gamma^\bullet + \#\alpha, \qquad \#(\mathsf{IM}(\Gamma;\, \sigma)) = \Gamma^\bullet + \sigma^\bullet + 1.$$

Given $M \in \Lambda_\bot$*, we define inductively the* size of its syntax-tree*, written* $\mathsf{tsize}(M)$*, by:*

$$\mathsf{tsize}(\bot) = \mathsf{tsize}(x) = 0, \quad \mathsf{tsize}(\lambda x.P) = \mathsf{tsize}(P) + 1, \quad \mathsf{tsize}(PQ) = \mathsf{tsize}(P) + \mathsf{tsize}(Q) + 1.$$

▶ **Example 43.**

(i) We have $\#([] \to []) = \#([] \to \mathsf{a}) = 4$, so $(x : [[] \to [], [] \to \mathsf{a}, \mathsf{a}])^\bullet = 9$.

(ii) Since $\#[[\mathsf{a}] \to \mathsf{a}] = 5$, we get $\#\mathsf{IT}(x : [[\mathsf{a}] \to \mathsf{a}];\, \mathsf{a}) = 6$, while $\#\mathsf{IM}(x : [[\mathsf{a}] \to \mathsf{a}];\, [\mathsf{a}]) = 7$.

(iii) $\mathsf{tsize}((\lambda x.\bot)(x\bot)) = 3$, $\mathsf{tsize}(\mathbf{P}_n) = n + 1$ and $\mathsf{tsize}(x\bot^{\sim n}) = n$, for all $n \geq 0$.

▶ **Lemma 44.** *Every run of the inhabitation algorithm terminates.*

Proof. We need to show that every run is a finite tree. Since we are considering finite multisets and all indices range over $\mathbb{N}$, the premises of each rule in Figure 2 are finitely many (i.e., a run is a finitely branching tree), whence it is enough to show that there is no infinite path (by König's Lemma). This follows from the fact that the measure # calculated on each premise of a rule, is strictly smaller than the measure associated with its conclusion. We proceed by cases on the rules applied, the cases $(\mathsf{bot}), (\mathsf{bot}^!)$, and (head_0) being vacuous.

Cases (abs) and $(\mathsf{sup}^!)$ follow straightforwardly from Definition 42.

Case $(\mathsf{head}_{>0})$ with premises $\mathsf{IM}(\Gamma_j;\, \sigma_j)$, for all $j\,(0 \leq j \leq n)$, and as a conclusion $\mathsf{IT}(\Gamma + x : [\sigma_0 \to \cdots \to \sigma_n \to \alpha];\, \alpha)$ for $\Gamma = \sum_{j=0}^n \Gamma_j$. The measure # applied to the j-th premise gives $\Gamma_j^\bullet + \sigma_j^\bullet + 1$; on the conclusion, it gives $\Gamma^\bullet + \sigma_0^\bullet + \cdots + \sigma_n^\bullet + 2(\#\alpha) + 3(n+1)$. In the worse case, namely $n = j = 0$ and $\#\alpha = 1$, we still get $\Gamma_0^\bullet + \sigma_0^\bullet + 1 < \Gamma_0^\bullet + \sigma_0^\bullet + 5$.

Case $(\mathsf{redlike})$ with premises $\mathsf{IM}(\Gamma_j;\, \sum_{i=0}^m \tau_{ij})$, for $0 \leq j \leq n$, and $\mathsf{IT}(\Gamma_{n+1}, x : [\alpha_i]_{0 \leq i \leq m};\, \alpha)$, and conclusion $\mathsf{IT}(\Gamma + y : [\tau_{i0} \to \cdots \to \tau_{in} \to \alpha_i]_{0 \leq i \leq m};\, \alpha)$ for $\Gamma = \sum_{j=0}^n \Gamma_j + \Gamma_{n+1}$. For the measure applied to the conclusion, easy calculations give the following number K:

$$\begin{array}{lll} K &=& \Gamma^\bullet + 3(n+1)(m+1) + \sum_{i=0}^m (\sum_{j=0}^n \tau_{ij}^\bullet + \#\alpha_i) + \#\alpha \\ &=& \Gamma_{n+1}^\bullet + 3(n+1)(m+1) + \sum_{j=0}^n (\Gamma_j^\bullet + \sum_{i=0}^m (\tau_{ij}^\bullet + \#\alpha_i)) + \#\alpha \end{array}$$

For the j-th premise we can easily check $\Gamma_j^\bullet + \sum_{i=0}^m \tau_{ij}^\bullet + 1 < K$. For the remaining one, we get $\Gamma_{n+1}^\bullet + \sum_{i=0}^m \#\alpha_i + \#\alpha$. In the worst case, i.e. $n = m = 0$, $\Gamma^\bullet = \Gamma_0^\bullet + \Gamma_1^\bullet = \Gamma_1^\bullet$ and $\tau_{00}^\bullet = 0$, we obtain $\Gamma^\bullet + \#\alpha_0 + \#\alpha < \Gamma^\bullet + \#\alpha_0 + \#\alpha + 3$. This concludes the proof. ◀

We show that the size of the approximants generated by $\mathsf{IT}(\Gamma;\,\alpha)$ is bounded by $\Gamma^\bullet + \#\alpha$. In fact, the coefficient 3 in the definition of $\#(\sigma \to \alpha)$ has been chosen to absorb the size of the "$\lambda x.$" and of the outer application in redex-like approximants as $(\lambda x.A)(yA_0 \cdots A_n)$.

▶ **Lemma 45.** *For a type environment Γ, $\alpha \in \mathbb{T}$, $\sigma \in \mathbb{T}^!$, we have:*
(i) $A \in \mathsf{IT}(\Gamma;\,\alpha)$ *entails* $\mathsf{tsize}(A) \le \#\mathsf{IT}(\Gamma;\,\alpha)$.
(ii) $A \in \mathsf{IM}(\Gamma;\,\sigma)$ *entails* $\mathsf{tsize}(A) < \#\mathsf{IM}(\Gamma;\,\sigma)$.

Proof. We prove (i) and (ii) by induction on a run of $A \in \mathsf{IT}(\Gamma;\,\alpha)$ (resp. $A \in \mathsf{IM}(\Gamma;\,\sigma)$).

Cases (bot), ($\mathsf{bot}^!$) and (head_0). Trivial, since $\mathsf{tsize}(\bot) = \mathsf{tsize}(x) = 0$ and $\mathsf{IM}(\Gamma;\,\alpha) \ge 1$.

Case ($\mathsf{sup}^!$) follows from I.H., because $A = \bigsqcup_{i\in I} A_i$ implies $\mathsf{tsize}(A) \le \sum_{i\in I} \mathsf{tsize}(A_i)$.

Case (abs) with $\alpha = \sigma \to \beta$. By induction hypothesis, we get $\mathsf{tsize}(A) \le \Gamma^\bullet + \sigma^\bullet + \#\beta$, therefore we obtain $\mathsf{tsize}(\lambda x.A) = \mathsf{tsize}(A) + 1 \le \Gamma^\bullet + \sigma^\bullet + \#\beta + 3 = \#\mathsf{IT}(\Gamma;\,\sigma \to \beta)$.

Case ($\mathsf{head}_{>0}$) with $\Gamma = \sum_{j=0}^{n} \Gamma_j + x : [\sigma_0 \to \cdots \to \sigma_n \to \alpha]$. By IH, $\mathsf{tsize}(A_j) \le \Gamma_j^\bullet + \sigma_j^\bullet$. So, $\mathsf{tsize}(xA_0 \cdots A_n) = \sum_{j=0}^{n} \mathsf{tsize}(A_j) + n + 1 \le \sum_{j=0}^{n} \Gamma_j^\bullet + \sigma_j^\bullet + 2\#\alpha + 3(n+1) = \#\mathsf{IT}(\Gamma;\,\alpha)$.

Case ($\mathsf{redlike}$) with $\Gamma = \sum_{j=0}^{n+1} \Gamma_j + y : [\tau_{i0} \to \cdots \to \tau_{in} \to \alpha_i]_{0\le i\le m}$. By I.H., we have $\mathsf{tsize}(A_j) \le \Gamma_j^\bullet + \sum_{i=0}^{m} \tau_{ij}^\bullet$ for all j ($0 \le j \le n$), and $\mathsf{tsize}(A) \le \Gamma_{n+1}^\bullet + \sum_{i=0}^{m} \#\alpha_i + \#\alpha$. Thus, $\mathsf{tsize}(yA_0 \cdots A_n) = \sum_{j=0}^{n} \mathsf{tsize}(A_j) + n + 1 \le \sum_{j=0}^{n} \Gamma_j^\bullet + \sum_{i=0}^{m} (\tau_{i0}^\bullet + \cdots + \tau_{in}^\bullet) + n + 1$ and:

$$\begin{aligned}
\mathsf{tsize}((\lambda x.A)(yA_0 \cdots A_n)) = {} & \mathsf{tsize}(\lambda x.A) + \mathsf{tsize}(yA_0 \cdots A_n) + 1 \\
\le {} & \Gamma_{n+1}^\bullet + \textstyle\sum_{i=0}^{m} \#\alpha_i + \#\alpha + \mathsf{tsize}(yA_0 \cdots A_n) + 2 \\
\le {} & \textstyle\sum_{j=0}^{n+1} \Gamma_j^\bullet + \sum_{i=0}^{m} (\tau_{i0}^\bullet + \cdots + \tau_{in}^\bullet + \#\alpha_i) + \#\alpha + n + 3 \\
\le {} & \textstyle\sum_{j=0}^{n+1} \Gamma_j^\bullet + \sum_{i=0}^{m} (\sum_{j=0}^{n} \tau_{ij}^\bullet + \#\alpha_i) + \#\alpha + 3(n+1)(m+1)
\end{aligned}$$

where the last inequation holds since $n + 3 \le 3(n+1)(m+1)$ for all $n, m \ge 0$. ◀

▶ **Theorem 46** (Termination)**.** *The inhabitation algorithm terminates.*

Proof. Fix an input (Γ, α). By Lemma 44, every run $A \in \mathsf{IT}(\Gamma;\,\alpha)$ terminates. The set $\{A \in \mathcal{A} \mid \mathrm{FV}(A) \subseteq \mathrm{dom}(\Gamma) \wedge \mathsf{tsize}(A) \le \Gamma^\bullet + \#\alpha\}$ is finite, because one cannot add variables or $\bot$ without adding applications. By Lemma 45, we get that the number of runs is finite. ◀

To better understand the inhabitation algorithm it is convenient to provide an effective way of constructing minimal approximants. We have seen in Definition 20(iii) that we are able to associate an approximant A_Π with every derivation Π in typed v-normal form. This last condition is always satisfied by derivations $\Pi \triangleright \Gamma \vdash A : \alpha$ for $A \in \mathcal{A}$ because approximants do not contain any occurrence of a v-redex. We now show that the approximants A_Π so constructed are minimal for (Γ, α) and that all such minimal approximants arise in this way.

▶ **Lemma 47.** *Let Γ be a type environment, $\alpha \in \mathbb{T}$ and $A \in \mathcal{A}$. The following are equivalent:*
1. $A \in \mathsf{IT}(\Gamma;\,\alpha)$.
2. $A = A_\Pi$ *for some derivation* $\Pi \triangleright \Gamma \vdash A : \alpha$.
3. A *is minimal for* (Γ, α).

Proof. To perform the induction properly, we prove simultaneously the analogous statement on $\sigma \in \mathbb{T}^!$: $A \in \mathsf{IM}(\Gamma;\,\sigma) \iff A = A_\Pi$ for some $\Pi \triangleright \Gamma \vdash A : \sigma \iff A$ is minimal for (Γ, σ).

($1 \Rightarrow 2$) By induction on a run of $A \in \mathsf{IT}(\Gamma;\,\alpha)$ (resp. $A \in \mathsf{IM}(\Gamma;\,\sigma)$).

Cases (bot), ($\mathsf{bot}^!$) and (head_0) are trivial.

Cases ($\mathsf{sup}^!$) and (abs). Easy. Use the I.H. and apply ($\mathrm{val}_{>0}$) and (lam), respectively.

Case ($\mathsf{head}_{>0}$) with $\Gamma = \sum_{j=0}^{n} \Gamma_j + \Gamma'$ where $\Gamma' = x : [\sigma_0 \to \cdots \to \sigma_n \to \alpha]$ and $A = xA_0 \cdots A_n$. Let $\Pi' \triangleright \Gamma_{n+1} \vdash x : \sigma_0 \to \cdots \to \sigma_n \to \alpha$ with $A_{\Pi'} = x$. By I.H., for every $j (0 \leq j \leq n)$, there is a derivation $\Pi_j \triangleright \Gamma_j \vdash A_j : \sigma_j$ such that $A_j = A_{\Pi_j}$. For Π, take:

$$\dfrac{\Pi' \triangleright \Gamma' \vdash x : \sigma_0 \to \cdots \to \sigma_n \to \alpha \quad \Pi_j \triangleright \Gamma_j \vdash A_j : \sigma_j \quad 0 \leq j \leq n}{\Gamma \vdash xA_0 \cdots A_n : \alpha}\ (\mathrm{app})$$

and conclude because $A_\Pi = A_{\Pi'} A_{\Pi_0} \cdots A_{\Pi_n} = xA_0 \cdots A_n$.

Case ($\mathsf{redlike}$) with $\Gamma = \sum_{j=0}^{n+1} \Gamma_j + \Gamma'$, where $\Gamma' = y : [\tau_0^i \to \cdots \to \tau_n^i \to \alpha_i]_{0 \leq i \leq m}$, and $A = (\lambda x.A')(yA_0 \cdots A_n)$. By I.H., there exists $\Pi_{n+1} \triangleright \Gamma_{n+1}, x : [\alpha_i]_{0 \leq i \leq m} \vdash A' : \alpha$ with $A' = A_{\Pi_{n+1}}$. Moreover, for each $0 \leq j \leq n$, there is $\Pi_j \triangleright \Gamma_j \vdash A_j : \sum_{i=0}^{m} \tau_j^i$ with $A_j = A_{\Pi_j}$. This holds exactly when there exists a decomposition $\Gamma_j = \sum_{i=0}^{m} \Gamma_j^i$ and $\Pi_j^i \triangleright \Gamma_j^i \vdash A_j : \tau_j^i$ satisfying $A_j = \bigsqcup_{i=0}^{m} A_{\Pi_j^i}$, although individually $A_j \neq A_{\Pi_j^i}$ might hold. Construct Π as:

$$\dfrac{\dfrac{\begin{array}{c}\Pi_{n+1} \triangleright \\ \Gamma_{n+1}, x : [\alpha_i]_{0 \leq i \leq m} \vdash A' : \alpha\end{array}}{\Gamma_{n+1} \vdash \lambda x.A' : [\alpha_i]_{0 \leq i \leq m} \to \alpha} \quad \dfrac{\dfrac{\begin{array}{c} y : [\tau_0^i \to \cdots \to \tau_n^i \to \alpha_i] \\ \vdash y : \tau_0^i \to \cdots \to \tau_n^i \to \alpha_i \end{array} \quad \begin{array}{c}\Pi_0^i \triangleright \\ \Gamma_0^i \vdash A_0 : \tau_0^i\end{array} \ \cdots \ \begin{array}{c}\Pi_n^i \triangleright \\ \Gamma_n^i \vdash A_n : \tau_n^i\end{array}}{\sum_{j=0}^{n} \Gamma_j^i + y : [\tau_0^i \to \cdots \to \tau_n^i \to \alpha_i] \vdash yA_0 \cdots A_n : \alpha_i}\ \forall i}{\sum_{j=0}^{n} \Gamma_j + \Gamma' \vdash yA_0 \cdots A_n : [\alpha_i]_{0 \leq i \leq m}}}{\sum_{j=0}^{n+1} \Gamma_j + \Gamma' \vdash (\lambda x.A')(yA_0 \cdots A_n) : \alpha}$$

It is now easy to check that $(\lambda x.A')(yA_0 \cdots A_n) = A_\Pi$, for the derivation Π above.

($2 \Rightarrow 3$) By straightforward induction on $\Pi \triangleright \Gamma \vdash A : \alpha$. In the case ($\mathrm{val}_{>0}$) with premises $(\Pi_i)_{i \in I}$, use the fact that $A_\Pi = \bigsqcup_{i \in I} A_{\Pi_i}$ is defined as the least upper bound.

($3 \Rightarrow 1$) By induction on a derivation $\Pi \triangleright \Gamma \vdash A : \alpha$, where A is minimal for (Γ, α). The only non-trivial case to handle is (app). We split into subcases depending on the shape of A.

Subcase $A = xA_0 \cdots A_n$ with $A_0 \in \mathcal{H}$. Then there is a decomposition $\Gamma = \sum_{j=0}^{n} \Gamma_j + x : [\sigma_0 \to \cdots \to \sigma_n \to \alpha]$ such that Π has subderivations $\Pi_j \triangleright \Gamma_j \vdash A_j : \sigma_j$ with A_j minimal for (Γ_j, σ_j). By I.H., $A_j \in \mathsf{IM}(\Gamma_j; \sigma_j)$ from which $xA_0 \cdots A_n \in \mathsf{IT}(\Gamma; \alpha)$ follows by ($\mathsf{head}_{>0}$).

Subcase $A = (\lambda x.A')(yHA_1 \cdots A_n)$. Then, the derivation $\Pi \triangleright \Gamma \vdash A : \alpha$ must have the shape above (see proof of ($1 \Rightarrow 2$), case ($\mathsf{redlike}$)) for some decomposition $\Gamma = \sum_{j=0}^{n+1} \Gamma_j + \Gamma'$, where $\Gamma' = y : [\tau_0^i \to \cdots \to \tau_n^i \to \alpha_i]_{0 \leq i \leq m}$ and setting $A_0 = H \in \mathcal{H}$. Since A is minimal for (Γ, α) and $\Gamma_j^i \vdash A_j : \tau_j^i$ for every j ($0 \leq j \leq n$), we must have A_j minimal for $(\Gamma_j, \sum_{i=0}^{m} \tau_j^i)$ and A' minimal for $((\Gamma_{n+1}, x : [\alpha_i]_{0 \leq i \leq m}), \alpha)$. By I.H., we obtain $A_j \in \mathsf{IM}(\Gamma_j; \sum_{i=0}^{m} \tau_j^i)$ and $A' \in \mathsf{IT}((\Gamma_{n+1}, x : [\alpha_i]_{0 \leq i \leq m}); \alpha)$. As $A_0 \in \mathcal{H}$, we get $A \in \mathsf{IT}(\Gamma; \alpha)$ by applying ($\mathsf{redlike}$). ◀

▶ **Theorem 48** (Soundness and Completeness)**.**

(i) *If* $A \in \mathsf{IT}(\Gamma; \alpha)$ *then, for all* $M \in \Lambda$ *satisfying* $A \sqsubseteq_\perp M$*, we have* $\Gamma \vdash M : \alpha$*.*

(ii) *If* $\Gamma \vdash M : \alpha$ *then there exists* $A \in \mathsf{IT}(\Gamma; \alpha)$ *such that* $A \in \mathcal{A}(M)$*.*

Proof. (i) By Lemma 47, we have $\Gamma \vdash A : \alpha$. Since $A \sqsubseteq_\perp M$, we conclude by Lemma 21(ii).

(ii) By the Approximation Theorem, there exists $A' \in \mathcal{A}(M)$ satisfying $\Gamma \vdash A' : \alpha$. Then, there is an approximant $A \uparrow A'$ which is minimal for (Γ, α). By Lemma 47, we obtain $A \in \mathsf{IT}(\Gamma; \alpha)$ and since $\mathcal{A}(M)$ is downward closed (by definition) we conclude $A \in \mathcal{A}(M)$. ◀

Conclusions

In this paper we have shown that the model $\mathcal{M}$ allows to characterize solvability semantically, but we believe that Theorem 36 extends to all relational models defined in [20] having a non-empty set of atoms, and whose type equivalence preserves the non-triviality of the types. The fact that $\mathcal{M}$ constitutes a model of CbV λ-calculus has been shown in [20] by exploiting the environmental definition *à la* Hindley-Longo (namely, Definition 10.0.1 in [26]). In future works, we plan to analyze the categorical construction behind this class of models as they do not seem to be an instance of any categorical definition proposed so far.

References

1. Samson Abramsky. Domain theory in logical form. *Ann. Pure Appl. Log.*, 51(1-2):1–77, 1991. doi:10.1016/0168-0072(91)90065-T.
2. Beniamino Accattoli and Giulio Guerrieri. Open call-by-value. In Atsushi Igarashi, editor, *Programming Languages and Systems – 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 206–226, 2016. doi:10.1007/978-3-319-47958-3_12.
3. Beniamino Accattoli and Giulio Guerrieri. Types of fireballs. In Sukyoung Ryu, editor, *Programming Languages and Systems – 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, volume 11275 of *Lecture Notes in Computer Science*, pages 45–66. Springer, 2018. doi:10.1007/978-3-030-02768-1_3.
4. Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, 2013. URL: http://www.cambridge.org/de/academic/subjects/mathematics/logic-categories-and-sets/lambda-calculus-types.
5. Henk P. Barendregt. *The lambda-calculus, its syntax and semantics*. Number 103 in Studies in Logic and the Foundations of Mathematics. North-Holland, second edition, 1984.
6. O. Bastonero, Alberto Pravato, and Simona Ronchi Della Rocca. Structures for lazy semantics. In David Gries and Willem P. de Roever, editors, *Programming Concepts and Methods, IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods (PROCOMET '98) 8-12 June 1998, Shelter Island, New York, USA*, volume 125 of *IFIP Conference Proceedings*, pages 30–48. Chapman & Hall, 1998.
7. Flavien Breuvart, Giulio Manzonetto, and Domenico Ruoppolo. Relational graph models at work. *Log. Methods Comput. Sci.*, 14(3), 2018. doi:10.23638/LMCS-14(3:2)2018.
8. Antonio Bucciarelli, Delia Kesner, and Simona Ronchi Della Rocca. Solvability = typability + inhabitation. *Log. Methods Comput. Sci.*, 17(1), 2021. URL: https://lmcs.episciences.org/7141.
9. Antonio Bucciarelli, Delia Kesner, and Simona Ronchi Della Rocca. Inhabitation for non-idempotent intersection types. *Log. Methods Comput. Sci.*, 14(3), 2018. doi:10.23638/LMCS-14(3:7)2018.
10. Albero Carraro and Giulio Guerrieri. A semantical and operational account of call-by-value solvability. In Anca Muscholl, editor, *Foundations of Software Science and Computation Structures – 17th International Conference, FOSSACS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8412 of *Lecture Notes in Computer Science*, pages 103–118. Springer, 2014. doi:10.1007/978-3-642-54830-7_7.
11. Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Math. Log. Q.*, 27(2-6):45–58, 1981. doi:10.1002/malq.19810270205.
12. Daniel de Carvalho. Execution time of λ-terms via denotational semantics and intersection types. *Math. Struct. Comput. Sci.*, 28(7):1169–1203, 2018. doi:10.1017/S0960129516000396.
13. Lavinia Egidi, Furio Honsell, and Simona Ronchi Della Rocca. Operational, denotational and logical descriptions: a case study. *Fundam. Informaticae*, 16(1):149–169, 1992.
14. Thomas Ehrhard. Collapsing non-idempotent intersection types. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic (CSL'12) – 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, volume 16 of *LIPIcs*, pages 259–273. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPIcs.CSL.2012.259.

15 Jean-Yves Girard. Normal functors, power series and λ-calculus. *Ann. Pure Appl. Log.*, 37(2):129–177, 1988. doi:10.1016/0168-0072(88)90025-5.

16 Giulio Guerrieri. Personal communication, 2017.

17 Giulio Guerrieri, Luca Paolini, and Simona Ronchi Della Rocca. Standardization and conservativity of a refined call-by-value lambda-calculus. *Log. Methods Comput. Sci.*, 13(4), 2017. doi:10.23638/LMCS-13(4:29)2017.

18 Furio Honsell and Marina Lenisa. Some results on the full abstraction problem for restricted lambda calculi. In Andrzej M. Borzyszkowski and Stefan Sokolowski, editors, *Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93, Gdansk, Poland, August 30 – September 3, 1993, Proceedings*, volume 711 of *Lecture Notes in Computer Science*, pages 84–104. Springer, 1993. doi:10.1007/3-540-57182-5_6.

19 Emma Kerinec, Giulio Manzonetto, and Michele Pagani. Revisiting call-by-value Böhm trees in light of their Taylor expansion. *Log. Methods Comput. Sci.*, 16(3), 2020. URL: https://lmcs.episciences.org/6638.

20 Giulio Manzonetto, Michele Pagani, and Simona Ronchi Della Rocca. New semantical insights into call-by-value λ-calculus. *Fundam. Informaticae*, 170(1-3):241–265, 2019. doi:10.3233/FI-2019-1862.

21 Luca Paolini and Simona Ronchi Della Rocca. Call-by-value solvability. *RAIRO Theor. Informatics Appl.*, 33(6):507–534, 1999. doi:10.1051/ita:1999130.

22 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.

23 Alberto Pravato, Simona Ronchi Della Rocca, and Luca Roversi. The call by value λ-calculus: a semantic investigation. *Mathematical Structures in Computer Science*, 9(5):617–650, 1999.

24 Laurent Regnier. Une équivalence sur les lambda-termes. *Theor. Comput. Sci.*, 126(2):281–292, 1994. doi:10.1016/0304-3975(94)90012-4.

25 Simona Ronchi Della Rocca. Intersection types and denotational semantics: An extended abstract (invited paper). In Silvia Ghilezan, Herman Geuvers, and Jelena Ivetic, editors, *22nd International Conference on Types for Proofs and Programs, TYPES 2016, May 23-26, 2016, Novi Sad, Serbia*, volume 97 of *LIPIcs*, pages 2:1–2:7. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.TYPES.2016.2.

26 Simona Ronchi Della Rocca and Luca Paolini. *The Parametric λ-Calculus: a Metamodel for Computation.* EATCS Series. Springer, Berlin, 2004.

27 Pawel Urzyczyn. The emptiness problem for intersection types. *J. Symb. Log.*, 64(3):1195–1215, 1999. doi:10.2307/2586625.

28 Pawel Urzyczyn. Personal communication, 2014.

29 Steffen van Bakel. Complete restrictions of the intersection type discipline. *Theor. Comput. Sci.*, 102(1):135–163, 1992. doi:10.1016/0304-3975(92)90297-S.

Predicative Aspects of Order Theory in Univalent Foundations

Tom de Jong
University of Birmingham, UK

Martín Hötzel Escardó
University of Birmingham, UK

Abstract

We investigate predicative aspects of order theory in constructive univalent foundations. By predicative and constructive, we respectively mean that we do not assume Voevodsky's propositional resizing axioms or excluded middle. Our work complements existing work on predicative mathematics by exploring what *cannot* be done predicatively in univalent foundations. Our first main result is that nontrivial (directed or bounded) complete posets are necessarily large. That is, if such a nontrivial poset is small, then weak propositional resizing holds. It is possible to derive full propositional resizing if we strengthen nontriviality to positivity. The distinction between nontriviality and positivity is analogous to the distinction between nonemptiness and inhabitedness. We prove our results for a general class of posets, which includes directed complete posets, bounded complete posets and sup-lattices, using a technical notion of a $\delta_{\mathcal{V}}$-complete poset. We also show that nontrivial locally small $\delta_{\mathcal{V}}$-complete posets necessarily lack decidable equality. Specifically, we derive weak excluded middle from assuming a nontrivial locally small $\delta_{\mathcal{V}}$-complete poset with decidable equality. Moreover, if we assume positivity instead of nontriviality, then we can derive full excluded middle. Secondly, we show that each of Zorn's lemma, Tarski's greatest fixed point theorem and Pataraia's lemma implies propositional resizing. Hence, these principles are inherently impredicative and a predicative development of order theory must therefore do without them. Finally, we clarify, in our predicative setting, the relation between the traditional definition of sup-lattice that requires suprema for all subsets and our definition that asks for suprema of all small families.

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

Keywords and phrases order theory, constructivity, predicativity, univalent foundations

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.8

1 Introduction

We investigate predicative aspects of order theory in constructive univalent foundations. By predicative and constructive, we respectively mean that we do not assume Voevodsky's propositional resizing axioms [26, 27] or excluded middle. Our work is situated in our larger programme of developing domain theory constructively and predicatively in univalent foundations. In previous work [12], we showed how to give a constructive and predicative account of many familiar constructions and notions in domain theory, such as Scott's D_∞ model of untyped λ-calculus and the theory of continuous dcpos. The present work complements this and other existing work on predicative mathematics (e.g. [2, 21, 6]) by exploring what *cannot* be done predicatively, as in [7, 8, 9, 10, 11]. We do so by showing that certain statements crucially rely on resizing axioms in the sense that they are equivalent to them. Such arguments are important in constructive mathematics. For example, the constructive failure of trichotomy on the real numbers is shown [4] by reducing it to a nonconstructive instance of excluded middle.

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi; Article No. 8; pp. 8:1–8:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Our first main result is that nontrivial (directed or bounded) complete posets are necessarily large. In [12] we observed that all our examples of directed complete posets have large carriers. We show here that this is no coincidence, but rather a necessity, in the sense that if such a nontrivial poset is small, then weak propositional resizing holds. It is possible to derive full propositional resizing if we strengthen nontriviality to positivity in the sense of [19]. The distinction between nontriviality and positivity is analogous to the distinction between nonemptiness and inhabitedness. We prove our results for a general class of posets, which includes directed complete posets, bounded complete posets and sup-lattices, using a technical notion of a $\delta_{\mathcal{V}}$-complete poset. We also show that nontrivial locally small $\delta_{\mathcal{V}}$-complete posets necessarily lack decidable equality. Specifically, we can derive weak excluded middle from assuming the existence of a nontrivial locally small $\delta_{\mathcal{V}}$-complete poset with decidable equality. Moreover, if we assume positivity instead of nontriviality, then we can derive full excluded middle.

Secondly, we prove that each of Zorn's lemma, Tarski's greatest fixed point theorem and Pataraia's lemma implies propositional resizing. Hence, these principles are inherently impredicative and a predicative development of order theory in univalent foundations must thus forgo them.

Finally, we clarify, in our predicative setting, the relation between the traditional definition of sup-lattice that requires suprema for all subsets and our definition that asks for suprema of all small families. This is important in practice in order to obtain workable definitions of dcpo, sup-lattice, etc. in the context of predicative univalent mathematics.

Our foundational setup is the same as in [12], meaning that our work takes places in intensional Martin-Löf Type Theory and adopts the univalent point of view [24]. This means that we work with the stratification of types as singletons, propositions (or subsingletons or truth values), sets, 1-groupoids, etc., and that we work with univalence. At present, higher inductive types other than propositional truncation are not needed. Often the only consequences of univalence needed here are functional and propositional extensionality. An exception is Section 2.3. Full details of our univalent type theory are given at the start of Section 2.

Related work

Curi investigated the limits of predicative mathematics in CZF [2] in a series of papers [7, 8, 9, 10, 11]. In particular, Curi shows (see [7, Theorem 4.4 and Corollary 4.11], [8, Lemma 1.1] and [9, Theorem 2.5]) that CZF cannot prove that various nontrivial posets, including sup-lattices, dcpos and frames, are small. This result is obtained by exploiting that CZF is consistent with the anti-classical generalized uniformity principle GUP [25, Theorem 4.3.5]. Our related Theorem 35 is of a different nature in two ways. Firstly, our theorem is in the spirit of reverse constructive mathematics [18]: Instead of showing that GUP implies that there are no non-trivial small dcpos, we show that the existence of a non-trivial small dcpo is *equivalent* to weak propositional resizing, and that the existence of a positive small dcpo is *equivalent* to full propositional resizing. Thus, if we wish to work with small dcpos, we are forced to assume resizing axioms. Secondly, we work in univalent foundations rather than CZF. This may seem a superficial difference, but a number of arguments in Curi's papers [9, 10] crucially rely on set-theoretical notions and principles such as transitive set, set-induction, weak regular extension axiom wREA, which cannot even be formulated in the underlying type theory of univalent foundations. Moreover, although Curi claims that the arguments of [7, 8] can be adapted to some version of Martin-Löf Type Theory, it is presently not known whether there is any model of univalent foundations which validates GUP.

Organization

Section 2: Foundations and size matters, including impredicativity, relation to excluded middle, univalence and closure under embedded retracts. *Section 3*: Nontrivial and positive $\delta_{\mathcal{V}}$-complete posets and reductions to impredicativity and excluded middle. *Section 4*: Predicative invalidity of Zorn's lemma, Tarski's fixed point theorem and Pataraia's lemma. *Section 5*: Comparison of completeness w.r.t. families and w.r.t. subsets. *Section 6*: Conclusion and future work.

2 Foundations and Size Matters

We work with a subset of the type theory described in [24] and we mostly adopt the terminological and notational conventions of [24]. We include $+$ (binary sum), Π (dependent products), Σ (dependent sum), Id (identity type), and inductive types, including $\mathsf{0}$ (empty type), $\mathsf{1}$ (type with exactly one element $\star : \mathsf{1}$), N (natural numbers). We assume a universe $\mathcal{U}_0$ and two operations: for every universe $\mathcal{U}$ a successor universe $\mathcal{U}^+$ with $\mathcal{U} : \mathcal{U}^+$, and for every two universes $\mathcal{U}$ and $\mathcal{V}$ another universe $\mathcal{U} \sqcup \mathcal{V}$ such that for any universe $\mathcal{U}$, we have $\mathcal{U}_0 \sqcup \mathcal{U} \equiv \mathcal{U}$ and $\mathcal{U} \sqcup \mathcal{U}^+ \equiv \mathcal{U}^+$. Moreover, $(-) \sqcup (-)$ is idempotent, commutative, associative, and $(-)^+$ distributes over $(-) \sqcup (-)$. We write $\mathcal{U}_1 :\equiv \mathcal{U}_0^+$, $\mathcal{U}_2 :\equiv \mathcal{U}_1^+, \dots$ and so on. If $X : \mathcal{U}$ and $Y : \mathcal{V}$, then $X + Y : \mathcal{U} \sqcup \mathcal{V}$ and if $X : \mathcal{U}$ and $Y : X \to \mathcal{V}$, then the types $\Sigma_{x:X} Y(x)$ and $\Pi_{x:X} Y(x)$ live in the universe $\mathcal{U} \sqcup \mathcal{V}$; finally, if $X : \mathcal{U}$ and $x, y : X$, then $\mathrm{Id}_X(x, y) : \mathcal{U}$. The type of natural numbers N is assumed to be in $\mathcal{U}_0$ and we postulate that we have copies $\mathsf{0}_{\mathcal{U}}$ and $\mathsf{1}_{\mathcal{U}}$ in every universe $\mathcal{U}$. We assume function extensionality and propositional extensionality tacitly, and univalence explicitly when needed. Finally, we use a single higher inductive type: the propositional truncation of a type X is denoted by $\lVert X \rVert$ and we write $\exists_{x:X} Y(x)$ for $\lVert \sum_{x:X} Y(x) \rVert$.

2.1 The Notion of Size

We introduce the fundamental notion of a type having a certain size and specify the impredicativity axioms under consideration (Section 2.2). We also note the relation to excluded middle (Section 2.2) and univalence (Section 2.3). Finally in Section 2.4 we review embeddings and sections and establish our main technical result on size, namely that having a certain size is closed under retracts whose sections are embeddings.

▶ **Definition 1** (Size, `UF-Slice.html` in [16]). *A type X in a universe $\mathcal{U}$ is said to* have size $\mathcal{V}$ *if it is equivalent to a type in the universe $\mathcal{V}$. That is, X* has-size $\mathcal{V} :\equiv \sum_{Y:\mathcal{V}} (Y \simeq X)$.

2.2 Impredicativity and Excluded Middle

We consider various impredicativity axioms and their relation to (weak) excluded middle. The definitions and propositions below may be found in [15, Section 3.36], so proofs are omitted here.

▶ **Definition 2** (Impredicativity axioms).

(i) *By* Propositional-Resizing$_{\mathcal{U},\mathcal{V}}$ *we mean the assertion that every proposition P in a universe $\mathcal{U}$ has size $\mathcal{V}$.*

(ii) *The type of all propositions in a universe $\mathcal{U}$ is denoted by $\Omega_{\mathcal{U}}$. Observe that $\Omega_{\mathcal{U}} : \mathcal{U}^+$. We write* Ω-Resizing$_{\mathcal{U},\mathcal{V}}$ *for the assertion that the type $\Omega_{\mathcal{U}}$ has size $\mathcal{V}$.*

(iii) *The type of all $\neg\neg$-stable propositions in a universe $\mathcal{U}$ is denoted by $\Omega_{\mathcal{U}}^{\neg\neg}$, where a proposition P is* $\neg\neg$-stable *if $\neg\neg P$ implies P. By* $\Omega_{\neg\neg}$-Resizing$_{\mathcal{U},\mathcal{V}}$ *we mean the assertion that the type $\Omega_{\mathcal{U}}^{\neg\neg}$ has size $\mathcal{V}$.*

(iv) *For the particular case of a single universe, we write* Ω-Resizing$_{\mathcal{U}}$ *and* $\Omega_{\neg\neg}$-Resizing$_{\mathcal{U}}$ *for the respective assertions that $\Omega_{\mathcal{U}}$ has size $\mathcal{U}$ and $\Omega_{\mathcal{U}}^{\neg\neg}$ has size $\mathcal{U}$.*

▶ **Proposition 3.**

(i) *The principle* Ω-Resizing$_{\mathcal{U},\mathcal{V}}$ *implies* Propositional-Resizing$_{\mathcal{U},\mathcal{V}}$ *for every two universes $\mathcal{U}$ and $\mathcal{V}$.*

(ii) *The conjunction of* Propositional-Resizing$_{\mathcal{U},\mathcal{V}}$ *and* Propositional-Resizing$_{\mathcal{V},\mathcal{U}}$ *implies* Ω-Resizing$_{\mathcal{U},\mathcal{V}^+}$ *for every two universes $\mathcal{U}$ and $\mathcal{V}$.*

It is possible to define a weaker variation of propositional resizing for $\neg\neg$-stable propositions only (and derive similar connections), but we don't have any use for it in this paper.

▶ **Definition 4** ((Weak) excluded middle)**.**

(i) Excluded middle *in a universe $\mathcal{U}$ asserts that for every proposition P in $\mathcal{U}$ either P or $\neg P$ holds.*

(ii) Weak excluded middle *in a universe $\mathcal{U}$ asserts that for every proposition P in $\mathcal{U}$ either $\neg P$ or $\neg\neg P$ holds.*

We note that weak excluded middle says precisely that $\neg\neg$-stable propositions are decidable and is equivalent to de Morgan's Law.

▶ **Proposition 5.** *Excluded middle implies impredicativity. Specifically,*

(i) *Excluded middle in $\mathcal{U}$ implies* Ω-Resizing$_{\mathcal{U},\mathcal{U}_0}$.

(ii) *Weak excluded middle in $\mathcal{U}$ implies* $\Omega_{\neg\neg}$-Resizing$_{\mathcal{U},\mathcal{U}_0}$.

2.3 Size and Univalence

Assuming univalence we can prove that Propositional-Resizing$_{\mathcal{U},\mathcal{V}}$ and Ω-Resizing$_{\mathcal{U},\mathcal{V}}$ are subsingletons. More generally, univalence allows us to prove that the statement that X has size $\mathcal{V}$ is a proposition, which is needed in Section 3.5.

▶ **Proposition 6** (cf. `has-size-is-subsingleton` in [15])**.** *If $\mathcal{V}$ and $\mathcal{U} \sqcup \mathcal{V}$ are univalent universes, then X* has-size *$\mathcal{V}$ is a proposition for every $X : \mathcal{U}$.*

The converse also holds in the following form.

▶ **Proposition 7.** *The type X* has-size *$\mathcal{U}$ is a proposition for every $X : \mathcal{U}$ if and only if $\mathcal{U}$ is a univalent universe.*

Proof. Note that X has-size $\mathcal{U}$ is $\sum_{Y:\mathcal{U}} Y \simeq X$, so this can be found in [15, Section 3.14]. ◀

2.4 Size and Retracts

We show our main technical result on size here, namely that having a size is closed under retracts whose sections are embeddings.

▶ **Definition 8** (Sections, retractions and embeddings)**.**

(i) *A* section *is a map $s : X \to Y$ together with a left inverse $r : Y \to X$, i.e. the maps satisfy $r \circ s \sim \mathrm{id}$. We call r the* retraction *and say that X is a* retract *of Y.*

(ii) *A function $f : X \to Y$ is an embedding if the map $\mathrm{ap}_f : (x = y) \to (f(x) = f(y))$ is an equivalence for every $x, y : X$. (See [24, Definition 4.6.1(ii)].)*

(iii) *A* section-embedding *is a section $s : X \to Y$ that moreover is an embedding. We also say that X is an* embedded retract *of Y.*

We recall the following facts about embeddings and sections.

▶ **Lemma 9.**

(i) *A function $f : X \to Y$ is an* embedding *if and only if all its fibres are subsingletons, i.e.* $\prod_{y:Y}$ is-subsingleton$(\mathrm{fib}_f(y))$. *(See [24, Proof of Theorem 4.6.3].)*

(ii) *If every section is an embedding, then every type is a set. (See [22, Remark 3.11(2)].)*

(iii) *Sections to sets are embeddings. (See [15,* `lc-maps-into-sets-are-embeddings`*].)*

In phrasing our results it is helpful to extend the notion of size from types to functions.

▶ **Definition 10** (Size (for functions), `UF-Slice.html` in [16])**.** *A function $f : X \to Y$ is said to* have size $\mathcal{V}$ *if every fibre has size $\mathcal{V}$.*

▶ **Lemma 11** (cf. `UF-Slice.html` in [16])**.**

(i) *A type X has size $\mathcal{V}$ if and only if the unique map $X \to \mathbf{1}_{\mathcal{U}_0}$ has size $\mathcal{V}$.*

(ii) *If $f : X \to Y$ has size $\mathcal{V}$ and Y has size $\mathcal{V}$, then so does X.*

(iii) *If $s : X \to Y$ is a section-embedding and Y has size $\mathcal{V}$, then s has size $\mathcal{V}$ too, regardless of the size of X.*

Proof. The first two claims follow from the fact that for any map $f : X \to Y$ we have an equivalence $X \simeq \sum_{y:Y} \mathrm{fib}_f(y)$ (see [24, Lemma 4.8.2]). For the third claim, suppose that $s : X \to Y$ an embedding with retraction $r : Y \to X$. By the second part of the proof of Theorem 3.10 in [22], we have $\mathrm{fib}_s(y) \simeq \|s(r(y)) = y\|$, from which the claim follows. ◀

▶ **Lemma 12.**

(i) *If X is an embedded retract of Y and Y has size $\mathcal{V}$, then so does X.*

(ii) *If X is a retract of a set Y and Y has size $\mathcal{V}$, then so does X.*

Proof. The first statement follows from (ii) and (iii) of Lemma 11. The second follows from the first and item (iii) of Lemma 9. ◀

3 Large Posets Without Decidable Equality

We show that constructively and predicatively many structures from order theory (directed complete posets, bounded complete posets, sup-lattices) are necessarily large and necessarily lack decidable equality. We capture these structures by a technical notion of a $\delta_{\mathcal{V}}$-complete poset in Section 3.1. In Section 3.2 we define when such structures are nontrivial and introduce the constructively stronger notion of positivity. Section 3.3 and Section 3.4 contain the two fundamental technical lemmas and the main theorems, respectively. Finally, Section 3.5 considers alternative formulations of being nontrivial and positive that ensure that these notions are properties, as opposed to data and shows how the main theorems remain valid, assuming univalence.

3.1 $\delta_{\mathcal{V}}$-complete Posets

We start by introducing a class of weakly complete posets that we call $\delta_{\mathcal{V}}$-complete posets. The notion of a $\delta_{\mathcal{V}}$-complete poset is a technical and auxiliary notion sufficient to make our main theorems go through. The important point is that many familiar structures (dcpos, bounded complete posets, sup-lattices) are $\delta_{\mathcal{V}}$-complete posets (see Examples 15).

▶ **Definition 13** ($\delta_{\mathcal{V}}$-complete poset, $\delta_{x,y,P}$, $\bigvee \delta_{x,y,P}$). *A* poset *is a type X with a subsingleton-valued binary relation $\sqsubseteq$ on X that is reflexive, transitive and antisymmetric. It is not necessary to require X to be a set, as this follows from the other requirements. A poset $(X, \sqsubseteq)$ is* $\delta_{\mathcal{V}}$-complete *for a universe $\mathcal{V}$ if for every pair of elements $x, y : X$ with $x \sqsubseteq y$ and every subsingleton P in $\mathcal{V}$, the family*

$$\begin{aligned} \delta_{x,y,P} : 1 + P &\to X \\ \mathrm{inl}(\star) &\mapsto x; \\ \mathrm{inr}(p) &\mapsto y; \end{aligned}$$

has a supremum $\bigvee \delta_{x,y,P}$ in X.

▶ Remark 14 (Every poset is $\delta_{\mathcal{V}}$-complete, classically). Consider a poset $(X, \sqsubseteq)$ and a pair of elements $x \sqsubseteq y$. If $P : \mathcal{V}$ is a decidable proposition, then we can define the supremum of $\delta_{x,y,P}$ by case analysis on whether P holds or not. For if it holds, then the supremum is y, and if it does not, then the supremum is x. Hence, if excluded middle holds in $\mathcal{V}$, then the family $\delta_{x,y,P}$ has a supremum for every $P : \mathcal{V}$. Thus, if excluded middle holds in $\mathcal{V}$, then every poset (in any universe) is $\delta_{\mathcal{V}}$-complete.

The above remark naturally leads us to ask whether the converse also holds, i.e. if every poset is $\delta_{\mathcal{V}}$-complete, does excluded middle in $\mathcal{V}$ hold? As far as we know, we can only get weak excluded middle in $\mathcal{V}$, as we will later see in Proposition 18. This proposition also shows that in the absence of excluded middle, the notion of $\delta_{\mathcal{V}}$-completeness isn't trivial. For now, we focus on the fact that, also constructively and predicatively, there are many examples of $\delta_{\mathcal{V}}$-complete posets.

▶ **Examples 15.**

(i) *Every $\mathcal{V}$-sup-lattices is $\delta_{\mathcal{V}}$-complete. That is, if a poset X has suprema for all families $I \to X$ with I in the universe $\mathcal{V}$, then X is $\delta_{\mathcal{V}}$-complete.*

(ii) *The $\mathcal{V}$-sup-lattice $\Omega_{\mathcal{V}}$ is $\delta_{\mathcal{V}}$-complete. The type $\Omega_{\mathcal{V}}$ of propositions in $\mathcal{V}$ is a $\mathcal{V}$-sup-lattice with the order given by implication and suprema by existential quantification. Hence, $\Omega_{\mathcal{V}}$ is $\delta_{\mathcal{V}}$-complete. Specifically, given propositions Q, R and P, the supremum of $\delta_{Q,R,P}$ is given by $Q \vee (R \times P)$.*

(iii) *The $\mathcal{V}$-powerset $\mathcal{P}_{\mathcal{V}}(X) :\equiv X \to \Omega_{\mathcal{V}}$ of a type X is $\delta_{\mathcal{V}}$-complete. Note that $\mathcal{P}_{\mathcal{V}}(X)$ is another example of a $\mathcal{V}$-sup-lattice (ordered by subset inclusion and with suprema given by unions) and hence $\delta_{\mathcal{V}}$-complete.*

(iv) *Every $\mathcal{V}$-bounded complete posets is $\delta_{\mathcal{V}}$-complete. That is, if $(X, \sqsubseteq)$ is a poset with suprema for all bounded families $I \to X$ with I in the universe $\mathcal{V}$, then $(X, \sqsubseteq)$ is $\delta_{\mathcal{V}}$-complete. A family $\alpha : I \to X$ is bounded if there exists some $x : X$ with $\alpha(i) \sqsubseteq x$ for every $i : I$. For example, the family $\delta_{x,y,P}$ is bounded by y.*

(v) *Every $\mathcal{V}$-directed complete poset (dcpo) is $\delta_{\mathcal{V}}$-complete, since the family $\delta_{x,y,P}$ is directed. We note that [12] provides a host of examples of $\mathcal{V}$-dcpos.*

3.2 Nontrivial and Positive Posets

In Remark 14 we saw that if we can decide a proposition P, then we can define $\bigvee \delta_{x,y,P}$ by case analysis. What about the converse? That is, if $\delta_{x,y,P}$ has a supremum and we know that it equals x or y, can we then decide P? Of course, if $x = y$, then $\bigvee \delta_{x,y,P} = x = y$, so we don't learn anything about P. But what if add the assumption that $x \neq y$? It turns out that constructively we can only expect to derive decidability of $\neg P$ in that case. This is due to the fact that $x \neq y$ is a negated proposition, which is rather weak constructively, leading us to later define (see Definition 20) a constructively stronger notion for elements of $\delta_{\mathcal{V}}$-complete posets.

▶ **Definition 16** (Nontrivial). *A poset* $(X, \sqsubseteq)$ *is* nontrivial *if we have designated* $x, y : X$ *with* $x \sqsubseteq y$ *and* $x \neq y$.

▶ **Lemma 17.** *Let* $(X, \sqsubseteq, x, y)$ *be a nontrivial poset. We have the following implications for every proposition* $P : \mathcal{V}$*:*

(i) *if the supremum of* $\delta_{x,y,P}$ *exists and* $x = \bigvee \delta_{x,y,P}$*, then* $\neg P$ *is the case.*
(ii) *if the supremum of* $\delta_{x,y,P}$ *exists and* $y = \bigvee \delta_{x,y,P}$*, then* $\neg\neg P$ *is the case.*

Proof. Let $P : \mathcal{V}$ be an arbitrary proposition. For (i), suppose that $x = \bigvee \delta_{x,y,P}$ and assume for a contradiction that we have $p : P$. Then $y \equiv \delta_{x,y,P}(\mathrm{inr}(p)) \sqsubseteq \bigvee \delta_{x,y,P} = x$, which is impossible by antisymmetry and our assumptions that $x \sqsubseteq y$ and $x \neq y$. For (ii), suppose that $y = \bigvee \delta_{x,y,P}$ and assume for a contradiction that $\neg P$ holds. Then $x = \bigvee \delta_{x,y,P} = y$, contradicting our assumption that $x \neq y$. ◀

▶ **Proposition 18** (cf. Section 4 of [12]). *Let* 2 *be the poset with exactly two elements* $0 \sqsubseteq 1$. *If* 2 *is* $\delta_{\mathcal{V}}$*-complete, then weak excluded middle in* $\mathcal{V}$ *holds.*

Proof. Suppose that 2 were $\delta_{\mathcal{V}}$-complete and let $P : \mathcal{V}$ be an arbitrary subsingleton. We must show that $\neg P$ is decidable. Since 2 has exactly two elements, the supremum $\bigvee \delta_{0,1,P}$ must be 0 or 1. But then we apply Lemma 17 to get decidability of $\neg P$. ◀

That the conclusion of the implication in Lemma 17(ii) cannot be strengthened to say that P is the case is shown by the following observation.

▶ **Proposition 19.** *Recall Examples 15, which show that* $\Omega_{\mathcal{V}}$ *is* $\delta_{\mathcal{V}}$*-complete. If for every two propositions* Q *and* R *with* $Q \sqsubseteq R$ *and* $Q \neq R$ *we have that the equality* $R = \bigvee \delta_{Q,R,P}$ *in* $\Omega_{\mathcal{V}}$ *implies* P *for every proposition* $P : \mathcal{V}$*, then excluded middle in* $\mathcal{V}$ *follows.*

Proof. Assume the hypothesis in the proposition. We are going to show that $\neg\neg P \to P$ for every proposition $P : \mathcal{V}$, from which excluded middle in $\mathcal{V}$ holds. Let P be a proposition in $\mathcal{V}$ and assume that $\neg\neg P$. This yields $\mathsf{0} \neq P$, so by assumption the equality $P = \bigvee \delta_{\mathsf{0},P,P}$ implies P. But, recalling item (ii) of Examples 15, we have exactly this equality $\bigvee \delta_{\mathsf{0},P,P} = P$. ◀

We have seen that having a pair of elements x, y with $x \sqsubseteq y$ and $x \neq y$ is very weak constructively. As promised in the introduction of this section, we now introduce a constructively stronger notion.

▶ **Definition 20** (Strictly below, $x \sqsubset y$). *Let* $(X, \sqsubseteq)$ *be a* $\delta_{\mathcal{V}}$*-complete poset and* $x, y : X$. *We say that* x *is* strictly below y *if* $x \sqsubseteq y$ *and, moreover, for every* $z \sqsupseteq y$ *and every proposition* $P : \mathcal{V}$*, the equality* $z = \bigvee \delta_{x,z,P}$ *implies* P.

Note that with excluded middle, $x \sqsubset y$ is equivalent to the conjunction of $x \sqsubseteq y$ and $x \neq y$. But constructively, the former is much stronger, as the following example and proposition illustrate.

▶ **Example 21** (Strictly below in $\Omega_{\mathcal{V}}$). Recall from Examples 15 that $\Omega_{\mathcal{V}}$ is $\delta_{\mathcal{V}}$-complete. Let $P : \mathcal{V}$ be an arbitrary proposition. Observe that $\mathsf{0}_{\mathcal{V}} \neq P$ precisely when $\neg\neg P$ holds. However, $\mathsf{0}_{\mathcal{V}}$ is strictly below P if and only if P holds.

▶ **Proposition 22.** *For a* $\delta_{\mathcal{V}}$*-complete poset* $(X, \sqsubseteq)$ *and* $x, y : X$*, we have that* $x \sqsubset y$ *implies both* $x \sqsubseteq y$ *and* $x \neq y$. *However, if the conjunction of* $x \sqsubseteq y$ *and* $x \neq y$ *implies* $x \sqsubset y$ *for every* $x, y : \Omega_{\mathcal{V}}$*, then excluded middle in* $\mathcal{V}$ *holds.*

Proof. Note that $x \sqsubset y$ implies $x \sqsubseteq y$ by definition. Now suppose that $x \sqsubset y$ and assume $x = y$ for a contradiction. Since we assumed $x \sqsubset y$, the equality $y = \bigvee \delta_{x,y,\mathsf{0}_{\mathcal{V}}}$ implies that $\mathsf{0}_{\mathcal{V}}$ holds. But this equality holds since $x = y$ by our other assumption, so $x \neq y$, as desired.

For $P : \Omega_{\mathcal{V}}$ we observed that $\mathsf{0}_{\mathcal{V}} \neq P$ is equivalent to $\neg\neg P$ and that $\mathsf{0}_{\mathcal{V}} \sqsubset P$ is equivalent to P, so if we had $((x \sqsubseteq y) \times (x \neq y)) \to x \sqsubset y$ in general, then we would have $\neg\neg P \to P$ for every proposition P in $\mathcal{V}$, which is equivalent to excluded middle in $\mathcal{V}$. ◀

▶ **Lemma 23.** *Let $(X, \sqsubseteq)$ be a $\delta_{\mathcal{V}}$-complete poset and $x, y, z : X$. The following hold:*
- **(i)** *If $x \sqsubseteq y \sqsubset z$, then $x \sqsubset z$.*
- **(ii)** *If $x \sqsubset y \sqsubseteq z$, then $x \sqsubset z$.*

Proof. For (i), assume $x \sqsubseteq y \sqsubset z$, let P be an arbitrary proposition in $\mathcal{V}$ and suppose that $z \sqsubseteq w$. We must show that $w = \bigvee \delta_{x,w,P}$ implies P. But $y \sqsubset z$, so we know that the equality $w = \bigvee \delta_{y,w,P}$ implies P. Now observe that $\bigvee \delta_{x,w,P} \sqsubseteq \bigvee \delta_{y,w,P}$, so if $w = \bigvee \delta_{x,w,P}$, then $w = \bigvee \delta_{y,w,P}$, finishing the proof. For (ii), assume $x \sqsubset y \sqsubseteq z$, let P be an arbitrary proposition in $\mathcal{V}$ and suppose that $z \sqsubseteq w$. We must show that $w = \bigvee \delta_{x,w,P}$ implies P. But $x \sqsubset y$ and $y \sqsubseteq w$, so this follows immediately. ◀

▶ **Proposition 24.** *Let $(X, \sqsubseteq)$ be a $\mathcal{V}$-sup-lattice and let $y : X$. The following are equivalent:*
- **(i)** *the least element of X is strictly below y;*
- **(ii)** *for every family $\alpha : I \to X$ with $I : \mathcal{V}$ and $y \sqsubseteq \bigvee \alpha$, there exists some element $i : I$.*
- **(iii)** *there exists some $x : X$ with $x \sqsubset y$.*

Proof. Write $\perp$ for the least element of X. By Lemma 23 we have:

$$\perp \sqsubset y \iff \exists_{x:X}(\perp \sqsubseteq x \sqsubset y) \iff \exists_{x:X}(x \sqsubset y),$$

which proves the equivalence of (i) and (iii). It remains to prove that (i) and (ii) are equivalent. Suppose that $\perp \sqsubset y$ and let $\alpha : I \to X$ with $y \sqsubseteq \bigvee \alpha$. Using $\perp \sqsubset y \sqsubseteq \bigvee \alpha$ and Lemma 23, we have $\perp \sqsubset \bigvee \alpha$. Hence, we only need to prove $\bigvee \alpha \sqsubseteq \bigvee \delta_{\perp,\bigvee \alpha,\exists i:I}$, but $\alpha_j \sqsubseteq \bigvee \delta_{\perp,\bigvee \alpha,\exists i:I}$ for every $j : I$, so this is true indeed. For the converse, assume that y satisfies (ii), suppose $z \sqsupseteq y$ and let $P : \mathcal{V}$ be a proposition such that $z = \bigvee \delta_{\perp,z,P}$. We must show that P holds. But notice that $y \sqsubseteq z = \bigvee \delta_{\perp,z,P} = \bigvee((p : P) \mapsto z)$, so P must be inhabited as y satisfies (ii). ◀

Item (ii) in Proposition 24 says exactly that y is a positive element in the sense of [19, p. 98]. We note that item (iii) in Proposition 24 makes sense even when $(X, \sqsubseteq)$ is not a $\mathcal{V}$-sup-lattice, but just a $\delta_{\mathcal{V}}$-complete poset. Accordingly, we make the following definition.

▶ **Definition 25** (Positive element)**.** *An element of a $\delta_{\mathcal{V}}$-complete poset is* positive *if it satisfies item (iii) in Proposition 24.*

An element of a $\mathcal{V}$-dcpo is called *compact* if it is inaccessible by directed joins of families indexed by types in $\mathcal{V}$ [12, Definition 44].

▶ **Proposition 26.** *A compact element x of a $\mathcal{V}$-dcpo with least element $\perp$ is positive if and only if $x \neq \perp$.*

Proof. One implication is taken care of by Proposition 22. For the converse, suppose that $x \neq \perp$. We show that $\perp$ is strictly below x. For if $x \sqsubseteq y = \bigvee \delta_{\perp,y,P}$, then by compactness of x, there must exist $i : 1 + P$ such that $x \sqsubseteq \delta_{\perp,y,P}(i)$ already. But i can't be equal to $\mathrm{inl}(\star)$, since x is assumed to be different from $\perp$. Hence, $i = \mathrm{inr}(p)$ and P must hold. ◀

Looking to strengthen the notion of a nontrivial poset, we make the following definition, whose terminology is inspired by Definition 25.

▶ **Definition 27** (Positive poset). *A $\delta_{\mathcal{V}}$-complete poset X is* positive *if we have designated $x, y : X$ with x strictly below y.*

▶ **Examples 28.**

(i) *Consider an element P of the $\delta_{\mathcal{V}}$-complete poset $\Omega_{\mathcal{V}}$. The pair $(0_{\mathcal{V}}, P)$ witnesses nontriviality of $\Omega_{\mathcal{V}}$ if and only if $\neg\neg P$ holds, while it witnesses positivity if and only if P holds.*

(ii) *Consider the $\mathcal{V}$-powerset $\mathcal{P}_{\mathcal{V}}(X)$ on a type X as a $\delta_{\mathcal{V}}$-complete poset (recall Examples 15). We write $\emptyset : \mathcal{P}_{\mathcal{V}}(X)$ for the map $x \mapsto 0_{\mathcal{V}}$. Say that a subset $A : \mathcal{P}_{\mathcal{V}}(X)$ is nonempty if $A \neq \emptyset$ and inhabited if there exists some $x : X$ such that $A(x)$ holds. The pair $(\emptyset, A)$ witnesses nontriviality of $\mathcal{P}_{\mathcal{V}}(X)$ if and only if A is nonempty, while it witnesses positivity if and only if A is inhabited. In particular, $\mathcal{P}_{\mathcal{V}}(X)$ is positive if and only if X is an inhabited type.*

3.3 Retract Lemmas

We show that the type of propositions in $\mathcal{V}$ is a retract of any positive $\delta_{\mathcal{V}}$-complete poset and that the type of $\neg\neg$-stable propositions in $\mathcal{V}$ is a retract of any nontrivial $\delta_{\mathcal{V}}$-complete poset.

▶ **Definition 29** ($\Delta_{x,y} : \Omega_{\mathcal{V}} \to X$). *Suppose that $(X, \sqsubseteq, x, y)$ is a nontrivial $\delta_{\mathcal{V}}$-complete poset. We define $\Delta_{x,y} : \Omega_{\mathcal{V}} \to X$ by the assignment $P \mapsto \bigvee \delta_{x,y,P}$.*

We will often omit the subscripts in $\Delta_{x,y}$ when it is clear from the context.

▶ **Definition 30** (Locally small). *A $\delta_{\mathcal{V}}$-complete poset $(X, \sqsubseteq)$ is* locally small *if its order has values of size $\mathcal{V}$, i.e. we have $\sqsubseteq_{\mathcal{V}} : X \to X \to \mathcal{V}$ with $(x \sqsubseteq y) \simeq (x \sqsubseteq_{\mathcal{V}} y)$ for every $x, y : X$.*

▶ **Examples 31.**

(i) *The $\mathcal{V}$-sup-lattices $\Omega_{\mathcal{V}}$ and $\mathcal{P}_{\mathcal{V}}(X)$ (for $X : \mathcal{V}$) are locally small.*

(ii) *All examples of $\mathcal{V}$-dcpos in [12] are locally small.*

▶ **Lemma 32.** *A locally small $\delta_{\mathcal{V}}$-complete poset $(X, \sqsubseteq)$ is nontrivial, witnessed by elements $x \sqsubseteq y$, if and only if the composite $\Omega_{\mathcal{V}}^{\neg\neg} \hookrightarrow \Omega_{\mathcal{V}} \xrightarrow{\Delta_{x,y}} X$ is a section.*

Proof. Suppose first that $(X, \sqsubseteq, x, y)$ is nontrivial and locally small. We define

$$\begin{aligned} r &: X \to \Omega_{\mathcal{V}}^{\neg\neg} \\ &z \mapsto z \not\sqsubseteq_{\mathcal{V}} x. \end{aligned}$$

Note that negated propositions are $\neg\neg$-stable, so r is well-defined. Let $P : \mathcal{V}$ be an arbitrary $\neg\neg$-stable proposition. We want to show that $r(\Delta_{x,y}(P)) = P$. By propositional extensionality, establishing logical equivalence suffices. Suppose first that P holds. Then $\Delta_{x,y}(P) \equiv \bigvee \delta_{x,y,P} = y$, so $r(\Delta_{x,y}(P)) = r(y) \equiv (y \not\sqsubseteq_{\mathcal{V}} x)$ holds by antisymmetry and our assumptions that $x \sqsubseteq y$ and $x \neq y$. Conversely, assume that $r(\Delta_{x,y}(P))$ holds, i.e. that we have $\bigvee \delta_{x,y,P} \not\sqsubseteq_{\mathcal{V}} x$. Since P is $\neg\neg$-stable, it suffices to derive a contradiction from $\neg P$. So assume $\neg P$. Then $x = \bigvee \delta_{x,y,P}$, so $r(\Delta_{x,y}(P)) = r(x) \equiv x \not\sqsubseteq_{\mathcal{V}} x$, which is false by reflexivity.

For the converse, assume that $\Omega_{\mathcal{V}}^{\neg\neg} \hookrightarrow \Omega_{\mathcal{V}} \xrightarrow{\Delta_{x,y}} X$ has a retraction $r : \Omega_{\mathcal{V}}^{\neg\neg} \to X$. Then $0_{\mathcal{V}} = r(\Delta_{x,y}(0_{\mathcal{V}})) = r(x)$ and $1_{\mathcal{V}} = r(\Delta_{x,y}(1_{\mathcal{V}})) = r(y)$, where we used that $0_{\mathcal{V}}$ and $1_{\mathcal{V}}$ are $\neg\neg$-stable. Since $0_{\mathcal{V}} \neq 1_{\mathcal{V}}$, we get $x \neq y$, so $(X, \sqsubseteq, x, y)$ is nontrivial, as desired. ◀

The appearance of the double negation in the above lemma is due to the definition of nontriviality. If we instead assume a positive poset X, then we can exhibit all of $\Omega_{\mathcal{V}}$ as a retract of X.

▶ **Lemma 33.** *A locally small $\delta_{\mathcal{V}}$-complete poset $(X, \sqsubseteq)$ is positive, witnessed by elements $x \sqsubseteq y$, if and only if for every $z \sqsupseteq y$, the map $\Delta_{x,z} : \Omega_{\mathcal{V}} \to X$ is a section.*

Proof. Suppose first that $(X, \sqsubseteq, x, y)$ is positive and locally small and let $z \sqsupseteq y$ be arbitrary. We define

$$\begin{aligned} r_z : X &\mapsto \Omega_{\mathcal{V}} \\ w &\mapsto z \sqsubseteq_{\mathcal{V}} w. \end{aligned}$$

Let $P : \mathcal{V}$ be arbitrary proposition. We want to show that $r_z(\Delta_{x,z}(P)) = P$. Because of propositional extensionality, it suffices to establish a logical equivalence between P and $r_z(\Delta_{x,z}(P))$. Suppose first that P holds. Then $\Delta_{x,z}(P) = z$, so $r_z(\Delta_{x,z}(P)) = r_z(z) \equiv (z \sqsubseteq_{\mathcal{V}} z)$ holds as well by reflexivity. Conversely, assume that $r_z(\Delta_{x,z}(P))$ holds, i.e. that we have $z \sqsubseteq_{\mathcal{V}} \bigvee \delta_{x,z,P}$. Since $\bigvee \delta_{x,z,P} \sqsubseteq z$ always holds, we get $z = \bigvee \delta_{x,z,P}$ by antisymmetry. But by assumption and Lemma 23, the element x is strictly below z, so P must hold.

For the converse, assume that for every $z \sqsupseteq y$, the map $\Delta_{x,z} : \Omega_{\mathcal{V}} \to X$ has a retraction $r_z : X \to \Omega_{\mathcal{V}}$. We must show that the equality $z = \Delta_{x,z}(P)$ implies P for every $z \sqsupseteq y$ and proposition $P : \mathcal{V}$. Assuming $z = \Delta_{x,z}(P)$, we have $1_{\mathcal{V}} = r_z(\Delta_{x,z}(1_{\mathcal{V}})) = r_z(z) = r_z(\Delta_{x,z}(P)) = P$, so P must hold indeed. Hence, $(X, \sqsubseteq, x, y)$ is positive, as desired. ◀

3.4 Reductions to Impredicativity and Excluded Middle

We present our main theorems here, which show that, constructively and predicatively, nontrivial $\delta_{\mathcal{V}}$-complete posets are necessarily large and necessarily lack decidable equality.

▶ **Definition 34** (Small). *A $\delta_{\mathcal{V}}$-complete poset is* small *if it is locally small and its carrier has size $\mathcal{V}$.*

▶ **Theorem 35.**
(i) *There is a nontrivial small $\delta_{\mathcal{V}}$-complete poset if and only if $\Omega_{\neg\neg}$-Resizing$_{\mathcal{V}}$ holds.*
(ii) *There is a positive small $\delta_{\mathcal{V}}$-complete poset if and only if Ω-Resizing$_{\mathcal{V}}$ holds.*

Proof. (i) Suppose that $(X, \sqsubseteq, x, y)$ is a nontrivial small $\delta_{\mathcal{V}}$-complete poset. By Lemma 32, we can exhibit $\Omega_{\mathcal{V}}^{\neg\neg}$ as a retract of X. But X has size $\mathcal{V}$ by assumption, so by Lemma 12 and the fact that $\Omega_{\mathcal{V}}^{\neg\neg}$ is a set, the type $\Omega_{\mathcal{V}}^{\neg\neg}$ has size $\mathcal{V}$ as well. For the converse, note that $(\Omega_{\mathcal{V}}^{\neg\neg}, \to, 0_{\mathcal{V}}, 1_{\mathcal{V}})$ is a nontrivial $\mathcal{V}$-sup-lattice with $\bigvee \alpha$ given by $\neg\neg\exists_{i:I}\alpha_i$. And if we assume $\Omega_{\neg\neg}$-Resizing$_{\mathcal{V}}$, then it is small.

(ii) Suppose that $(X, \sqsubseteq, x, y)$ is a positive small poset. By Lemma 33, we can exhibit $\Omega_{\mathcal{V}}$ as a retract of X. But X has size $\mathcal{V}$ by assumption, so by Lemma 12 and the fact that $\Omega_{\mathcal{V}}$ is a set, the type $\Omega_{\mathcal{V}}$ has size $\mathcal{V}$ as well. For the converse, note that $(\Omega_{\mathcal{V}}, \to, 0_{\mathcal{V}}, 1_{\mathcal{V}})$ is a positive $\mathcal{V}$-sup-lattice. And if we assume Ω-Resizing$_{\mathcal{V}}$, then it is small. ◀

▶ **Lemma 36** (`retract-is-discrete` and `subtype-is-¬¬-separated` in [16]).
(i) *Types with decidable equality are closed under retracts.*
(ii) *Types with $\neg\neg$-stable equality are closed under retracts.*

▶ **Theorem 37.** *There is a nontrivial locally small $\delta_{\mathcal{V}}$-complete poset with decidable equality if and only if weak excluded middle in $\mathcal{V}$ holds.*

Proof. Suppose that $(X, \sqsubseteq, x, y)$ is a nontrivial locally small $\delta_{\mathcal{V}}$-complete poset with decidable equality. Then by Lemmas 32 and 36, the type $\Omega_{\mathcal{V}}^{\neg\neg}$ must have decidable equality too. But negated propositions are $\neg\neg$-stable, so this yields weak excluded middle in $\mathcal{V}$. For the converse, note that $(\Omega_{\mathcal{V}}^{\neg\neg}, \to, 0_{\mathcal{V}}, 1_{\mathcal{V}})$ is a nontrivial $\mathcal{V}$-sup-lattice that has decidable equality if and only if weak excluded middle in $\mathcal{V}$ holds. ◀

▶ **Theorem 38.** *The following are equivalent:*

(i) *There is a positive locally small $\delta_{\mathcal{V}}$-complete poset with $\neg\neg$-stable equality.*
(ii) *There is a positive locally small $\delta_{\mathcal{V}}$-complete poset with decidable equality.*
(iii) *Excluded middle in $\mathcal{V}$ holds.*

Proof. Note that (ii) $\Rightarrow$ (i), so we are left to show that (iii) $\Rightarrow$ (ii) and that (i) $\Rightarrow$ (iii). For the first implication, note that $(\Omega_{\mathcal{V}}, \to, 0_{\mathcal{V}}, 1_{\mathcal{V}})$ is a positive $\mathcal{V}$-sup-lattice that has decidable equality if and only if excluded middle in $\mathcal{V}$ holds. To see that (i) implies (iii), suppose that $(X, \sqsubseteq, x, y)$ is a positive locally small $\delta_{\mathcal{V}}$-complete poset with $\neg\neg$-stable equality. Then by Lemmas 33 and 36 the type $\Omega_{\mathcal{V}}$ must have $\neg\neg$-stable equality. But this implies that $\neg\neg P \to P$ for every proposition P in $\mathcal{V}$ which is equivalent to excluded middle in $\mathcal{V}$. ◀

Lattices, bounded complete posets and dcpos are necessarily large and necessarily lack decidable equality in our predicative constructive setting. More precisely,

▶ **Corollary 39.**

(i) *There is a nontrivial small $\mathcal{V}$-sup-lattice (or $\mathcal{V}$-bounded complete poset or $\mathcal{V}$-dcpo) if and only if $\Omega_{\neg\neg}$-Resizing$_{\mathcal{V}}$ holds.*
(ii) *There is a positive small $\mathcal{V}$-sup-lattice (or $\mathcal{V}$-bounded complete poset or $\mathcal{V}$-dcpo) if and only if Ω-Resizing$_{\mathcal{V}}$ holds.*
(iii) *There is a nontrivial locally small $\mathcal{V}$-sup-lattice (or $\mathcal{V}$-bounded complete poset or $\mathcal{V}$-dcpo) with decidable equality if and only if weak excluded middle in $\mathcal{V}$ holds.*
(iv) *There is a positive locally small $\mathcal{V}$-sup-lattice (or $\mathcal{V}$-bounded complete poset or $\mathcal{V}$-dcpo) with decidable equality if and only if excluded middle in $\mathcal{V}$ holds.*

3.5 Unspecified Nontriviality and Positivity

The above notions of non-triviality and positivity are data rather than property. Indeed, a nontrivial poset $(X, \sqsubseteq)$ is (by definition) equipped with two designated points $x, y : X$ such that $x \sqsubseteq y$ and $x \neq y$. It is natural to wonder if the propositionally truncated versions of these two notions yield the same conclusions. In this section we show that this is indeed the case if we assume univalence. The need for the univalence assumption comes from the fact that the notion of having a given size is property precisely if univalence holds, as shown in Propositions 6 and 7.

▶ **Definition 40** (Nontrivial/positive in an unspecified way). *A poset $(X, \sqsubseteq)$ is* nontrivial in an unspecified way *if there exist some elements $x, y : X$ such that $x \sqsubseteq y$ and $x \neq y$, i.e. $\exists_{x:X}\exists_{y:X}((x \sqsubseteq y) \times (x \neq y))$. Similarly, we can define when a poset is* positive in an unspecified way *by truncating the notion of positivity.*

▶ **Theorem 41.** *Suppose that the universes $\mathcal{V}$ and $\mathcal{V}^+$ are univalent.*

(i) *There is a small $\delta_{\mathcal{V}}$-complete poset that is nontrivial in an unspecified way if and only if $\Omega_{\neg\neg}$-Resizing$_{\mathcal{V}}$ holds.*
(ii) *There is a small $\delta_{\mathcal{V}}$-complete poset that is positive in an unspecified way if and only if Ω-Resizing$_{\mathcal{V}}$ holds.*

Proof. (i) Suppose that $(X, \sqsubseteq)$ is a $\delta_{\mathcal{V}}$-complete poset that is nontrivial in an unspecified way. By Proposition 6 and univalence of $\mathcal{V}$ and $\mathcal{V}^+$, type $\Omega_{\mathcal{V}}^{\neg\neg}$ has-size $\mathcal{V}$ is a proposition. By the universal property of the propositional truncation, in proving that $\Omega_{\mathcal{V}}^{\neg\neg}$ has-size $\mathcal{V}$ we can therefore assume that are given points $x, y : X$ with $x \sqsubseteq y$ and $x \neq y$. The result then follows from Theorem 35. (ii) By reduction to item (ii) of Theorem 35. ◀

Similarly, we can prove the following theorems by reduction to Theorems 37 and 38.

▶ **Theorem 42.**

(i) *There is a locally small $\delta_{\mathcal{V}}$-complete poset with decidable equality that is nontrivial in an unspecified way if and only if weak excluded middle in $\mathcal{V}$ holds.*

(ii) *There is a locally small $\delta_{\mathcal{V}}$-complete poset with decidable equality that is positive in an unspecified way if and only if excluded middle in $\mathcal{V}$ holds.*

4 Maximal Points and Fixed Points

In this section we construct a particular example of a $\mathcal{V}$-sup-lattice that will prove very useful in studying the predicative validity of some well-known principles in order theory.

▶ **Definition 43** (Lifting, cf. [14])**.** *Fix a proposition $P_{\mathcal{U}}$ in a universe $\mathcal{U}$. Lifting $P_{\mathcal{U}}$ with respect to a universe $\mathcal{V}$ is defined by*

$$\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}}) :\equiv \sum_{Q:\Omega_{\mathcal{V}}} (Q \to P_{\mathcal{U}}).$$

This is a subtype of $\Omega_{\mathcal{V}}$ and it is closed under $\mathcal{V}$-suprema (in particular, it contains the least element).

▶ **Examples 44.**

(i) *If $P_{\mathcal{U}} :\equiv \mathbb{0}_{\mathcal{U}}$, then $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}}) \simeq \left(\sum_{Q:\Omega_{\mathcal{V}}} \neg Q\right) \simeq \left(\sum_{Q:\Omega_{\mathcal{V}}} Q = \mathbb{0}_{\mathcal{V}}\right) \simeq \mathbb{1}$.*

(ii) *If $P_{\mathcal{U}} :\equiv \mathbb{1}_{\mathcal{U}}$, then $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}}) \equiv \left(\sum_{Q:\Omega_{\mathcal{V}}} (Q \to \mathbb{1}_{\mathcal{U}})\right) \simeq \Omega_{\mathcal{V}}$.*

What makes $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}})$ useful is the following observation.

▶ **Lemma 45.** *Suppose that the poset $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}})$ has a maximal element $Q : \Omega_{\mathcal{V}}$. Then $P_{\mathcal{U}}$ is equivalent to Q, which is the greatest element of $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}})$. In particular, $P_{\mathcal{U}}$ has size $\mathcal{V}$. Conversely, if $P_{\mathcal{U}}$ is equivalent to a proposition $Q : \Omega_{\mathcal{V}}$, then Q is the greatest element of $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}})$.*

Proof. Suppose that $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}})$ has a maximal element $Q : \Omega_{\mathcal{V}}$. We wish to show that $Q \simeq P_{\mathcal{U}}$. By definition of $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}})$, we already have that $Q \to P_{\mathcal{U}}$. So only the converse remains. Therefore suppose that $P_{\mathcal{U}}$ holds. Then, $\mathbb{1}_{\mathcal{V}}$ is an element of $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}})$. Obviously $Q \to \mathbb{1}_{\mathcal{V}}$, but Q is maximal, so actually $Q = \mathbb{1}_{\mathcal{V}}$, that is, Q holds, as desired. Thus, $Q \simeq P_{\mathcal{U}}$. It is then straightforward to see that Q is actually the greatest element of $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}})$, since $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}}) \simeq \sum_{Q':\Omega_{\mathcal{V}}}(Q' \to Q)$. For the converse, assume that $P_{\mathcal{U}}$ is equivalent to a proposition $Q : \Omega_{\mathcal{V}}$. Then, as before, $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}}) \simeq \sum_{Q':\Omega_{\mathcal{V}}}(Q' \to Q)$, which shows that Q is indeed the greatest element of $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}})$. ◀

▶ **Corollary 46.** *Let $P_{\mathcal{U}}$ be a proposition in $\mathcal{U}$. The $\mathcal{V}$-sup-lattice $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}})$ has all $\mathcal{V}$-infima if and only if $P_{\mathcal{U}}$ has size $\mathcal{V}$.*

Proof. Suppose first that $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}})$ has all $\mathcal{V}$-infima. Then it must have a infimum for the empty family $\mathbb{0}_{\mathcal{V}} \to \mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}})$. But this infimum must be the greatest element of $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}})$. So by Lemma 45 the proposition $P_{\mathcal{U}}$ must have size $\mathcal{V}$.

Conversely, suppose that $P_{\mathcal{U}}$ is equivalent to a proposition $Q : \mathcal{V}$. Then the infimum of a family $\alpha : I \to \mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}})$ with $I : \mathcal{V}$ is given by $(Q \times \Pi_{i:I}\alpha_i) : \mathcal{V}$. ◀

▶ **Definition 47** (Zorn's-Lemma$_{\mathcal{V},\mathcal{U},\mathcal{T}}$)**.** *Let $\mathcal{U}$, $\mathcal{V}$ and $\mathcal{T}$ be universes.* Zorn's-Lemma$_{\mathcal{V},\mathcal{U},\mathcal{T}}$ *asserts that every pointed $\mathcal{V}$-dcpo with carrier in $\mathcal{U}$ and order taking values in $\mathcal{T}$ (cf. [12]) has a maximal element.*

It important to note that Zorn's lemma does *not* imply the Axiom of Choice in the absence of excluded middle [3]. If it did, then the following would be useless, since the Axiom of Choice implies excluded middle, which in turn implies propositional resizing.

▶ **Theorem 48.** Zorn's-Lemma$_{\mathcal{V},\mathcal{V}^+\sqcup\mathcal{U},\mathcal{V}}$ *implies* Propositional-Resizing$_{\mathcal{U},\mathcal{V}}$.

In particular, Zorn's-Lemma$_{\mathcal{V},\mathcal{V}^+,\mathcal{V}}$ implies Propositional-Resizing$_{\mathcal{V}^+,\mathcal{V}}$.

Proof. Suppose that Zorn's-Lemma$_{\mathcal{V},\mathcal{V}^+\sqcup\mathcal{U},\mathcal{V}}$ were true. Then $\mathcal{L}_\mathcal{V}(P) : \mathcal{V}^+ \sqcup \mathcal{U}$ has a maximal element for every $P : \Omega_\mathcal{U}$. Hence, by Lemma 45, every $P : \Omega_\mathcal{U}$ has size $\mathcal{V}$. ◀

We can also use Lemma 45 to show that the following version of Tarski's fixed point theorem [23] is not available predicatively.

▶ **Definition 49** (Tarski's-Theorem$_{\mathcal{V},\mathcal{U},\mathcal{T}}$)**.** *The assertion* Tarski's-Theorem$_{\mathcal{V},\mathcal{U},\mathcal{T}}$ *says that every monotone endofunction on a $\mathcal{V}$-sup-lattice with carrier in a universe $\mathcal{U}$ and order taking values in a universe $\mathcal{T}$ has a greatest fixed point.*

▶ **Theorem 50.** Tarski's-Theorem$_{\mathcal{V},\mathcal{V}^+\sqcup\mathcal{U},\mathcal{V}}$ *implies* Propositional-Resizing$_{\mathcal{U},\mathcal{V}}$.

In particular, Tarski's-Theorem$_{\mathcal{V},\mathcal{V}^+,\mathcal{V}}$ implies Propositional-Resizing$_{\mathcal{V}^+,\mathcal{V}}$.

Proof. Suppose that Tarski's-Theorem$_{\mathcal{V},\mathcal{V}^+\sqcup\mathcal{U},\mathcal{V}}$ were true and let $P : \Omega_\mathcal{U}$ be arbitrary. Consider the $\mathcal{V}$-sup-lattice $\mathcal{L}_\mathcal{V}(P) : \mathcal{V}^+ \sqcup \mathcal{U}$. By assumption, the identity map on this poset has a greatest fixed point, but this must be the greatest element of $\mathcal{L}_\mathcal{V}(P)$, which implies that P has size $\mathcal{V}$ by Lemma 45. ◀

Another famous fixed point theorem, for dcpos this time, is due to Pataraia [20, 13] which says that every monotone endofunction on a pointed dcpo has a least fixed point. (A dcpo is called pointed if it has a least element.) A crucial step in proving Pataraia's theorem is the observation that every dcpo has a greatest monotone inflationary endofunction. (An endomap $f : X \to X$ is inflationary when $x \sqsubseteq f(x)$ for every $x : X$.) We refer to this intermediate result as Pataraia's lemma.

▶ **Definition 51** (Pataraia's-Lemma$_{\mathcal{V},\mathcal{U},\mathcal{T}}$, Pataraia's-Theorem$_{\mathcal{V},\mathcal{U},\mathcal{T}}$)**.**
- **(i)** Pataraia's-Theorem$_{\mathcal{V},\mathcal{U},\mathcal{T}}$ *says that every monotone endofunction on a pointed $\mathcal{V}$-dcpo with carrier in a universe $\mathcal{U}$ and order taking values in a universe $\mathcal{T}$ has a least fixed point.*
- **(ii)** Pataraia's-Lemma$_{\mathcal{V},\mathcal{U},\mathcal{T}}$ *says that every $\mathcal{V}$-dcpo with carrier in a universe $\mathcal{U}$ and order taking values in a universe $\mathcal{T}$ has a greatest monotone inflationary endofunction.*

A careful analysis of the proof in [13, Section 2] shows that in our predicative setting we can still prove that Pataraia's-Lemma$_{\mathcal{V},\mathcal{U}\sqcup\mathcal{T},\mathcal{U}\sqcup\mathcal{T}}$ implies Pataraia's-Theorem$_{\mathcal{V},\mathcal{U},\mathcal{T}}$. However, Pataraia's lemma is not available predicatively.

▶ **Theorem 52.** Pataraia's-Lemma$_{\mathcal{V},\mathcal{V}^+\sqcup\mathcal{U},\mathcal{V}}$ *implies* Propositional-Resizing$_{\mathcal{U},\mathcal{V}}$.

In particular, Pataraia's-Lemma$_{\mathcal{V},\mathcal{V}^+,\mathcal{V}}$ implies Propositional-Resizing$_{\mathcal{V}^+,\mathcal{V}}$.

Proof. Suppose that Pataraia's-Lemma$_{\mathcal{V},\mathcal{V}^+\sqcup\mathcal{U},\mathcal{V}}$ were true and let $P : \Omega_\mathcal{U}$ be arbitrary. Consider the $\mathcal{V}$-dcpo $\mathcal{L}_\mathcal{V}(P) : \mathcal{V}^+ \sqcup \mathcal{U}$. By assumption, it has a greatest monotone inflationary endomap $g : \mathcal{L}_\mathcal{V}(P) \to \mathcal{L}_\mathcal{V}(P)$. We claim that $g(0_\mathcal{V})$ is a maximal element of $\mathcal{L}_\mathcal{V}(P)$, which would finish the proof by Lemma 45. So suppose that we have $Q : \mathcal{L}_\mathcal{V}(P)$ with $g(0_\mathcal{V}) \sqsubseteq Q$. Then we must show that $Q \sqsubseteq g(0_\mathcal{V})$. Define $f_Q : \mathcal{L}_\mathcal{V}(P) \to \mathcal{L}_\mathcal{V}(P)$ by $Q' \mapsto Q' \vee Q$. Note that f_Q is monotone and inflationary, so that $f_Q \sqsubseteq g$. Hence, $Q = f_Q(0_\mathcal{V}) \sqsubseteq g(0_\mathcal{V})$, as desired. ◀

▶ Remark 53. For a *single* universe $\mathcal{U}$, the usual proofs (see resp. [23] and [13, Section 2]) of Tarski's-Theorem$_{\mathcal{U},\mathcal{U},\mathcal{U}}$, Pataraia's-Lemma$_{\mathcal{U},\mathcal{U},\mathcal{U}}$ and (hence) Pataraia's-Theorem$_{\mathcal{U},\mathcal{U},\mathcal{U}}$ are also valid in our predicative setting. However, in light of Theorem 35, these statements are not useful predicatively, because one would never be able to find interesting examples of posets to apply the statements to.

Finally, we note that Zorn's lemma implies Pataraia's lemma with the following universe parameters. Together with Theorem 52 this yields another proof that Zorn's-Lemma$_{\mathcal{V},\mathcal{V}^+,\mathcal{V}}$ implies Propositional-Resizing$_{\mathcal{V}^+,\mathcal{V}}$.

▶ **Lemma 54.** Zorn's-Lemma$_{\mathcal{V},\mathcal{U}\sqcup\mathcal{T},\mathcal{U}\sqcup\mathcal{T}}$ *implies* Pataraia's-Lemma$_{\mathcal{V},\mathcal{U},\mathcal{T}}$.

Proof. Assume Zorn's-Lemma$_{\mathcal{V},\mathcal{U}\sqcup\mathcal{T},\mathcal{U}\sqcup\mathcal{T}}$ and let $D : \mathcal{U}$ be $\mathcal{V}$-dcpo with order taking values in $\mathcal{T}$. Consider the type MI_D of monotone and inflationary endomaps on D. We can order these maps pointwise to get a $\mathcal{V}$-dcpo with carrier and order taking values in $\mathcal{U} \sqcup \mathcal{T}$. Finally, MI_D has a least element: the identity map. Hence, by our assumption, it has a maximal element $g : D \to D$. It remains to show that g is in fact the greatest element. To this end, let $f : D \to D$ be an arbitrary monotone inflationary endomap on D. We must show that $f \sqsubseteq g$. Since f is inflationary, we have $g \sqsubseteq f \circ g$. So by maximality of g, we get $g = f \circ g$. But f is monotone and g is inflationary, so $f \sqsubseteq f \circ g = g$, finishing the proof. ◀

The answer to the question whether Pataraia's theorem (or similarly, a least fixed point theorem version of Tarki's theorem) is inherently impredicative or (by contrast) does admit a predicative proof has eluded us thus far.

5 Families and Subsets

In traditional impredicative foundations, completeness of posets is usually formulated using subsets. For instance, dcpos are defined as posets D such that every directed subset D has a supremum in D. Examples 15 are all formulated using small families instead of subsets. While subsets are primitive in set theory, families are primitive in type theory, so this could be an argument for using families above. However, that still leaves the natural question of how the family-based definitions compare to the usual subset-based definitions, especially in our predicative setting, unanswered. This section aims to answer this question. We first study the relation between subsets and families predicatively and then clarify our definitions in the presence of impredicativity. In our answers we will consider sup-lattices, but similar arguments could be made for posets with other sorts of completeness, such as dcpos.

All Subsets

We first show that simply asking for completeness w.r.t. all subsets is not satisfactory from a predicative viewpoint. In fact, we will now see that even asking for all subsets $X \to \Omega_{\mathcal{T}}$ for some fixed universe $\mathcal{T}$ is problematic from a predicative standpoint.

▶ **Theorem 55.** *Let $\mathcal{U}$ and $\mathcal{V}$ be universes and fix a proposition $P_{\mathcal{U}} : \mathcal{U}$. Recall $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}})$ from Definition 43, which has $\mathcal{V}$-suprema. Let $\mathcal{T}$ be any type universe. If $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}})$ has suprema for all subsets $\mathcal{L}_{\mathcal{V}}(P_U) \to \Omega_{\mathcal{T}}$, then $P_{\mathcal{U}}$ has size $\mathcal{V}$ independently of $\mathcal{T}$.*

Proof. Let $\mathcal{T}$ be a type universe and consider the subset S of $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}})$ given by $Q \mapsto \mathbb{1}_{\mathcal{T}}$. Note that S has a supremum in $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}})$ if and only if $\mathcal{L}_{\mathcal{V}}(P_{\mathcal{U}})$ has a greatest element, but by Lemma 45, the latter is equivalent to $P_{\mathcal{U}}$ having size $\mathcal{V}$. ◀

All Subsets Whose Total Spaces Have Size $\mathcal{V}$

The proof above illustrates that if we have a subset $S : X \to \Omega_{\mathcal{T}}$, then there is no reason why the total space $\sum_{x:X} x \in S :\equiv \sum_{x:X}(S(x)$ holds) should have size $\mathcal{T}$. In fact, for $S(x) :\equiv \mathbb{1}_{\mathcal{T}}$ as above, the latter is equivalent to asking that X has size $\mathcal{T}$.

▶ **Definition 56** (Total space of a subset, $\mathbb{T}$)**.** *Let $\mathcal{T}$ be a universe, X a type and $S : X \to \Omega_{\mathcal{T}}$ a subset of X. The* total space *of S is defined as $\mathbb{T}(S) :\equiv \sum_{x:X} x \in S$.*

A naive attempt to solve the problem described in Theorem 55 would be to stipulate that a $\mathcal{V}$-sup-lattice X should have suprema for all subsets $S : X \to \Omega_{\mathcal{V}}$ for which $\mathbb{T}(S)$ has size $\mathcal{V}$. Somewhat less naively, we might be more liberal and ask for suprema of subsets $S : X \to \Omega_{\mathcal{U} \sqcup \mathcal{V}}$ for which $\mathbb{T}(S)$ has size $\mathcal{V}$. Here the carrier of X is in a universe $\mathcal{U}$. Perhaps surprisingly, even this more liberal definition is too weak to be useful as the following example shows.

▶ **Example 57** (Naturally occurring subsets whose total spaces are not necessarily small)**.** Let X be a poset with carrier in $\mathcal{U}$ and suppose that it has suprema for all (directed) subsets $S : X \to \Omega_{\mathcal{U} \sqcup \mathcal{V}}$ for which $\mathbb{T}(S)$ has size $\mathcal{V}$. Now let $f : X \to X$ be a Scott continuous endofunction on X. We would want to construct the least fixed point of f as the supremum of the directed subset $S :\equiv \{\bot, f(\bot), f^2(\bot), \dots\}$. Now, how do we show that its total space $\mathbb{T}(S) \equiv \sum_{x:X}(\exists_{n:\mathsf{N}}\, x = f^n(\bot))$ has size $\mathcal{V}$? A first guess might be that $\mathsf{N} \simeq \mathbb{T}(S)$, which would do the job. However, it's possible that $f^m(\bot) = f^{m+1}(\bot)$ for some natural number m, which would mean that $\mathbb{T}(S) \simeq \mathrm{Fin}(m)$ for the least such m. The problem is that in the absence of decidable equality on X we might not be able to decide which is the case. But X seldom has decidable equality, as we saw in Theorems 37 and 38.

▶ Remark 58. The example above also makes clear that it is undesirable to impose an injectivity condition on families, as the family $\mathsf{N} \to X, n \mapsto f^n(\bot)$ is not necessarily injective. In fact, for every type $X : \mathcal{U}$ there is an equivalence between embeddings $I \hookrightarrow X$ with $I : \mathcal{V}$ and subsets of X whose total spaces have size $\mathcal{V}$, cf. [16, `Slice.html`].

All $\mathcal{V}$-covered Subsets

The point of Example 57 is analogous to the difference between Bishop finiteness and Kuratowski finiteness. Inspired by this, we make the following definition.

▶ **Definition 59** ($\mathcal{V}$-covered subset)**.** *Let X be a type, $\mathcal{T}$ a universe and $S : X \to \Omega_{\mathcal{T}}$ a subset of X. We say that S is* $\mathcal{V}$-covered *for a universe $\mathcal{V}$ if we have a type $I : \mathcal{V}$ with a surjection $e : I \twoheadrightarrow \mathbb{T}(S)$.*

In the example above, the subset $S :\equiv \{\bot, f(\bot), f^2(\bot), \dots\}$ is $\mathcal{U}_0$-covered, because $\mathsf{N} \twoheadrightarrow \mathbb{T}(S)$.

▶ **Theorem 60.** *For $X : \mathcal{U}$ and any universe $\mathcal{V}$ we have an equivalence between $\mathcal{V}$-covered subsets $X \to \Omega_{\mathcal{U} \sqcup \mathcal{V}}$ and families $I \to X$ with $I : \mathcal{V}$.*

Proof. The forward map φ is given by $(S, I, e) \mapsto (I, \mathrm{pr}_1 \circ e)$. In the other direction, we define ψ by mapping (I, α) to the triple (S, I, e) where S is the subset of X given by $S(x) :\equiv \exists_{i:I}\, x = \alpha(i)$ and $e : I \twoheadrightarrow \mathbb{T}(S)$ is defined as $e(i) :\equiv (\alpha(i), |(i, \mathrm{refl})|)$. The composite $\varphi \circ \psi$ is easily seen to be equal to the identity. To show that $\psi \circ \varphi$ equals the identity, we need the following intermediate result, which is proved using function extensionality and path induction.

▷ Claim. Let $S, S' : X \to \Omega_{\mathcal{U} \sqcup \mathcal{V}}$, $e : I \to \mathbb{T}(S)$ and $e' : I \to \mathbb{T}(S')$. If $S = S'$ and $\mathrm{pr}_1 \circ e \sim \mathrm{pr}_1 \circ e'$, then $(S, e) = (S', e')$.

The result then follows from the claim using function extensionality and propositional extensionality. ◀

▶ **Corollary 61.** *Let X be a poset with carrier in $\mathcal{U}$ and let $\mathcal{V}$ be any universe. Then X has suprema for all $\mathcal{V}$-covered subsets $X \to \Omega_{\mathcal{U} \sqcup \mathcal{V}}$ if and only if X has suprema for all families $I \to X$ with $I : \mathcal{V}$.*

Families and Subsets in the Presence of Impredicativity

Finally, we compare our family-based approach to the subset-based approach in the presence of impredicativity.

▶ **Theorem 62.** *Assume Ω-Resizing$_{\mathcal{T},\mathcal{U}_0}$ for every universe $\mathcal{T}$. Then the following are equivalent for a poset X in a universe $\mathcal{U}$:*
- **(i)** *X has suprema for all subsets;*
- **(ii)** *X has suprema for all $\mathcal{U}$-covered subsets;*
- **(iii)** *X has suprema for all subsets whose total spaces have size $\mathcal{U}$;*
- **(iv)** *X has suprema for all families $I \to X$ with $I : \mathcal{U}$.*

Proof. Clearly (i) ⇒ (ii) ⇒ (iii). We show that (iii) implies (i), which proves the equivalence of (i)–(iii). Assume that X has suprema for all subsets whose total spaces have size $\mathcal{U}$ and let $S : X \to \Omega_{\mathcal{T}}$ be any subset of X. Using Ω-Resizing$_{\mathcal{T},\mathcal{U}_0}$, the total space $\mathbb{T}(S)$ has size $\mathcal{U}$. So X has a supremum for S by assumption, as desired. Finally, (ii) and (iv) are equivalent by Corollary 61. ◀

Notice that (iv) in Theorem 62 implies that X has suprema for all families $I \to X$ with $I : \mathcal{V}$ and $\mathcal{V}$ such that $\mathcal{V} \sqcup \mathcal{U} \equiv \mathcal{U}$. Typically, in the examples of [12] for instance, $\mathcal{U} :\equiv \mathcal{U}_1$ and $\mathcal{V} :\equiv \mathcal{U}_0$, so that $\mathcal{V} \sqcup \mathcal{U} \equiv \mathcal{U}$ holds. Thus, our $\mathcal{V}$-families-based approach generalizes the traditional subset-based approach.

6 Conclusion

Firstly, we have shown, constructively and predicatively, that nontrivial dcpos, bounded complete posets and sup-lattices are all necessarily large and necessarily lack decidable equality. We did so by deriving a weak impredicativity axiom or weak excluded middle from the assumption that such nontrivial structures are small or have decidable equality, respectively. Strengthening nontriviality to the (classically equivalent) positivity condition, we derived a strong impredicativity axiom and full excluded middle.

Secondly, we proved that Zorn's lemma, Tarski's greatest fixed point theorem and Pataraia's lemma all imply impredicativity axioms. Hence, these principles are inherently impredicative and a predicative development of order theory (in univalent foundations) must thus do without them.

Thirdly, we clarified, in our predicative setting, the relation between the traditional definition of a lattice that requires completeness with respect to subsets and our definition that asks for completeness with respect to small families.

In future work, we wish to study the predicative validity of Pataraia's theorem and Tarski's *least* fixed point theorem. Curi [9, 10] develops predicative versions of Tarki's fixed point theorem in some extensions of CZF. It is not clear whether these arguments could be adapted to univalent foundations, because they rely on the set-theoretical principles discussed in the introduction. The availability of such fixed-point theorems would be especially useful for application to inductive sets [1], which we might otherwise introduce in univalent foundations using higher inductive types [24]. In another direction, we have developed a notion of apartness [5] for continuous dcpos [12] that is related to the notion of being strictly below introduced in this paper. Namely, if $x \sqsubseteq y$ are elements of a continuous dcpo, then x is strictly below y if x is apart from y. In upcoming work, we give a constructive analysis of the Scott topology [17] using this notion of apartness.

References

1. Peter Aczel. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, pages 739–782. Elsevier, 1977. `doi:10.1016/S0049-237X(08)71120-0`.
2. Peter Aczel and Michael Rathjen. Notes on constructive set theory. Book draft, August 2010. URL: `https://www1.maths.leeds.ac.uk/~rathjen/book.pdf`.
3. J. L. Bell. Zorn's lemma and complete Boolean algebras in intuitionistic type theories. *The Journal of Symbolic Logic*, 62(4):1265–1279, 1997. `doi:10.2307/2275642`.
4. Douglas Bridges and Fred Richman. *Varieties of Constructive Mathematics*, volume 97 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1987.
5. Douglas S. Bridges and Luminiţa Simona Vîţă. *Apartness and Uniformity: A Constructive Development*. Springer, 2011. `doi:10.1007/978-3-642-22415-7`.
6. Thierry Coquand, Giovanni Sambin, Jan Smith, and Silvio Valentini. Inductively generated formal topologies. *Annals of Pure and Applied Logic*, 124(1–3):71–106, 2003. `doi:10.1016/s0168-0072(03)00052-6`.
7. Giovanni Curi. On some peculiar aspects of the constructive theory of point-free spaces. *Mathematical Logic Quarterly*, 56(4):375–387, 2010. `doi:10.1002/malq.200910037`.
8. Giovanni Curi. On the existence of Stone-Čech compactification. *The Journal of Symbolic Logic*, 75(4):1137–1146, 2010. `doi:10.2178/jsl/1286198140`.
9. Giovanni Curi. On Tarski's fixed point theorem. *Proceedings of the American Mathematical Society*, 143(10):4439–4455, 2015. `doi:10.1090/proc/12569`.
10. Giovanni Curi. Abstract inductive and co-inductive definitions. *The Journal of Symbolic Logic*, 83(2):598–616, 2018. `doi:10.1017/jsl.2018.13`.
11. Giovanni Curi and Michael Rathjen. Formal Baire space in constructive set theory. In Ulrich Berger, Hannes Diener, Peter Schuster, and Monika Seisenberger, editors, *Logic, Construction, Computation*, volume 3 of *Ontos Matematical Logic*, pages 123–136. De Gruyter, 2012. `doi:10.1515/9783110324921.123`.
12. Tom de Jong and Martín Hötzel Escardó. Domain theory in constructive and predicative univalent foundations. In Christel Baier and Jean Goubault-Larrecq, editors, *29th EACSL Annual Conference on Computer Science Logic (CSL 2021)*, volume 183 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:18. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.CSL.2021.28`.
13. Martín H. Escardó. Joins in the frame of nuclei. *Applied Categorical Structures*, 11:117–124, 2003. `doi:10.1023/A:1023555514029`.
14. Martín H. Escardó and Cory M. Knapp. Partial elements and recursion via dominances in univalent type theory. In Valentin Goranko and Mads Dam, editors, *26th EACSL Annual Conference on Computer Science Logic (CSL 2017)*, volume 82 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.CSL.2017.21`.
15. Martín Hötzel Escardó. Introduction to univalent foundations of mathematics with Agda, November 2020. `arXiv:1911.00580`.

16 Martín Hötzel Escardó. Various new theorems in constructive univalent mathematics written in Agda. `https://github.com/martinescardo/TypeTopology`, June 2020. Agda development.

17 G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *Continuous Lattices and Domains*, volume 93 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 2003. `doi:10.1017/CBO9780511542725`.

18 Hajime Ishihara. Reverse mathematics in Bishop's constructive mathematics. *Philosophia Scientiæ*, CS 6:43–59, 2006. `doi:10.4000/philosophiascientiae.406`.

19 Peter T. Johnstone. Open locales and exponentiation. In J. W. Gray, editor, *Mathematical Applications of Category Theory*, volume 30 of *Contemporary Mathematics*, pages 84–116. American Mathematical Society, 1984. `doi:10.1090/conm/030/749770`.

20 Dito Pataraia. A constructive proof of Tarski's fixed-point theorem for dcpos. Presented at the 65th Peripatetic Seminar on Sheaves and Logic, 1997.

21 Giovanni Sambin. Intuitionistic formal spaces – a first communication. In *Mathematical logic and its applications*, pages 187–204. Springer, 1987. `doi:10.1007/978-1-4613-0897-3_12`.

22 Michael Shulman. Idempotents in intensional type theory. *Logical Methods in Computer Science*, 12(3):1–24, 2016. `doi:10.2168/LMCS-12(3:9)2016`.

23 Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955. `doi:10.2140/pjm.1955.5.285`.

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

25 Benno van den Berg. *Predicative topos theory and models for constructive set theory*. PhD thesis, Utrecht University, 2006. URL: `http://dspace.library.uu.nl/handle/1874/8850`.

26 Vladimir Voevodsky. Resizing rules – their use and semantic justification. Slides from a talk at TYPES, Bergen, 11 September, 2011. URL: `https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/2011_Bergen.pdf`.

27 Vladimir Voevodsky. An experimental library of formalized mathematics based on the univalent foundations. *Mathematical Structures in Computer Science*, 25(5):1278–1294, 2015.

A Strong Call-By-Need Calculus

Thibaut Balabonski
Université Paris-Saclay, CNRS, ENS Paris-Saclay, LMF, Gif-sur-Yvette, 91190, France

Antoine Lanco
Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, Gif-sur-Yvette, 91190, France

Guillaume Melquiond
Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, Gif-sur-Yvette, 91190, France

Abstract

We present a *call-by-need* λ-calculus that enables *strong* reduction (that is, reduction inside the body of abstractions) and guarantees that arguments are only evaluated if needed and at most once. This calculus uses explicit substitutions and subsumes the existing strong-call-by-need strategy, but allows for more reduction sequences, and often shorter ones, while preserving the *neededness*.

The calculus is shown to be *normalizing* in a strong sense: Whenever a λ-term t admits a normal form n in the λ-calculus, then *any* reduction sequence from t in the calculus eventually reaches a representative of the normal form n. We also exhibit a restriction of this calculus that has the *diamond* property and that only performs reduction sequences of minimal length, which makes it systematically better than the existing strategy. We have used the Abella proof assistant to formalize part of this calculus, and discuss how this experiment affected its design.

2012 ACM Subject Classification Theory of computation → Operational semantics

Keywords and phrases strong reduction, call-by-need, evaluation strategy, normalization

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.9

Related Version *Full Version*: https://hal.inria.fr/hal-03149692

1 Introduction

Lambda-calculus is seen as the standard model of computation in functional programming languages, once equipped with an *evaluation strategy* [26]. The most famous evaluation strategies are *call-by-value*, which eagerly evaluates the arguments of a function before resolving the function call, *call-by-name*, where the arguments of a function are evaluated when they are needed, and *call-by-need* [28, 5], which extends call-by-name with a memoization or sharing mechanism to remember the value of an argument that has already been evaluated.

The strength of call-by-name is that it only evaluates terms whose value is effectively needed, at the (possibly huge) cost of evaluating some terms several times. Conversely, the strength *and* weakness of call-by-value (by far the most used strategy in actual programming languages) is that it evaluates each function argument exactly once, even when its value is not actually needed and when its evaluation does not terminate. At the cost of memoization, call-by-need combines the benefits of call-by-value and call-by-name, by only evaluating needed arguments and evaluating them only once.

A common point of these strategies is that they are concerned with *evaluation*, that is computing *values*. As such they operate in the subset of λ-calculus called *weak reduction*, in which there is no reduction inside λ-abstractions, the latter being already considered to be values. Some applications however, such as proof assistants or partial evaluation, require reducing inside λ-abstractions, and possibly aiming for the actual normal form of a λ-term.

The first known abstract machine computing the normal form of a term is due to Crégut [16] and implements normal order reduction. More recently, several lines of work have transposed the known evaluation strategies to strong reduction strategies or abstract

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi; Article No. 9; pp. 9:1–9:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

machines: call-by-value [19, 10, 3], call-by-name [1], and call-by-need [9, 11]. Some non-advertised strong extensions of call-by-name or call-by-need can also be found in the internals of proof assistants, notably Coq.

These strong strategies are mostly conservative over their underlying weak strategy, and often proceed by *iteratively* applying a weak strategy to open terms. In other words, they use a restricted form of strong reduction to enable reduction to normal form, but do not try to take advantage of strong reduction to obtain shorter reduction sequences. Since call-by-need has been shown to capture optimal weak reduction [8], it is known that the deliberate use of strong reduction [20] is the only way of allowing shorter reduction sequences.

This paper presents a strong call-by-need calculus, which obeys the following guidelines. First, it only reduces needed redexes. Second, it keeps a level of sharing at least equal to that of call-by-value and call-by-need. Third, it tries to enable strong reduction as generally as possible. This calculus builds on the syntax and a part of the meta-theory of λ-calculus with explicit substitutions, which we recall in Section 2.

Neededness of a redex is undecidable in general, thus the first and third guidelines are antagonist. Section 3 resolves this tension by exposing a simple syntactic criterion capturing more needed redexes than what is already used in call-by-need strategies. Through reducing only needed redexes, our calculus enjoys a normalization preservation theorem that is stronger than usual: Any λ-term that is *weakly* normalizing in the pure λ-calculus (there is at least one reduction sequence to a normal form, but some other sequences may diverge) will be *strongly* normalizing in our calculus (any reduction sequence is normalizing). This strong normalization theorem, related to the usual *completeness* results of call-by-name or call-by-need strategies, is completely dealt with using a system of non-idempotent intersection types. This avoids the traditional tedious syntactic commutation lemmas, hence providing more elegant proofs. This is an improvement over the technique used in previous works [22, 9].

While our calculus contains the strong call-by-need strategy introduced in [9], it also allows more liberal call-by-need strategies that anticipate some strong reduction steps in order to reduce the overall length of the reduction to normal form. Section 4 presents a restriction of the calculus that guarantees reduction sequences of minimal length.

Finally, Section 5 presents a formalization of parts of our results in Abella [6]. Beyond the proof safety provided by such a tool, this formalization has also influenced the design of our strong call-by-need calculus itself in a positive way. In particular, it promoted a presentation based on SOS-style local reduction rules [27], which later became a lever for a more efficient use of non-idempotent intersection types. The formalization can be found at the following address: `https://hal.inria.fr/hal-03149692`.

2 The host calculus λ_{c}

Our strong call-by-need calculus is included in an already known calculus λ_{c}, that serves as a technical tool in [9] and which we name our *host calculus*. This calculus gives the general shape of reduction rules allowing memoization and comes with a system of non-idempotent intersection types. Its reduction however is not constrained by any notion of neededness.

The λ_{c}-calculus uses the following syntax of λ-terms with explicit substitutions, which is isomorphic to the original syntax of the call-by-need calculus using let-bindings [5].

$$t \in \Lambda_{\mathsf{c}} \quad ::= \quad x \mid \lambda x.t \mid t\ t \mid t[x\backslash t]$$

The free variables $\mathsf{fv}(t)$ of a term t are defined as usual. We call pure λ-term a term that contains no explicit substitution. We write $\mathcal{C}$ for a context, *i.e.*, a term with exactly one hole $\square$, and L for a context with the specific shape $\square[x_1\backslash t_1]\ldots[x_n\backslash t_n]$ ($n \geqslant 0$). We write $\mathcal{C}[t]$

for the term obtained by plugging the subterm t in the hole of the context $\mathcal{C}$ (with possible capture of free variables of t by binders in $\mathcal{C}$), or tL when the context is of the specific shape L. We also write $\mathcal{C}[\![t]\!]$ for plugging a term t whose free variables are not captured by $\mathcal{C}$. The *values* we consider are λ-abstractions.

Reduction in λ_{c} is defined by the following three reduction rules, applied in any context. Rather than using traditional propagation rules for explicit substitutions [21], it builds on the *Linear Substitution Calculus* [25, 4, 2] which is more similar to the let-in constructs commonly used for defining call-by-need.

$$\begin{array}{rcll} (\lambda x.t)L\ u & \to_{\mathsf{dB}} & t[x\backslash u]L & \\ \mathcal{C}[\![x]\!][x\backslash vL] & \to_{\mathsf{lsv}} & \mathcal{C}[\![v]\!][x\backslash v]L & \text{with } v \text{ value} \\ t[x\backslash u] & \to_{\mathsf{gc}} & t & \text{with } x \notin \mathsf{fv}(t) \end{array}$$

The rule $\to_{\mathsf{dB}}$ describes β-reduction “at a distance”. It applies to a β-redex whose λ-abstraction is possibly hidden by a list of explicit substitutions. This rule is a combination of a single use of β-reduction with a repeated use of the structural rule lifting the explicit substitutions at the left of an application. The rule $\to_{\mathsf{lsv}}$ describes the linear substitution of a value, *i.e.*, the substitution of one occurrence of the variable x bound by an explicit substitution. It has to be understood as a lookup operation. Similarly to $\to_{\mathsf{dB}}$, this rule embeds a repeated use of a structural rule for unnesting explicit substitutions. Note that this calculus only allows the substitution of λ-abstractions, and not of variables as it is sometimes seen [24]. This restricted behavior is enough for the main results of this paper, and will allow a more compact presentation. Finally, the rule $\to_{\mathsf{gc}}$ describes garbage collection of an explicit substitution for a variable that does not live anymore. Reduction by any of these rules in any context is written $t \to_{\mathsf{c}} u$.

A term t of λ_{c} is related to a pure λ-term $t^\star$ by the unfolding operation which applies all the explicit substitutions. We write $t\{x\backslash u\}$ for the meta-level substitution of x by u in t.

$$\begin{array}{rclcrcl} x^\star & = & x & \qquad & (t\ u)^\star & = & t^\star\ u^\star \\ (\lambda x.t)^\star & = & \lambda x.(t^\star) & & (t[x\backslash u])^\star & = & (t^\star)\{x\backslash u^\star\} \end{array}$$

Through unfolding, any reduction step $t \to_{\mathsf{c}} u$ in λ_{c} is related to a sequence of reductions $t^\star \to_\beta^* u^\star$ in the pure λ-calculus.

The host calculus λ_{c} comes with a system of non-idempotent intersection types [15, 18], defined in [23] by adding explicit substitutions to an original system from [18]. A type τ may be a type variable α or an arrow type $\mathcal{M} \to \tau$, where $\mathcal{M}$ is a multiset $\{\!\{\sigma_1, \ldots, \sigma_n\}\!\}$ of types. A typing environment Γ associates to each variable in its domain a multiset of types. This multiset contains one type for each potential use of the variable, and may be empty if the variable is not actually used. A typing judgment $\Gamma \vdash t : \tau$ assigns exactly one type to the term t. As shown by the typing rules in Fig. 1, an argument of an application or of an explicit substitution may be typed several times in a derivation. Note that, in the rules, the subscript $\sigma \in \mathcal{M}$ quantifies on all the instances of elements in the multiset $\mathcal{M}$. This type system is known to characterize λ-terms that are weakly normalizing for β-reduction, if associated with the side condition that the empty multiset $\{\!\{\}\!\}$ does not appear at a positive position in the typing judgment. Posititive type occurrences $\mathcal{T}_+(\Gamma \vdash t : \tau)$ and negative type occurrences $\mathcal{T}_-(\Gamma \vdash t : \tau)$ of a typing judgment are defined by the following equations.

$$\begin{array}{rclcrcl} \mathcal{T}_+(\alpha) & = & \{\alpha\} & \qquad & \mathcal{T}_-(\alpha) & = & \emptyset \\ \mathcal{T}_+(\mathcal{M}) & = & \{\mathcal{M}\} \cup \bigcup_{\sigma\in\mathcal{M}} \mathcal{T}_+(\sigma) & & \mathcal{T}_-(\mathcal{M}) & = & \bigcup_{\sigma\in\mathcal{M}} \mathcal{T}_-(\sigma) \\ \mathcal{T}_+(\mathcal{M} \to \sigma) & = & \{\mathcal{M} \to \sigma\} \cup \mathcal{T}_-(\mathcal{M}) \cup \mathcal{T}_+(\sigma) & & \mathcal{T}_-(\mathcal{M} \to \sigma) & = & \mathcal{T}_+(\mathcal{M}) \cup \mathcal{T}_-(\sigma) \\ \mathcal{T}_+(\Gamma \vdash t : \sigma) & = & \mathcal{T}_+(\sigma) \cup \bigcup_{x\in\mathsf{dom}(\Gamma)} \mathcal{T}_-(\Gamma(x)) & & & & \end{array}$$

$$\dfrac{}{x : \{\!\{\sigma\}\!\} \vdash x : \sigma}\ \textsc{ty-var} \qquad \dfrac{\Gamma \vdash t : \mathcal{M} \to \tau \qquad (\Delta_\sigma \vdash u : \sigma)_{\sigma \in \mathcal{M}}}{\Gamma + \sum_{\sigma \in \mathcal{M}} \Delta_\sigma \vdash t\ u : \tau}\ \textsc{ty-@}$$

$$\dfrac{\Gamma; x : \mathcal{M} \vdash t : \tau}{\Gamma \vdash \lambda x.t : \mathcal{M} \to \tau}\ \textsc{ty-}\lambda \qquad \dfrac{\Gamma; x : \mathcal{M} \vdash t : \tau \qquad (\Delta_\sigma \vdash u : \sigma)_{\sigma \in \mathcal{M}}}{\Gamma + \sum_{\sigma \in \mathcal{M}} \Delta_\sigma \vdash t[x \backslash u] : \tau}\ \textsc{ty-es}$$

Figure 1 Typing rules for λ_{c}.

▶ **Theorem 1** (Typability [17, 12]). *If the pure λ-term t is weakly normalizing for β-reduction, then there is a typing judgment $\Gamma \vdash t : \tau$ such that $\{\!\{\}\!\} \notin \mathcal{T}_+(\Gamma \vdash t : \tau)$.*

A typing derivation Φ for a typing judgment $\Gamma \vdash t : \tau$ (written $\Phi \rhd \Gamma \vdash t : \tau$) defines in t a set of *typed positions*, which are the positions of the subterms of t for which the derivation Φ contains a subderivation. More precisely:

- ε is a typed position for any derivation;
- if Φ ends with rule TY-λ, TY-@ or TY-ES, then $0p$ is a typed position of Φ if p is a typed position of the subderivation Φ' relative to the first premise;
- if Φ ends with rule TY-@ or TY-ES, then $1p$ is a typed position of Φ if p is a typed position of the subderivation Φ' relative to one of the instances of the second premise.

Note that, in the latter case, there is no instance of the second premise and no typed position $1p$ when the multiset $\mathcal{M}$ is empty. On the contrary, when $\mathcal{M}$ has several elements, we get the union of the typed positions contributed by each instance.

These typed positions have an important property; they satisfy a *weighted* subject reduction theorem ensuring a decreasing derivation size, which we will use in the next section. We call size of a derivation Φ the number of nodes of the derivation tree.

▶ **Theorem 2** (Weighted subject reduction [9]). *If $\Phi \rhd \Gamma \vdash t : \tau$ and $t \to_{\mathsf{c}} t'$ by reduction of a redex at a typed position, then there is a derivation $\Phi' \rhd \Gamma \vdash t' : \tau$ with Φ' smaller than Φ.*

3 Strong call-by-need calculus λ_{sn}

Our strong call-by-need calculus is defined by the same terms and reduction rules as λ_{c}, with restrictions on where the reduction rules can be applied. These restrictions ensure in particular that only the needed redexes are reduced. Notice that gc-reduction is never needed in this calculus and will thus be ignored from now on.

3.1 Reduction in λ_{sn}

The main reduction relation is written $t \to_{\mathsf{sn}} t'$ and represents one step of dB or lsv reduction at an eligible position of the term t. The starting point is the same as the one for the original (weak) call-by-need calculus. Since the argument of a function is not always needed, we do not reduce in advance the right part of an application $t\ u$. Instead, we first evaluate t to an answer $(\lambda x.t')L$, then apply a dB-reduction to put the argument u in the environment of t', and then go on with the resulting term $t'[x\backslash u]L$, evaluating u only if and when it is required.

Strong reduction. The previous principle is enough for weak reduction, but new behaviors appear with strong reduction. For instance, consider the top-level term $\lambda x.x\ t\ u$. It is an abstraction, which would not need to be further evaluated in weak call-by-need. Here however, we have to reduce it further to reach its putative normal form. So, let us gather some knowledge on the term. Given its position, we know that this abstraction will never be applied to an argument. This means in particular that its variable x will never be substituted by anything; it is blocked and is now part of the rigid structure of the term. Following [9], we say that variable x is *frozen*. As for the arguments t and u given to the frozen variable x, they will always remain at their respective positions and their neededness is guaranteed. So, the calculus allows their reduction. Moreover, these subterms t and u will never be applied to other subterms; they are in *top-level-like* positions and can be treated as independent terms. In particular, assuming that the top-level term is $\lambda x.x\ (\lambda y.t')\ u$ (that is, t is the abstraction $\lambda y.t'$), the variable y will never be substituted and both variables x and y can be seen as frozen in the subterm t'.

Let us now consider the top-level term $(\lambda x.x\ (\lambda y.t')\ u)\ v$, *i.e.*, the previous one applied to some argument v. The analysis becomes radically different. Indeed, both abstractions in this term are at positions where they may eventually interact with other parts of the term: $(\lambda x \ldots)$ is already applied to an argument, while $(\lambda y.t')$ might eventually be substituted at some position inside v whose properties are not yet known. Thus, none of the abstractions is at a top-level-like position and we cannot rule out the possibility that some occurrences of x or y become substituted eventually. Consequently, neither x nor y can be considered as frozen. In addition, notice that the subterms $\lambda y.t'$ and u are not even guaranteed to be needed in $(\lambda x.x\ (\lambda y.t')\ u)\ v$. Thus our calculus shall not allow them to be reduced yet.

Therefore, the set of top-level-like positions of a subterm t, and more importantly the set of its positions that are eligible for reduction largely depend on the context surrounding t. Consequently, the bulk of the definition of $t \to_{\mathsf{sn}} t'$ is an inductive relation $t \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'$ that plays two roles: identifying a position where a reduction rule can be applied in t, depending on some outer context information, and performing said reduction. The information on the context is abstracted by two parameters of the inductive relation:

- a flag μ indicating whether t is at a top-level-like position ($\top$) or not ($\bot$);
- the set φ of variables that are frozen at the considered position.

The flow of this information along the inductive rules is a critical aspect of the definitions.

Since the identification of positions that are eligible for reduction does not depend on the choice of the rule dB or lsv, the inductive reduction relation is also parametric with respect to the rule. This is the role of the parameter ρ of $\xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}}$, whose value can be dB, lsv, or others that we will introduce shortly. Thus, the top-level reduction relation $t \to_{\mathsf{sn}} t'$ holds whenever $t \xrightarrow{\mathsf{dB},\varphi,\top}_{\mathsf{sn}} t'$ or $t \xrightarrow{\mathsf{lsv},\varphi,\top}_{\mathsf{sn}} t'$, where the flag μ is $\top$, and the set φ is typically empty when t is closed, or contains the free variables of t otherwise.

Inductive rules. The inference rules for $\xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}}$ are given in Fig. 3. Notice that information about φ and μ flow outside-in, that is from top-level to the position of the reduction, or equivalently upward in the inference rules, while ρ flows the other way. Notice also that in this paper, we define top-level-like positions and frozen variables only through these inductive rules.

Rule @-LEFT makes reduction always possible on the left of an application, but as shown by the premise this position is not a $\top$ position. Rule @-RIGHT on the other hand allows reducing on the right of an application, and even doing so in $\top$ mode, but only when the application is led by a frozen variable.

$$\frac{x \in \varphi}{x \in \mathcal{S}_\varphi} \qquad \frac{t \in \mathcal{S}_\varphi}{t\ u \in \mathcal{S}_\varphi} \qquad \frac{t \in \mathcal{S}_\varphi}{t[x\backslash u] \in \mathcal{S}_\varphi} \qquad \frac{t \in \mathcal{S}_{\varphi\cup\{x\}} \qquad u \in \mathcal{S}_\varphi}{t[x\backslash u] \in \mathcal{S}_\varphi}$$

Figure 2 Structures of λ_{sn}.

$$\textsc{@-left}\ \frac{t \xrightarrow{\rho,\varphi,\bot}_{\mathsf{sn}} t'}{t\ u \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'\ u} \qquad \textsc{@-right}\ \frac{t \in \mathcal{S}_\varphi \qquad u \xrightarrow{\rho,\varphi,\top}_{\mathsf{sn}} u'}{t\ u \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t\ u'} \qquad \lambda\textsc{-top}\ \frac{t \xrightarrow{\rho,\varphi\cup\{x\},\top}_{\mathsf{sn}} t'}{\lambda x.t \xrightarrow{\rho,\varphi,\top}_{\mathsf{sn}} \lambda x.t'} \qquad \lambda\textsc{-bot}\ \frac{t \xrightarrow{\rho,\varphi,\bot}_{\mathsf{sn}} t'}{\lambda x.t \xrightarrow{\rho,\varphi,\bot}_{\mathsf{sn}} \lambda x.t'}$$

$$\textsc{es-left}\ \frac{t \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'}{t[x\backslash u] \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'[x\backslash u]} \qquad \textsc{es-left-}\varphi\ \frac{t \xrightarrow{\rho,\varphi\cup\{x\},\mu}_{\mathsf{sn}} t' \qquad u \in \mathcal{S}_\varphi}{t[x\backslash u] \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'[x\backslash u]} \qquad \textsc{es-right}\ \frac{t \xrightarrow{\mathsf{id}_x,\varphi,\mu}_{\mathsf{sn}} t \qquad u \xrightarrow{\rho,\varphi,\bot}_{\mathsf{sn}} u'}{t[x\backslash u] \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t[x\backslash u']}$$

$$\textsc{id}\ \frac{}{x \xrightarrow{\mathsf{id}_x,\varphi,\mu}_{\mathsf{sn}} x} \qquad \textsc{sub}\ \frac{}{x \xrightarrow{\mathsf{sub}_{x\backslash v},\varphi,\mu}_{\mathsf{sn}} v} \qquad \textsc{dB}\ \frac{t \to_{\mathsf{db}} t'}{t \xrightarrow{\mathsf{dB},\varphi,\mu}_{\mathsf{sn}} t'} \qquad \textsc{lsv}\ \frac{t \xrightarrow{\varphi,\mu}_{\mathsf{lsv}} t'}{t \xrightarrow{\mathsf{lsv},\varphi,\mu}_{\mathsf{sn}} t'}$$

Figure 3 Reduction rules for λ_{sn}.

The latter criterion is made formal through the notion of *structure*, which is an application $x\ t_1\ \dots\ t_n$ led by a frozen variable x, possibly interlaced with explicit substitutions (Fig. 2). As implied by the last rule in Fig. 2, an explicit substitution in a structure may even affect the leading variable, provided that the content of the substitution is itself a structure. We write $\mathcal{S}_\varphi$ the set of structures under a set φ of frozen variables. It differs from the notion in [9] in that it does not require the term to be in normal form.

Notice that frozen variables in a term t are either free variables of t, or variables introduced by binders in t. As such they obey the usual renaming conventions. In particular, the third and fourth rules in Fig. 2 implicitly require that the variable x bound by the explicit substitution is fresh and hence *not* in the set φ. We keep this *freshness convention* in all the definitions of the paper.

Rules λ-TOP and λ-BOT make reduction always possible inside a λ-abstraction, *i.e.*, unconditional strong reduction. If the abstraction is in a $\top$ position, its variable is added to the set of frozen variables while reducing the body of the abstraction. Rules ES-LEFT and ES-LEFT-φ show that it is always possible to reduce a term affected by an explicit substitution. If the substitution contains a structure, the variable bound by the substitution can be added to the set of frozen variables. Rule ES-RIGHT restricts reduction inside a substitution to the case where an occurrence of the substituted variable is at a reducible position. It uses an auxiliary rule id_x, which propagates using the same inductive reduction relation, to probe a term for the presence of some variable x at a reduction position. By freshness, $x \notin \varphi$. This auxiliary rule does not modify the term to which it applies, as witnessed by its base case ID.

$$\textsc{dB-base}\quad \frac{}{(\lambda x.t)\ u \to_{\mathsf{db}} t[x\backslash u]} \qquad \textsc{lsv-base}\quad \frac{t \xrightarrow{\mathsf{sub}_{x\backslash v},\varphi,\mu}_{\mathsf{sn}} t' \qquad v \text{ value}}{t[x\backslash v] \xrightarrow{\varphi,\mu}_{\mathsf{lsv}} t'[x\backslash v]}$$

$$\textsc{dB-}\sigma\quad \frac{t\ u \to_{\mathsf{db}} v}{t[x\backslash w]\ u \to_{\mathsf{db}} v[x\backslash w]} \qquad \textsc{lsv-}\sigma\quad \frac{t[x\backslash u] \xrightarrow{\varphi,\mu}_{\mathsf{lsv}} t'}{t[x\backslash u[y\backslash w]] \xrightarrow{\varphi,\mu}_{\mathsf{lsv}} t'[y\backslash w]} \qquad \textsc{lsv-}\sigma\textsc{-}\varphi\quad \frac{t[x\backslash u] \xrightarrow{\varphi\cup\{y\},\mu}_{\mathsf{lsv}} t' \qquad w \in \mathcal{S}_\varphi}{t[x\backslash u[y\backslash w]] \xrightarrow{\varphi,\mu}_{\mathsf{lsv}} t'[y\backslash w]}$$

Figure 4 Auxiliary reduction rules for λ_{sn}.

Rules DB and LSV are the base cases for applying reductions dB or lsv. Using the notations of λ_{c}, they allow the following reductions.

$$\begin{array}{lcll} (\lambda x.t)L\ u & \xrightarrow{\mathsf{dB},\varphi,\mu}_{\mathsf{sn}} & t[x\backslash u]L & \\ \mathcal{C}[\![x]\!][x\backslash vL] & \xrightarrow{\mathsf{lsv},\varphi,\mu}_{\mathsf{sn}} & \mathcal{C}[\![v]\!][x\backslash v]L & \quad \text{with } v \text{ value, and } \mathcal{C} \text{ a suitable context} \end{array}$$

Each is defined using an auxiliary reduction relation dealing with the list L of explicit substitutions. These auxiliary reductions are given in Fig. 4.

Rules DB-BASE and LSV-BASE describe the base cases of the auxiliary reductions, where the list L is empty. Note that, while DB-BASE is an axiom, the inference rule LSV-BASE uses as a premise a reduction $\xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}}$ using a new reduction rule $\rho = \mathsf{sub}_{x\backslash v}$. This reduction rule substitutes one occurrence of the variable x at a reducible position by the value v (with, by freshness, $x \notin \varphi$). As seen for id_x above, this reduction rule propagates using the same inductive reduction relation, and its base case is the rule SUB in Fig. 3. The presence of this premise $t \xrightarrow{\mathsf{sub}_{x\backslash v},\varphi,\mu}_{\mathsf{sn}} t'$ in the rule is the primary reason why the auxiliary relation $\xrightarrow{\varphi,\mu}_{\mathsf{lsv}}$ is parameterized by φ and μ. The combination of the rules LSV and LSV-BASE makes it possible, in the case of a lsv-reduction, to resume the search for a reducible variable in the context in which the substitution has been found (instead of resetting the context). In [9], a similar effect was achieved using a more convoluted condition on a composition of contexts.

Rule DB-σ makes it possible to float out an explicit substitution applied to the left part of an application. That is, if a dB-reduction is possible without the substitution, then the reduction is performed and the substitution is applied to the result. Rules LSV-σ and LSV-σ-φ achieve the same effect with the nested substitutions applied to the value substituted by an lsv-reduction step. As with rule ES-LEFT-φ, if the substitution is a structure, the variable can be frozen. This difference between LSV-σ and LSV-σ-φ can be ignored until Sec. 4.

Finally, note that the strong call-by-need strategy introduced in [9] is included in our calculus. One can recover this strategy by imposing two restrictions on $\xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}}$:

- remove the rule λ-BOT, so as to only reduce abstractions that are in top-level-like positions;
- restrict the rule @-RIGHT to the case where the left member of the application is a structure *in normal form*, since the strategy imposes left-to-right reduction of structures.

$$\frac{x \in \varphi}{x \in \mathcal{N}_\varphi} \qquad \frac{t \in \mathcal{N}_\varphi \quad t \in \mathcal{S}_\varphi \quad u \in \mathcal{N}_\varphi}{t\ u \in \mathcal{N}_\varphi} \qquad \frac{t \in \mathcal{N}_{\varphi \cup \{x\}}}{\lambda x.t \in \mathcal{N}_\varphi}$$

$$\frac{t \in \mathcal{N}_{\varphi \cup \{x\}} \quad u \in \mathcal{N}_\varphi \quad u \in \mathcal{S}_\varphi}{t[x\backslash u] \in \mathcal{N}_\varphi} \qquad \frac{t \in \mathcal{N}_\varphi}{t[x\backslash u] \in \mathcal{N}_\varphi}$$

Figure 5 Normal forms of λ_{sn}.

Example. The reduction $(\lambda a.a\ x)[x\backslash(\lambda y.t)[z\backslash u]\ v] \to_{\mathsf{sn}} (\lambda a.a\ x)[x\backslash t[y\backslash v][z\backslash u]]$ is allowed by λ_{sn}, as shown by the following derivation. The left branch of the derivation checks that an occurrence of the variable x is actually at a needed position in $\lambda a.a\ x$, while its right branch reduces the argument of the substitution.

$$\dfrac{\dfrac{\dfrac{a \in \mathcal{S}_{\{a\}} \qquad \dfrac{}{x \xrightarrow{\mathsf{id}_x,\{a\},\top}_{\mathsf{sn}} x}\ \textsc{id}}{a\ x \xrightarrow{\mathsf{id}_x,\{a\},\top}_{\mathsf{sn}} a\ x}\ \textsc{@-right}}{\lambda a.a\ x \xrightarrow{\mathsf{id}_x,\emptyset,\top}_{\mathsf{sn}} \lambda a.a\ x}\ \lambda\textsc{-top} \qquad \dfrac{\dfrac{\dfrac{}{(\lambda y.t)\ v \to_{\mathsf{db}} t[y\backslash v]}\ \textsc{db-base}}{(\lambda y.t)[z\backslash u]\ v \to_{\mathsf{db}} t[y\backslash v][z\backslash u]}\ \textsc{db-}\sigma}{(\lambda y.t)[z\backslash u]\ v \xrightarrow{\mathsf{dB},\emptyset,\bot}_{\mathsf{sn}} t[y\backslash v][z\backslash u]}\ \textsc{db}}{(\lambda a.a\ x)[x\backslash(\lambda y.t)[z\backslash u]\ v] \xrightarrow{\mathsf{dB},\emptyset,\top}_{\mathsf{sn}} (\lambda a.a\ x)[x\backslash t[y\backslash v][z\backslash u]]}\ \textsc{es-right}$$

3.2 Soundness

The calculus λ_{sn} is sound with respect to the λ-calculus, in the sense that any normalizing reduction in λ_{sn} can be related to a normalizing β-reduction through unfolding. This section establishes this result (Th. 6). All proofs in this section are formalized in Abella.

The first part of the proof requires relating λ_{sn}-reduction to β-reduction.

▶ **Lemma 3** (Simulation). *If $t \to_{\mathsf{sn}} t'$ then $t^\star \to_\beta^* t'^\star$.*

Proof. By induction on the reduction $t \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'$. ◀

The second part requires relating the normal forms of λ_{sn} to β-normal forms. The normal forms of λ_{sn} correspond to the normal forms of the strong call-by-need strategy [9]. They can be characterized by the inductive definition given in Fig. 5.

▶ **Lemma 4** (Normal forms). *$t \in \mathcal{N}_\varphi$ if and only if there is no reduction $t \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'$.*

Proof. The first part (a term cannot be in normal form and reducible) is by induction on the reduction rules. The second part (any term is either a normal form or a reducible term) is by induction on t. ◀

▶ **Lemma 5** (Unfolded normal forms). *If $t \in \mathcal{N}_\varphi$ then $t^\star$ is a normal form in the λ-calculus.*

Proof. By induction on $t \in \mathcal{N}_\varphi$. ◀

Soundness is a direct consequence of the three previous lemmas.

▶ **Theorem 6** (Soundness). *Let t be a λ_{sn}-term. If there is a reduction $t \to_{\mathsf{sn}}^* u$ with u a λ_{sn}-normal form, then $u^\star$ is the β-normal form of $t^\star$.* ◀

This theorem implies that all the λ_{sn}-normal forms of a term t are equivalent modulo unfolding. This mitigates the fact that the calculus, without a gc rule, is not confluent. For instance, the term $(\lambda x.x)\ (\lambda y.(\lambda z.z)y)$ admits two normal forms $(\lambda y.z[z\backslash y])[x\backslash\lambda y.(\lambda z.z)y]$ and $(\lambda y.z[z\backslash y])[x\backslash\lambda y.z[z\backslash y]]$, but both of them unfold to $\lambda y.y$.

3.3 Completeness

Our strong call-by-need calculus is complete with respect to normalization in the λ-calculus in a strong sense: Whenever a λ-term t admits a normal form in the pure λ-calculus, every reduction path in λ_{sn} eventually reaches a representative of this normal form. This section is devoted to proving this completeness result (Th. 12). The proof relies on the non-idempotent intersection type system in the following way. First, typability (Th. 1) ensures that any weakly normalizing λ-term admits a typing derivation (with no positive occurrence of $\{\!\{\}\!\}$). Second, we prove that any λ_{sn}-reduction in a typed λ_{sn}-term t (with no positive occurrence of $\{\!\{\}\!\}$) is at a typed position of t (Th. 11). Third, weighted subject reduction (Th. 2) provides a decreasing measure for λ_{sn}-reduction. Finally, the obtained normal form is related to the β-normal form using Lemmas 3, 4, and 5.

The proof of the forthcoming typed reduction (Th. 11) uses a refinement of the non-idempotent intersection types system of λ_{c}, given in Fig. 6. Both systems derive the same typing judgments with the same typed positions. The refined system however features an annotated typing judgment $\Gamma \vdash^{\mu}_{\varphi} t : \tau$ embedding the same context information that are used in the inductive reduction relation $\xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}}$, namely the set φ of frozen variables at the considered position and a marker μ of top-level-like positions. These annotations are faithful counterparts to the corresponding annotations of λ_{sn} reduction rules; their information flows upward in the inference rules following the same criteria.

In particular, the rule for typing an abstraction is split in two versions TY-λ-$\bot$ and TY-λ-$\top$, the latter being applicable to $\top$ positions and thus freezing the variable bound by the abstraction (in both rules, by freshness convention we assume $x \notin \varphi$). The rule for typing an application is also split in two version: TY-@-$\mathcal{S}$ is applicable when the left part of the application is a structure and marks the right part as a $\top$ position, while TY-@ is applicable otherwise. Note that this second rule allows the argument of the application to be typed even if its position is not (yet) reducible, but its typing is in a $\bot$ position. Finally, the rule for typing an explicit substitution is similarly split in two versions, depending on whether the content of the substitution is a structure or not, and handling the set of frozen variables accordingly. In both cases, the content of the substitution is typed in a $\bot$ position, since this position is never top-level-like. We write $\Phi \rhd \Gamma \vdash^{\mu}_{\varphi} t : \tau$ if there is a derivation Φ of the annotated typing judgment $\Gamma \vdash^{\mu}_{\varphi} t : \tau$. We denote $\mathsf{fzt}(\Phi)$ the set of types associated to frozen variables in judgments of the derivation Φ.

▶ **Lemma 7** (Typing derivation annotation). *If there is a derivation $\Phi \rhd \Gamma \vdash t : \tau$, then for any φ and μ there is a derivation $\Phi' \rhd \Gamma \vdash^{\mu}_{\varphi} t : \tau$ such that the sets of typed positions in Φ and Φ' are equal.*

Proof. By induction on Φ, since annotations do not interfere with typing. ◀

The converse property is also true, by erasing of the annotations, but is not used in the proof of the completeness result.

The most crucial part of the proof of Th. 11 is ensuring that any argument of a typed structure is itself at a typed position. This follows from the following three lemmas.

▶ **Lemma 8** (Typed structure). *If $\Gamma \vdash^{\mu}_{\varphi} t : \tau$ and $t \in \mathcal{S}_{\varphi}$, there is $x \in \varphi$ such that $\tau \in \mathcal{T}_{+}(\Gamma(x))$.*

$$\frac{}{x : \{\!\{\sigma\}\!\} \vdash^{\mu}_{\varphi} x : \sigma}\ \text{TY-VAR} \qquad \frac{\Gamma; x : \mathcal{M} \vdash^{\perp}_{\varphi} t : \tau}{\Gamma \vdash^{\perp}_{\varphi} \lambda x.t : \mathcal{M} \to \tau}\ \text{TY-}\lambda\text{-}\perp \qquad \frac{\Gamma; x : \mathcal{M} \vdash^{\top}_{\varphi \cup \{x\}} t : \tau}{\Gamma \vdash^{\top}_{\varphi} \lambda x.t : \mathcal{M} \to \tau}\ \text{TY-}\lambda\text{-}\top$$

$$\frac{\Gamma \vdash^{\perp}_{\varphi} t : \mathcal{M} \to \tau \qquad (\Delta_\sigma \vdash^{\perp}_{\varphi} u : \sigma)_{\sigma \in \mathcal{M}}}{\Gamma + \sum_{\sigma \in \mathcal{M}} \Delta_\sigma \vdash^{\mu}_{\varphi} t\, u : \tau}\ \text{TY-@} \qquad \frac{\Gamma \vdash^{\perp}_{\varphi} t : \mathcal{M} \to \tau \quad t \in \mathcal{S}_\varphi \quad (\Delta_\sigma \vdash^{\top}_{\varphi} u : \sigma)_{\sigma \in \mathcal{M}}}{\Gamma + \sum_{\sigma \in \mathcal{M}} \Delta_\sigma \vdash^{\mu}_{\varphi} t\, u : \tau}\ \text{TY-@-}\mathcal{S}$$

$$\frac{\Gamma; x : \mathcal{M} \vdash^{\mu}_{\varphi} t : \tau \quad (\Delta_\sigma \vdash^{\perp}_{\varphi} u : \sigma)_{\sigma \in \mathcal{M}}}{\Gamma + \sum_{\sigma \in \mathcal{M}} \Delta_\sigma \vdash^{\mu}_{\varphi} t[x\backslash u] : \tau}\ \text{TY-ES} \qquad \frac{\Gamma; x : \mathcal{M} \vdash^{\mu}_{\varphi \cup \{x\}} t : \tau \quad u \in \mathcal{S}_\varphi \quad (\Delta_\sigma \vdash^{\perp}_{\varphi} u : \sigma)_{\sigma \in \mathcal{M}}}{\Gamma + \sum_{\sigma \in \mathcal{M}} \Delta_\sigma \vdash^{\mu}_{\varphi} t[x\backslash u] : \tau}\ \text{TY-ES-}\varphi$$

Figure 6 Annotated system for non-idempotent intersection types.

Proof. By induction on the structure of t.[1] The most interesting case is the one of an explicit substitution $t_1[x\backslash t_2]$. The induction hypothesis applied on t_1 can give the variable x which does not appear in the conclusion, but in that case t_2 is guaranteed to be a structure whose type contains τ. ◀

▶ **Lemma 9** (Subformula property)**.**

1. *If* $\Phi \rhd \Gamma \vdash^{\top}_{\varphi} t : \tau$ *then* $\begin{cases} \mathcal{T}_+(\mathsf{fzt}(\Phi)) & \subseteq & \bigcup_{x \in \varphi} \mathcal{T}_+(\Gamma(x)) \cup \mathcal{T}_-(\tau) \\ \mathcal{T}_-(\mathsf{fzt}(\Phi)) & \subseteq & \bigcup_{x \in \varphi} \mathcal{T}_-(\Gamma(x)) \cup \mathcal{T}_+(\tau) \end{cases}$
2. *If* $\Phi \rhd \Gamma \vdash^{\perp}_{\varphi} t : \tau$ *then* $\begin{cases} \mathcal{T}_+(\mathsf{fzt}(\Phi)) & \subseteq & \bigcup_{x \in \varphi} \mathcal{T}_+(\Gamma(x)) \\ \mathcal{T}_-(\mathsf{fzt}(\Phi)) & \subseteq & \bigcup_{x \in \varphi} \mathcal{T}_-(\Gamma(x)) \end{cases}$

Proof. By mutual induction on the typing derivations.[1] Most cases are fairly straightforward. The only difficult case comes from the rule TY-@-$\mathcal{S}$, in which there is a premise $\Delta \vdash^{\top}_{\varphi} u : \sigma$ with mode $\top$ but with a type σ that does not clearly appear in the conclusion. Here we need the typed structure (Lem. 8) to conclude. ◀

▶ **Lemma 10** (Typed structure argument)**.** *If* $\Phi \rhd \Gamma \vdash^{\mu}_{\varphi} t : \tau$ *with* $\{\!\{\}\!\} \notin \mathcal{T}_+(\Gamma \vdash t : \tau)$*, then every typing judgment of the shape* $\Gamma' \vdash^{\mu'}_{\varphi'} s : \mathcal{M} \to \sigma$ *in* Φ *with* $s \in \mathcal{S}_{\varphi'}$ *satisfies* $\mathcal{M} \neq \{\!\{\}\!\}$*.*

Proof. Let $\Gamma' \vdash^{\mu'}_{\varphi'} s : \mathcal{M} \to \sigma$ in Φ with $s \in \mathcal{S}_{\varphi'}$. By Lemma 8, there is $x \in \varphi'$ such that $\mathcal{M} \to \sigma \in \mathcal{T}_+(\Gamma'(x))$. Then $\mathcal{M} \in \mathcal{T}_-(\Gamma'(x))$ and $\mathcal{M} \in \mathcal{T}_-(\mathsf{fzt}(\Phi))$. By Lemma 9, $\mathcal{M} \in \mathcal{T}_+(\Gamma \vdash^{\mu}_{\varphi} t : \tau)$, thus $\mathcal{M} \neq \{\!\{\}\!\}$. ◀

▶ **Theorem 11** (Typed reduction)**.** *If* $\Phi \rhd \Gamma \vdash^{\mu}_{\varphi} t : \tau$ *with* $\{\!\{\}\!\} \notin \mathcal{T}_+(\Gamma \vdash t : \tau)$*, then every* λ_{sn}*-reduction* $t \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'$ *is at a typed position.*

Proof. We prove by induction on $t \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'$ that, if $\Phi \rhd \Gamma \vdash^{\mu}_{\varphi} t : \tau$ with Φ such that any typing judgment $\Gamma' \vdash^{\mu'}_{\varphi'} s : \mathcal{M} \to \sigma$ in Φ with $s \in \mathcal{S}_{\varphi'}$ satisfies $\mathcal{M} \neq \{\!\{\}\!\}$, then $t \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'$ reduces at a typed position (the restriction on Φ is enabled by Lemma 10). Since all other reduction cases concern positions that are systematically typed, we focus here on @-RIGHT and ES-RIGHT.

[1] See appendix for the complete proof.

$$\frac{t \xrightarrow{\mathsf{sub}_{x\backslash v},\varphi,\mu}_{\mathsf{sn}} t' \qquad v \in \mathcal{N}_{\varphi,\emptyset,\perp}}{t[x\backslash v] \xrightarrow{\varphi,\mu}_{\mathsf{lsv}} t'[x\backslash v]}\ \text{LSV-BASE}$$

Figure 7 New rule LSV-BASE for $\lambda_{\mathsf{sn}+}$.

- Case @-RIGHT: $t\ u \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t\ u'$ with $t \in \mathcal{S}_\varphi$ and $u \xrightarrow{\rho,\varphi,\top}_{\mathsf{sn}} u'$, assuming $\Phi \rhd \Gamma \vdash^{\mu}_{\varphi} t\ u : \sigma$. By inversion of the last rule in Φ we know there is a subderivation $\Phi' \rhd \Gamma' \vdash^{\perp}_{\varphi} t : \mathcal{M} \to \sigma$ and by hypothesis $\mathcal{M} \neq \{\!\{\}\!\}$. Then u is typed in Φ and we can conclude by induction hypothesis.
- Case ES-RIGHT: $t[x\backslash u] \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t[x\backslash u']$ with $t \xrightarrow{\mathsf{id}_x,\varphi,\mu}_{\mathsf{sn}} t'$ and $u \xrightarrow{\rho,\varphi,\perp}_{\mathsf{sn}} u'$, assuming $\Phi \rhd \Gamma \vdash^{\mu}_{\varphi} t[x\backslash u] : \tau$. By inversion of the last rule in Φ we kown there is a subderivation $\Phi' \rhd \Gamma'; x : \mathcal{M} \vdash^{\mu}_{\varphi} t : \tau$. By induction hypothesis we know that reduction $t \xrightarrow{\mathsf{id}_x,\varphi,\mu}_{\mathsf{sn}} t'$ is at a typed position in Φ', thus x is typed in t and $\mathcal{M} \neq \{\!\{\}\!\}$. Then u is typed in Φ and we can conclude by induction hypothesis on $u \xrightarrow{\rho,\varphi,\perp}_{\mathsf{sn}} u'$. ◀

▶ **Theorem 12** (Completeness). *If a λ-term t is weakly normalizing in the λ-calculus, then t is strongly normalizing in λ_{sn}. Moreover, if n_β is the normal form of t in the λ-calculus, then any normal form n_{sn} of t in λ_{sn} is such that $n^{\star}_{\mathsf{sn}} = n_\beta$.*

Proof. Let t be a pure λ-term that admits a normal form n_β for β-reduction. By Theorem 1 there exists a derivable typing judgment $\Gamma \vdash t : \tau$ such that $\{\!\{\}\!\} \notin \mathcal{T}_+(\Gamma \vdash t : \tau)$. Thus by Theorems 11 and 2, the term t is strongly normalizing for $\to_{\mathsf{sn}}$. Let $t \to^{*}_{\mathsf{sn}} n_{\mathsf{sn}}$ be a maximal reduction in λ_{sn}. By Lemma 4, $n_{\mathsf{sn}} \in \mathcal{N}_\varphi$, and by Lemma 5, $n^{\star}_{\mathsf{sn}}$ is a normal form in the λ-calculus. Moreover, by simulation (Lem. 3), there is a reduction $t^{\star} \to^{*}_{\beta} n^{\star}_{\mathsf{sn}}$. By uniqueness of the normal form in the λ-calculus, $n^{\star}_{\mathsf{sn}} = n_\beta$. ◀

Note that, despite the fact that λ_{sn} does not enjoy the diamond property, our theorems of soundness (Th. 6) and completeness (Th. 12) imply that, in λ_{sn}, a term is weakly normalizing if and only if it is strongly normalizing.

4 Relatively optimal strategies

Our proposed λ_{sn}-calculus guarantees that, in the process of reducing a term to its strong normal form, only needed redexes are ever reduced. This does not tell anything about the length of reduction sequences, though. Indeed, a term might be substituted several times before being reduced, thus leading to duplicate computations. To prevent this duplication, we introduce a notion of *local normal form*, which is used to restrict the *value* criterion in the LSV-BASE rule. This restricted calculus, named $\lambda_{\mathsf{sn}+}$, has the same rules as λ_{sn} (Fig. 3 and 4), except that LSV-BASE is replaced by the rule shown in Fig. 7.

We then show that this restriction is strong enough to guarantee the diamond property. Finally, we explain why our restricted calculus only produces minimal sequences, among all the reduction sequences allowed by λ_{sn}. This makes it a relatively optimal strategy.

4.1 Local normal forms

In λ_{c} and λ_{sn}, substituted terms can be arbitrary values. In particular, they might be abstractions whose body contains some redexes. Since substituted variables can appear multiple times, this would cause the redex to be reduced several times if the value is

$$\frac{x \in \varphi \cup \omega}{x \in \mathcal{N}_{\varphi,\omega,\mu}}\ \text{VAR} \qquad \frac{t \in \mathcal{N}_{\varphi\cup\{x\},\omega,\top}}{\lambda x.t \in \mathcal{N}_{\varphi,\omega,\top}}\ \lambda\text{-}\varphi \qquad \frac{t \in \mathcal{N}_{\varphi,\omega\cup\{x\},\bot}}{\lambda x.t \in \mathcal{N}_{\varphi,\omega,\bot}}\ \lambda\text{-}\omega \qquad \frac{t \in \mathcal{N}_{\varphi,\omega,\mu}}{t[x \setminus u] \in \mathcal{N}_{\varphi,\omega,\mu}}\ \text{ES}$$

$$\frac{t \in \mathcal{N}_{\varphi,\omega,\mu} \qquad t \in \mathcal{S}_\varphi \qquad u \in \mathcal{N}_{\varphi,\omega,\top}}{t\ u \in \mathcal{N}_{\varphi,\omega,\mu}}\ @\text{-}\varphi \qquad \frac{t \in \mathcal{N}_{\varphi,\omega,\mu} \qquad t \in \mathcal{S}_\omega}{t\ u \in \mathcal{N}_{\varphi,\omega,\mu}}\ @\text{-}\omega$$

$$\frac{t \in \mathcal{N}_{\varphi\cup\{x\},\omega,\mu} \qquad u \in \mathcal{N}_{\varphi,\omega,\bot} \qquad u \in \mathcal{S}_\varphi}{t[x \setminus u] \in \mathcal{N}_{\varphi,\omega,\mu}}\ \text{ES-}\varphi \qquad \frac{t \in \mathcal{N}_{\varphi,\omega\cup\{x\},\mu} \qquad u \in \mathcal{S}_\omega}{t[x \setminus u] \in \mathcal{N}_{\varphi,\omega,\mu}}\ \text{ES-}\omega$$

Figure 8 Local normal forms.

substituted too soon. Let us illustrate this phenomenon on the following example, where $\mathsf{id} = \lambda x.x$. The sequence of reductions does not depend on the set φ of frozen variables nor on the position μ, so we do not write them to lighten a bit the notations. Subterms that are about to be substituted or reduced are underlined.

$$\begin{array}{lll}
\underline{(\lambda w.w\ w)\ (\lambda y.\mathsf{id}\ y)} & \xrightarrow{\mathsf{db}}_{\mathsf{sn}} & (\underline{w}\ w)[w\backslash\lambda y.\mathsf{id}\ y] \\
& \xrightarrow{\mathsf{lsv}}_{\mathsf{sn}} & ((\lambda y.\underline{\mathsf{id}\ y})\ w)[w\backslash\lambda y.\mathsf{id}\ y] \\
& \xrightarrow{\mathsf{db}}_{\mathsf{sn}} & \underline{((\lambda y.x[x\backslash y])\ w)}[w\backslash\lambda y.\mathsf{id}\ y] \\
& \xrightarrow{\mathsf{db}}_{\mathsf{sn}} & \underline{x}[x\backslash y][y\backslash w][w\backslash\lambda y.\mathsf{id}\ y] \\
& \xrightarrow{\mathsf{lsv}\times 3}_{\mathsf{sn}} & (\lambda y.\underline{\mathsf{id}\ y})[x\backslash\lambda y.\mathsf{id}\ y][y\backslash\lambda y.\mathsf{id}\ y][w\backslash\lambda y.\mathsf{id}\ y] \\
& \xrightarrow{\mathsf{db}}_{\mathsf{sn}} & (\lambda y.x[x\backslash y])[x\backslash\lambda y.\mathsf{id}\ y][y\backslash\lambda y.\mathsf{id}\ y][w\backslash\lambda y.\mathsf{id}\ y]
\end{array}$$

Notice how $\mathsf{id}\ y$ is reduced twice, which would not have happened if the second reduction had focused on the body of the abstraction.

This suggests that a substitution should only be allowed if the substituted term is in normal form. But such a strong requirement is incompatible with our calculus, as it would prevent the abstraction $\lambda y.y\ \Omega$ (with Ω a diverging term) to ever be substituted in the following example, thus preventing normalization (with a a closed term).

$$\begin{array}{lll}
\underline{w}\ (\lambda x.a)[w\backslash\lambda y.y\ \Omega] & \xrightarrow{\mathsf{lsv}}_{\mathsf{sn}} & \underline{(\lambda y.y\ \Omega)\ (\lambda x.a)}[w\backslash\lambda y.y\ \Omega] \\
& \xrightarrow{\mathsf{db}}_{\mathsf{sn}} & (\underline{y}\ \Omega)[y\backslash\lambda x.a][w\backslash\lambda y.y\ \Omega] \\
& \xrightarrow{\mathsf{lsv}}_{\mathsf{sn}} & (\underline{(\lambda x.a)\ \Omega})[y\backslash\lambda x.a][w\backslash\lambda y.y\ \Omega] \\
& \xrightarrow{\mathsf{db}}_{\mathsf{sn}} & a[x\backslash\Omega][y\backslash\lambda x.a][w\backslash\lambda y.y\ \Omega]
\end{array}$$

Notice how the sequence of reductions has progressively removed all the occurrences of Ω, until the only term left to reduce is the closed term a.

To summarize, substituting any value is too permissive and can cause duplicate computations, while substituting only normal forms is too restrictive as it prevents normalization. So, we need some relaxed notion of normal form, which we call *local normal form*. The intuition is as follows. The term $\lambda y.y\ \Omega$ is not in normal form, because it could be reduced if it was in a $\top$ position. But in a $\bot$ position, variable y is not frozen, which prevents any further reduction of $y\ \Omega$. The inference rules are presented in Fig. 8.

If an abstraction is in a $\top$ position, its variable is added to the set φ of frozen variables, as in Fig. 3. But if an abstraction is in a $\bot$ position, its variable is added to a new set ω, as shown in rule λ-ω of Fig. 8. That is what will happen to y in $\lambda y.y\ \Omega$.

For an application, the left part is still required to be a structure. But if the leading variable of the structure is not frozen (and thus in ω), our λ_{sn}-calculus guarantees that no reduction will occur in the right part of the application. So, this part does not need to be constrained in any way. This is rule @-ω of Fig. 8. It applies to our example, since $y\ \Omega$ is a structure led by $y \in \omega$. Substitutions are handled in a similar way, as shown by rule ES-ω.

4.2 Diamond property

As mentioned before, in both λ_{c} and λ_{sn}, terms might be substituted as soon as they are values, thus potentially causing duplicate computations. As a consequence, these calculi cannot have the diamond property, as shown on the following example.

$$
\begin{tikzcd}
 & ((\lambda x.(\lambda y.y)\ x)\ w)[w\backslash\lambda x.(\lambda y.y)\ x] \arrow[r, "3"'] & ((\lambda x.y[y\backslash x])\ w)[w\backslash\lambda x.(\lambda y.y)\ x] \arrow[d, phantom, "\neq"] \\
(w\ w)[w\backslash\lambda x.(\lambda y.y)\ x] \arrow[ur, "1"'] \arrow[dr, "2"] & & \\
 & (w\ w)[w\backslash\lambda x.y[y\backslash x]] \arrow[r, "4"] & ((\lambda x.y[y\backslash x])\ w)[w\backslash\lambda x.y[y\backslash x]]
\end{tikzcd}
$$

In λ_{sn}, the leftmost term can be reduced, either by rule lsv (arrow 1) because the substituted term is a value, or by rule dB (arrow 2). The top term can only be reduced by rule dB (arrow 3) because the substitution variable is not reachable. The bottom term can only be reduced by rule lsv (arrow 4) because the substituted term is not reducible. The two new terms are different, thus breaking the diamond property. It would take one more reduction step (in λ_{c}) for the top sequence to reach the bottom-right term. But in our restricted calculus $\lambda_{\mathsf{sn+}}$, arrow 1 is forbidden, since the substituted term is not in local normal form. By preventing such sequences, the diamond property is restored.

▶ **Theorem 13** (Diamond). *Suppose $t \xrightarrow{\rho_1,\varphi,\mu}_{\mathsf{sn+}} t_1$ and $t \xrightarrow{\rho_2,\varphi,\mu}_{\mathsf{sn+}} t_2$. Assume that, if ρ_1 and ρ_2 are* sub *or* id*, then they apply to separate variables. Then there exists t' such that $t_1 \xrightarrow{\rho_2,\varphi,\mu}_{\mathsf{sn+}} t'$ and $t_2 \xrightarrow{\rho_1,\varphi,\mu}_{\mathsf{sn+}} t'$.*

Proof. The statement has first to be generalized so that the steps $t \to t_1$ and $t \to t_2$ can use the main reduction $\xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}}$ or the auxiliary reductions $\to_{\mathsf{db}}$ and $\xrightarrow{\varphi,\mu}_{\mathsf{lsv}}$. Then it becomes a tedious but rather unsurprising induction on t, with reasoning by case on the last inference rule applied on each side. One notable case is when the two reductions are respectively given by rules @-LEFT and @-RIGHT. Indeed, the reduction on the left does not interfere with the reduction on the right thanks to a stability property of structures (Lem. 14 below). ◀

▶ **Lemma 14** (Stability of structures). *If $t \in \mathcal{S}_\varphi$ and $t \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn+}} t'$ then $t' \in \mathcal{S}_\varphi$*

4.3 Relative optimality

The $\lambda_{\mathsf{sn+}}$-calculus is a restriction of λ_{sn} that requires terms to be eagerly reduced to local normal form before they can be substituted (Fig. 7). This eager reduction is never wasted: λ_{sn} (and a fortiori its subset $\lambda_{\mathsf{sn+}}$) only reduces needed redexes, that is redexes that are necessarily reduced in any reduction to normal form. As a consequence, reductions in $\lambda_{\mathsf{sn+}}$ are never longer than equivalent reductions in λ_{sn}. On the contrary, by forcing some reductions to be performed before a term is substituted (*i.e.*, potentially duplicated), this strategy produces in many cases reduction sequences that are strictly shorter than the ones given by the original strong call-by-need strategy [9].

▶ **Theorem 15** (Minimality). *With $t' \in \mathcal{N}_\varphi$, if $t \to^n_{\mathsf{sn}} t'$ and $t \to^m_{\mathsf{sn+}} t'$ then $m \leqslant n$.*

Remark that this minimality result is relative to λ_{sn}. The reduction sequences of $\lambda_{\mathsf{sn}+}$ are not necessarily optimal with respect to the unconstrained λ_{c} or λ-calculi. For instance, neither $\lambda_{\mathsf{sn}+}$ nor λ_{sn} allow reducing r in the term $(\lambda x.x\ (x\ a))\ (\lambda y.y\ r)$ prior to its duplication.

5 Formalization in Abella

We used the Coq proof assistant for our first attempts to formalize our results. We experimented both with the locally nameless approach [13] and parametric higher-order abstract syntax [14]. While we might eventually have succeeded using the locally nameless approach, having to manually handle binders felt way too cumbersome. So, we turned to a dedicated formal system, Abella [6], in the hope that it would make syntactic proofs more straightforward. This section describes our experience with this tool.[2]

5.1 Nominal variables and λ-tree syntax

Our initial motivation for using Abella was the availability of nominal variables through the `nabla` quantifier. Indeed, in order to open a bound term, one has to replace the bound variable with a fresh global variable. This freshness is critical to avoid captures; but handling it properly causes a lot of bureaucracy in the proofs. By using nominal variables, which are guaranteed to be fresh by the logic, this issue disappears.

Here is an excerpt of our original definition of the `nf` predicate, which states that a term is in normal form for our calculus. The second line states that any nominal variable is in normal form, while the third line states that an abstraction is in normal form, as long as the abstracted term is in normal form for any nominal variable.

```
Define nf : trm -> prop by
  nabla x, nf x;
  nf (abs U) := nabla x, nf (U x);
  ...
```

Note that Abella is based on a λ-tree approach (higher-order abstract syntax). In the above excerpt, `U` has a bound variable and `(U x)` substitutes it with the fresh variable `x`. More generally, `(U V)` is the term in which the bound variable is substituted with the term `V`.

This approach to fresh variables was error-prone at first. Several of our formalized theorems ended up being pointless, despite seemingly matching the statements of our pen-and-paper formalization. Consider the following example. This proposition states that, if `T` is a structure with respect to `x`, and if `U` is related to `T` by the unfolding relation `star`, then `U` is also a structure with respect to `x`.

```
forall T U, nabla x,
struct T x -> star T U -> struct U x.
```

Notice that the nominal variable `x` is quantified after `T`. As a consequence, its freshness ensures that it does not occur in `T`. Thus, the proposition is vacuously true, since `T` cannot be a structure with respect to a variable that does not occur in it. Had the quantifiers been exchanged, the statement would have been fine. Unfortunately, Abella kind of requires universal quantifiers to happen before nominal ones in theorem statements, thus exacerbating the issue. The correct way to state the above proposition is by carefully lifting any term in which a given free variable could occur:

[2] See appendix for the definitions and the statement of the main theorems, and online material for the full development.

```
forall T U, nabla x,
struct (T x) x -> star (T x) (U x) -> struct (U x) x.
```

Once one has overcome these hurdles, advantages become apparent. For example, to state that some free variable does not occur in a term, not lifting this term is sufficient. And if it needed to be lifted for some other reason, one can always equate it to a constant λ-tree. For instance, one of our theorems needs to state that the free variable `x` occurring in `T` cannot occur in `U`, by virtue of `star`. This is expressed as follows (with `y\V` denoting $y \mapsto V$):

```
star (T x) (U x) -> exists V, U = (y\ V).
```

5.2 Judgments, contexts, and derivations

Abella provides two levels of logic: a minimal logic used for specifications and an intuitionistic logic used for inductive reasoning over relations. At first, we only used the reasoning logic. By doing so, we were using Abella as if we were using Coq, except for the additional `nabla` quantifiers. We knew of the benefits of the specification logic when dealing with judgments and contexts; but in the case of the untyped λ-calculus, we could not see any use for those.

Our point of view started to shift once we had to manipulate sets of free variables, in order to distinguish which of them were frozen. We could have easily formalized such sets by hand; but since Abella is especially designed to handle sets of binders, we gave it a try. Let us consider the above predicate `nf` anew, except that it is now defined using λ-Prolog rules (`pi` is the universal quantifier in the specification logic).

```
nf X :- frozen X.
nf (abs U) :- pi x\ frozen x => nf (U x).
...
```

Specification-level propositions have the form `{L |- P}`, with `P` a proposition defined in λ-Prolog and `L` a list of propositions representing the context of `P`. Consider the proposition `{L |- nf (abs T)}`. If there were only the two rules above, there would be only three ways of deriving the proposition. Indeed, it can be derived from `{L |- frozen (abs T)}` (first rule). It can also be derived from `nabla x, {L, frozen x |- nf (T x)}` (second rule). Finally, the third way to derive it is if `nf (abs T)` is already a member of the context `L`.

The second and third derivations illustrate how Abella automates the handling of contexts. But where Abella shines is that some theorems come for free when manipulating specification-level properties, especially when it comes to substitution. Suppose that one wants to prove `{L |- P (T U)}`, *i.e.*, term `T` whose bound variable was replaced with `U` satisfies predicate `P` in context `L`. The simplest way is if one can prove `nabla x, {L |- P (T x)}`. In that case, one can instantiate the nominal variable `x` with `U` and conclude.

But more often that not, `x` occurs in the context, *e.g.*, `{L, Q x |- P (T x)}` instead of `{L |- P (T x)}`. Then, proving `{L |- P (T U)}` is just a matter of proving `{L |- Q x}`. But, what if the latter does not hold? Suppose one can only prove `{L |- R x}`, with `R V :- Q V`. In that case, one can reason on the derivation of `{L, Q x |- P (T x)}` and prove that `{L, R x |- P (T x)}` necessarily holds, by definition of `R`. This ability to inductively reason on derivations is a major strength of Abella.

Having to manipulate contexts led us to revisit most of our pen-and-paper concepts. For example, a structure was no longer defined as a relation with respect to its leading variable (*e.g.*, `struct T x`) but with respect to all the frozen variables (*e.g.*, `{frozen x |- struct T}`). In turn, this led us to handle live variables purely through their addition to contexts: $\varphi \cup \{x\}$. Our freshness convention is a direct consequence, as in Fig. 2 for example.

Performing specification-level proofs does not come without its own set of issues, though. As explained earlier, a proposition `{L |- nf (abs T)}` is derivable from the consequent being part of the context `L`, which is fruitless. The way around it is to define a predicate describing contexts that are well-formed, *e.g.*, `L` contains only propositions of the form `(nf x)` with `x` nominal. As a consequence, the case above can be eliminated because `(abs T)` is not a nominal variable. Unfortunately, defining these predicates and proving the associated helper lemmas is tedious and extremely repetitive. Thus, the user is encouraged to reuse existing context predicates rather than creating dedicated new ones, hence leading to sloppy and convoluted proofs. Having Abella provide some automation for handling well-formed contexts would be a welcome improvement.

5.3 Functions and relations

Our Abella formalization assumes a type `trm` and three predefined ways to build elements of that type: application, abstraction, and explicit substitution. For example, a term $t[x\backslash u]$ of our calculus will be denoted `(es (x\t) u)` with `t` containing some occurrences of `x`.

```
type app trm -> trm -> trm.
type abs (trm -> trm) -> trm.
type es  (trm -> trm) -> trm -> trm.
```

Since Abella does not provide functions, we instead use a relation to define the unfolding function $t \mapsto t^\star$. Of particular interest is the way binders are handled; they are characterized by stating that they are their own image: `star x x`.

```
star (app U V) (app X Y) :- star U X, star V Y.
star (abs U) (abs X) :- pi x\ star x x => star (U x) (X x).
star (es U V) (X Y) :- star V Y, pi x\ star x x => star (U x) (X x).
```

Since this is just a relation, we have to prove that it is defined over all the closed terms of our calculus, that it maps only to pure λ-terms, and that it maps to exactly one λ-term. Needless to say, all of that would be simpler if Abella had native support for functions.

6 Conclusion

This paper presents a λ-calculus dedicated to strong reduction. In the spirit of a call-by-need strategy with explicit substitutions, it builds on a linear substitution calculus [2]. Our calculus, however, embeds a syntactic criterion that ensures that only needed redexes are considered. Moreover, by delaying substitutions until they are in so-called local normal forms rather than just values, all the reduction sequences are of minimal length.

Properly characterizing these local normal forms proved difficult and lots of iterations were needed until we reached the presented definition. Our original approach relied on evaluation contexts, as in the original presentation of a strong call-by-need strategy [9]. While tractable, this made the proof of the diamond property long and tedious. It is the use of Abella that led us to reconsider this approach. Indeed, the kind of reasoning Abella favors forced us to give up on evaluation contexts and look for reduction rules that were much more local in nature. In turn, these changes made the relation with typing more apparent. In hindsight, this would have avoided a large syntactic proof in [9].

Due to decidability, our syntactic criterion can characterize only part of the needed redexes at a given time. All the needed reductions will eventually happen, but detecting the neededness of a redex too late might prevent the optimal reduction. It is an open question whether some other simple criterion would characterize more needed redexes, and thus potentially allow for even shorter sequences than our calculus.

Even with the current criterion, there is still work to be done. First and foremost, the Abella formalization should be completed to at least include the diamond property. There are also some potential improvements to consider. For example, our calculus could be made to not substitute variables that are not applied (rule LSV-BASE), following [29, 3] but it opens the question of how to characterize the normal forms then. Another venue for investigation is how this work interacts with fully lazy sharing, wich avoids more duplications but whose properties are tightly related to weak reduction [7]. Finally, this paper stops at describing the reduction rules of our calculus and does not investigate what an efficient abstract machine would look like.

References

1. Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. A strong distillery. In Xinyu Feng and Sungwoo Park, editors, *Programming Languages and Systems*, volume 9458 of *Lecture Notes in Computer Science*, pages 231–250, 2015. `doi:10.1007/978-3-319-26529-2_13`.
2. Beniamino Accattoli, Eduardo Bonelli, Delia Kesner, and Carlos Lombardi. A nonstandard standardization theorem. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 659–670, 2014. `doi:10.1145/2535838.2535886`.
3. Beniamino Accattoli, Andrea Condoluci, and Claudio Sacerdoti Coen. Strong call-by-value is reasonable, implosively, 2021. `arXiv:2102.06928`.
4. Beniamino Accattoli and Delia Kesner. The structural λ-calculus. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic*, pages 381–395, 2010. `doi:10.5555/1887459.1887491`.
5. Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. A call-by-need lambda calculus. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 233–246, 1995. `doi:10.1145/199448.199507`.
6. David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2):1–89, 2014. `doi:10.6092/issn.1972-5787/4650`.
7. Thibaut Balabonski. A unified approach to fully lazy sharing. In John Field and Michael Hicks, editors, *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 469–480, 2012. `doi:10.1145/2103656.2103713`.
8. Thibaut Balabonski. Weak optimality, and the meaning of sharing. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming*, ICFP'13, pages 263–274, September 2013. `doi:10.1145/2500365.2500606`.
9. Thibaut Balabonski, Pablo Barenbaum, Eduardo Bonelli, and Delia Kesner. Foundations of strong call by need. *Proc. ACM Program. Lang.*, 1(ICFP), 2017. `doi:10.1145/3110264`.
10. Malgorzata Biernacka, Dariusz Biernacki, Witold Charatonik, and Tomasz Drab. An abstract machine for strong call by value. In Bruno C. d. S. Oliveira, editor, *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020*, volume 12470 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 2020. `doi:10.1007/978-3-030-64437-6_8`.
11. Malgorzata Biernacka and Witold Charatonik. Deriving an Abstract Machine for Strong Call by Need. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:20, 2019. `doi:10.4230/LIPIcs.FSCD.2019.8`.
12. Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. Non-idempotent intersection types for the Lambda-Calculus. *Logic Journal of the IGPL*, 25(4):431–464, July 2017. `doi:10.1093/jigpal/jzx018`.
13. Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, 2012. `doi:10.1007/s10817-011-9225-2`.
14. Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *13th ACM SIGPLAN International Conference on Functional Programming*, ICFP, pages 143–156, 2008. `doi:10.1145/1411204.1411226`.

15 M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980. doi:10.1305/ndjfl/1093883253.

16 Pierre Crégut. An abstract machine for lambda-terms normalization. In *ACM Conference on LISP and Functional Programming*, LFP '90, page 333–340, 1990. doi:10.1145/91556.91681.

17 Daniel de Carvalho. *Sémantiques de la logique linéaire et temps de calcul.* PhD thesis, Université Aix-Marseille II, 2007.

18 Philippa Gardner. Discovering needed reductions using type theory. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, pages 555–574, 1994.

19 Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *7th ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, page 235–246, 2002. doi:10.1145/581478.581501.

20 Carsten Kehler Holst and Darsten Krogh Gomard. Partial evaluation is fuller laziness. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '91, page 223–233, 1991. doi:10.1145/115865.115890.

21 Delia Kesner. A theory of explicit substitutions with safe and full composition. *Logical Methods in Computer Science*, 5(3), 2009. doi:10.2168/LMCS-5(3:1)2009.

22 Delia Kesner. Reasoning about call-by-need by means of types. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures*, pages 424–441, 2016. doi:10.1007/978-3-662-49630-5_25.

23 Delia Kesner and Daniel Ventura. Quantitative types for the linear substitution calculus. In Josep Diaz, Ivan Lanese, and Davide Sangiorgi, editors, *Theoretical Computer Science*, volume 8705 of *Lecture Notes in Computer Science*, pages 296–310, 2014. doi:10.1007/978-3-662-44602-7_23.

24 John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *J. Funct. Program.*, 8(3):275–317, May 1998. doi:10.1017/S0956796898003037.

25 Robin Milner. Local bigraphs and confluence: Two conjectures. *Electron. Notes Theor. Comput. Sci.*, 175(3):65–73, 2007. doi:10.1016/j.entcs.2006.07.035.

26 Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975. doi:10.1016/0304-3975(75)90017-1.

27 Gordon D. Plotkin. A structural approach to operational semantics. Technical report, DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

28 Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford, 1971.

29 Nobuko Yoshida. Optimal reduction in weak-lambda-calculus with shared environments. In *Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, page 243–252, 1993. doi:10.1145/165180.165217.

A Formal definitions

This appendix describes the main definitions of the Abella formalization. The reduction rules of λ_{sn} and $\lambda_{\mathsf{sn+}}$ presented in Fig. 3 are as follows.

```
step R top (abs T) (abs T') :-
  pi x\ frozen x => step R top (T x) (T' x).
step R B (abs T) (abs T') :- pi x\ omega x => step R bot (T x) (T' x).
step R B (app T U) (app T' U) :- step R bot T T'.
step R B (app T U) (app T U') :- struct T, step R top U U'.
step R B (es T U) (es T' U) :- pi x\ omega x => step R B (T x) (T' x).
step R B (es T U) (es T' U) :-
  pi x\ frozen x => step R B (T x) (T' x), struct U.
step R B (es T U) (es T U') :-
  pi x\ active x => step (idx x) B (T x) (T x), step R bot U U'.
step (idx X) B X X :- active X.
```

```
step (sub X V) B X V :- active X.
step db B T T' :- aux_db T T'.
step lsv B T T' :- aux_lsv B T T'.
```

A small difference with the core of the paper is the predicate `active`, which characterizes the variable being considered id_x (`idx`) and $\mathsf{sub}_{x\backslash v}$ (`sub`). This predicate is just a cheap way of remembering that the active variable is fresh yet not frozen. Similarly, the predicate `omega` is used in two rules to tag a variable as being neither frozen nor active. Another difference is rule λ-BOT. While the antecedent of the rule is at position $\bot$ as in the paper, the consequent is in any position rather than just $\bot$. Since any term reducible in position $\bot$ is provably reducible in position $\top$, this is just a conservative generalization of the rule.

The auxiliary rules for $\lambda_{\mathsf{sn}+}$, as given in Fig. 4 and Fig. 7 for rule LSV-BASE, are the same as in the core of the paper.

```
aux_db (app (abs T) U) (es T U).
aux_db (app (es T W) U) (es T' W) :-
  pi x\ aux_db (app (T x) U) (T' x).
aux_lsv B (es T (abs V)) (es T' (abs V)) :-
  pi x\ active x => step (sub x (abs V)) B (T x) (T' x),
  lnf bot (abs V).
aux_lsv B (es T (es U W)) (es T' W) :-
  pi x\ omega x => aux_lsv B (es T (U x)) (T' x).
aux_lsv B (es T (es U W)) (es T' W) :-
  pi x\ frozen x => aux_lsv B (es T (U x)) (T' x), struct W.
```

Finally, an actual reduction is just comprised of rules DB and LSV in a $\top$ position:

```
red T T' :- step db top T T'.
red T T' :- step lsv top T T'.
```

The normal forms of λ_{sn} and $\lambda_{\mathsf{sn}+}$, given in Fig. 5, are as follows.

```
nf X :- frozen X.
nf (app U V) :- nf U, nf V, struct U.
nf (abs U) :- pi x\ frozen x => nf (U x).
nf (es U V) :- pi x\ frozen x => nf (U x), nf V, struct V.
nf (es U V) :- pi x\ nf (U x).
```

They make use of structures (`struct`), as given in Fig. 2.

```
struct X :- frozen X.
struct (app U V) :- struct U.
struct (es U V) :- pi x\ struct (U x).
struct (es U V) :- pi x\ frozen x => struct (U x), struct V.
```

The local norm forms of Fig. 8 are as follows. As for the `step` relation, one of the rules for abstraction was generalized with respect to the paper. This time, it is for the $\top$ position, since any term that is locally normal in a $\top$ position is locally normal in any position.

```
lnf B X :- frozen X.
lnf B X :- omega X.
lnf B (app T U) :- lnf B T, struct T, lnf top U.
lnf B (app T U) :- lnf B T, struct_omega T.
lnf B (abs T) :- pi x\ frozen x => lnf top (T x).
lnf bot (abs T) :- pi x\ omega x => lnf bot (T x).
lnf B (es T U) :- pi x\ active x => lnf B (T x).
lnf B (es T U) :- pi x\ frozen x => lnf B (T x), lnf bot U, struct U.
lnf B (es T U) :- pi x\ omega x => lnf B (T x), struct_omega U.
```

Structures with respect to the set ω use a dedicated predicate `struct_omega`, which is just a duplicate of `struct`. Another approach, perhaps more elegant, would have been to parameterize `struct` with either `frozen` or `omega`.

```
struct_omega X :- omega X.
struct_omega (app U V) :- struct_omega U.
struct_omega (es U V) :- pi x\ struct_omega (U x).
struct_omega (es U V) :-
  pi x\ omega x => struct_omega (U x), struct_omega V.
```

Normal forms of the λ-calculus are defined as follows:

```
nfb X :- frozen X.
nfb (abs T) :- pi x\ frozen x => nfb (T x).
nfb (app T U) :- nfb T, nfb U, notabs T.
notabs T :- frozen T.
notabs (app T U).
```

The definition of λ_{sn}-terms is sometimes useful to allow induction on terms rather than induction on one of the previous predicates.

```
trm (app U V) :- trm U, trm V.
trm (abs U) :- pi x\ trm x => trm (U x).
trm (es U V) :- pi x\ trm x => trm (U x), trm V.
```

Finally, let us remind the definitions of a pure λ-term, of the unfolding operation from λ_{c} to λ, of a β-reduction, and of a sequence of zero or more β-reductions.

```
pure (app U V) :- pure U, pure V.
pure (abs U) :- pi x\ pure x => pure (U x).
star (app U V) (app X Y) :- star U X, star V Y.
star (abs U) (abs X) :- pi x\ star x x => star (U x) (X x).
star (es U V) (X Y) :- star V Y, pi x\ star x x => star (U x) (X x).

beta (app M N) (app M' N) :- beta M M'.
beta (app M N) (app M N') :- beta N N'.
beta (abs R) (abs R') :- pi x\ beta (R x) (R' x).
beta (app (abs R) M) (R M).
betas M M.
betas M N :- beta M P, betas P N.
```

B Formally verified properties

This appendix states the theorems that were fully proved using Abella. First comes the simulation property (Lem. 3), which states that, if $T \to_{\mathsf{sn}+} U$, then $T^\star \to_\beta^* U^\star$.

```
Theorem simulation' : forall T U T* U*,
  {star T T*} -> {star U U*} -> {red T U} -> {betas T* U*}.
```

Then comes the fact that (local) normal forms are exactly the terms that are not reducible in $\lambda_{\mathsf{sn}+}$ (Lem. 4).

```
Theorem lnf_nand_red : forall T U,
  {lnf top T} -> {red T U} -> false.
Theorem nf_nand_red : forall T U,
  {nf T} -> {red T U} -> false.
```

```
Theorem lnf_or_red : forall T,
  {trm T} -> {lnf top T} \/ exists U, {red T U}.
Theorem nf_or_red : forall T,
  {trm T} -> {nf T} \/ exists U, {red T U}.
```

Finally, if T is a normal form of λ_{sn}, then $T^\star$ is a normal form of the λ-calculus (Lem. 5).

```
Theorem nf_star' : forall T T*,
  {nf T} -> {star T T*} -> {nfb T*}.
```

C Proof of the subformula properties

We recall here Lemma 8:
If $\Gamma \vdash^{\mu}_{\varphi} t : \tau$ and $t \in \mathcal{S}_\varphi$, then there is $x \in \varphi$ such that $\tau \in \mathcal{T}_+(\Gamma(x))$.

Proof. By induction on the structure of t.

- Case $t = x$. By inversion of $x \in \mathcal{S}_\varphi$ we deduce $x \in \varphi$. Moreover the only rule applicable to derive $\Gamma \vdash^{\mu}_{\varphi} x : \tau$ is TY-VAR, which gives the conclusion.
- Case $t = t_1\ t_2$. By inversion of $t_1\ t_2 \in \mathcal{S}_\varphi$ we deduce $t_1 \in \mathcal{S}_\varphi$. Moreover the only rules applicable to derive $\Gamma \vdash^{\mu}_{\varphi} t_1\ t_2 : \tau$ are TY-@ and TY-@-$\mathcal{S}$. Both have a premise $\Gamma' \vdash^{\perp}_{\varphi} t_1 : \mathcal{M} \to \tau$ with $\Gamma' \subseteq \Gamma$, to which the induction hypothesis applies, ensuring $\mathcal{M} \to \tau \in \mathcal{T}_+(\Gamma'(x))$ and thus $\tau \in \mathcal{T}_+(\Gamma'(x))$ and $\tau \in \mathcal{T}_+(\Gamma(x))$.
- Case $t = t_1[x\backslash t_2]$. We reason by case on the last rules applied to derive $t_1[x\backslash t_2] \in \mathcal{S}_\varphi$ and $\Gamma \vdash^{\mu}_{\varphi} t_1[x\backslash t_2] : \tau$. There are two possible rules for each.
  - Case where $t_1[x\backslash t_2] \in \mathcal{S}_\varphi$ is deduced from $t_1 \in \mathcal{S}_\varphi$ (with $x \notin \varphi$) and $\Gamma \vdash^{\mu}_{\varphi} t_1[x\backslash t_2] : \tau$ comes from rule TY-ES. This rule has in particular a premise $\Gamma' \vdash^{\mu}_{\varphi} t_1 : \tau$ for a $\Gamma' = \Gamma''; x : \mathcal{M}$ such that $\Gamma'' \subseteq \Gamma$. We thus have by induction hypothesis on t_1 that $\tau \in \mathcal{T}_+(\Gamma'(y))$ for some $y \in \varphi \cap \mathsf{dom}(\Gamma')$. Since $y \in \varphi$ and $x \notin \varphi$ we have $y \neq x$. Then $y \in \mathsf{dom}(\Gamma'')$ and $y \in \mathsf{dom}(\Gamma)$, and $\Gamma(y) = \Gamma''(y)$.
  - In the three other cases, we have:
    1. a hypothesis $t_1 \in \mathcal{S}_\varphi$ or $t_1 \in \mathcal{S}_{\varphi \cup \{x\}}$, from which we deduce $t_1 \in \mathcal{S}_{\varphi \cup \{x\}}$,
    2. a hypothesis $\Gamma' \vdash^{\mu}_{\varphi} t_1 : \tau$ or $\Gamma' \vdash^{\mu}_{\varphi \cup \{x\}} t_1 : \tau$ (for a $\Gamma' = \Gamma''; x : \mathcal{M}$ such that $\Gamma'' \subseteq \Gamma$), from which we deduce $\Gamma' \vdash^{\mu}_{\varphi \cup \{x\}} t_1 : \tau$, and
    3. a hypothesis $t_2 \in \mathcal{S}_\varphi$, coming from the derivation of $t_1[x\backslash t_2]$ or the derivation of $\Gamma \vdash^{\mu}_{\varphi} t_1[x\backslash t_2] : \tau$ (or both).

    Then by induction hypothesis on t_1 we have $\tau \in \mathcal{T}_+(\Gamma'(y))$ for some $y \in \varphi \cup \{x\}$.
    - If $y \neq x$, then $y \in \varphi$ and $\Gamma(y) = \Gamma''(y)$, which allows a direct conclusion.
    - If $y = x$, then $\tau \in \mathcal{T}_+(\Gamma'(x))$ implies $\mathcal{M} \neq \{\!\{\}\!\}$. Let $\sigma \in \mathcal{M}$ with $\tau \in \mathcal{T}_+(\sigma)$. The instance of the rule TY-ES or TY-ES-φ we consider thus has at least one premise $\Delta \vdash^{\perp}_{\varphi} t_2 : \sigma$ with $\Delta \subseteq \Gamma$. Since $t_2 \in \mathcal{S}_\varphi$, by induction hypothesis on t_2 there is $z \in \varphi \cap \mathsf{dom}(\Delta)$ such that $\sigma \in \mathcal{T}_+(\Delta(z))$. Then $\tau \in \mathcal{T}_+(\Delta(z))$, and $\tau \in \Gamma$. ◀

We recall here Lemma 9:

1. If $\Phi \rhd \Gamma \vdash^{\top}_{\varphi} t : \tau$ then $\begin{cases} \mathcal{T}_+(\mathsf{fzt}(\Phi)) & \subseteq & \bigcup_{x \in \varphi} \mathcal{T}_+(\Gamma(x)) \cup \mathcal{T}_-(\tau) \\ \mathcal{T}_-(\mathsf{fzt}(\Phi)) & \subseteq & \bigcup_{x \in \varphi} \mathcal{T}_-(\Gamma(x)) \cup \mathcal{T}_+(\tau) \end{cases}$
2. If $\Phi \rhd \Gamma \vdash^{\perp}_{\varphi} t : \tau$ then $\begin{cases} \mathcal{T}_+(\mathsf{fzt}(\Phi)) & \subseteq & \bigcup_{x \in \varphi} \mathcal{T}_+(\Gamma(x)) \\ \mathcal{T}_-(\mathsf{fzt}(\Phi)) & \subseteq & \bigcup_{x \in \varphi} \mathcal{T}_-(\Gamma(x)) \end{cases}$

Proof. By mutual induction on the typing derivations.

- Both properties are immediate in case TY-VAR, where $\mathsf{fzt}(\Phi) = \{\sigma\}$.
- Cases for abstractions.
  - If $\Phi \rhd \Gamma \vdash^{\perp}_{\varphi} \lambda x.t : \mathcal{M} \to \tau$ by rule TY-λ-$\perp$ with premise $\Phi' \rhd \Gamma; x : \mathcal{M} \vdash^{\perp}_{\varphi} t : \tau$. Write $\Gamma' = \Gamma; x : \mathcal{M}$. By induction hypothesis we have $\mathcal{T}_+(\mathsf{fzt}(\Phi')) \subseteq \bigcup_{y\in\varphi} \mathcal{T}_+(\Gamma'(y))$. Since $x \notin \varphi$ by renaming convention, we deduce that $\mathcal{T}_+(\mathsf{fzt}(\Phi')) \subseteq \bigcup_{y\in\varphi} \mathcal{T}_+(\Gamma(y))$ and $\mathcal{T}_+(\mathsf{fzt}(\Phi)) \subseteq \bigcup_{y\in\varphi} \mathcal{T}_+(\Gamma(y))$. The same applies to negative type occurrences, which concludes the case.
  - If $\Phi \rhd \Gamma \vdash^{\top}_{\varphi} \lambda x.t : \mathcal{M} \to \tau$ by rule TY-λ-$\top$ with premise $\Phi' \rhd \Gamma; x : \mathcal{M} \vdash^{\top}_{\varphi\cup\{x\}} t : \tau$. Write $\Gamma' = \Gamma; x : \mathcal{M}$. By induction hypothesis we have

$$\begin{array}{rcl} \mathcal{T}_+(\mathsf{fzt}(\Phi')) & \subseteq & \bigcup_{y\in(\varphi\cup\{x\})} \mathcal{T}_+(\Gamma'(y)) \cup \mathcal{T}_-(\tau) \\ & = & \bigcup_{y\in\varphi} \mathcal{T}_+(\Gamma(y)) \cup \mathcal{T}_+(\mathcal{M}) \cup \mathcal{T}_-(\tau) \\ & = & \bigcup_{y\in\varphi} \mathcal{T}_+(\Gamma(y)) \cup \mathcal{T}_-(\mathcal{M} \to \tau) \end{array}$$

    Thus $\mathcal{T}_+(\mathsf{fzt}(\Phi)) \subseteq \bigcup_{y\in\varphi} \mathcal{T}_+(\Gamma(y)) \cup \mathcal{T}_-(\mathcal{M} \to \tau)$. The same applies to negative occurrences, which concludes the case.
- Cases for application.
  - Cases for TY-@ are by immediate application of the induction hypotheses.
  - If $\Phi \rhd \Gamma \vdash^{\mu}_{\varphi} t\ u : \tau$ by rule TY-@-$\mathcal{S}$, with premises $\Phi_t \rhd \Gamma_t \vdash^{\perp}_{\varphi} t : \mathcal{M} \to \tau$, $t \in \mathcal{S}_\varphi$ and $\Phi_\sigma \rhd \Delta_\sigma \vdash^{\top}_{\varphi} u : \sigma$ for $\sigma \in \mathcal{M}$, with $\Gamma_t \subseteq \Gamma$ and $\Gamma_\sigma \subseteq \Gamma$ for all $\sigma \in \mathcal{M}$. Independently of the value of μ, we show that $\mathcal{T}_+(\mathsf{fzt}(\Phi)) \subseteq \bigcup_{x\in\varphi} \mathcal{T}_+(\Gamma(x))$ and $\mathcal{T}_-(\mathsf{fzt}(\Phi)) \subseteq \bigcup_{x\in\varphi} \mathcal{T}_-(\Gamma(x))$ to conclude on both sides of the mutual induction.
    Directly from the induction hypothesis, $\mathcal{T}_+(\mathsf{fzt}(\Phi_t)) \subseteq \bigcup_{x\in\varphi} \Gamma_t(x) \subseteq \mathcal{T}_+(\mathsf{fzt}(\Phi))$. By induction hypothesis on the other premises we have $\mathcal{T}_+(\mathsf{fzt}(\Phi_\sigma)) \subseteq \bigcup_{x\in\varphi} \Gamma_\sigma(x) \cup \mathcal{T}_-(\tau)$ for $\sigma \in \mathcal{M}$. We immediately have $\bigcup_{x\in\varphi} \Gamma_\sigma(x) \subseteq \bigcup_{x\in\varphi} \Gamma(x)$. We conclude by showing that $\mathcal{T}_-(\sigma) \subseteq \mathcal{T}_+(\Gamma_t(x))$ for some $x \in \varphi$. Since $t \in \mathcal{S}_\varphi$, by the first subformula property and the typing hypothesis on t we deduce that there is a $x \in \varphi$ such that $\mathcal{M} \to \tau \in \mathcal{T}_+(\Gamma_t(x))$. By closeness of type occurrences sets $\mathcal{T}_+(\tau)$ this means $\mathcal{T}_+(\mathcal{M} \to \tau) \subseteq \mathcal{T}_+(\Gamma_t(x))$. By definition $\mathcal{T}_+(\mathcal{M} \to \tau) = \mathcal{T}_-(\mathcal{M}) \cup \mathcal{T}_+(\tau) = \bigcup_{\sigma\in\mathcal{M}} \mathcal{T}_-(\sigma) \cup \mathcal{T}_+(\tau)$, which allows us to conclude the proof that $\bigcup_{x\in\varphi} \Gamma_\sigma(x) \cup \mathcal{T}_-(\tau) \subseteq \bigcup_{x\in\varphi} \Gamma(x)$.
    The same argument also applies to negative positions, and concludes the case.
- Cases for explicit substitution immediately follow the induction hypothesis. ◀

A Bicategorical Model for Finite Nondeterminism

Zeinab Galal
IRIF, Université de Paris, France

Abstract

Finiteness spaces were introduced by Ehrhard as a refinement of the relational model of linear logic. A finiteness space is a set equipped with a class of finitary subsets which can be thought of being subsets that behave like finite sets. A morphism between finiteness spaces is a relation that preserves the finitary structure. This model provided a semantics for finite non-determism and it gave a semantical motivation for differential linear logic and the syntactic notion of Taylor expansion. In this paper, we present a bicategorical extension of this construction where the relational model is replaced with the model of generalized species of structures introduced by Fiore et al. and the finiteness property now relies on finite presentability.

2012 ACM Subject Classification Theory of computation → Linear logic; Theory of computation → Categorical semantics

Keywords and phrases Differential linear logic, Species of structures, Finiteness, Bicategorical semantics

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.10

Funding This work was partly funded by the ANR project PPSANR-19-CE48-0014.

1 Introduction

1.1 Quantitative semantics

In quantitative semantics, the interpretation of a program provides information on the number of times the program uses its input to compute a given output whereas qualitative semantics only allows us to recover which inputs were used. Quantitative semantics originates from Girard's normal functor semantics of system F [16]. His original intuition was to interpret types as vector spaces such that linear maps between them correspond to programs using their arguments exactly once and analytic functions correspond to general programs.

This approach led to the birth of linear logic but it does not directly provide a model for it. Indeed, the exponential modality of linear logic leads to infinite dimensional vector spaces which are no longer isomorphic to their double dual, a property required to model classical negation. Topological vector spaces were therefore considered to circumvent this issue [17, 6, 8]. In this setting, the series interpreting a program usually has infinite support describing all its possible behaviors for all possible inputs which allows for the study of non-deterministic languages.

1.2 Controlling non-deterministic computation

Finiteness spaces are a model of linear logic introduced by Ehrhard as a way to enforce finite interactions between programs and reject infinite computations [9]. Finiteness spaces do not provide a model of PCF since the fixpoint operator is not a morphism in the model. Vaux showed however that it allows for primitive recursion and is hence a model of Gödel's system T [26].

The construction of the finiteness spaces model is done in two steps: the first step is a double glueing construction (in the sense of Hyland and Schalk [20]) on the relational model **Rel**. A finiteness space $A = (|A|, \mathscr{F}A)$ is a countable set $|A|$ together with a set of

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi; Article No. 10; pp. 10:1–10:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

finitary subsets $\mathscr{F}A$ such that the intersection of a finitary subset in $\mathscr{F}A$ together with a finitary subset in the dual type $\mathscr{F}A^{\perp}$ is always finite. Morphisms between finiteness spaces are relations that preserve the finitary structure backward and forward.

The second step is parameterized by a fixed field (or commutative semi-ring) $\mathcal{R}$: for every finiteness space, one can define a vector space (or semi-module) whose vectors are linear combinations with finitary support, and this space is endowed with a topology induced by the duality. In this setting, morphisms in the linear category correspond to linear continuous maps between these vector spaces and non-linear maps correspond to analytic maps for which there is a natural notion of differentiation. This construction provided the semantical motivation for differential linear logic and the syntactic notion of Taylor expansion which associates a formal sum of resource terms to a given term [11, 10]. Finiteness spaces were also used to characterize strongly normalizing terms in non-deterministic λ-calculus [25]. More recently, finiteness spaces were used in the theory of generalized power series rings and topological groupoids [5, 1].

This finiteness space construction yields a model of controlled non-determinism: the objects can be infinite dimensional vector spaces and the morphisms are series with possibly infinite support but whenever an explicit computation is made, the result is always finite. It corresponds to the operational property that a program always has a finite number of reduction paths for a given input and output.

1.3 Generalized species of structures

In this paper, we use species of structures to extend the finiteness construction on the relational model to a bicategorical setting. Species of structures were originally introduced by Joyal as a unified framework for the theory of generating series in enumerative combinatorics [21]. Fiore et al. then presented a generalized definition that both encompasses Joyal's species and constitutes a model of differential linear logic [13]. This model of generalized species is based on the bicategory of profunctors **Prof** and it can be considered as a generalization of the differential relational model **Rel**. It follows the line of research of categorifying λ-calculus models by replacing sets or preorders by richer categorical structures [7, 19]. Generalized species are also connected to the Girard's normal functor model [16] which was later extended by Hasegawa [18].

The exponential modality in the model of generalized species is based on the free symmetric monoidal completion for small categories which generalizes the finite multiset construction for the relational model. Morphisms in the co-Kleisli bicategory correspond to the notion of analytic functors which provide the series counterpart of generalized species [12].

1.4 Finiteness spaces with profunctors

In the original model of relational finiteness spaces, types are interpreted as pairs $A = (|A|, \mathscr{F}A)$ of countable set $|A|$ with a set of so-called finitary subsets $\mathscr{F}A \subseteq \mathcal{P}(|A|)$ satisfying $\mathscr{F}A = (\mathscr{F}A)^{\perp\perp}$. In our setting, the types will correspond to pairs $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$ of a locally finite category $|\mathbf{A}|$ equipped with a full subcategory of finite presheaves $\mathscr{F}\mathbf{A} \hookrightarrow [|\mathbf{A}|^{\mathrm{op}}, \mathbf{FinSet}]$ such that $\mathscr{F}\mathbf{A} \cong (\mathscr{F}\mathbf{A})^{\perp\perp}$.

The categorification of the orthogonality relation allows us to work in a better behaved setting of *focused orthogonalities* where forward preservation is equivalent to backward preservation for morphisms preserving the finiteness structure [20]. In our case, a morphism from $(|\mathbf{A}|, \mathscr{F}\mathbf{A})$ to $(|\mathbf{B}|, \mathscr{F}\mathbf{B})$ will be a finite profunctor $P : |\mathbf{A}| \nrightarrow_{\mathsf{f}} |\mathbf{B}|$ such that $(P)\mathscr{F}\mathbf{A} \hookrightarrow \mathscr{F}\mathbf{B}$ which will imply that $(P^{\perp})\mathscr{F}\mathbf{B}^{\perp} \hookrightarrow \mathscr{F}\mathbf{A}^{\perp}$. We follow the same pattern of the double-glueing construction for 1-categories to obtain a bicategory of finiteness spaces and profunctors between them where computations are enforced to be finite and show that all the differential linear logic constructions in **Prof** can be refined to our bicategory.

Notation

- For an integer $n \in \mathbb{N}$, we write $\underline{n}$ for the set $\{1, \ldots, n\}$.
- For a small category $\mathcal{A}$, we denote by $\widehat{\mathcal{A}}$ the presheaf category $[\mathcal{A}^{\mathrm{op}}, \mathbf{Set}]$ and write $\mathbf{y}_{\mathcal{A}} : \mathcal{A} \to \widehat{\mathcal{A}}$ for the Yoneda embedding.
- We denote by $\mathbf{1}$ the category with one object and one morphism and by $\mathbf{0}$ the empty category.
- We use $\cong$ for natural isomorphisms between functors or category isomorphisms and $\simeq$ for equivalences.

2 Relational Finiteness Spaces

The model of relational finiteness spaces is obtained from **Rel** via a glueing construction along hom-functors using the following orthogonality relation:

▶ **Definition 1.** *For a countable set S, subsets $x \in \mathbf{Rel}(1, S) \cong \mathcal{P}(S)$ and $x' \in \mathbf{Rel}(S, 1) \cong \mathcal{P}(S)$, we say that x and x' are* orthogonal *if $x \cap x'$ is a finite set and we denote it by $x \perp_S x'$.*

The idea is that morphisms in $\mathbf{Rel}(1, S)$ are thought of as closed programs of type S and morphisms in $\mathbf{Rel}(S, 1)$ correspond to counter-programs or environments. The orthogonality relation allows for more control on interactions between programs and environments as we require their interaction to always be finite even if the type S is infinite. For a subset $\mathscr{F} \subseteq \mathcal{P}(S)$, we define its *orthogonal* as $\mathscr{F}^{\perp} := \{x \in \mathcal{P}(S) \mid \forall x' \in \mathscr{F}, x \perp x'\} \subseteq \mathcal{P}(S)$. This orthogonality relation induces a Galois connection on $\mathcal{P}\mathcal{P}(S)$

$$\begin{tikzcd} \mathcal{P}\mathcal{P}(S) \arrow[r, bend left, "(-)^{\perp}"] \arrow[r, phantom, "\perp"] & \mathcal{P}\mathcal{P}(S) \arrow[l, bend left, "(-)^{\perp}"] \end{tikzcd}$$

where finiteness spaces, introduced below, are its fixpoints $\mathscr{F} = \mathscr{F}^{\perp\perp}$.

▶ **Definition 2.** *A* relational finiteness space *is a pair $A = (|A|, \mathscr{F}(A))$ where $|A|$ is a countable set and $\mathscr{F}(A)$ is a subset of $\mathcal{P}(|A|)$ satisfying $\mathscr{F}(A) = \mathscr{F}(A)^{\perp\perp}$.*

For any countable set S, the smallest finiteness structure is given by the set of finite subsets of S, $\mathcal{P}_{\mathrm{fin}}(S)$ whose orthogonal is given by the whole powerset $\mathcal{P}(S)$. For a relational finiteness space A, while elements of $\mathscr{F}(A)$ may be infinite subsets of $|A|$, they are called *finitary subsets* as they "behave" like finite sets in that $\mathscr{F}(A)$ is closed under inclusion (for $x \in \mathscr{F}(A)$, if $x' \subseteq x$, then $x' \in \mathscr{F}(A)$) and finite unions.

▶ **Definition 3.** *The category* **FinRel** *has objects finiteness spaces and morphisms are relations that preserve the finitary structure forward and backward. Explicitly, for finiteness spaces $A = (|A|, \mathscr{F}(A))$ and $B = (|B|, \mathscr{F}(B))$, a relation $R \subseteq |A| \times |B|$ induces two functions $R^{\star}$ and $R_{\star}$ given by $R_{\star} : x \mapsto \{b \in |B| \mid \exists a \in |A|, (a, b) \in R\}$ and $R^{\star} : y \mapsto \{a \in |A| \mid \exists b \in |B|, (a, b) \in R\}$. The relation R is said to be a* morphism of finiteness spaces *from A to B if for all $x \in \mathscr{F}(A)$, $R_{\star} \cdot x \in \mathscr{F}(B)$ and for all $y \in \mathscr{F}(B)^{\perp}$, $R^{\star} \cdot y \in \mathscr{F}(A)^{\perp}$.*

$$\begin{tikzcd} \mathcal{P}(|A|) \arrow[r, "R_\star"] & \mathcal{P}(|B|) \\ \mathscr{F}(A) \arrow[u, hook] \arrow[r, dotted] & \mathscr{F}(B) \arrow[u, hook] \end{tikzcd} \qquad \begin{tikzcd} \mathcal{P}(|B|) \arrow[r, "R^\star"] & \mathcal{P}(|A|) \\ \mathscr{F}(B)^\perp \arrow[u, hook] \arrow[r, dotted] & \mathscr{F}(A)^\perp \arrow[u, hook] \end{tikzcd}$$

Formally, the category **FinRel** is the tight orthogonality category in the sense of Hyland and Schalk obtained from the orthogonality relation defined above [20]. Ehrhard showed that the linear logic structure from **Rel** can be lifted to **FinRel** which constitutes a model of differential linear logic [10]. The morphisms in the co-Kleisli category of **FinRel** play the role of supports for power series for the second part of the construction: for a fixed field (or semi-ring) $\mathcal{R}$, we can define for every relational finiteness space $A = (|A|, \mathscr{F}(A))$, the following vector space (or semi-module): $\mathcal{R}\langle A\rangle := \{X \in \mathcal{R}^{|A|} \mid \mathrm{support}(X) \in \mathscr{F}(A)\}$. Ehrhard showed that $\mathcal{R}\langle A\rangle$ can be endowed with a topology $\mathcal{T}_A$ such that a matrix $M \in \mathcal{R}\langle A \multimap B\rangle$ corresponds to a linear continuous map $\mathcal{R}\langle A\rangle \to \mathcal{R}\langle B\rangle$ and a matrix $M \in \mathcal{R}\langle !A \multimap B\rangle$ corresponds to an analytic map $\mathcal{R}\langle A\rangle \to \mathcal{R}\langle B\rangle$ [9].

3 Profunctorial Finiteness Spaces

3.1 Orthogonality on bicategories

We work with a fragment of **Prof** where the objects are locally finite categories, it has the important consequence that finitely presentable presheaves are always finite presheaves as we will see below.

▶ **Definition 4.** *A small category $\mathcal{A}$ is said to be* locally finite *if it is enriched over finite sets i.e. for any objects $a, a' \in \mathcal{A}$, the homset $\mathcal{A}(a, a')$ is finite.*

▶ **Definition 5.** *For a category $\mathcal{A}$, a presheaf $X : \mathcal{A}^{op} \to$ **Set** is said to be* finite *if for all $a \in \mathcal{A}$, $X(a)$ is a finite set. We denote by $\widehat{\mathcal{A}}_{\mathrm{fin}} \hookrightarrow \widehat{\mathcal{A}}$ the full subcategory of finite presheaves. Note that the Yoneda embedding $\mathbf{y}_{\mathcal{A}}$ for a locally finite category $\mathcal{A}$ factors through the inclusion $\widehat{\mathcal{A}}_{\mathrm{fin}} \hookrightarrow \widehat{\mathcal{A}}$ by an embedding $\mathcal{A} \hookrightarrow \widehat{\mathcal{A}}_{\mathrm{fin}}$.*

For presheaf categories, finitely presentable objects can be characterized as presheaves that are isomorphic to a finite colimit of representables. For a locally finite category $\mathcal{A}$, since a finite colimit of finite presheaves is also a finite presheaf, there is an embedding from the subcategory of finitely presentable objects $\widehat{\mathcal{A}}_{\mathrm{fp}}$ to $\widehat{\mathcal{A}}_{\mathrm{fin}}$.

▶ **Definition 6.** *A profunctor $F : \mathcal{A} \nrightarrow \mathcal{B}$ between two small categories $\mathcal{A}$ and $\mathcal{B}$ is a functor $F : \mathcal{A} \times \mathcal{B}^{\mathrm{op}} \to$ **Set** or equivalently a functor $F : \mathcal{A} \to \widehat{\mathcal{B}}$. F is said to be a* finite profunctor *if it can be factored as a functor $F : \mathcal{A} \to \widehat{\mathcal{B}}_{\mathrm{fin}}$ through the embedding $\widehat{\mathcal{B}}_{\mathrm{fin}} \hookrightarrow \widehat{\mathcal{B}}$. In other words, for all $a \in \mathcal{A}$ and $b \in \mathcal{B}$, $F(a, b)$ is a finite set. A finite profunctor will be denoted by $F : \mathcal{A} \nrightarrow_{\mathsf{f}} \mathcal{B}$.*

The composite of two profunctors $F : \mathcal{A} \nrightarrow \mathcal{B}$ and $G : \mathcal{B} \nrightarrow \mathcal{C}$ is the profunctor $G \circ F : \mathcal{A} \nrightarrow \mathcal{C}$ given by the coend formula:

$$(a, c) \quad \mapsto \quad \int^{b \in \mathcal{B}} F(a, b) \times G(b, c) \cong \left(\sum_{b \in \mathcal{B}} F(a, b) \times G(b, c) \right) /\!\sim$$

where $\sim$ is the least equivalence relation such that $(b, F(a, f)(s), t) \sim (b', s, G(f, c)(t))$ for $s \in F(a, b')$, $t \in G(b, c)$ and $f : b \to b' \in \mathcal{B}$. Composition of profunctors is associative only up to natural isomorphisms which puts us in the setting of a bicategory [3]. Note that the composite of two finite profunctors between locally finite categories need not to be finite (since the sum above can be infinite if $\mathcal{B}$ has an infinite object set for example) but we will see how finiteness structures will enable us to make this notion compositional.

▶ **Definition 7.** *Let $\mathcal{A}$ be a locally finite category, $X : \mathcal{A}^{op} \to \mathbf{Set}$ a presheaf and $X' : \mathcal{A} \to \mathbf{Set}$ a copresheaf, we say that X and X' are* orthogonal *and write $X \perp_{\mathcal{A}} X'$ if the set $\langle X, X' \rangle := \int^{a\in\mathcal{A}} X(a) \times X'(a)$ is finite.*

In the bicategorical case, presheaves in $\widehat{\mathcal{A}}$ or equivalently profunctors $\mathbf{1} \nrightarrow \mathcal{A}$ (where $\mathbf{1}$ is the terminal category) are thought of as closed programs of type $\mathcal{A}$ and co-presheaves in $\widehat{\mathcal{A}^{\mathrm{op}}}$ or profunctors $\mathcal{A} \nrightarrow \mathbf{1}$ correspond to environments. In our setting, the interaction between a program $X : \mathcal{A}^{\mathrm{op}} \to \mathbf{Set}$ and an environment $X' : \mathcal{A} \to \mathbf{Set}$ corresponds to their composition in $\mathbf{Prof}$: $X' \circ X = \int^{a\in\mathcal{A}} X(a) \times X'(a)$.

Adding the orthogonality structure on categories allows us to work in a setting where we enforce this composite to always be finite. Note that the condition in Definition 7 becomes $X' \circ X \in \mathbf{FinSet} \hookrightarrow \mathbf{Set} \cong \mathbf{Prof}(\mathbf{1}, \mathbf{1})$. In the case of 1-categories, for $\mathcal{C}$ a model of linear logic with monoidal units $\mathbf{1}$ and $\perp$, and for $\perp\!\!\!\perp \subseteq \mathcal{C}(\mathbf{1}, \perp)$ a distinguished *pole*, if the orthogonality relation $\perp_c \hookrightarrow \mathcal{C}(\mathbf{1}, c) \times \mathcal{C}(c, \perp)$ is given by:

$$\perp_c \quad = \quad \{(x, x') \in \mathcal{C}(\mathbf{1}, c) \times \mathcal{C}(c, \perp) \mid x' \circ x \in \perp\!\!\!\perp\}$$

we say that the orthogonality is *focused* and it is one of the better behaved cases [20]. It implies in particular that for all $x \in \mathcal{C}(\mathbf{1}, c)$, $f \in \mathcal{C}(c, d)$ and $y \in \mathcal{C}(d, \perp)$, $f \circ x \perp_d y$ if and only if $x \perp_c y \circ f$. In the general case, a morphism preserving the orthogonality needs to preserve it forward and backward whereas in the focused case, forward preservation becomes equivalent to backward preservation which simplifies the proofs significantly since we do not have to prove both directions every time. Unlike the relational case, the orthogonality in the categorified setting becomes focused so that the two preservation conditions for relations of Definition 3 reduce to a single preservation condition for profunctors as we will see in Definition 14.

▶ **Lemma 8.** *For all $X : \mathbf{1} \nrightarrow_{\mathsf{f}} \mathcal{A}$, $Y : \mathcal{B} \nrightarrow_{\mathsf{f}} \mathbf{1}$ and $F : \mathcal{A} \nrightarrow_{\mathsf{f}} \mathcal{B}$, we have:*

$$F \circ X \perp_{\mathcal{B}} Y \qquad \Leftrightarrow \qquad X \perp_{\mathcal{A}} Y \circ F.$$

Proof. It follows from the fact that the sets $\langle F \circ X, Y \rangle$ and $\langle X, Y \circ F \rangle$ are both isomorphic to $\int^{a\in\mathcal{A}} \int^{b\in\mathcal{B}} F(a, b) \times X(a) \times Y(b)$. ◀

For a set A considered as a discrete category, a subset $x \subseteq A$ can be viewed as a presheaf $x : A^{\mathrm{op}} \to \mathbf{Set}$ (or a copresheaf $x : A \to \mathbf{Set}$) that maps $a \in A$ to the singleton $\{\star\}$ if $a \in x$ and to the empty set otherwise. Hence, for $x \subseteq A$ viewed as a presheaf and $x' \subseteq A$ viewed as a copresheaf, $x \cap x'$ is finite is equivalent to the set $\int^{a\in A} x(a) \times x'(a)$ being finite. This analogy provides the connection between the bicategorical case and the relational case.

▶ **Definition 9.** *For a subcategory $\mathcal{C} \hookrightarrow \widehat{\mathcal{A}}_{\mathrm{fin}}$, we denote by $\mathcal{C}^{\perp}$, the full subcategory of $\widehat{\mathcal{A}^{op}}_{\mathrm{fin}}$ of finite copresheaves X' such that for all $X \in \mathcal{C}$, $X' \perp_{\mathcal{A}} X$.*

Let $\mathbf{Sub}(\widehat{\mathcal{A}})$ be the poset of full subcategories of $\widehat{\mathcal{A}}$, the orthogonality relation induces a Galois connection:

$$\begin{tikzcd}
\mathbf{Sub}(\widehat{\mathcal{A}}) \arrow[r, bend left, "(-)^{\perp}"] \arrow[r, phantom, "\perp"] & \mathbf{Sub}(\widehat{\mathcal{A}^{\mathrm{op}}})^{\mathrm{op}} \arrow[l, bend left, "(-)^{\perp}"]
\end{tikzcd}$$

whose fixed points are full subcategories $\mathcal{C}$ verifiying $\mathcal{C}^{\perp\perp} \cong \mathcal{C}$.

▶ **Definition 10.** *A* finiteness structure *is a pair* $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$ *of a locally finite category* $|\mathbf{A}|$ *and a full subcategory* $\mathscr{F}\mathbf{A} \hookrightarrow \widehat{|\mathbf{A}|}_{\mathrm{fin}}$ *verifying* $\mathscr{F}\mathbf{A} \cong \mathscr{F}\mathbf{A}^{\perp\perp}$*.*

▶ **Lemma 11.** *For a finiteness structure* $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$*, the subcategory of finitely presentable objects* $\widehat{|\mathbf{A}|}_{\mathrm{fp}} \hookrightarrow \widehat{|\mathbf{A}|}_{\mathrm{fin}}$ *is always a full subcategory of* $\mathscr{F}\mathbf{A}$*.*

Proof. If X is finitely presentable, then X is isomorphic to a finite colimit of representables $X \cong \varinjlim_{i\in I} |\mathbf{A}|(-, a_i) : |\mathbf{A}|^{\mathrm{op}} \to \mathbf{Set}$. For any $X' \in (\mathscr{F}\mathbf{A})^{\perp}$,

$$\langle X, X' \rangle = \int^{a\in|\mathbf{A}|} X(a) \times X'(a) \cong \varinjlim_{i\in I} \int^{a\in|\mathbf{A}|} |\mathbf{A}|(a, a_i) \times X'(a) \cong \varinjlim_{i\in I} X'(a_i).$$

Since a finite colimit of finite sets is finite, we obtain that $X \perp_{\mathbf{A}} X'$ as desired. ◀

The minimal finiteness structure is $(|\mathbf{A}|, \widehat{|\mathbf{A}|}_{\mathrm{fp}})$ and its orthogonal is the maximal finiteness structure $(|\mathbf{A}|, \widehat{|\mathbf{A}|}_{\mathrm{fin}})$ so for any finiteness structure $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$, we have

$$(|\mathbf{A}|, \widehat{|\mathbf{A}|}_{\mathrm{fp}}) \longhookrightarrow \mathbf{A} \longhookrightarrow (|\mathbf{A}|, \widehat{|\mathbf{A}|}_{\mathrm{fin}}).$$

▶ **Lemma 12.** *If $\mathcal{A}$ is a finite category (both the object and morphism sets are finite), then there is a unique finiteness structure given by* $\widehat{\mathcal{A}}_{\mathrm{fin}}$*.*

Proof. By Lemma 11, it suffices to show that if $\mathcal{A}$ is finite, then any finite presheaf $X : \mathcal{A}^{op} \to \mathbf{FinSet}$ is finitely presentable. If $\mathcal{A}$ is finite, then the category of elements $\int X$ of X is finite as well and since $X \cong \varinjlim(\int X \to \mathcal{A} \to \widehat{\mathcal{A}})$, X is a finite colimit of representables and hence is finitely presentable. ◀

In the relational case, for a finiteness structure $A = (|A|, \mathscr{F}A)$, $\mathscr{F}A$ can be larger than $\mathcal{P}_{\mathrm{fin}}(|A|)$ but its elements "behave" like finite sets in the sense that $x \subseteq y \in \mathscr{F}(A)$ implies $x \in \mathscr{F}(A)$ and a finite union of finitary elements is finitary. In the categorical case, $\mathscr{F}(\mathbf{A})$ can be thought of as a category larger than $\widehat{|\mathbf{A}|}_{\mathrm{fp}}$ but its elements "behave" like finitely presentable elements as $\mathscr{F}(\mathbf{A})$ is closed under retractions and finite colimits.

▶ **Lemma 13.** *Let* $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}(\mathbf{A}))$ *be a finiteness structure, then the following two properties hold:*

1. *if X' is a retract of an element $X \in \mathscr{F}(\mathbf{A})$, then $X' \in \mathscr{F}(\mathbf{A})$;*
2. *$\mathscr{F}(\mathbf{A})$ is closed under finite colimits.*

Proof. Let $\alpha : X \Rightarrow X'$ be a retraction in $\widehat{|\mathbf{A}|}$. Since a retraction is an epimorphism and colimits in $\widehat{|\mathbf{A}|}$ are computed pointwise, for every $a \in |\mathbf{A}|$, $\alpha_a : X(a) \to X'(a)$ is a surjection. Hence, for every $Y \in \mathscr{F}(\mathbf{A})^{\perp}$,

$$\langle Y, X \rangle = \int^{a\in|\mathbf{A}|} Y(a) \times X(a) \quad \twoheadrightarrow \quad \int^{a\in|\mathbf{A}|} Y(a) \times X'(a) = \langle Y, X' \rangle$$

which implies that $\langle Y, X' \rangle$ is a finite set as well so that $X' \in \mathscr{F}(\mathbf{A})$. The second property follows from the fact that a finite colimit of finite sets is finite. ◀

▶ **Definition 14.** *Given two finiteness structures* $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$ *and* $\mathbf{B} = (|\mathbf{B}|, \mathscr{F}\mathbf{B})$*, a finite profunctor* $F : |\mathbf{A}| \nrightarrow_{\mathsf{f}} |\mathbf{B}|$ *is called a* finiteness profunctor *if* $\widehat{F} := \mathbf{Lan}_{\mathbf{y}_{|\mathbf{A}|}} F : \widehat{|\mathbf{A}|} \to \widehat{|\mathbf{B}|}$ *verifies* $\widehat{F}(\mathscr{F}\mathbf{A}) \hookrightarrow \mathscr{F}\mathbf{B}$ *i.e if there exists a functor* $\mathscr{F}\mathbf{A} \to \mathscr{F}\mathbf{B}$ *making the diagram below commute:*

$$\begin{tikzcd}
\widehat{|\mathbf{A}|} \arrow[r, "\widehat{F}"] & \widehat{|\mathbf{B}|} \\
\mathscr{F}\mathbf{A} \arrow[u, hook] \arrow[r, dotted] & \mathscr{F}\mathbf{B} \arrow[u, hook]
\end{tikzcd}$$

▶ **Lemma 15.** *Given two finiteness structures* $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$ *and* $\mathbf{B} = (|\mathbf{B}|, \mathscr{F}\mathbf{B})$*, a profunctor* $F : |\mathbf{A}| \nrightarrow_{\mathsf{f}} |\mathbf{B}|$ *is a finiteness profunctor* $\mathbf{A} \nrightarrow_{\mathsf{f}} \mathbf{B}$ *if and only if* $F^{\perp} : (|\mathbf{B}|^{op}, \mathscr{F}\mathbf{B}^{\perp}) \nrightarrow_{\mathsf{f}} (|\mathbf{A}|^{op}, \mathscr{F}\mathbf{A}^{\perp})$ *is also a finiteness profunctor.*

Proof. Direct consequence of Lemma 8. ◀

Since the categories $\mathbf{Prof}(\mathbf{1}, |\mathbf{A}|)$ and $\widehat{|\mathbf{A}|}$ are isomorphic, we will abuse notation and identify presheaves $X \in \widehat{|\mathbf{A}|}$ with profunctors $\mathbf{1} \nrightarrow |\mathbf{A}|$ and write $F \circ X$ instead of $\widehat{F}X$. Under this isomorphism, we can reformulate the condition of Definition 14 as follows: F is a finiteness profunctor if for all presheaves X in $\mathscr{F}\mathbf{A}$, $F \circ X$ is in $\mathscr{F}\mathbf{B}$. Likewise, using the isomorphism $\mathbf{Prof}(|\mathbf{B}|, \mathbf{1}) \cong \widehat{|\mathbf{B}|^{\mathrm{op}}}$, $F^{\perp}$ is a finiteness profunctor if for all copresheaves Y in $\mathscr{F}\mathbf{B}^{\perp}$, $Y \circ F$ is in $\mathscr{F}\mathbf{A}^{\perp}$.

▶ **Definition 16.** *Define* $\mathbf{FinProf}$ *to be the bicategory whose* 0*-cells are finiteness structures,* 1*-cells are finiteness profunctors as in Definition 14 and* 2*-cells are natural transformations between such profunctors.*

Proof. We show below that $\mathbf{FinProf}$ is indeed a bicategory.

Identity For a finiteness structure $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$, $\mathrm{id}_{|\mathbf{A}|} : |\mathbf{A}| \nrightarrow |\mathbf{A}|$ is a finite profunctor as $|\mathbf{A}|$ is a locally finite category. Since $\mathrm{id}_{|\mathbf{A}|}$ verifies $\widehat{id_{|\mathbf{A}|}} \cong id_{\widehat{|\mathbf{A}|}}$, it is a finiteness profunctor $\mathbf{A} \nrightarrow_{\mathsf{f}} \mathbf{A}$.

Composition Let $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A}), \mathbf{B} = (|\mathbf{B}|, \mathscr{F}\mathbf{B})$ and $\mathbf{C} = (|\mathbf{C}|, \mathscr{F}\mathbf{C})$ be finiteness structures and $F : \mathbf{A} \nrightarrow_{\mathsf{f}} \mathbf{B}$ and $G : \mathbf{B} \nrightarrow_{\mathsf{f}} \mathbf{C}$ be finiteness profunctors. It is clear that if $\widehat{F}(\mathscr{F}\mathbf{A}) \hookrightarrow \mathscr{F}\mathbf{B}$ and $\widehat{G}(\mathscr{F}\mathbf{B}) \hookrightarrow \mathscr{F}\mathbf{C}$, then $\widehat{G \circ F}(\mathscr{F}\mathbf{A}) \cong \widehat{G} \circ \widehat{F}(\mathscr{F}\mathbf{A}) \hookrightarrow \mathscr{F}\mathbf{C}$. It remains to show that $G \circ F$ is a finite profunctor. For all $a \in |\mathbf{A}|$ and $c \in |\mathbf{C}|$, we have

$$(G \circ F)(a, c) = \int^{b \in |\mathbf{B}|} F(a, b) \times G(b, c) \cong \widehat{G}(\widehat{F}(\mathbf{y}(a)))(c).$$

Since $\mathbf{y}(a) \in \mathscr{F}\mathbf{A}$, $\widehat{G}(\widehat{F}(\mathbf{y}(a))$ is an element of $\mathscr{F}\mathbf{C}$ so it is a finite presheaf, which implies that $\widehat{G}(\widehat{F}(\mathbf{y}(a)))(c)$ is finite as desired. ◀

We obtain as a corollary of Lemma 15 that the mapping $\mathbf{A} \mapsto \mathbf{A}^{\perp} := (|\mathbf{A}|^{\mathrm{op}}, \mathscr{F}\mathbf{A}^{\perp})$ can be extended to a full and faithful functor $\mathbf{FinProf}^{\mathrm{op}} \to \mathbf{FinProf}$.

▶ **Lemma 17.** *The forgetful functor* $\mathcal{U} : \mathbf{FinProf} \to \mathbf{Prof}$ *is locally fully faithful and injective on* 1*-cells. Explicitly, for finiteness structures* $\mathbf{A}$ *and* $\mathbf{B}$*, the induced functor* $\mathbf{FinProf}(\mathbf{A}, \mathbf{B}) \to \mathbf{Prof}(|\mathbf{A}|, |\mathbf{B}|)$ *is injective on objects and fully faithful.*

4 Linear Logic Structure

In this section, we prove that the differential linear logic structure in **Prof** can be lifted to **FinProf**. While the full definition of a bicategorical model of linear logic has yet to be spelled out, the standard 1-categorical constructions have canonical bicategorical analogues which we use. The proofs will make use of the lemma below that shows how certain families of adjoint equivalences needed for the linear logic structure can be lifted from **Prof** to **FinProf** using the fact that the forgetful functor is locally fully faithful.

▶ **Lemma 18.** *Let $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}$ be categories and $(L : \mathcal{A} \to \mathcal{B}, R : \mathcal{B} \to \mathcal{A}, \eta, \varepsilon)$ be an adjoint equivalence. Let $L' : \mathcal{C} \to \mathcal{D}$, $R' : \mathcal{D} \to \mathcal{C}$, $F : \mathcal{C} \to \mathcal{A}$ and $G : \mathcal{D} \to \mathcal{B}$ be functors such that F and G are fully faithful, $GL' = LF$ and $FR' = RG$. Then L' and R' are adjoint equivalent $L' \dashv R'$.*

$$\begin{tikzcd}
\mathcal{A} \arrow[r, bend left, "L"] \arrow[r, phantom, "\simeq\perp"] & \mathcal{B} \arrow[l, bend left, "R"] \\
\mathcal{C} \arrow[u, "F"] \arrow[r, bend left, "L'"] & \mathcal{D} \arrow[u, "G"'] \arrow[l, bend left, "R'"]
\end{tikzcd}$$

Proof. For objects $c \in \mathcal{C}$ and $d \in \mathcal{D}$, using the hypotheses above, we have:

$$\mathcal{C}(c, R'd) \cong \mathcal{A}(Fc, FR'(d)) = \mathcal{A}(Fc, RGd) \cong \mathcal{B}(LFc, Gd) = \mathcal{B}(GL'c, Gd) \cong \mathcal{D}(L'c, d)$$

which implies that $L' \dashv R'$.

For $c \in \mathcal{C}$, the component of the unit η' of the adjunction $L' \dashv R'$ is the morphism η'_c determined by $F(\eta'_c) = \eta_{F(c)}$. It is an isomorphism since F is fully faithful and hence conservative. We can show that the counit of the adjunction $L' \dashv R'$ is an isomorphism in a similar fashion. ◀

4.1 Additive structure

Similarly to the 1-categorical case, **FinProf** is endowed with a finite biproduct structure. For a family of categories $(\mathcal{A}_i)_{i\in I}$, we denote by $\&_i \mathcal{A}_i$ their coproduct in **Cat**. There is an isomorphism $\widehat{\&_i \mathcal{A}_i} \cong \prod_i \widehat{\mathcal{A}_i}$, so we will often identify a presheaf $Z \in \widehat{\&_i \mathcal{A}_i}$ with a tuple of presheaves $(Z_i)_{i\in I} \in \prod_i \widehat{\mathcal{A}_i}$.

▶ **Lemma 19.** *For a finite family of finiteness structures $(\mathbf{A}_i)_{i\in I}$, $\&_i \mathbf{A}_i := (\&_i |\mathbf{A}_i|, \prod_i \mathscr{F}\mathbf{A}_i)$ is a finiteness structure.*

Proof. It suffices to show that $(\prod_i \mathscr{F}\mathbf{A}_i)^\perp \cong \prod_i (\mathscr{F}\mathbf{A}_i)^\perp$. ◀

▶ **Definition 20.** *For a family of finiteness structures $(\mathbf{A}_i)_{i\in I}$, we define the finiteness structure $\oplus_i \mathbf{A}_i$ by $(\&_i |\mathbf{A}_i|, (\mathscr{F}(\&_i \mathbf{A}_i^\perp))^\perp)$.*

▶ **Lemma 21.** *The empty category $\mathbf{0}$ with its presheaf category $(\mathbf{0}, \widehat{\mathbf{0}})$ forms a finiteness structure that is the neutral for $\&$ and $\oplus$.*

▶ **Lemma 22.** *For a finite family of finiteness structures $(\mathbf{A}_i)_{i\in I}$, the profunctors $\pi_i : \&_i |\mathbf{A}_i| \nrightarrow |\mathbf{A}_i|$ and $\mathrm{inj}_i : |\mathbf{A}_i| \nrightarrow \&_i |\mathbf{A}_i|$ are finiteness profunctors $\&_i \mathbf{A}_i \nrightarrow_{\mathsf{f}} \mathbf{A}_i$ and $\mathbf{A}_i \nrightarrow_{\mathsf{f}} \oplus_i \mathbf{A}_i$ respectively. They induce adjoint equivalences:*

$$\mathbf{FinProf}(\mathbf{X}, \&_i \mathbf{A}_i) \simeq \prod_i \mathbf{FinProf}(\mathbf{X}, \mathbf{A}_i) \text{ and } \mathbf{FinProf}(\oplus_i \mathbf{A}_i, \mathbf{X}) \simeq \prod_i \mathbf{FinProf}(\mathbf{A}_i, \mathbf{X}).$$

Proof. The profunctors π_i and inj_i are given by $\pi_i : ((i, a_i), a) \mapsto |\mathbf{A}_i|(a, a_i)$ and $\mathrm{inj}_i : (a, (i, a_i)) \mapsto |\mathbf{A}_i|(a_i, a)$ so they are finite profunctors since the category $|\mathbf{A}_i|$ is locally finite. For $Z \in \mathscr{F}(\&_i \mathbf{A}_i)$ and $X \in \mathscr{F}\mathbf{A}_i^\perp$, $\langle \pi_i Z, X\rangle \cong \langle Z_i, X\rangle \in \mathbf{FinSet}$ which implies that $\pi_i \in \mathbf{FinProf}(\&_i \mathbf{A}_i, \mathbf{A}_i)$. Likewise, for X in $\mathscr{F}\mathbf{A}_i$ and $Z \in \mathscr{F}(\oplus_i \mathbf{A}_i)^\perp$, $\langle \mathrm{inj}_i X, Z\rangle \cong \langle X, Z_i\rangle \in \mathbf{FinSet}$ so that $\mathrm{inj}_i \in \mathbf{FinProf}(\mathbf{A}_i, \oplus_i \mathbf{A}_i)$.

Using Lemma 18, the adjoint equivalences above follow from the biproduct structure in **Prof** where we have adjoint equivalences $\mathbf{Prof}(|\mathbf{X}|, \&_i |\mathbf{A}_i|) \simeq \prod_i \mathbf{Prof}(|\mathbf{X}|, |\mathbf{A}_i|)$ and $\mathbf{Prof}(\&_i |\mathbf{A}_i|, |\mathbf{X}|) \simeq \prod_i \mathbf{Prof}(|\mathbf{A}_i|, |\mathbf{X}|)$. ◀

4.2 Star-Autonomous Structure

The bicategory **Prof** is symmetric monoidal with tensor product given by the cartesian product of categories $(\mathcal{A}, \mathcal{B}) \mapsto \mathcal{A} \times \mathcal{B}$ and monoidal unit $\mathbf{1}$. The duality functor $\mathcal{A} \mapsto \mathcal{A}^{\mathrm{op}}$ provides **Prof** with a compact closed structure. Adding the orthogonality structure allows for less degenerate model as the bicategory **FinProf** is now $*$-autonomous with dualizing object $\mathbf{1}$. To show that the symmetric monoidal structure in **Prof** lifts to **FinProf**, it suffices to prove that the tensor product lifts to a pseudo-functor $\mathbf{FinProf} \times \mathbf{FinProf} \to \mathbf{FinProf}$ and that the symmetry, associator and left and right unitors pseudo-natural transformations have components in **FinProf**.

For relational finiteness spaces, the tensor product of $A = (|A|, \mathscr{F}(A))$ and $B = (|B|, \mathscr{F}(B))$ is the smallest structure that contains all products $x \times y$ of subsets $x \in \mathscr{F}(A)$ and $y \in \mathscr{F}(B)$. Since the set $\{x \times y \mid x \in \mathscr{F}(A), y \in \mathscr{F}(B)\}$ is not necessarily closed under double orthogonality $A \otimes B$ is defined as $(|A| \times |B|, \{x \times y \mid x \in \mathscr{F}(A), y \in \mathscr{F}(B)\}^{\perp\perp})$. In the categorified case, the construction is similar, for finiteness structures $\mathbf{A}$ and $\mathbf{B}$, $\mathscr{F}(\mathbf{A} \otimes \mathbf{B})$ is the smallest finiteness structure containing all products $X \times Y$ for $X \in \mathscr{F}(\mathbf{A})$ and $Y \in \mathscr{F}(\mathbf{B})$ where $X \times Y : (|\mathbf{A}| \times |\mathbf{B}|)^{\mathrm{op}} \to \mathbf{Set}$ is the presheaf given by the pointwise product $(a, b) \mapsto X(a) \times Y(b)$.

▶ **Definition 23.** *For finiteness structures $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$ and $\mathbf{B} = (|\mathbf{B}|, \mathscr{F}\mathbf{B})$, their tensor product is defined as $\mathbf{A} \otimes \mathbf{B} := (|\mathbf{A}| \times |\mathbf{B}|, \mathscr{F}(\mathbf{A} \otimes \mathbf{B}))$ where $\mathscr{F}(\mathbf{A} \otimes \mathbf{B})$ is the full subcategory of $\widehat{|\mathbf{A}| \times |\mathbf{B}|}_{\mathrm{fin}}$ whose object set is given by $\{X \times Y \mid X \in \mathscr{F}\mathbf{A} \text{ and } Y \in \mathscr{F}\mathbf{B}\}^{\perp\perp}$.*

▶ **Lemma 24.** *For finiteness profunctors $F_1 : \mathbf{A}_1 \nrightarrow_{\mathsf{f}} \mathbf{B}_1$ and $F_2 : \mathbf{A}_2 \nrightarrow_{\mathsf{f}} \mathbf{B}_2$, the profunctor $F_1 \otimes F_2 : |\mathbf{A}_1| \times |\mathbf{A}_2| \nrightarrow |\mathbf{B}_1| \times |\mathbf{B}_2|$ given by $(F_1 \otimes F_2)((a_1, a_2), (b_1, b_2)) := F_1(a_1, b_1) \times F_2(a_2, b_2)$ is in $\mathbf{FinProf}(\mathbf{A}_1 \otimes \mathbf{A}_2, \mathbf{B}_1 \otimes \mathbf{B}_2)$.*

Proof. Using Lemma 15, we show that $(F_1 \otimes F_2)^\perp \mathscr{F}(\mathbf{B}_1 \otimes \mathbf{B}_2)^\perp \hookrightarrow \mathscr{F}(\mathbf{A}_1 \otimes \mathbf{A}_2)^\perp$. Let Z be in $\mathscr{F}(\mathbf{B}_1 \otimes \mathbf{B}_2)^\perp$ i.e. for all $Y_1 \in \mathscr{F}\mathbf{B}_1$ and $Y_2 \in \mathscr{F}\mathbf{B}_2$, $\langle Z, Y_1 \times Y_2\rangle \in \mathbf{FinSet}$. $(F_1 \otimes F_2)^\perp(Z) \in \mathscr{F}(\mathbf{A}_1 \otimes \mathbf{A}_2)^\perp$ is equivalent to:

$$\begin{aligned} &\forall X_1 \in \mathscr{F}\mathbf{A}_1, \forall X_2 \in \mathscr{F}\mathbf{A}_2, \langle (F_1 \otimes F_2)^\perp(Z), X_1 \times X_2\rangle \in \mathbf{FinSet} \\ \Leftrightarrow &\forall X_1 \in \mathscr{F}\mathbf{A}_1, \forall X_2 \in \mathscr{F}\mathbf{A}_2, \langle Z, (F_1 X_1) \times (F_2 X_2)\rangle \in \mathbf{FinSet} \end{aligned}$$

Since $F_1 X_1$ is in $\mathscr{F}\mathbf{B}_1$ and $F_2 X_2$ is in $\mathscr{F}\mathbf{B}_2$, we obtain the desired result. ◀

▶ **Lemma 25.** $(\mathbf{1}, \mathbf{FinSet})$ *is the tensor unit.*

Proof. Let $\mathbf{A}$ be a finiteness structure, we show that $\mathscr{F}(\mathbf{A})^\perp \cong \mathscr{F}(\mathbf{A} \otimes \mathbf{1})^\perp \cong \mathscr{F}(\mathbf{1} \otimes \mathbf{A})^\perp$ so that the components of the left unitor $l_{|\mathbf{A}|} : |\mathbf{A}| \times |\mathbf{1}| \nrightarrow |\mathbf{A}|$ and right unitor $r_{|\mathbf{A}|} : |\mathbf{1}| \times |\mathbf{A}| \nrightarrow |\mathbf{A}|$ are in **FinProf**. Let $Y \in \mathscr{F}(\mathbf{A})^\perp$, $X \in \mathscr{F}(\mathbf{A})$ and $S \in \mathbf{FinSet}$. We have $\langle Y, X \times S\rangle \in \mathbf{FinSet} \Leftrightarrow \langle Y \times S, X\rangle \in \mathbf{FinSet}$. Since $\mathscr{F}(\mathbf{A})^\perp$ is closed under finite colimits, $Y \times S$ is in $\mathscr{F}(\mathbf{A})^\perp$ which implies the desired result. Now, for $Y \in \mathscr{F}(\mathbf{A} \otimes \mathbf{1})^\perp$ and $X \in \mathscr{F}(\mathbf{A})$, $\langle Y, X\rangle \cong \langle Y, X \times \{*\}\rangle \in \mathbf{FinSet}$ so that $Y \in \mathscr{F}(\mathbf{A})^\perp$ as desired. The proof for $\mathscr{F}(\mathbf{A})^\perp \cong \mathscr{F}(\mathbf{1} \otimes \mathbf{A})^\perp$ is similar. ◀

▶ **Lemma 26.** *For finiteness structures $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$ and $\mathbf{B} = (|\mathbf{B}|, \mathscr{F}\mathbf{B})$, the categories $\mathscr{F}(\mathbf{A} \otimes \mathbf{B})$ and $\mathscr{F}(\mathbf{B} \otimes \mathbf{A})$ are isomorphic which implies that the component of the symmetry $\sigma_{|\mathbf{A}|,|\mathbf{B}|} : |\mathbf{A}| \times |\mathbf{B}| \nrightarrow |\mathbf{B}| \times |\mathbf{A}|$ is in $\mathbf{FinProf}(\mathbf{A} \otimes \mathbf{B}, \mathbf{B} \otimes \mathbf{A})$.*

Proof. Immediate. ◀

Showing that the associator has components in **FinProf** is difficult to prove directly so we make use of the duality between the tensor and the internal hom to do it.

▶ **Lemma 27.** *For finiteness structures $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$ and $\mathbf{B} = (|\mathbf{B}|, \mathscr{F}\mathbf{B})$, define the finiteness structure $\mathbf{A} \multimap \mathbf{B}$ as $(|\mathbf{A}|^{\mathrm{op}} \times |\mathbf{B}|, \mathscr{F}(\mathbf{A} \multimap \mathbf{B}))$ where $\mathscr{F}(\mathbf{A} \multimap \mathbf{B})$ is the full subcategory of finite presheaves $\widehat{|\mathbf{A}|^{op} \times |\mathbf{B}|}_{\mathrm{fin}}$ that verify Definition 14.*

Proof. We prove that $\mathbf{A} \multimap \mathbf{B}$ is indeed a finiteness structure. We first show that for $X \in \mathscr{F}\mathbf{A}$ and $Y' \in \mathscr{F}\mathbf{B}^\perp$, $X \times Y' \in \mathscr{F}(\mathbf{A} \multimap \mathbf{B})^\perp$. Indeed, for $F \in \mathscr{F}(\mathbf{A} \multimap \mathbf{B})$, we have:

$$\langle X \times Y', F\rangle = \int^{a \in |\mathbf{A}|, b \in |\mathbf{B}|} X(a) \times Y'(b) \times F(a,b) \cong \langle Y', FX\rangle \in \mathbf{FinSet}.$$

Now, let $W \in \mathscr{F}(\mathbf{A} \multimap \mathbf{B})^{\perp\perp}$, we want to show that $W \in \mathscr{F}(\mathbf{A} \multimap \mathbf{B})$, i.e. that for all $X \in \mathscr{F}\mathbf{A}$, $WX \in \mathscr{F}\mathbf{B}$. Let $Y' \in \mathscr{F}\mathbf{B}^\perp$, $\langle Y', WX\rangle \cong \langle X \times Y', W\rangle \in \mathbf{FinSet}$ by the previous remark. ◀

▶ **Lemma 28.** *For finiteness structures $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$ and $\mathbf{B} = (|\mathbf{B}|, \mathscr{F}\mathbf{B})$, the categories $\mathscr{F}(\mathbf{A} \otimes \mathbf{B})$ and $\mathscr{F}(\mathbf{A} \multimap \mathbf{B}^\perp)^\perp$ are isomorphic.*

Proof. We prove that $\mathscr{F}(\mathbf{A} \otimes \mathbf{B})^\perp \cong \mathscr{F}(\mathbf{A} \multimap \mathbf{B}^\perp)$. Let $F : \mathbf{A} \nrightarrow \mathbf{B}^{op}$, we have:

$$\begin{aligned} F \in \mathscr{F}(\mathbf{A} \otimes \mathbf{B})^\perp &\Leftrightarrow \forall X \in \mathscr{F}(\mathbf{A}), \forall Y \in \mathscr{F}(\mathbf{B}) \langle F, X \times Y\rangle \in \mathbf{FinSet} \\ &\Leftrightarrow \forall X \in \mathscr{F}(\mathbf{A}), \forall Y \in \mathscr{F}(\mathbf{B}) \langle FX, Y\rangle \in \mathbf{FinSet} \\ &\Leftrightarrow \forall X \in \mathscr{F}(\mathbf{A}), FX \in \mathscr{F}(\mathbf{B})^\perp \Leftrightarrow F \in \mathscr{F}(\mathbf{A} \multimap \mathbf{B}^\perp) \end{aligned}$$ ◀

▶ **Lemma 29.** *For finiteness structures $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$, $\mathbf{B} = (|\mathbf{B}|, \mathscr{F}\mathbf{B})$ and $\mathbf{C} = (|\mathbf{C}|, \mathscr{F}\mathbf{C})$, the categories $\mathscr{F}((\mathbf{A} \otimes \mathbf{B}) \multimap \mathbf{C})$ and $\mathscr{F}(\mathbf{A} \multimap (\mathbf{B} \multimap \mathbf{C}))$ are isomorphic.*

Proof. Let $F : |\mathbf{A}| \times |\mathbf{B}| \nrightarrow_{\mathsf{f}} |\mathbf{C}|$ be in $\mathscr{F}((\mathbf{A} \otimes \mathbf{B}) \multimap \mathbf{C})$ and denote by $\overline{F} : |\mathbf{A}| \nrightarrow_{\mathsf{f}} |\mathbf{B}|^{\mathrm{op}} \times |\mathbf{C}|$ the corresponding profunctor obtained from the isomorphism $\mathbf{Prof}(|\mathbf{A}| \times |\mathbf{B}|, |\mathbf{C}|) \cong \mathbf{Prof}(|\mathbf{A}|, |\mathbf{B}|^{\mathrm{op}} \times |\mathbf{C}|)$. Let $X \in \mathscr{F}(\mathbf{A})$, we want to show that $\overline{F}X$ is in $\mathscr{F}(\mathbf{B} \multimap \mathbf{C})$, i.e. for all $Y \in \mathscr{F}(\mathbf{B})$, $\overline{F}(X)(Y) \in \mathscr{F}(\mathbf{C})$. We have that $X \times Y$ is in $\mathscr{F}(\mathbf{A} \otimes \mathbf{B})$ so that $F \circ (X \times Y) \cong \overline{F}(X)(Y)$ is in $\mathscr{F}(\mathbf{C})$.

For the other direction, let $G : |\mathbf{A}| \nrightarrow_{\mathsf{f}} |\mathbf{B}|^{\mathrm{op}} \times |\mathbf{C}|$ be in $\mathscr{F}(\mathbf{A} \multimap (\mathbf{B} \multimap \mathbf{C}))$ and denote by $\overline{G}$ the corresponding profunctor in $\mathbf{Prof}(|\mathbf{A}| \times |\mathbf{B}|, |\mathbf{C}|)$. We show that $\overline{G}^\perp \in \mathscr{F}(\mathbf{C}^\perp \multimap (\mathbf{A} \otimes \mathbf{B})^\perp)$. Let $Z \in \mathscr{F}(\mathbf{C})^\perp$, we want $\overline{G}^\perp Z \in \mathscr{F}(\mathbf{A} \otimes \mathbf{B})^\perp$ i.e. for all $X \in \mathscr{F}\mathbf{A}$ and $Y \in \mathscr{F}\mathbf{B}$, $\langle \overline{G}^\perp Z, X \times Y\rangle \in \mathbf{FinSet}$. Since $\langle \overline{G}^\perp Z, X \times Y\rangle \cong \langle G(X)(Y), Z\rangle$, we obtain the desired result. ◀

▶ **Corollary 30.** *For finiteness structures* $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$*,* $\mathbf{B} = (|\mathbf{B}|, \mathscr{F}\mathbf{B})$ *and* $\mathbf{C} = (|\mathbf{C}|, \mathscr{F}\mathbf{C})$*, the component of the associator* $\alpha_{|\mathbf{A}|,|\mathbf{B}|,|\mathbf{C}|} : (|\mathbf{A}| \times |\mathbf{B}|) \times |\mathbf{C}| \nrightarrow |\mathbf{A}| \times (|\mathbf{B}| \times |\mathbf{C}|)$ *given by:*

$$((a_1, b_1, c_1), (a_2, b_2, c_2)) \mapsto |\mathbf{A}|(a_2, a_1) \times |\mathbf{B}|(b_2, b_1) \times |\mathbf{C}|(c_2, c_1)$$

is a finiteness profunctor in $\mathbf{FinProf}((\mathbf{A} \otimes \mathbf{B}) \otimes \mathbf{C}, \mathbf{A} \otimes (\mathbf{B} \otimes \mathbf{C}))$*.*

Proof. It suffices to show that the categories $\mathscr{F}((\mathbf{A} \otimes \mathbf{B}) \otimes \mathbf{C})$ and $\mathscr{F}(\mathbf{A} \otimes (\mathbf{B} \otimes \mathbf{C}))$ are isomorphic. By Lemmas 28 and 29, we have

$$\begin{aligned}\mathscr{F}((\mathbf{A} \otimes \mathbf{B}) \otimes \mathbf{C}) &\cong \mathscr{F}((\mathbf{A} \otimes \mathbf{B}) \multimap \mathbf{C}^\perp)^\perp \cong \mathscr{F}(\mathbf{A} \multimap (\mathbf{B} \multimap \mathbf{C}^\perp))^\perp \\ &\cong \mathscr{F}(\mathbf{A} \multimap (\mathbf{B} \multimap \mathbf{C}^\perp)^{\perp\perp})^\perp \cong \mathscr{F}(\mathbf{A} \otimes (\mathbf{B} \otimes \mathbf{C})).\end{aligned}$$ ◀

A symmetric monoidal bicategory $\mathscr{B}$ is $*$-autonomous if there exists a full and faithful functor $(-)^* : \mathscr{B}^{\mathrm{op}} \to \mathscr{B}$ verifying $\mathbf{A} \simeq \mathbf{A}^{**}$ and for every objects $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$, a pseudo-natural family of adjoint equivalences $\mathscr{B}(\mathbf{A} \otimes \mathbf{B}, \mathbf{C}^*) \simeq \mathscr{B}(\mathbf{A}, (\mathbf{B} \otimes \mathbf{C})^*)$.

▶ **Proposition 31.** $\mathbf{FinProf}$ *a* $\star$*-autonomous bicategory.*

Proof. The duality $(-)^\perp : \mathbf{A} \mapsto \mathbf{A}^\perp = (|\mathbf{A}|^{\mathrm{op}}, \mathscr{F}\mathbf{A}^\perp)$ induces a full and faithful functor by Lemma 15. For finiteness structures $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$, $\mathbf{B} = (|\mathbf{B}|, \mathscr{F}\mathbf{B})$ and $\mathbf{C} = (|\mathbf{C}|, \mathscr{F}\mathbf{C})$, by Lemma 18, there is a pseudo-natural family of adjoint equivalences $\mathbf{FinProf}(\mathbf{A} \otimes \mathbf{B}, \mathbf{C}^\perp) \simeq \mathbf{FinProf}(\mathbf{A}, (\mathbf{B} \otimes \mathbf{C})^\perp)$. ◀

The interpretation of the $⅋$ connective is defined by dualizing the tensor $\mathbf{A} ⅋ \mathbf{B} = (\mathbf{A}^\perp \otimes \mathbf{B}^\perp)^\perp$. In the compact closed bicategory $\mathbf{Prof}$, the two connectives have the same interpretation whereas in $\mathbf{FinProf}$, adding the orthogonality eliminates this degeneracy. The inclusion $\mathscr{F}(\mathbf{A} \otimes \mathbf{B}) \hookrightarrow \mathscr{F}(\mathbf{A} ⅋ \mathbf{B})$ always hold which implies that we can interpret the mix rule in $\mathbf{FinProf}$. It can be derived from the set inclusion

$$\{X \times Y \mid X \in \mathscr{F}\mathbf{A}^\perp \text{ and } Y \in \mathscr{F}\mathbf{B}^\perp\} \hookrightarrow \{X \times Y \mid X \in \mathscr{F}\mathbf{A} \text{ and } Y \in \mathscr{F}\mathbf{B}\}^\perp$$

and the fact that $\mathscr{F}(\mathbf{A} ⅋ \mathbf{B})$ has object set $\{X \times Y \mid X \in \mathscr{F}(\mathbf{A})^\perp \text{ and } Y \in \mathscr{F}(\mathbf{B})^\perp\}^\perp$.

The other inclusion does not hold in general: consider the presheaf $P : ((!\mathbf{1})^{\mathrm{op}} \times !\mathbf{1})^{\mathrm{op}} \to \mathbf{Set}$ given by $(n, m) \mapsto !\mathbf{1}(m, n)$ corresponding to the identity profunctor $!\mathbf{1} \nrightarrow_{\mathsf{f}} !\mathbf{1}$. P is in $\mathscr{F}(!\mathbf{1} \multimap !\mathbf{1}) \cong \mathscr{F}((!\mathbf{1})^\perp ⅋ !\mathbf{1})$ but it is not in $\mathscr{F}((!\mathbf{1})^\perp \otimes !\mathbf{1})$. Indeed, let $Q : (!\mathbf{1})^{\mathrm{op}} \times !\mathbf{1}) \to \mathbf{Set}$ be dually given by $(n, m) \mapsto !\mathbf{1}(n, m)$, it verifies that for all $X \in \mathscr{F}(!\mathbf{1})^\perp$ and $Y \in \mathscr{F}(!\mathbf{1})$,

$$\langle X \times Y, Q \rangle = \int^{n,m} X(n) \times Y(m) \times !\mathbf{1}(n, m) \cong \langle X, Y \rangle \in \mathbf{FinSet}$$

which implies that Q is in $\mathscr{F}((!\mathbf{1})^\perp \otimes !\mathbf{1})^\perp$. However, $\langle P, Q \rangle = \int^{n,m} !\mathbf{1}(m, n) \times !\mathbf{1}(n, m) \cong \int^n !\mathbf{1}(n, n) \notin \mathbf{FinSet}$.

4.3 Exponential structure

The exponential modality in the setting of generalized species presented by Fiore et al. relies on the free symmetric strict monoidal completion construction for a small category.

▶ **Definition 32.** *For a small category* $\mathcal{A}$*, define* $!\mathcal{A}$ *as the category whose objects are finite sequences* $\langle a_1, \dots, a_n \rangle$ *of objects of* $\mathcal{A}$ *and a morphism* f *between two sequences* $\langle a_1, \dots, a_n \rangle$ *and* $\langle b_1, \dots b_n \rangle$ *consists of a pair* $(\sigma, (f_i)_{i \in \underline{n}})$ *where* σ *is a permutation in the symmetric group* $\mathfrak{S}_n$ *and* $(f_i : a_i \to b_{\sigma(i)})_{i \in \underline{n}}$ *is a sequence of morphisms in* $\mathcal{A}$*.*

The category $!\mathcal{A}$ described above is symmetric monoidal with tensor product $\otimes : (u, v) \mapsto u \otimes v$ given by the concatenation of sequences and unit the empty sequence. This construction induces a 2-monad on **Cat** which lifts to a pseudo-monad on **Prof** [14]. By dualization, one obtains a pseudo-comonad on **Prof** where the counit der and the comultiplication dig have the following components:

$$\begin{aligned} \mathrm{der}_{\mathcal{A}} &: !\mathcal{A} \nrightarrow \mathcal{A} & \mathrm{dig}_{\mathcal{A}} &: !\mathcal{A} \nrightarrow !!\mathcal{A} \\ (u, a) &\mapsto !\mathcal{A}(\langle a \rangle, u) & (u, \langle u_1, \dots, u_n \rangle) &\mapsto !\mathcal{A}(u_1 \otimes \cdots \otimes u_n, u) \end{aligned}$$

For a profunctor $P : \mathcal{A} \nrightarrow \mathcal{B}$, the action of the pseudo-comonad is given by

$$!P : (u, v) \mapsto \sum_{\sigma \in \mathfrak{S}_n} \prod_{i=1}^{n} P(u_i, v_{\sigma(i)})$$

if $u \in !\mathcal{A}$ and $v \in !\mathcal{B}$ are sequences of length n and $!P : (u, v) \mapsto \emptyset$ if u and v have different lengths. Generalized species correspond to the 1-cells in the co-Kleisli bicategory $\mathbf{Prof}_!$. For a species $F : !\mathcal{A} \nrightarrow \mathcal{B}$, its comonadic lifting $F^! : !\mathcal{A} \nrightarrow !\mathcal{B}$ is given by $(!F) \circ \mathrm{dig}_{\mathcal{A}}$. The composite of two species $G : !\mathcal{B} \nrightarrow \mathcal{C}$ and $F : !\mathcal{A} \nrightarrow \mathcal{B}$ in $\mathbf{Prof}_!$ is then given by $G \circ F^! : !\mathcal{A} \nrightarrow \mathcal{C}$.

We show in this section that the comonadic structure described above can be refined to the setting of finiteness structures.

▶ **Definition 33.** *For a finiteness structure $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$, we define $!(\mathbf{A}, \mathscr{F}\mathbf{A}) := (!\,|\mathbf{A}|, \mathscr{F}!\mathbf{A})$ where $\mathscr{F}!\mathbf{A}$ is the full subcategory of $\widehat{!\,|\mathbf{A}|}_{\mathrm{fin}}$ with object set $\{X^! \mid X \in \mathscr{F}\mathbf{A}\}^{\perp\perp}$.*

Note that for a presheaf $X : |\mathbf{A}|^{\mathrm{op}} \to \mathbf{Set}$ (seen as a species $!\,\mathbf{0} \nrightarrow |\mathbf{A}|$), its lifting $X^! : (!\mathbf{A})^{\mathrm{op}} \to \mathbf{Set}$ is given by $\langle a_1, \dots, a_n \rangle \mapsto !X \circ \mathrm{dig}_{\mathbf{0}}(\langle a_1, \dots, a_n \rangle) \cong \prod_{i \in \underline{n}} X(a_i)$. In particular, if X is a finite presheaf, then so is $X^!$.

Joyal presented the notion of analytic functor as the Taylor series counterpart of combinatorial species [22]. Fiore extended Joyal's results in the setting of generalized species and showed that there is a biequivalence between the bicategory of generalized species (restricted to groupoids) and the 2-category of analytic functors [12]. For small categories $\mathcal{A}$ and $\mathcal{B}$, a functor $P : \widehat{\mathcal{A}} \to \widehat{\mathcal{B}}$ is said to be *analytic* if there exists a generalized species $F : !\mathcal{A} \nrightarrow \mathcal{B}$ such that P is isomorphic to $\mathbf{Lan}_{s_{\mathcal{A}}} F$ (denoted by $\widetilde{F}$):

$$\begin{tikzcd} !\mathcal{A} \arrow[rr, "F"] \arrow[dr, "s_{\mathcal{A}}"'] & \arrow[d, phantom, "\Downarrow"] & \widehat{\mathcal{B}} \\ & \widehat{\mathcal{A}} \arrow[ur, dotted, "\mathbf{Lan}_{s_{\mathcal{A}}} F = \widetilde{F}"'] & \end{tikzcd}$$

where $s_{\mathcal{A}} : !\mathcal{A} \to \widehat{\mathcal{A}}$ is the functor that maps a sequence $\langle a_1, \dots a_n \rangle$ in $!\mathcal{A}$ to the presheaf $\sum_{i=1}^{n} \mathbf{y}_{\mathcal{A}}(a_i)$ in $\widehat{\mathcal{A}}$ so that $\widetilde{F}$ is given by $X \mapsto \int^{u \in !\mathcal{A}} F(u) \times \widehat{\mathcal{A}}(s_{\mathcal{A}}(u), X) \cong \int^{u \in !\mathcal{A}} F(u) \times X^!(u)$.

▶ **Lemma 34.** *For finiteness structures $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$ and $\mathbf{B} = (|\mathbf{B}|, \mathscr{F}\mathbf{B})$, a species $F : |!\mathbf{A}| \nrightarrow_{\mathsf{f}} |\mathbf{B}|$ is in $\mathscr{F}(!\mathbf{A} \multimap \mathbf{B})$ (viewed as a finite presheaf $(|!\mathbf{A}|^{\mathrm{op}} \times |\mathbf{B}|)^{\mathrm{op}} \to \mathbf{Set}$) if and only if for all $X \in \mathscr{F}(\mathbf{A})$, $\widetilde{F}X$ is in $\mathscr{F}(\mathbf{B})$.*

Proof. Assume that F is in $\mathscr{F}(!\mathbf{A} \multimap \mathbf{B})$ and let X be in $\mathscr{F}(\mathbf{A})$. Since $X^!$ is in $\mathscr{F}(!\mathbf{A})$, we have that $FX^! \cong \widetilde{F}X \in \mathscr{F}(\mathbf{B})$.

For the other direction, it suffices to show that if for all $X \in \mathscr{F}(\mathbf{A})$, $FX^!$ is in $\mathscr{F}(\mathbf{B})$, then $F^{\perp}(\mathscr{F}(\mathbf{B})^{\perp}) \hookrightarrow (\mathscr{F}(!\mathbf{A})^{\perp})$. Let Y be in $(\mathscr{F}\mathbf{B})^{\perp}$ and $X \in \mathscr{F}\mathbf{A}$, since $\langle F^{\perp}Y, X^! \rangle \cong \langle FX^!, Y \rangle \in \mathbf{FinSet}$, we obtain the desired result. ◀

We obtain as a corollary that for a finiteness structure $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$, $(\mathscr{F}!\mathbf{A})^\perp$ is isomorphic to the full subcategory of finite copresheaves $P : |!\mathbf{A}| \to \mathbf{Set}$ (or equivalently finite profunctors $|!\mathbf{A}| \nrightarrow_{\mathsf{f}} \mathbf{1}$) such that $\widetilde{P}(\mathscr{F}\mathbf{A}) \hookrightarrow \mathbf{FinSet}$.

▶ **Example 35.** In particular, $\mathscr{F}(!\,\mathbf{1})^\perp$ is isomorphic to species whose analytic functor maps finite sets to finite sets. In other words, $F : !\,\mathbf{1} \to \mathbf{Set}$ must verify that for all $S \in \mathbf{FinSet}$, $\sum\limits_{n\in\mathbb{N}} F(n) \times_{\mathfrak{S}_n} S^n$ is finite.

Similarly to relational finiteness spaces, we can see here that the fixpoint operator cannot be interpreted in **FinProf**. Indeed, consider the species of binary trees $B : !\,\mathbf{1} \nrightarrow \mathbf{1}$, it is a solution of the fixpoint equation $B = 1 + X \cdot B^2$ where $1 : !\,\mathbf{1} \nrightarrow \mathbf{1}$ is the species $(u, \star) \mapsto !\,\mathbf{1}(\langle\rangle, u)$ whose analytic functor $\mathbf{Set} \to \mathbf{Set}$ is the constant $S \mapsto \{\star\}$ and $X : !\,\mathbf{1} \nrightarrow \mathbf{1}$ is the species $(u, \star) \mapsto !\,\mathbf{1}(\langle\star\rangle, u)$ whose analytic functor $\mathbf{Set} \to \mathbf{Set}$ is the identity $S \mapsto S$ (see [4] for more details). Both 1 and X are finiteness species since their analytic functors restrict to $\mathbf{FinSet} \to \mathbf{FinSet}$. The species of binary trees however has analytic functor $\mathbf{Set} \to \mathbf{Set}$ given by $S \mapsto \sum_{n\in\mathbb{N}} C_n \times S^n$ where C_n is the nth Catalan number so this functor can not be restricted as a functor $\mathbf{FinSet} \to \mathbf{FinSet}$.

▶ **Lemma 36.** *For finiteness structures $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$ and $\mathbf{B} = (|\mathbf{B}|, \mathscr{F}\mathbf{B})$, if $F : \mathbf{A} \nrightarrow_{\mathsf{f}} \mathbf{B}$ is a finiteness profunctor, then $!F : |!\mathbf{A}| \nrightarrow |!\mathbf{B}|$ is in $\mathbf{FinProf}(!\mathbf{A}, !\mathbf{B})$.*

Proof. We show that $(!F)(\mathscr{F}!\mathbf{B}^\perp) \hookrightarrow \mathscr{F}!\mathbf{A}^\perp$. Let P be in $\mathscr{F}!\mathbf{B}^\perp$, i.e. for all Y in $\mathscr{F}\mathbf{B}$, $\widetilde{P}Y$ is in **FinSet**.

$$\begin{aligned}
(!F)(P) \in \mathscr{F}!\mathbf{A}^\perp &\Leftrightarrow \forall X \in \mathscr{F}\mathbf{A}, \int^{u\in|!\mathbf{A}|, v\in|!\mathbf{B}|} !F(u,v) \times P(v) \times X^!(u) \in \mathbf{FinSet} \\
&\Leftrightarrow \forall X \in \mathscr{F}\mathbf{A}, \int^{v\in|!\mathbf{B}|} P(v) \times (!F \circ X^!)(v) \in \mathbf{FinSet} \\
&\Leftrightarrow \forall X \in \mathscr{F}\mathbf{A}, \int^{v\in|!\mathbf{B}|} P(v) \times (F \circ X)^!(v) \in \mathbf{FinSet}
\end{aligned}$$

Since FX is in $\mathscr{F}\mathbf{B}$, $(FX)^! \in \mathscr{F}!\mathbf{B}$ which implies the desired result. ◀

We now show that the pseudo-comonad structure in **Prof** can be restricted to **FinProf**.

▶ **Lemma 37.** *For a finiteness structure $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$, the component of the counit pseudo-natural transformation $\mathrm{der}_{|\mathbf{A}|} : |!\mathbf{A}| \nrightarrow |\mathbf{A}|$ is in $\mathbf{FinProf}(!\mathbf{A}, \mathbf{A})$.*

Proof. Since $!\,|\mathbf{A}|$ is locally finite, $\mathrm{der}_{|\mathbf{A}|}$ is a finite profunctor. By Lemma 15, it remains to show that $\mathrm{der}^\perp_{|\mathbf{A}|}((\mathscr{F}\mathbf{A})^\perp) \hookrightarrow (\mathscr{F}!\mathbf{A})^\perp$ i.e. that for all $X' \in (\mathscr{F}\mathbf{A})^\perp$ and $X \in \mathscr{F}\mathbf{A}$, $(\mathrm{der}^\perp_{|\mathbf{A}|})X' \perp X^!$.

$$\begin{aligned}
&\langle(\mathrm{der}^\perp_{|\mathbf{A}|})X', X^!\rangle = \int^{u\in!|\mathbf{A}|} X^!(u) \times \int^{a\in|\mathbf{A}|} !\,|\mathbf{A}|\,(\langle a\rangle, u) \times X'(a) \\
&\cong \int^{a,a'\in|\mathbf{A}|} X(a') \times |\mathbf{A}|\,(a,a') \times X'(a) \cong \int^{a\in|\mathbf{A}|} X(a) \times X'(a) \in \mathbf{FinSet}
\end{aligned}$$ ◀

▶ **Lemma 38.** *For a finiteness structure $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$, the component of the comultiplication pseudo-natural transformation $\mathrm{dig}_{|\mathbf{A}|} : |!\mathbf{A}| \nrightarrow |!!\mathbf{A}|$ is in $\mathbf{FinProf}(!\mathbf{A}, !!\mathbf{A})$.*

Proof. Since $!\,|\mathbf{A}|$ is locally finite, $\mathrm{dig}_{|\mathbf{A}|}$ is a finite profunctor. We show that $(\mathrm{dig}^\perp_{|\mathbf{A}|})(\mathscr{F}!!\mathbf{A})^\perp \hookrightarrow (\mathscr{F}!\mathbf{A})^\perp$.

For a presheaf X in $\mathscr{F}\mathbf{A}$ considered as a species $!\mathbf{0} \nrightarrow |\mathbf{A}|$, we have $\mathrm{dig}_{|\mathbf{A}|} \circ X^! = \mathrm{dig}_{|\mathbf{A}|} \circ !X \circ \mathrm{dig}_{\mathbf{0}} \cong !!X \circ \mathrm{dig}_{!\mathbf{0}} \circ \mathrm{dig}_{\mathbf{0}} \cong !!X \circ !\mathrm{dig}_{\mathbf{0}} \circ \mathrm{dig}_{\mathbf{0}} \cong X^{!!}$, the first isomorphism follows from the pseudo-naturality of dig and the last from the pseudo-comonad axioms. Hence, for W in $\mathscr{F}!!\mathbf{A}^\perp$ and X in $\mathscr{F}\mathbf{A}$, we have $\langle(\mathrm{dig}^\perp_{|\mathbf{A}|})W, X^!\rangle \cong \langle W, \mathrm{dig}_{|\mathbf{A}|}X^!\rangle \cong \langle W, X^{!!}\rangle$. Since $X^{!!}$ is in $\mathscr{F}!!\mathbf{A}$, we obtain the desired result. ◀

4.4 Cartesian closed structure

We show in this section that the cartesian closed structure of $\mathbf{Prof}_!$ exhibited by Fiore et al. [13] can be extended to $\mathbf{FinProf}$.

▶ **Definition 39.** *A cartesian bicategory $\mathscr{B}$ is* closed *if for every pair of objects $A, B \in \mathscr{B}$, we have:*

1. *an exponential object $A \Rightarrow B$ together with an evaluation map $\mathrm{Ev}_{A,B} \in \mathscr{B}((A \Rightarrow B) \& A, B)$ and*
2. *for every $X \in \mathscr{B}$, an adjoint equivalence pseudo-natural in A, B and X:*

$$\begin{tikzcd} \mathscr{B}(X, B^A) \arrow[r, bend left, "\mathrm{Ev}_{A,B} \circ ((-) \,\&\, A)"] \arrow[r, phantom, "\perp"] & \mathscr{B}(X \,\&\, A, B) \arrow[l, bend left, "\Lambda"] \end{tikzcd}$$

For finiteness structures $\mathbf{A}$ and $\mathbf{B}$, the exponential object $\mathbf{A} \Rightarrow \mathbf{B}$ is given by $!\mathbf{A} \multimap \mathbf{B}$. We first show that the Seely adjoint equivalence in $\mathbf{Prof}$ lifts to $\mathbf{FinProf}$.

▶ **Lemma 40.** *For finiteness structures $\mathbf{A} = (|\mathbf{A}|, \mathscr{F}\mathbf{A})$ and $\mathbf{B} = (|\mathbf{B}|, \mathscr{F}\mathbf{B})$, the Seely profunctors $S_{|\mathbf{A}|,|\mathbf{B}|} : !(|\mathbf{A}| \,\&\, |\mathbf{B}|) \nrightarrow !\,|\mathbf{A}| \otimes !\,|\mathbf{B}|$ and $I_{|\mathbf{A}|,|\mathbf{B}|} : !\,|\mathbf{A}| \otimes !\,|\mathbf{B}| \nrightarrow !(|\mathbf{A}| \,\&\, |\mathbf{B}|)$ induce an adjoint equivalence $!(\mathbf{A} \,\&\, \mathbf{B}) \simeq !\mathbf{A} \otimes !\mathbf{B}$ in* $\mathbf{FinProf}$.

Proof.

- We first show that $S_{|\mathbf{A}|,|\mathbf{B}|} : !(|\mathbf{A}| \,\&\, |\mathbf{B}|) \nrightarrow !\,|\mathbf{A}| \otimes !\,|\mathbf{B}|$ given by $(w,(u,v)) \mapsto !\,|\mathbf{A}|\,(u,\pi_1 w) \times !\,|\mathbf{B}|\,(v,\pi_2 w)$ is in $\mathbf{FinProf}(!(\mathbf{A}\&\mathbf{B}), !\mathbf{A}\otimes!\mathbf{B})$ i.e. $(S^\perp_{|\mathbf{A}|,|\mathbf{B}|})\mathscr{F}(!\mathbf{A}\otimes!\mathbf{B})^\perp \hookrightarrow (\mathscr{F}!(\mathbf{A} \,\&\, \mathbf{B}))^\perp$.

  Let T be in $\mathscr{F}(!\mathbf{A} \otimes !\mathbf{B})^\perp$, we want to show that for all $W = (W_1, W_2) \in \mathscr{F}(\mathbf{A} \,\&\, \mathbf{B})$, $\langle S^\perp_{|\mathbf{A}|,|\mathbf{B}|}(T), W^!\rangle \in \mathbf{FinSet}$. The set $\langle S^\perp_{|\mathbf{A}|,|\mathbf{B}|}(T), W^!\rangle$ is isomorphic to:

$$\begin{aligned} &\int^{w \in !(|\mathbf{A}|\&|\mathbf{B}|)} W^!(w) \times \int^{u \in !|\mathbf{A}|, v \in !|\mathbf{B}|} !\,|\mathbf{A}|\,(u,\pi_1 w) \times !\,|\mathbf{B}|\,(v,\pi_2 w) \times T(u,v) \\ \cong &\int^{u \in !|\mathbf{A}|, v \in !|\mathbf{B}|} W_1^!(u) \times W_2^!(v) \times T(u,v) \end{aligned}$$

  Since W is in $\mathscr{F}(\mathbf{A} \,\&\, \mathbf{B})$, W_1 and W_2 are in $\mathscr{F}(\mathbf{A})$ and $\mathscr{F}(\mathbf{B})$ respectively, so that $W_1^!$ and $W_2^!$ are in $\mathscr{F}(!\mathbf{A})$ and $\mathscr{F}(!\mathbf{B})$ respectively. Hence, $T \perp W_1^! \times W_2^!$ as desired.
- We show that $I_{|\mathbf{A}|,|\mathbf{B}|} : !\,|\mathbf{A}| \otimes !\,|\mathbf{B}| \nrightarrow !(|\mathbf{A}| \,\&\, |\mathbf{B}|)$ given by $((u,v),w) \mapsto !\,|\mathbf{A}|\,(\pi_1 w, u) \times !\,|\mathbf{B}|\,(\pi_2 w, v)$ is in $\mathscr{F}((!\mathbf{A}\otimes!\mathbf{B}) \multimap !(\mathbf{A}\&\mathbf{B}))$. By Lemma 29, $\mathscr{F}((!\mathbf{A}\otimes!\mathbf{B}) \multimap !(\mathbf{A}\&\mathbf{B})) \cong \mathscr{F}(!\mathbf{A} \multimap (!\mathbf{B} \multimap !(\mathbf{A} \,\&\, \mathbf{B}))$ and using Lemma 27 twice, it suffices to show that for all $X \in \mathscr{F}\mathbf{A}$ and $Y \in \mathscr{F}\mathbf{B}$, $(I_{|\mathbf{A}|,|\mathbf{B}|}X^!)Y^!$ is in $\mathscr{F}!(\mathbf{A} \,\&\, \mathbf{B})$. Let Z be $\mathscr{F}(!(\mathbf{A} \,\&\, \mathbf{B}))^\perp$, the set $\langle(I_{\mathbf{A},\mathbf{B}}X^!)Y^!, Z\rangle$ is isomorphic to:

$$\int^{w\in!(\mathbf{A}\&\mathbf{B}),u\in!\mathbf{A},v\in!\mathbf{B}} Z(w)\times !\mathbf{A}(\pi_1 w,u)\times !\mathbf{B}(\pi_2 w,v)\times X^!(u)\times Y^!(v)$$
$$\cong \int^{w\in!(\mathbf{A}\&\mathbf{B})} Z(w)\times (X,Y)^!(w)$$

Since $(X,Y)^!$ is in $\mathscr{F}(!(\mathbf{A}\,\&\,\mathbf{B}))$, we obtain the desired result. ◀

It remains to show that the non-linear evaluation and currying preserve the finiteness structure. The non-linear evaluation $\mathrm{Ev}_{|\mathbf{A}|,|\mathbf{B}|} : !((|\mathbf{A}| \Rightarrow |\mathbf{B}|)\,\&\,|\mathbf{A}|) \nrightarrow |\mathbf{B}|$ is given by the composite $\mathrm{ev}_{!|\mathbf{A}|,|\mathbf{B}|}\circ(\mathrm{der}_{|\mathbf{A}|\Rightarrow|\mathbf{B}|}\otimes\mathrm{id})\circ S_{|\mathbf{A}|\Rightarrow|\mathbf{B}|,|\mathbf{A}|}$ where $\mathrm{ev}_{|\mathbf{A}|,|\mathbf{B}|} : \mathbf{A}\otimes(\mathbf{A}\multimap\mathbf{B}) \nrightarrow \mathbf{B}$ is the linear evaluation coming from the monoidal closed structure in the linear bicategory $\mathbf{FinProf}$. As a composite of finiteness profunctors, $\mathrm{Ev}_{|\mathbf{A}|,|\mathbf{B}|}$ is in $\mathbf{FinProf}_!((\mathbf{A}\Rightarrow\mathbf{B})\,\&\,\mathbf{A},\mathbf{B})$. For a finiteness species P in $\mathbf{FinProf}_!(\mathbf{A}\,\&\,\mathbf{B},\mathbf{C})$, its *currying* $\Lambda(P)\in\mathbf{FinProf}_!(\mathbf{A},\mathbf{B}\Rightarrow\mathbf{C})$ is given by $\lambda(P\circ I_{|\mathbf{A}|,|\mathbf{B}|})$ where $\lambda : \mathbf{FinProf}(!\mathbf{A}\otimes!\mathbf{B},\mathbf{C}) \to \mathbf{FinProf}(!\mathbf{A},!\mathbf{B}\multimap\mathbf{C})$ is provided by the monoidal closed structure on $\mathbf{FinProf}$.

▶ **Theorem 41.** $\mathbf{FinProf}_!$ *is cartesian closed.*

Proof. Direct consequence of the remarks above and Lemma 18. ◀

4.5 Differential structure

The bicategory of generalized species $\mathbf{Prof}_!$ is a model of differential linear logic where differentiation on analytic functors generalises the standard differential operation on formal power series [13]. We show in this section that the differential structure extends to $\mathbf{FinProf}$. It suffices to show that the codereliction, coweakening and cocontraction pseudo-natural transformations have components in $\mathbf{FinProf}$ and all the coherence axioms will be immediately verified.

▶ **Lemma 42.** *For a finiteness structure* $\mathbf{A}=(|\mathbf{A}|,\mathscr{F}\mathbf{A})$*, the component of codereliction pseudo-natural transformation* $\overline{\mathrm{der}}_{|\mathbf{A}|} : |\mathbf{A}| \nrightarrow !\,|\mathbf{A}|$ *given by* $(a,u)\mapsto !\,|\mathbf{A}|\,(u,\langle a\rangle)$ *is a finiteness profunctor* $\mathbf{A}\nrightarrow_{\mathsf{f}} !\mathbf{A}$.

Proof. Since $|\mathbf{A}|$ is locally finite, $\overline{\mathrm{der}}_{|\mathbf{A}|}$ is a finite profunctor. By Lemma 15, it remains to show that $\overline{\mathrm{der}}^{\perp}_{|\mathbf{A}|}((\mathscr{F}!\mathbf{A})^{\perp})\hookrightarrow(\mathscr{F}\mathbf{A})^{\perp}$ i.e. that for all $Z\in(\mathscr{F}!\mathbf{A})^{\perp}$ and $X\in\mathscr{F}\mathbf{A}$, $(\overline{\mathrm{der}}^{\perp}_{|\mathbf{A}|})Z\perp X$.

$$\langle(\overline{\mathrm{der}}^{\perp}_{|\mathbf{A}|})Z,X\rangle = \int^{u\in!|\mathbf{A}|,a\in|\mathbf{A}|} Z(u)\times !\,|\mathbf{A}|\,(u,\langle a\rangle)\times X(a)$$
$$\cong \int^{a\in|\mathbf{A}|} Z(\langle a\rangle)\times X(a) \hookrightarrow \int^{u\in!|\mathbf{A}|} Z(u)\times X^!(u)\in\mathbf{FinSet}$$

The last inclusion follows from the isomorphism $X^!(\langle a\rangle)\cong X(a)$. ◀

Since the components of the coweakening $\overline{\mathrm{w}}_{|\mathbf{A}|} : \mathbf{1} \nrightarrow !\,|\mathbf{A}|$ and cocontraction $\overline{\mathrm{c}}_{|\mathbf{A}|} : !\,|\mathbf{A}|\times !\,|\mathbf{A}| \nrightarrow !\,|\mathbf{A}|$ pseudo-natural transformations are obtained from the Seely equivalences and the biproduct structure, it is immediate that they can be extended to $\mathbf{FinProf}$. It implies that the deriving pseudo-natural transformation $\delta_{|\mathbf{A}|} : |\mathbf{A}| \nrightarrow !\,|\mathbf{A}|\times|\mathbf{A}|$ given by

$$!\,|\mathbf{A}|\times|\mathbf{A}| \xrightarrow{\mathrm{id}\times\overline{\mathrm{der}}_{|\mathbf{A}|}} !\,|\mathbf{A}|\times !\,|\mathbf{A}| \xrightarrow{\overline{\mathrm{c}}_{|\mathbf{A}|}} !\,|\mathbf{A}|$$

is therefore a finiteness profunctor $!\mathbf{A} \otimes \mathbf{A} \nrightarrow_{\mathsf{f}} !\mathbf{A}$ so that for a finiteness species $F : !\mathbf{A} \nrightarrow \mathbf{B}$ its differential $F \circ \delta_{|\mathbf{A}|} : !\mathbf{A} \otimes \mathbf{A} \nrightarrow_{\mathsf{f}} \mathbf{B}$ given by $((u, a), b) \mapsto F(u \otimes \langle a \rangle, b)$ is also a finiteness species.

Conclusion and perspectives

We have constructed a new bicategorical model of differential linear logic categorifying the finiteness model first introduced by Ehrhard [9]. The resulting cartesian closed bicategory refines the model of generalized species by Fiore et al. [13]. The objects are endowed with an additional structure which enables to enforce finite computations as morphisms are species that preserve the finiteness structure.

In future work, we aim to prove that our construction can be generalized to the setting of enriched species studied by Gambino and Joyal [15]. In the 1-categorical model of finiteness spaces, we can express various forms of non-determinism depending on the semi-ring of scalars chosen for the series coefficients. In our case, the analogous variation would come from changing the enrichment basis. In particular, for species enriched over vector spaces, our construction will guarantee that computations are always finite dimensional even if we work in an infinite dimensional setting which could lead to interesting applications for the semantics of quantum λ-calculus [24] and stochastic rewriting systems [2].

In this paper, we have worked on a focused orthogonality on the subclass of finitely presented objects. Our construction opens the way for a lot of variation in terms of the chosen class of objects: for example, restricting the interactions to absolutely presentable objects could yield to a model of totality in the spirit of the one studied by Loader [23].

References

1 Joey Beauvais-Feisthauer, Richard Blute, Ian Dewan, Blair Drummond, and Pierre-Alain Jacqmin. Finiteness spaces, étale groupoids and their convolution algebras. In *Semigroup Forum*, pages 1–16. Springer, 2020.

2 Nicolas Behr. On Stochastic Rewriting and Combinatorics via Rule-Algebraic Methods. In Patrick Bahr, editor, *Proceedings 11th International Workshop on Computing with Terms and Graphs (TERMGRAPH 2020)*, volume 334, pages 11–28. Open Publishing Association, 2021.

3 Jean Bénabou. Distributors at work, 2000. Lecture notes written by Thomas Streicher. URL: `https://www2.mathematik.tu-darmstadt.de/~streicher/FIBR/DiWo.pdf`.

4 F. Bergeron, G. Labelle, and P. Leroux. *Combinatorial species and tree-like structures*, volume 67 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, 1998. Translated from the 1994 French original by Margaret Readdy, With a foreword by Gian-Carlo Rota.

5 Richard Blute, Robin Cockett, Pierre-Alain Jacqmin, and Philip Scott. Finiteness spaces and generalized power series. *Electronic Notes in Theoretical Computer Science*, 341:5–22, 2018.

6 Richard F. Blute. Hopf algebras and linear logic. *Mathematical Structures in Computer Science*, 6(2):189–212, 1996.

7 Gian Luca Cattani and Glynn Winskel. Profunctors, open maps and bisimulation. *Mathematical Structures in Computer Science*, 15:553–614, June 2005.

8 Thomas Ehrhard. On köthe sequence spaces and linear logic. *Mathematical. Structures in Comp. Sci.*, 12(5):579–623, 2002.

9 Thomas Ehrhard. Finiteness spaces. *Mathematical Structures in Computer Science*, 15(4):615–646, 2005. 32 pages.

10 Thomas Ehrhard. An introduction to differential linear logic: proof-nets, models and antiderivatives. *Mathematical Structures in Computer Science*, 28(7):995–1060, 2018.

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

12 Marcelo Fiore. Analytic functors between presheaf categories over groupoids. *Theor. Comput. Sci.*, 546:120–131, 2014.

13 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.

14 Marcelo Fiore, Nicola Gambino, Martin Hyland, and Glynn Winskel. Relative pseudomonads, Kleisli bicategories, and substitution monoidal structures. *Selecta Mathematica*, 24(3):2791–2830, 2018.

15 Nicola Gambino and André Joyal. *On Operads, Bimodules and Analytic Functors*. Memoirs of the American Mathematical Society. American Mathematical Society, 2017.

16 Jean-Yves Girard. Normal functors, power series and λ-calculus. *Ann. Pure Appl. Logic*, 37(2):129–177, 1988.

17 Jean-Yves Girard. Coherent Banach Spaces: a continuous denotational semantics extended abstract. *Electronic Notes in Theoretical Computer Science*, 3:81–87, 1996. Linear Logic 96 Tokyo Meeting.

18 Ryu Hasegawa. Two applications of analytic functors. *Theoretical Computer Science*, 272(1):113–175, 2002. Theories of Types and Proofs 1997.

19 Martin Hyland. Some reasons for generalising domain theory. *Mathematical Structures in Computer Science*, 20(2):239–265, 2010.

20 Martin Hyland and Andrea Schalk. Glueing and orthogonality for models of linear logic. *Theoretical Computer Science*, 294(1):183–231, 2003. Category Theory and Computer Science.

21 André Joyal. Une théorie combinatoire des séries formelles. *Adv. in Math.*, 42(1):1–82, 1981.

22 André Joyal. Foncteurs analytiques et espèces de structures. In Gilbert Labelle and Pierre Leroux, editors, *Combinatoire énumérative*, pages 126–159, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

23 Ralph Loader. Linear logic, totality and full completeness. In *In Proceedings of LiCS '94*, pages 292–298. Press, 1994.

24 Michele Pagani, Peter Selinger, and Benoît Valiron. Applying quantitative semantics to higher-order quantum computing. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 647–658, 2014.

25 Michele Pagani, Christine Tasson, and Lionel Vaux. Strong normalizability as a finiteness structure via the Taylor expansion of *lambda*-terms. In *International Conference on Foundations of Software Science and Computation Structures*, pages 408–423. Springer, 2016.

26 Lionel Vaux. A non-uniform finitary relational semantics of system T. *RAIRO – Theoretical Informatics and Applications*, 47(1):111–132, 2013.

Failure of Cut-Elimination in the Cyclic Proof System of Bunched Logic with Inductive Propositions

Kenji Saotome[1]
Nagoya University, Japan

Koji Nakazawa
Nagoya University, Japan

Daisuke Kimura
Toho University, Japan

Abstract

Cyclic proof systems are sequent-calculus style proof systems that allow circular structures representing induction, and they are considered suitable for automated inductive reasoning. However, Kimura et al. have shown that the cyclic proof system for the symbolic heap separation logic does not satisfy the cut-elimination property, one of the most fundamental properties of proof systems. This paper proves that the cyclic proof system for the bunched logic with only nullary inductive predicates does not satisfy the cut-elimination property. It is hard to adapt the existing proof technique chasing contradictory paths in cyclic proofs since the bunched logic contains the structural rules. This paper proposes a new proof technique called proof unrolling. This technique can be adapted to the symbolic heap separation logic, and it shows that the cut-elimination fails even if we restrict the inductive predicates to nullary ones.

2012 ACM Subject Classification Theory of computation → Proof theory; Theory of computation → Separation logic

Keywords and phrases cyclic proofs, cut-elimination, bunched logic, separation logic, linear logic

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.11

Funding This work was partially supported by the Japan Society for the Promotion of Science (JSPS), Core-to-Core Program (A. Advanced Research Networks), and Grantz-in-Aid for Scientific Research KAKENHI 18K11161.

1 Introduction

Static verification of software often needs to check the validity of entailments, which are implications between logical formulas. One of the ways to check entailments is an automated proof search in some proof systems.

The *bunched logic* [9] was introduced to reason compositional properties of resources with some additional logical connectives such as the multiplicative conjunction. The *separation logic* [11], which is based on the bunched logic, is one of the most successful logical foundations for verification of heap-manipulating programs using pointers. For inductive reasoning in these logics, Brotherston et al. proposed some *cyclic proof systems* for the bunched logic [3] and the separation logic [4, 5]. The cyclic proof systems allow cycles in proofs, which correspond to induction. They offer an efficient way for automated validity checking of entailments with inductive definitions since they provide a proof search algorithm that does not require finding induction hypothesis formulas a priori.

[1] Corresponding author

The *cut-elimination property* of proof systems means that the provability does not change with or without the cut rule:

$$\frac{A \vdash C \quad C \vdash B}{A \vdash B}\ (Cut).$$

From a theoretical viewpoint, the cut-elimination property means that applying lemma is admissible, and it implies significant properties such as the subformula property and consistency. The cut-elimination property is also important from a practical viewpoint: When the cut rule is included as a candidate of the next rules during an automated proof search, we have to find a suitable cut formula, namely the formula C in the cut rule above. In general, cut formulas are independent of formulas in the conclusion of cut rules, and we have to find them heuristically.

Hence, we expect proof systems to enjoy the cut-elimination property, and it holds in many proof systems such as Gentzen's *LK* for the first-order logic and the (non-cyclic) proof system *LBI* for the bunched logic [10]. Furthermore, it has been shown that the cut-elimination property holds in some infinitary proof systems [6, 7, 2]. The cut-elimination processes in the existing proofs are not closed under the regularity of infinitary proof trees, and that suggests that the cut-elimination does not hold in the cyclic proof systems since cyclic proofs are regular infinitary proofs.

Kimura et al. [8] showed that the cut-elimination property fails for Brotherston's cyclic proof system [4] for the symbolic heaps, which are restricted forms of the separation logic formulas. They gave a counterexample entailment $\mathrm{ls}(x, y) \vdash \mathrm{sl}(x, y)$, where both $\mathrm{ls}(x, y)$ and $\mathrm{sl}(x, y)$ are inductive predicates that represent the semantically same data structure, namely singly-linked list from x to y, but are defined in the different ways. They assumed the existence of a cut-free cyclic proof of this counterexample and showed that a unique infinite path in the cyclic proof is a contradictory path, namely, an infinite path in which the sizes of sequents are strictly increasing. The contradictory path leads to a contradiction since it breaks the finiteness of the cyclic proof.

In [8], they guessed that the cut-elimination would not hold for the bunched logic either, but suggested that their proof technique needs some modification to handle the structural rules, the left weakening and the left contraction rules, in the bunched logic. The structural rules cause much more possibilities of paths than the symbolic heap separation logic, and we have to find a contradictory path from them. For example, we can assume a segment of a cyclic proof of the sequent $\mathrm{P}_{AB} \vdash \mathrm{P}_{BA}$ in the bunched logic as in Figure 1, where P_{AB} and P_{BA} are inductively defined as

$$\begin{aligned}
&\mathrm{P}_{AB} := \mathrm{P}_B \mid \mathrm{P}_{AB} * A && \mathrm{P}_A := I \mid \mathrm{P}_A * A\\
&\mathrm{P}_{BA} := \mathrm{P}_A \mid \mathrm{P}_{BA} * B && \mathrm{P}_B := I \mid \mathrm{P}_B * B.
\end{aligned}$$

Here, the separators "," and ";" on the left-hand sides of sequents correspond to the multiplicative conjunction ($*$) and the additive conjunction ($\wedge$), respectively. The proposition constants I and $\top$ are the units for $*$ and $\wedge$, respectively. The rule (UL) unfolds predicates on the left-hand side from bottom to top. The rule (E) replaces the left-hand side with an equivalent one. The rules (W) and (C) are the left weakening and the left contraction rules, respectively. The rule ($\top$) is admissible using the left weakening rule, and a link between two sequents marked with (†) forms a cycle, which satisfies the soundness condition for the cyclic proofs, the global trace condition [6]. Therefore, the rightmost path contains no contradiction. Furthermore, the part ($\star$) is easily proved. This means that, to find a contradictory path, we

$$
\dfrac{
\dfrac{\dfrac{\vdots\ (\star)}{\mathrm{P}_B \vdash \mathrm{P}_{BA}}}{\mathrm{P}_{AB}; \mathrm{P}_B \vdash \mathrm{P}_{BA}}\,(W)
\qquad
\dfrac{
\dfrac{
\dfrac{\vdots\ (\#)}{\mathrm{P}_{AB}; (\mathrm{P}_B, \top) \vdash \mathrm{P}_{BA}}
\quad
\dfrac{\dfrac{\dfrac{\mathrm{P}_{AB}; (\mathrm{P}_{AB}, \top) \vdash \mathrm{P}_{BA}\ (\dagger)}{\mathrm{P}_{AB}; (\mathrm{P}_{AB}, (A, \top)) \vdash \mathrm{P}_{BA}}\,(\top)}{\mathrm{P}_{AB}; ((\mathrm{P}_{AB}, A), \top) \vdash \mathrm{P}_{BA}}\,(E)}{\mathrm{P}_{AB}; (\mathrm{P}_{AB} * A, \top) \vdash \mathrm{P}_{BA}}\,(*L)
}{
\dfrac{\mathrm{P}_{AB}; (\mathrm{P}_{AB}, \top) \vdash \mathrm{P}_{BA}(\dagger)}{\mathrm{P}_{AB}; (\mathrm{P}_{AB}, A) \vdash \mathrm{P}_{BA}}\,(\top)
}\,(UL)^2
}{\mathrm{P}_{AB}; \mathrm{P}_{AB} * A \vdash \mathrm{P}_{BA}}\,(*L)
}{
\dfrac{\mathrm{P}_{AB}; \mathrm{P}_{AB} \vdash \mathrm{P}_{BA}}{\mathrm{P}_{AB} \vdash \mathrm{P}_{BA}}\,(C)
}\,(UL)^1
$$

Figure 1 A proof segment in the cyclic proof system of the bunched logic.

$$
\begin{array}{c}
\vdots\ (\#') \\
\dfrac{\dfrac{I * A^m; (I, \top) \vdash \mathrm{P}_{BA}\ (\dagger)}{I * A^m; (I, (A, \top)) \vdash \mathrm{P}_{BA}}\,(\top)}{I * A^m; ((I, A), \top) \vdash \mathrm{P}_{BA}}\,(E) \\
\vdots \\
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{I * A^m; (I * A^{m-2}, \top) \vdash \mathrm{P}_{BA}}{I * A^m; (I * A^{m-2}, (A, \top)) \vdash \mathrm{P}_{BA}}\,(\top)}{I * A^m; ((I * A^{m-2}, A), \top) \vdash \mathrm{P}_{BA}}\,(E)}{I * A^m; (I * A^{m-1}, \top) \vdash \mathrm{P}_{BA}\ (\dagger)}\,(*L)}{I * A^m; (I * A^{m-1}, A) \vdash \mathrm{P}_{BA}}\,(\top)}{I * A^m; I * A^m \vdash \mathrm{P}_{BA}}\,(*L)}{I * A^m \vdash \mathrm{P}_{BA}}\,(C)
\end{array}
$$

Figure 2 Proof unrolling.

have to chase it in the part $(\#)$, and hence we sometimes have to choose the right assumption (at $(UL)^1$), and also have to choose the left assumption (at $(UL)^2$). Therefore, it is hard to find such a contradictory path in cyclic proofs.

Kimura et al. also mentioned a possibility to recover the cut-elimination property by restricting the number of arities (to unary or nullary) for inductive predicates. Restricting arities of inductive predicates may drastically change the situation as the result of Tatsuta et al [12]. They showed the decidability of the entailment checking problem for the symbolic heap separation logic with only unary inductive predicates whereas the problem for that with general inductive predicates is known to be undecidable [1].

In this paper, we show that the cut-elimination property fails for the cyclic proof system of the bunched logic [3] by a counterexample only with nullary inductive predicates. We develop a proof technique called *proof unrolling*. For a cut-free cyclic proof of $\Gamma \vdash \phi$, by using proof unrolling, we can construct a cut-free non-cyclic proof of $\Delta \vdash \phi$ for any Δ obtained by unfolding inductive predicates in Γ. For the example in Figure 1 and the formula $I * A^m = ((I * A) * \cdots * A) * A$ (m copies of A's) obtained by unfolding P_{AB}, we can construct the non-cyclic proof of $I * A^m \vdash \mathrm{P}_{BA}$ in Figure 2 by proof unrolling. During the proof unrolling, we unroll the cycle (at $(\dagger)$), and choose cases at the rule (UL) depending on the unfolding tree of P_{AB} to obtain $I * A^m$. We will show that, for any cyclic proof of $\mathrm{P}_{AB} \vdash \mathrm{P}_{BA}$, if m is sufficiently large, any path in the non-cyclic proof by proof unrolling corresponds to a contradictory path in the original cyclic proof. The remaining path in the part ($\#'$) of Figure 2 corresponds to a contradictory path in the part ($\#$) of Figure 1. Hence, the existence of a cyclic proof of $\mathrm{P}_{AB} \vdash \mathrm{P}_{BA}$ derives a contradiction.

The proof unrolling is a general technique almost independent of a choice of logic. We can straightforwardly adapt our proof to any cyclic proof system of a logic that contains a connective representing resource composition such as the separation logic and the multiplicative linear logic. Hence, the cut-elimination fails for the cyclic proof system of the separation logic even if we restrict inductive predicates to nullary ones.

The structure of the paper is as follows. Section 2 introduces a simple fragment of the propositional bunched logic BI_{ID0} with inductive definitions, and its cyclic proof system $CLBI^{\omega}_{ID0}$, which is a subsystem of $CLBI^{\omega}_{ID}$ given by Brotherston [3]. Section 3 presents our proof unrolling technique. Section 4 proves the main result of this paper, which shows that the cut-elimination property does not hold in $CLBI^{\omega}_{ID0}$ using the proof unrolling technique. It also discusses that our proof technique can be adapted to other systems including $CLBI^{\omega}_{ID}$. Section 5 concludes.

2 Bunched Logic with Inductive Propositions

In this section, we define the syntax and semantics of a core of the bunched logic BI_{ID0}, which is based on the logic in [3]. In BI_{ID0}, atomic and inductive predicates are restricted to nullary ones, which we call atomic propositions and inductive propositions, respectively. We also define proof systems for BI_{ID0}: one is the ordinary proof system LBI_{ID0}, and the other is the cyclic proof system $CLBI^{\omega}_{ID0}$.

In the following sections, we will prove that cuts cannot be eliminated in $CLBI^{\omega}_{ID0}$, and this result can be easily extended to the system in [3].

2.1 Syntax of BI_{ID0}

We use metavariables A, B,... for atomic propositions and P, Q,... for inductive propositions. We implicitly fix a language Σ consisting of atomic and inductive propositions. Note that in BI_{ID0}, we have neither terms, variables, nor function symbols.

▶ **Definition 1** (Formulas of BI_{ID0}). *Let I and $\top$ be propositional constants. The* formulas *of BI_{ID0}, denoted by ϕ, ψ,..., are defined as*

$$\phi ::= I \mid \top \mid A \mid P \mid \phi * \phi \mid \phi \wedge \phi.$$

In this paper, $*$ and $\wedge$ are treated as left-associative operators, that is, we write $\phi_1 * \phi_2 * \phi_3$ for $(\phi_1 * \phi_2) * \phi_3$. The notation A^n denotes $A * \cdots * A$ where the number of A's is n. We also use the notation $P * A^n$ for $P * A * \cdots * A$, namely $(\cdots((P * A) * A)\cdots) * A)$.

▶ **Definition 2** (Bunch). *The* bunches*, denoted by $\Gamma, \Delta, \ldots$, are defined as*

$$\Gamma, \Delta ::= \phi \mid \Gamma, \Gamma \mid \Gamma; \Gamma.$$

We sometimes use terminologies of trees to bunches by identifying a bunch as a tree whose internal nodes are labeled by "," or ";", and whose leaves are labeled by a formula. We write $\Gamma(\Delta)$ to mean that Γ of which Δ is a subtree. For a bunch $\Gamma(\Delta)$, $\Gamma(\Delta')$ is a bunch obtained by replacing the subtree Δ of Γ by Δ'.

The labels "," and ";" intuitively mean $*$ and $\wedge$, respectively. For a bunch Γ, we define the bunch formula ϕ_Γ as the formula defined as:

$$\begin{aligned}
\phi_\Gamma &= \Gamma, && (\Gamma \text{ is a formula});\\
\phi_{\Gamma_1,\Gamma_2} &= \phi_{\Gamma_1} * \phi_{\Gamma_2};\\
\phi_{\Gamma_1;\Gamma_2} &= \phi_{\Gamma_1} \wedge \phi_{\Gamma_2}.
\end{aligned}$$

▶ **Definition 3** (Equivalence of bunches). *Define the bunch equivalence $\equiv$ as the least equivalence relation satisfying:*

- *commutative monoid equations for ',' and I;*
- *commutative monoid equations for ';' and $\top$;*
- *congruence: if $\Delta \equiv \Delta'$ then $\Gamma(\Delta) \equiv \Gamma(\Delta')$.*

▶ **Definition 4** (Size of formulas and bunches). *Let ϕ be a formula and Γ be a bunch. The size of ϕ (denoted by $|\phi|$) is as*

$$\begin{aligned}
|\phi| &= 1 && (\phi = I \text{ or } \top \text{ or } A \text{ or } P);\\
|\phi| &= |\psi| + |\psi'| + 1 && (\phi = \psi * \psi' \text{ or } \psi \wedge \psi').
\end{aligned}$$

The size of Γ (denoted by $|\Gamma|$) is as

$$\begin{aligned}
|\Gamma| &= |\phi| && (\Gamma = \phi);\\
|\Gamma| &= |\Delta| + |\Delta'| + 1 && (\Gamma = \Delta, \Delta' \text{ or } \Delta; \Delta').
\end{aligned}$$

▶ **Definition 5** (Inductive definition). *An* inductive definition clause *of P is of the form $P := \phi$. For a set Φ of inductive definition clauses of inductive propositions, we define $\Phi_P = \{\phi \mid P := \phi \in \Phi\}$. We say that P is defined by $P := \phi_1 \mid \cdots \mid \phi_k$ in Φ if and only if $\Phi_P = \{\phi_1, \cdots, \phi_k\}$.*

▶ **Definition 6** (BI_{ID0} sequent). *Let Γ be a bunch and ϕ be a formula. $\Gamma \vdash \phi$ is called a* BI_{ID0} sequent. *Γ is called the* antecedent *of $\Gamma \vdash \phi$ and ϕ is called the* succedent *of $\Gamma \vdash \phi$. We define $L(\Gamma \vdash \phi) = \Gamma$ and $R(\Gamma \vdash \phi) = \phi$.*

2.2 Semantics of BI_{ID0}

We recall a *standard model* [3] as the semantics of BI_{ID0}. In the following, we fix a set Φ of inductive definition clauses.

▶ **Definition 7** (BI_{ID0} standard model). A BI_{ID0} standard model *is a tuple $M = (\langle R, \circ, e\rangle, \mathbf{A}^M)$ satisfying the following:*

- *$\langle R, \circ, e\rangle$ is a partial commutative monoid with the unit e;*
- *$\mathbf{A}^M$ is a set consisting of $A^M \subseteq R$ for each atomic proposition A.*

Let M be a BI_{ID0} standard model and let $r \in R$. We define the satisfaction relation $M, r \models \phi$ by

$$
\begin{aligned}
M, r \models \top &\iff \textit{true} \\
M, r \models I &\iff r = e \\
M, r \models A &\iff r \in A^M \ \textit{(for atomic proposition } A\textit{)} \\
M, r \models P^{(0)} & \quad \textit{never holds} \\
M, r \models P^{(m+1)} &\iff M, r \models \phi[P_1^{(m)}, \ldots, P_k^{(m)}/P_1, \ldots, P_k] \\
&\quad\ \textit{for some } \phi \in \Phi_P \textit{ containing inductive propositions } P_1, \ldots, P_k \\
M, r \models P &\iff M, r \models P^{(m)} \textit{ for some } m \\
M, r \models \phi_1 \wedge \phi_2 &\iff M, r \models \phi_1 \textit{ and } M, r \models \phi_2 \\
M, r \models \phi_1 * \phi_2 &\iff r = r_1 \circ r_2 \textit{ and } M, r_1 \models \phi_1 \textit{ and } M, r_2 \models \phi_2 \textit{ for some } r_1, r_2 \in R\ ,
\end{aligned}
$$

where $P^{(m)}$ are auxiliary proposition symbols, and $\phi[P_1^{(m)}, \ldots, P_k^{(m)}/P_1, \ldots, P_k]$ is the formula obtained by replacing each P_i by $P_i^{(m)}$. We define $M, r \models \Gamma$ as $M, r \models \phi_\Gamma$.

By defining in this way, the satisfaction relation for inductive propositions is the same as that in the standard model of [3].

▶ **Definition 8** (Validity). *Let M be a standard model. A sequent $\Gamma \vdash \phi$ is true in M, denoted by $\Gamma \models_M \phi$, if and only if, $M, r \models \Gamma$ implies $M, r \models \phi$ for any r. A sequent $\Gamma \vdash \phi$ is valid, denoted by $\Gamma \models \phi$, if and only if, it is true for any standard models. $\Gamma \models_M \Delta$ and $\Gamma \models \Delta$ are similarly defined.*

▶ **Example 9.** An example of the standard models is the *multiset model*. Let the set of atomic propositions Σ be $\{A, B\}$. The multiset model M_{multi} for Σ is the tuple $(\langle R_{\mathrm{multi}}, \uplus, \emptyset \rangle, \mathbf{A}^{M_{\mathrm{multi}}})$ such that

- R_{multi} is the set of multisets consisting of a and b;
- $\uplus$ is the merging operation of two multisets;
- A^M and B^M are $\{\{\mathsf{a}\}\}$ and $\{\{\mathsf{b}\}\}$, respectively.

For example, $M_{\mathrm{multi}}, \{\mathsf{a}\} \models A$, $M_{\mathrm{multi}}, \{\mathsf{a}, \mathsf{b}\} \models A * B$, and $M_{\mathrm{multi}}, \{\mathsf{a}, \mathsf{a}\} \models A * A * I$ are true, and $M_{\mathrm{multi}}, \{\mathsf{a}\} \models B$ and $M_{\mathrm{multi}}, \{\mathsf{a}\} \models A * A$ are false.

2.3 Inference rules of LBI_{ID0} and $CLBI^{\omega}_{ID0}$

This and the next subsection define two proof systems LBI_{ID0} and $CLBI^{\omega}_{ID0}$. The system LBI_{ID0} is a non-cyclic proof system and the system $CLBI^{\omega}_{ID0}$is a cyclic proof system. The common inference rules of them are given as follows.

▶ **Definition 10.** *The common inference rules of the proof systems LBI_{ID0} and $CLBI^{\omega}_{ID0}$ are the following.*

$$
\frac{}{\phi \vdash \phi}\ (Ax) \qquad \frac{\Gamma \vdash \phi \quad \Delta(\phi) \vdash \psi}{\Delta(\Gamma) \vdash \psi}\ (Cut),
$$

$$
\frac{\Gamma(\Delta) \vdash \phi}{\Gamma(\Delta; \Delta') \vdash \phi}\ (W) \qquad \frac{\Gamma(\Delta; \Delta) \vdash \phi}{\Gamma(\Delta) \vdash \phi}\ (C) \qquad \frac{\Gamma \vdash \phi}{\Delta \vdash \phi}\ (E) \quad (\Delta \equiv \Gamma),
$$

$$
\frac{\Gamma(\phi, \psi) \vdash \chi}{\Gamma(\phi * \psi) \vdash \chi}\ (*L) \qquad \frac{\Gamma \vdash \phi \quad \Delta \vdash \psi}{\Gamma, \Delta \vdash \phi * \psi}\ (*R) \qquad \frac{\Gamma(\phi; \psi) \vdash \chi}{\Gamma(\phi \wedge \psi) \vdash \chi}\ (\wedge L) \qquad \frac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi}\ (\wedge R).
$$

$$\frac{\Gamma(\phi_1) \vdash \phi \quad \cdots \quad \Gamma(\phi_n) \vdash \phi}{\Gamma(P) \vdash \phi}\ (UL) \qquad \frac{\Gamma \vdash \phi_i}{\Gamma \vdash P}\ (UR) \quad (1 \leq i \leq n),$$

where the inductive predicate P is defined by $P := \phi_1 \mid \dots \mid \phi_n$. (UL) and (UR) are called unfolding rules. The formula ϕ in (Cut) is called its cut formula.

2.4 Proofs in LBI_{ID0} and $CLBI^{\omega}_{ID0}$

Let Seq be the set of the BI_{ID0} sequents, Rules be the set of the common inference rules of LBI_{ID0} and $CLBI^{\omega}_{ID0}$, and Rules^+ be the set $\text{Rules} \cup \{(Bud)\}$.

▶ **Definition 11** (LBI_{ID0} Proof)**.** *An LBI_{ID0}* proof *is a tuple $\text{Pr} = (N, l, r)$ satisfying the following:*

- *N is the set of nodes for a finite tree. The elements of N are strings of positive integers, the root is the empty string ε, and children of v are $v1$, $v2, \dots,$ where vi is a concatenation of the string v and the integer i.*
- *$l : N \to \text{Seq}$ is a label function.*
- *$r : N \to \text{Rules}$ is a rule function.*
- *If $r(v) \in \text{Rules}$ is a rule with n premises, then v has exactly n children, and $\dfrac{l(v1) \quad \dots \quad l(vn)}{l(v)}\ r(v)$ is a correct rule instance of LBI_{ID0}.*

An LBI_{ID0} proof $\text{Pr} = (N, l, r)$ is called an LBI_{ID0} proof of $l(\varepsilon)$. When $r(v)$ is not (Cut) for any $v \in N$, Pr is called a cut-free LBI_{ID0} proof.

▶ **Definition 12** ($CLBI^{\omega}_{ID0}$ pre-proof)**.** *A $CLBI^{\omega}_{ID0}$ pre-proof is a tuple $\text{Pr} = (N, l, r, \rho)$ satisfying the following:*

- *N and l are defined similarly as those of the LBI_{ID0} proofs.*
- *$r : N \to \text{Rules}^+$ is a rule function.*
- *$\rho : \{v \in N \mid r(v) = (Bud)\} \to N$ is a bud-companion function.*
- *If $r(v) \in \text{Rules}$ is a rule with n premises, then v has exactly n children, and $\dfrac{l(v1) \quad \dots \quad l(vn)}{l(v)}\ r(v)$ is a correct rule instance.*
- *If $r(v) = (Bud)$, then v has no child and we have $l(v) = l(\rho(v))$.*

When $r(v) = (Bud)$, v is called a bud, *and $\rho(v)$ is called the* companion *of v.*

▶ **Definition 13** (Path)**.** *Let $\text{Pr} = (N, l, r, \rho)$ be a $CLBI^{\omega}_{ID0}$ pre-proof. The proof graph $G(\text{Pr})$ is a directed graph whose set of the nodes are N, and which has an edge from v to v' if and only if either v' is a child of v or v' is the companion of v. A* path *in Pr is a path in $G(\text{Pr})$.*

The path of LBI_{ID0} is defined in the same way except for the bud-companion edges. We consider both finite and infinite paths in proofs. We use α for either a natural number or the ordinal ω, and we denote a path by $(v_i)_{i<\alpha}$.

▶ **Definition 14** (Trace)**.** *Let $(v_i)_{i<\alpha}$ be a path in a $CLBI^{\omega}_{ID0}$ pre-proof Pr. A* trace *along $(v_i)_{i<\alpha}$ is a sequence of occurrences of inductive predicates $(P_i)_{i<\alpha}$ such that each P_i occurs in $L(l(v_i))$, and satisfies the following conditions:*

- *If $r(v_i) = (UL)$ and P_i is unfolded by this rule instance, P_{i+1} appears as a subformula in the unfolding result of P_i in $L(l(v_{i+1}))$. In this case, i is called a* progressing point *of the trace $(P_i)_{i<\alpha}$.*
- *Otherwise, P_{i+1} is the subformula occurrence in $L(l(v_{i+1}))$ corresponding to P_i in $L(l(v_i))$.*

If a trace contains infinitely many progressing points, it is called an infinitely progressing trace.

▶ **Definition 15** ($CLBI^{\omega}_{ID0}$ Proof). *A $CLBI^{\omega}_{ID0}$ pre-proof* $\mathrm{Pr} = (N, l, r, \rho)$ *is called a* $CLBI^{\omega}_{ID0}$ proof *when it satisfies the* global trace condition, *that is, for every infinite path* $(v_i)_{i<\omega}$ *in* Pr, *there is an infinitely progressing trace following some tail of the path* $(v_i)_{n\le i<\omega}$. *A $CLBI^{\omega}_{ID0}$ proof* $\mathrm{Pr} = (N, l, r, \rho)$ *is called a $CLBI^{\omega}_{ID0}$ proof of $l(\varepsilon)$. When $r(v)$ is not (Cut) for any $v \in N$,* Pr *is called a* cut-free $CLBI^{\omega}_{ID0}$ proof.

Both the proof systems LBI_{ID0} and $CLBI^{\omega}_{ID0}$ are subsystems of $CLBI^{\omega}_{ID}$ in [3], and hence their soundness follows from the soundness of $CLBI^{\omega}_{ID}$.

▶ **Theorem 16** (Soundness of LBI_{ID0} and $CLBI^{\omega}_{ID0}$). *If $\Gamma \vdash \phi$ is provable in either LBI_{ID0} or $CLBI^{\omega}_{ID0}$, then $\Gamma \vdash \phi$ is valid.*

3 Proof Unrolling

In this section, we introduce a new technique, called *proof unrolling*, for constructing a non-cyclic proof from a given cyclic proof: we first define a non-cyclic proof system that is a variant of LBI_{ID0} (say LBI'_{ID0}), and then, for a cyclic proof of $\Gamma \vdash \phi$ in $CLBI^{\omega}_{ID0}$ and Γ' obtained from Γ by unfolding inductive propositions, construct a non-cyclic proof of $\Gamma' \vdash \phi$ in LBI'_{ID0}.

▶ **Definition 17** (Unfolded formula and unfolded bunch). *The set* $\mathrm{Unf}(\phi)$ *of* unfolded formulas *of ϕ is defined with auxiliary sets $\mathrm{Unf}^m(\phi)$, which is the set of formulas without inductive propositions obtained by at most m-time unfoldings of inductive predicates in ϕ, as follows:*

$$
\begin{aligned}
\mathrm{Unf}(\phi) &= \bigcup_m \mathrm{Unf}^{(m)}(\phi);\\
\mathrm{Unf}^{(m)}(\phi) &= \{\phi\} \qquad \textit{(when ϕ is I, $\top$, or an atomic proposition)};\\
\mathrm{Unf}^{(m)}(\phi_1 * \phi_2) &= \{\phi'_1 * \phi'_2 \mid \phi'_1 \in \mathrm{Unf}^{(m)}(\phi_1) \textit{ and } \phi'_2 \in \mathrm{Unf}^{(m)}(\phi_2)\};\\
\mathrm{Unf}^{(m)}(\phi_1 \wedge \phi_2) &= \{\phi'_1 \wedge \phi'_2 \mid \phi'_1 \in \mathrm{Unf}^{(m)}(\phi_1) \textit{ and } \phi'_2 \in \mathrm{Unf}^{(m)}(\phi_2)\};\\
\mathrm{Unf}^{(0)}(P) &= \emptyset;\\
\mathrm{Unf}^{(m+1)}(P) &= \bigcup_{\phi \in \Phi_P} \mathrm{Unf}^{(m)}(\phi).
\end{aligned}
$$

The set $\mathrm{Unf}(\Gamma)$ *of* unfolded bunches *of Γ is defined as follows:*

$$
\begin{aligned}
\mathrm{Unf}(\Gamma) &= \mathrm{Unf}(\phi) \qquad \textit{(when $\Gamma = \phi$)}\\
\mathrm{Unf}(\Gamma, \Gamma') &= \{\Delta, \Delta' \mid \Delta \in \mathrm{Unf}(\Gamma) \textit{ and } \Delta' \in \mathrm{Unf}(\Gamma')\}\\
\mathrm{Unf}(\Gamma; \Gamma') &= \{\Delta; \Delta' \mid \Delta \in \mathrm{Unf}(\Gamma) \textit{ and } \Delta' \in \mathrm{Unf}(\Gamma')\}.
\end{aligned}
$$

Before discussing the proof unrolling technique, we define an weakened variant of the rule (Ax) in LBI_{ID0}.

▶ **Definition 18.** *We consider the following inference rule.*

$$
\frac{}{\phi \vdash \psi}\,(Ax') \qquad \phi \in \mathrm{Unf}(\psi)
$$

We define LBI'_{ID0} as LBI_{ID0} in which (Ax) is replaced by (Ax').

▶ **Lemma 19.** *If a sequent is cut-free provable in LBI'_{ID0}, then it is cut-free provable in LBI_{ID0}, and hence LBI'_{ID0} is sound.*

Proof. It is sufficient to prove $\phi \vdash \psi$ is cut-free provable in LBI_{ID0} for any n and $\phi \in \mathrm{Unf}^{(n)}(\psi)$, and it is proved by induction on (n, ψ). The only nontrivial case is the case where $n > 1$, $\psi = P$, and $\phi \in \mathrm{Unf}^{(n)}(P)$. In this case, for some definition clause ψ' of P, we have $\phi \in \mathrm{Unf}^{(n-1)}(\psi')$. By the induction hypothesis, we have $\phi \vdash \psi'$, and hence we have $\phi \vdash P$ by the rule (UR). ◀

▶ **Lemma 20.** *If $\Delta \in \mathrm{Unf}(\Gamma)$, then $\Delta \models \Gamma$ holds.*

Proof. It is proved by induction on Γ and the soundness of the rule (Ax') by Lemma 19. ◀

▶ **Lemma 21.** *If an LBI'_{ID0} proof contains a finite path $(v_i)_{i\leq n}$ such that $l(v_0) = \Gamma \vdash \phi$, $l(v_n) = \Gamma' \vdash \phi$, and $r(v_i)$ is either (W), (C), (E), or $(*L)$ for $0 \leq i < n$, then we have $\Gamma \models \Gamma'$.*

Proof. It is sufficient to show that $\Gamma \models \Gamma'$ holds for any rule instance

$$\frac{\Gamma' \vdash \phi}{\Gamma \vdash \phi}\ (R),$$

where (R) is either (W), (C), (E), or $(*L)$. It is easily proved. ◀

▶ **Lemma 22.** *Let (R) be a rule of $CLBI^{\omega}_{ID0}$ except for (Cut). If $\Gamma \vdash \phi$ is inferred by (R) from the premises $\Gamma_1 \vdash \phi_1, \ldots, \Gamma_n \vdash \phi_n$, and $\Delta \in \mathrm{Unf}(\Gamma)$, we have the following.*

1. *If $(R) = (Ax)$, $\Delta \vdash \phi$ is inferred by (Ax').*
2. *If $(R) = (UL)$, $\Delta \in \mathrm{Unf}(\Gamma_i)$ and $\phi = \phi_i$ hold for some i.*
3. *Otherwise, $\Delta \vdash \phi$ is inferred by (R) from $\Delta_1 \vdash \phi_1, \ldots, \Delta_n \vdash \phi_n$ for some $\Delta_i \in \mathrm{Unf}(\Gamma_i)$ $(1 \leq i \leq n)$.*

Proof.
1. By the definition of (Ax').
2. In the definition of $\Delta \in \mathrm{Unf}(\Gamma)$, we choose an inductive definition clause of P, which is unfolded by the rule (UL). If the clause is i-th one, we can choose a premise $\Gamma_i \vdash \phi$ such that $\Delta \in \mathrm{Unf}(\Gamma_i)$ holds.
3. If (R) is a left rule, by the definition of the unfolded bunches, $\Delta \vdash \phi$ contains the corresponding connectives of the principal formula in $\Gamma \vdash \phi$ for (R). Otherwise, it is easily proved. ◀

▶ **Definition 23** (UL path)**.** *A finite path $(v_i)_{i\leq m}$ in a cyclic proof (N, l, r, ρ) is called a UL path when $r(v_i)$ is either (UL) or (Bud) for any i such that $0 \leq i < m$.*

▶ **Lemma 24** (Proof unrolling)**.** *Let $\mathrm{Pr}_1 = (N_1, l_1, r_1, \rho_1)$ be a cut-free $CLBI^{\omega}_{ID0}$ proof of $\Gamma_1 \vdash \phi$ and $\Gamma_2 \in \mathrm{Unf}(\Gamma_1)$. We can construct a cut-free LBI'_{ID0} proof $\mathrm{Pr}_2 = (N_2, l_2, r_2)$ of $\Gamma_2 \vdash \phi$ accompanied with a mapping $f : N_2 \to N_1$ such that the following hold:*

- *$f(\varepsilon) = \varepsilon$.*
- *For any $v \in N_2$, $L(l_2(v)) \in \mathrm{Unf}(L(l_1(f(v))))$ and $R(l_2(v)) = R(l_1(f(v)))$.*
- *For any $v \in N_2$, there is a UL path $(v_i)_{0\leq i\leq m}$ in Pr_1 such that $v_0 = f(v)$, $r_1(v_m) = r_2(v)$, and $f(vn) = v_m n$.*

Proof. (Sketch) We can construct Pr_2 from Pr_1 by unrolling the cyclic structures and choosing the premises of (UL) depending on the definition of the unfolded bunch Γ_2. Lemma 22 guarantees that this construction works well and the global trace condition guarantees that the construction eventually terminates for the unfolded bunch Γ_2 since any infinite path in Pr_1 has an infinitely progressing trace. ◀

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$I \vdash I(2)$}
\RightLabel{(UR)}
\UnaryInfC{$I \vdash \mathrm{P}_A$}
\AxiomC{$\mathrm{P}_{AA} \vdash \mathrm{P}_A(8)(\dagger)$}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$A \vdash A(9)$}
\RightLabel{$(*R)$}
\BinaryInfC{$\mathrm{P}_{AA}, A \vdash \mathrm{P}_A * A(7)$}
\RightLabel{(UR)}
\UnaryInfC{$\mathrm{P}_{AA}, A \vdash \mathrm{P}_A(6)$}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$A \vdash A(10)$}
\RightLabel{$(*R)$}
\BinaryInfC{$(\mathrm{P}_{AA}, A), A \vdash \mathrm{P}_A * A(5)$}
\RightLabel{(UR)}
\UnaryInfC{$(\mathrm{P}_{AA}, A), A \vdash \mathrm{P}_A(4)$}
\RightLabel{$(*L)$}
\UnaryInfC{$\mathrm{P}_{AA} * A, A \vdash \mathrm{P}_A(3)$}
\RightLabel{$(*L)$}
\UnaryInfC{$\mathrm{P}_{AA} * A * A \vdash \mathrm{P}_A$}
\RightLabel{(UL)}
\BinaryInfC{$\mathrm{P}_{AA} \vdash \mathrm{P}_A(1)(\dagger)$}
\end{prooftree}
$$

Figure 3 $CLBI^{\omega}_{ID0}$ proof of $\mathrm{P}_{AA} \vdash \mathrm{P}_A$.

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{(Ax')}
\UnaryInfC{$I \vdash I(2)$}
\RightLabel{(UR)}
\UnaryInfC{$I \vdash \mathrm{P}_A(8)$}
\RightLabel{$(*L)$}
\UnaryInfC{$I, A \vdash \mathrm{P}_A * A(7)$}
\RightLabel{(UR)}
\UnaryInfC{$I, A \vdash \mathrm{P}_A(6)$}
\AxiomC{}
\RightLabel{(Ax')}
\UnaryInfC{$A \vdash A(10)$}
\RightLabel{$(*L)$}
\BinaryInfC{$(I, A), A \vdash \mathrm{P}_A * A(5)$}
\RightLabel{(UR)}
\UnaryInfC{$(I, A), A \vdash \mathrm{P}_A(4)$}
\RightLabel{$(*L)$}
\UnaryInfC{$I * A, A \vdash \mathrm{P}_A(3)$}
\RightLabel{$(*L)$}
\UnaryInfC{$I * A * A \vdash \mathrm{P}_A(8)$}
\AxiomC{}
\RightLabel{(Ax')}
\UnaryInfC{$A \vdash A(9)$}
\RightLabel{$(*R)$}
\BinaryInfC{$I * A * A, A \vdash \mathrm{P}_A * A(7)$}
\RightLabel{(UR)}
\UnaryInfC{$I * A * A, A \vdash \mathrm{P}_A(6)$}
\AxiomC{}
\RightLabel{(Ax')}
\UnaryInfC{$A \vdash A(10)$}
\RightLabel{$(*R)$}
\BinaryInfC{$(I * A * A, A), A \vdash \mathrm{P}_A * A(5)$}
\RightLabel{(UR)}
\UnaryInfC{$(I * A * A, A), A \vdash \mathrm{P}_A(4)$}
\RightLabel{$(*L)$}
\UnaryInfC{$I * A * A * A, A \vdash \mathrm{P}_A(3)$}
\RightLabel{$(*L)$}
\UnaryInfC{$I * A * A * A * A \vdash \mathrm{P}_A(1)$}
\end{prooftree}
$$

Figure 4 LBI'_{ID0} proof of $I * A * A * A * A \vdash \mathrm{P}_A$ constructed by proof unrolling.

Intuitively, a cyclic proof of $\Gamma \vdash \phi$ contains several (possibly infinite) cases according to the unfolding of inductive propositions in Γ. The proof unrolling technique takes one case among them by $\Gamma' \in \mathrm{Unf}(\Gamma)$ and extracts a non-cyclic proof of $\Gamma' \vdash \phi$ from the cyclic proof of $\Gamma \vdash \phi$.

▶ **Example 25.** We consider two inductive propositions P_A and P_{AA}, which are defined by

$$\mathrm{P}_A := I \mid \mathrm{P}_A * A \qquad\qquad \mathrm{P}_{AA} := I \mid \mathrm{P}_{AA} * A * A.$$

For these inductive propositions, the sequent $\mathrm{P}_{AA} \vdash \mathrm{P}_A$ is provable in $CLBI^{\omega}_{ID0}$ as Figure 3. The sequents marked $(\dagger)$ are corresponding bud and companion. The numbers (1), (2), ... are identifiers of sequents.

From this cyclic proof, we can construct an LBI'_{ID0} (non-cyclic) proof of $I*A*A*A*A \vdash P_A$ for $I * A * A * A * A \in \mathrm{Unf}(P_{AA})$ by the proof unrolling as Figure 4. The identifiers of sequents indicate the corresponding nodes in the cyclic proof, where we unroll the cycle at $(\dagger)$ twice, and for (UL) in the cyclic proof, we choose the right premise twice at (3) and the left premise at (2).

$$
\begin{prooftree}
\AxiomC{$\mathrm{P}_{AB} \vdash \mathrm{P}_{BA}(@)$}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$\mathrm{P}_A \vdash \mathrm{P}_A$}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$A \vdash A$}
\RightLabel{$(*R)$}
\BinaryInfC{$\mathrm{P}_A, A \vdash \mathrm{P}_A * A$}
\RightLabel{(UR)}
\UnaryInfC{$\mathrm{P}_A, A \vdash \mathrm{P}_A$}
\RightLabel{(UR)}
\UnaryInfC{$\mathrm{P}_A, A \vdash \mathrm{P}_{BA}$}
\AxiomC{$\mathrm{P}_{BA}, A \vdash \mathrm{P}_{BA}(\#)$}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$B \vdash B$}
\RightLabel{$(*R)$}
\BinaryInfC{$(\mathrm{P}_{BA}, A), B \vdash \mathrm{P}_{BA} * B$}
\RightLabel{(E)}
\UnaryInfC{$(\mathrm{P}_{BA}, B), A \vdash \mathrm{P}_{BA} * B$}
\RightLabel{(UR)}
\UnaryInfC{$(\mathrm{P}_{BA}, B), A \vdash \mathrm{P}_{BA}$}
\RightLabel{$(*L)$}
\UnaryInfC{$\mathrm{P}_{BA} * B, A \vdash \mathrm{P}_{BA}$}
\RightLabel{(UL)}
\BinaryInfC{$\mathrm{P}_{BA}, A \vdash \mathrm{P}_{BA}(\#)$}
\RightLabel{(Cut)}
\BinaryInfC{$\mathrm{P}_{AB}, A \vdash \mathrm{P}_{BA}$}
\RightLabel{$(*L)$}
\UnaryInfC{$\mathrm{P}_{AB} * A \vdash \mathrm{P}_{BA}\,(1)$}
\end{prooftree}
$$

is the subproof of the following proof figure:

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$I \vdash I$}
\RightLabel{(UR)}
\UnaryInfC{$I \vdash \mathrm{P}_A$}
\RightLabel{(UR)}
\UnaryInfC{$I \vdash \mathrm{P}_{BA}$}
\AxiomC{$\mathrm{P}_B \vdash \mathrm{P}_{BA}(\dagger)$}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$B \vdash B$}
\RightLabel{$(*R)$}
\BinaryInfC{$\mathrm{P}_B, B \vdash \mathrm{P}_{BA} * B$}
\RightLabel{(UR)}
\UnaryInfC{$\mathrm{P}_B, B \vdash \mathrm{P}_{BA}$}
\RightLabel{$(*L)$}
\UnaryInfC{$\mathrm{P}_B * B \vdash \mathrm{P}_{BA}$}
\RightLabel{(UL)}
\BinaryInfC{$\mathrm{P}_B \vdash \mathrm{P}_{BA}(\dagger)$}
\AxiomC{$\vdots$ the above proof figure}
\noLine
\UnaryInfC{$\mathrm{P}_{AB} * A \vdash \mathrm{P}_{BA}\,(1)$}
\RightLabel{(UL)}
\BinaryInfC{$\mathrm{P}_{AB} \vdash \mathrm{P}_{BA}(@)$}
\end{prooftree}
,
$$

Each bud marked $(\dagger)$, $(@)$, or $(\#)$ has its companion with the same mark.

Figure 5 $CLBI^{\omega}_{ID0}$ proof of $\mathrm{P}_{AB} \vdash \mathrm{P}_{BA}$.

4 Failure of Cut-Elimination

In this section, we give a counterexample of the cut-elimination property in $CLBI^{\omega}_{ID0}$. We fix the language Σ consisting of the atomic propositions A and B, and the inductive propositions P_{AB}, P_{BA}, P_A, and P_B. We also fix the set Φ of inductive definitions for P_{AB}, P_{BA}, P_A, and P_B defined by:

$$
\begin{aligned}
&\mathrm{P}_{AB} := \mathrm{P}_B \mid \mathrm{P}_{AB} * A; &\qquad& \mathrm{P}_A := I \mid \mathrm{P}_A * A;\\
&\mathrm{P}_{BA} := \mathrm{P}_A \mid \mathrm{P}_{BA} * B; && \mathrm{P}_B := I \mid \mathrm{P}_B * B.
\end{aligned}
$$

Intuitively, P_A and P_B mean $I * A^n$ and $I * B^m$ with arbitrary $n, m \geq 0$, respectively. P_{AB} and P_{BA} mean $(I * B^m) * A^n$ and $(I * A^m) * B^n$ with arbitrary $n, m \geq 0$, respectively. We note that P_{AB} and P_{BA} are logically equivalent in the standard models since the separating conjunction $*$ and the formula I are interpreted as a commutative monoid operator and the unit of it, respectively.

The intention of the name P_{AB} is that, during the unfolding of P_{AB}, A's appear first, and then B's appear in the unfolding of P_B. P_{BA} is also named by a similar intention.

Our main result will be obtained by showing the entailment $\mathrm{P}_{AB} \vdash \mathrm{P}_{BA}$ is a counter-example for the cut-elimination. We need to show two things: One is that $\mathrm{P}_{AB} \vdash \mathrm{P}_{BA}$ is provable in $CLBI^{\omega}_{ID0}$ with (Cut), and the other is that $\mathrm{P}_{AB} \vdash \mathrm{P}_{BA}$ is not cut-free provable in $CLBI^{\omega}_{ID0}$.

First, we show that $\mathrm{P}_{AB} \vdash \mathrm{P}_{BA}$ is provable in $CLBI^{\omega}_{ID0}$ with (Cut).

▶ **Proposition 26.** $\mathrm{P}_{AB} \vdash \mathrm{P}_{BA}$ *is provable in* $CLBI^{\omega}_{ID0}$.

Proof. The proof figures in Figure 5 show this proposition. ◀

To show that $\mathrm{P}_{AB} \vdash \mathrm{P}_{BA}$ is not cut-free provable in $CLBI^{\omega}_{ID0}$, we assume that it is cut-free provable to derive a contradiction. For this purpose, we will consider only the multiset model M_{multi} introduced in Example 9. We omit M_{multi} in the satisfaction relation, that is, $r \models \phi$ means $M_{\mathrm{multi}}, r \models \phi$. We write $\{\mathsf{a}^n\}$ for the multiset consisting of n a's.

We shall describe our proof approach before starting the formal discussion. We assume the existence of a cut-free cyclic proof of $\mathrm{P}_{AB} \vdash \mathrm{P}_{BA}$. By the proof unrolling, we can construct proofs of $\phi \vdash \mathrm{P}_{BA}$ in LBI'_{ID0} for any unfolded formula ϕ of P_{AB}. Hence we have proofs of $I * A^n \vdash \mathrm{P}_{BA}$ for arbitrary n. We consider parts of the proofs of $I * A^n \vdash \mathrm{P}_{BA}$ which contain the conclusion and do not contain the rule (UR). We call such parts the proof segments. In such a proof segment, $\{\mathsf{a}^n\} \in M_{\mathrm{multi}}$ satisfies every antecedent. Then, $\{\mathsf{a}^n\}$ also satisfies every antecedent in the corresponding part of the cyclic proof. Since the cyclic proof is finite, for a sufficiently large n, the antecedents cannot contain A^n, but they must contain either P_{AB} or $\top$, and then both $\{\mathsf{a}^n\}$ and $\{\mathsf{a}^n, \mathsf{b}\}$ satisfy the antecedents. On the other hand, since the proof segment does not contain (UR), every succedent is P_{BA}. When we unfold P_{BA}, we have to decide either P_A or $\mathrm{P}_{BA} * B$. However, neither of them can be satisfied by both $\{\mathsf{a}^n\}$ and $\{\mathsf{a}^n, \mathsf{b}\}$.

To achieve our plan, we prepare some definitions and theorems.

▶ **Definition 27** (P_{AB}-formula and P_{AB}-bunch). *A P_{AB}-formula $\phi_{\mathrm{P}_{AB}}$ is defined as follows:*

$$\phi_{\mathrm{P}_{AB}} ::= I \mid \top \mid A \mid B \mid \mathrm{P}_{AB} \mid \mathrm{P}_B \mid \mathrm{P}_{AB} * A \mid \mathrm{P}_B * B.$$

A P_{AB}-bunch $\Gamma_{\mathrm{P}_{AB}}$ is a bunch all of whose leaves are P_{AB}-formulas.

▶ **Lemma 28.** *Let (N, l, r, ρ) be a cut-free $CLBI^{\omega}_{ID0}$ proof of $\mathrm{P}_{AB} \vdash \phi$. For any $v \in N$, $L(l(v))$ is a P_{AB}-bunch.*

Proof. This lemma is proved by induction on the size of N. ◀

▶ **Lemma 29.** *Let Γ be a P_{AB}-bunch. If we have $\{\mathsf{a}^i\} \models \Gamma$ for $i > 2^{|\Gamma|}$, then we also have $\{\mathsf{a}^i, \mathsf{b}\} \models \Gamma$.*

Proof. It is proved by induction on Γ. The only nontrivial case is the case of $\Gamma = \Delta, \Delta'$. In this case, we have $\{\mathsf{a}^j\} \models \Delta$ and $\{\mathsf{a}^{j'}\} \models \Delta'$ for some j and j' such that $j + j' = i$. By the assumption, we have $i > 2 \cdot 2^{|\Gamma|-1} > 2 \cdot 2^{\max(|\Delta|, |\Delta'|)}$. Hence either $j > 2^{|\Delta|}$ or $j' > 2^{|\Delta'|}$ holds. By the induction hypothesis, we have either $\{\mathsf{a}^j, \mathsf{b}\} \models \Delta$ or $\{\mathsf{a}^{j'}, \mathsf{b}\} \models \Delta'$ holds. Therefore we have $\{\mathsf{a}^i, \mathsf{b}\} \models \Gamma$. ◀

▶ **Definition 30** (Proof segment). *Let $Pr_1 = (N_1, l_1, r_1)$ be a LBI'_{ID0} proof. $Pr = (N_2, l_2, r_2)$ is a proof segment of Pr_1 when it enjoys the following conditions:*
- *$N_2 \subseteq N_1$ holds, and $vi \in N_2$ implies $v \in N_2$.*
- *For any $v \in N_2$, $l_2(v) = l_1(v)$ and $r_2(v) = r_1(v)$ hold.*

Note that leaves of a proof segment are not necessarily assigned the rule (Ax').

▶ **Proposition 31.** $\mathrm{P}_{AB} \vdash \mathrm{P}_{BA}$ *is not cut-free provable in $CLBI^{\omega}_{ID0}$.*

Proof. This proposition is shown by contradiction. We assume that there is a cut-free $CLBI^{\omega}_{ID0}$ proof $\mathrm{Pr}_1 = (N_1, l_1, r_1, \rho_1)$ of $\mathrm{P}_{AB} \vdash \mathrm{P}_{BA}$. Let $n = max\{|L(l_1(v))| \mid v \in N_1\}$.

Since $I * A^{2^n+1} \in \mathrm{Unf}(\mathrm{P}_{AB})$, we can construct a cut-free LBI'_{ID0} proof $\mathrm{Pr}_2 = (N_2, l_2, r_2)$ of $I * A^{2^n+1} \vdash \mathrm{P}_{BA}$ and the mapping $f : N_2 \to N_1$ by Lemma 24.

Let $\mathrm{Pr}_2^{BA} = (N_2^{BA}, l_2^{BA}, r_2^{BA})$ be the biggest proof segment of Pr_2 such that $R(l_2^{BA}(v)) = \mathrm{P}_{BA}$ for any $v \in N_2^{BA}$. Note that Pr_2^{BA} is not empty since $R(l_2(\varepsilon)) = \mathrm{P}_{BA}$. For any $v \in N_2^{BA}$, $r_2^{BA}(v)$ is either (W), (C), $(*L)$, (E), (Ax'), or (UR). In particular, (Ax') and

(UR) are only applied to leaves of Pr_2^{BA}, and the other rules are not applied to leaves since these rules do not change the succedents. We have $\{\mathsf{a}^{2^n+1}\} \models I * A^{2^n+1}$ in the multiset model, and hence we have $\{\mathsf{a}^{2^n+1}\} \models L(l_2^{BA}(v))$ holds for any $v \in N_2^{BA}$ by Lemma 21.

Let v be a leaf node of Pr_2^{BA}. Then, $r_2^{BA}(v)$ is either (Ax') or (UR).

In the case of (Ax'), by Lemma 24, there is a UL path from $f(v)$ to some v' in Pr_1 such that $r_1(v') = (Ax)$. By Lemma 28, $l_1(v') = \Gamma \vdash \mathrm{P}_{BA}$ for some P_{AB}-bunch Γ, and it contradicts $r_1(v') = (Ax)$ since P_{BA} is not a P_{AB}-bunch. Hence, (Ax') is not the case.

In the case of (UR), let v' be the premise of v in Pr_2. Since we have $l_2^{BA}(v) = l_2(v) = \Gamma \vdash \mathrm{P}_{BA}$ for some Γ, $l_2(v')$ is either $\Gamma \vdash \mathrm{P}_{BA} * B$ or $\Gamma \vdash \mathrm{P}_A$, but it is proved as follows that both of them are not the case.

For $l_2(v') = \Gamma \vdash \mathrm{P}_{BA} * B$, we have $\{\mathsf{a}^{2^n+1}\} \models \Gamma$ and $\{\mathsf{a}^{2^n+1}\} \not\models \mathrm{P}_{BA} * B$, and hence $\Gamma \vdash \mathrm{P}_{BA} * B$ is invalid. It contradicts the soundness of LBI'_{ID0}. Hence, this is not the case.

For $l_2(v') = \Gamma \vdash \mathrm{P}_A$, we have $l_1(f(v')) = \Gamma' \vdash \mathrm{P}_A$ for some P_{AB}-bunch Γ' such that $\Gamma \in \mathrm{Unf}(\Gamma')$. Then, we have $\{\mathsf{a}^{2^n+1}\} \models \Gamma'$ by Lemma 20, and $\{\mathsf{a}^{2^n+1}, \mathsf{b}\} \models \Gamma'$ by Lemma 29 and $2^n + 1 > 2^{|\Gamma'|}$. Since $\{\mathsf{a}^{2^n+1}, \mathsf{b}\} \not\models \mathrm{P}_A$, it contradicts the soundness of LBI'_{ID0}. Hence, this is not the case.

Therefore, there is no possible rule at the leaves of Pr_2^{BA}, and hence there is no cut-free $CLBI^{\omega}_{ID0}$ proof of $\mathrm{P}_{AB} \vdash \mathrm{P}_{BA}$. ◀

▶ **Theorem 32** (Failure of cut-elimination in $CLBI^{\omega}_{ID0}$). *$CLBI^{\omega}_{ID0}$ does not enjoy the cut-elimination property.*

Proof. By Proposition 26 and Proposition 31, $\mathrm{P}_{AB} \vdash \mathrm{P}_{BA}$ is a counterexample. ◀

This result is easily extended to the original cyclic proof system $CLBI^{\omega}_{ID}$ in [3], which contains full logical connectives of the bunched logic and inductive predicates with arbitrary arity.

▶ **Corollary 33** (Failure of cut elimination in $CLBI^{\omega}_{ID}$). *$CLBI^{\omega}_{ID}$ does not enjoy cut-elimination property.*

Proof. $\mathrm{P}_{AB} \vdash \mathrm{P}_{BA}$ is a counterexample. It is provable in $CLBI^{\omega}_{ID}$, since the proof in Figure 5 is also a $CLBI^{\omega}_{ID}$ proof with cuts. If there is a cut-free $CLBI^{\omega}_{ID}$ proof of $\mathrm{P}_{AB} \vdash \mathrm{P}_{BA}$, it is a cut-free $CLBI^{\omega}_{ID0}$ proof since neither logical connectives other than $*$, inductive predicates accompanied by some arguments, nor first-order terms can occur in the proof. ◀

5 Conclusion and Future Work

We have proved by the proof unrolling technique that the cut-elimination fails for the cyclic proof system of the bunched logic $CLBI^{\omega}_{ID}$ in [3] only with nullary inductive predicates.

For a logic with a connective representing resource composition such as the separation logic and the multiplicative linear logic, we can straightforwardly adapt our proof technique to the cyclic proof system for the logic.

For the separation logic, we allow arbitrary substitution in the definition of Unf for existentially quantified variables as

$$\mathrm{Unf}^{(m+1)}(P) = \bigcup_{\exists \vec{x}.\phi(\vec{x}) \in \Phi_P \text{ and } \vec{t}\,:\text{ arbitrary terms}} \mathrm{Unf}^{(m)}(\phi(\vec{t})),$$

and we reread the atomic propositions A and B in our proof as to the following nullary predicates, for example,

$$A = \exists x(x \mapsto x) \qquad\qquad B = \exists x(x \mapsto \mathrm{nil}),$$

and then we can prove that the cut-elimination fails for the cyclic proof system of the separation logic with only nullary predicates.

We can adapt the proof unrolling to cyclic proof system $CLKID^\omega$ [6] for the first-order logic when we consider a cut-free cyclic proof that contains only positive occurrences of inductive predicates. However, the proof in Section 4 depends on the multiset model, and it is an interesting question if we can apply our proof idea for the first-order logic. Another direction of future work is to find reasonable restrictions for the inductive predicates to recover the cut-elimination property in the cyclic proof systems. Our result shows that the restriction on the arity of predicates is not sufficient.

References

1 T. Antonopoulos, N. Gorogiannis, C. Haase, M. Kanovich, and J. Ouaknine. Foundations for decision problems in separation logic with general inductive predicates. In *Proceedings of FoSSaCS 2014*, volume 8412 of *LNCS*, pages 411–425, 2014.

2 D. Baelde, A. Doumane, and A. Saurin. Infinitary proof theory: the multiplicative additive case. In *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *LIPIcs*, pages 42:1–42:17, 2016.

3 J. Brotherston. Formalised inductive reasoning in the logic of bunched implications. In *Proceedings of SAS '07*, volume 4634 of *LNCS*, pages 87–103, 2007.

4 J. Brotherston, D. Distefano, and R. L. Petersen. Automated cyclic entailment proofs in separation logic. In *Proceedings of CADE-23*, volume 6803 of *LNAI*, pages 131–146, 2011.

5 J. Brotherston, N. Gorogiannis, and R. L. Petersen. A generic cyclic theorem prover. In *Proceedings of APLAS 2012*, volume 7705 of *LNCS*, pages 350–367, 2012.

6 J. Brotherston and A. Simpson. Sequent calculi for induction and infinite descent. *Journal of Logic and Computation*, 21(6):1177–1216, 2011.

7 J. Fortier and L. Santocanale. Cuts for circular proofs: semantics and cut-elimination. In *Computer Science Logic 2013 (CSL 2013)*, volume 23 of *LIPIcs*, pages 248–262, 2013.

8 D. Kimura, K. Nakazawa, T. Terauchi, and H. Unno. Failure of cut-elimination in cyclic proofs of separation logic. *Comupter Software*, 37(1):39–52, 2020.

9 P. O'Hearn and D. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

10 D. J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Springer, 2002.

11 J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS 2002*, pages 55–74, 2002.

12 M. Tatsuta and D. Kimura. Separation logic with monadic inductive definitions and implicit existentials. In *Proceedings of APLAS 2015*, volume 9458 of *LNCS*, pages 69–89, 2015.

A Functional Abstraction of Typed Invocation Contexts

Youyou Cong
Tokyo Institute of Technology, Japan

Chiaki Ishio
Ochanomizu University, Tokyo,Japan

Kaho Honda
Ochanomizu University, Tokyo, Japan

Kenichi Asai
Ochanomizu University, Tokyo, Japan

Abstract

In their paper "A Functional Abstraction of Typed Contexts", Danvy and Filinski show how to derive a type system of the `shift` and `reset` operators from a CPS translation. In this paper, we show how this method scales to Felleisen's `control` and `prompt` operators. Compared to `shift` and `reset`, `control` and `prompt` exhibit a more dynamic behavior, in that they can manipulate a *trail* of contexts surrounding the invocation of captured continuations. Our key observation is that, by adopting a functional representation of trails in the CPS translation, we can derive a type system that allows fine-grain reasoning of programs involving manipulation of invocation contexts.

2012 ACM Subject Classification Theory of computation → Functional constructs; Theory of computation → Control primitives; Theory of computation → Type structures

Keywords and phrases delimited continuations, control operators, control and prompt, CPS translation, type system

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.12

Supplementary Material *Model (Agda Formalization)*: `https://github.com/YouyouCong/fscd21-artifact`; archived at `swh:1:dir:9eaf9840fc9b223e030f633c3f9b3b5ea7b47bc6`

Funding *Youyou Cong*: supported in part by JSPS KAKENHI under Grant No. 19K24339.
Kenichi Asai: supported in part by JSPS KAKENHI under Grant No. JP18H03218.

Acknowledgements We sincerely thank the reviewers for their constructive feedback.

1 Introduction

Delimited continuations have been proven useful in diverse domains. Their applications range from representation of monadic effects [19], to formalization of partial evaluation [13], and to implementation of automatic differentiation [41]. As a means to handle delimited continuations, researchers have designed a variety of *control operators* [18, 15, 21, 16, 32]. Among them, Danvy and Filinski's `shift`/`reset` operators [15] have a solid theoretical foundation: there are a canonical CPS translation [15], a general type system [14], and a set of equational axioms [25]. Recent work by Materzok and Biernacki [32, 31] has also fostered understanding of `shift0` and `reset0`, by establishing similar artifacts for these operators. Other variants, however, are not as well-understood as the aforementioned ones, due to their complex semantics.

Understanding the subtleties of control operators is important, especially given the rapid adoption of *algebraic effects and handlers* [36, 6] observed in the past decade. Effect handlers can be thought of as a form of exception handlers that provide access to delimited continuations. As suggested by the similarity in the functionality, effect handlers have a close connection with control operators [20, 35], and in fact, they are often implemented using control operators provided by the host language [27, 28]. This means, a well-established theory of control operators is crucial for safer and more efficient implementation of effect handlers.

In this paper, we formalize a typed calculus of `control` and `prompt`, a pair of control operators proposed by Felleisen [18]. These operators bring an interesting behavior into programs: when a captured continuation k is invoked, the subsequent computation may capture the context surrounding the invocation of k. From a practical point of view, the ability to manipulate invocation contexts is useful for implementing sophisticated algorithms, such as list reversing [8] and breadth-first traversal [10]. From a theoretical perspective, on the other hand, this ability makes it hard to type programs in a way that fully reflects their runtime behavior.

We address the challenge with typing by rigorously following Danvy and Fillinski's [14] recipe for building a type system of a delimited control calculus. The idea is to analyze the CPS translation of the calculus, and identify all the constraints that are necessary for making a translated expression well-typed. In fact, the recipe has already been applied to the `control` and `prompt` [26] operators, but the type system obtained is not satisfactory for two reasons. First, the type system imposes certain restrictions on the contexts in which a captured continuation may be invoked. Second, the type system does not precisely describe the way contexts compose and propagate during evaluation. We show that, by choosing a right representation of invocation contexts in the CPS translation, we can build a type system without such limitations.

Below is a summary of our specific contributions:

- We present a type system of `control` and `prompt` that allows fine-grain reasoning of programs involving manipulation of invocation contexts. The type system is the `control`/`prompt`-equivalent of Danvy and Filinski's [14] type system for `shift`/`reset`, in that it incorporates all and only constraints that are imposed by the CPS translation.
- We prove three properties of our calculus: type soundness, type preservation of the CPS translation, and termination of well-typed programs. Among these, termination relies on the precise typing of invocation contexts available in our calculus; indeed, the property does not hold for the existing type system of `control` and `prompt` [26].

We begin with an informal account of `control` and `prompt` (Section 2), highlighting the dynamic behavior of these operators. We next formalize an untyped calculus of `control`/`prompt` (Section 3) and its CPS translation (Section 4), which is equivalent to the translation given by Shan [40]. Then, from the CPS translation, we derive a type system of our calculus (Section 5), and prove its properties (Section 6). Lastly, we discuss related work (Section 7) and conclude with future directions (Section 8).

As an artifact, we provide a formalization of our calculus and proofs in the Agda proof assistant [34]. The code is checked using Agda version 2.6.0.1, and is available online at:

`https://github.com/YouyouCong/fscd21-artifact`

Relation to Prior Work. This is an updated and extended version of our previous paper [2]. The primary contributions of this paper are a complete proof of type soundness of the proposed calculus, and a proper formalization of the target language of the CPS translation. We have also changed the title to clarify the kind of contexts considered in the paper.

2 Control and Prompt

As a motivating example, consider the following program:

$$\langle (\mathcal{F}k_1.\, \texttt{is0}\ (k_1\ 5)) + (\mathcal{F}k_2.\, \texttt{b2s}\ (k_2\ 8))\rangle$$

Throughout the paper, we write $\mathcal{F}$ to mean `control` and $\langle\rangle$ to mean `prompt`. We also assume two primitive functions: `is0`, which tells us if a given integer is zero or not, and `b2s`, which converts a boolean into a string `"true"` or `"false"`.

Under the call-by-value, left-to-right evaluation strategy, the above program evaluates in the following way:

$$\begin{aligned}
&\langle (\mathcal{F}k_1.\, \texttt{is0}\ (k_1\ 5)) + (\mathcal{F}k_2.\, \texttt{b2s}\ (k_2\ 8))\rangle\\
&= \langle \texttt{is0}\ (k_1\ 5)\,[\lambda x.\, x + (\mathcal{F}k_2.\, \texttt{b2s}\ (k_2\ 8))/k_1]\rangle\\
&= \langle \texttt{is0}\ (5 + (\mathcal{F}k_2.\, \texttt{b2s}\ (k_2\ 8)))\rangle\\
&= \langle \texttt{b2s}\ (k_2\ 8)\,[\lambda x.\, \texttt{is0}\ (5 + x)/k_2]\rangle\\
&= \langle \texttt{b2s}\ (\texttt{is0}\ (5 + 8))\rangle\\
&= \langle \texttt{b2s}\ (\texttt{is0}\ 13)\rangle\\
&= \langle \texttt{b2s}\ \texttt{false}\rangle\\
&= \langle \texttt{"false"}\rangle\\
&= \texttt{"false"}
\end{aligned}$$

The first `control` operator captures the delimited context up to the enclosing `prompt`, namely $[.] + (\mathcal{F}k_2.\, \texttt{b2s}\ (k_2\ 8))$ (where $[.]$ denotes a hole). The captured context is then reified into a function $\lambda x.\, x + (\mathcal{F}k_2.\, \texttt{b2s}\ (k_2\ 8))$, and evaluation shifts to the body $\texttt{is0}\ (k_1\ 5)$, where k_1 is the reified continuation. After β-reducing the invocation of k_1, we obtain another `control` in the evaluation position. This `control` captures the context $\texttt{is0}\ (5 + [.])$, which is a composition of *two* contexts: the addition context originally surrounding the `control` construct, and the application of `is0` surrounding the invocation of k_1. The context is then reified into a function $\lambda x.\, \texttt{is0}\ (5 + x)$, and evaluation shifts to the body $\texttt{b2s}\ (k_2\ 8)$, where k_2 is the reified continuation. By β-reducing the invocation of k_2, we obtain the expression $\texttt{b2s}\ (\texttt{is0}\ (5 + 8))$, where the original delimited context, the invocation context of k_1, and the invocation context of k_2 are all composed together. The expression returns the value `"false"` to the enclosing `prompt` clause, and the evaluation of the whole program finishes with this value.

From the above example, we can make two observations. First, a `control` operator can capture the context surrounding the invocation of a previously captured continuation. More generally, `control` may capture a *trail* of such invocation contexts. The ability comes from the absence of the delimiter in the body of captured continuations. Indeed, if we replace `control` with `shift` ($\mathcal{S}$) in the above program, the second `shift` would have no access to the context $\texttt{is0}\ [.]$, since the first `shift` would insert a `reset` into the continuation k_1. As a consequence, the program gets stuck after the application of k_2.

$$\begin{aligned}
&\langle (\mathcal{S}k_1.\, \texttt{is0}\ (k_1\ 5)) + (\mathcal{S}k_2.\, \texttt{b2s}\ (k_2\ 8))\rangle\\
&= \langle \texttt{is0}\ (k_1\ 5)\,[\lambda x.\, \langle x + (\mathcal{S}k_2.\, \texttt{b2s}\ (k_2\ 8))\rangle/k_1]\rangle\\
&= \langle \texttt{is0}\ \langle 5 + (\mathcal{S}k_2.\, \texttt{b2s}\ (k_2\ 8))\rangle\rangle\\
&= \langle \texttt{is0}\ \langle \texttt{b2s}\ (k_2\ 8)\,[\lambda x.\, \langle 5 + x\rangle/k_2]\rangle\rangle\\
&= \langle \texttt{is0}\ \langle \texttt{b2s}\ \langle 5 + 8\rangle\rangle\rangle\\
&= \langle \texttt{is0}\ \langle \texttt{b2s}\ 13\rangle\rangle
\end{aligned}$$

Syntax

$v ::= c \mid x \mid \lambda x.\, e$ Values $\qquad e ::= v \mid e\ e \mid \mathcal{F}k.\, e \mid \langle e \rangle$ Expressions

Evaluation Contexts

$E ::= [.] \mid E\ e \mid v\ E \mid \langle E \rangle$ General Contexts

$F ::= [.] \mid F\ e \mid v\ F$ Pure Contexts

Reduction Rules

$$E[(\lambda x.\, e)\ v] \rightsquigarrow E[e\,[v/x]] \qquad (\beta)$$

$$E[\langle F[\mathcal{F}k.\, e]\rangle] \rightsquigarrow E[\langle e\,[\lambda x.\, F[x]/k]\rangle] \qquad (\mathcal{F})$$

$$E[\langle v \rangle] \rightsquigarrow E[v] \qquad (\mathcal{P})$$

Figure 1 $\lambda_{\mathcal{F}}$: A Calculus of `control` and `prompt`.

The second observation is that a trail of invocation contexts can be *heterogeneous*. In our particular example, the first continuation k_1 is called in a `int`-to-`bool` context, whereas the second continuation k_2 is called in a `bool`-to-`string` context. These are apparently distinct types, and furthermore, the input and output types of each context are also different.

It turns out that our motivating example would be judged ill-typed by the existing type system for `control` and `prompt` [26]. This is because the type system imposes the following restrictions on the type of invocation contexts.

- All invocation contexts within a `prompt` clause must have the same type.
- For each invocation context, the input and output types must be the same.

We claim that, a fully general type system of `control` and `prompt` should be more flexible about the type of invocation contexts. Now the question is: Is it possible to allow such flexibility? Our answer is "yes". As we will see in Section 5, we can build a type system that accommodates invocation contexts having varying types, and that accepts our motivating example as a well-typed program.

3 $\lambda_{\mathcal{F}}$: A Calculus of `control` and `prompt`

In Figure 1, we present $\lambda_{\mathcal{F}}$, a λ-calculus featuring the `control` and `prompt` operators. The calculus has a separate syntactic category for values, which, in addition to variables and abstractions, has a set of constants c, such as integers, booleans, and string literals. Expressions consist of values, application, and delimited control constructs `control` and `prompt`.

We equip $\lambda_{\mathcal{F}}$ with a call-by-value, left-to-right evaluation strategy. As is usual with delimited control calculi, there are two groups of evaluation contexts: general contexts (E) and pure contexts (F). Their difference is that general contexts may contain `prompt` surrounding a hole, while pure contexts can never have such `prompt`. The distinction is used in the reduction rule ($\mathcal{F}$) of `control`, which says, `control` always captures the context up to the nearest enclosing `prompt`. In the reduct, we see that the body of a captured continuation is *not* surrounded by `prompt`, as we observed in the previous section. On the other hand, the body of `control` is evaluated in a `prompt` clause. The reduction rule ($\mathcal{P}$) for `prompt` simply removes a delimiter surrounding a value.

Note that $\lambda_{\mathcal{F}}$ is currently presented as an untyped calculus. We will introduce types in Section 5, according to the CPS translation to be defined in the next section.

Syntax

$$v ::= c \mid x \mid \lambda x.\, e \mid () \qquad \text{Values}$$
$$e ::= v \mid e\ e \mid (\texttt{case}\ t\ \texttt{of}\ () \Rightarrow e \mid k \Rightarrow e) \qquad \text{Expressions}$$

Evaluation Contexts

$$E ::= [.] \mid E\ e \mid v\ E \mid (\texttt{case}\ E\ \texttt{of}\ () \Rightarrow e \mid k \Rightarrow e)$$

Reduction Rules

$$E[(\lambda x.\, e)\ v] \rightsquigarrow E[e\,[v/x]] \qquad (\beta)$$
$$E[\texttt{case}\ ()\ \texttt{of}\ () \Rightarrow e_1 \mid k \Rightarrow e_2] \rightsquigarrow E[e_1] \qquad (\textsc{case-}())$$
$$E[\texttt{case}\ v\ \texttt{of}\ () \Rightarrow e_1 \mid k \Rightarrow e_2] \rightsquigarrow E[e_2\,[v/k]] \qquad (\textsc{case-}k)$$

Figure 2 λ_C: Target Calculus of CPS Translation.

4 CPS Translation

As we mentioned earlier, the type system of a delimited control calculus is often derived from a translation into continuation-passing style (CPS) [14]. When the source calculus has `control` and `prompt`, a CPS translation exposes both continuations and trails of invocation contexts. Trails can be represented either as a list of functions [8, 9] or as a composition of functions [40]. While previous work [26] on typing `control` and `prompt` adopts the list representation, we adopt the functional representation, as it fits better for the purpose of building a general type system (see Section 5 for details).

4.1 λ_C: Target Calculus of CPS Translation

In Figure 2, we define the target calculus of the CPS translation, which we call λ_C. The calculus is a pure, call-by-value λ-calculus featuring the unit value (), which represents an empty trail, and a case analysis construct, which allows inspection of trails. Note that a non-empty trail is represented as a regular function.

As in $\lambda_{\mathcal{F}}$, we evaluate λ_C programs under a call-by-value, left-to-right strategy. The particular choice of evaluation strategy is not necessary in our setting, but it is mandatory if the source and target calculi of the CPS translation have non-control effects (such as non-termination and I/O), because the result of the translation may have non-tail calls.

4.2 The CPS Translation

In Figure 3, we present the CPS translation $[\![\_]\!]$ from $\lambda_{\mathcal{F}}$ to λ_C, which is equivalent to the translation given by Shan [40]. The translation converts an expression into a function that takes in a continuation k and a trail t. The trail is the composition of the invocation contexts encountered so far, and is used together with a continuation to produce an answer (hence a continuation now receives a trail). Below, we detail the translation of three representative constructs: variables, `prompt`, and `control`.

Variables. The translation of a variable is an η-expanded version of the standard, call-by-value translation. The trivial use of the current trail t communicates the fact that a variable can never change the trail during evaluation. In general, the CPS translation of a pure expression uniformly calls the continuation with an unmodified trail.

Prompt. The translation of `prompt` has the same structure as the translation of variables, because `prompt` forms a pure expression. The translated body $[\![e]\!]$ is run with the identity continuation k_{id} and an empty trail $()$[1], describing the behavior of `prompt` as a control delimiter. Note that, in this CPS translation, the identity continuation is *not* the identity function. It receives a value v and a trail t, and behaves differently depending on whether t is empty or not. When t is empty, the identity continuation simply returns v. When t is non-empty, t must be a function composed of one or more invocation contexts, which looks like $\lambda x.\, E_n[\dots E_1[x] \dots]$. In this case, the identity continuation builds an expression $E_n[\dots E_1[v] \dots]$ by calling the trail with v and $()$.

Control. The translation of `control` shares the same pattern with the translation of `prompt`, because its body is evaluated in a `prompt` clause (as defined by the $(\mathcal{F})$ rule in Figure 1). The translated body $[\![e]\!]$ is applied a substitution that replaces the variable c with the trail $t \mathbin{@} (k' :: t')$, describing how the trail is extended when a captured continuation is invoked[2]. Recall that, in this CPS translation, trails are represented as functions. The @ and :: operators are thus defined as a function producing a function[3]. More specifically, these operators compose contexts in a first-captured, first-called manner (as we can see from the second clause of ::). Notice that :: is defined as a *recursive* function[4]. The reason is that, when extending a trail t with a continuation k, we need to produce a function that takes in a trail t', which in turn must be composed with a continuation k'.

The CPS translation is correct with respect to the definitional abstract machine given by Biernacka et al. [7]. The statement is proved by Shan [40], using the functional correspondence [1] between evaluators and abstract machines.

As a last note, let us mention here that the alternative CPS translation of `control` and `prompt`, where trails are represented as lists, can be obtained by replacing $()$ with the empty list, and the two operations @ and :: with ones that work on lists.

5 Type System

Having defined a CPS translation, we now derive a type system of $\lambda_{\mathcal{F}}$. We proceed in three steps. First, we specify the syntax of trail types (Section 5.1). Next, we identify an appropriate form of typing judgment (Section 5.2). Lastly, we define the typing rules of individual syntactic constructs (Section 5.3). In each step, we contrast our outcome with its counterpart in Kameyama and Yonezawa's [26] type system, showing how different representations of trails in the CPS translation lead to different typing principles.

---

1 The identity continuation k_{id} and the empty trail $()$ correspond to the `send` function and the `#f` value of Shan [40], respectively.

2 There is in fact a superficial difference between our CPS translation and Shan's original translation [40]. In the rule for `control`, we replace the continuation variable c with the function $\lambda x.\, \lambda k'.\, \lambda t'.\, k\ x\ (t \mathbin{@} (k' :: t'))$, while Shan replaces c with $\lambda x.\, \lambda k'.\, \lambda t'.\, (k :: t)\ x\ (k' :: t')$. However, by expanding the definition of @ and ::, we can easily see that the two functions are equivalent. We prefer the one that uses @ because it is closer to the abstract machine given by Biernacki et al. [9], as well as the list-based CPS translation derived from it.

3 The :: function is equivalent to Shan's `compose` function.

4 While recursive, the :: function is guaranteed to terminate, as the types of the two arguments become smaller in every three successive recursive calls (or they reach the base case in fewer steps).

$$
\begin{aligned}
[\![c]\!] &= \lambda k.\, \lambda t.\, k\ c\ t \\
[\![x]\!] &= \lambda k.\, \lambda t.\, k\ x\ t \\
[\![\lambda x.\, e]\!] &= \lambda k.\, \lambda t.\, k\ (\lambda x.\, \lambda k'.\, \lambda t'.\, [\![e]\!]\ k'\ t'\ )\ t \\
[\![e_1\ e_2]\!] &= \lambda k.\, \lambda t.\, [\![e_1]\!]\ (\lambda v_1.\, \lambda t_1.\, [\![e_2]\!]\ (\lambda v_2.\, \lambda t_2.\, v_1\ v_2\ k\ t_2)\ t_1)\ t \\
[\![\mathcal{F}c.\, e]\!] &= \lambda k.\, \lambda t.\, [\![e]\!]\, [\lambda x.\, \lambda k'.\, \lambda t'.\, k\ x\ (t \mathbin{@} (k' :: t'))/c]\ k_{id}\ () \\
[\![\langle e \rangle]\!] &= \lambda k.\, \lambda t.\, k\ ([\![e]\!]\ k_{id}\ ())\ t
\end{aligned}
$$

$$
\begin{aligned}
k_{id} &= \lambda v.\, \lambda t.\, \texttt{case}\ t\ \texttt{of}\ () \Rightarrow v \mid k \Rightarrow k\ v\ () \\
\_@\_ &= \lambda t.\, \lambda t'.\, \texttt{case}\ t\ \texttt{of}\ () \Rightarrow t' \mid k \Rightarrow k :: t' \\
\_::\_ &= \lambda k.\, \lambda t.\, \texttt{case}\ t\ \texttt{of}\ () \Rightarrow k \mid k' \Rightarrow \lambda v.\, \lambda t'.\, k\ v\ (k' :: t')
\end{aligned}
$$

Figure 3 CPS Translation of $\lambda_{\mathcal{F}}$ Expressions.

5.1 Syntax of Trail Types

Recall from Section 4.1 that, in λ_C, trails have two possible forms: $()$ or a function. Correspondingly, in $\lambda_{\mathcal{F}}$, trail types μ are defined by a two-clause grammar: $\bullet \mid \tau \rightarrow \langle \mu \rangle\ \tau'$. The latter type is interpreted in the following way.

- The trail accepts a value of type τ.
- The trail is to be composed with a context of type μ.
- After the composition, the trail produces a value of type τ'.

Put differently, τ is the input type of the innermost invocation context, τ' is the output type of the context to be composed in the future, and μ is the type of this future context.

To better understand non-empty trail types, let us revisit the example from Section 2.

$$
\begin{aligned}
&\langle (\mathcal{F}k_1.\, \texttt{is0}\ (k_1\ 5)) + (\mathcal{F}k_2.\, \texttt{b2s}\ (k_2\ 8)) \rangle \\
&= \langle \texttt{is0}\ (k_1\ 5)\, [\lambda x.\, x + (\mathcal{F}k_2.\, \texttt{b2s}\ (k_2\ 8))/k_1] \rangle \\
&= \langle \texttt{is0}\ (5 + (\mathcal{F}k_2.\, \texttt{b2s}\ (k_2\ 8))) \rangle \\
&= \langle \texttt{b2s}\ (k_2\ 8)\, [\lambda x.\, \texttt{is0}\ (5 + x)/k_2] \rangle \\
&= \langle \texttt{b2s}\ (\texttt{is0}\ (5 + 8)) \rangle \\
&= \texttt{"false"}
\end{aligned}
$$

When the continuation k_1 is invoked, the trail is extended with the context `is0` $[.]$. This context will be composed with the invocation context `b2s` $[.]$ of k_2 later in the reduction sequence. Therefore, the trail at this point is given type $\texttt{int} \rightarrow \langle \texttt{bool} \rightarrow \langle \bullet \rangle\ \texttt{string} \rangle\ \texttt{string}$, consisting of the input type of `is0`, the type of `b2s`, and the output type of `b2s`.

When the continuation k_2 is invoked, the trail is extended with the context `b2s` $[.]$ (hence the whole trail looks like `b2s` (`is0` $[.]$)). This context will not be composed with any further contexts in the subsequent steps of reduction. Therefore, the trail at this point is given type $\texttt{int} \rightarrow \langle \bullet \rangle\ \texttt{string}$, consisting of the input type of `is0`, the type of an empty trail, and the output type of `b2s`.

Observe that our trail types can be inhabited by heterogeneous trails, where the input and output types of each invocation context may be different. The flexibility is exactly what we wish a general type system of `control` and `prompt` to have, as we discussed in Section 2.

Comparison with Previous Work. In the CPS translation of Kameyama and Yonezawa [26], a trail is treated as a list of invocation contexts. Such a list is given a recursive type $\mathtt{Trail}(\rho)$ defined as follows:

$$\mathtt{Trail}(\rho) = \mu X.\,\mathtt{list}(\rho \to X \to \rho)$$

We can easily see that the definition restricts the type of invocation contexts in two ways. First, all invocation contexts in a trail must have the same type. This is because lists are homogeneous by definition. Second, each invocation context must have equal input and output types. This is a direct consequence of the first restriction. The two restrictions prevent one from invoking a continuation in a context such as `is0` [.] or `b2s` [.]. Moreover, the use of the list type makes empty and non-empty trails indistinguishable at the level of types, and extension of trails undetectable in types. On the other hand, these limitations allow one to use an ordinary expression type (such as `int`, instead of a type designed specifically for trails) to encode the information of trails in the `control`/`prompt` calculus. That is, if a trail has type $\mathtt{Trail}(\rho)$ in the target, it has type ρ in the source.

5.2 Typing Judgment

We next turn our attention to the typing of a CPS-translated expression. Suppose e is a $\lambda_{\mathcal{F}}$ expression of type τ. In the general case, the CPS counterpart of e is typed in the following way:

$$[\![e^\tau]\!] = \lambda k^{\tau \to \mu_\alpha \to \alpha}.\, \lambda t^{\mu_\beta}.\, e'^{\beta}$$

Here, α and β are answer types, representing the return type of the enclosing `prompt` before and after evaluation of e. It is well-known that delimited control can make the two answer types distinct [14], and since they are needed for deciding the typability of programs, they must be integrated into the typing judgment. The other pair of types, μ_β and μ_α, are trail types, representing the composition of invocation contexts encountered before and after evaluation of e. As `control` can extend a given trail by invoking a captured continuation, the two trail types may be different, and have to be integrated into the typing judgment.

Summing up the above discussion, we conclude that a fully general typing judgment for `control` and `prompt` must carry five types, as follows:

$$\Gamma \vdash e : \tau\, \langle \mu_\alpha \rangle\, \alpha\, \langle \mu_\beta \rangle\, \beta$$

We place the types in the same order as their appearance in the annotated CPS expression. That is, the first three types τ, μ_α, and α correspond to the continuation of e, the next one μ_β represents the trail required by e, and the last one β stands for the eventual value returned by e. We will hereafter call α and β initial and final answer types, and μ_β and μ_α initial and final trail types – be careful of the direction in which answer types and trail types change.

With the typing judgment specified, we can define the syntax of expression types in $\lambda_{\mathcal{F}}$ (Figure 4). Expression types are formed with base types ι (such as `int` and `bool`) and arrow types $\tau_1 \to \tau_2\, \langle \mu_\alpha \rangle\, \alpha\, \langle \mu_\beta \rangle\, \beta$. Notice that the codomain of arrow types carries five components. These types represent the control effect of a function's body, and correspond exactly to the five types that appear in a typing judgment.

Comparison with Previous Work. In the type system developed by Kameyama and Yonezawa [26], a CPS-translated expression is typed in the following way:

$$\lambda k^{\tau \to \mathtt{Trail}(\rho) \to \alpha}.\, \lambda t^{\mathtt{Trail}(\rho)}.\, e'^{\beta}$$

It is obvious that the typing is not as general as ours, since the two trail types are equal. This constraint is imposed by the list representation of trails: since a list type is insensitive to extension, we can always use a trail of the same type for the evaluation of e and the rest of the computation. Thus, Kameyama and Yonezawa arrive at a typing judgment carrying four types, with the last one (ρ) representing the information of trails:

$$\Gamma \vdash e : \tau, \alpha, \beta / \rho$$

Correspondingly, they assign source functions an arrow type of the form $\tau_1 \to \tau_2, \alpha, \beta/\rho$.

5.3 Typing Rules

Now we are ready to define the typing rules of $\lambda_{\mathcal{F}}$ (Figure 4). As in the previous section, we elaborate the typing rules of variables, `prompt`, and `control`.

Variables. Recall that the CPS translation of variables is an η-expanded version of the standard translation. If we annotate the types of each subexpression, a translated variable would look like:

$$\lambda k^{\tau \to \mu_\alpha \to \alpha}.\, \lambda t^{\mu_\alpha}.\, (k\ x\ t)^{\alpha}$$

We see duplicate occurrences of the answer type α and the trail type μ_α. The duplication arises from the application $k\ x\ t$, and reflects the fact that a variable cannot change the answer type or the trail type. By a straightforward conversion from the annotated expression into a typing judgment, we obtain rule (Var) in Figure 4. In general, when the subject of a typing judgment is a pure construct, the answer types and trail types both coincide.

Prompt. We next analyze the CPS translation of `prompt`, again with type annotations.

$$\lambda k^{\tau \to \mu_\alpha \to \alpha}.\, \lambda t^{\mu_\alpha}.\, (k\ (\llbracket e \rrbracket^{(\beta \to \mu_{id} \to \beta') \to \bullet \to \tau}\ k_{id}\ ())\ t)^{\alpha}$$

As $\langle e \rangle$ is a pure expression, we again have equal answer types α and trail types μ_α for the whole expression. The initial trail type $\bullet$ and final answer type τ of e are determined by the application $\llbracket e \rrbracket\ k_{id}\ ()$ and $k\ (\llbracket e \rrbracket\ k_{id}\ ())$, respectively. What is left is to ensure that the application of $\llbracket e \rrbracket$ to the identity continuation k_{id} is type-safe. In our type system, we use a relation $\textsf{is-id-trail}(\tau, \mu, \tau')$ to ensure this type safety. The relation holds when the type $\tau \to \mu \to \tau'$ can be assigned to the identity continuation. The valid combination of τ, μ, and τ' is derived from the definition of the identity continuation, repeated below:

$$\lambda v^{\tau}.\, \lambda t^{\mu}.\, \texttt{case}\ t\ \texttt{of}\ () \Rightarrow v^{\tau'} \mid k \Rightarrow (k\ v\ ())^{\tau'}$$

When t is an empty trail $()$ of type $\bullet$, the return value of k_{id} is v, which has type τ. Since the expected return type of k_{id} is τ', we need the equality $\tau \equiv \tau'$.

When t is a non-empty trail k of type $\tau_1 \to \mu \to {\tau_1}'$, the return value of k_{id} is the result of the application $k\ v\ ()$, which has type ${\tau_1}'$. Since the expected return type of k_{id} is τ', we need the equality $\tau' \equiv {\tau_1}'$. Furthermore, since k must accept v and $()$ as arguments, we need the equalities $\tau \equiv \tau_1$ and $\mu \equiv \bullet$.

We define is-id-trail as an encoding of these constraints, and in the rule (Prompt), we use $\textsf{is-id-trail}(\beta, \mu_{id}, \beta')$ to constrain the type of the continuation of e. Now, it is statically guaranteed that e can be safely evaluated in an empty context.

Syntax of Types

$$\tau, \alpha, \beta ::= \iota \mid \tau \to \tau \, \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\beta \rangle \, \beta \qquad \text{Expression Types}$$
$$\mu, \mu_\alpha, \mu_\beta ::= \bullet \mid \tau \to \langle \mu \rangle \, \tau \qquad \text{Trail Types}$$

Typing Rules

$$\boxed{\Gamma \vdash e : \tau \, \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\beta \rangle \, \beta}$$

$$\frac{c : \iota \in \Sigma}{\Gamma \vdash c : \iota \, \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\alpha \rangle \, \alpha} \; (\textsc{Const}) \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \, \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\alpha \rangle \, \alpha} \; (\textsc{Var})$$

$$\frac{\Gamma, \, x : \tau_1 \vdash e : \tau_2 \, \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\beta \rangle \, \beta}{\Gamma \vdash \lambda x. \, e : (\tau_1 \to \tau_2 \, \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\beta \rangle \, \beta) \, \langle \mu_\gamma \rangle \, \gamma \, \langle \mu_\gamma \rangle \, \gamma} \; (\textsc{Abs})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (\tau_1 \to \tau_2 \, \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\beta \rangle \, \beta) \, \langle \mu_\gamma \rangle \, \gamma \, \langle \mu_\delta \rangle \, \delta \\ \Gamma \vdash e_2 : \tau_1 \, \langle \mu_\beta \rangle \, \beta \, \langle \mu_\gamma \rangle \, \gamma\end{array}}{\Gamma \vdash e_1 \; e_2 : \tau_2 \, \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\delta \rangle \, \delta} \; (\textsc{App})$$

$$\frac{\begin{array}{c}\Gamma, \, k : \tau \to \tau_1 \, \langle \mu_1 \rangle \, {\tau_1}' \, \langle \mu_2 \rangle \, \alpha \vdash e : \gamma \, \langle \mu_{id} \rangle \, \gamma' \, \langle \bullet \rangle \, \beta \\ \textsf{is-id-trail}(\gamma, \mu_{id}, \gamma') \\ \textsf{compatible}((\tau_1 \to \langle \mu_1 \rangle \, {\tau_1}'), \mu_2, \mu_0) \\ \textsf{compatible}(\mu_\beta, \mu_0, \mu_\alpha)\end{array}}{\Gamma \vdash \mathcal{F}k. \, e : \tau \, \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\beta \rangle \, \beta} \; (\textsc{Control}) \qquad \frac{\begin{array}{c}\Gamma \vdash e : \beta \, \langle \mu_{id} \rangle \, \beta' \, \langle \bullet \rangle \, \tau \\ \textsf{is-id-trail}(\beta, \mu_{id}, \beta')\end{array}}{\Gamma \vdash \langle e \rangle : \tau \, \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\alpha \rangle \, \alpha} \; (\textsc{Prompt})$$

Auxiliary Relations

$$\begin{aligned}
\textsf{is-id-trail}(\tau, \bullet, \tau') &= \tau \equiv \tau' \\
&\quad \text{(first branch of } k_{id} \text{ in Figure 3)} \\
\textsf{is-id-trail}(\tau, (\tau_1 \to \langle \mu \rangle \, \tau_1'), \tau') &= (\tau \equiv \tau_1) \wedge (\tau' \equiv \tau_1') \wedge (\mu \equiv \bullet) \\
&\quad \text{(second branch of } k_{id} \text{ in Figure 3)}
\end{aligned}$$

$$\begin{aligned}
\textsf{compatible}(\bullet, \mu_2, \mu_3) &= \mu_2 \equiv \mu_3 \\
&\quad \text{(first branch of @ in Figure 3)} \\
\textsf{compatible}(\mu_1, \bullet, \mu_3) &= \mu_1 \equiv \mu_3 \\
&\quad \text{(first branch of :: in Figure 3)} \\
\textsf{compatible}((\tau_1 \to \langle \mu_1 \rangle \, \tau_1'), \mu_2, \bullet) &= \bot \\
&\quad \text{(no counterpart in Figure 3)} \\
\textsf{compatible}((\tau_1 \to \langle \mu_1 \rangle \, \tau_1'), \mu_2, (\tau_3 \to \langle \mu_3 \rangle \, \tau_3')) &= (\tau_1 \equiv \tau_3) \wedge (\tau_1' \equiv \tau_3') \wedge (\textsf{compatible}(\mu_2, \mu_3, \mu_1)) \\
&\quad \text{(second branch of :: in Figure 3)}
\end{aligned}$$

Figure 4 Type System of $\lambda_{\mathcal{F}}$. We assume a global signature Σ mapping constants to base types.

Control. Lastly, we apply the same method to `control`. Here is the annotated CPS translation:

$$\lambda k^{\tau \to \mu_\alpha \to \alpha}.\, \lambda t^{\mu_\beta}.\, [\![e]\!]^{(\gamma \to \mu_{id} \to \gamma') \to \bullet \to \beta}\, [\lambda x^{\tau}.\, \lambda k'^{\tau_1 \to \mu_1 \to \tau_1'}.\, \lambda t'^{\mu_2}.\, k\ x\ (t \mathbin{@} (k' :: t')^{\mu_0})/c]\ k_{id}\ ()$$

As the body e of `control` is evaluated in a `prompt` clause, we again have an empty initial trail type for e, and we know that the types γ, μ_{id}, and γ' must satisfy the $\mathsf{is\text{-}id\text{-}trail}$ relation. What is left is to ensure that the composition of contexts in $t \mathbin{@} (k' :: t')$ is type-safe. In our type system, we use a relation $\mathsf{compatible}(\mu_1, \mu_2, \mu_3)$ to ensure this type safety. The relation holds when composing a context of type μ_1 and another context of type μ_2 results in a context of type μ_3. Intuitively, the relation can be thought of as an addition over trail types, and the valid combination of μ_1, μ_2, and μ_3 is derived from the definition of the @ and :: functions.

$$\begin{aligned} t^{\mu_1} \mathbin{@} t'^{\mu_2} &= \texttt{case}\ t\ \texttt{of}\ () \Rightarrow t'^{\mu_3} \mid k \Rightarrow (k :: t')^{\mu_3} \\ k^{\tau_1 \to \mu_1 \to \tau_1'} :: t^{\mu_2} &= \texttt{case}\ t\ \texttt{of}\ () \Rightarrow k^{\tau_3 \to \mu_3 \to \tau_3'} \mid k' \Rightarrow (\lambda v.\, \lambda t'.\, k\ v\ (k' :: t'))^{\tau_3 \to \mu_3 \to \tau_3'} \end{aligned}$$

The first clause of @ and that of :: are straightforward: they tell us that the empty trail type $\bullet$ serves as the left and right identity of the addition.

The second clause of :: requires more careful reasoning. The return value of this case is the result of the application $k\ v\ (k' :: t')$, which has type τ_1'. Since the expected return type of :: is τ_3', we need the equality $\tau_1' \equiv \tau_3'$. Moreover, since k must accept v and $k' :: t'$ as arguments, we need the equality $\tau_1 \equiv \tau_3$, as well as a recursive use of $\mathsf{compatible}$, where the third type is μ_1.

The definition of @ and :: further tells us that, when either of their arguments is non-empty, the result of composition cannot be an empty trail. In terms of types, this can be rephrased as: when one of μ_1 and μ_2 is an arrow type, μ_3 cannot be the empty trail type.

We define $\mathsf{compatible}$ as an encoding of these constraints, and in the (Control) rule, we use two instances of this relation to constrain the type of contexts appearing in $t \mathbin{@} (k' :: t')$. Among the two instances, the first one $\mathsf{compatible}((\tau_1 \to \langle \mu_1 \rangle\ \tau_1'), \mu_2, \mu_0)$ states that consing k' to t' is type-safe, and the result has type μ_0. The second one $\mathsf{compatible}(\mu_\beta, \mu_0, \mu_\alpha)$ states that appending t to $k' :: t'$ is type-safe, and the result has type μ_α, which is required by the continuation k of the whole `control` expression.

Comparison with Previous Work. In the type system of Kameyama and Yonezawa [26], the typing rules for `control` and `prompt` are defined as follows:

$$\frac{\Gamma, k : \tau \to \rho, \rho, \alpha/\rho \vdash e : \gamma, \gamma, \beta/\gamma}{\Gamma \vdash \mathcal{F}k.\, e : \tau, \alpha, \beta/\rho}\ (\textsc{Control}) \qquad \frac{\Gamma \vdash e : \rho, \rho, \tau/\rho}{\Gamma \vdash \langle e \rangle : \tau, \alpha, \alpha/\sigma}\ (\textsc{Prompt})$$

The rules are simpler than the corresponding rules in our type system. In particular, there is no equivalent of $\mathsf{is\text{-}id\text{-}trail}$ or $\mathsf{compatible}$, since the homogeneous nature of trails makes those relations trivial. Note that the input and output types shared among invocation contexts come from the body of `prompt`, namely the first occurrence of ρ in the premise of (Prompt).

5.4 Typing Motivating Example

We now show that the motivating example discussed in Section 2 is judged well-typed in $\lambda_{\mathcal{F}}$[5]. The well-typedness of the whole program largely relies on the well-typedness of the two `control` constructs, so let us look at the typing of these constructs:

[5] Our online artifact includes an Agda implementation of this example (`exp4` in `lambdaf.agda`).

$$\vdash \mathcal{F}k_1.\,\texttt{is0}\ (k_1\ 5) : \texttt{int}\,\langle \mu_1 \rangle\,\texttt{string}\,\langle \bullet \rangle\,\texttt{string}$$

$$\vdash \mathcal{F}k_2.\,\texttt{b2s}\ (k_2\ 8) : \texttt{int}\,\langle \mu_2 \rangle\,\texttt{string}\,\langle \mu_1 \rangle\,\texttt{string}$$

For brevity, we write μ_1 to mean $\texttt{int} \to \langle \texttt{bool} \to \langle \bullet \rangle\ \texttt{string} \rangle\ \texttt{string}$, and μ_2 to mean $\texttt{int} \to \langle \bullet \rangle\ \texttt{string}$. We can see how the trail type changes from empty ($\bullet$), to one that refers to a future context (μ_1), and to one that mentions no further context (μ_2). In particular, μ_2 is the result of "adding" μ_1 and the type of `b2s` [.]; that is, the invocation of k_2 *discharges* the future context awaited by `is0` [.]. The trail type μ_2 serves as the final trail type of the body of the enclosing `prompt`, and as it allows us to establish the is-id-trail relation required by (Prompt), we can conclude that the whole program is well-typed.

6 Properties

The type system of $\lambda_{\mathcal{F}}$ enjoys various pleasant properties. First, the type system is sound, that is, well-typed programs do not go wrong [33]. Following Wright and Felleisen [42], we prove type soundness via the preservation and progress theorems.

▶ **Theorem 1** (Preservation). *If* $\Gamma \vdash e : \tau\,\langle \mu_\alpha \rangle\,\alpha\,\langle \mu_\beta \rangle\,\beta$ *and* $e \rightsquigarrow e'$*, then* $\Gamma \vdash e' : \tau\,\langle \mu_\alpha \rangle\,\alpha\,\langle \mu_\beta \rangle\,\beta$*.*

Proof. The proof is by induction on the typing derivation, and is formalized in Agda (the `Reduce` relation in `lambdaf-red.agda`). Note that, to prove type preservation of the `control` reduction (rule ($\mathcal{F}$) in Figure 1), we need to define a set of typing rules for evaluation contexts. ◀

▶ **Theorem 2** (Progress). *If* $\bullet \vdash e : \tau\,\langle \mu_\alpha \rangle\,\alpha\,\langle \mu_\beta \rangle\,\beta$*, then either (i) e is a value, (ii) e takes a step, or (iii) e is a stuck term of the form* $F[\mathcal{F}k.\,e']$*.*

Proof. The proof is by induction on the typing derivation. The third alternative is commonly found in the progress property of effectful calculi [3, 43]. We can remove this alternative by refining our type system to one that can decide the purity of an expression; with this refinement, we can state the usual progress theorem for pure expressions (which include top-level programs). ◀

▶ **Theorem 3** (Type Soundness). *If* $\bullet \vdash \langle e \rangle : \tau\,\langle \mu_\alpha \rangle\,\alpha\,\langle \mu_\alpha \rangle\,\alpha$*, then evaluation of* $\langle e \rangle$ *does not get stuck.*

Proof. The statement is a direct implication of preservation and progress. The need for the top-level `prompt` stems from the fact that a well-typed, closed expression may be a stuck term (corresponding to the third clause of the progress theorem). ◀

Secondly, our CPS translation preserves typing, *i.e.*, it converts a well-typed $\lambda_{\mathcal{F}}$ expression into a well-typed λ_C expression. To establish this theorem, we define the type system of λ_C (Figure 5) and a CPS translation * on $\lambda_{\mathcal{F}}$ types (Figure 6).

Let us elaborate on rule (Case) in Figure 5, which is the only non-trivial typing rule. This rule is used to type the case analysis construct in the three auxiliary functions of the CPS translation, namely k_{id}, @, and ::. Unlike the standard typing rule for case analysis, rule (Case) type-checks the two branches using equality assumptions $\mu \equiv \bullet$ and

Syntax of Types

$$\tau = \iota \mid \tau \to \tau \mid \bullet$$

Typing Rules

$$\frac{c : \iota \in \Sigma}{\Gamma \vdash x : \iota}\ (\textsc{Const}) \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ (\textsc{Var}) \qquad \frac{\Gamma,\, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2}\ (\textsc{Abs})$$

$$\frac{}{\Gamma \vdash () : \bullet}\ (\textsc{Unit}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ (\textsc{App})$$

$$\frac{\begin{array}{c}\Gamma \vdash t : \mu^* \quad \Gamma,\, \mu \equiv \bullet \vdash e_1 : \tau \\ \forall \tau_1, \mu_1, {\tau_1}'.\ \Gamma,\, k : \mu^*,\, \mu \equiv \tau_1 \to \langle \mu_1 \rangle\, {\tau_1}' \vdash e_2 : \tau\end{array}}{\Gamma \vdash \texttt{case}\ t\ \texttt{of}\ () \Rightarrow e_1 \mid k \Rightarrow e_2 : \tau}\ (\textsc{Case})$$

Figure 5 Type System of λ_C. We assume a global signature Σ mapping constants to base types.

$\mu \equiv \tau_1 \to \langle \mu_1 \rangle\, {\tau_1}'$[6]. These assumptions, together with the is-id-trail and compatible relations, allow us to fill in the gap between the expected and actual return types. To see how the assumptions work, consider the typing of k_{id}:

$$\lambda v^{\tau}.\, \lambda t^{\mu}.\, \texttt{case}\ t\ \texttt{of}\ () \Rightarrow v^{\tau'} \mid k \Rightarrow (k\ v\ ())^{\tau'}$$

In the first branch, we see an inconsistency between the expected return type τ' and the actual return type τ. However, by the typing rules defined in Figure 4, we know that k_{id} is used only when the relation is-id-trail(τ, μ, τ') holds, and that if $\mu \equiv \bullet$, we have $\tau \equiv \tau'$. The equality assumption $\mu \equiv \bullet$ made available by rule (Case) allows us to derive $\tau \equiv \tau'$ and conclude that the first branch has the correct type. Similarly, in the second branch, we use the equality assumption $\mu \equiv \tau_1 \to \langle \mu_1 \rangle\, \tau_1'$ to derive $\tau \equiv \tau_1$, $\tau' \equiv \tau_1'$, and $\mu_1 \equiv \bullet$, which imply the well-typedness of the application $k\ v\ ()$. The @ and :: functions can be typed in an analogous way.

▶ **Theorem 4** (Type Preservation of CPS Translation). *If $\Gamma \vdash e : \tau \,\langle \mu_\alpha \rangle\, \alpha \,\langle \mu_\beta \rangle\, \beta$ in $\lambda_{\mathcal{F}}$, then $\Gamma^* \vdash [\![e]\!] : (\tau^* \to \mu_\alpha^* \to \alpha^*) \to \mu_\beta^* \to \beta^*$ in λ_C.*

Proof. The proof is by induction on the typing derivation, and is formalized in Agda (the `cpse` function in `cps.agda`). With the carefully designed rule for case analysis, we can prove the statement in a straightforward manner, as our type system is directly derived from the CPS translation. ◀

Thirdly, and most interestingly, our type system enjoys termination.

[6] The use of equality assumptions in (Case) is inspired by *dependent pattern matching* [12] available in dependently typed languages. Our case analysis is weaker than the dependent variant, in that the return type only depends on the *type* of the scrutinee, not on the scrutinee itself.

Translation of Expression Types

$$\iota^* = \iota$$
$$(\tau_1 \to \tau_2 \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\beta \rangle \, \beta)^* = \tau_1^* \to (\tau_2^* \to \mu_\alpha^* \to \alpha^*) \to \mu_\beta^* \to \beta^*$$

Translation of Trail Types

$$\bullet^* = \bullet$$
$$(\tau \to \langle \mu \rangle \; \tau')^* = \tau^* \to \mu^* \to \tau'^*$$

Figure 6 CPS Translation of $\lambda_{\mathcal{F}}$ Types.

▶ **Theorem 5** (Termination). *If $\Gamma \vdash e : \tau \, \langle \bullet \rangle \, \alpha \, \langle \bullet \rangle \, \alpha$, then there exists some value v such that $e \rightsquigarrow^* v$, where $\rightsquigarrow^*$ is the reflexive, transitive closure of $\rightsquigarrow$ defined in Figure 1.*

Proof. The statement is witnessed by a CPS interpreter of $\lambda_{\mathcal{F}}$ implemented in Agda (the `go` function in `lambdaf.agda`). Since every well-typed Agda program terminates, and since our interpreter is judged well-typed, we know that evaluation of $\lambda_{\mathcal{F}}$ expressions must terminate. ◀

The termination property is unique to our type system. In the existing type system of Kameyama and Yonezawa [26], it is possible to write a well-typed program that does not evaluate to a value, as shown below:

$$\begin{aligned}
&\langle (\mathcal{F}k_1.\, k_1 \; 1; k_1 \; 1); (\mathcal{F}k_2.\, k_2 \; 1; k_2 \; 1) \rangle \\
&= \langle k_1 \; 1; k_1 \; 1 \, [\lambda x.\, x; (\mathcal{F}k_2.\, k_2 \; 1; k_2 \; 1)/k_1] \rangle \\
&= \langle (\mathcal{F}k_2.\, k_2 \; 1; k_2 \; 1); ((\lambda x.\, x; (\mathcal{F}k_2.\, k_2 \; 1; k_2 \; 1)) \; 1) \rangle \\
&= \langle k_2 \; 1; k_2 \; 1 \, [\lambda y.\, y; (\lambda x.\, x; (\mathcal{F}k_2.\, k_2 \; 1; k_2 \; 1)) \; 1/k_2] \rangle \\
&= \langle (\mathcal{F}k_2.\, k_2 \; 1; k_2 \; 1); (\lambda y.\, y; (\lambda x.\, x; (\mathcal{F}k_2.\, k_2 \; 1; k_2 \; 1)) \; 1) \; 1 \rangle \\
&= \ldots
\end{aligned}$$

We see that the two succeeding invocations of captured continuations result in duplication of `control`, leading to a looping behavior.

The well-typedness of the above program in Kameyama and Yonezawa's type system is due to the limited expressiveness of trail types. More precisely, their trail types are mere expression types, which carry no information about the type of contexts to be composed in the future. In our type system, on the other hand, trail types explicitly mention the type of future contexts. This prevents us from duplicating expressions forever, which in turn allows us to statically reject the above looping program.

7 Related Work

Variations of Control Operators. There are four variants of delimited control operators in the style of `control` and `prompt`, differing in whether the control operator keeps the surrounding delimiter, and whether it inserts a delimiter into the captured continuation [16].

Among those variants, `shift` and `reset` [15] are called *static*, as the extent of a captured continuation can always be determined from the lexical structure of the program. Other variants are all *dynamic*, since the control operator may capture the invocation contexts of previously captured continuations (as `control` does), or the meta-contexts outside of the original innermost delimiter (as `shift0` [32] does), or both kinds of contexts (as `control0` [16] does). Dynamic control operators all have a semantics that involves a trail-like structure, containing the contexts beyond the lexically enclosing one.

Type Systems for Control Operators. The CPS-based approach to designing type systems has been applied to several variants of delimited control operators, including `shift/reset` [14, 3], `control/prompt` [26], and `shift0/prompt0` [32]. While Danvy and Filinski [14] consider all expressions as effectful (like we do), subsequent studies distinguish between pure and effectful expressions. This is typically done by not mentioning the answer type (and trail type) of syntactically pure expressions. Having pure expressions makes more programs typable [3, 26, 32], and allows more efficient compilation via a selective CPS translation [37, 32, 4].

Algebraic Effects and Handlers. In the past decade, algebraic effects and handlers [6, 36] have become a popular tool for handling delimited continuations. A prominent feature of effect handlers is that a captured continuation is used at the delimiter site. This makes it unnecessary to keep track of answer types in the type system, as we can decide within a handler whether the use of a continuation is consistent with the actual context. The irrelevance of answer types in turn makes the connection between the type system and CPS translation looser. Indeed, type systems of effect handlers [5, 27] existed before their CPS semantics [29, 24, 23]. Also, type-preserving CPS translation of effect handlers is an open problem in the community [23].

8 Conclusion and Future Work

In this paper, we show how to derive a general type system for the `control` and `prompt` operators. The main idea is to identify all the typing constraints from a CPS translation, where trails are represented as a composition of functions.

The present study is part of a long-term project on formalizing delimited control facilities whose theory is not yet fully developed. In the rest of this section, we describe several directions for future work.

Implementation. Having designed a type system for `control` and `prompt`, a natural next step is to implement a language based on the type system. To make the language practical, we need to address the following challenges. First, we must extend our type system with a form of effect polymorphism or subtyping [26, 32], in order to allow a function or continuation to be called in different contexts. We are currently attempting to adapt Kameyama and Yonezawa's treatment of trail polymorphism to a setting where every typing judgment carries two trail types. Second, we need to design an algorithm for type inference and type checking. We conjecture that answer types can be left implicit in the user program, because it is the case in a calculus featuring `shift` and `reset` [3]. On the other hand, we anticipate that some of the trail types need to be explicitly given by the user, as it does not seem always possible to synthesize the intermediate trail types (μ_0, $\tau_1 \rightarrow \langle\mu_1\rangle\ \tau_1{}'$, and μ_2) in the (Control) rule. Once we have done these, we will develop an implementation (possibly as an extension of OchaCaml [30]) and experiment with various programs from the continuations literature.

Equational Theory. The semantics of `control` and `prompt` is currently given in the form of a CPS translation or an abstract machine [40, 9]. A more direct approach to specifying the semantics of these operators is to establish an *equational theory*, that is, we identify a set of equations that are sound and complete with respect to the existing semantics. Such equations are particularly useful for compilation: for instance, they enable converting an optimization in a CPS compiler into a rewrite in a direct-style (DS) program [38]. We intend to develop an equational theory for `control` and `prompt`, following previous studies on `call/cc` [38], `shift/reset` [25], and `shift0/reset0` [31].

Reflection. An equational theory can be strengthened to a *reflection* [39] by defining a DS translation that serves as a left inverse of the CPS translation. Having a reflection means every reduction in the DS calculus has a corresponding reduction in the CPS calculus, and vice versa. We seek to establish a reflection for `control` and `prompt`, by extending Biernacki et al.'s [11] reflection for `shift` and `reset`.

Control0/Prompt0 and Shallow Effect Handlers. The `control0` and `prompt0` operators are a variation of `control` and `prompt` that remove the matching delimiter upon capturing of a continuation (which is a feature of `shift0` and `reset0`). We plan to formalize a typed calculus of `control0/prompt0`, as well as their equational theory, by combining the insights from our work on `control/prompt` and previous studies on `shift0/reset0` [32, 31]. As shown by Piróg et al. [35], there exists a pair of macro translations [17] between `control0/prompt0` and *shallow effect handlers* [22]. Therefore, an equational theory for `control0/prompt0` could potentially serve as a stepping stone to optimization of shallow handlers, which has not yet been explored [43].

References

1 Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, PPDP '03, pages 8–19, New York, NY, USA, 2003. ACM. doi:10.1145/888251.888254.

2 Kenichi Asai, Youyou Cong, and Chiaki Ishio. A functional abstraction of typed trails. Short paper presented at the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2021), 2021.

3 Kenichi Asai and Yukiyoshi Kameyama. Polymorphic delimited continuations. In *Proceedings of the 5th Asian Conference on Programming Languages and Systems*, APLAS'07, pages 239–254, Berlin, Heidelberg, 2007. Springer-Verlag.

4 Kenichi Asai and Chihiro Uehara. Selective CPS transformation for shift and reset. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '18, pages 40–52, New York, NY, USA, December 2017. ACM. doi:10.1145/3162069.

5 Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. In Reiko Heckel and Stefan Milius, editors, *Algebra and Coalgebra in Computer Science*, pages 1–16, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

6 Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015.

7 Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1, 2005.

8 Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. A dynamic continuation-passing style for dynamic delimited continuations. *BRICS Report Series*, 13(15), 2006.

9 Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. A dynamic continuation-passing style for dynamic delimited continuations. *ACM Trans. Program. Lang. Syst.*, 38(1), 2015. `doi:10.1145/2794078`.

10 Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the static and dynamic extents of delimited continuations. *Science of Computer Programming*, 60(3):274–297, 2006.

11 Dariusz Biernacki, Mateusz Pyzik, and Filip Sieczkowski. A reflection on continuation-composing style. In *Proceedings of 5th International Conference on Formal Structures for Computation and Deduction*, FSCD '20. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020.

12 Thierry Coquand. Pattern matching with dependent types. In *Proceedings of the Third Workshop on Logical Frameworks*, 1992.

13 Olivier Danvy. Type-directed partial evaluation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 242–257. ACM, 1996.

14 Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. BRICS 89/12, 1989.

15 Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160. ACM, 1990.

16 R. Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *J. Funct. Program.*, 17(6):687–730, November 2007. `doi:10.1017/S0956796807006259`.

17 Matthias Felleisen. On the expressive power of programming languages. In *Selected Papers from the Symposium on 3rd European Symposium on Programming*, ESOP '90, pages 35–75, New York, NY, USA, 1991. Elsevier North-Holland, Inc.

18 Mattias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 180–190, New York, NY, USA, 1988. ACM. `doi:10.1145/73560.73576`.

19 Andrzej Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 446–457, New York, NY, USA, 1994. ACM. `doi:10.1145/174675.178047`.

20 Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *Proc. ACM Program. Lang.*, 1(ICFP):13:1–13:29, August 2017. `doi:10.1145/3110257`.

21 Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 12–23, New York, NY, USA, 1995. ACM. `doi:10.1145/224164.224173`.

22 Daniel Hillerström and Sam Lindley. Shallow effect handlers. In *Asian Symposium on Programming Languages and Systems*, APLAS '18, pages 415–435. Springer, 2018.

23 Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect handlers via generalised continuations. *Journal of Functional Programming*, 30, 2020. `doi:10.1017/S0956796820000040`.

24 Daniel Hillerström, Sam Lindley, Robert Atkey, and KC Sivaramakrishnan. Continuation passing style for effect handlers. In *Proceedings of 2nd International Conference on Formal Structures for Computation and Deduction*, FSCD '17, pages 18:1–18:19. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

25 Yukiyoshi Kameyama and Masahito Hasegawa. A sound and complete axiomatization of delimited continuations. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 177–188. ACM, 2003.

26 Yukiyoshi Kameyama and Takuo Yonezawa. Typed dynamic control operators for delimited continuations. In *International Symposium on Functional and Logic Programming*, FLOPS '08, pages 239–254. Springer, 2008.

27 Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 145–158, New York, NY, USA, 2013. ACM. doi:10.1145/2500365.2500590.

28 Oleg Kiselyov and K. C. Sivaramakrishnan. Eff directly in ocaml. In *ML Workshop*, 2016.

29 Daan Leijen. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, pages 486–499, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.3009872.

30 Moe Masuko and Kenichi Asai. Caml Light+ shift/reset= Caml Shift. In *Theory and Practice of Delimited Continuations*, TPDC '11, pages 33–46, 2011.

31 Marek Materzok. Axiomatizing subtyped delimited continuations. In *Computer Science Logic 2013*, CSL 2013. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

32 Marek Materzok and Dariusz Biernacki. Subtyping delimited continuations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 81–93, New York, NY, USA, 2011. ACM. doi:10.1145/2034773.2034786.

33 Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

34 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

35 Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Typed equivalence of effect handlers and delimited control. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

36 Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, ESOP '09, pages 80–94. Springer, 2009.

37 Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 317–328, New York, NY, USA, 2009. ACM. doi:10.1145/1596550.1596596.

38 Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and symbolic computation*, 6(3):289–360, 1993.

39 Amr Sabry and Philip Wadler. A reflection on call-by-value. *ACM transactions on programming languages and systems (TOPLAS)*, 19(6):916–941, 1997.

40 Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 20(4):371–401, 2007.

41 Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M Essertel, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–31, 2019.

42 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

43 Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. Effect handlers, evidently. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–29, 2020. doi:10.1145/3408981.

Beth Semantics and Labelled Deduction for Intuitionistic Sentential Calculus with Identity

Didier Galmiche ✉
Université de Lorraine, CNRS, LORIA, Nancy, France

Marta Gawek ✉
Université de Lorraine, CNRS, LORIA, Nancy, France

Daniel Méry ✉
Université de Lorraine, CNRS, LORIA, Nancy, France

Abstract

In this paper we consider the intuitionistic sentential calculus with Suszko's identity (ISCI). After recalling the basic concepts of the logic and its associated Hilbert proof system, we introduce a new sound and complete class of models for ISCI which can be viewed as algebraic counterparts (and extensions) of sheaf-theoretic topological models of intuitionistic logic. We use this new class of models, called Beth semantics for ISCI, to derive a first labelled sequent calculus and show its adequacy w.r.t. the standard Hilbert axiomatization of ISCI. This labelled proof system, like all other current proof systems for ISCI that we know of, does not enjoy the subformula property, which is problematic for achieving termination. We therefore introduce a second labelled sequent calculus in which the standard rules for identity are replaced with new special rules and show that this second calculus admits cut-elimination. Finally, using a key regularity property of the forcing relation in Beth models, we show that the eigenvariable condition can be dropped, thus leading to the termination and decidability results.

2012 ACM Subject Classification Theory of computation

Keywords and phrases Algebraic Semantics, Beth Models, Labelled Deduction, Intuitionistic Logic

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.13

1 Introduction

In this paper we consider the intuitionistic sentential calculus with identity (ISCI) which extends intuitionistic logic with Suszko's identity operator $\approx$ introduced in [12] for non-Fregean logics, and studied in the context of classical logic in [9] and [1].

Under the usual Fregean interpretation, the question of the equivalence of two formulas reduces to the problem of asking whether or not they have the same logical value. In presence of the non-truth functional identity operator, the rejection of the Fregean axiom makes it possible for two logically equivalent formulas to be considered non-identical in Suszko's sense. The philosophical motivation behind the Sentential Calculus with Identity (SCI) is related to the ontology of situations. In classical logic, only two situations are possible: truth and falsity, and truth (resp. falsity) is described and witnessed by any true (resp. false) proposition. According to [1], this is unfortunate and could be remedied by allowing a new identity connective $\approx$ to describe and witness the fact that two propositions denote the same situation. From this point of view, SCI can be considered as a generalization of classical logic in which we assume that there are more than (and at least) two different situations [7, 9].

In this paper, our aim is to revisit the interpretation of the identity connective on the grounds of intuitionistic logic [3] and to propose a new labelled sequent calculus with good properties like termination from which we can obtain the decidability of the logic. Related works include sequent calculi for both the classical and intuitionistic variants of SCI [2]. Such sequent calculi are obtained following the strategy described in [10] and do not have the

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi; Article No. 13; pp. 13:1–13:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

subformula property. They have been compared with other proof systems for SCI [6, 13] but cannot lead to a decidability procedure for SCI [2]. In the case of the intuitionistic version ISCI, there exists an initial algebraic semantics that combines the ideas of the matrix semantics for sentential calculi with the Kripke semantics of intuitionistic logic. An Hilbert proof system is provided in [9]. A Kripke semantics for ISCI is introduced in [3] along with a sequent calculus for which cut elimination holds. However, since the sequent calculus is not analytic, the cut elimination theorem does not provide a decidability argument.

In Section 2 we introduce ISCI and its standard Hilbert calculus $\mathsf{H}_{\mathsf{ISCI}}$. In Section 3, we propose a new class of models for ISCI, called Beth semantics, which can be viewed as algebraic counterparts (and extensions) of sheaf-theoretic topological models of intuitionistic logic. We first show that general Beth models are complete w.r.t. $\mathsf{H}_{\mathsf{ISCI}}$ (Th. 11). Then, we define the more specific class of regular Beth models and show that they are also complete w.r.t. $\mathsf{H}_{\mathsf{ISCI}}$ (Th. 14). In Section 4, we introduce a first labelled calculus $\mathsf{L}^{1ec}_{\mathsf{ISCI}}$ which is proved complete w.r.t. $\mathsf{H}_{\mathsf{ISCI}}$ (Th. 22) and also w.r.t. Beth models (Th. 23). In Section 5, we derive a second labelled calculus $\mathsf{L}^{2ec}_{\mathsf{ISCI}}$ with new rules for identity and show that $\mathsf{L}^{2ec}_{\mathsf{ISCI}}$ is also complete w.r.t. $\mathsf{H}_{\mathsf{ISCI}}$ (Th. 25) and w.r.t. Beth models (Th. 26), but more interestingly, we show that any $\mathsf{L}^{1ec}_{\mathsf{ISCI}}$-proof can be translated into an $\mathsf{L}^{2ec}_{\mathsf{ISCI}}$-proof (Th. 27). Moreover, we show that cut is admissible in $\mathsf{L}^{2ec}_{\mathsf{ISCI}}$, leading to the cut-free labelled calculus $\mathsf{L}^{2e}_{\mathsf{ISCI}}$ (Th. 33). In Section 6, we derive $\mathsf{L}^{2}_{\mathsf{ISCI}}$, a liberalized variant of $\mathsf{L}^{2e}_{\mathsf{ISCI}}$ in which the eigenvariable condition can be dropped. We show the soundness of $\mathsf{L}^{2}_{\mathsf{ISCI}}$ w.r.t. regular Beth models (Th. 40), which implies the soundness of regular Beth models w.r.t. $\mathsf{H}_{\mathsf{ISCI}}$ and the soundness of all our labelled calculi w.r.t. regular Beth models, as depicted and summarized in the picture below

$$\begin{tikzcd}
\models \mathrm{A} \arrow[r, "\text{Th. 11}"] \arrow[dr] & \vdash_{\mathsf{H}_{\mathsf{ISCI}}} \mathrm{A} \arrow[r, "\text{Th. 22}"] & \vdash_{\mathsf{L}^{1ec}_{\mathsf{ISCI}}} \mathrm{A} \arrow[r, "\text{Th. 27}"] & \vdash_{\mathsf{L}^{2ec}_{\mathsf{ISCI}}} \mathrm{A} \arrow[r, "\text{Th. 33}"] & \vdash_{\mathsf{L}^{2e}_{\mathsf{ISCI}}} \mathrm{A} \arrow[r, "\text{Th. 34}"] & \vdash_{\mathsf{L}^{2}_{\mathsf{ISCI}}} \mathrm{A} \arrow[dllll, "\text{Th. 40}"] \\
& \models_{\mathrm{r}} \mathrm{A} \arrow[u, "\text{Th. 14}"'] & & & &
\end{tikzcd}$$

Finally, we discuss and give arguments for the termination of $\mathsf{L}^{2}_{\mathsf{ISCI}}$, from which we deduce the decidability of ISCI.

2 Intuitionistic Sentential Calculus with Identity

In this section, we recall the basic notions of intuitionistic sentential calculus with Suszko's identity (ISCI) [9, 12]. ISCI extends propositional intuitionistic logic by adding a set of axioms that formalizes the non-truth functional nature of the identity connective $\approx$. The Hilbert-style system for ISCI [3, 9] is introduced and illustrated with examples.

▶ **Definition 1.** *Let* $\mathbf{P} = \{\, \mathrm{p}, \mathrm{q}, \dots \}$ *be a countable set of propositional letters. The formulas of* ISCI*, the set of which is denoted* $\mathbf{F}$*, are given by the grammar:*

$$\mathrm{A} ::= \mathbf{P} \mid \bot \mid \mathrm{A} \wedge \mathrm{A} \mid \mathrm{A} \vee \mathrm{A} \mid \mathrm{A} \supset \mathrm{A} \mid \mathrm{A} \approx \mathrm{A}$$

Formulas of the form $\mathrm{A} \approx \mathrm{B}$ are called *equations*. We write $\mathbf{F}_{/\approx}$ for the restriction of $\mathbf{F}$ to equations. Negation $\neg\mathrm{A}$ and truth $\top$ are respectively defined as $\mathrm{A} \supset \bot$ and $\bot \supset \bot$. To reduce the amount of parentheses, we interpret connectives up to left associativity according to the following strictly decreasing order of precedence: $\neg, \approx, \wedge, \vee, \supset$. Therefore, $\mathrm{A} \wedge \mathrm{B} \wedge \mathrm{A} \vee \mathrm{C} \supset \neg\mathrm{A} \approx \mathrm{B} \supset \mathrm{C}$ means $((((\mathrm{A} \wedge \mathrm{B}) \wedge \mathrm{A}) \vee \mathrm{C}) \supset ((\neg\mathrm{A}) \approx \mathrm{B})) \supset \mathrm{C}$.

ISCI can be axiomatized by adding the four identity axioms described in Figure 1 to any axiom schemata for intuitionistic logic (IL). We call "$\mathsf{H}_{\mathsf{ISCI}}$" the Hilbert proof system consisting of the four axioms for identity, the ten axioms for IL and the rule of modus

$(\approx_1)$	$A \approx A$		
$(\approx_2)$	$(A \approx B) \supset (\neg A \approx \neg B)$		
$(\approx_3)$	$(A \approx B) \supset (B \supset A)$		
$(\approx_4)$	$(A \approx B) \wedge (C \approx D) \supset (A \otimes C) \approx (B \otimes D)$ where $\otimes \in \{ \wedge, \vee, \supset, \approx \}$		
(IL_1)	$A \supset (B \supset A)$	(IL_2)	$(A \supset B) \supset ((A \supset (B \supset C)) \supset (A \supset C))$
(IL_3)	$A \supset (B \supset (A \wedge B))$	(IL_4)	$(A \wedge B) \supset A$
(IL_5)	$(A \wedge B) \supset B$	(IL_6)	$(A \supset C) \supset ((B \supset C) \supset ((A \vee B) \supset C))$
(IL_7)	$A \supset (A \vee B)$	(IL_8)	$B \supset (A \vee B)$
(IL_9)	$(A \supset B) \supset ((A \supset \neg B) \supset \neg A)$	(IL_{10})	$\neg A \supset (A \supset B)$
(MP)	From A and $A \supset B$ deduce B.		

Figure 1 Axioms for ISCI.

(1)	$A \approx B$	assumption
(2)	$B \approx B$	$\approx_1$
(3)	$((B \approx B) \wedge (A \approx B)) \supset ((B \approx A) \approx (B \approx B))$	$\approx_4$
(4)	$(B \approx B) \supset ((A \approx B) \supset ((B \approx B) \wedge (A \approx B)))$	IL_3
(5)	$(A \approx B) \supset ((B \approx B) \wedge (A \approx B))$	MP 2, 4
(6)	$(B \approx B) \wedge (A \approx B)$	MP 1, 5
(7)	$(B \approx A) \approx (B \approx B)$	MP 3, 6
(8)	$((B \approx A) \approx (B \approx B)) \supset ((B \approx B) \supset (B \approx A))$	$\approx_3$
(9)	$(B \approx B) \supset (B \approx A)$	MP 7, 8
(10)	$B \approx A$	MP 2, 9

Figure 2 Proof of $\approx$-symmetry: $A \approx B \vdash_{H_{ISCI}} B \approx A$.

ponens. We write $S \vdash_{H_{ISCI}} B$ to mean that a formula B is derivable in H_{ISCI} from a finite set $S = \{ A_1, \ldots, A_n \}$ of assumptions. Whenever S is empty, B is called a *thesis* or a *theorem* of H_{ISCI} and we write $\vdash_{H_{ISCI}} B$ instead of $\emptyset \vdash_{H_{ISCI}} B$. Let us note that $\emptyset \vdash_{H_{ISCI}} B$ iff $\top \vdash_{H_{ISCI}} B$ and that the deduction theorem holds for H_{ISCI}, i.e. $A_1, \ldots, A_n \vdash_{H_{ISCI}} B$ iff $\vdash_{H_{ISCI}} A_1 \wedge \ldots \wedge A_n \supset B$.

Figure 2 and Figure 3 show that identity is a symmetric and transitive connective.

3 Beth Semantics for ISCI

In this section we propose a new class of models, which we call Beth semantics for ISCI. Let us recall that there already exists an algebraic semantics for ISCI [9]. A Kripke semantics has also been recently investigated in [3]. Kripke-style semantics are usually better suited to the construction of labelled proof systems than algebraic semantics since the forcing relation enables an easy interpretation of a labelled formula $A : x$ as $\rho(x) \Vdash A$, where $\rho(x)$ is the denotation of the label x in some suitable class of models. Kripke models have succeeded in becoming the most popular forcing semantics for intuitionistic logic. One reason for this success is their very natural interpretation of disjunction as $m \Vdash A \vee B$ iff $m \Vdash A$ or $m \Vdash B$, whereas Beth and topological models require more complex notions such as bars and covers.

The models we propose in this section interpret disjunctions in a way which is similar to their interpretation in (sheaf-theoretic) topogical models of intuitionistic logic, but in the more algebraic context of distributive bounded lattices (Heyting algebras). While we pay the price of losing the very natural Kripke interpretation of disjunction, we gain a regularity property that allows us to build a labelled proof system that does not require any eigenvariable conditions, thus opening the way for simpler termination arguments.

▶ **Definition 2.** *Let* $\mathbf{M}$ *be a set of elements, called* worlds, *such that* $\omega, \pi \in \mathbf{M}$ *and* $\omega \neq \pi$.

(1)	$A \approx B$	assumption
(2)	$B \approx C$	assumption
(3)	$(A \approx B) \supset (B \approx A)$	$\approx$ - symmetry
(4)	$B \approx A$	MP 1, 3
(5)	$(B \approx A) \supset ((B \approx C) \supset ((B \approx A) \wedge (B \approx C)))$	IL_3
(6)	$(B \approx C) \supset ((B \approx A) \wedge (B \approx C))$	MP 4, 5
(7)	$(B \approx A) \wedge (B \approx C)$	MP 3, 6
(8)	$((B \approx A) \wedge (B \approx C)) \supset ((B \approx B) \approx (A \approx C))$	$\approx_4$
(9)	$(B \approx B) \approx (A \approx C)$	MP 7, 8
(10)	$(B \approx B) \approx (A \approx C) \supset (A \approx C) \approx (B \approx B)$	$\approx$ - symmetry
(11)	$(A \approx C) \approx (B \approx B)$	MP 9, 10
(12)	$((A \approx C) \approx (B \approx B)) \supset ((B \approx B) \supset (A \approx C))$	$\approx_3$
(13)	$(B \approx B) \supset (A \approx C)$	MP 11, 12
(14)	$B \approx B$	$\approx_1$
(15)	$A \approx C$	MP 13, 14

Figure 3 Proof of $\approx$ - transitivity: $(A \approx B), (B \approx C) \vdash_{H_{ISCI}} A \approx C$.

A Beth frame *is a bounded distributive lattice* $\mathcal{F} = (\mathbf{M}, \leqslant, \sqcup, \omega, \sqcap, \pi)$ *with* ω *and* π *as least and greatest elements respectively.*

▶ **Definition 3.** *A* Beth pre-model *is a triple* $\mathcal{M} = (\mathcal{F}, [\cdot], \Vdash)$*, where* $\mathcal{F}$ *is a Beth frame, and* $[\cdot]$ *is a* valuation function *from* $\mathbf{M}$ *to* $\wp(\mathbf{P} \cup \mathbf{F}_{/\approx})$*, such that for all worlds* m *and* n*:*
$(\mathcal{M}_\pi)$ $[\pi] = \mathbf{P} \cup \mathbf{F}_{/\approx}$,
$(\mathcal{M}_K)$ *if* $m \leqslant n$ *then* $[m] \subseteq [n]$,
$(\mathcal{M}_{\approx_1})$ $A \approx A \in [m]$,
$(\mathcal{M}_{\approx_2})$ *if* $A \approx B \in [m]$ *then* $\neg A \approx \neg B \in [m]$,
$(\mathcal{M}_{\approx_4})$ *for all* $\otimes \in \{\wedge, \vee, \supset, \approx\}$, *if* $A \approx B, C \approx D \in [m]$ *then* $A \otimes C \approx B \otimes D \in [m]$.
The forcing relation $\Vdash$ *is inductively defined as the smallest relation on* $\mathbf{M} \times \mathbf{F}$ *such that:*

- $m \Vdash p$ *iff* $p \in [m]$,
- $m \Vdash A \approx B$ *iff* $A \approx B \in [m]$,
- $m \Vdash \bot$ *iff* $\pi \leqslant m$,
- $m \Vdash A \wedge B$ *iff* $m \Vdash A$ *and* $m \Vdash B$,
- $m \Vdash A \supset B$ *iff for all worlds* n*, if* $n \Vdash A$ *then* $m \sqcup n \Vdash B$,
- $m \Vdash A \vee B$ *iff there exist two worlds* n_1, n_2 *such that* $n_1 \sqcap n_2 \leqslant m$*,* $n_1 \Vdash A$ *and* $n_2 \Vdash B$.

A Beth-model *is a Beth pre-model in which* $\Vdash$ *satisfies the following admissibility condition:*
$(\mathcal{M}_{\approx_3})$ *if* $m \Vdash A \approx B$ *then* $m \Vdash B \supset A$.

As usual, a formula A is *true* (or *satisfied*) in a Beth model $\mathcal{M}$, written $\mathcal{M} \vDash A$, iff $m \Vdash A$ for all worlds m in $\mathcal{M}$ (or equivalently, iff $\omega \Vdash A$) and *valid*, written $\vDash A$, iff it is true in all Beth models. It is routine to show that $\mathcal{M}_\pi$ and $\mathcal{M}_K$ extend from propositional letters and equations to all formulas. $\mathcal{M}_K$ is the well-known Kripke monotonicity condition, which applies to equations in our setting (see [3] for a discussion on alternative choices). Let us remark that $\mathcal{M}_\pi$ implies that all Beth models have a world π that forces all formulas, including inconsistency ($\bot$).

3.1 Completeness of Beth models

A standard way of proving the completeness of a given semantics is to build a canonical model that relates the denotation of formulas to a derivability relation that syntactically defines the logic under consideration (often an Hilbert proof system). Algebraic semantics are usually obtained through Lindenbaum-Tarski constructions that mostly rely on equivalence classes of formulas w.r.t. the underlying derivability relation (for ISCI, we would consider classes such as $\dot{\mathrm{A}} = \{\, \mathrm{B} \mid \mathrm{B} \vdash_{\mathsf{H_{ISCI}}} \mathrm{A}$ and $\mathrm{A} \vdash_{\mathsf{H_{ISCI}}} \mathrm{B} \,\})$. Following an idea of Beth, we replace equivalence classes with theories of formulas to build a canonical model for ISCI in which the forcing relation faithfully mimics the derivability relation in $\mathsf{H_{ISCI}}$.

▶ **Definition 4.** *The* theory $\mathrm{A^t}$ *associated with a formula* A *is the set* $\{\, \mathrm{B} \mid \mathrm{A} \vdash_{\mathsf{H_{ISCI}}} \mathrm{B} \,\}$.

Let $\mathbf{T}$ denote the set $\{\, \mathrm{A^t} \mid \mathrm{A} \in \mathbf{F} \,\}$ of theories generated by all formulas of ISCI. Reading $\mathrm{A} \vdash_{\mathsf{H_{ISCI}}} \mathrm{B}$ as "$\mathrm{A} \leqslant \mathrm{B}$", all sets of formulas can be preordered by derivability in $\mathsf{H_{ISCI}}$. We define $min(X)$ as the set $\{\, \mathrm{A} \in X \mid \forall \mathrm{B} \in X, \mathrm{A} \vdash_{\mathsf{H_{ISCI}}} \mathrm{B} \,\}$ of all formulas that are minimal in X w.r.t. $\vdash_{\mathsf{H_{ISCI}}}$. It follows that for all theories $X \in \mathbf{T}$, $X = \mathrm{A^t}$ for all $\mathrm{A} \in min(X)$. Moreover, for all formulas $\mathrm{A}, \mathrm{B} \in \mathbf{F}$, if $X = \mathrm{A^t} = \mathrm{B^t}$ then both $\mathrm{A} \vdash_{\mathsf{H_{ISCI}}} \mathrm{B}$ and $\mathrm{B} \vdash_{\mathsf{H_{ISCI}}} \mathrm{A}$.

▶ **Definition 5.** *The* canonical Beth frame for ISCI *is the structure* $\mathcal{T} = (\mathbf{T}, \subseteq, \sqcup, \top^{\mathrm{t}}, \sqcap, \perp^{\mathrm{t}})$, *where for all theories* $X, Y \in \mathbf{T}$*:*

$$X \sqcap Y = X \cap Y \textit{ and } X \sqcup Y = \bigcup \{\, (\mathrm{A} \wedge \mathrm{B})^{\mathrm{t}} \mid \mathrm{A} \in min(X), \mathrm{B} \in min(Y) \,\}.$$

▶ **Lemma 6.** *For all theories* $X, Y \in \mathbf{T}$ *and all formulas* $\mathrm{A} \in min(X), \mathrm{B} \in min(Y)$*, the canonical Beth frame for* ISCI *satisfies the following properties:*

$$(a)\ X \sqcap Y = (\mathrm{A} \vee \mathrm{B})^{\mathrm{t}}, \quad (b)\ X \sqcup Y = (\mathrm{A} \wedge \mathrm{B})^{\mathrm{t}}, \quad (c)\ X \subseteq Y \textit{ iff } \mathrm{B} \vdash_{\mathsf{H_{ISCI}}} \mathrm{A}.$$

Proof. Since $\mathrm{A} \in min(X)$ and $\mathrm{B} \in min(Y)$ we have both $X = \mathrm{A^t}$ and $Y = \mathrm{B^t}$.

For (a), by definition $\mathrm{A^t} \sqcap \mathrm{B^t} = \mathrm{A^t} \cap \mathrm{B^t}$. Firstly, we show $\mathrm{A^t} \cap \mathrm{B^t} \subseteq (\mathrm{A} \vee \mathrm{B})^{\mathrm{t}}$. If $\mathrm{C} \in \mathrm{A^t} \cap \mathrm{B^t}$ then $\mathrm{A} \vdash_{\mathsf{H_{ISCI}}} \mathrm{C}$ and $\mathrm{B} \vdash_{\mathsf{H_{ISCI}}} \mathrm{C}$, which implies $\mathrm{A} \vee \mathrm{B} \vdash_{\mathsf{H_{ISCI}}} \mathrm{C}$ (by axiom IL_6). Thus, $\mathrm{C} \in (\mathrm{A} \vee \mathrm{B})^{\mathrm{t}}$. Secondly, we show $(\mathrm{A} \vee \mathrm{B})^{\mathrm{t}} \subseteq \mathrm{A^t} \cap \mathrm{B^t}$. If $\mathrm{C} \in (\mathrm{A} \vee \mathrm{B})^{\mathrm{t}}$, then $\mathrm{A} \vee \mathrm{B} \vdash_{\mathsf{H_{ISCI}}} \mathrm{C}$. Since axioms IL_7 and IL_8 imply $\mathrm{A} \vdash_{\mathsf{H_{ISCI}}} \mathrm{A} \vee \mathrm{B}$ and $\mathrm{B} \vdash_{\mathsf{H_{ISCI}}} \mathrm{A} \vee \mathrm{B}$, we have $\mathrm{A} \vdash_{\mathsf{H_{ISCI}}} \mathrm{C}$ and $\mathrm{B} \vdash_{\mathsf{H_{ISCI}}} \mathrm{C}$. Thus, $\mathrm{C} \in \mathrm{A^t} \cap \mathrm{B^t}$.

For (b), by definition, $(\mathrm{A} \wedge \mathrm{B})^{\mathrm{t}} \subseteq X \sqcup Y$. We show $X \sqcup Y \subseteq (\mathrm{A} \wedge \mathrm{B})^{\mathrm{t}}$. If $\mathrm{C} \in X \sqcup Y$ then $\mathrm{C} \in (\mathrm{F} \wedge \mathrm{G})^{\mathrm{t}}$ for some $\mathrm{F} \in min(X)$ and some $\mathrm{G} \in min(Y)$. Since $X = \mathrm{A^t} = \mathrm{F^t}$ and $Y = \mathrm{B^t} = \mathrm{G^t}$, we have $\mathrm{A} \vdash_{\mathsf{H_{ISCI}}} \mathrm{F}$ and $\mathrm{B} \vdash_{\mathsf{H_{ISCI}}} \mathrm{G}$, which implies $\mathrm{A} \wedge \mathrm{B} \vdash_{\mathsf{H_{ISCI}}} \mathrm{F} \wedge \mathrm{G}$. By definition, $\mathrm{C} \in (\mathrm{F} \wedge \mathrm{G})^{\mathrm{t}}$ implies $\mathrm{F} \wedge \mathrm{G} \vdash_{\mathsf{H_{ISCI}}} \mathrm{C}$. Thus, $\mathrm{A} \wedge \mathrm{B} \vdash_{\mathsf{H_{ISCI}}} \mathrm{C}$ implies $\mathrm{C} \in (\mathrm{A} \wedge \mathrm{B})^{\mathrm{t}}$.

For (c), we show that $\mathrm{B} \vdash_{\mathsf{H_{ISCI}}} \mathrm{A}$ iff $\mathrm{A^t} \subseteq \mathrm{B^t}$. If $\mathrm{B} \vdash_{\mathsf{H_{ISCI}}} \mathrm{A}$ then for all $\mathrm{C} \in \mathrm{A^t}$, we have $\mathrm{A} \vdash_{\mathsf{H_{ISCI}}} \mathrm{C}$, from which it follows that $\mathrm{B} \vdash_{\mathsf{H_{ISCI}}} \mathrm{C}$, i.e. $\mathrm{C} \in \mathrm{B^t}$. Conversely, since $\mathrm{A} \vdash_{\mathsf{H_{ISCI}}} \mathrm{A}$ implies $\mathrm{A} \in \mathrm{A^t}$, if $\mathrm{A^t} \subseteq \mathrm{B^t}$ then $\mathrm{A} \in \mathrm{B^t}$, i.e. $\mathrm{B} \vdash_{\mathsf{H_{ISCI}}} \mathrm{A}$. ◀

Lemma 6 shows that, in the canonical Beth frame $\mathcal{T}$, the partial order defined as set inclusion mimics derivability in $\mathsf{H_{ISCI}}$. Moreover, the lattice meet $\sqcap$ and join $\sqcup$ respectively correspond to disjunction and conjunction in the logic. It then easily follows that $\mathcal{T}$ is a bounded distributive lattice since $\wedge$ and $\vee$ distribute over one another in the logic. Let us note that while the meet of two theories coincides with intersection, their join does not coincide with union since for any two distinct propositional letters p and q, we have $\mathrm{p} \wedge \mathrm{q} \in (\mathrm{p} \wedge \mathrm{q})^{\mathrm{t}}$, but neither $\mathrm{p} \wedge \mathrm{q} \in \mathrm{p^t}$, nor $\mathrm{p} \wedge \mathrm{q} \in \mathrm{q^t}$ (since neither $\mathrm{p} \vdash_{\mathsf{H_{ISCI}}} \mathrm{p} \wedge \mathrm{q}$, nor $\mathrm{q} \vdash_{\mathsf{H_{ISCI}}} \mathrm{p} \wedge \mathrm{q}$).

▶ **Definition 7.** *The* canonical Beth model for ISCI *is the triple* $\mathcal{M}^{\mathrm{t}} = (\mathcal{T}, [\cdot], \Vdash)$*, where the* canonical valuation *is defined as* $[X] = \bigcup \{\, \mathrm{A^t} \mid \mathrm{A} \in min(X) \,\} \cap (\mathbf{P} \cup \mathbf{F}_{/\approx})$ *for all* $X \in \mathbf{T}$.

▶ **Lemma 8.** *The canonical valuation satisfies the conditions of Definition 3 and for all theories* $X \in \mathbf{T}$ *and all formulas* $\mathrm{A} \in min(X)$*,* $[X] = \mathrm{A^t} \cap (\mathbf{P} \cup \mathbf{F}_{/\approx})$.

Proof. $[X] = \mathrm{A}^{\mathrm{t}} \cap (\mathbf{P} \cup \mathbf{F}_{/\approx})$ for all $\mathrm{A} \in min(X)$ follows from the fact that $\mathrm{C}^{\mathrm{t}} = \mathrm{D}^{\mathrm{t}}$ for all $\mathrm{C}, \mathrm{D} \in min(X)$, which implies $\bigcup\{ \mathrm{B}^{\mathrm{t}} \mid \mathrm{B} \in min(X) \} = \mathrm{A}^{\mathrm{t}}$ for all $\mathrm{A} \in min(X)$.

Case $\mathcal{M}_\pi$: By definition, $[\bot^{\mathrm{t}}] = \{ \mathrm{B} \mid \mathrm{B} \in \bot^{\mathrm{t}} \cap (\mathbf{P} \cup \mathbf{F}_{/\approx}) \}$. Since $\bot \vdash_{\mathsf{H_{ISCI}}} \mathrm{B}$ for all formulas B, we have $\bot^{\mathrm{t}} = \mathbf{F}$, which implies $\bot^{\mathrm{t}} \cap (\mathbf{P} \cup \mathbf{F}_{/\approx}) = (\mathbf{P} \cup \mathbf{F}_{/\approx}) = [\bot^{\mathrm{t}}]$.

Case $\mathcal{M}_{\mathrm{K}}$: Suppose we have $X, Y \in \mathbf{T}$ such that $X \subseteq Y$, then $X = \mathrm{A}^{\mathrm{t}}$ and $Y = \mathrm{B}^{\mathrm{t}}$ for some $\mathrm{A} \in min(X)$ and some $\mathrm{B} \in min(Y)$. Since $X \subseteq Y$ implies $\mathrm{A}^{\mathrm{t}} \subseteq \mathrm{B}^{\mathrm{t}}$, if $\mathrm{C} \in [X] = \mathrm{A}^{\mathrm{t}} \cap (\mathbf{P} \cup \mathbf{F}_{/\approx})$, then $\mathrm{C} \in \mathrm{B}^{\mathrm{t}} \cap (\mathbf{P} \cup \mathbf{F}_{/\approx}) = [Y]$. Thus, $[X] \subseteq [Y]$.

The other cases $\mathcal{M}_{\approx i \in \{1,2,4\}}$ easily follow from the $\mathsf{H_{ISCI}}$ axioms $\approx_{i \in \{1,2,4\}}$. ◀

▶ **Lemma 9.** *For all $X \in \mathbf{T}$, for all $\mathrm{A} \in min(X)$, $X \Vdash \mathrm{B}$ iff $\mathrm{A}^{\mathrm{t}} \Vdash \mathrm{B}$ iff $\mathrm{B} \in \mathrm{A}^{\mathrm{t}}$ iff $\mathrm{A} \vdash_{\mathsf{H_{ISCI}}} \mathrm{B}$.*

Proof. By definition of a theory we have $\mathrm{B} \in \mathrm{A}^{\mathrm{t}}$ iff $\mathrm{A} \vdash_{\mathsf{H_{ISCI}}} \mathrm{B}$. Moreover, since $X = \mathrm{A}^{\mathrm{t}}$ for all $\mathrm{A} \in min(X)$, we only need to prove that $\mathrm{A}^{\mathrm{t}} \Vdash \mathrm{B}$ iff $\mathrm{B} \in \mathrm{A}^{\mathrm{t}}$ by structural induction on B.

Base case: $\mathrm{B} \in (\mathbf{P} \cup \mathbf{F}_{/\approx})$. Lemma 8 implies $\mathrm{B} \in [\mathrm{A}^{\mathrm{t}}]$ iff $\mathrm{B} \in \mathrm{A}^{\mathrm{t}}$. Since $\mathrm{A}^{\mathrm{t}} \Vdash \mathrm{B}$ iff $\mathrm{B} \in [\mathrm{A}^{\mathrm{t}}]$ by Definition 3, $\mathrm{A}^{\mathrm{t}} \Vdash \mathrm{B}$ iff $\mathrm{B} \in \mathrm{A}^{\mathrm{t}}$.

Case $\mathrm{B} = \mathrm{B}_1 \vee \mathrm{B}_2$:

$$\begin{array}{rll}
\mathrm{A}^{\mathrm{t}} \Vdash \mathrm{B}_1 \vee \mathrm{B}_2 & \Leftrightarrow\ \exists\, \mathrm{C}_1{}^{\mathrm{t}}, \mathrm{C}_2{}^{\mathrm{t}}.\ \mathrm{C}_1{}^{\mathrm{t}} \sqcap \mathrm{C}_2{}^{\mathrm{t}} \subseteq \mathrm{A}^{\mathrm{t}}, \mathrm{C}_1{}^{\mathrm{t}} \Vdash \mathrm{B}_1, \mathrm{C}_2{}^{\mathrm{t}} \Vdash \mathrm{B}_2 & \\
& \Leftrightarrow\ \exists\, \mathrm{C}_1{}^{\mathrm{t}}, \mathrm{C}_2{}^{\mathrm{t}}.\ (\mathrm{C}_1 \vee \mathrm{C}_2)^{\mathrm{t}} \subseteq \mathrm{A}^{\mathrm{t}}, \mathrm{B}_1 \in \mathrm{C}_1{}^{\mathrm{t}}, \mathrm{B}_2 \in \mathrm{C}_2{}^{\mathrm{t}} & \text{Lem. 6, I.H.} \\
& \Leftrightarrow\ \exists\, \mathrm{C}_1, \mathrm{C}_2.\ \mathrm{A} \vdash_{\mathsf{H_{ISCI}}} \mathrm{C}_1 \vee \mathrm{C}_2, \mathrm{C}_1 \vdash_{\mathsf{H_{ISCI}}} \mathrm{B}_1, \mathrm{C}_2 \vdash_{\mathsf{H_{ISCI}}} \mathrm{B}_2 & \text{Lem. 6, Def. 4} \\
& \Leftrightarrow\ \mathrm{A} \vdash_{\mathsf{H_{ISCI}}} \mathrm{B}_1 \vee \mathrm{B}_2 & \text{Logic} \\
& \Leftrightarrow\ \mathrm{B}_1 \vee \mathrm{B}_2 \in \mathrm{A}^{\mathrm{t}} & \text{Def. 4}
\end{array}$$

Case $\mathrm{B} = \mathrm{B}_1 \supset \mathrm{B}_2$:

$$\begin{array}{rll}
\mathrm{A}^{\mathrm{t}} \Vdash \mathrm{B}_1 \supset \mathrm{B}_2 & \Leftrightarrow\ \forall\, \mathrm{C}^{\mathrm{t}}.\ \text{if } \mathrm{C}^{\mathrm{t}} \Vdash \mathrm{B}_1 \text{ then } \mathrm{A}^{\mathrm{t}} \sqcup \mathrm{C}^{\mathrm{t}} \Vdash \mathrm{B}_2 & \\
& \Leftrightarrow\ \forall\, \mathrm{C}^{\mathrm{t}}.\ \text{if } \mathrm{B}_1 \in \mathrm{C}^{\mathrm{t}} \text{ then } \mathrm{B}_2 \in (\mathrm{A} \wedge \mathrm{C})^{\mathrm{t}} & \text{Lem. 6, I.H.} \\
& \Leftrightarrow\ \forall\, \mathrm{C}.\ \text{if } \mathrm{C} \vdash_{\mathsf{H_{ISCI}}} \mathrm{B}_1 \text{ then } \mathrm{A} \wedge \mathrm{C} \vdash_{\mathsf{H_{ISCI}}} \mathrm{B}_2 & \text{Def. 4} \\
& \Leftrightarrow\ \mathrm{A} \vdash_{\mathsf{H_{ISCI}}} \mathrm{B}_1 \supset \mathrm{B}_2 & \text{Logic} \\
& \Leftrightarrow\ \mathrm{B}_1 \supset \mathrm{B}_2 \in \mathrm{A}^{\mathrm{t}} & \text{Def. 4}
\end{array}$$

The other cases are similar. ◀

▶ **Lemma 10.** *The canonical Beth model $\mathcal{M}^{\mathrm{t}}$ satisfies the admissibility condition $\mathcal{M}_{\approx_3}$.*

Proof. Any $X \in \mathbf{T}$ such that $X \Vdash \mathrm{A} \approx \mathrm{B}$ entails $\mathrm{C} \vdash_{\mathsf{H_{ISCI}}} \mathrm{A} \approx \mathrm{B}$ for all $\mathrm{C} \in min(X)$ by Lemma 9, which implies $\mathrm{C} \vdash_{\mathsf{H_{ISCI}}} \mathrm{B} \supset \mathrm{A}$ by axiom $(\approx_3)$. Thus, $X \Vdash \mathrm{B} \supset \mathrm{A}$ by Lemma 9. ◀

▶ **Theorem 11.** *Beth models for* ISCI *are complete, i.e., if $\vDash \mathrm{A}$ then $\vdash_{\mathsf{H_{ISCI}}} \mathrm{A}$.*

Proof. We show that $\nvdash_{\mathsf{H_{ISCI}}} \mathrm{A}$ implies $\nvDash \mathrm{A}$. Suppose that $\nvdash_{\mathsf{H_{ISCI}}} \mathrm{A}$, then $\top \nvdash_{\mathsf{H_{ISCI}}} \mathrm{A}$ which implies $\mathrm{A} \notin \top^{\mathrm{t}}$. By Lemma 9 we get $\top^{\mathrm{t}} \nVdash \mathrm{A}$ in $\mathcal{M}^{\mathrm{t}}$, which by definition implies $\nvDash \mathrm{A}$. ◀

3.2 Regular Beth Models

We now show that the canonical Beth model for ISCI satisfies a regularity property that is essential for the termination arguments in Section 6.3.

▶ **Definition 12.** *Let $\mathcal{M} = (\mathcal{F}, [\cdot], \Vdash)$ be a Beth model. $\mathcal{M}$ is* regular *iff for all formulas A, if $m \Vdash \mathrm{A}$ for some world m, then there exists a world m_{A}, called* A-minimal*, such that $m_{\mathrm{A}} \Vdash \mathrm{A}$ and for all worlds n, $n \Vdash \mathrm{A}$ implies $m_{\mathrm{A}} \leqslant n$. We write $\vDash_{\mathrm{r}}$ (instead of $\vDash$) for the restriction of validity to the class of* regular *Beth models.*

▶ **Lemma 13.** *The canonical model $\mathcal{M}^{\mathrm{t}}$ is regular: for all formulas A, A^{t} is A-minimal.*

Proof. Suppose that $\mathrm{B}^{\mathrm{t}} \Vdash \mathrm{A}$ for an arbitrary theory B^{t}. Then, $\mathrm{B} \vdash_{\mathsf{H_{ISCI}}} \mathrm{A}$ by Lemma 9, which implies $\mathrm{A}^{\mathrm{t}} \subseteq \mathrm{B}^{\mathrm{t}}$ by Lemma 6. ◀

▶ **Theorem 14.** *Regular Beth models for* ISCI *are complete: if* $\vDash_{\mathrm{r}} \mathrm{A}$ *then* $\vdash_{\mathsf{H}_{\mathsf{ISCI}}} \mathrm{A}$.

Proof. The result is an immediate consequence of Lemma 13. ◀

▶ **Theorem 15.** *Regular Beth models for* ISCI *are sound: if* $\vdash_{\mathsf{H}_{\mathsf{ISCI}}} \mathrm{A}$ *then* $\vDash_{\mathrm{r}} \mathrm{A}$.

Proof. The result follows from Theorems 22, 27, 33, 34 and 40. ◀

Let us remark that non-regular Beth models are neither sound for ISCI, nor for IL. Indeed, $\mathrm{p} \vee \mathrm{p} \supset \mathrm{p}$ is a theorem of IL, but $\omega \nVdash \mathrm{p} \vee \mathrm{p} \supset \mathrm{p}$ in the Beth model $((\mathbf{M}, \leqslant, \sqcup, \omega, \sqcap, \pi), [\cdot], \Vdash)$, where $\mathbf{M} = \{\omega, m_1, m_2, \pi\}$, $m \leqslant n$ iff $m = \omega$ or $n = \pi$, $[\omega] = \{\mathrm{A} \approx \mathrm{A} \mid \mathrm{A} \in \mathbf{F}\}$, $[m_1] = [m_2] = [\omega] \cup \{\mathrm{p}\}$, and $[\pi] = \mathbf{P} \cup \mathbf{F}_{/\approx}$.

▶ **Theorem 16.** *In a regular Beth model $\mathcal{M}$, if $m \Vdash \mathrm{A}$ and $n \Vdash \mathrm{A}$ then $m \sqcap n \Vdash \mathrm{A}$.*

Proof. Since $\mathcal{M}$ is regular, $m \Vdash \mathrm{A}$ and $n \Vdash \mathrm{A}$ imply the existence of an A-minimal world m_{A}. Since $m_{\mathrm{A}} \leqslant m$ and $m_{\mathrm{A}} \leqslant n$ imply $m_{\mathrm{A}} \leqslant m \sqcap n$, $m \sqcap n \Vdash \mathrm{A}$ by Kripke monotonicity. ◀

4 Labelled Deduction for ISCI

In this section we propose a new labelled sequent calculus, called $\mathsf{L}^{\mathsf{1ec}}_{\mathsf{ISCI}}$, which is derived from the Beth models described in Section 3. The methodology is inspired by and in the spirit of our works on labelled deduction in BI and bi-intuitionistic logic [4, 5]. Let us note that there exists a label-free sequent calculus for ISCI [3], built following the strategy described in [10, 11], which like $\mathsf{L}^{\mathsf{1ec}}_{\mathsf{ISCI}}$ does not enjoy the subformula property.

4.1 A Labelling Algebra

Let $\mathbf{L}^{\mathrm{n}}$ be the set $\{\mathrm{S} \mid \mathrm{S} \subset \mathbb{N} \text{ and } |\mathrm{S}| = \mathrm{n}\}$ of all subsets of $\mathbb{N}$ of size (cardinal) n. The set $\mathbf{L}^*$ of *label letters* is defined as $\bigcup_{\mathrm{n} \in \mathbb{N}} \mathbf{L}^{\mathrm{n}}$. Let $\mathbf{L}^{\mathrm{u}} = \{\emptyset, \mathbb{N}\}$ be the set of *label units*, the set $\mathbf{L}$ of *labels* is then defined as $\mathbf{L}^* \cup \mathbf{L}^{\mathrm{u}}$. We use the (possibly subscripted or primed) letters $\mathrm{a}, \mathrm{b}, \mathrm{c}$ to denote labels which are singletons (i.e., elements of $\mathbf{L}^1$) and save the letters $\mathrm{x}, \mathrm{y}, \mathrm{z}$ to denote arbitrary labels. A label x is a *sublabel* of a label y if $\mathrm{x} \subseteq \mathrm{y}$.

We work with a labelling algebra $\mathcal{L}$ defined as the lattice $(\mathbf{L}, \subseteq, \cup, \emptyset, \cap, \mathbb{N})$, where join $\cup$ and meet $\cap$ are standard set union and intersection. We consider that $\cup$ binds stronger than $\cap$ and we shall frequently write xy instead of $\mathrm{x} \cup \mathrm{y}$ ($\mathrm{xx}' \cap \mathrm{yy}'$ should therefore be read as $(\mathrm{x} \cup \mathrm{x}') \cap (\mathrm{y} \cup \mathrm{y}')$). In this paper, we shall only use examples with label letters built from the subset $\{1, \ldots, 9\}$. Therefore, we shall use the more concise notation 13 to unambiguously refer to $\{1, 3\}$ (and not to the label letter $\{13\}$).

4.2 The Labelled Sequent Calculus $\mathsf{L}^{\mathsf{1ec}}_{\mathsf{ISCI}}$

▶ **Definition 17.** *A* labelled formula *is a pair* (C, x)*, written* $\mathrm{C} : \mathrm{x}$*, where* C *is a formula and* x *is a label. A* labelled sequent *is a pair* (Γ, Δ)*, written* $\Gamma \vdash \Delta$*, where* Γ, Δ *are multi-sets of labelled formulas.*

We use the generic notation $O(T)$ to mean that the object T is a subobject of an object O (for some well defined notion of object inclusion). For example, when S is a set, $\mathrm{S}(e_1, \ldots, e_n)$ means that $\{e_1, \ldots, e_n\}$ is a subset of S. Similarly, if F and G are formulas, F(G) means that G is a subformula of F and if x is a label, x(y) means that y is a sublabel of x. If Δ is a set or multi-set of labelled formulas, we define $\mathrm{x} \subseteq \Delta$ as $\exists \mathrm{A} : \mathrm{y} \in \Delta$ such that $\mathrm{x} \subseteq \mathrm{y}$, which is more shortly written $\Delta(\mathrm{x})$. The notation $\mathrm{x} \subseteq \mathrm{A} : \mathrm{y}$ is a shorthand for $\mathrm{x} \subseteq \{\mathrm{A} : \mathrm{y}\}$. A labelled sequent $\Gamma \vdash \Delta$ is *connected* iff $\mathrm{x} \subseteq \Delta$ for all $\mathrm{A} : \mathrm{x} \in \Gamma$.

$$\dfrac{}{\Gamma(A:x)\vdash\Delta(A:y)}\,\mathrm{id}(x\subseteq y) \qquad \dfrac{}{\Gamma(\bot:x)\vdash\Delta(A:y)}\,\bot_L(x\subseteq y)$$

$$\dfrac{\Gamma,A:x,B:x\vdash\Delta}{\Gamma(A\wedge B:x)\vdash\Delta}\,\wedge_L \qquad \dfrac{\Gamma\vdash\Delta,A:x \quad \Gamma\vdash\Delta,B:x}{\Gamma\vdash\Delta(A\wedge B:x)}\,\wedge_R$$

$$\dfrac{\Gamma\vdash\Delta,A:y \quad \Gamma,B:x\cup y\vdash\Delta}{\Gamma(A\supset B:x)\vdash\Delta}\,\supset_L(x\cup y\subseteq\Delta) \qquad \dfrac{\Gamma,A:a\vdash\Delta,B:x\cup a}{\Gamma\vdash\Delta(A\supset B:x)}\,\supset_R(a\not\subseteq\Gamma\cup\Delta)$$

$$\dfrac{\Gamma,A:x\cup a\vdash\Delta,C:y\cup a \quad \Gamma,B:x\cup b\vdash\Delta,C:y\cup b}{\Gamma(A\vee B:x)\vdash\Delta(C:y)}\,\vee_L(a\neq b\not\subseteq\Gamma\cup\Delta,x\subseteq y)$$

$$\dfrac{\Gamma\vdash\Delta,A:x,B:x}{\Gamma\vdash\Delta(A\vee B:x)}\,\vee_R \qquad \dfrac{\Gamma\vdash\Delta,C:x \quad C:x,\Gamma\vdash\Delta}{\Gamma\vdash\Delta}\,\mathrm{cut}(x\subseteq\Delta)$$

$$\dfrac{\Gamma,A\approx A:x\vdash\Delta}{\Gamma\vdash\Delta}\,{\approx_{L1}}(x\subseteq\Delta) \qquad \dfrac{\Gamma,\neg A\approx\neg B:x\vdash\Delta}{\Gamma(A\approx B:x)\vdash\Delta}\,{\approx_{L2}} \qquad \dfrac{\Gamma,B\supset A:x\vdash\Delta}{\Gamma(A\approx B:x)\vdash\Delta}\,{\approx_{L3}}$$

$$\dfrac{\Gamma,A\otimes C\approx B\otimes D:x\vdash\Delta}{\Gamma(A\approx B:x,C\approx D:x)\vdash\Delta}\,{\approx_{L4}} \qquad \dfrac{\Gamma,A\otimes A\approx B\otimes B:x\vdash\Delta}{\Gamma(A\approx B:x)\vdash\Delta}\,{\approx_{L4'}}$$

Figure 4 Labelled Sequent Calculus $\mathsf{L}^{\mathsf{1ec}}_{\mathsf{ISCI}}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{B\approx B:\emptyset\vdash B\approx B:\emptyset}\,\mathrm{id}}{\vdash B\approx B:\emptyset}\,{\approx_{L1}} \quad \dfrac{}{B\approx A:1\vdash B\approx A:1}\,\mathrm{id}}{(B\approx B)\supset(B\approx A):1\vdash B\approx A:1}\,\supset_L}{(B\approx A)\approx(B\approx B):1\vdash B\approx A:1}\,{\approx_{L3}}}{B\approx B:1,A\approx B:1\vdash B\approx A:1}\,{\approx_{L4}}}{A\approx B:1\vdash B\approx A:1}\,{\approx_{L1}}}{\vdash(A\approx B)\supset(B\approx A):\emptyset}\,\supset_R$$

Figure 5 $\mathsf{L}^{\mathsf{1ec}}_{\mathsf{ISCI}}$-Proof of $\approx$-symmetry.

The labelled calculus $\mathsf{L}^{\mathsf{1ec}}_{\mathsf{ISCI}}$ is given in Figure 4. The only structural rule in $\mathsf{L}^{\mathsf{1ec}}_{\mathsf{ISCI}}$ is cut. All lattice properties of Beth models are implicity reflected in our labelling algebra by our choice of labels as subsets of $\mathbb{N}$. The rules $\supset_R$ and $\vee_L$ have *eigenvariable* (or *freshness*) conditions on the label letters a, b they introduce. Since connectedness plays a significant role in our forthcoming proof of cut elimination, the rules of $\mathsf{L}^{\mathsf{1ec}}_{\mathsf{ISCI}}$ have been carefully designed so as to preserve this property from their conclusion to their premise. For instance, the cut rule has a side condition that requires the label of the cut formula to occur as a sublabel on the right-hand side of the conclusion.

▶ **Definition 18.** *A formula* A *is a* theorem of *(or* derivable in*)* $\mathsf{L}^{\mathsf{1ec}}_{\mathsf{ISCI}}$*, written* $\vdash_{\mathsf{L}^{\mathsf{1ec}}_{\mathsf{ISCI}}} A$*, if the labelled sequent* $\vdash A:\emptyset$ *is derivable from the rules given in Figure 4.*

The proof rules in Figure 4 are formulated in a non-destructive way, i.e. they preserve (a copy of) their principal formulas in their premise. This is only a technical choice that makes the proofs of the forthcoming admissibility results shorter, but we shall use the more standard destructive versions of the rules in our examples to keep them more concise.

Figure 5 gives a labelled proof in $\mathsf{L}^{\mathsf{1ec}}_{\mathsf{ISCI}}$ of the symmetry of the identity connective, from which one can easily derive a symmetry rule $\approx_{LS}$, as illustrated in Figure 6, which proves the transitivity of $\approx$. More examples are given in the proof of Lemma 20.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{B \approx B : \emptyset \vdash B \approx B : \emptyset}\,\mathrm{id}}{\vdash B \approx B : \emptyset}\,{\approx_{L1}} \quad \dfrac{}{A \approx C : 1 \vdash A \approx C : 1}\,\mathrm{id}}{(B \approx B) \supset (A \approx C) : 1 \vdash A \approx C : 1}\,{\supset_{L}}}{(A \approx C) \approx (B \approx B) : 1 \vdash A \approx C : 1}\,{\approx_{L3}}}{(B \approx B) \approx (A \approx C) : 1 \vdash A \approx C : 1}\,{\approx_{LS}}}{B \approx A : 1, B \approx C : 1 \vdash A \approx C : 1}\,{\approx_{L4}}}{A \approx B : 1, B \approx C : 1 \vdash A \approx C : 1}\,{\approx_{LS}}}{(A \approx B) \wedge (B \approx C) : 1 \vdash A \approx C : 1}\,{\wedge_{L}}}{\vdash (A \approx B) \wedge (B \approx C) \supset (A \approx C) : \emptyset}\,{\supset_{R}}$$

Figure 6 $\mathsf{L}^{1ec}_{\mathsf{ISCI}}$-Proof of $\approx$-Transitivity.

4.3 Soundness and Completeness of $\mathsf{L}^{1ec}_{\mathsf{ISCI}}$

▶ **Theorem 19** (Soundness). *If* $\vdash_{\mathsf{L}^{1ec}_{\mathsf{ISCI}}} A$ *then* $\vdash_{\mathsf{H}_{\mathsf{ISCI}}} A$.

Proof. A corollary of Theorems 27, 33, 34 and 40. ◀

▶ **Lemma 20.** *All of the axioms for* $\approx$ *given in Figure 1 are derivable in* $\mathsf{L}^{1ec}_{\mathsf{ISCI}}$.

Proof. Axiom $\approx_1$:

$$\dfrac{\dfrac{}{A \approx A : \emptyset \vdash A \approx A : \emptyset}\,\mathrm{id}}{\vdash A \approx A : \emptyset}\,{\approx_{L1}}$$

Axioms $\approx_2$, $\approx_3$:

$$\dfrac{\dfrac{\dfrac{}{\neg A \approx \neg B : 1 \vdash \neg A \approx \neg B : 1}\,\mathrm{id}}{A \approx B : 1 \vdash \neg A \approx \neg B : 1}\,{\approx_{L2}}}{\vdash (A \approx B) \supset (\neg A \approx \neg B) : \emptyset}\,{\supset_{R}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{B : 2 \vdash B : 2}\,\mathrm{id} \quad \dfrac{}{A : 12 \vdash A : 12}\,\mathrm{id}}{B \supset A : 1, B : 2 \vdash A : 12}\,{\supset_{L}}}{A \approx B : 1, B : 2 \vdash A : 12}\,{\approx_{L3}}}{A \approx B : 1 \vdash B \supset A : 1}\,{\supset_{R}}}{\vdash (A \approx B) \supset (B \supset A) : \emptyset}\,{\supset_{R}}$$

Axiom $\approx_4$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(A \otimes C) \approx (B \otimes D) : 1 \vdash (A \otimes C) \approx (B \otimes D) : 1}\,\mathrm{id}}{A \approx B : 1, C \approx D : 1 \vdash (A \otimes C) \approx (B \otimes D) : 1}\,{\approx_{L4}}}{(A \approx B) \wedge (C \approx D) : 1 \vdash (A \otimes C) \approx (B \otimes D) : 1}\,{\wedge_{L}}}{\vdash (A \approx B) \wedge (C \approx D) \supset (A \otimes C) \approx (B \otimes D) : \emptyset}\,{\supset_{R}}$$

◀

▶ **Lemma 21.** *All of the axioms for* IL *given in Figure 1 are derivable in* $\mathsf{L}^{1ec}_{\mathsf{ISCI}}$.

Proof. Axiom IL_6:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A : 34 \vdash A : 234}\,\mathrm{id} \quad \dfrac{}{C : 1234 \vdash C : 1234}\,\mathrm{id}}{A \supset C : 1, A : 34 \vdash C : 1234}\,{\supset_{L}} \quad \dfrac{\dfrac{}{B : 35 \vdash B : 135}\,\mathrm{id} \quad \dfrac{}{C : 1235 \vdash C : 1235}\,\mathrm{id}}{B \supset C : 2, B : 35 \vdash C : 1235}\,{\supset_{L}}}{A \supset C : 1, B \supset C : 2, A \vee B : 3 \vdash C : 123}\,{\vee_{L}}}{A \supset C : 1, B \supset C : 2 \vdash (A \vee B) \supset C : 12}\,{\supset_{R}}}{A \supset C : 1 \vdash (B \supset C) \supset ((A \vee B) \supset C) : 1}\,{\supset_{R}}}{\vdash (A \supset C) \supset ((B \supset C) \supset ((A \vee B) \supset C)) : \emptyset}\,{\supset_{R}}$$

Axioms IL_7, IL_8, IL_{10}:

$$\dfrac{\dfrac{\dfrac{}{A : 1 \vdash A : 1, B : 1}\,\mathrm{id}}{A : 1 \vdash A \vee B : 1}\,{\vee_{R}}}{\vdash A \supset (A \vee B) : \emptyset}\,{\supset_{R}} \qquad \dfrac{\dfrac{\dfrac{}{B : 1 \vdash A : 1, B : 1}\,\mathrm{id}}{B : 1 \vdash A \vee B : 1}\,{\vee_{R}}}{\vdash B \supset (A \vee B) : \emptyset}\,{\supset_{R}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A : 2 \vdash A : 2}\,\mathrm{id} \quad \dfrac{}{\bot : 12 \vdash B : 12}\,{\bot_{L}}}{\neg A : 1, A : 2 \vdash B : 12}\,{\supset_{L}}}{\neg A : 1 \vdash A \supset B : 1}\,{\supset_{R}}}{\vdash \neg A \supset (A \supset B) : \emptyset}\,{\supset_{R}}$$

$$\frac{\Gamma \vdash \Delta, C\sigma : y}{\Gamma(A \approx B : x) \vdash \Delta(C : y)} \approx_{LR} (x \subseteq y, A \neq B) \qquad \frac{}{\Gamma \vdash \Delta(A \approx A : x)} \approx_R$$

$$\sigma = [B \mapsto A] \text{ if } |A| \leqslant |B| \text{ and } [A \mapsto B] \text{ otherwise.}$$

Figure 7 L^{2ec}_{ISCI} "Special" Identity Rules.

Rule MP: We use admissibility of weakening, which is stated and proved in the paper for L^{2ec}_{ISCI} in Lemma 29, but which also holds for L^{1ec}_{ISCI} with a similar proof.

$$\dfrac{\dfrac{\vdash A : \emptyset}{\vdash B : \emptyset, A : \emptyset} \quad \dfrac{\dfrac{\vdash A \supset B : \emptyset}{A : \emptyset \vdash B : \emptyset, A \supset B : \emptyset} \quad \dfrac{\dfrac{}{A : \emptyset \vdash A : \emptyset}\,\text{id} \quad \dfrac{}{B : \emptyset \vdash B : \emptyset}\,\text{id}}{A \supset B : \emptyset, A : \emptyset \vdash B : \emptyset}\,\supset_L}{A : \emptyset \vdash B : \emptyset}\,\text{cut}}{\vdash B : \emptyset}\,\text{cut}$$

The other cases are similar. ◀

▶ **Theorem 22** (H$_{ISCI}$ completeness). *If* $\vdash_{\mathsf{H}_{ISCI}} A$ *then* $\vdash_{\mathsf{L}^{1ec}_{ISCI}} A$.

Proof. A direct consequence of Lemma 20 and Lemma 21. ◀

▶ **Theorem 23** (Beth completeness). *If* $\vDash A$ *then* $\vdash_{\mathsf{L}^{1ec}_{ISCI}} A$.

Proof. If $\vDash A$ then Theorem 11 yields $\vdash_{\mathsf{H}_{ISCI}} A$, which by Theorem 22 implies $\vdash_{\mathsf{L}^{1ec}_{ISCI}} A$. ◀

5 The Labelled Calculus L^{2ec}_{ISCI}

L^{1ec}_{ISCI} is not very interesting from the point of view of termination as it lacks the subformula property. Indeed, even if we eliminate the cut rule from L^{1ec}_{ISCI}, we can still introduce infinitely many subformulas using the identity rule $\approx_{L1}$. Moreover, defining the size $|A|$ of a formula A as the number of its connectives, it is easy to see that the identity rules $\approx_{L4}$ and $\approx_{L4'}$ introduce in their single premiss an active formula the size of which is greater than the size of the principal formula in their conclusion.

As a first step toward termination we define L^{2ec}_{ISCI} as the variant of L^{1ec}_{ISCI} in which all of the identity rules of Figure 4 are replaced with the identity rules of Figure 7. Depending on the size of A and B, the rule $\approx_{LR}$ simultaneously replaces all occurrences of the formula B in C with the formula A whenever $|A| \leqslant |B|$ and A is not syntactically equal to B.

5.1 Soundness and Completeness

▶ **Theorem 24** (Soundness). *If* $\vdash_{\mathsf{L}^{2ec}_{ISCI}} A$ *then* $\vdash_{\mathsf{H}_{ISCI}} A$.

Proof. A corollary of Theorems 33, 34 and 40. ◀

▶ **Theorem 25** (H$_{ISCI}$ completeness). *If* $\vdash_{\mathsf{H}_{ISCI}} A$ *then* $\vdash_{\mathsf{L}^{2ec}_{ISCI}} A$.

Proof. Similar to the proof of Theorem 22. ◀

▶ **Theorem 26** (Beth completeness). *If* $\vDash A$ *then* $\vdash_{\mathsf{L}^{2ec}_{ISCI}} A$.

Proof. If $\vDash A$ then Theorem 11 yields $\vdash_{\mathsf{H}_{ISCI}} A$, which by Theorem 25 implies $\vdash_{\mathsf{L}^{2ec}_{ISCI}} A$. ◀

▶ **Theorem 27** ($\mathsf{L}^{1ec}_{\mathsf{ISCI}}$ to $\mathsf{L}^{2ec}_{\mathsf{ISCI}}$). *If Π is an $\mathsf{L}^{1ec}_{\mathsf{ISCI}}$ proof of A, then there exists a translation $t(\Pi)$ of Π which is an $\mathsf{L}^{2ec}_{\mathsf{ISCI}}$ proof of A.*

Proof. The proof is by induction on the height of $\mathsf{L}^{1ec}_{\mathsf{ISCI}}$ proofs. Since $\mathsf{L}^{2ec}_{\mathsf{ISCI}}$ only differs from $\mathsf{L}^{1ec}_{\mathsf{ISCI}}$ on the identity rules, the base cases for axioms are immediate and we only need to show that $\mathsf{L}^{2ec}_{\mathsf{ISCI}}$ can simulate $\mathsf{L}^{1ec}_{\mathsf{ISCI}}$ identity rules. We assume without loss of generality that $|A| \leqslant |B|$ and $|C| \leqslant |D|$. Moreover, in the translated proofs below, the occurrences of $\approx_{LR}$ only actually exist when the formulas on both sides of the principal identity connective are not syntactically equal.

Case $\approx_{L1}$:

$$\dfrac{\dfrac{\Pi_1}{\Gamma, A \approx A : x \vdash \Delta}}{\Gamma \vdash \Delta}\approx_{L1} \quad\rightsquigarrow\quad \dfrac{\dfrac{}{\Gamma \vdash \Delta, A \approx A : x}\approx_R \quad \dfrac{t(\Pi_1) \text{ from I.H.}}{\Gamma, A \approx A : x \vdash \Delta}}{\Gamma \vdash \Delta}\text{cut}$$

Case $\approx_{L2}$:

$$\dfrac{\dfrac{\Pi_1}{\Gamma, \neg A \approx \neg B : x \vdash \Delta}}{\Gamma(A \approx B : x) \vdash \Delta}\approx_{L2}$$

$$\wr\!\wr$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma(A \approx B : x) \vdash \Delta, \neg A \approx \neg A : x}\approx_R}{\Gamma(A \approx B : x) \vdash \Delta, \neg A \approx \neg B : x}\approx_{LR}(A \neq B) \quad \dfrac{t(\Pi_1) \text{ from I.H.}}{\Gamma, \neg A \approx \neg B : x \vdash \Delta}}{\Gamma(A \approx B : x) \vdash \Delta}\text{cut}$$

Case $\approx_{L3}$:

$$\dfrac{\dfrac{\Pi_1}{\Gamma, B \supset A : x \vdash \Delta}}{\Gamma(A \approx B : x) \vdash \Delta}\approx_{L3} \quad\rightsquigarrow\quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma(A \approx B : x), A : a \vdash \Delta, A : xa}\text{id}}{\Gamma(A \approx B : x) \vdash \Delta, A \supset A : x}\supset_R}{\Gamma(A \approx B : x) \vdash \Delta, B \supset A : x}\approx_{LR}(A \neq B) \quad \dfrac{t(\Pi_1) \text{ from I.H.}}{\Gamma, B \supset A : x \vdash \Delta}}{\Gamma(A \approx B : x) \vdash \Delta}\text{cut}$$

Case $\approx_{L4}$:

$$\dfrac{\dfrac{\Pi_1}{\Gamma, A \otimes C \approx B \otimes D : x \vdash \Delta}}{\Gamma(A \approx B : x, C \approx D : x) \vdash \Delta}\approx_{L4}$$

$$\wr\!\wr$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma(A \approx B : x, C \approx D : x) \vdash \Delta, A \otimes C \approx A \otimes C : x}\approx_R}{\Gamma(A \approx B : x, C \approx D : x) \vdash \Delta, A \otimes C \approx A \otimes D : x}\approx_{LR}(C \neq D)}{\Gamma(A \approx B : x, C \approx D : x) \vdash \Delta, A \otimes C \approx B \otimes D : x}\approx_{LR}(A \neq B) \quad \dfrac{t(\Pi_1) \text{ from I.H.}}{\Gamma, A \otimes C \approx B \otimes D : x \vdash \Delta}}{\Gamma(A \approx B : x, C \approx D : x) \vdash \Delta}\text{cut}$$

Case $\approx_{L4'}$:

$$\dfrac{\dfrac{\Pi_1}{\Gamma, A \otimes A \approx B \otimes B : x \vdash \Delta}}{\Gamma(A \approx B : x) \vdash \Delta}\approx_{L4'}$$

$$\wr\!\wr$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma(A \approx B : x) \vdash \Delta, A \otimes A \approx A \otimes A : x}\approx_R}{\Gamma(A \approx B : x) \vdash \Delta, A \otimes A \approx B \otimes B : x}\approx_{LR}(A \neq B) \quad \dfrac{t(\Pi_1) \text{ from I.H.}}{\Gamma, A \otimes A \approx B \otimes B : x \vdash \Delta}}{\Gamma(A \approx B : x) \vdash \Delta}\text{cut}$$

◀

5.2 Cut Elimination in L^{2ec}_{ISCI}

We now eliminate the cut rule from L^{2ec}_{ISCI}. The cut free version of L^{2ec}_{ISCI} is denoted L^{2e}_{ISCI} (the c superscript is dropped). Let us write $h(\Pi)$ for the height of a proof Π defined as the length of its longest branch. For a proof system S and a formula or labelled sequent s, the notation $\models^{n}_{\mathsf{S}} s$ means that s is derivable in S with a proof Π such that $h(\Pi) \leqslant n$ ($n \in \mathbb{N}$).

Label substitution is defined as follows: if $\mathrm{y} \subseteq \mathrm{x}$ then $\mathrm{x}[\mathrm{u}/\mathrm{y}] = (\mathrm{x} - \mathrm{y}) \cup \mathrm{u}$, otherwise $\mathrm{x}[\mathrm{u}/\mathrm{y}] = \mathrm{x}$. For instance, $374[\emptyset/7] = \{3, 7, 4\} - \{7\} \cup \emptyset = \{3, 4\} = 34$. Label substitutions straightforwardly extend to labelled formulas and labelled sequents.

▶ **Lemma 28.** *Let* $s = \Gamma \vdash \Delta$*. If* $\models^{n}_{\mathsf{L}^{2e}_{ISCI}} s$ *then* $\models^{n}_{\mathsf{L}^{2e}_{ISCI}} s[\mathrm{u}/\mathrm{c}]$*, where* $\mathrm{c} \in \mathbf{L}^1$ *or* $\mathrm{c} = \emptyset$*.*

Proof. By induction on the height h of the proof of $\Gamma \vdash \Delta$. The base case $h = 0$ is when s is the conclusion of an axiom.

Case id: s is of the form $\Gamma(\mathrm{A} : \mathrm{x}) \vdash \Delta(\mathrm{A} : \mathrm{y})$ with $\mathrm{x} \subseteq \mathrm{y}$. If $\mathrm{c} \not\subseteq \mathrm{y}$ then $s[\mathrm{u}/\mathrm{c}] = s$ and the result is immediate. Otherwise, $\mathrm{c} \subseteq \mathrm{y}$ and $\mathrm{y} = (\mathrm{y} - \mathrm{c}) \cup \mathrm{c}$. Since $\mathrm{x} \subseteq \mathrm{y}$, $\mathrm{y} = (\mathrm{y} - \mathrm{x}) \cup \mathrm{x}$ implies $\mathrm{y} = (\mathrm{y} - (\mathrm{x} \cup \mathrm{c})) \cup (\mathrm{x} - \mathrm{c}) \cup \mathrm{c}$. Hence, $\mathrm{y}[\mathrm{u}/\mathrm{c}] = (\mathrm{y} - (\mathrm{x} \cup \mathrm{c})) \cup (\mathrm{x} - \mathrm{c}) \cup \mathrm{u}$. We then show that $s[\mathrm{u}/\mathrm{c}]$ remains an axiom for A by showing that $\mathrm{x}[\mathrm{u}/\mathrm{c}] \subseteq \mathrm{y}[\mathrm{u}/\mathrm{c}]$. If $\mathrm{c} \not\subseteq \mathrm{x}$ then $\mathrm{x}[\mathrm{u}/\mathrm{c}] = \mathrm{x} = \mathrm{x} - \mathrm{c}$ and $\mathrm{x} - \mathrm{c} \subseteq \mathrm{y}[\mathrm{u}/\mathrm{c}]$. If $\mathrm{c} \subseteq \mathrm{x}$ then $\mathrm{x}[\mathrm{u}/\mathrm{c}] = ((\mathrm{x} - \mathrm{c}) \cup \mathrm{c})[\mathrm{u}/\mathrm{c}] = (\mathrm{x} - \mathrm{c}) \cup \mathrm{u}$ and $(\mathrm{x} - \mathrm{c}) \cup \mathrm{u} \subseteq \mathrm{y}[\mathrm{u}/\mathrm{c}]$.

Case $\perp_{\mathrm{L}}$: Similar to Case id.

For the inductive case $h = n + 1$, let r be the last rule applied (which has s as a conclusion). If r requires the introduction of eigenvariables we proceed as follows.

Case $\vee_{\mathrm{L}}$: s is of the form $\Gamma, \mathrm{A} \vee \mathrm{B} : \mathrm{x} \vdash \Delta(\mathrm{C} : \mathrm{y})$ and is obtained by the rule $\vee_{\mathrm{L}}$ from the premise $s_1 = \Gamma, \mathrm{A} \vee \mathrm{B} : \mathrm{x}, \mathrm{A} : \mathrm{xa} \vdash \Delta, \mathrm{C} : \mathrm{ya}$ and $s_2 = \Gamma, \mathrm{A} \vee \mathrm{B} : \mathrm{x}, \mathrm{B} : \mathrm{xb} \vdash \Delta, \mathrm{C} : \mathrm{yb}$, where $\mathrm{a}, \mathrm{b} \not\subseteq \Gamma \cup \Delta$, which have proofs Π_1, Π_2 such that $h(\Pi_1), h(\Pi_2) \leqslant n$. We choose two labels $\mathrm{a}' \neq \mathrm{b}'$ such that $\mathrm{a}', \mathrm{b}' \not\subseteq \Gamma \cup \Delta$ and $\mathrm{a}', \mathrm{b}' \not\subseteq \mathrm{xyuabc}$. By I.H. on Π_1 and Π_2 with substitutions $[\mathrm{a}'/\mathrm{a}]$ and $[\mathrm{b}'/\mathrm{b}]$ we get proofs Π'_1 and Π'_2 of $\Gamma, \mathrm{A} \vee \mathrm{B} : \mathrm{x}, \mathrm{A} : \mathrm{xa}' \vdash \Delta, \mathrm{C} : \mathrm{ya}'$ and $\Gamma, \mathrm{A} \vee \mathrm{B} : \mathrm{x}, \mathrm{B} : \mathrm{xb}' \vdash \Delta, \mathrm{C} : \mathrm{yb}'$. Then, by I.H. on Π'_1 and Π'_2 with substitution $[\mathrm{u}/\mathrm{c}]$, we get proofs Π''_1 and Π''_2 of $\Gamma[\mathrm{u}/\mathrm{c}], \mathrm{A} \vee \mathrm{B} : \mathrm{x}[\mathrm{u}/\mathrm{c}], \mathrm{A} : \mathrm{x}[\mathrm{u}/\mathrm{c}]\mathrm{a}' \vdash \Delta[\mathrm{u}/\mathrm{c}], \mathrm{C} : \mathrm{y}[\mathrm{u}/\mathrm{c}]\mathrm{a}'$ and $\Gamma[\mathrm{u}/\mathrm{c}], \mathrm{A} \vee \mathrm{B} : \mathrm{x}[\mathrm{u}/\mathrm{c}], \mathrm{B} : \mathrm{x}[\mathrm{u}/\mathrm{c}]\mathrm{b}' \vdash \Delta[\mathrm{u}/\mathrm{c}], \mathrm{C} : \mathrm{y}[\mathrm{u}/\mathrm{c}]\mathrm{b}'$ from which we infer the conclusion $\Gamma[\mathrm{u}/\mathrm{c}], \mathrm{A} \vee \mathrm{B} : \mathrm{x}[\mathrm{u}/\mathrm{c}] \vdash \Delta[\mathrm{u}/\mathrm{c}]$ by the rule $\vee_{\mathrm{L}}$.

Case $\supset_{\mathrm{R}}$: Similar to Case $\vee_{\mathrm{L}}$.

If r does not require eigenvariables, we apply the I.H. on all of the premise of r since they have proofs of height strictly less than $n + 1$ and we conclude $s[\mathrm{u}/\mathrm{c}]$ by reapplying r. ◀

▶ **Lemma 29.** *If* $\models^{n}_{\mathsf{L}^{2e}_{ISCI}} \Gamma \vdash \Delta$ *then* $\models^{n}_{\mathsf{L}^{2e}_{ISCI}} \Gamma, \Gamma' \vdash \Delta, \Delta'$*.*

Proof. By induction on the height h of a proof Π of $\Gamma \vdash \Delta$. For $h = 0$, it is immediate that when $\Gamma \vdash \Delta$ is an axiom, then so is $\Gamma, \Gamma' \vdash \Delta, \Delta'$. For $h = n + 1$, let r be the last rule applied in Π. If r is not $\supset_{\mathrm{R}}$ or $\vee_{\mathrm{L}}$, we apply the I.H. on the premise of r and conclude by reapplying r. Otherwise, we first use Lemma 28 to replace the eigenvariables in all of the premise of r with variables not occurring in $\Gamma \cup \Gamma' \cup \Delta \cup \Delta'$ and then apply the I.H. to the modified premise before concluding with a new instance of r. ◀

▶ **Lemma 30.** *All* L^{2e}_{ISCI} *rules are height preserving invertible.*

Proof. A k-ary proof rule r with premise $s_1 \dots s_k$ and conclusion s is height preserving invertible if $\models^{n}_{\mathsf{L}^{2e}_{ISCI}} s$ implies $\models^{n}_{\mathsf{L}^{2e}_{ISCI}} s_i$ for all $1 \leqslant i \leqslant k$. Let $s = \Gamma \vdash \Delta$. Since proof rules are non-destructive, each premiss s_i can be represented as $\Gamma, \Gamma_i \vdash \Delta, \Delta_i$, where Γ_i, Δ_i are the active parts of r. If $k = 0$ (for axioms), the result is immediate. Otherwise, if we have a proof Π of s, then by Lemma 29, we have a proof Π_i of s_i such that $h(\Pi_i) \leqslant h(\Pi)$. ◀

▶ **Lemma 31.** *If* $\vDash^{n}_{\mathsf{L}^{2e}_{\mathsf{ISCI}}} \Gamma(\mathrm{A}:\mathrm{x}, \mathrm{A}:\mathrm{y}) \vdash \Delta$ *and* $\mathrm{x} \subseteq \mathrm{y}$ *then* $\vDash^{n}_{\mathsf{L}^{2e}_{\mathsf{ISCI}}} \Gamma(\mathrm{A}:\mathrm{x}) \vdash \Delta$*. Similarly, if* $\vDash^{n}_{\mathsf{L}^{2e}_{\mathsf{ISCI}}} \Gamma \vdash \Delta(\mathrm{A}:\mathrm{x}, \mathrm{A}:\mathrm{y})$ *and* $\mathrm{y} \subseteq \mathrm{x}$ *then* $\vDash^{n}_{\mathsf{L}^{2e}_{\mathsf{ISCI}}} \Gamma \vdash \Delta(\mathrm{A}:\mathrm{x})$*.*

Proof. By induction on the height of the proofs, using Lemma 30. ◀

▶ **Lemma 32.** *If* Π *is a proof of either* $\Gamma, \mathrm{A}:\mathrm{x} \vdash \Delta$*, or* $\Gamma \vdash \Delta, \mathrm{A}:\mathrm{x}$*, in which* $\mathrm{A}:\mathrm{x}$ *is never principal for any sequent in* Π*, then there exists a proof* Π' *of* $\Gamma \vdash \Delta$ *such that* $h(\Pi') \leqslant h(\Pi)$*.*

Proof. By induction on the height of the proof Π deleting all occurrences of $\mathrm{A}:\mathrm{x}$. ◀

▶ **Theorem 33** (Cut elimination)**.** *The* cut *rule is admissible in* $\mathsf{L}^{2ec}_{\mathsf{ISCI}}$*.*

Proof. Our proof follows the pattern given in [10] or in [8] for Boolean BI. We define the cut rank of (an instance) of the cut rule as the pair $(|\mathrm{C}|, h(\Pi_1) + h(\Pi_2))$, where C is the cut formula and $\Pi_{i\in\{1,2\}}$ is the proof whose conclusion is the sequent s_i corresponding to the i-th premiss above the cut. For the base case we consider that one of the premiss of the cut has a proof of height 0. For the inductive step, we distinguish three cases: $\mathrm{C}:\mathrm{z}$ is not principal in s_1, $\mathrm{C}:\mathrm{z}$ is principal only in s_1, $\mathrm{C}:\mathrm{z}$ is principal in both s_1 and s_2. We only do a few illustrative or difficult cases. More cases are given in the appendix (see Theorem 43).

Cases $\mathrm{n_1}.\bot_\mathrm{L}$, $\mathrm{p_1}.\bot_\mathrm{L}$: s_1 is the conclusion of $\bot_\mathrm{L}$. If $\mathrm{C}:\mathrm{z}$ is not principal in s_1 (Case $\mathrm{n_1}.\bot_\mathrm{L}$), then let $\mathrm{A}:\mathrm{y}$ denote the principal formula of $\bot_\mathrm{L}$ in s_1. By the side condition of $\bot_\mathrm{L}$, we have $\mathrm{x} \subseteq \mathrm{y}$. If $\mathrm{C}:\mathrm{z}$ is principal in s_1 (Case $\mathrm{p_1}.\bot_\mathrm{L}$), then by the connectedness property, $\bot:\mathrm{x} \in \Gamma$ implies $\mathrm{A}:\mathrm{y} \in \Delta$ for some A and y such that $\mathrm{x} \subseteq \mathrm{y}$. In both cases, we eliminate the cut rule as follows:

$$\dfrac{\dfrac{}{\Gamma(\bot:\mathrm{x}) \vdash \Delta(\mathrm{A}:\mathrm{y}), \mathrm{C}:\mathrm{z}}\,\bot_\mathrm{L} \qquad \begin{array}{c}\Pi_2\\ \vdots \\ \mathrm{C}:\mathrm{z}, \Gamma \vdash \Delta(\mathrm{A}:\mathrm{y})\end{array}}{\Gamma(\bot:\mathrm{x}) \vdash \Delta(\mathrm{A}:\mathrm{y})}\,\mathrm{cut} \quad \rightsquigarrow \quad \dfrac{}{\Gamma(\bot:\mathrm{x}) \vdash \Delta(\mathrm{A}:\mathrm{y})}\,\bot_\mathrm{L}$$

Case $\mathrm{p_1}.\vee_\mathrm{R}\mathrm{p_2}.\vee_\mathrm{L}$: $\mathrm{C}:\mathrm{z}$ is principal in both s_1 and s_2, C has the form $\mathrm{A} \vee \mathrm{B}$, $\mathrm{z} \subseteq \mathrm{y}$.

$$\dfrac{\dfrac{\begin{array}{c}\Pi_1\\ \vdots \\ \Gamma \vdash \Delta, \mathrm{A}\vee\mathrm{B}:\mathrm{z}, \mathrm{A}:\mathrm{z}, \mathrm{B}:\mathrm{z}\end{array}}{\Gamma \vdash \Delta, \mathrm{A}\vee\mathrm{B}:\mathrm{z}}\,\vee_\mathrm{R} \qquad \dfrac{\begin{array}{c}\Pi_2^1\\ \vdots \\ \mathrm{A}:\mathrm{za}, \mathrm{A}\vee\mathrm{B}:\mathrm{z}, \Gamma \vdash \Delta, \mathrm{D}:\mathrm{ya}\end{array} \quad \begin{array}{c}\Pi_2^2\\ \vdots \\ \mathrm{B}:\mathrm{zb}, \mathrm{A}\vee\mathrm{B}:\mathrm{z}, \Gamma \vdash \Delta, \mathrm{D}:\mathrm{yb}\end{array}}{\mathrm{A}\vee\mathrm{B}:\mathrm{z}, \Gamma \vdash \Delta(\mathrm{D}:\mathrm{y})}\,\vee_\mathrm{L}}{\Gamma \vdash \Delta}\,\mathrm{cut}$$

We first use a cut on $\mathrm{A}\vee\mathrm{B}:\mathrm{z}$ of strictly lower cut height to get the following proof:

$$\Pi_3 \left\{ \dfrac{\begin{array}{c}\Pi_1\\ \vdots \\ \Gamma \vdash \Delta, \mathrm{A}\vee\mathrm{B}:\mathrm{z}, \mathrm{A}:\mathrm{z}, \mathrm{B}:\mathrm{z}\end{array} \qquad \dfrac{\Pi_2^{1'} \text{ from Lemma 29} \quad \Pi_2^{2'} \text{ from Lemma 29}}{\mathrm{A}\vee\mathrm{B}:\mathrm{z}, \Gamma \vdash \Delta, \mathrm{A}:\mathrm{z}, \mathrm{B}:\mathrm{z}}\,\vee_\mathrm{L}}{\Gamma \vdash \Delta, \mathrm{A}:\mathrm{z}, \mathrm{B}:\mathrm{z}}\,\mathrm{cut} \right.$$

We apply Lemma 28 on Π_2^1 with $[\emptyset/\mathrm{a}]$ and on Π_2^2 with $[\emptyset/\mathrm{b}]$ to get:

$$\Pi_4 \left\{ \begin{array}{c}\Pi_2^1[\emptyset/\mathrm{a}]\\ \vdots \\ \mathrm{A}:\mathrm{z}, \mathrm{A}\vee\mathrm{B}:\mathrm{z}, \Gamma \vdash \Delta(\mathrm{D}:\mathrm{y}), \mathrm{D}:\mathrm{y}\end{array} \right. \qquad \Pi_5 \left\{ \begin{array}{c}\Pi_2^2[\emptyset/\mathrm{b}]\\ \vdots \\ \mathrm{B}:\mathrm{z}, \mathrm{A}\vee\mathrm{B}:\mathrm{z}, \Gamma \vdash \Delta(\mathrm{D}:\mathrm{y}), \mathrm{D}:\mathrm{y}\end{array} \right.$$

We apply Lemma 31 on Π_4 and Π_5 to get:

$$\begin{array}{c}\Pi_4'\\ \vdots \\ \mathrm{A}:\mathrm{z}, \mathrm{A}\vee\mathrm{B}:\mathrm{z}, \Gamma \vdash \Delta(\mathrm{D}:\mathrm{y})\end{array} \qquad \begin{array}{c}\Pi_5'\\ \vdots \\ \mathrm{B}:\mathrm{z}, \mathrm{A}\vee\mathrm{B}:\mathrm{z}, \Gamma \vdash \Delta(\mathrm{D}:\mathrm{y})\end{array}$$

We use two cuts on $\mathrm{A}\vee\mathrm{B}:\mathrm{z}$ of strictly lower cut height to get Π_6, Π_7, which are finally combined with Π_3 to obtain a proof with two cuts on strictly smaller formulas.

$$\Pi_6 \left\{ \dfrac{\dfrac{\dfrac{\Pi_1' \text{ from Lemma 29}}{\mathrm{A:z,\Gamma \vdash \Delta, A\vee B:z, A:z, B:z}}}{\mathrm{A:z,\Gamma\vdash\Delta,A\vee B:z}}\vee_{\mathrm{R}} \qquad \dfrac{\Pi_4'}{\mathrm{A:z,A\vee B:z,\Gamma\vdash\Delta(D:y)}}}{\mathrm{A:z,\Gamma\vdash\Delta}}\,\mathrm{cut} \right.$$

$$\Pi_7 \left\{ \dfrac{\dfrac{\dfrac{\Pi_1'' \text{ from Lemma 29}}{\mathrm{B:z,\Gamma \vdash \Delta, A\vee B:z, A:z, B:z}}}{\mathrm{B:z,\Gamma\vdash\Delta,A\vee B:z}}\vee_{\mathrm{R}} \qquad \dfrac{\Pi_5'}{\mathrm{B:z,A\vee B:z,\Gamma\vdash\Delta(D:y)}}}{\mathrm{B:z,\Gamma\vdash\Delta}}\,\mathrm{cut} \right.$$

$$\dfrac{\dfrac{\dfrac{\Pi_3}{\mathrm{\Gamma\vdash\Delta,A:z,B:z}} \qquad \dfrac{\Pi_7' \text{ from Lemma 29}}{\mathrm{B:z,\Gamma\vdash\Delta,A:z}}}{\mathrm{\Gamma\vdash\Delta,A:z}}\,\mathrm{cut} \qquad \dfrac{\Pi_6}{\mathrm{A:z,\Gamma\vdash\Delta}}}{\mathrm{\Gamma\vdash\Delta}}\,\mathrm{cut}$$

◀

6 Liberalizing $\mathsf{L}^{2e}_{\mathsf{ISCI}}$ and Decidability

Even restricted to the simple case of intuitionistic logic, the termination of a labelled proof system is not straightforward. A problem is the rules $\supset_{\mathrm{L}}$ and $\vee_{\mathrm{R}}$ (called *β-rules*) can be used several times as long as there are yet untried labels satisfying their requirements. Combined with the fact that the rules $\supset_{\mathrm{R}}$ and $\vee_{\mathrm{L}}$ (called *α-rules*) require the systematic introduction of fresh singleton labels, the proof-search process might degenerate into the construction of infinite branches when there are α-formulas in the scope of β-formulas.

Let us assume a globally fixed[1] total injective *indexing function* $\mathrm{i} : \mathbf{F}\times\mathbb{N}\to\mathbf{L}^1$ that given a formula A and an index n maps the pair (A,n) to the singleton label denoted $\mathrm{i}^{\mathrm{n}}_{\mathrm{A}}$. We define $\mathsf{L}^{2}_{\mathsf{ISCI}}$ as the labelled proof system obtained from $\mathsf{L}^{2e}_{\mathsf{ISCI}}$ in which the eigenvariable requirements are dropped by replacing the α-rules of Figure 4 with the following ones:

$$\dfrac{\mathrm{\Gamma, A : i^1_{A\supset B}\vdash \Delta, B : x\cup i^1_{A\supset B}}}{\mathrm{\Gamma\vdash\Delta(A\supset B:x)}}\supset_{\mathrm{R}}$$

$$\dfrac{\mathrm{\Gamma, A:x\cup i^1_{A\vee B}\vdash\Delta, C:y\cup i^1_{A\vee B}} \qquad \mathrm{\Gamma, B:x\cup i^2_{A\vee B}\vdash\Delta, C:y\cup i^2_{A\vee B}}}{\mathrm{\Gamma(A\vee B:x)\vdash\Delta(C:y)}}\vee_{\mathrm{L}}(\mathrm{x}\subseteq\mathrm{y})$$

▶ **Theorem 34.** *If* $\Vdash^{\mathrm{n}}_{\mathsf{L}^{2e}_{\mathsf{ISCI}}}\mathrm{A}$ *then* $\Vdash^{\mathrm{n}}_{\mathsf{L}^{2}_{\mathsf{ISCI}}}\mathrm{A}$.

Proof. By induction on the height of the $\mathsf{L}^{2e}_{\mathsf{ISCI}}$ proof of A. ◀

6.1 Validity of the Replacement Law for ISCI

▶ **Lemma 35.** *Let* A, B, C *be formulas and let* $\mathrm{C[A\mapsto B]}$ *be the formula obtained from* C *by simultaneously replacing all occurrences of* A *in* C *with* B*. Then, the formula* $\mathrm{(A\approx B)\supset(C\approx C[A\mapsto B])}$ *is valid in Beth semantics.*

Proof. Let $\mathcal{M}$ be a Beth model and m be a world such that $m\Vdash\mathrm{A\approx B}$. If C does not contain any occurrence of A then $\mathrm{C[A\mapsto B]=C}$ and condition $\mathcal{M}_{\approx_1}$ of Definition 3 then implies $m\Vdash\mathrm{C\approx C}$. If C contains at least one occurrence of A, let $d(\mathrm{F,C})$ denote the depth at which a subformula F is nested in C. We proceed by induction on the depth

[1] The use of a globally fixed indexing function is just for technical convenience. One could also associate each derivation with a partial indexing function defined only on the formulas occurring in that derivation.

$d = min\{\, d(\mathrm{A},\mathrm{C}) \mid \mathrm{A} \in \mathrm{C}\,\}$ of the least deeply nested occurrence(s) of A in C (e.g., if $\mathrm{C} = (\mathrm{A} \supset \mathrm{D}) \wedge ((\mathrm{A} \vee \mathrm{B}) \approx \mathrm{A})$ then $d = 2$). The base case is when $d = 0$, i.e., when $\mathrm{C} = \mathrm{A}$. Thus, $\mathrm{C}[\mathrm{A} \mapsto \mathrm{B}] = \mathrm{B}$ and we have $m \Vdash \mathrm{A} \approx \mathrm{B}$ by assumption. For the inductive case, C is of the form $\mathrm{C}_1 \otimes \mathrm{C}_2$, where $\otimes$ is a binary connective. We assume as an I.H. that the property holds for all formulas C and all d' such that $0 \leq d' < d$. By definition of a substitution, $(\mathrm{C}_1 \otimes \mathrm{C}_2)[\mathrm{A} \mapsto \mathrm{B}] = \mathrm{C}_1[\mathrm{A} \mapsto \mathrm{B}] \otimes \mathrm{C}_2[\mathrm{A} \mapsto \mathrm{B}]$. For $\mathrm{C_i} \in \{\,\mathrm{C}_1, \mathrm{C}_2\,\}$, if A does not occur in $\mathrm{C_i}$ then $\mathrm{C_i}[\mathrm{A} \mapsto \mathrm{B}] = \mathrm{C_i}$. Thus, $m \Vdash \mathrm{C_i} \approx \mathrm{C_i}[\mathrm{A} \mapsto \mathrm{B}]$ by condition $\mathcal{M}_{\approx_1}$ of Definition 3. Otherwise, if A occurs in $\mathrm{C_i}$ then $m \Vdash \mathrm{C_i} \approx \mathrm{C_i}[\mathrm{A} \mapsto \mathrm{B}]$ by I.H. Hence, by condition $\mathcal{M}_{\approx_4}$ of Definition 3, we get $(\mathrm{C}_1 \otimes \mathrm{C}_2) \approx (\mathrm{C}_1[\mathrm{A} \mapsto \mathrm{B}] \otimes \mathrm{C}_2[\mathrm{A} \mapsto \mathrm{B}])$. ◀

▶ **Lemma 36.** *If $m \Vdash \mathrm{A} \approx \mathrm{B}$ then for all formulas C, $m \Vdash \mathrm{C}$ iff $m \Vdash \mathrm{C}[\mathrm{A} \mapsto \mathrm{B}]$.*

Proof. By Lemma 35, if $m \Vdash \mathrm{A} \approx \mathrm{B}$ then $m \Vdash \mathrm{C} \approx \mathrm{D}$, where $\mathrm{D} = \mathrm{C}[\mathrm{A} \mapsto \mathrm{B}]$. By symmetry of $\approx$, $m \Vdash \mathrm{C} \approx \mathrm{D}$ implies $m \Vdash \mathrm{D} \approx \mathrm{C}$. Therefore, by condition $\mathcal{M}_{\approx_3}$ of Definition 3, we get both $m \Vdash \mathrm{D} \supset \mathrm{C}$ and $m \Vdash \mathrm{C} \supset \mathrm{D}$. Consequently, if $m \Vdash \mathrm{C}$, then $m \Vdash \mathrm{C} \supset \mathrm{D}$ implies $m \Vdash \mathrm{D}$. Conversely, if $m \Vdash \mathrm{D}$, then $m \Vdash \mathrm{D} \supset \mathrm{C}$ implies $m \Vdash \mathrm{C}$. ◀

6.2 Liberalized Soundness

To show that $\mathsf{L}^2_{\mathsf{ISCI}}$ is sound even in the absence of the eigenvariable condition, we take advantage of the completeness of ISCI w.r.t. regular Beth models (Theorem 14) by semantically interpreting (realizing) the unique index $\mathrm{i_A}$ of a formula A by an A-minimal world.

▶ **Definition 37** (Realization). *Let $\mathcal{M}$ be a regular Beth model. Let $s = \Gamma \vdash \Delta$ be a labelled sequent. A* realization of s in $\mathcal{M}$ *is a partial function $\rho : \mathbf{L} \to \mathbf{M}$ such that:*

- *$\rho(\emptyset) = \omega$, $\rho(\mathbb{N}) = \pi$, $\rho(\mathrm{i}^{\mathrm{n}\in\{1,2\}}_{\mathrm{A}_1\otimes\mathrm{A}_2}) = m_{\mathrm{A_n}}$ for all $\mathrm{i}^{\mathrm{n}\in\{1,2\}}_{\mathrm{A}_1\otimes\mathrm{A}_2} \subseteq s$ and $\rho(\mathrm{x} \cup \mathrm{y}) = \rho(\mathrm{x}) \sqcup \rho(\mathrm{y})$,*
- *for all $\mathrm{x}, \mathrm{y} \subseteq \Gamma$, if $\mathrm{x} \subseteq \mathrm{y}$ then $\rho(\mathrm{x}) \leqslant \rho(\mathrm{y})$ holds in $\mathcal{M}$,*
- *for all $\mathrm{A} : \mathrm{x}$ in Γ, $\rho(\mathrm{x}) \Vdash \mathrm{A}$ and for all $\mathrm{A} : \mathrm{x}$ in Δ, $\rho(\mathrm{x}) \nVdash \mathrm{A}$.*

A sequent s is realizable in $\mathcal{M}$ *if there exists a realization of s in $\mathcal{M}$, and* realizable *if it is realizable in some regular Beth model $\mathcal{M}$.*

▶ **Lemma 38.** *If the sequent $s = \Gamma \vdash \Delta$ is an initial sequent in an $\mathsf{L}^2_{\mathsf{ISCI}}$-proof, i.e., a leaf sequent that is the conclusion of a zero-premiss rule, then s is not realizable.*

Proof. If s is realizable, then we have a realization ρ of s in some regular Beth model $\mathcal{M}$. We proceed by case analysis on the zero-premiss rule of which s is the conclusion.

Case id: $s = \Gamma, \mathrm{A} : \mathrm{x} \vdash \Delta, \mathrm{A} : \mathrm{y}$ with $\mathrm{x} \subseteq \mathrm{y}$, which implies the contradiction $\rho(\mathrm{y}) \nVdash \mathrm{A}$ since $\rho(\mathrm{x}) \leqslant \rho(\mathrm{y})$ and $\rho(\mathrm{x}) \Vdash \mathrm{A}$ imply $\rho(\mathrm{y}) \Vdash \mathrm{A}$ by Kripke monotonicity.

Case $\perp_\mathrm{L}$: $s = \Gamma, \perp : \mathrm{x} \vdash \Delta, \mathrm{A} : \mathrm{y}$ with $\mathrm{x} \subseteq \mathrm{y}$, which implies the contradiction $\rho(\mathrm{y}) \nVdash \mathrm{A}$ since $\rho(\mathrm{x}) = \rho(\mathrm{y}) = \pi$ and $\pi \Vdash \mathrm{A}$ for all A.

Case $\approx_\mathrm{R}$: $s = \Gamma \vdash \Delta, \mathrm{A} \approx \mathrm{A} : \mathrm{x}$, which implies the contradiction $\rho(\mathrm{x}) \nVdash \mathrm{A} \approx \mathrm{A}$. ◀

▶ **Lemma 39.** *Every proof rule in $\mathsf{L}^2_{\mathsf{ISCI}}$ preserves realizability in regular Beth models.*

Proof. By case analysis of the proof rules of $\mathsf{L}^2_{\mathsf{ISCI}}$. We show that whenever the conclusion of a rule is realizable in some regular model $\mathcal{M}$ for some realization ρ, then at least one of its premise is also realizable in $\mathcal{M}$ for some extension of ρ. We write $s = \Gamma \vdash \Delta$ for the sequent which is the conclusion of the rule and $s_i = \Gamma_i \vdash \Delta_i$ for the i-th premiss (for $i \in \{\,1, 2\,\}$). Since ρ realizes both Γ and Δ in s, ρ also realizes Γ_i and Δ_i in s_i since $\Gamma_i \subseteq \Gamma$ and $\Delta_i \subseteq \Delta$. Therefore, we only need to consider the principal and active parts of each rule.

Case $\vee_{\mathrm{L}}$**:** If ρ realizes $s = \Gamma(\mathrm{A} \vee \mathrm{B} : \mathrm{x}) \vdash \Delta(\mathrm{C} : \mathrm{y})$ in $\mathcal{M}$, then $\rho(\mathrm{x}) \Vdash \mathrm{A} \vee \mathrm{B}$ implies that there exist $n_1, n_2 \in \mathbf{M}$ such that $n_1 \sqcap n_2 \leqslant \rho(\mathrm{x})$, $n_1 \Vdash \mathrm{A}$ and $n_2 \Vdash \mathrm{B}$. Moreover, $\mathrm{a} = \mathrm{i}^1_{\mathrm{A}\vee\mathrm{B}}$ and $\mathrm{b} = \mathrm{i}^2_{\mathrm{A}\vee\mathrm{B}}$. If $\rho(\mathrm{a})$ is already defined then $\rho(\mathrm{a}) = m_{\mathrm{A}}$ by definition. Otherwise, we extend ρ by setting $\rho(\mathrm{a}) = m_{\mathrm{A}}$. We proceed similarly for $\rho(\mathrm{b})$ to get $\rho(\mathrm{b}) = m_{\mathrm{B}}$. Since m_{A} is A-minimal, we get $m_{\mathrm{A}} \leqslant n_1$ and $\rho(\mathrm{x}) \sqcup \rho(\mathrm{a}) \Vdash \mathrm{A}$. Similarly, since m_{B} is B-minimal, we get $m_{\mathrm{B}} \leqslant n_2$ and $\rho(\mathrm{x}) \sqcup \rho(\mathrm{b}) \Vdash \mathrm{B}$. Moreover, $m_{\mathrm{A}} \leqslant n_1$ and $m_{\mathrm{B}} \leqslant n_2$ imply $m_{\mathrm{A}} \sqcap m_{\mathrm{B}} \leqslant n_1 \sqcap n_2$. Thus, $\mathrm{xa} \cap \mathrm{xb} = \mathrm{x}$ implies $(\rho(\mathrm{x}) \sqcup \rho(\mathrm{a})) \sqcap (\rho(\mathrm{x}) \sqcup \rho(\mathrm{b})) = \rho(\mathrm{x})$. Now if both $\rho(\mathrm{y}) \sqcup \rho(\mathrm{a}) \Vdash \mathrm{C}$ and $\rho(\mathrm{y}) \sqcup \rho(\mathrm{b}) \Vdash \mathrm{C}$ then, since $\mathcal{M}$ is a regular Beth model, there exists a C-minimal world m_{C}. Thus, $m_{\mathrm{C}} \leqslant \rho(\mathrm{y}) \sqcup \rho(\mathrm{a})$ and $m_{\mathrm{C}} \leqslant \rho(\mathrm{y}) \sqcup \rho(\mathrm{b})$, which implies $m_{\mathrm{C}} \leqslant (\rho(\mathrm{y}) \sqcup \rho(\mathrm{a})) \sqcap (\rho(\mathrm{y}) \sqcup \rho(\mathrm{b})) = \rho(\mathrm{y})$. Hence, $\rho(\mathrm{y}) \Vdash \mathrm{C}$, which is a contradiction since $\rho(\mathrm{y}) \nVdash \mathrm{C}$ by definition. Therefore, either $\rho(\mathrm{y}) \sqcup \rho(\mathrm{a}) \nVdash \mathrm{C}$ and s_1 is realizable, or $\rho(\mathrm{y}) \sqcup \rho(\mathrm{b}) \nVdash \mathrm{C}$ and s_2 is realizable.

Case $\vee_{\mathrm{R}}$**:** If ρ realizes $s = \Gamma \vdash \Delta(\mathrm{A} \vee \mathrm{B} : \mathrm{x})$ then $\rho(\mathrm{x}) \nVdash \mathrm{A} \vee \mathrm{B}$. Suppose that $\rho(\mathrm{x}) \Vdash \mathrm{A}$. Since $\mathcal{M}$ is regular there exists an A-minimal world m_{A}. Since $m_{\mathrm{A}} \Vdash \mathrm{A}$ and $\pi \Vdash \mathrm{B}$ by definition, we have $m_{\mathrm{A}} \sqcap \pi = m_{\mathrm{A}} \leqslant \rho(\mathrm{x})$ which implies the contradiction $\rho(\mathrm{x}) \Vdash \mathrm{A} \vee \mathrm{B}$. Similarly, if $\rho(\mathrm{x}) \Vdash \mathrm{B}$ we also get the contradiction $\rho(\mathrm{x}) \Vdash \mathrm{A} \vee \mathrm{B}$. Hence, s_1 is realizable.

Case $\approx_{\mathrm{LR}}$**:** This case directly follows from Lemma 36.

The other cases are similar. ◀

▶ **Theorem 40** (Liberalized soundness). *If* $\vdash_{\mathsf{L}^2_{\mathsf{ISCI}}} \mathrm{A}$ *then* $\vDash^{\mathrm{r}} \mathrm{A}$.

Proof. Suppose that $\vdash_{\mathsf{L}^2_{\mathsf{ISCI}}} \mathrm{A}$, then there exists an $\mathsf{L}^2_{\mathsf{ISCI}}$-proof Π of $\vdash \mathrm{A} : \emptyset$. If $\nvDash^{\mathrm{r}} \mathrm{A}$, then there is a regular Beth model $\mathcal{M}$ such that $\omega \nVdash \mathrm{A}$. Since $\vdash \mathrm{A} : \emptyset$ is trivially realizable, Lemma 39 implies that Π contains a branch the sequents of which are all realizable. Since Π is a proof, this branch ends with an initial sequent s that is the conclusion of an axiom rule. Lemma 38 then implies that s is not realizable, which is a contradiction. Therefore, $\vDash^{\mathrm{r}} \mathrm{A}$. ◀

6.3 Termination and Decidability

Giving a full-fledged proof that $\mathsf{L}^2_{\mathsf{ISCI}}$ is a terminating proof-system is out of the scope of this paper as it would require a detailed proof-search algorithm with a well defined proof strategy. Moreover, since $\mathsf{L}^2_{\mathsf{ISCI}}$ proof rules as formulated non-destructively, we would also need a suitable notion of (sequent) saturation to decide whether a labelled formula is fully analyzed or not. For instance, an occurrence of $\mathrm{A} \wedge \mathrm{B} : \mathrm{x}$ on the left-hand side of a sequent $\Gamma \vdash \Delta$ would be considered fully analyzed whenever $\mathrm{A} : \mathrm{y}$ and $\mathrm{B} : \mathrm{z}$ occur in Γ for some labels y, z such that $\mathrm{y}, \mathrm{z} \subseteq \mathrm{x}$. We now sketch the proof that $\mathsf{L}^2_{\mathsf{ISCI}}$ has a finite proof search space.

▶ **Theorem 41** (Termination). $\mathsf{L}^2_{\mathsf{ISCI}}$ *is a terminating proof system.*

Proof sketch. Firstly, without any eigenvariable requirements, only finitely many singleton labels can occur in an $\mathsf{L}^2_{\mathsf{ISCI}}$ derivation of A. Since labels occurring in an $\mathsf{L}^2_{\mathsf{ISCI}}$ derivation of A are finite unions of singleton labels, there can only be finitely many of them. Secondly, let $n = |\mathrm{A}|$ and let $At(\mathrm{A})$ be the set of propositional letters occurring in A. It is easy to see that the active formula introduced by an instance of the rule $\approx_{\mathrm{LR}}$ has a size $m \leqslant n$ and is built using only atoms in $At(\mathrm{A})$ (this can be viewed as a weak form of subformula property). There can only be finitely many formulas of size $\leqslant n$ built from $At(\mathrm{A})$. Finally, with a finitely many subformulas and labels, one can only generate a finite number of labelled formulas. Therefore, only finitely many unsaturated labelled sequents can occur in a $\mathsf{L}^2_{\mathsf{ISCI}}$ derivation of A. Thus, the proof search space for $\vdash \mathrm{A} : \emptyset$ in $\mathsf{L}^2_{\mathsf{ISCI}}$ is finite. ◀

▶ **Corollary 42** (Decidability). ISCI *is a decidable logic.*

References

1 S.L. Bloom and R. Suszko. Investigations into the Sentential Calculus with Identity. *Notre Dame Journal of Formal Logic*, 13(3):289–308, 1972.
2 S. Chlebowski. Sequent Calculi for SCI. *Studia Logica*, 106(3):541–563, 2018.
3 S. Chlebowski and D. Leszczyńska-Jasion. An Investigation into Intuitionistic Logic with Identity. *Bulletin of the Section of Logic*, 48(4):259–283, 2019.
4 D. Galmiche, D. Méry, and D. Pym. The semantics of BI and Resource Tableaux. *Mathematical Structures in Computer Science*, 15(6):1033–1088, 2005.
5 D. Galmiche and D. Méry. A Connection-based Characterization of Bi-intuitionistic Validity. *Journal of Automated Reasoning*, 51(1):3–26, 2013. doi:10.1007/s10817-013-9279-4.
6 J. Golińska-Pilarek. Rasiowa–Sikorski proof system for the non-Fregean sentential logic. *Journal of Applied Non-Classical Logics*, 17(4):511–519, 2007.
7 J. Golińska-Pilarek and T. Huuskonen. Non-Fregean Propositional Logic with Quantifiers. *Notre Dame Journal of Formal Logic*, 57(2):249–279, 2016. doi:10.1215/00294527-3470547.
8 Z. Hou, R. Goré, and A. Tiu. A labelled sequent calculus for BBI: Proof theory and proof search. *Journal of Logic and Computation*, 28(4):809–872, 2018. doi:10.1007/978-3-642-40537-2_16.
9 P. Lukowski. Intuitionistic sentential calculus with identity. *Bulletin of the Section of Logic*, 19(3):92–99, 1990.
10 S. Negri and J. von Plato. Cut elimination in the presence of axioms. *Bulletin of Symbolic Logic*, 4(4):418–435, 1998.
11 S. Negri and J. von Plato. *Proof Analysis - A Contribution to Hilbert's Last Problem*. Cambridge University Press, 2014. URL: http://www.cambridge.org/de/academic/subjects/philosophy/logic/proof-analysis-contribution-hilberts-last-problem?format=PB.
12 R. Suszko. Abolition of the Fregean axiom. In *Logic Colloquium*, pages 169–239, 1975. Springer.
13 A. Wasilewska. DFC-algorithms for Suszko logic and one-to-one Gentzen type formalizations. *Studia Logica*, 43(4):395–404, 1984.

A Appendix

A.1 Cut Elimination

▶ **Theorem 43** (Cut elimination). *The* cut *rule is admissible in* $\mathsf{L}^{2ec}_{\mathsf{ISCI}}$.

Proof. Our proof follows the pattern given in [10] or in [8] for Boolean BI. We define the cut rank of (an instance) of the cut rule as the pair $(|\mathrm{C}|, h(\Pi_1) + h(\Pi_2))$, where C is the cut formula and $\Pi_{i\in\{1,2\}}$ is the proof whose conclusion is the sequent s_i corresponding to the i-th premiss above the cut. For the base case we consider that one of the premiss of the cut has a proof of height 0. For the inductive step, we distinguish three cases: $\mathrm{C}:\mathrm{z}$ is not principal in s_1, $\mathrm{C}:\mathrm{z}$ is principal only in s_1, $\mathrm{C}:\mathrm{z}$ is principal in both s_1 and s_2.

Case n_1.id: s_1 is the conclusion of id, $\mathrm{C}:\mathrm{z}$ is not principal in s_1, $\mathrm{x} \subseteq \mathrm{y}$.

$$\dfrac{\dfrac{}{\Gamma(\mathrm{A}:\mathrm{x}) \vdash \Delta(\mathrm{A}:\mathrm{y}), \mathrm{C}:\mathrm{z}}\,\mathrm{id} \qquad \begin{array}{c}\Pi_2 \\ \hdashline \mathrm{C}:\mathrm{z}, \Gamma \vdash \Delta\end{array}}{\Gamma(\mathrm{A}:\mathrm{x}) \vdash \Delta(\mathrm{A}:\mathrm{y})}\,\mathrm{cut} \quad \rightsquigarrow \quad \dfrac{}{\Gamma(\mathrm{A}:\mathrm{x}) \vdash \Delta(\mathrm{A}:\mathrm{y})}\,\mathrm{id}$$

Case p_1.id: s_1 is the conclusion of id, $\mathrm{C}:\mathrm{z}$ is principal in s_1, $\mathrm{x} \subseteq \mathrm{z}$.

$$\dfrac{\dfrac{}{\Gamma(\mathrm{C}:\mathrm{x}) \vdash \Delta, \mathrm{C}:\mathrm{z}}\,\mathrm{id} \qquad \begin{array}{c}\Pi_2 \\ \hdashline \mathrm{C}:\mathrm{z}, \Gamma \vdash \Delta\end{array}}{\Gamma(\mathrm{C}:\mathrm{x}) \vdash \Delta}\,\mathrm{cut} \quad \rightsquigarrow \quad \begin{array}{c}\Pi_2' \text{ from Lemma 31} \\ \hdashline \Gamma(\mathrm{C}:\mathrm{x}) \vdash \Delta\end{array}$$

Case n_2.id: s_2 is the conclusion of id, $\mathrm{C}:\mathrm{z}$ is not principal in s_2. Similar to Case n_1.id.
Case p_2.id: s_2 is the conclusion of id, $\mathrm{C}:\mathrm{z}$ is principal in s_2. Similar to Case p_1.id.
Case $n_1.\bot_L$: s_1 is the conclusion of $\bot_L$, $\mathrm{C}:\mathrm{z}$ is not principal in s_1, $\mathrm{x} \subseteq \mathrm{y}$.

$$\dfrac{\dfrac{}{\Gamma(\bot:\mathrm{x}) \vdash \Delta(\mathrm{A}:\mathrm{y}), \mathrm{C}:\mathrm{z}}\,\bot_L \qquad \overset{\Pi_2}{\mathrm{C}:\mathrm{z}, \Gamma \vdash \Delta}}{\Gamma(\bot:\mathrm{x}) \vdash \Delta(\mathrm{A}:\mathrm{y})}\,\mathrm{cut} \quad \rightsquigarrow \quad \dfrac{}{\Gamma(\bot:\mathrm{x}) \vdash \Delta(\mathrm{A}:\mathrm{y})}\,\bot_L$$

Case $p_1.\bot_L$: s_1 is the conclusion of $\bot_L$, $\mathrm{C}:\mathrm{z}$ is principal in s_1, $\mathrm{x} \subseteq \mathrm{z}$.

$$\dfrac{\dfrac{}{\Gamma(\bot:\mathrm{x}) \vdash \Delta, \mathrm{C}:\mathrm{z}}\,\bot_L \qquad \overset{\Pi_2}{\mathrm{C}:\mathrm{z}, \Gamma \vdash \Delta}}{\Gamma(\bot:\mathrm{x}) \vdash \Delta}\,\mathrm{cut} \quad \rightsquigarrow \quad \dfrac{\text{connectedness: } \mathrm{x} \subseteq \mathrm{u}}{\Gamma(\bot:\mathrm{x}) \vdash \Delta(\mathrm{A}:\mathrm{u})}\,\bot_L$$

Case $n_2.\bot_L$: s_2 is the conclusion of $\bot_L$, $\mathrm{C}:\mathrm{z}$ is not principal in s_2. Similar to Case $\mathrm{n}_1.\bot_L$.
Case $p_2.\bot_L$: s_2 is the conclusion of $\bot_L$, $\mathrm{C}:\mathrm{z}$ is principal in s_2. Similar to Case $\mathrm{p}_1.\bot_L$.
Case $n_1.\approx_R$: s_1 is the conclusion of $\approx_R$, $\mathrm{C}:\mathrm{z}$ is not principal in s_1.

$$\dfrac{\dfrac{}{\Gamma \vdash \Delta(\mathrm{A} \approx \mathrm{A}:\mathrm{x}), \mathrm{C}:\mathrm{z}}\,\approx_R \qquad \overset{\Pi_2}{\mathrm{C}:\mathrm{z}, \Gamma \vdash \Delta}}{\Gamma \vdash \Delta(\mathrm{A} \approx \mathrm{A}:\mathrm{x})}\,\mathrm{cut} \quad \rightsquigarrow \quad \dfrac{}{\Gamma \vdash \Delta(\mathrm{A} \approx \mathrm{A}:\mathrm{x})}\,\approx_R$$

Case $p_1.\approx_R$: s_1 is the conclusion of $\approx_R$, $\mathrm{C}:\mathrm{z}$ is principal in s_1. Then, C has the form $\mathrm{A} \approx \mathrm{A}$ for some A. Since $\mathrm{A} \neq \mathrm{A}$ is not satisfiable, $\mathrm{A} \approx \mathrm{A}:\mathrm{x}$ can never be the principal formula of an occurrence of $\approx_{LR}$ in Π_2. Therefore, the only way for $\mathrm{A} \approx \mathrm{A}:\mathrm{x}$ to be principal in Π_2 is if s_2 is the conclusion of an occurrence of id, which then implies that Δ contains an occurrence of $\mathrm{A} \approx \mathrm{A}:\mathrm{y}$ for some $\mathrm{x} \subseteq \mathrm{y}$. In this case, we have

$$\dfrac{\dfrac{}{\Gamma \vdash \Delta, \mathrm{A} \approx \mathrm{A}:\mathrm{x}}\,\approx_R \qquad \overset{\Pi_2}{\mathrm{A} \approx \mathrm{A}:\mathrm{x}, \Gamma \vdash \Delta(\mathrm{A} \approx \mathrm{A}:\mathrm{y})}}{\Gamma \vdash \Delta(\mathrm{A} \approx \mathrm{A}:\mathrm{y})}\,\mathrm{cut} \quad \rightsquigarrow \quad \dfrac{}{\Gamma \vdash \Delta(\mathrm{A} \approx \mathrm{A}:\mathrm{y})}\,\approx_R$$

Otherwise, $\mathrm{A} \approx \mathrm{A}:\mathrm{x}$ is never principal in Π_2 and we apply Lemma 32 on Π_2 to get a proof Π_2' of $\Gamma \vdash \Delta$ as follows

$$\dfrac{\dfrac{}{\Gamma \vdash \Delta, \mathrm{A} \approx \mathrm{A}:\mathrm{x}}\,\approx_R \qquad \overset{\Pi_2}{\mathrm{A} \approx \mathrm{A}:\mathrm{x}, \Gamma \vdash \Delta}}{\Gamma \vdash \Delta}\,\mathrm{cut} \quad \rightsquigarrow \quad \overset{\Pi_2' \text{ from Lemma 32}}{\Gamma \vdash \Delta}$$

Case $n_2.\approx_R$: s_2 is the conclusion of $\approx_R$, $\mathrm{C}:\mathrm{z}$ is not principal in s_2. Similar to Case $\mathrm{n}_1.\approx_R$.

$$\dfrac{\overset{\Pi_1}{\Gamma \vdash \Delta, \mathrm{C}:\mathrm{z}} \qquad \dfrac{}{\mathrm{C}:\mathrm{z}, \Gamma \vdash \Delta(\mathrm{A} \approx \mathrm{A}:\mathrm{x})}\,\approx_R}{\Gamma \vdash \Delta(\mathrm{A} \approx \mathrm{A}:\mathrm{x})}\,\mathrm{cut} \quad \rightsquigarrow \quad \dfrac{}{\Gamma \vdash \Delta(\mathrm{A} \approx \mathrm{A}:\mathrm{x})}\,\approx_R$$

Case $p_2.\approx_R$: cannot happen ($\mathrm{A} \approx \mathrm{A}:\mathrm{z}$ on the left-hand side cannot be principal for $\approx_R$).

Case $n_1.r_1$: $\mathrm{C}:\mathrm{z}$ is not principal in s_1, r is a rule with one premiss and active parts Γ', Δ'.

$$\dfrac{\dfrac{\overset{\Pi_1}{\Gamma, \Gamma' \vdash \Delta, \Delta', \mathrm{C}:\mathrm{z}}}{\Gamma \vdash \Delta, \mathrm{C}:\mathrm{z}}\,\mathrm{r} \qquad \overset{\Pi_2}{\mathrm{C}:\mathrm{z}, \Gamma \vdash \Delta}}{\Gamma \vdash \Delta}\,\mathrm{cut} \quad \rightsquigarrow \quad \dfrac{\dfrac{\overset{\Pi_1}{\Gamma, \Gamma' \vdash \Delta, \Delta', \mathrm{C}:\mathrm{z}} \qquad \overset{\Pi_2' \text{ from Lemma 29}}{\mathrm{C}:\mathrm{z}, \Gamma, \Gamma' \vdash \Delta, \Delta'}}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\,\mathrm{cut}}{\Gamma \vdash \Delta}\,\mathrm{r}$$

The rank of the new cut is $(|\mathrm{C}|, h(\Pi_1) + h(\Pi_2'))$, which is strictly lower than the rank $(|\mathrm{C}|, 1 + h(\Pi_1) + h(\Pi_2))$ of the original cut.

Case $p_1n_2.r_1$: $C:z$ is only principal in s_1, r is a rule with one premiss and active parts Γ', Δ'. Similar to Case $n_1.r_1$.

$$\dfrac{\overset{\Pi_1}{\Gamma \vdash \Delta, C:z} \qquad \dfrac{\overset{\Pi_2}{C:z, \Gamma, \Gamma' \vdash \Delta, \Delta'}}{C:z, \Gamma \vdash \Delta}\,\mathrm{r}}{\Gamma \vdash \Delta}\,\mathrm{cut} \quad \rightsquigarrow \quad \dfrac{\dfrac{\overset{\Pi_1' \text{ from Lemma 29}}{\Gamma, \Gamma' \vdash \Delta, \Delta', C:z} \qquad \overset{\Pi_2}{C:z, \Gamma, \Gamma' \vdash \Delta, \Delta'}}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\,\mathrm{cut}}{\Gamma \vdash \Delta}\,\mathrm{r}$$

The rank of the new cut is $(|C|, h(\Pi_1') + h(\Pi_2))$, which is strictly lower than the rank $(|C|, h(\Pi_1) + h(\Pi_2) + 1)$ of the original cut.

Case $n_1.r_2$: $C:z$ is not principal in s_1, r is a rule with two premise and active parts Γ', Δ' in the first premiss and Γ'', Δ'' in the second one. We apply Lemma 29 twice on Π_2 to get Π_2' and Π_2''.

$$\dfrac{\dfrac{\overset{\Pi_1^1}{\Gamma, \Gamma' \vdash \Delta, \Delta', C:z} \qquad \overset{\Pi_1^2}{\Gamma, \Gamma'' \vdash \Delta, \Delta'', C:z}}{\Gamma \vdash \Delta, C:z}\,\mathrm{r} \qquad \overset{\Pi_2}{C:z, \Gamma \vdash \Delta}}{\Gamma \vdash \Delta}\,\mathrm{cut} \quad \rightsquigarrow$$

$$\dfrac{\dfrac{\overset{\Pi_1^1}{\Gamma, \Gamma' \vdash \Delta, \Delta', C:z} \qquad \overset{\Pi_2'}{C:z, \Gamma, \Gamma' \vdash \Delta, \Delta'}}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\,\mathrm{cut} \qquad \dfrac{\overset{\Pi_1^2}{\Gamma, \Gamma'' \vdash \Delta, \Delta'', C:z} \qquad \overset{\Pi_2''}{C:z, \Gamma, \Gamma'' \vdash \Delta, \Delta''}}{\Gamma, \Gamma'' \vdash \Delta, \Delta''}\,\mathrm{cut}}{\Gamma \vdash \Delta}\,\mathrm{r}$$

The ranks $(|C|, h(\Pi_1^1) + h(\Pi_2'))$ and $(|C|, h(\Pi_1^2) + h(\Pi_2''))$ of the two new cuts are strictly lower than the rank $(|C|, 1 + max(h(\Pi_1^1), h(\Pi_1^2)) + h(\Pi_2))$ of the original cut.

Case $p_1n_2.r_2$: $C:z$ is only principal in s_1, r is a rule with two premises and active parts Γ', Δ' in the first premiss and Γ'', Δ'' in the second one. We apply Lemma 29 twice on Π_1 to get Π_1' and Π_1''. Similar to Case $n_1.r_2$.

$$\dfrac{\overset{\Pi_1}{\Gamma \vdash \Delta, C:z} \qquad \dfrac{\overset{\Pi_2^1}{C:z, \Gamma, \Gamma' \vdash \Delta, \Delta'} \qquad \overset{\Pi_2^2}{C:z, \Gamma, \Gamma'' \vdash \Delta, \Delta''}}{C:z, \Gamma \vdash \Delta}\,\mathrm{r}}{\Gamma \vdash \Delta}\,\mathrm{cut} \quad \rightsquigarrow$$

$$\dfrac{\dfrac{\overset{\Pi_1'}{\Gamma, \Gamma' \vdash \Delta, \Delta', C:z} \qquad \overset{\Pi_2^1}{C:z, \Gamma, \Gamma' \vdash \Delta, \Delta'}}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\,\mathrm{cut} \qquad \dfrac{\overset{\Pi_1''}{\Gamma, \Gamma'' \vdash \Delta, \Delta'', C:z} \qquad \overset{\Pi_2^2}{C:z, \Gamma, \Gamma'' \vdash \Delta, \Delta''}}{\Gamma, \Gamma'' \vdash \Delta, \Delta''}\,\mathrm{cut}}{\Gamma \vdash \Delta}\,\mathrm{r}$$

The ranks $(|C|, h(\Pi_1') + h(\Pi_2^1))$ and $(|C|, h(\Pi_1'') + h(\Pi_2^2))$ of the two new cuts are strictly lower than the rank $(|C|, h(\Pi_1) + max(h(\Pi_2^1), h(\Pi_2^2)) + 1)$ of the original cut.

Case $p_1.\wedge_R p_2.\wedge_L$: $C:z$ is principal in both s_1 and s_2, C has the form $A \wedge B$.

$$\dfrac{\dfrac{\overset{\Pi_1^1}{\Gamma \vdash \Delta, A \wedge B:z, A:z} \qquad \overset{\Pi_1^2}{\Gamma \vdash \Delta, A \wedge B:z, B:z}}{\Gamma \vdash \Delta, A \wedge B:z}\,\wedge_R \qquad \dfrac{\overset{\Pi_2}{A \wedge B:z, A:z, B:z, \Gamma \vdash \Delta}}{A \wedge B:z, \Gamma \vdash \Delta}\,\wedge_L}{\Gamma \vdash \Delta}\,\mathrm{cut}$$

We use three cuts on $A \wedge B:z$ of strictly lower cut height to get the following proofs:

$$\Pi_3 \left\{ \dfrac{\overset{\Pi_1' \text{ from Lemma 29}}{A:z, B:z, \Gamma \vdash \Delta, A \wedge B:z}\,\wedge_R \qquad \overset{\Pi_2}{A \wedge B:z, A:z, B:z, \Gamma \vdash \Delta}}{A:z, B:z, \Gamma \vdash \Delta}\,\mathrm{cut} \right.$$

$$\Pi_4 \left\{ \dfrac{\begin{array}{c}\Pi_1^2\\ \hdashline \Gamma\vdash\Delta, B:z, A\wedge B:z\end{array} \qquad \dfrac{\begin{array}{c}\Pi_2' \text{ from Lemma 29}\\ \hdashline A\wedge B:z, A:z, B:z, \Gamma\vdash\Delta, B:z\end{array}}{A\wedge B:z, \Gamma\vdash\Delta, B:z}\,\wedge_L}{\Gamma\vdash\Delta, B:z}\,\text{cut}\right.$$

$$\Pi_5 \left\{ \dfrac{\begin{array}{c}\Pi_1^1\\ \hdashline \Gamma\vdash\Delta, A:z, A\wedge B:z\end{array} \qquad \dfrac{\begin{array}{c}\Pi_2'' \text{ from Lemma 29}\\ \hdashline A\wedge B:z, A:z, B:z, \Gamma\vdash\Delta, A:z\end{array}}{A\wedge B:z, \Gamma\vdash\Delta, A:z}\,\wedge_L}{\Gamma\vdash\Delta, A:z}\,\text{cut}\right.$$

We construct the following proof using two cuts on strictly smaller formulas:

$$\dfrac{\begin{array}{c}\Pi_5\\ \hdashline \Gamma\vdash\Delta, A:z\end{array} \qquad \dfrac{\begin{array}{c}\Pi_4' \text{ from Lemma 29}\\ \hdashline A:z, \Gamma\vdash\Delta, B:z\end{array} \qquad \begin{array}{c}\Pi_3\\ \hdashline A:z, B:z, \Gamma\vdash\Delta\end{array}}{A:z, \Gamma\vdash\Delta}\,\text{cut}}{\Gamma\vdash\Delta}\,\text{cut}$$

Case $p_1.\supset_L p_2.\supset_R$: $C:z$ is principal in both s_1 and s_2, C has the form $A \supset B$.

$$\dfrac{\dfrac{\begin{array}{c}\Pi_1\\ \hdashline A:a, \Gamma\vdash\Delta, B:z\cup a\end{array}}{\Gamma\vdash\Delta, A\supset B:z}\,\supset_R \qquad \dfrac{\begin{array}{c}\Pi_2^1\\ \hdashline A\supset B:z, \Gamma\vdash\Delta, A:x\end{array} \quad \begin{array}{c}\Pi_2^2\\ \hdashline A\supset B:z, B:z\cup x, \Gamma\vdash\Delta\end{array}}{A\supset B:z, \Gamma\vdash\Delta}\,\supset_L}{\Gamma\vdash\Delta}\,\text{cut}$$

We first apply Lemma 28 on Π_1 to replace a with x.

$$\begin{array}{c}\Pi_3\\ \hdashline A:x, \Gamma\vdash\Delta, B:z\cup x\end{array}$$

We then apply Lemma 29 on Π_1 to get Π_1':

$$\begin{array}{c}\Pi_1'\\ \hdashline A:a, \Gamma\vdash\Delta, B:z\cup a, A:x\end{array}$$

We combine Π_1' and Π_2^1 to get the following proof with a cut of strictly lower cut height:

$$\Pi_4 \left\{ \dfrac{\dfrac{\begin{array}{c}\Pi_1'\\ \hdashline A:a, \Gamma\vdash\Delta, B:z\cup a, A:x\end{array}}{\Gamma\vdash\Delta, A:x, A\supset B:z}\,\supset_R \qquad \begin{array}{c}\Pi_2^1\\ \hdashline A\supset B:z, \Gamma\vdash\Delta, A:x\end{array}}{\Gamma\vdash\Delta, A:x}\,\text{cut}\right.$$

Applying Lemma 29 on Π_4 we get Π_4':

$$\begin{array}{c}\Pi_4'\\ \hdashline \Gamma\vdash\Delta, B:z\cup x, A:x\end{array}$$

Since $B:z\cup x$ occurs on the left-hand side of the conclusion of Π_2^2, $z\cup x$ necessarily is a sublabel of some label in Δ. We can therefore apply Lemma 29 on Π_1 to get Π_1'':

$$\begin{array}{c}\Pi_1''\\ \hdashline A:a, \Gamma, B:z\cup x\vdash\Delta, B:z\cup a\end{array}$$

We combine Π_1'' and Π_2^2 to get the following proof with a cut of strictly lower cut height:

$$\Pi_5 \left\{ \dfrac{\dfrac{\dfrac{\Pi_1''}{A:a,\Gamma,B:z\cup x\vdash\Delta,B:z\cup a}}{\Gamma,B:z\cup x\vdash\Delta,A\supset B:z}\supset_R \quad \dfrac{\Pi_2^2}{A\supset B:z,\Gamma,B:z\cup x\vdash\Delta}}{\Gamma,B:z\cup x\vdash\Delta}\text{cut} \right.$$

We finally cut on strictly smaller formulas:

$$\dfrac{\dfrac{\dfrac{\Pi_4'}{\Gamma\vdash\Delta,B:z\cup x,A:x} \quad \dfrac{\Pi_3}{A:x,\Gamma\vdash\Delta,B:z\cup x}}{\Gamma\vdash\Delta,B:z\cup x}\text{cut} \quad \dfrac{\Pi_5}{B:z\cup x,\Gamma\vdash\Delta}}{\Gamma\vdash\Delta}\text{cut}$$

◀

New Minimal Linear Inferences in Boolean Logic Independent of Switch and Medial

Anupam Das
University of Birmingham, UK

Alex A. Rice
University of Cambridge, UK

Abstract

A *linear inference* is a valid inequality of Boolean algebra in which each variable occurs at most once on each side. Equivalently, it is a linear rewrite rule on Boolean terms that constitutes a valid implication. Linear inferences have played a significant role in structural proof theory, in particular in models of *substructural logics* and in normalisation arguments for *deep inference* proof systems.

Systems of linear logic and, later, deep inference are founded upon two particular linear inferences, *switch* : $x \wedge (y \vee z) \to (x \wedge y) \vee z$, and *medial* : $(w \wedge x) \vee (y \wedge z) \to (w \vee y) \wedge (x \vee z)$. It is well-known that these two are not enough to derive all linear inferences (even modulo all valid linear equations), but beyond this little more is known about the structure of linear inferences in general. In particular despite recurring attention in the literature, the smallest linear inference not derivable under switch and medial ("switch-medial-independent") was not previously known.

In this work we leverage recently developed *graphical* representations of linear formulae to build an implementation that is capable of more efficiently searching for switch-medial-independent inferences. We use it to find two "minimal" 8-variable independent inferences and also prove that no smaller ones exist; in contrast, a previous approach based directly on formulae reached computational limits already at 7 variables. One of these new inferences derives some previously found independent linear inferences. The other exhibits structure seemingly beyond the scope of previous approaches we are aware of; in particular, its existence contradicts a conjecture of Das and Strassburger.

2012 ACM Subject Classification Theory of computation → Equational logic and rewriting; Theory of computation → Proof theory; Theory of computation → Linear logic

Keywords and phrases rewriting, linear inference, proof theory, linear logic, implementation

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.14

Supplementary Material An associated implementation can be found here:
Software (Source Code): `https://github.com/alexarice/lin_inf` [25]
archived at `swh:1:dir:47d6487bfda3d3848d7289f3b5cfae1824e2ae78`

Funding This work was supported by a UKRI Future Leaders Fellowship, *Structure vs. Invariants in Proofs*, project reference MR/S035540/1. Alex Rice acknowledges funding from the Royal Society.

Acknowledgements The authors would like to thank Lutz Strassburger, Ross Horne and Matteo Acclavio for several interesting discussions surrounding this work. We are also grateful to the anonymous reviewers for their valuable feedback and suggestions.

1 Introduction

A *linear inference* is a valid implication $\varphi \to \psi$ of Boolean logic, where φ and ψ are *linear*, i.e. each variable occurs at most once in each of φ and ψ. Such implications have played a crucial role in many areas of structural proof theory. For instance the inference *switch*,

$$\mathsf{s} \;:\; x \wedge (y \vee z) \;\to\; (x \wedge y) \vee z$$

governs the logical behaviour of the *multiplicative* connectives $⅋$ and $\otimes$ of linear logic [16], and similarly the inference *medial*,

$$\mathsf{m} \ : \ (w \wedge x) \vee (y \wedge z) \ \rightarrow \ (w \vee y) \wedge (x \vee z)$$

together with the structural rules *weakening* and *contraction*, governs the logical behaviour of the *additive* connectives $\oplus$ and & [26, 27]. Both of these inferences are fundamental to *deep inference* proof theory, in particular allowing weakening and contraction to be reduced to atomic form [6, 5], thereby admitting elegant "geometric" proof normalisation procedures based on *atomic flows* [18, 19]. One particular feature of these normalisation procedures is that they are robust under the addition of further linear inferences to the system, thanks to the atomisation of structural steps.

On the other hand the set of *all* linear inferences L plays an essential role in certain models of linear logic and related substructural logics. In particular, the multiplicative fragment of Blass' *game semantics* model of linear logic validates *just* the linear inferences (there called "binary tautologies") [3], and this coincides too with the multiplicative fragment of Japaridze's *computability logic*, cf., e.g., [20]. From a complexity theoretic point of view, the set L is sufficiently rich to encode all of Boolean logic: it is **coNP**-complete [30, 15].

It was recently shown by one of the authors, together with Strassburger, that, despite its significance, L admits no feasible[1] axiomatisation by linear inferences unless **coNP** = **NP** [14, 15], resolving a long-standing open problem of Blass and Japaridze for their respective logics (see, e.g., [21]). From a Boolean algebra point of view, this means that the class of linear Boolean inequalities has no feasible basis (unless **coNP** = **NP**). From a proof theoretic point of view this means that any propositional proof system (in the Cook-Reckhow sense [8, 9], see also [22]) must necessarily admit some "structural" behaviour, even when restricted to proving only linear inferences (unless **coNP** = **NP**).

An immediate consequence of this result is that s and m above do not suffice to generate all linear inferences (unless **coNP** = **NP**), even modulo all valid linear equations.[2] In fact, this was known before the aforementioned result, due to the identification of an explicit 36 variable inference in [30].[3] Already in that work the question was posed whether such an inference was minimal, and since then the identification of a minimal $\{\mathsf{s}, \mathsf{m}\}$-independent linear inference has been a recurring theme in the literature of this area.

It has been verified in [11] that a minimal $\{\mathsf{s}, \mathsf{m}\}$-independent linear inference must be "non-trivial", as long as we admit all true linear equations. Intuitively, "non-triviality" rules out pathological inferences such as $x \wedge y \rightarrow x \vee y$ or $x \wedge (y \vee z) \rightarrow x \vee (y \wedge z)$. For these inferences the variable, say, y is, in a sense, redundant; it turns out that they may be derived in $\{\mathsf{s}, \mathsf{m}\}$, modulo linear equations, from a smaller non-trivial "core". We recall these arguments in Section 2.

Furthermore [11] identified a 10 variable linear inference that is not derivable by switch and medial (even under linear equations), which Strassburger conjectured was minimal [29]. Around the same time Šipraga attempted a computational approach, searching for independent linear inferences by brute force [31]. However, computational limits were reached already at 7 variables. In particular, every linear inference of up to 6 variables is already derivable by switch and medial, modulo linear equations; due to the aforementioned 10 variable inference, any minimal independent linear inference must have size 7,8,9, or 10.

---

[1] By "feasible", in this work, we always mean polynomial-time computable. This is a natural condition arising from proof theory [8, 9], and is also required for the result to be meaningful: it prevents us just taking the entire set L as an axiomatisation.

[2] The valid linear equations are just associativity, commutativity, and unit laws, cf. [14, 15].

[3] Strassburger refers to the inference as a "balanced tautology", but like the "binary tautologies" of Blass and Japaridze, these are equivalent to linear inferences. In particular we recast Strassburger's example as a bona fide linear inference in Section 3.1.

Since 2013 there have been significant advances in the area, in particular through the proliferation of *graph-theoretic* tools. Indeed, the interplay between formulae and graphs was heavily exploited for the aforementioned result of [14, 15]. Since then, multiple works have emerged in the world of linear proof theory that treat these graphs as "first class citizens", comprising a now active area of research [23, 2, 1, 7].

Contribution

In this work we revisit the question of minimal $\{\mathsf{s},\mathsf{m}\}$-independent linear inferences by exploiting the aforementioned recent graph theoretic techniques. Such an approach vastly reduces the computational resources necessary and, in particular, we are able to provide a conclusive result: the smallest $\{\mathsf{s},\mathsf{m}\}$-independent linear inference has size 8. In fact there are two minimal such ones:[4]

$$\begin{array}{rl} & (z \vee (w \wedge w')) \wedge ((x \wedge x') \vee ((y \vee y') \wedge z')) \\ \to & (z \wedge (x \vee y)) \vee ((w \vee y') \wedge ((w' \wedge x') \vee z')) \end{array} \tag{1}$$

$$\begin{array}{rl} & ((w \wedge w') \vee (x \wedge x')) \wedge ((y \wedge y') \vee (z \wedge z')) \\ \to & (w \wedge y) \vee ((x \vee (w' \wedge z')) \wedge ((x' \wedge y') \vee z)) \end{array} \tag{2}$$

We dedicate some discussion to each of these separately in Section 3.2, and include a manual verification of their soundness and $\{\mathsf{s},\mathsf{m}\}$-independence in Appendix A, as a sanity check.

Our main contribution is an implementation that checks inference for $\{\mathsf{s},\mathsf{m}\}$-derivability, which was able to confirm that all 7 variable linear inference are derivable from switch and medial. In fact we found (1) independently of the implementation presented in this paper.[5] Ultimately, we improved the implementation to run on inferences of size 8 too, and our inference (1) was duly found, as well as (2) above and its dual. One highlight of this find is that it exhibits a peculiar structural property that *refutes* Conjecture 7.9 from [15], as we explain in Section 3.2.2.

Our implementation [25] is split into a *library* and an *executable*, where the executable implements our search algorithm described in Section 5.2, and the library contains foundations for working with linear inferences using the graph theoretic techniques presented in Section 4. These are written in Rust and designed to be relatively fast while maintaining readability. Our intention is that this could form a reusable base for future investigations in the area, both for linear formulae and for the recent linear graph theoretic settings of [23, 2, 1, 7].

2 Preliminaries

Throughout this paper we shall work with a countably infinite set of **variables**, written x, y, z etc. A **linear formula** on a (finite) set of variables $\mathcal{V}$ is defined recursively as follows:

- $\top$ and $\bot$ are linear formulae on $\varnothing$, the empty set of variables (called **units** or **constants**).
- x and $\neg x$ are linear formulae on $\{x\}$, for each variable x.[6]
- If φ is a linear formula on $\mathcal{V}_1$ and ψ is a linear formula on $\mathcal{V}_2$, with $\mathcal{V}_1 \cap \mathcal{V}_2 = \varnothing$, then $\varphi \vee \psi$ and $\varphi \wedge \psi$ are linear formulae on $\mathcal{V}_1 \cup \mathcal{V}_2$.

[4] Minimal with respect to inter-derivability; unique up to associativity, commutativity, renaming of variables and De Morgan duality.

[5] These two developments were respectively communicated via blog posts [24] and [13].

[6] Note that the restriction of negation to only variables does not compromise expressivity, since the De Morgan laws preserve linearity on a set of variables.

A linear formula that does not contain $\top$ or $\bot$ is **constant-free**. A linear formula with no negated variables (i.e. formulas of form $\neg x$) is **negation-free**. Later in the paper, we will be able to restrict our search to inferences between constant-free negation-free formulae.

In what follows, we shall omit explicit consideration of variable sets, assuming that they are disjoint whenever required by the notation being used.

A relation $\sim$ on linear formulae is **closed under contexts** if for all φ, ψ, χ, we have:

$$\begin{array}{ll} \varphi \sim \psi \implies \varphi \wedge \chi \sim \psi \wedge \chi & \varphi \sim \psi \implies \varphi \vee \chi \sim \psi \vee \chi \\ \varphi \sim \psi \implies \chi \wedge \varphi \sim \chi \wedge \psi & \varphi \sim \psi \implies \chi \vee \varphi \sim \chi \vee \psi \end{array}$$

An equivalence relation (on linear formulae) that is closed under contexts is called a (linear) **congruence**.

▶ **Definition 1** (Linear equations). *Let $\sim_{\mathsf{ac}}$ be the smallest congruence satisfying,*

$$\begin{array}{ll} \varphi \vee \psi \sim_{\mathsf{ac}} \psi \vee \varphi & \varphi \wedge (\psi \wedge \chi) \sim_{\mathsf{ac}} (\varphi \wedge \psi) \wedge \chi \\ \varphi \wedge \psi \sim_{\mathsf{ac}} \psi \wedge \varphi & \varphi \vee (\psi \vee \chi) \sim_{\mathsf{ac}} (\varphi \vee \psi) \vee \chi \end{array}$$

$\sim_{\mathsf{u}}$ is the smallest congruence satisfying:

$$\begin{array}{llll} \varphi \wedge \top \sim_{\mathsf{u}} \varphi & \varphi \vee \bot \sim_{\mathsf{u}} \varphi & \top \wedge \varphi \sim_{\mathsf{u}} \varphi & \bot \vee \varphi \sim_{\mathsf{u}} \varphi \\ \varphi \wedge \bot \sim_{\mathsf{u}} \bot & \varphi \vee \top \sim_{\mathsf{u}} \top & \bot \wedge \varphi \sim_{\mathsf{u}} \bot & \top \vee \varphi \sim_{\mathsf{u}} \top \end{array} \tag{3}$$

$\sim_{\mathsf{acu}}$ is the smallest congruence containing both $\sim_{\mathsf{ac}}$ and $\sim_{\mathsf{u}}$.

Note that we can have $\varphi \sim_{\mathsf{u}} \psi$ even when φ and ψ have different sets of variables. Moreover, $\sim_{\mathsf{u}}$ generates a unique normal form of linear formulae by maximally eliminating constants:

▶ **Proposition 2** (Folklore, e.g. [10]). *Every formula is $\sim_{\mathsf{u}}$-equivalent to a unique constant-free formula, or is equivalent to $\bot$ or $\top$.*

▶ Remark 3 (On logical equivalence). Clearly, if $\varphi \sim_{\mathsf{acu}} \psi$ then φ and ψ are logically equivalent. In fact, for linear formulae, we also have a converse: two linear formulae φ and ψ are logically equivalent if and only if $\varphi \sim_{\mathsf{acu}} \psi$ [14, 15]. This property follows from Proposition 2 above, the results of Section 2.2, and the graphical representation of linear formulae and their semantics in Section 4.

2.1 Linear inferences

A **linear inference** is just a valid implication $\varphi \to \psi$ (with respect to usual Boolean semantics) where φ and ψ are linear formulae. The left-hand side (LHS) and right-hand side (RHS) of a linear inference, generally speaking, need not be linear formulae on the same variables. Nonetheless we shall occasionally refer to linear inferences "on $\mathcal{V}$" or "on n variables", assuming that the LHS and RHS are both linear formulae on some fixed $\mathcal{V}$ with $|\mathcal{V}| = n$.

There are two linear inferences we shall particularly focus on, due to their prevalence in structural proof theory. **Switch** is the following inference on 3 variables,

$$\mathsf{s} : x \wedge (y \vee z) \to (x \wedge y) \vee z \tag{4}$$

and **medial** is the following inference on 4 variables:

$$\mathsf{m} : (w \wedge x) \vee (y \wedge z) \to (w \vee y) \wedge (x \vee z) \tag{5}$$

We may compose switch and medial (and more generally an arbitrary set of linear inferences) to form new linear inferences by construing them as *term rewriting rules*. More generally, we will consider rewriting derivations modulo the equivalence relations $\sim_{\mathsf{ac}}$ and $\sim_{\mathsf{acu}}$ we introduced earlier. In the latter case, as previously mentioned, the underlying set of variables may change during a derivation, though Proposition 2 will later allow us to work with some fixed set of variables throughout $\{\mathsf{s}, \mathsf{m}\}$ derivations.

▶ **Definition 4** (Rewriting). *We write $\to_{\mathsf{s}}$ and $\to_{\mathsf{m}}$ for the term rewrite systems generated by* (4) *and* (5) *respectively. I.e. $\to_{\mathsf{s}}$ and $\to_{\mathsf{m}}$ are the smallest relations satisfying* (4) *and* (5)*, respectively, closed under substitution and contexts. Write $\varphi \rightsquigarrow_{\mathsf{m}} \psi$ if there are φ', ψ' s.t. $\varphi \sim_{\mathsf{ac}} \varphi' \to_{\mathsf{m}} \psi' \sim_{\mathsf{ac}} \psi$, and $\varphi \rightsquigarrow_{\mathsf{mu}} \psi$ for the same with $\sim_{\mathsf{ac}}$ replaced by $\sim_{\mathsf{acu}}$. Define $\rightsquigarrow_{\mathsf{s}}$, $\rightsquigarrow_{\mathsf{su}}$, $\rightsquigarrow_{\mathsf{ms}}$, $\rightsquigarrow_{\mathsf{msu}}$ similarly (in particular, $\rightsquigarrow_{\mathsf{ms}} = \rightsquigarrow_{\mathsf{m}} \cup \rightsquigarrow_{\mathsf{s}}$).*

We write $\overset{*}{\rightsquigarrow}_{\mathsf{ms}}$ for the reflexive transitive closure of $\rightsquigarrow_{\mathsf{ms}}$, and say $\varphi \to \psi$ is **$\{\mathsf{s}, \mathsf{m}\}$-*derivable*** *if $\varphi \overset{*}{\rightsquigarrow}_{\mathsf{ms}} \psi$. We may similarly write $\varphi \overset{*}{\rightsquigarrow}_{\mathsf{s}} \psi$ (or $\varphi \overset{*}{\rightsquigarrow}_{\mathsf{m}} \psi$), saying $\varphi \to \psi$ is $\{\mathsf{s}\}$-derivable (resp., $\{\mathsf{m}\}$-derivable), and similarly for other sets of linear inferences.*

Finally, we also write $\overset{*}{\rightsquigarrow}_{\mathsf{msu}}$ for the reflexive transitive closure of $\rightsquigarrow_{\mathsf{msu}}$, and say that $\varphi \to \psi$ is **$\{\mathsf{s}, \mathsf{m}\}$-*derivable with units*** *if $\varphi \overset{*}{\rightsquigarrow}_{\mathsf{msu}} \psi$. Similarly for $\overset{*}{\rightsquigarrow}_{\mathsf{su}}$, $\overset{*}{\rightsquigarrow}_{\mathsf{mu}}$ and other sets of linear inferences.*

Clearly, s and m are *valid*, so any derivation $\varphi \overset{*}{\rightsquigarrow}_{\mathsf{msu}} \psi$ comprises a linear inference.

▶ **Example 5** ("Mix"). Units can help us derive even constant-free linear inferences. For instance, **mix**: $\varphi \wedge \psi \to \varphi \vee \psi$ is $\{\mathsf{s}, \mathsf{m}\}$-derivable with units:

$$\varphi \wedge \psi \sim_{\mathsf{acu}} \varphi \wedge (\bot \vee \psi) \to_{\mathsf{s}} (\varphi \wedge \bot) \vee \psi \sim_{\mathsf{acu}} \psi \wedge (\top \vee \varphi) \to_{\mathsf{s}} (\psi \wedge \top) \vee \varphi \sim_{\mathsf{acu}} \varphi \vee \psi$$

Note that mix is not derivable without using instances of $\sim_{\mathsf{u}}$.

▶ **Example 6** (Weakening and duality). By setting $\varphi = \top$ and $\psi = \bot$ in Example 5, we have:

$$\bot \sim_{\mathsf{acu}} \top \wedge \bot \overset{*}{\rightsquigarrow}_{\mathsf{msu}} \top \vee \bot \sim_{\mathsf{acu}} \top$$

Using this we may $\{\mathsf{s}, \mathsf{m}\}$-derive **weakening**, $\varphi \to \varphi \vee \chi$, with units as follows:

$$\varphi \sim_{\mathsf{acu}} \varphi \vee (\bot \wedge \chi) \overset{*}{\rightsquigarrow}_{\mathsf{msu}} \varphi \vee (\top \wedge \chi) \sim_{\mathsf{acu}} \varphi \vee \chi$$

Notice that $\overset{*}{\rightsquigarrow}_{\mathsf{msu}}$ is closed under De Morgan duality: If $\varphi \overset{*}{\rightsquigarrow}_{\mathsf{msu}} \chi$ and $\bar{\varphi}$ and $\bar{\chi}$ are obtained from φ and χ, respectively, by flipping each $\vee$ to a $\wedge$ and vice versa, then $\bar{\chi} \overset{*}{\rightsquigarrow}_{\mathsf{msu}} \bar{\varphi}$. This follows by direct inspection of s, m and each clause of $\sim_{\mathsf{acu}}$; indeed the same property holds for $\overset{*}{\rightsquigarrow}_{\mathsf{ms}}$ by the same reasoning. As a result, we also have that **coweakening**, $\varphi \wedge \chi \to \varphi$, is $\{\mathsf{s}, \mathsf{m}\}$-derivable with units.

We are now able to state the main theorem of this paper:

▶ **Theorem 7.** *Suppose φ is a linear formula over $\mathcal{V}_1$ and ψ is a linear formula over $\mathcal{V}_2$ and $\mathsf{r} : \varphi \to \psi$ is a linear inference. Then if $|\mathcal{V}_1 \cap \mathcal{V}_2| \leq 7$ we have that $\varphi \overset{*}{\rightsquigarrow}_{\mathsf{msu}} \psi$.*

Furthermore, there is a valid linear inference $\varphi \to \psi$ on 8 variables with $\varphi \not\overset{*}{\rightsquigarrow}_{\mathsf{msu}} \psi$, so 7 is maximal with the property above.

2.2 Trivial inferences

In order to state Theorem 7 above in its most general form, we have allowed linear formulae to include constants and negation, and linear inferences to be between formulae with different variable sets. However it turns out that we may proceed to prove Theorem 7, without loss of generality, by working with constant-free, negation-free formulae on some fixed set of variables, as was already shown in [11]. This is done by defining the notion of a *trivial* inference, whose $\{\mathsf{s}, \mathsf{m}\}$-derivability, with units, may be reduced to that of a smaller non-trivial inference.

▶ **Definition 8.** *An inference $\varphi \to \psi$ is **trivial at a variable** x if $\varphi[\top/x] \to \psi[\bot/x]$ is again a valid inference. An inference is **trivial** if it is trivial at one of its variables.*

▶ **Example 9.** The mix inference from Example 5, $x \wedge y \to x \vee y$, is trivial at x and trivial at y. Note, however, that it is not trivial at x and y "at the same time", in the sense that the simultaneous substition of $\bot$ for x and y in the LHS and $\top$ for x and y in the RHS does not result in a valid implication. In contrast, the linear inference $w \wedge (x \vee y) \to w \vee (x \wedge y)$ from [11] is, indeed, trivial at x and y "at the same time".

Neither switch nor medial are trivial.

▶ Remark 10 (Global vs local triviality). Note that triviality is closed under composition by linear inferences: if $\varphi \to \psi$ is trivial at x and $\psi \to \chi$ is valid, then $\varphi \to \chi$ is trivial at x. Similarly for $\chi \to \psi$ if $\chi \to \varphi$ is valid. One pertinent feature is that the converse does not hold: there are "globally" trivial derivations that are nowhere "locally" trivial. For instance consider the following derivation (from [15, Remark 5.6]):

$$w \wedge x \wedge (y \vee z) \rightsquigarrow_{\mathsf{s}} w \wedge ((x \wedge y) \vee z) \rightsquigarrow_{\mathsf{s}} (w \wedge z) \vee (x \wedge y) \rightsquigarrow_{\mathsf{m}} (w \vee x) \wedge (y \vee z)$$

The derived inference is just an instance of mix, from Example 5, on the redex $w \wedge x$, which is trivial. However, no local step is trivial.

To prove (the first half of) Theorem 7, in Section 5 we will actually prove the following apparent weakening of that statement:

▶ **Theorem 11.** *Let $n < 8$. Let φ and ψ be constant-free negation-free linear formulae on n variables and suppose $\varphi \to \psi$ is a non-trivial linear inference. Then $\varphi \overset{*}{\rightsquigarrow}_{\mathsf{ms}} \psi$.*

In fact this statement is no weaker at all, and we will now see how the consideration of triviality allows us to only deal with such special cases without loss of generality.

▶ **Proposition 12** ([11, Theorem 34])**.** *Let φ and ψ be linear formulae on $\mathcal{V}_1$ and $\mathcal{V}_2$, respectively, and let $\mathsf{r} : \varphi \to \psi$ be a linear inference. There is a non-trivial linear inference $\mathsf{r}' : \varphi' \to \psi'$ on some $\mathcal{V}' \subseteq \mathcal{V}_1 \cap \mathcal{V}_2$ such that $\mathsf{r} : \varphi \to \psi$ is $\{\mathsf{s}, \mathsf{m}, \mathsf{r}'\}$-derivable with units.*

Note in particular that, in the statement above, if r' is $\{\mathsf{s}, \mathsf{m}\}$-derivable with units, then so is r. This is also the case for the next result.

▶ **Proposition 13.** *Let $\mathsf{r} : \varphi \to \psi$ be a non-trivial linear inference among variables $\mathcal{V} \neq \varnothing$. Then there is a constant-free negation-free non-trivial linear inference $\mathsf{r}' : \varphi' \to \psi'$ on $\mathcal{V}$ s.t. $\mathsf{r} : \varphi \to \psi$ is $\{\mathsf{s}, \mathsf{m}, \mathsf{r}'\}$-derivable with units.*

Proof. First, note that both φ and ψ must be linear formulae on $\mathcal{V}$, since $\varphi \to \psi$ is non-trivial. For the same reason, no variable can occur positively in φ and negatively in ψ or vice-versa, since $\varphi \to \psi$ is non-trivial, and so any negated variable may be safely replaced by its positive counterpart. From here, we simply set φ' and ψ' to be the constant-free formulae (uniquely) obtained from Proposition 2 by $\sim_{\mathsf{u}}$. Non-triviality of r' follows from that of r by logical equivalence. ◀

▶ **Corollary 14.** *The statement of Theorem 11 implies (the first half of) the statement of Theorem 7.*

Proof. Let r be as in Theorem 7. Let r' be the non-trivial linear inference obtained by Proposition 12 above, and let r'' be the non-trivial constant-free negation-free linear inference thence obtained by Proposition 13. By Theorem 11, r'' is $\{\mathsf{s},\mathsf{m}\}$-derivable and so, by Proposition 12 and Proposition 13, r is also $\{\mathsf{s},\mathsf{m}\}$-derivable with units. ◀

It is clear that if an inference is derivable with switch and medial then it is also derivable with switch, medial, and units. The following proposition, while not necessary for the proof of Corollary 14, allows the the converse in some cases, and is the reason why our search algorithm in Section 5 will only check for $\{\mathsf{s},\mathsf{m}\}$-derivability.

▶ **Proposition 15** (Follows from [11], Lemma 28). *Suppose $\varphi \to \psi$ is a non-trivial constant-free negation-free linear inference that is $\{\mathsf{s},\mathsf{m}\}$-derivable with units. Then $\varphi \to \psi$ is also $\{\mathsf{s},\mathsf{m}\}$-derivable (without units).*

The idea here is to systematically rewrite a derivation with units to one without, line by line under Proposition 13. Crucially, the invariant of non-triviality constrains the contexts in which constants may occur, ensuring that the constant-elimination procedure preserves instances of s or m.

2.3 Minimality of inferences

Let us take a moment to explain the various notions of "inference minimality" that we shall mention in this work.

Size minimality refers simply to the number of variables the inference contains. E.g. when we say that the 8-variable inferences in the next section are size minimal (or "smallest") non-$\{\mathsf{s},\mathsf{m}\}$-derivable with units (or **$\{\mathsf{s},\mathsf{m}\}$-independent** linear inferences, we mean that there are no $\{\mathsf{s},\mathsf{m}\}$-independent linear inferences with fewer variables.

A linear inference $\varphi \to \psi$ is **logically minimal** if there is no $\sim_{\mathsf{acu}}$-distinct interpolating linear formula. I.e. if $\varphi \to \chi$ and $\chi \to \psi$ are linear inferences, then χ is $\sim_{\mathsf{acu}}$-equivalent to φ or ψ (and so, by Remark 3, is logically equivalent to φ or ψ).

Finally, a linear inference $\varphi \to \psi$ is **$\{\mathsf{s},\mathsf{m}\}$-minimal** if there is no formula χ s.t. $\varphi \rightsquigarrow_{\mathsf{ms}} \chi$ or $\chi \rightsquigarrow_{\mathsf{ms}} \psi$ and $\chi \to \psi$ or $\varphi \to \chi$, respectively, is a valid linear inference which is not a logical equivalence.

It is clear from the definitions that any logically minimal inference is also $\{\mathsf{s},\mathsf{m}\}$-minimal, though the converse may not be true. The reason for considering $\{\mathsf{s},\mathsf{m}\}$-minimality is that it is easier to systematically check by hand. In fact, the implementation we give later in Section 5 further verifies that our new 8-variable inferences are logically minimal.

Logical minimality also serves an important purpose for our proof of Theorem 11, as it allows the following reduction, greatly reducing the search space for our implementation, in fact to nearly 1% of its original size for 8 variable inferences:[7]

▶ **Lemma 16.** *Suppose the statement of Theorem 11 holds whenever $\varphi \to \psi$ is logically minimal. Then the statement of Theorem 11 holds (even when $\varphi \to \psi$ is not logically minimal).*

Proof. Suppose we have a non-trivial inference between constant-free negation-free linear inferences $\varphi \to \psi$. Then $\varphi \to \psi$ can be refined into a chain of logically minimal linear inferences $\varphi \to \chi_0 \to \cdots \to \chi_n \to \psi$. All of these must be non-trivial, as triviality of any of them would imply triviality of $\varphi \to \psi$, cf. Remark 10. Therefore if all such inferences are derivable from switch and medial (with units) then so is $\varphi \to \psi$, by transitivity. ◀

[7] $^{5364}/_{514486} \approx 1.04\%$

3 New 8-variable $\{\mathsf{s}, \mathsf{m}\}$-independent linear inferences

In this section we shall present the new 8-variable linear inferences of this work ((1) and (2) from the introduction), and give self-contained arguments for their $\{\mathsf{s}, \mathsf{m}\}$-independence and $\{\mathsf{s}, \mathsf{m}\}$-minimality, as a sort of sanity check for the implementation described in the next section. We shall also briefly discuss some of their structural properties, in reference to previous works in the area. Thanks to the results of the previous section, in particular Proposition 13 and Remark 10, we shall only consider non-trivial constant-free negation-free linear inferences with the same variables in the LHS and RHS. Furthermore, by Proposition 15 we shall only consider $\{\mathsf{s}, \mathsf{m}\}$-derivability (i.e., without units).

3.1 Previous linear inferences

In [30] Strassburger presented a 36-variable inference that is $\{\mathsf{s}, \mathsf{m}\}$-independent, by an encoding of the pigeonhole principle with 4 pigeons and 3 holes. He there referred to it as a "balanced" tautology, but in our setting it is a linear inference that can be written as follows:[8]

$$\bigwedge_{i=1}^{3}\bigwedge_{j=1}^{i}(x_{ij} \vee x'_{ij}) \wedge \bigwedge_{i=1}^{3}\bigwedge_{j=1}^{i}(y_{ij} \vee y'_{ij}) \wedge \bigwedge_{i=1}^{3}\bigwedge_{j=1}^{i}(z_{ij} \vee z'_{ij})$$

$$\to \left[\begin{array}{llllll} & ((x_{11} \vee x_{21} \vee x_{31}) & \wedge & (y_{11} \vee y_{21} \vee y_{31}) & \wedge & (z_{11} \vee z_{21} \vee z_{31})) \\ \vee & ((x'_{11} \vee x_{22} \vee x_{32}) & \wedge & (y'_{11} \vee y_{22} \vee y_{32}) & \wedge & (z'_{11} \vee z_{22} \vee z_{32})) \\ \vee & ((x'_{21} \vee x'_{22} \vee x_{33}) & \wedge & (y'_{21} \vee y'_{22} \vee y_{33}) & \wedge & (z'_{21} \vee z'_{22} \vee z_{33})) \\ \vee & ((x'_{31} \vee x'_{32} \vee x'_{33}) & \wedge & (y'_{31} \vee y'_{32} \vee y'_{33}) & \wedge & (z'_{31} \vee z'_{32} \vee z'_{33})) \end{array}\right]$$

In [11] Das noticed that a more succinct encoding of the pigeonhole principle could be carried out, with only 3 pigeons and 2 holes, resulting in a 10-variable $\{\mathsf{s}, \mathsf{m}\}$-independent linear inference. A variation of that, e.g. as used in [12], is the following:

$$\begin{array}{ll} & (z \vee (w \wedge w')) \wedge (y \vee y') \wedge (u \vee u') \wedge ((x \wedge x') \vee z') \\ \to & (z \wedge (x \vee y)) \vee (u \wedge x') \vee (w' \wedge u') \vee ((w \vee y') \wedge z') \end{array} \tag{6}$$

In fact this is not a $\{\mathsf{s}, \mathsf{m}\}$-minimal inference, but we write this one here for comparison to one of the new 8-variable inferences in the next subsection. It can be checked valid and non-trivial by simply checking all cases, or by use of a solver. We do not give an argument for $\{\mathsf{s}, \mathsf{m}\}$-independence here, but such an argument is similar to the one we give for an 8-variable inference Equation (7), which is given the next subsection.

3.2 The two minimal 8 variable $\{\mathsf{s}, \mathsf{m}\}$-independent linear inferences

Pre-empting Section 5.2, let us explicitly give the two minimal linear inferences found by our algorithm and justify their $\{\mathsf{s}, \mathsf{m}\}$-independence and $\{\mathsf{s}, \mathsf{m}\}$-minimality, as a sort of sanity check for our implementation later. As we will see, they both turn out to be significant in their own right, which is why we take the time to consider them separately.

[8] We write Strassburger's inference by encoding each q_{i1j} as x_{ij}, each q_{i2j} as y_{ij}, each q_{i3j} as z_{ij}, and using "primed" variables instead of duals, with the LHS of the inference being the appropriate instances of excluded middle.

3.2.1 A refinement of the 3-2-pigeonhole-principle

First let us consider the 8 variable linear inference that may be used to derive Equation (6), cf. Appendix A.1 (identical to (1) from the introduction):

$$\begin{array}{rl} & (z \vee (w \wedge w')) \wedge ((x \wedge x') \vee ((y \vee y') \wedge z')) \\ \to & (z \wedge (x \vee y)) \vee ((w \vee y') \wedge ((w' \wedge x') \vee z')) \end{array} \tag{7}$$

Recalling the notion of "duality" from Example 6, let us formally define the **dual** of a linear inference $\varphi \to \chi$ to be the linear inference $\bar{\chi} \to \bar{\varphi}$, where $\bar{\varphi}$ and $\bar{\chi}$ are obtained from φ and χ, respectively, by flipping all $\vee$s to $\wedge$s and vice-versa. Considering linear inferences up to renaming of variables, we have:

▶ **Observation 17.** (7) *is self-dual.*

Indeed, the formula structure of the RHS is clearly the dual of that of the LHS, and the mapping from a variable in the LHS to the variable at the same position in the RHS is, in fact, an involution. I.e., u is mapped to itself; v is mapped to y which in turn is mapped to v; v' is mapped to w which is in turn mapped to v'; x is mapped to y' which in turn is mapped to x; and z is mapped to itself. Validity may be routinely checked by any solver, but we give a case analyis of assignments in Appendix A.2.

We may also establish $\{\mathsf{s}, \mathsf{m}\}$-independence and $\{\mathsf{s}, \mathsf{m}\}$-minimality by checking all applications of s or m to the LHS (note that we do not need to check the RHS, by Observation 17 above). This analysis is given explicitly in Appendix A.4.

3.2.2 A counterexample to a conjecture of Das and Strassburger

Finally, our search algorithm found a completely new linear inference (identical to (2) from the introduction):

$$\begin{array}{rl} & ((w \wedge w') \vee (x \wedge x')) \wedge ((y \wedge y') \vee (z \wedge z')) \\ \to & (w \wedge y) \vee ((x \vee (w' \wedge z')) \wedge ((x' \wedge y') \vee z)) \end{array} \tag{8}$$

Again, validity is routine, but a case analysis is given in Appendix A.3. We may establish $\{\mathsf{s}, \mathsf{m}\}$-independence and $\{\mathsf{s}, \mathsf{m}\}$-minimality again by checking all possible rule applications. This analysis is given in Appendix A.5.

This new inference exhibits a rather interesting property, which we shall frame in terms of the following notion, since it will be used in the next section:

▶ **Definition 18.** *Let φ be a linear formula on a variable set $\mathcal{V}$. For distinct $x, y \in \mathcal{V}$, the **least common connective** (lcc) of x and y in φ is the connective $\vee$ or $\wedge$ at the root of the smallest subformula of φ containing both x and y.*

Note that, in the inference (8) above, the lcc of w' and x' changes from $\vee$ to $\wedge$, but the lcc of y and y' changes from $\wedge$ to $\vee$. No such example of a minimal linear inference exhibiting both of these properties was known before; switch, medial and all of the linear inferences of this section either preserve $\vee$-lccs or preserve $\wedge$-lccs. In fact, Das and Strassburger showed that any valid linear inference preserving $\wedge$-lccs is already derivable by medial [15, Theorem 7.5], and further conjectured that there was no minimal inference that preserves neither $\wedge$-lccs nor $\vee$-lccs. Naturally, our new inference is a counterexample to that:

▶ **Theorem 19.** *Conjecture 7.9 from [15] is false.*

4 A graph-theoretic presentation of linear inferences

A significant cause of algorithmic complexity when searching for linear inferences is the multitude of formulae equivalent modulo associativity and commutativity ($\sim_{\mathsf{ac}}$). For example, for 7 variables, there are 42577920 formulae (ignoring units), yet only 78416 equivalence classes. Under Remark 3 it would be ideal if we could deal with $\sim_{\mathsf{ac}}$-equivalence classes directly, realising logical and syntactic notions on them in a natural way. This is precisely what is accomplished by the graph-theoretic notion of a *relation web*, cf. [17, 28, 14, 15].

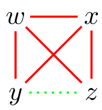
Throughout this section we work only with constant-free negation-free linear formulae, cf. Theorem 11. Recall the notion of *least common connective* (lcc) from Definition 18.

▶ **Definition 20.** *Let φ be a linear formula on a variable set $\mathcal{V}$. The* ***relation web*** *(or simply* ***web****) of φ, written $\mathcal{W}(\varphi)$, is a simple undirected graph with:*

- *The set of nodes of $\mathcal{W}(\varphi)$ is just $\mathcal{V}$, i.e. the variables occurring in φ.*
- *For $x, y \in \mathcal{V}$, there is an edge between x and y in $\mathcal{W}(\varphi)$ if the lcc of x and y in φ is $\wedge$.*

When we draw graphs, we will draw a solid red line x —— y if there is an edge between x and y, and a green dotted line x ⋯⋯ y otherwise.

▶ **Example 21.** Let φ be the linear formula $w \wedge (x \wedge (y \vee z))$. $\mathcal{W}(\varphi)$ is the following graph:



Note that linear formulae equivalent up to associativity and commutativity have the same relation web, since $\sim_{\mathsf{ac}}$ does not affect the lccs. For instance, if $\psi = (w \wedge x) \wedge (z \vee y)$, then $\mathcal{W}(\psi)$ is still just the relation web above. In fact, we also have the converse:

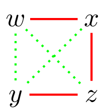
▶ **Proposition 22** (E.g., [15], Proposition 3.5)**.** *Given linear formulae φ and ψ, $\varphi \sim_{\mathsf{ac}} \psi$ if and only if $\mathcal{W}(\varphi) = \mathcal{W}(\psi)$.*

Thus relation webs are natural representations of equivalence classes of linear formulae modulo associativity and commutativity.

It is easy to see that the image of $\mathcal{W}$ is just the *cographs*. A **cograph** is either a single node, or has the form $\mathcal{R}$ —— $\mathcal{S}$ or $\mathcal{R}$ ⋯⋯ $\mathcal{S}$ for cographs $\mathcal{R}$ and $\mathcal{S}$.[9] A **cograph decomposition** of a cograph $\mathcal{R}$ is just a definition tree according to these construction rules (its "cotree"), from which we may easily extract a linear formula with web $\mathcal{R}$. Note from Example 21 that the cograph decomposition of a relation web need not be unique, since formulae equivalent modulo associativity and commutativity have the same relation web.

Cographs admit an elegant *local* characterisation by means of forbidden subgraphs:

▶ **Definition 23.** *P_4 is the following graph:*



*A graph G is P_4-**free** if none of its induced*[10] *subgraphs are isomorphic to P_4.*

---

[9] Formally, $\mathcal{R}$ —— $\mathcal{S}$ has as nodes the disjoint union of the nodes of $\mathcal{R}$ and the nodes of $\mathcal{S}$; edges within the $\mathcal{R}$ component are inherited from $\mathcal{R}$ and similarly for $\mathcal{S}$; there is also an edge between every node in $\mathcal{R}$ and every node in $\mathcal{S}$. $\mathcal{R}$ ⋯⋯ $\mathcal{S}$ is defined similarly, but without the last clause.

[10] An induced subgraph is one whose edges are just those of G restricted to a subset of the nodes.

▶ **Proposition 24** (E.g. [17, 28]). *A graph is a cograph if and only if it is P_4-free. Thus, relation webs are just the P_4-free graphs whose nodes are variables.*

Note, in particular, that this characterisation gives us an easy way to check whether a graph is the web of some formula: just check every 4-tuple of nodes for a P_4. What is more, we may also verify several semantic properties of linear inferences, such as validity and triviality, directly at the level of relation webs:

▶ **Proposition 25** (Follows from Proposition 4.4 and Theorem 4.6 in [15]). *Let φ and ψ be linear formulae on the same set of variables. $\varphi \to \psi$ is a valid linear inference if and only if for every maximal clique P of $\mathcal{W}(\varphi)$, there is some $Q \subseteq P$ such that Q is a maximal clique of $\mathcal{W}(\psi)$.*

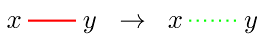
▶ **Proposition 26** (E.g., [15], Proposition 5.7). *Let φ and ψ be linear formulae on the same variables. $\varphi \to \psi$ is a linear inference that is trivial at x if and only if for every maximal clique P of $\mathcal{W}(\varphi)$, there is some $Q \subseteq P \setminus \{x\}$ such that Q is a maximal clique of $\mathcal{W}(\psi)$.*

Note that the criterion for triviality is a strict strengthening of that for validity, as we would expect. For both of the results above, there is a *dual* characterisation in terms of *maximal stable sets* instead of maximal cliques. For instance, the characterisation of validity morally states "whenever φ evaluates to 1, then ψ evaluates to 1". The dual characterisation is that for every maximal stable set Q of $\mathcal{W}(\psi)$ there is a maximal stable set P of $\mathcal{W}(\varphi)$ with $P \subseteq Q$, which morally states "whenever ψ evaluates to 0, then φ evaluates to 0". We will not make use of these dual characterisations in this work.

▶ **Example 27** (Validity of switch and medial, triviality of mix). The switch and medial inferences can be construed as the following "graph rewrite" rules on relation webs, respectively:

It is easy to see that the validity criterion of Proposition 25 holds for each of these rules. For s, the maximal cliques $\{x, y\}$ and $\{x, z\}$ in the LHS are mapped to $\{x, y\}$ and $\{z\}$ in the RHS respectively. For m, the maximal cliques $\{w, x\}$ and $\{y, z\}$ in the LHS are mapped to themselves in the RHS.

Now consider the trivial inference $x \wedge y \to x \vee y$, construed as the graph rewrite rule:



We can easily verify the criterion for triviality at x from Proposition 26 since the only maximal clique on the LHS, $\{x, y\}$ has $\{y\} \subseteq \{x, y\} \setminus \{x\}$ as a maximal clique on the RHS.

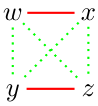
▶ Remark 28. With the results in this section, the notation for inferences between formulae can be equally used for relation webs. For example, for webs $\mathcal{R}$ and $\mathcal{S}$, we can write $\mathcal{R} \rightsquigarrow_{\mathsf{ms}} \mathcal{S}$ is valid to mean that the inference between (any choice of) the underlying linear formulae is an instance of $\rightsquigarrow_{\mathsf{ms}}$, and $\mathcal{R} \overset{*}{\rightsquigarrow}_{\mathsf{ms}} \mathcal{S}$ to mean the there is a derivation from switch and medial between the underlying formulae. Since these relations are invariant under associativity and commutativity, they are independent of the particular cograph decomposition chosen.

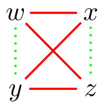
Furthermore, to prove Theorem 11, it is sufficient to show that for all webs $\mathcal{R}$ and $\mathcal{S}$ with size less than 8, if $\mathcal{R} \to \mathcal{S}$ is valid and non-trivial then $\mathcal{R} \overset{*}{\rightsquigarrow}_{\mathsf{ms}} \mathcal{S}$.

The final component needed to be able to work fully with webs is a way to check if a given inference is an instance of switch or medial. Such characterisations exist:

▶ **Proposition 29** ([28, Theorem 5]). *Let $\mathcal{R} \to \mathcal{S}$ represent a constant-free negation-free non-trivial linear inference. Then $\mathcal{R} \to \mathcal{S}$ is derivable from medial if and only if:*

- *Whenever x —— y in $\mathcal{R}$, also x —— y in $\mathcal{S}$.*
- *Whenever $x \cdots\cdots y$ in $\mathcal{R}$ but x —— y in $\mathcal{S}$ there exists w and z such that*



is an induced subgraph of $\mathcal{R}$ and


is an induced subgraph of $\mathcal{S}$.

The second condition can, in fact, be replaced by simply requiring that $\mathcal{R} \to \mathcal{S}$ is valid [15, Theorem 7.5]. A relation web characterisation for switch derivability can also be found in [28, Theorem 6.2], however we do not use it in our implementation.

5 Implementation

As stated in previous sections, Theorem 11 is proved using a computational search. In this section we describe the algorithm used to search for $\{\mathsf{s}, \mathsf{m}\}$-independent inferences, as well as some of the optimisations we employ so that this search finishes in a reasonable time. Many of these optimisations may be of self-contained theoretical interest.

The implementation is written in Rust,[11] which offers a combination of good performance (both in terms of speed and memory management) but also provides a variety of high level abstractions such as algebraic data types. Furthermore, it has built-in support for iterators, allowing the code to be written in a more functional style, and has a built-in testing framework, meaning that sanity checks can be built into the code base. The code is available at [25] and has been split into two parts: a *library* containing types for undirected linear graphs and formulae and some operations on them, and an *executable* which implements the search algorithm using this library as a base.

5.1 Library

The library portion of the implementation defines methods for working with relation webs, as well as the ability to convert formulae to relation webs and vice versa. The majority of the library consists of the `LinGraph` trait, which is an interface for types that can be treated as undirected graphs. This allows us to query the edges between variables as well as perform more involved operations such as checking whether a graph is P_4-free. We may also ask whether a pair of relation webs forms a valid linear inference and check whether the inference is trivial using Propositions 25 and 26.

Storing graphs and relation webs. The library was designed with the intention of storing graphs as compactly as possible. Therefore there are implementations of `LinGraph` which pack the data (a series of bits for whether there exists an edge between each pair of nodes) into various integer types. The implementation is given for unsigned 8 bit, 16 bit, 32 bit (which can store up to 8 variable graphs), 64 bit, and 128 bit integers. Furthermore there is an implementation using vectors (variable length arrays) of Boolean values, which is less memory efficient but can store relation webs of arbitrary size. A further improvement could be to use an external library implementing bit arrays to make a memory efficient, yet infinitely scalable implementation.

[11] `https://www.rust-lang.org/`

Checking an inference between graphs. In order to implement linear inference checking, we use a data type representing maximal cliques of a relation web, which we represent as the trait `MClique`. It is possible to use Rust's inbuilt `HashSet`[12] to do this but, as above, a more memory efficient solution is provided where we store the data in a single integer, with each bit determining whether a node is contained in the clique. For example a maximal clique on an 8 node graph can be encoded into a single byte. While checking for linear inferences and triviality, the main operation on maximal cliques is asking whether one is a subset of the other. This operation can be carried out very quickly using bitwise operations. Lastly we also need a way to generate the maximal cliques of a relation web. This is done using the Bron-Kerbosch algorithm [4], which is fast enough for our purposes (as we are only finding the maximal cliques of relatively small graphs).

Working with isomorphism. There is also code for working with isomorphisms of graphs, which is used in the search algorithm to shrink the search space further. This is implemented as a module where permutations and operations on these permutations are defined, as well as having the ability to apply a permutation to the nodes of a graph, to get a new but isomorphic graph.

Generating all P_4-free graphs. The library also has a function that allows all P_4-free graphs of a certain size to be generated. The naive algorithm for doing this which simply generates all graphs and checks each one for being P_4-free is computationally infeasible for graphs with more than a few variables, as the number of graphs scales superexponentially with the number of variables (for instance there are 2^{21} 7-variable relation webs). Instead, we use a recursive algorithm that generates all P_4-free graphs of size n by first generating the P_4-free graphs of size $n-1$ and then checking all possible extensions of these graphs to see if they are P_4-free. Correctness of this procedure is due to the fact that induced subgraphs of a P_4-free are themselves P_4-free. In fact, a further optimisation is also added: when we check whether the extensions are P_4-free, it is sufficient to only check if subsets of the nodes containing the added node are not isomorphic to P_4, instead of checking every subset.

Sanity checks. Finally, the library also contains some automated tests used as sanity checks on the code, which may be used to check various implementations against each other.

5.2 Search algorithm

The main part of the implementation is a search algorithm to find logically minimal non-trivial inferences between relation webs that are not derivable from switch and medial. The search algorithm functions in multiple phases. After each phase the results are serialised and saved to disk so that the algorithm can be restarted from this point.

Phase 1: generating P_4-free graphs on n nodes. Suppose we are searching for $\{\mathsf{s}, \mathsf{m}\}$-independent linear inferences between webs on n variables. The first phase, as described in the previous section, is to gather all P_4-free graphs with n nodes.

[12] `https://doc.rust-lang.org/std/collections/struct.HashSet.html`

Phase 2: identifying isomorphism classes and canonical representatives. To describe the second phase we need to introduce some new notions. Without loss of generality, we will assume henceforth that the variable set is given by $\mathcal{V} = \{0, \ldots, n-1\}$.

▶ **Definition 30.** *Note that the function $\iota : \{(x, y) \in \mathbb{N} \times \mathbb{N} \mid x < y\} \to \mathbb{N}$ given by $\iota(x, y) = x + \sum_{i<y} i$ is a bijection. Define the* ***numerical representation*** *of a linear graph $\mathcal{R}$, written $N(\mathcal{R})$, to be the natural number whose $\iota(x, y)^{th}$ least significant bit is 1 if and only if $(x, y) \in \mathcal{R}$.*

This is the encoding used to store graph in integers as described in the previous section.

▶ **Definition 31.** *Given a bijection $\rho : \mathcal{V} \to \mathcal{V}$, we write $\rho(\mathcal{R})$ for the graph on $\mathcal{V}$ with edges $(\rho(x), \rho(y))$ for each edge $(x, y) \in \mathcal{R}$. $\mathcal{R}$ and $\mathcal{S}$ are* ***isomorphic*** *if $\mathcal{S} = \rho(\mathcal{R})$, for some bijection $\rho : \mathcal{V} \to \mathcal{V}$, in which case ρ is called an* ***isomorphism*** *from $\mathcal{R}$ to $\mathcal{S}$.*

As isomorphism is an equivalence relation, we can partition the set of P_4-free graphs into isomorphism classes. It can readily be checked that N (from Definition 30) is injective and can therefore be used to induce a strict total ordering on graphs. Say that a relation web is **least** if it is the smallest element in its isomorphism class (with respect to this ordering induced from N).

The second phase of the algorithm is to identify these least relation webs, as well as identify the isomorphism between every relation web and its isomorphic least relation web. It will become clear why this data is needed later on in the section. To obtain this, first the relation webs are sorted (by numerical representation) and then, taking each graph $\mathcal{R}$ in turn, applying every possible permutation to its nodes, and seeing if any result in a smaller web (with respect to N). If none do then we record it as a least relation web (with the identity isomorphism). Otherwise suppose it is isomorphic to $\mathcal{R}'$ with isomorphism ρ where $N(\mathcal{R}') < N(\mathcal{R})$. As we are checking graphs in order, we must already know that $\mathcal{R}'$ is isomorphic to least graph $\mathcal{R}''$ with isomorphism π. Then we can record $\mathcal{R}$ as being isomorphic to $\mathcal{R}''$ with isomorphism $\pi \circ \rho$. This allows us to use the following lemma.

▶ **Lemma 32.** *The statement of Theorem 11 follows from the following: for any valid non-trivial logically minimal inference $\mathcal{R} \to \mathcal{S}$ on $n < 8$ variables, where $\mathcal{R}$ is least, we have $\mathcal{R} \overset{*}{\leadsto}_{\mathsf{ms}} \mathcal{S}$.*

Proof. To show the statement of Theorem 11, let $\mathcal{R}$ and $\mathcal{S}$ be relation webs on n variables and suppose $\mathcal{R} \to \mathcal{S}$ is a non-trivial linear inference (cf. Remark 28). By Lemma 16, we can further assume that $\mathcal{R} \to \mathcal{S}$ is logically minimal. Then let ρ be an isomorphism from $\mathcal{R}$ to $\mathcal{R}'$ least isomorphic to $\mathcal{R}$, and let $\mathcal{S}' = \rho(\mathcal{S})$. Then $\mathcal{R} \overset{*}{\leadsto}_{\mathsf{ms}} \mathcal{S}$ if and only if $\mathcal{R}' \overset{*}{\leadsto}_{\mathsf{ms}} \mathcal{S}'$, as required. ◀

The above lemma allows us to only search inferences from least webs to arbitrary webs. This increases the speed of the search greatly as it turns out there are relatively few least webs. For example, there are 78416 P_4-free graphs with 7 variables with only 180 of them being least (the number of isomorphism classes). Note that we may not similarly restrict the RHS of inferences to least webs. This means we need to know the maximal cliques of every P_4-free graph to determine whether there are inferences between them.

Phase 3: generating all maximal cliques. In phase three we generate all the maximal cliques of the graphs found in phase one and store them so that they do not need to be recomputed every time we check a linear inference. As we can store each maximal clique in a single byte, storing all this data is feasible.

Phase 4: generating "least" linear inferences. With the maximal clique data, phase four of generating a list of all valid linear inferences (from a least web to an arbitrary web) can be easily done by iterating through all possible combinations and checking them using Proposition 25.

Phase 5: checking for non-triviality. Similarly phase five of checking which of these inferences are non-trivial is also simple using Proposition 26. This data is stored in a `HashMap` of sets for quick indexing.

Phase 6: restricting to logically minimal inferences. Phase six is now to restrict our inferences to only those that are logically minimal. Write $\Phi_{\mathcal{R}}$ be the set of webs distinct from $\mathcal{R}$ that $\mathcal{R}$ (non-trivially) implies. We calculate, for a least web $\mathcal{R}$, the set $M_{\mathcal{R}}$ of webs $\mathcal{S}$ with $\mathcal{R} \to \mathcal{S}$ a logically minimal linear inference using the identity:

$$M_{\mathcal{R}} = \Phi_{\mathcal{R}} \setminus \bigcup_{\mathcal{R}' \in \Phi_{\mathcal{R}}} \Phi_{\mathcal{R}'}$$

Note that to calculate this, we need to be able to generate $\Phi_{\mathcal{R}}$ for arbitrary (i.e. not necessarily least) webs. This is where the isomorphism data stored in phase two becomes useful, as if ρ is an isomorphism from $\mathcal{R}$ to $\mathcal{R}'$, with $\mathcal{R}'$ least, we can use,

$$\Phi_{\mathcal{R}} = \{\rho^{-1}(\mathcal{S}) \mid \mathcal{S} \in \Phi_{\mathcal{R}'}\}$$

to generate $\Phi_{\mathcal{R}}$, where we already have $\Phi_{\mathcal{R}'}$. In the implementation, we generate each $\Phi_{\mathcal{R}}$ on the fly (from $\Phi_{\mathcal{R}'}$), though we could have pre-generated all of these, which might provide further speedup for this phase.

Phase 7: checking for switch-medial derivability. The last phase is to check the remaining inferences, of which there are now few enough to feasibly do so. Logically minimal inferences have one further benefit: a logically minimal inference (and in fact any $\{\mathsf{s}, \mathsf{m}\}$-minimal inference) $\varphi \to \psi$ is derivable from switch and medial if and only if it is derivable from a *single* switch or medial step. To check if it is a medial we can use the criterion for medial derivability from Proposition 29.

To check if the inference $\mathcal{R} \to \mathcal{S}$ is a switch, we simply run through all possible cograph decompositions of $\mathcal{R}$ and check if any of the possible switch applications yields $\mathcal{S}$. It would have been possible to use the criterion for switch derivability from [28] (mentioned at the end of Section 4), but running through possible partitions of the nodes of $\mathcal{R}$ was fast enough and easier to implement.

Evaluation and main results. After running all phases on 7 variables, we found that there were 78416 P_4-free graphs of which 180 were least. There were 35110 non-trivial inferences from a least web to an arbitrary web of which 1352 were minimal. Of these minimal inferences, 968 were an instance of switch, 384 were an instance of medial, and there were no other inferences, which completes the proof of Theorem 11.

Furthermore, the algorithm was fast enough to run on 8 variables, where there were 1320064 P_4-free graphs of which 522 were least. There were 514486 non-trivial inferences from a least web to an arbitrary web of which 5364 were minimal, Of these, 3506 were an instance of switch, 1770 were an instance of medial, and there were 88 other inferences. After quotienting out by isomorphism (as restricting to inferences from least graphs does not rule out self isomorphisms on the LHS of the inference), we were left with 3 inferences, of which two were dual to each other leaving the logically minimal $\{\mathsf{s}, \mathsf{m}\}$-independent inferences given in Section 3. These give a proof of Theorem 7, the main theorem of this paper.

6 Conclusions

In this work we undertook a computational approach towards the classification of linear inferences. To this end we succeeded in exhausting the linear inferences up to 8 variables, showing that there are two (distinct) 8 variable linear inferences that are independent of switch and medial. One of these new inferences contradicts a Conjecture 7.9 from [15]. Conversely, all linear inferences on 7 variables or fewer are already derivable using switch and medial.

We point out that it should be possible to adapt our implementation to a variety of logics and, in particular, graph-based systems such as those from [2, 1, 7]. This would be an interesting avenue for future work.

References

1 Matteo Acclavio, Ross Horne, and Lutz Straßburger. An analytic propositional proof system on graphs. *CoRR*, abs/2012.01102, 2020. `arXiv:2012.01102`.

2 Matteo Acclavio, Ross Horne, and Lutz Straßburger. Logic beyond formulas: A proof system on graphs. In Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller, editors, *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, pages 38–52. ACM, 2020. `doi:10.1145/3373718.3394763`.

3 Andreas Blass. A game semantics for linear logic. *Ann. Pure Appl. Log.*, 56(1-3):183–220, 1992. `doi:10.1016/0168-0072(92)90073-9`.

4 Coenraad Bron and Joep Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.

5 Kai Brünnler. *Deep inference and symmetry in classical proofs*. PhD thesis, Dresden University of Technology, Germany, 2003. URL: `http://hsss.slub-dresden.de/hsss/servlet/hsss.urlmapping.MappingServlet?id=1064911987703-3819`.

6 Kai Brünnler and Alwen Fernanto Tiu. A local system for classical logic. In Robert Nieuwenhuis and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba, December 3-7, 2001, Proceedings*, volume 2250 of *Lecture Notes in Computer Science*, pages 347–361. Springer, 2001. `doi:10.1007/3-540-45653-8_24`.

7 Cameron Calk, Anupam Das, and Tim Waring. Beyond formulas-as-cographs: an extension of boolean logic to arbitrary graphs. *CoRR*, abs/2004.12941, 2020. `arXiv:2004.12941`.

8 Stephen A. Cook and Robert A. Reckhow. On the lengths of proofs in the propositional calculus (preliminary version). In Robert L. Constable, Robert W. Ritchie, Jack W. Carlyle, and Michael A. Harrison, editors, *Proceedings of the 6th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1974, Seattle, Washington, USA*, pages 135–148. ACM, 1974. `doi:10.1145/800119.803893`.

9 Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *J. Symb. Log.*, 44(1):36–50, 1979. `doi:10.2307/2273702`.

10 Anupam Das. On the proof complexity of cut-free bounded deep inference. In Kai Brünnler and George Metcalfe, editors, *Automated Reasoning with Analytic Tableaux and Related Methods - 20th International Conference, TABLEAUX 2011, Bern, Switzerland, July 4-8, 2011. Proceedings*, volume 6793 of *Lecture Notes in Computer Science*, pages 134–148. Springer, 2011. `doi:10.1007/978-3-642-22119-4_12`.

11 Anupam Das. Rewriting with Linear Inferences in Propositional Logic. In Femke van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications (RTA 2013)*, volume 21 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 158–173, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.RTA.2013.158`.

12 Anupam Das. An unavoidable contraction loop in monotone deep inference, 2017. URL: `http://cs.bath.ac.uk/ag/das/con-loop.pdf`.

13 Anupam Das. A new linear inference of size 8. The Proof Theory Blog, June 2020. URL: `https://prooftheory.blog/2020/06/25/new-linear-inference/`.

14 Anupam Das and Lutz Straßburger. No complete linear term rewriting system for propositional logic. In Maribel Fernández, editor, *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland*, volume 36 of *LIPIcs*, pages 127–142. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. `doi:10.4230/LIPIcs.RTA.2015.127`.

15 Anupam Das and Lutz Straßburger. On linear rewriting systems for boolean logic and some applications to proof theory. *Log. Methods Comput. Sci.*, 12(4), 2016. `doi:10.2168/LMCS-12(4:9)2016`.

16 Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. `doi:10.1016/0304-3975(87)90045-4`.

17 Alessio Guglielmi. A system of interaction and structure. *ACM Trans. Comput. Log.*, 8(1):1, 2007. `doi:10.1145/1182613.1182614`.

18 Alessio Guglielmi and Tom Gundersen. Normalisation control in deep inference via atomic flows. *Log. Methods Comput. Sci.*, 4(1), 2008. `doi:10.2168/LMCS-4(1:9)2008`.

19 Tom Gundersen. *A General View of Normalisation through Atomic Flows*. PhD thesis, University of Bath, UK, 2009. URL: `https://tel.archives-ouvertes.fr/tel-00441540`.

20 Giorgi Japaridze. Introduction to cirquent calculus and abstract resource semantics. *CoRR*, abs/math/0506553, 2005. `arXiv:math/0506553`.

21 Giorgi Japaridze. Elementary-base cirquent calculus I: parallel and choice connectives. *CoRR*, abs/1707.04823, 2017. `arXiv:1707.04823`.

22 Jan Krajícek. The cook-reckhow definition. *CoRR*, abs/1909.03691, 2019. `arXiv:1909.03691`.

23 Lê Thành Dung Nguyên and Thomas Seiller. Coherent interaction graphs. In Thomas Ehrhard, Maribel Fernández, Valeria de Paiva, and Lorenzo Tortora de Falco, editors, *Proceedings Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity-TLLA@FLoC 2018, Oxford, UK, 7-8 July 2018*, volume 292 of *EPTCS*, pages 104–117, 2018. `doi:10.4204/EPTCS.292.6`.

24 Alex Rice. Linear inferences of size 7. The Proof Theory Blog, October 2020. URL: `https://prooftheory.blog/2020/10/01/linear-inferences-of-size-7/`.

25 Alex Rice. `lin_inf` rust crate. `https://github.com/alexarice/lin_inf`, 2021.

26 Lutz Straßburger. A local system for linear logic. In Matthias Baaz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 9th International Conference, LPAR 2002, Tbilisi, Georgia, October 14-18, 2002, Proceedings*, volume 2514 of *Lecture Notes in Computer Science*, pages 388–402. Springer, 2002. `doi:10.1007/3-540-36078-6_26`.

27 Lutz Straßburger. *Linear logic and noncommutativity in the calculus of structures*. PhD thesis, Dresden University of Technology, Germany, 2003. URL: `http://hsss.slub-dresden.de/hsss/servlet/hsss.urlmapping.MappingServlet?id=1063208959250-7293`.

28 Lutz Straßburger. A characterization of medial as rewriting rule. In Franz Baader, editor, *Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4533 of *Lecture Notes in Computer Science*, pages 344–358. Springer, 2007. `doi:10.1007/978-3-540-73449-9_26`.

29 Lutz Straßburger. Personal communication, 2012.

30 Lutz Straßburger. Extension without cut. *Ann. Pure Appl. Log.*, 163(12):1995–2007, 2012. `doi:10.1016/j.apal.2012.07.004`.

31 Alvin Šipraga. An automated search of linear inference rules. Summer research project. Supervised by Alessio Guglielmi and Anupam Das, 2012. URL: `http://arcturus.su/mimir/autolininf.pdf`.

A Further proofs and examples

Proof sketch of Proposition 2. Write $\rightsquigarrow$ for the rewriting relation obtained by orienting every pair of (3) left-to-right. Clearly $\rightsquigarrow$ is terminating since each step decreases formula size. For confluence, note that every critical pair must reduce to the same constant:

$$\begin{array}{llll} \bot \vee \bot \rightsquigarrow \bot & \bot \vee \top \rightsquigarrow \top & \top \vee \bot \rightsquigarrow \top & \top \vee \top \rightsquigarrow \top \\ \bot \wedge \bot \rightsquigarrow \bot & \bot \wedge \top \rightsquigarrow \bot & \top \wedge \bot \rightsquigarrow \bot & \top \wedge \top \rightsquigarrow \top \end{array}$$

◀

A.1 Recovering an 8 variable inference

The reason for writing the variation (6) in Section 3.1 instead of the one originally presented in [11] is that it allows us to recover one of the new 8-variable inferences, by a particular reduction first noticed in a blog post [13].

By setting $x' = u' = \neg u$ in (6) and simplifying, we obtain the linear inference:

$$\begin{array}{rl} & (z \vee (w \wedge w')) \wedge (y \vee y') \wedge ((x \wedge x') \vee z') \\ \to & (z \wedge (x \vee y)) \vee (w' \wedge x') \vee ((w \vee y') \wedge z') \end{array}$$

Again, the inference above is not $\{\mathsf{s}, \mathsf{m}\}$-minimal, since there are two possible applications of switch to the LHS that nonetheless imply the RHS:

$$((z \wedge (y \vee y')) \vee (w \wedge w')) \wedge ((x \wedge x') \vee z') \quad \text{or} \quad (z \vee (w \wedge w')) \wedge ((x \wedge x') \vee ((y \vee y') \wedge z'))$$

Furthermore, are two switch applications leading to the RHS that are nonetheless implied by their respective formulae above:[13]

$$((z \vee (w' \wedge x')) \wedge (x \vee y)) \vee ((w \vee y') \wedge z') \quad \text{or} \quad (z \wedge (x \vee y)) \vee ((w \vee y') \wedge ((w' \wedge x') \vee z'))$$

The two resulting linear inferences are, in fact, isomorphic and indeed $\{\mathsf{s}, \mathsf{m}\}$-minimal, as we shall explain in the next subsection. As we have already mentioned, the fact that this is a *logically minimal* linear inference is shown by means of the implementation presented in Section 5.

A.2 Validity of Equation 7

We consider each assignment that satisfies the LHS and argue that it also satisfies the RHS:
- $\{z, x, x'\}$ satisfies $z \wedge (x \vee y)$.
- $\{z, y, z'\}$ satisfies $z \wedge (x \vee y)$.
- $\{z, y', z'\}$ satisfies $(w \vee y') \wedge ((w' \wedge x') \vee z')$.
- $\{w, w', x, x'\}$ satisfies $(w \vee y') \wedge ((w' \wedge x') \vee z')$.
- $\{w, w', y, z'\}$ and $\{w, w', y', z'\}$ satisfy $(w \vee y') \wedge ((w' \wedge x') \vee z')$.

A.3 Validity of Equation 8

We consider each assignment that satisfies the LHS and argue that it also satisfies the RHS:
- $\{w, w', y, y'\}$ satisfies $w \wedge y$.
- $\{w, w', z, z'\}$ satisfies $w' \wedge z'$ and z.
- $\{x, x', y, y'\}$ satisfies x and $x' \wedge y'$.
- $\{x, x', z, z'\}$ satisfies x and z.

[13] Note that these switch applications were overlooked in the blog post [13].

A.4 $\{s, m\}$-independence and $\{s, m\}$-minimality of Equation 7

There are two possible medial applications to the subformula $(x \wedge x') \vee ((y \vee y') \wedge z')$ resulting in the following new LHSs:

- $(z \vee (w \wedge w')) \wedge (x \vee y \vee y') \wedge (x' \vee z')$. In this case $\{z, y', x'\}$ is a countermodel.
- $(z \vee (w \wedge w')) \wedge (x \vee z') \wedge (x' \vee y \vee y')$. In this case $\{z, z', x'\}$ is a countermodel.

There are two possible switch applications to the subformula $(y \vee y') \wedge z'$ resulting in the following new LHSs:

- $(z \vee (w \wedge w')) \wedge ((x \wedge x') \vee y \vee (y' \wedge z'))$. In this case $\{w, w', y\}$ is a countermodel.
- $(z \vee (w \wedge w')) \wedge ((x \wedge x') \vee y' \vee (y \wedge z'))$. In this case $\{z, y'\}$ is a countermodel.

Finally any other switch application is on the top-level conjunction, resulting in a formula of the form $z \vee X$, $(w \wedge w') \vee X$, $(x \wedge x') \vee X$ or $((y \vee y') \wedge z') \vee X$, which admits a countermodel $\{z\}$, $\{w, w'\}$, $\{x, x'\}$ or $\{y, z'\}$, respectively.

A.5 $\{s, m\}$-independence and $\{s, m\}$-minimality of Equation 8

Let us first consider rules applicable to the LHS. There are four possible medial applications, resulting in the following new LHSs:

- $(w \vee x) \wedge (w' \vee x') \wedge ((y \wedge y') \vee (z \wedge z'))$. In this case $\{w, x', y, y'\}$ is a countermodel.
- $(w \vee x') \wedge (w' \vee x) \wedge ((y \wedge y') \vee (z \wedge z'))$. In this case $\{x', w', y, y'\}$ is a countermodel.
- $((w \wedge w') \vee (x \wedge x')) \wedge (y \vee z) \wedge (y' \vee z')$. In this case $\{x, x', y, z'\}$ is a countermodel.
- $((w \wedge w') \vee (x \wedge x')) \wedge (y \vee z') \wedge (y' \vee z)$. In this case $\{w, w', z', y'\}$ is a countermodel.

Any switch application to the LHS must be on the top-level conjunction, and will have the form $(a \wedge a') \vee X$, for $a \in \{w, x, y, z\}$. However, $\{w, w'\}$, $\{x, x'\}$, $\{y, y'\}$ and $\{z, z'\}$ are each countermodels for the RHS.

Now let us consider the possible rule applications leading to the RHS. There are two possible medial instances, coming from the following new RHSs:

- $(w \wedge y) \vee (x \wedge x' \wedge y') \vee (w' \wedge z' \wedge z)$. In this case $\{x, x', z, z'\}$ is a countermodel.
- $(w \wedge y) \vee (x \wedge z) \vee (w' \wedge z' \wedge x' \wedge y')$. In this case $\{w, w', z, z'\}$ is a countermodel.

Now let us consider the switch instances:

- If the contractum of the switch is $x \vee (w' \wedge z')$, then $\{x, x', y, y'\}$ is a countermodel.
- If the contractum of the switch is $(x' \wedge y') \vee z$, then $\{w, w', z, z'\}$ is a countermodel.
- If the redex of the switch has the form $w \wedge X$ or $y \wedge X$, then $\{x, x', z, z'\}$ is a countermodel.
- If the redex of the switch has the form $X \wedge (x \vee (w' \wedge z'))$ or $X \wedge ((x' \wedge y') \vee z)$, then $\{w, w', y, y'\}$ is a countermodel.

A Modular Associative Commutative (AC) Congruence Closure Algorithm

Deepak Kapur ✉

Department of Computer Science, University of New Mexico, Albuquerque, NM, USA

Abstract

Algorithms for computing congruence closure of ground equations over uninterpreted symbols and interpreted symbols satisfying associativity and commutativity (AC) properties are proposed. The algorithms are based on a framework for computing the congruence closure by abstracting nonflat terms by constants as proposed first in Kapur's congruence closure algorithm (RTA97). The framework is general, flexible, and has been extended also to develop congruence closure algorithms for the cases when associative-commutative function symbols can have additional properties including idempotency, nilpotency and/or have identities, as well as their various combinations. The algorithms are modular; their correctness and termination proofs are simple, exploiting modularity. Unlike earlier algorithms, the proposed algorithms neither rely on complex AC compatible well-founded orderings on nonvariable terms nor need to use the associative-commutative unification and extension rules in completion for generating canonical rewrite systems for congruence closures. They are particularly suited for integrating into Satisfiability modulo Theories (SMT) solvers.

2012 ACM Subject Classification Software and its engineering; Theory of computation; Mathematics of computing

Keywords and phrases Congruence Closure, Associative and Commutative, Word Problems, Finitely Presented Algebras, Equational Theories

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.15

Funding Research partially supported by the NSF award: CCF-1908804.

Acknowledgements Heartfelt thanks to the referees for their reports which substantially improved the presentation.

1 Introduction

Equality reasoning arises in many applications including compiler optimization, functional languages, and reasoning about data bases, most importantly, reasoning about different aspects of software and hardware. The significance of the congruence closure algorithms on ground equations in compiler optimization and verification applications was recognized in the mid 70's and early 80's, leading to a number of algorithms for computing the congruence closure of ground equations on uninterpreted function symbols [8, 28, 25]. Whereas congruence closure algorithms were implemented in earlier verification systems [28, 25, 19, 32], their role has become particularly critical in Satisfiability modulo Theories (SMT) solvers as a glue to combine different decision procedure for various theories.

We present algorithms for the congruence closure of ground equations which in addition to uninterpreted function symbols, have symbols with the associative (A) and commutative (C) properties. Using these algorithms, it can be decided whether another ground equation follows from a finite set of ground equations with associative-commutative (AC) symbols and uninterpreted symbols. Canonical forms (unique normal forms) can be associated with congruence classes. Further, a unique reduced ground congruence closure presentation can be associated with a finite set of ground equations, enabling checks whether two different finite sets of ground equations define the same congruence closure or one is contained in the other. In the presence of disequations on ground terms with AC and uninterpreted symbols, a finite set of ground equations and disequations can be checked for satisfiability.

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi; Article No. 15; pp. 15:1–15:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The main contributions of the paper are (i) a modular combination framework for the congruence closure of ground equations with multiple AC symbols, uninterpreted symbols, and constants, leading to (ii) modular and simple algorithms that can use flexible termination orderings on ground terms and do not need to use AC/E unification algorithms for generating canonical forms; (iii) the termination and correctness proofs of these algorithms are modular and easier. The key insights are based on extending the author's previous work presented in [13, 14]: introduction of new constants for nested subterms, resulting in flat and constant equations, extended to purification of mixed subterms with many AC symbols by flattening AC ground terms and introducing new constants for pure AC terms in each AC symbol, resulting in disjoint subsets of ground equations on single AC symbols with shared constants. The result of this transformation is a finite union of disjoint subsets of ground equations with shared constants: (i) a finite set of constant equations, (ii) a finite set of flat equations with uninterpreted symbols, and (iii) for each AC symbol, a finite set of equations on pure flattened terms in a single AC symbol.

With the above decomposition, reduced canonical rewrite systems are generated for each of the subsystems using their respective termination orderings that extend a common total ordering on constants. A combination of the reduced canonical rewrite systems is achieved by propagating constant equalities among various rewrite systems; whenever new implied constant equalities are generated, each of the reduced canonical rewrite systems must be updated with additional computations to ensure their canonicity. Due to this modularity and factoring/decomposition, the termination and correctness proofs can be done independently of each subsystem, providing considerable flexibility in choosing termination orderings.

The combination algorithm terminates when no additional implied constant equalities are generated. Since there are only finitely many constants in the input and only finitely many constants are needed for purification, the termination of the combination algorithm is guaranteed. The result is a reduced canonical rewrite system corresponding to the AC congruence closure of the input ground equations, which is unique for a fixed family of total orderings on constants and different pure AC terms in each AC symbol. The reduced canonical rewrite system can be used to generate canonical signatures of ground terms with respect to the congruence closure.

The framework provides flexibility in choosing orderings on constants and terms with different AC symbols, enabling canonical forms suitable for applications instead of restrictions imposed due to the congruence closure algorithms. Interpreted AC symbols can be further enriched with properties including idempotency, nilpotency, existence of identities, and simply commutativity, without restrictions on orderings on mixed terms. Termination and correctness proofs of congruence closure algorithms are modular and simple in contrast to complex arguments and proofs in [5, 24]. These features of the proposed algorithms make them attractive for integration into SMT solvers as their implementation does not need heavy duty infrastructure including AC unification, extension rules, and AC compatible orderings.

The next subsection contrasts in detail, the results of this paper with the previous methods, discussing the advantages of the proposed framework and the resulting algorithms. Section 2 includes definitions of congruence closure with uninterpreted and interpreted symbols. This is followed by a review of key constructions used in the congruence closure algorithm over uninterpreted symbols as proposed in [13]. Section 3 introduces purification and flattening of ground terms with AC and uninterpreted symbols by extending the signature and introducing new constants. This is followed by an algorithm first reported in [11] for computing the congruence closure and the associated canonical rewrite system from ground equations with a single AC symbol and constants. It is shown how additional properties of AC symbols such as idempotency, nilpotency and identity can be integrated into the algorithm. In the next subsection, an algorithm for computing congruence closure of AC ground equations with multiple AC symbols and constants is presented. Section 4 generalizes to the case of combination of AC symbols and uninterpreted symbols. Section 5 discusses a variety of examples illustrating the proposed algorithms. Section 6 illustrates the power and elegance of the proposed framework by demonstrating how the congruence closure algorithm for two AC symbols can be further generalized to get a Gröbner basis algorithm on polynomial ideals over the integers. Section 7 concludes with some ideas for further investigation. Appendix includes proofs of some of the results in the paper.

1.1 Related Work

Congruence closure algorithms have been developed and analyzed for over four decades [8, 28, 25]. The algorithms presented here use the framework first informally proposed in [13] for congruence closure in which the author separated the algorithm into two parts: (i) constant equivalence closure, and (ii) nonconstant flat terms related to constants by flattening nested terms by introducing new constants to stand for them, and (iii) update nonconstant rules as constant equivalence closure evolves. This simplified the presentation, the correctness argument as well as the complexity analysis, and made the framework easier to generalize to other settings including conditional congruence closure [14] and semantic congruence closure [1]. Further, it enables the generation of a reduced unique canonical rewrite system for a congruence closure, assuming a total ordering on constants; most importantly, the framework gives freedom in choosing orderings on ground terms, leading to desired canonical forms appropriate for applications.

To generate congruence closure in the presence of AC symbols, the proposed framework builds on the author and his collaborators' work dating back to 1985, where they demonstrated how an ideal-theoretic approach based on Gröbner basis algorithms could be employed for word problems and unification problems over commutative algebras [11].

Congruence closure algorithms on ground equations with interpreted symbols can be viewed as special cases of the Knuth-Bendix completion procedure [20] on (nonground) equations with universal properties characterizing the semantics of the interpreted symbols. In case of equations with AC symbols, Peterson and Stickel's extension of the Knuth-Bendix completion [27] using extension rules, AC unification and AC compatible orderings can be used for congruence closure over AC symbols. For an arbitrary set E of universal axioms characterizing the semantics of interpreted symbols, E-completion with coherence check and E-unification along with E-compatible orderings need to be used. Most of the general purpose methods do not terminate in general. Even though the Knuth-Bendix procedure can be easily proved to terminate on ground equations of uninterpreted terms, that is not necessarily case for its extensions for other ground formulas.

In [6], a generalization of the Knuth-Bendix completion procedure [20] to handle AC symbols [27] is adapted to develop decision algorithms for word problems over finitely presented commutative semigroups; this is equivalent to the congruence closure of ground equations with a single AC symbol on constants. Related methods using extension rules introduced to handle AC symbols and AC rewriting for solving word problems over other finite presented commutative algebras were subsequently reported in [29].

In [24], the authors used the completion algorithm discussed in [11] and a total AC-compatible polynomial reduction ordering on congruence classes of AC ground terms to establish the existence of a ground canonical AC system first with one AC symbol. To extend their method to multiple AC symbols, particularly the instances of distributivity property relating ground terms in two AC symbols $*$ and $+$: the authors had to combine an AC-compatible total reduction ordering on terms along with complex polynomial interpretations with polynomial ranges, resulting in a complicated proof to orient the distributivity axiom from left to right. Using this highly specialized generalization of polynomial orderings, it was proved in [24] that every ground AC theory has a finite canonical system which also serves as its congruence closure.

The proposed approach, in contrast, is orthogonal to ordering arguments on AC ground terms; instead a total ordering on constants in the extended signature is extended to many different possible orderings on pure terms with a single AC symbol is sufficient to compute a canonical ground AC rewrite system. Different orderings on AC terms with different AC symbols can be used; for example, for ground terms of an AC symbol $+$ could be oriented in a completely different way than ground terms for another AC symbol $*$. Instances of the distributivity property expressed on different AC ground terms can also be oriented in different nonuniform ways. This leads to flexible ordering requirements on uninterpreted and interpreted symbols based on the properties desired of canonical forms.

In [21], a different approach was taken for computing a finite canonical rewrite system for ground equations on AC symbols. Marche first proved a general result about AC ground theories that for any finite set of ground equations with AC symbols, if there is an equivalent canonical rewrite system modulo AC, then that rewrite system must be finite. He gave an AC completion procedure, which does not terminate even on ground equations; he then proved its termination on ground equations with AC symbols using a special control on its inference rules using a total ordering on AC ground terms in [24]. Neither in [24] nor in [21], any explicit mention is made of uninterpreted symbols appearing in ground equations.

Similar to [6], several approaches based on adapting Peterson and Stickel's generalization of the Knuth-Bendix completion procedure to consider special ground theories have been reported [29, 21]. In [4, 5], the authors adapted Kapur's congruence closure [13] using its key ideas to an abstract inference system ($Table$ for new constant symbols defining flat terms introducing during flattening of nested nonconstant terms in $Make\_Rule$ were called D-rule for defining a flat term and C-rule for introducing a new constant symbol). Various congruence closure algorithms, including Sethi, Downey and Tarjan [8], Nelson and Oppen [25] and Shostak [28], from the literature can be expressed as different combinations of these inference steps. They also proposed an extension of this inference system to AC function symbols, essentially integrating it with [27] of the Knuth-Bendix completion procedure using extension rules, adapted to ground equations with AC symbols. All of these approaches based on Paterson and Stickel's generalization used extension rules introduced in [27] to define rewriting modulo AC theories so that a local-confluence test for rules with AC symbols could be developed using AC unification. During completion on ground terms, rules with variables appear in intermediate computations. All of these approaches suffer from having to consider many unnecessary inferences due to extension rules and AC unification, as it is well-known that AC unification can generate doubly exponentially many unifiers [18].

An approach based on normalized rewriting was proposed in [22] and decision procedures were reported for ground AC theories with AC symbols satisfying additional properties including idempotency, nilpotency and identity as well as their combinations. This was an attempt to integrate Le Chenadec's method [29] for finitely presented algebraic structures with Peterson and Stickel's AC completion, addressing weaknesses in E-completion and constrained rewriting, while considering additional axioms of AC symbols, including identity, idempotency and nilpotency for which termination orderings are difficult to design. However, that approach had to redefine local confluence for normalized rewriting and normalized critical pairs, leading to a complex completion procedure whose termination and proof of correctness needed extremely sophisticated machinery of normalized proofs.

The algorithms presented in this paper, in contrast, are very different and are based on an approach first presented in [11] by the author with his collaborators. Their termination and correctness proofs are based on the termination and correctness proofs of a congruence closure algorithm for uninterpreted symbols (if present) and the termination and correctness of an algorithm for deciding the word problems of a finitely presented commutative semigroup using Dickson's Lemma. Since the combination is done by propagating equalities on shared constants among various components, the termination and correctness proofs of the combination algorithm become much easier since there are only finitely many constants to consider, as determined by the size of the input ground equations.

A detailed comparison leads to several reasons why the proposed algorithms are simpler, modular, easier to understand and prove correct: (i) there is no need in the proposed approach to use extension rules whereas almost all other approaches are based on adapting AC/E completion procedures for this setting requiring considerable/sophisticated infrastructure including AC unification and E/Normalized rewriting As a result, proofs of correctness and termination become complex using heavy machinery including proof orderings and normalized proof methods not to mention arguments dealing with fairness of completion procedures. (ii) all require complex total AC compatible orderings. In contrast, ordering restrictions in the proposed algorithms are dictated by individual components–little restriction for the uninterpreted part, independent orderings on $+$-monomials for each AC symbol $+$ insofar as orderings on constants are shared by all parts, thus giving considerable flexibility in choosing termination orderings. In most related approaches except for [24], critical pairs computed using expensive AC unification steps are needed, which are likely to make the algorithms inefficient; it is well-known that many superfluous critical pairs are generated due to AC unification. These advantages make us predict that the proposed algorithms can be easily integrated with SMT solvers since they do not require sophisticated machinery of AC-unification and AC-compatible orderings, extension rules and AC completion.

2 Preliminaries

Let F be a set of function symbols including constants and $GT(F)$ be the ground terms constructed from F; sometimes, we will write it as $GT(F, C)$ to highlight the constants of F. We will abuse the terminology by calling a k-ary function symbol as a function symbol if $k > 0$ and constant if $k = 0$. A function term is meant to be a nonconstant term with a nonconstant outermost symbol. Symbols in F are either uninterpreted (to mean no semantic property of such a function is assumed) or interpreted satisfying properties expressed as universally quantified equations (called universal equations).

2.1 Congruence Relations

▶ **Definition 1.** *Given a finite set $S = \{a_i = b_i | 1 \leq i \leq m\}$ of ground equations where $a_i, b_i \in GT(F)$, the congruence closure $CC(S)$ is inductively defined as follows: (i) $S \subseteq CC(S)$, (ii) for every $a \in GT(F)$, $a = a \in CC(S)$, (iii) if $a = b \in CC(S)$, $b = a \in CC(S)$, (iv) if $a = b$ and $b = c \in CC(S)$, $a = c \in CC(S)$, and (v) for every nonconstant $f \in F$ of arity $k > 0$, if for all $1 \leq k$, $a_i = b_i \in CC(S)$, then $f(a_1, \cdots, a_k) = f(b_1, \cdots, b_k) \in CC(S)$. Nothing else is in $CC(S)$.*

$CC(S)$ is thus the smallest relation that includes S and is closed under reflexivity, symmetry, transitivity, and under function application. It is easy to see that $CC(S)$ is also the equational theory of S [2, 1].

2.2 Kapur's Congruence Closure Algorithm for Uninterpreted Symbols

The algorithm in [13, 14] for computing congruence closure of a finite set S of ground equations serves as the main building block in this paper. The algorithm extends the input signature by introducing new constant symbols to recursively stand for each nonconstant subterm and generates two types of equations: (i) constant equations, and (ii) flat terms of the form $f(c_1, \cdots, c_k)$ equal to constants. A disequation is converted to a disequation on constants by introducing new symbols for the terms. It can be proved that the congruence closure of ground equations on the extended signature when restricted to the original signature, is indeed the congruence closure of the original equations [1].

Using a total ordering on constants (typically with new constants introduced to extend the signature being smaller than constants from the input), the output of the algorithm in [13, 14] is a reduced canonical rewrite system R_S associated with $CC(S)$ (as well as S) that includes *function*, also called *flat* rules of the form $f(c_1, \cdots, c_k) \to d$ and *constant* rules $c \to d$ such that no two left sides of the rules are identical; further, all constants are in canonical forms. As proved in [13] (see also [26]),

▶ **Theorem 2** ([13]). *Given a set S of ground equations, a reduced canonical rewrite system R_S on the extended signature, consisting of nonconstant flat rules $f(c_1, \ldots, c_k) \to d$, and constant rules $c \to d$, can be generated from S in $O(n^2)$ steps. The complexity can be further reduced to $O(n * log(n))$ steps if all function symbols are binary or unary. For a given total ordering $\gg$ on constants, R_S is unique for S, subject to the introduction of the same set of new constants for nonconstant subterms.*

As shown in [8] (see [26]), function symbols of arity > 2 can be encoded using binary symbols using additional linearly many steps.

The canonical form of a ground term g using R_S is denoted by $\hat{g}$ and is its canonical signature (in the extended language). Ground terms g_1, g_2 are congruent in $CC(S)$ iff $\hat{g_1} = \hat{g_2}$.

2.3 AC Congruence Closure

The above definition of congruence closure $CC(S)$ is extended to consider interpreted symbols. Let IE be a finite set of universally quantified equations with variables, specifying properties of interpreted function symbols in F. For example, the properties of an AC symbol f are: $\forall x, y, z,\ f(x, y) = f(y, x),\ \ f(x, f(y, z)) = f(f(x, y), z)$. An idempotent symbol g, for another example, is specified as $\forall x,\ g(x, x) = x$. To incorporate the semantics of these properties:

(vi) from a universal axiom $s = t \in IE$, for each variable x in s, t, for any ground substitution σ, i.e., $\sigma(x) \in GT(F)$, $\sigma(s) = \sigma(t) \in CC(S)$.

$CC(S)$ is thus the smallest relation that includes S and is closed under reflexivity, symmetry, transitivity, function application, and the substitution of variables in IE by ground terms. $CC(S)$ is also the ground equational theory of S.

Given a finite set S of ground equations with uninterpreted and interpreted symbols, the congruence closure membership problem is to check whether another ground equation $u = v \in CC(S)$ (meaning semantically that $u = v$ follows from S, written as $S \models u = v$). A related problem is whether given two sets S_1 and S_2 of ground equations, $CC(S_2) \subseteq CC(S_1)$, equivalently $S_1 \models S_2$. Birkhoff's theorem relates the syntactic properties, the equational theory, and the semantics of S.

If S also includes ground disequations, then besides checking the unsatisfiability of a finite set of ground equations and disequations, new disequations can be derived in case S is satisfiable. The inference rule for deriving new disequations for an uninterpreted symbol is:

$f(c_1, \ldots, c_k) \neq f(d_1, \cdots, d_k) \implies (c_1 \neq d_1 \vee \cdots \vee c_k \neq d_k)$.

In particular, if f is unary, then the disequation is immediately derived.

Disequations in case of interpreted symbols generate more interesting formulas. In case of a commutative symbol f, for example, the disequation $f(a,b) \neq f(c,d)$ implies $(a \neq c \vee a \neq c \vee b \neq d \vee c \neq d)$. For an AC symbol g, as an example, the disequation $g(a, g(b,c)) \neq g(a, g(a, g(a,c)))$ implies $(a \neq g(a,c) \vee b \neq c \vee b \neq g(a,a) \vee \cdots)$.

To emphasize the presence of AC symbols, let $ACCC(S)$ stand for the AC congruence closure of S with AC symbols; we will interchangeably use $ACCC(S)$ and $CC(S)$.

2.4 Flattening and Purification

Let F include a finite set C of constants, a finite set F_U of uninterpreted symbols, and a finite set F_{AC} of AC symbols. i.e., $F = F_{AC} \cup F_U \cup C$.

Following [13], ground equations in $GT(F)$ with AC symbols are transformed into three kinds of equations by introducing new constants for subterms: (i) constant equations of the form $c = d$, (ii) flat equations with uninterpreted symbols of the form $h(c_1, \cdots, c_k) = d$, and (iii) for each $f \in F_{AC}$, $f(c_1 \cdots, c_j) = f(d_1, \cdots, d_{j'})$, where c's, d's are constants in C, $h \in F_U$, and every AC symbol f is viewed to be variadic (including $f(c) = c$). Nested subterms of every AC symbol f are repeatedly flattened: $f(f(s_1, s_2), s_3)$ to $f(s_1, s_2, s_3)$ and $f(s_1, f(s_2, s_3))$ to $f(s_1, s_2, s_3)$ until all arguments to f are constants or nonconstant terms with outermost symbols different from f. Nonconstant arguments of a mixed AC term $f(t_1, \cdots, t_k)$ are transformed to $f(u_1, \cdots, u_k)$, where u_i's are new constants, with $t_i = u_i$ if t_i is not a constant. A subterm whose outermost function symbol is uninterpreted, is also flattened by introducing new constants for their nonconstant arguments. These transformations are recursively applied on the equations with new constants.

New constants are introduced only for nonconstant subterms and their number is minimized by introducing a single new constant for each distinct subterm irrespective of its number of occurrences (which is equivalent to representing terms by directed acyclic graphs (DAGs) with full sharing whose each non-leaf node is assigned a distinct constant). As an example, $((f(a,b) * g(a)) + f(a + (a + b), (a * b) + b)) * ((g(a) + ((f(a,b) + a) + a)) + (g(a) * b)) = a$ is purified and flattened with new constants u_i's, resulting in $\{f(a,b) = u_1, g(a) = u_2, u_1 * u_2 = u_3, a + a + b = u_4, a * b = u_5, u_5 + b = u_6, f(u_4, u_6) = u_7, u_3 + u_7 = u_8, u_2 * b = u_9, u_2 + u_1 + a + a + u_9 = u_{10}, u_7 * u_{10} = a$

The arguments of an AC symbol are represented as a multiset since the order does not matter but multiplicity does. For an AC symbol f, let $f(M)$ be a flattened term $f(a_1, \cdots, a_k)$ with $M = \{\{a_1, \cdots, a_k\}\}$, a multiset of constants; $f(M)$ is called an **f-monomial**. In case f has its identity e, i.e., $f(x, e) = x$, then e is written as is, or $f(\{\{\}\})$. A singleton constant c is written as is or equivalently $f(\{\{c\}\})$. An f-monomial $f(M_1)$ is equal to $f(M_2)$ iff the multisets M_1 and M_2 are equal.

Without any loss of generality, the input to the algorithms below are assumed to be the above set of flattened ground equations on constants.

3 Congruence Closure with Associative-Commutative (AC) Functions

The focus in this section is on interpreted symbols with the associative-commutative properties; later, uninterpreted symbols are considered.

Checking whether a ground equation on AC terms is in the congruence closure $ACCC(S)$ of a finite set S on ground equations is the word problem over finitely presented commutative algebraic structures, presented by S characterizing their interpretations as discussed in [11, 29, 6]. In the presence of disequations over AC ground terms, one is also interested in determining whether the set of ground equations and disequations is satisfiable or not.

Another goal is to associate a reduced canonical rewrite system as a unique presentation of $ACCC(S)$ and a canonical signature with every AC congruence class in the AC congruence closure of a satisfiable S.

For a single AC symbol f and a finite set S of monomial equations $\{f(M_i) = f(M_i') | 1 \leq i \leq k\}$, $ACCC(S)$ is the reflexive, symmetric and transitive closure of S closed under f: if $f(M_1) = f(M_2)$ and $f(N_1) = f(N_2)$ in $ACCC(S)$, then $f(M_1 \cup N_1) = f(M_2 \cup N_2)$ is also in $ACCC(S)$. In case of multiple AC symbols, for every AC symbol $g \neq f$, $g(f(M_1), f(N_1)) = g(f(M_2), f(N_2)) \in ACCC(S)$.

3.1 Congruence Closure with a Single AC Symbol

As in [13], we follow a rewrite-based approach for computing the AC congruence closure $ACCC(S)$ by generating a canonical rewrite system from S. To make rewrite rules from equations in S, a total ordering $\gg$ on the set C of constants is extended to a total ordering on f-monomials and denoted as $\gg_f$. One of the main advantages of the proposed approach is the flexibility in using termination orderings on f-monomials, both from the literature on termination orderings on term rewriting systems as well as well-founded orderings (also called admissible orderings) from the literature on symbolic computation including Gröbner basis.

Using the terminology from the Gröbner basis literature, an ordering $\gg_f$ on the set of f-monomials, $GT(\{f\}, C)$, is called *admissible* iff i) $f(A) \gg_f f(B)$ if the multiset B is a proper subset of the multiset A (subterm property) for any nonempty multiset M, and (ii) for any multiset B, $f(A_1) \gg_f f(A_2) \implies f(A_1 \cup B) \gg_f f(A_2 \cup B)$ (the compatibility property). $f(\{\{\}\})$ may or may not be included in $GT(F)$ depending upon an application.

From S, a rewrite system R_S is associated with S by orienting nontrivial equations in S (after deleting trivial equations $t = t$ from S) using $\gg_f$: a ground equation $f(A_1) = f(A_2)$ is oriented into a terminating rewrite rule $f(A_1) \to f(A_2)$, where $f(A_1) \gg_f f(A_2)$. The rewriting relation induced by this rewrite rule is defined below.

▶ **Definition 3.** *A flattened term $f(M)$ is* rewritten *in one step, denoted by $\to_{AC}$ (or simply $\to$), using a rule $f(A_1) \to f(A_2)$ to $f(M')$ iff $A_1 \subseteq M$ and $M' = (M - A_1) \cup A_2$, where $-, \cup$ are operations on multisets.*

Given that $f(A_1) \gg_f f(A_2)$, it follows that $f(M) \gg_f f(M')$, implying the rewriting terminates. Standard notation and concepts from [2] are used to represent and study properties of the reflexive and transitive closure and transitive closure of $\to_{AC}$ induced by R_S; the reflexive, symmetric and transitive closure of $\to_{AC}$ is the AC congruence closure $ACCC(S)$ of S. Below, the subscript AC is dropped from $\to_{AC}$, f is dropped from S_f and $\gg_f$ whenever obvious from the context.

A rewrite relation $\to$ defined by R_S is called terminating iff there are no infinite rewrite chains of the form $t_0 \to t_1 \to \cdots \to t_k \to \cdots$. A rewrite relation $\to$ is *locally confluent* iff for any term t such that $t \to u_1, t \to u_2$, there exists v such that $u_1 \to^* v, u_2 \to^* v$. $\to$ is confluent iff for any term t such that $t \to^* u_1, t \to^* u_2$, there exists v such that $u_1 \to^* v, u_2 \to^* v$. $\to$ is canonical iff it is terminating and locally-confluent (and hence also terminating and confluent). A term t is in normal form iff there is no u such that $t \to u$.

An f-monomial $f(M)$ is in normal form with respect to R_S iff $f(M)$ cannot be rewritten using any rule in R_S.

Define a nonstrict partial ordering on f-monomials, informally capturing when an f-monomial rewrites another f-monomial, called the **Dickson** ordering: $f(M) \gg_D f(M')$ iff M' is a subset of M. Observe that the strict subpart of this ordering, while well-founded, is not total; for example, two distinct singleton multisets (constants) $\{\{a\}\} \neq \{\{b\}\}$ cannot be compared. This ordering is later used to show the termination of the completion algorithm.

A rewrite system R_S is called *reduced* iff neither the left side nor the right side of any rule in R_S can be rewritten by any of the other rules in R_S.

As in [11], the local confluence of R_S can be checked using the following constructions of superposition and critical pair.

▶ **Definition 4.** *Given two distinct rewrite rules $f(A_1) \to f(A_2)$, $f(B_1) \to f(B_2)$, let $AB = (A_1 \cup B_1) - (A_1 \cap B_1)$; $f(AB)$ is then the* superposition *of the two rules, and the* critical pair *is $(f((AB - A_1) \cup A_2), f((AB - B_1) \cup B_2))$.*

To illustrate, consider two rules $f(a,b) \to a, f(b,c) \to b$; their superposition $f(a,b,c)$ leads to the critical pair $(f(a,c), f(a,b))$.

A rule can have a constant on its left side and a nonconstant on its right side. As stated before, a singleton constant stands for the multiset containing that constant.

A critical pair is *nontrivial* iff the normal forms of its two components in $\to_{AC}$ as multisets are not the same (i.e., they are not joinable). A nontrivial critical pair generates an implied equality relating distinct normal forms of its two components.

For the above two rewrite rules, normal forms of two sides are $(f(a,c), a)$, respectively, indicating that the two rules are not locally confluent. A new derived equality is generated: $f(a,c) = a$ which is in $ACCC(\{f(a,b) = a, f(b,c) = b\})$.

It is easy to prove that if A_1, B_1 are disjoint multisets, their critical pair is trivial. Many critical pair criteria to identify additional trivial critical pairs have been investigated and proposed in [7, 17, 3].

▶ **Lemma 5.** *An AC rewrite system R_{S_f} is* locally confluent *iff the critical pair: $(f((AB - A_1) \cup A_2), f((AB - B_1) \cup B_2))$ between every pair of distinct rules $f(A_1) \to f(A_2), f(B_1) \to f(B_2)$ is joinable, where $AB = (A_1 \cup B_1) - (A_1 \cap B_1)$.*

See the Appendix for a proof.

Using the above local confluence check, a completion procedure is designed in the classical manner; equivalently, a nondeterministic algorithm can be given as a set of inference rules [2]. If a given rewrite system is not locally confluent, then new rules generated from nontrivial critical pairs (that are not joinable) are added until the resulting rewrite system is locally confluent. New rules can always be oriented since an ordering on f-monomials is assumed to be total. This completion algorithm is a special case of Gröbner basis algorithm on monomials built using a single AC symbol. The result of the completion algorithm is a locally confluent and terminating rewrite system for $ACCC(S)$.

Doing the completion algorithm on the two rules in the above examples, the derived equality is oriented into a new rule $f(a,c) \to a$. The system $\{f(a,b) \to a, f(b,c) \to b, f(a,c) \to a\}$ is indeed locally-confluent. This canonical rewrite system is a presentation of the congruence closure of $\{f(a,b) = a, f(b,c) = b\}$. Using the rewrite system, membership in its congruence closure can be decided by rewriting: $f(a,b,b) = f(a,b,c) \in ACCC(S)$ whereas $f(a,b,b) \neq f(a,a,b)$.

A simple completion algorithm is presented for the sake of completeness. It takes as input, a finite set S_C of constant equations and a finite set S_f of equations on f-monomials, and a total ordering $\gg_f$ on f-monomials extending a total ordering ordering $\gg$ on constants, and computes a reduced canonical rewrite system R_f (interchangeably written as R_S) such that $ACCC(S) = ACCC(S_{R_f})$, where S_{R_f} is the set of equations $l = r$ for every $l \to r \in R_f$.

Algorithm 1 1AC-Completion($S = S_f \cup S_C$, $\gg_f$).

1. Orient constant equations in S_C into terminating rewrite rules R_C using $\gg$ and interreduce them. Equivalently, using Tarjan's Union-Find data structure, for every constant $c \in C$, compute, from S_C, the equivalence class $[c]$ of constants containing c and make $R_C = \cup_{c\in C}\{c \to \hat{c} \mid c \neq \hat{c}$ and $\hat{c}$ is the least element in $[c]\}$.
   Initialize R_f to be R_C. Let $T := S_f$.
2. Pick an f-monomial equation $l = r \in T$ using some selection criterion (typically an equation of the smallest size) and remove it from T. Compute normal forms $\hat{l}, \hat{r}$ using R_f. If equal, then discard the equation, otherwise, orient into a terminating rewrite rule using $\gg_f$. Without any loss of generality, let the rule be $\hat{l} \to \hat{r}$.
3. Generate critical pairs between $\hat{l} \to \hat{r}$ and every f-rule in R_f, adding them to T.[1]
4. Add the new rule $\hat{l} \to \hat{r}$ into R_f; interreduce other rules in R_f using the new rule.
   (i) For every rule $l \to r$ in R_f whose left side l is reduced by $\hat{l} \to \hat{r}$, remove $l \to r$ from R_f and insert $l = r$ in T. If l cannot be reduced but r can be reduced, then reduce r by the new rule and generate a normal form r' of the result. Replace $l \to r$ in R_f by $l \to r'$.
5. Repeat the previous three steps until the critical pairs among all pairs of rules in R_f are joinable, and T becomes empty.
6. Output R_f as the canonical rewrite system associated with S.

▶ **Theorem 6.** *The algorithm* **1AC-Completion** *terminates, i.e., in Step 4, rules to R_f cannot be added infinitely often.*

See the Appendix for a proof.

▶ **Theorem 7.** *Given a finite set S of ground equations with a single AC symbol f and constants, and a total admissible ordering $\gg_f$ on flattened AC terms and constants, a reduced canonical rewrite system R_f is generated by the above completion procedure, which serves as a decision procedure for $ACCC(S)$.*

The proof the theorem is classical, typical of a correctness proof of a completion algorithm based on ensuring local confluence by adding new rules generated from superpositions whose critical pairs are not joinable.

▶ **Theorem 8.** *Given a total ordering $\gg_f$ on f-monomials, there is a unique reduced canonical rewrite system associated with S_f*

See the Appendix for a proof.

The complexity of this specialized decision procedure has been proved to require exponential space and double exponential upper bound on time complexity [23, 31].

The above completion algorithm generates a unique reduced canonical rewrite system R_f for the congruence closure $ACCC(S)$ because of interreduction of rules whenever a new rule is added to R_f; R_f (R_S) thus serves as its unique presentation. Using the same ordering $\gg$ on f-monomials, two sets S_1, S_2 of AC ground equations have identical (modulo presentation of multisets as AC terms) reduced rewrite systems $R_{S_1} = R_{S_2}$ iff $ACCC(S_1) = ACCC(S_2)$, thus generalizing the result for the uninterpreted case. Every f-monomial in $GT(\{f\}, C)$ has its canonical signature–its canonical form computed using R_f generated from S.

3.2 Idempotent and/or Nilpotent AC Symbols with Identity

If an AC symbol f has additional properties such as nilpotency, idempotency and/or unit, the above completion algorithm can be easily extended by expanding the local confluence check. Along with the above discussed critical pairs from a distinct pair of rules, additional critical pairs must be considered from each rule in R_f. We discuss below the case of an AC symbol being idempotent in detail; analysis for a nilpotent AC symbol, an AC symbol with identity, and various combination of properties is similar.

For any rule $f(M) \to f(N)$ where f is idempotent and M, N do not have duplicates, for every constant $a \in M$, generate a superposition $f(M \cup \{\{a\}\})$, a new critical pair $(f(N \cup \{\{a\}\}), f(M))$ and check its joinability. It can be proved that R_f is locally confluent iff the critical pairs constructed as above, from each distinct pair of rules in R_f and the new critical pair from each rule are joinable; see the Appendix for a proof. For an example, from $f(a, b) \to c$ with an idempotent f, the superpositions are $f(a, a, b)$ and $f(a, b, b)$, leading to the critical pairs: $(f(a, c), f(a, b))$ and $(f(b, c), f(a, b))$, respectively, which further reduces to $(f(a, c), c)$ and $(f(b, c), c)$, respectively. See the Appendix for a proof sketch.

For a nilpotent AC symbol f with $f(x, x) = e$, for every rule $f(M) \to f(N)$, generate a critical pair $(f(N \cup \{\{a\}\}), f((M - \{\{a\}\}) \cup \{\{e\}\}))$. If f has identity, say e, no additional critical pair is needed since from every rule $f(M) \to f(N)$, $(f(N \cup \{\{e\}\}), f(M))$ are trivially joinable. The termination proof from the previous subsection extends to each of these cases and their combination.

3.3 Computing Congruence Closure with Multiple AC symbols

The extension of the above algorithm for computing congruence closure with a single AC symbol to multiple AC symbols is straightforward.

Given a total ordering $\gg$ on constants, for each AC symbol f, define a total well-founded admissible ordering $\gg_f$ on f-monomials extending $\gg$. However, nonconstant f-monomials are not comparable with nonconstant g-monomials for $f \neq g$.

From S_C and each S_f, reduced canonical rewrite systems R_C and R_f, respectively are independently generated using the algorithm of the previous subsection. Equalities on shared constants must be propagated until no additional implied equalities are generated.

Any constant equality $c = d$ generated in an $R_f, f \in F_{AC}$, is oriented as a rewrite rule; wlog $c \to d$ is added to R_C with c reduced to d everywhere in various R_f's. If c appears in the left side of a rule in some R_g, it is removed from R_g and instead viewed as an equation, which is further reduced and the normalized equation is oriented as a rewrite rule using $\gg_g$ and checked for local confluence with other rules in R_g. This process is continued until no new implied constant equalities are generated and local confluence of each R_f is restored.

A subtle issue is when two distinct R_f and R_g, $f \neq g$, have rewrite rules with the same constant on their left sides, i.e., there is a rule $c \to f(M) \in R_f$ and $c \to g(N) \in R_g, f \neq g$. This implies that a constant c is congruent to both nonconstant f-monomial as well as g-monomial. The above can happen if in $\gg_f$ and $\gg_g$, $c \gg_f f(M)$ and $c \gg_g g(N)$, respectively. However, $f(M)$ and $g(N)$ are noncomparable in $\gg_f$ or $\gg_g$ since they have two different AC symbols. We will call this case to be that of *a shared constant having two distinct normal forms in different AC symbols.*

A new constant u is introduced with $c \gg u$, as is usually the case with new constants; $\gg_f$ and $\gg_g$ are extended to include monomials in which u appears with the constraint that $f(M) \gg_f u$ and $g(N) \gg_g u$. Add a rule $c \to u \in R_C$, replace the rule $c \to f(M) \in R_f$ by $f(M) \to u$ and $c \to g(N) \in R_g$ by $g(N) \to u$. These restrictions are easily satisfied.

The replacements of $c \to f(M)$ to $f(M) \to u$ in R_f and similarly of $c \to g(N)$ to $g(N) \to u$ in R_g may violate the local confluence of R_f and R_g. New superpositions are generated in R_f as well as R_g, possibly leading to new rules. After local confluence is restored, the result is new R'_f and R'_g. To illustrate, consider $S = \{c = a + b, c = a * b\}$ with AC $+, *$. For an ordering $c \gg b \gg a$ with both $\gg_+$ and $\gg_*$ being pure lexicographic,

$R_+ = \{c \to a+b\}, R_* = \{c \to a*b\}$. These canonical rewrite systems have a shared constant c with two different normal forms. Introduce a new constant u with $c \gg b \gg a \gg u$; make $R_S = \{a+b \to u, a*b \to u, c \to u\}$. The reader must have observed that whereas in the extended signature, the above rewrite system is unique, reduced, and canonical, on the original signature, it is not even locally confluent. A choice about whether the canonical form of c is an f-monomial or a g-monomial is not made as part a of this algorithm since nonconstant f-monomials and g-monomials are noncomparable.

A reduced canonical rewrite system $R = R_C \cup \bigcup_{f \in F_{AC}} R_f$ is generated from S to compute the AC congruence closure $ACCC(S)$ in which rules have distinct left sides.

Algorithm 2 Combination Algorithm ($S = S_C \ \cup\ \bigcup_{f \in F_{AC}} S_f, \ \ \{\gg_f | f \in f_{AC}\}$).

1. Generate a reduced canonical rewrite system R_C from S_C using the total ordering $\gg$ on C; equivalently, as in step 1 of the algorithm for the single AC symbol, Tarjan's Union-Find data structure, can be employed.
2. Normalize each S_f using R_c, resulting in equations on f-monomials on canonical constants. Wlog, we will continue to call the result S_f. This step is eagerly applied as new constant equalities on constants are generated in the steps below.
3. Run the congruence closure algorithm on each S_f from the previous subsection using the ordering $\gg_f$, generating a reduced canonical rewrite system R_f for the congruence closure $ACCC(S_f)$.
4. If any of R_f's generates an implied constant equality, say $c \to d$, include it in R_C and inter-reduce. If $c \to d$ and any other constant rewrite rule generated from R_C, reduces a rule, say $g(M) \to g(N)$ in any R_g, two cases are considered: (i) $g(M)$ is rewritten using the new rules: move $g(M) = g(N)$ from R_g to ${T_g}^2$, and check for the local confluence of the modified R_g along with T_g by generating new superposition, if any, and adding new rules in R_g. (ii) $g(M)$ is not rewritten but $g(N)$ is, then replace $g(M) \to g(N')$, where $g(N')$ is a normal form of $g(N)$ using R_g.
5. **Shared constant with canonical forms in different AC symbols:**
   If two different canonical rewrite subsystems $R_f, R_g, f \neq g$ have identical constants as the left sides, i.e. if there is a rule $c \to f(M) \in R_f$ and $c \to g(N) \in R_g, f \neq g$, introduce a new constant u, make $c \gg u$ extending $\gg_f$ on f-monomials with u making $f(M) \gg_f u$ and $g(N) \gg_g u$, and add a rule $c \to u \in R_C$, replace rules $c \to f(M) \in R_f$ by $f(M) \to u$ and $c \to g(N) \in R_g$ by $g(N) \to u$. Since u is a new symbol, orderings on f-monomials and g-monomials are extended to satisfy these requirements.
   Replacing $c \to f(M)$ by $f(M) \to u$ can result in additional superpositions with other rules in R_f and possibly new rules using u; this applies to R_g as well. After ensuring local-confluence of all new superpositions and adding new rules if needed, reduced canonical rewrite systems are generated for each R_f.
6. If no new constant equalities are generated and the set of rewrite systems R_f's do not satisfy the shared constant condition, the algorithm terminates.
7. Output the combined rewrite system consisting of a reduced canonical R_C on constants and a reduced canonical R_f for each $f \in F_{AC}$. These canonical Rewrite systems do not share a constant symbol appearing on the left side of any rule.

The termination and correctness of the algorithm follows from the termination and correctness of the algorithm for a single AC symbol and the fact there are finitely many new constant equalities that can be added.

The result of the above algorithm is a finite reduced canonical rewrite system $R_S = R_C \cup \bigcup_{f \in F_{AC}} R_{S_f}$, a disjoint union of sets of reduced canonical rewrite rules on f-monomials for each AC symbol f, along with a canonical rewrite system R_C consisting of constant rules such that the left sides of rules are distinct. R_S is unique in the extended signature assuming a family of total admissible orderings on f-monomials for every $f \in F_{AC}$, extending a total ordering on constants; this assumes a fixed choice of new constants standing for the same set of subterms during purification and flattening. In the original signature, however, R_S is neither unique nor even locally confluent (or canonical) if it includes a shared constant having multiple canonical forms in two different AC symbols as illustrated in the above example. It then becomes necessary to compare monomials in different AC symbols.

▶ **Theorem 9.** *R_S as defined above is a unique reduced canonical rewrite system in the extended signature for a given family $\{\gg_f \mid f \in f_{AC}\}$ of admissible orderings on f-monomials extending a total ordering on constants, such that $ACCC(R_S)$, with rewrite rules in R_S viewed as equations, on the original signature is $ACCC(S)$.*

The proof of the theorem follows from the fact that (i) each R_{S_f} is reduced and canonical, and is unique for S_f using $\gg_f$, (ii) the left sides of all rules are distinct, (iii) these rewrite systems are normalized using R_C.

4 Congruence Closure with Uninterpreted and Multiple AC symbols

The algorithm presented in the previous subsection to compute AC congruence closure with multiple AC symbols is combined with Kapur's congruence closure algorithm for uninterpreted symbols. The combination is straightforward given that the output of the congruence closure algorithm is a unique reduced canonical rewrite system consisting of flat rules of the form $h(a_1, \cdots, a_k) \to b$ and constant rules $a \to b$. There is no interaction between flat rules and other rules generated from AC monomials. When new constant equalities are generated, they can reduce flat rules, making the left sides of some flat rules equal, resulting in additional equalities which are handled in the same way as constant equalities generated during completion on equations on f-monomials. All other steps are the same as in the case of the congruence closure algorithm in the previous subsection for multiple AC symbols. The output of this general algorithm share the properties of the output of the congruence closure over multiple AC symbols.

It is preferable to generate R_C first and then generate R_U to check if any implied constant equalities are generated. The result is a reduced canonical rewrite system $R_C \cup R_U$ for the congruence closure of $S_C \cup S_U$ over uninterpreted symbols. R_C, R_U can be computed very fast in $O(n * log(n))$ steps, whereas computing R_f from a set of f-monomial equations is very expensive, so it always pays off to deduce constant equalities from R_C and R_U. During the computation of generating canonical rewrite systems for R_f from $S_f, f \in F_{AC}$, if a new constant equality is implied and generated, it is eagerly used to update $R_C \cup R_U$ to generate any new implied equalities, and used to update monomial equations and monomial rewrite systems constructed so far.

The algorithm from the previous subsection for computing reduced canonical rewrite systems from each S_f is applied, looking for new constant equalities generated and checking shared constant condition. As discussed in the previous subsection, in both cases, local confluence of R_f's may have to restored for checking additional superpositions, leading to possibly new rules.

The result is a modular combination, whose termination and correctness is established in terms of the termination and correctness of its various components: (i) the termination and correctness of algorithms for generating canonical rewrite systems from ground equations in a single AC symbol, for each AC symbol in F_{AC}, and their combination together with each other, and (ii) the termination and correctness of congruence closure over uninterpreted symbols and its combination with the AC congruence closure for multiple AC symbols.

Given a total ordering $\gg$ on C, let $\gg_U = \gg \cup \{h \gg_U c, h \in F_U, c \in C\}$. For each AC symbol f, define a total admissible ordering $\gg_f$ on f-monomials extending $\gg$ on C.

Algorithm 3 General Congruence Closure($S = S_C \cup S_U \cup \cup_{f \in F_{AC}} S_f, \gg_U \cup \{\gg_f \mid f \in F_{AC}\}$).

1. From $S_C \cup S_U$, generate a reduced canonical rewrite system $R_C \cup R_U$ representing the congruence closure over uninterpreted symbols such that each rule has its left side as $h(c_1, \cdots, c_k) \to c$ or $a \to b$ and no two left sides are the same.
2. Normalize $S_f, f \in F_{AC}$ using R_C.
3. Run the AC congruence closure for multiple AC symbols from the previous subsection, on the output from the previous step, using $\gg_f$ for each $f \in F_{AC}$.
4. If new constant equalities are generated and/or shared constant condition is satisfied, redo steps 1, 2, 3, restoring local confluence of R_C, R_U and each R_f.
5. Repeat this step until no more new constant equalities are generated and until shared constant condition is satisfied, leading to the termination of the algorithm.

Since there are finitely many constants, bounded by the size of input ground equations, only finitely many constant equalities on them can be added during the propagation. As a result, the termination of the general algorithm follows from the termination of the algorithms for each of the components–S_U and S_f, for every AC symbol f. The correctness proof of the general algorithm is also structured in a modular fashion using the correctness proofs of the components S_U and S_f, $f \in F_{AC}$.

► **Theorem 10.** *Given a set S of ground equations in $GT(F)$, the above algorithm generates a reduced canonical rewrite system R_S on the extended signature such that $R_S = R_C \cup R_u \cup \bigcup_{f \in F_{AC}} R_f$, where each of R_C, R_U, R_f is a reduced canonical rewrite system using a set of total admissible monomial orderings $\gg_f$ on f-monomials, which extends a ordering $\gg_U$ on uninterpreted symbols and constants as defined above, and $ACCC(R_S)$, with rules in R_S viewed as equations, on the original signature is $ACCC(S)$. Further, for this given set of orderings $\gg_U$ and $\{\gg_f \mid f \in F_{AC}\}$, R_S is unique for S in the extended signature.*

It should be noted that it is not necessary to run a completion algorithm for each AC symbol separately, instead, they can be interleaved along with any new constant equalities generated to reduce constants appearing in other R_f eagerly. Even though there are new constants introduced in the generation of a reduced canonical rewrite system if two different R_f, R_g have the same constant appearing on the left sides of rules with f-monomials and g-monomials on their right side, the number of choices for canonical forms is finite given that there are only finite many AC symbols and finitely many constants.

5 Examples

The proposed algorithms are illustrated using several examples which are mostly picked from the above cited literature with the goal of not only to show how the proposed algorithms work, but also contrast them with the algorithms reported in the literature.

▶ **Example 1.** Consider an example from [5]: $S = \{f(a,c) = a,\ f(c, g(f(b,c))) = b,\ g(f(b,c)) = f(b,c)\}$ with g being uninterpreted and f being an AC symbol. A number of variations of the same example will be considered.

Mixed term $f(c, g(f(b,c)))$ is purified by introducing new symbols u_1 for $f(b,c)$ and u_2 for $g(u_1)$, gives $\{f(a,c) = a,\ f(b,c) = u_1,\ g(u_1) = u_2,\ f(c,u_2) = b, u_2 = u_1\}$. (i) $S_C = \{u_2 = u_1\}$, (ii) $S_U = \{g(u_1) = u_2\}$, and (iii) $S_f = \{f(a,c) = a, f(b,c) = u_1, f(c,u_2) = b\}$.

Different total orderings on constants are used to illustrate how different canonical forms can be generated. Consider a total ordering $f \gg g \gg a \gg b \gg u_2 \gg u_1$. $R_C = \{1.\ u_2 \to u_1\}$ normalizes the uninterpreted equation and it is oriented as: $R_U = \{2.\ g(u_1) \to u_1\}$.

To generate a reduced canonical rewrite system for AC f-terms, an admissible ordering on f-terms must be chosen. The degree-lexicographic ordering on monomials will be used for simplicity: $f(M_1) \gg_f f(M_2)$ iff $|M_1| > |M_2|$ or $|M_1| = |M_2| \wedge M_1 - M_2$ includes a constant $\gg$ every constant in $M_2 - M_1$. f-equations are normalized using rules 1 and 2, and oriented: $\{3.\ f(a,c) \to a,\ 4.\ f(c,u_1) \to b,\ 5.\ f(b,c) \to u_1\}$.

Applying the AC congruence closure completion algorithm, the superposition between the rules $3, 5$ is $f(a,b,c)$ with the critical pair: $(f(a,b), f(a,u_1))$, leading to a rewrite rule 6. $f(a,b) \to f(a,u_1)$; the superposition between the rules $3, 4$ is $f(a,c,u_1)$ with the critical pair: $(f(a,u_1), f(a,b))$ which is trivial by rule 6. The superposition between the rules $4, 5$ is $f(b,c,u_1)$ with the critical pair: $(f(b,b), f(u_1,u_1))$ giving : 7. $f(b,b) \to f(u_1,u_1)$. The rewrite system $R_f = \{3,4,5,6,7\}$ is a reduced canonical rewrite system for S_f. $R_S = \{1,2\} \cup R_f$ is a reduced canonical rewrite system associated with the AC congruence closure of the input and serves as its decision procedure.

In the original signature, the rewrite system R_S is: $\{g(f(b,c))) \to f(b,c),\ f(a,c) \to a,\ f(b,c,c) \to b,\ f(a,b) \to f(a,b,c),\ f(b,b) \to f(b,b,c,c)\}$ with 5 becoming trivial. The reader would observe this rewrite system is locally confluent but not terminating.

Consider a different ordering: $f \gg g \gg a \gg b \gg u_1 \gg u_2$. This gives rise to a related rewrite system in which u_1 is replaced by u_2: $\{u_1 \to u_2, g(u_2) \to u_2, f(a,c) \to a, f(c,u_2) \to b, f(b,c) \to u_2, f(a,b) \to f(a,u_2), f(b,b) \to f(u_2,u_2)\}$. Compare it in the original signature with the above system: $R_S = \{f(b,c) \to g(f(b,c)), g(g(f(b,c))) \to g(f(b,c)), f(a,c) \to a, f(c, g(f(b,c))) \to b, f(a,b) \to f(a, g(f(b,c))), f(b,b) \to f(g(f(b,c)), g(f(b,c)))\}$.

▶ **Example 2.** This example illustrates interaction between congruence closures over uninterpreted symbols and AC symbols. Let $S = \{g(b) = a, g(d) = c, a * c = c, b * c = b, a * b = d\}$ where g is uninterpreted and $*$ is AC. Let $* \gg g \gg a \gg b \gg c \gg d$ with $g, *$. Applying the steps of the algorithm, R_U, the congruence closure over uninterpreted symbols, is $\{g(b) \to a, g(d) \to c\}$, Completion on the $*$-equations using degree lexicographic ordering, oriented as $\{a * c \to c, b * c \to b, a * b \to d\}$, generates an implied constant equality $b = d$ from the critical pair of $a * c \to c, b * c \to b$. Using the rewrite rule $b \to d$, the AC rewrite system reduces to: $R_* = \{a * c \to c, c * d \to d, a * d \to d\}$, which is canonical.

The implied constant equality $b = d$ is added to $R_C : \{b \to d\}$ and is also propagated to R_U, which makes the left sides of $g(b) \to a$ and $g(d) \to c$ equal, generating another implied equality $a = c$. This equality is oriented and added to R_C: $\{a \to c, b \to d\}$, and R_U becomes $\{g(d) \to c\}$. This implied equality is propagated to the AC rewrite system on $*$. R_C normalizes R_* to $\{c * c \to c, c * d \to d, a * d \to d\}$.

The output of the algorithm is a canonical rewrite system: $\{g(d) \to c, b \to d, a \to c, c*c \to c, c * d \to d\}$. In general, propagation of equalities can result in the left sides of the rules in AC subsystems change, generating new superpositions, much like in the uninterpreted case.

▶ **Example 3.** Consider another example with multiple AC symbols from [24]: $S = \{a+b = a*b, a*c = g(e), e = e'\}$ with $+, *$ being AC and g as uninterpreted, to illustrate flexibility in choosing orderings in our framework.

Purification leads to introduction of new constants : $\{1.\ a+b = u_0,\ 2.\ a*b = u_1,\ 3.\ a*c = u_2, 4.\ g(e) = u_2, 5.\ e = e', 6.\ u_0 = u_1\}$. Depending upon a total ordering on constants a, b, c, e, e', there are thus many possibilities depending upon the desired canonical forms.

Recall that $a+b$ cannot be compared with $a*b$, however u_0 and u_1 can be compared. If canonical forms are desired so that the canonical form of $a+b$ is $a*b$, the ordering should include $u_0 \gg u_1$. Consider an ordering $a \gg b \gg c \gg e \gg e' \gg u_2 \gg u_0 \gg u_1$. Degree-lexicographic ordering is used on $+$ and $*$ monomials. This gives rise to: $R_C = \{6.\ u_0 \to u_1, 5.\ e \to e'\}$, $R_U = \{4'.\ g(e') \to u_2\}$ along with $R_+ = \{1.\ a+b \to u_1\}$ and $R_* = \{2.\ a*b \to u_1, 3.\ a*c \to u_2\}$. The canonical rewrite system for $*$ is: $\{2.\ a*b \to u_1, 3.\ a*c \to u_2, 7.\ b*u_2 \to c*u_1\}$.

The rewrite system $R_S = \{1.\ a+b \to u_1,\ 2.\ a*b \to u_1,\ 2.\ u_0 \to u_1,\ 3.\ a*c \to u_2,\ 4'.\ g(e') \to u_2,\ 5.\ e \to e',\ 6.\ b*u_2 \to c*u_1\}$ is canonical. Both $a+b$ and $a*b$ have the same normal form u_1 standing for $a*b$ in the original signature.

With $u_1 \gg u_0$, another canonical rewrite system $R_S = \{1.\ a+b \to u_0,\ 2.\ a*b \to u_0,\ 6'.\ u_1 \to u_0,\ 3.\ a*c \to u_2,\ 4.\ g(e') \to u_2,\ 5.\ e \to e',\ 6.\ b*u_2 \to c*u_0\}$ in which $a+b$ and $a*b$ have the normal form u_0 standing for $a+b$, different from the one above.

6 A Gröbner Basis Algorithm as an AC Congruence Closure

Buchberger's Gröbner basis algorithm when extended to polynomial ideals over integers [10] can be interestingly viewed as a special congruence closure algorithm with multiple AC symbols $+$ and $*$ which in addition, satisfy the properties of a commutative ring with unit. A manuscript proposing this new perspective on Gröbner basis algorithms with interesting implications is under preparation [15]. Below, we illustrate this new insight using an example from [10]. Relationship between Gröbner basis algorithm and the Knuth-Bendix completion procedure has been investigated in [9, 30, 12, 22], but the proposed insight is novel.

The ring structure of polynomials gives rise to additional interaction when a canonical rewrite system for the congruence closure of $+$-monomials is combined with a canonical rewrite system for the congruence closure of $*$-monomials. Along with the identities for $+$, $x+0 = x$, and for $*$, $x*1 = x$, and the distributivity axiom, $+$ also has an inverse operation: $x + -(x) = 0, -0 = 0, -(-x) = x, -(x+y) = -x + -y$. Abusing the notation, $x + -(y)$ will be written as $x - y$. In addition, $x*0 = 0$. With the distributivity rule: $x*(y+z) = (x*y)+(x*z)$, these axioms when oriented from left to right constitute a canonical rewrite system for a commutative ring with unit and are used for normalizing terms to polynomials.

An additive monomial $c\ t$, where $c \neq 0$, is an abbreviation of repeating $*$-monomial t c times; if c is positive, say 3, then $3\ t$ is an abbreviation for $t+t+t$; similarly if c is negative, say -2, then $-2\ t$ is an abbreviation for $-t-t$. A $*$-monomial with unit coefficient is a pure term expressed in $*$, but a monomial $3\ y*y*y$, for example,is a mixed term $y*y*y+y*y*y+y*y*y$ with $+$ as the outermost symbol which has $*$ subterms.

Consider an example [10]; a related example is also discussed in [22], so an interested reader is invited to contrast the proposed approach with the one there. The input basis is: $7\ x*x*y = 3\ x, 4\ x*y*y = x*y, 3\ y*y*y = 0$ of a polynomial ideal over the integers [10].

Purification of the above equations leads to: 1. $7\ u_1 = 3\ x$, 2. $4\ u_2 = u_3$, 3. $3\ u_4 = 0$ with 4. $x*x*y = u_1$, 5. $x*y*y = u_2$, 6. $x*y = u_3$.7. $y*y*y = u_4$.

Let a total ordering on all constants be: $u_1 \gg u_2 \gg u_4 \gg u_3 \gg x \gg y$. Extend it using the degree-lexicographic ordering on $+$-monomials as well as $*$-monomials, Orienting $*$ equations: $R_* = \{4.\ x*x*y \rightarrow u_1, 5.\ x*y*y \rightarrow u_2, 6.\ x*y \rightarrow u_3, 7.\ y*y*y \rightarrow u_4\}$. Orienting $+$ equations, $R_+ = \{1.\ 7\ u_1 \rightarrow 3\ x, 2.\ 4\ u_2 \rightarrow u_3, 3.\ 3\ u_4 \rightarrow 0\}$.

Completion on the ground equations on $*$ terms generates a reduced canonical rewrite system: $R_* = \{6.\ x*y \rightarrow u_3, 7.\ y*y*y \rightarrow u_4, 8.\ u_3*y \rightarrow u_2, 9.\ u_3*x \rightarrow u_1, 10.\ u_2*x \rightarrow u_3*u_3, 11.\ u_1*y \rightarrow u_3*u_3, 12.\ u_1*u_4 \rightarrow u_2*u_2, 13.\ u_4*x*x \rightarrow u_2*u_3, 14.\ u_3*u_3*u_3 \rightarrow u_1*u_2\}$. This system captures relationships among all product monomials appearing in the input. The canonical rewrite system for $+$ is: $R_+ = \{1.\ 7\ u_1 \rightarrow 3\ x, 2.\ 4\ u_2 \rightarrow u_3, 3.\ 3\ u_4 \rightarrow 0\}$.

Rules in R_+ and R_* interact, leading to new superpositions and critical pairs: for $c\ u \rightarrow r \in R_+, u*m \rightarrow r' \in R_*$, where $c \in \mathbb{Z} - \{0\}$, u is a constant and m is *-monomial, the superposition is $(c\ u)*m$ generating the critical pair $(r*m, c\ r')$, which is normalized using distributivity and other rules. It can be shown that only this superposition needs to be considered to check local confluence of $R_+ \cup R_*$[15].

As an example, rules $3:\ \ 3\ u_4 \rightarrow 0$ and $12:\ \ u_1*u_4 \rightarrow u_2*u_2$ give the superposition $u_1*u_4+u_1*u_4+u_1*u_4$, which is $3u_1*u_4$, generating the critical pair $(3\ u_2*u_2, 0)$; rules 3 and 13 gives a trivial critical pair. Considering all such superpositions, the resulting canonical rewrite system can be shown to include $\{3\ x \rightarrow 0, u_1 \rightarrow 0, u_2 \rightarrow u_3, 3\ u_4 \rightarrow 0\}$ among other rules. When converted into the original signature, this is precisely the Gröbner basis reported as generated using Kandri-Rody and Kapur's algorithm: $\{3\ x \rightarrow 0, x*x*y \rightarrow 0, x*y*y \rightarrow x*y, 3\ y*y*y \rightarrow 0\}$ as reported in [10].

The above illustrates the power and elegance of the proposed combination framework. Comparing with [22], it is reported there that a completion procedure using normalized rewriting generated 90 critical pairs in contrast to AC completion procedure [27] computed 1990 critical pairs; in the proposed approach, much fewer critical pairs are generated in contrast, without needing to use any AC unification algorithm or extension rules.

The proposed approach can also be used to compute Gröbner basis of polynomial ideals with coefficients over finite fields such as $\mathbb{Z}_p$ for a prime number p, as well as domain with zero divisor such as $\mathbb{Z}_4$ [16]. For example, in case of $\mathbb{Z}_5$, another rule is added: $1+1+1+1+1 \rightarrow 0$ added to the input basis.

7 Conclusion

A modular algorithm for computing the congruence closure of ground equations expressed using AC function symbols and uninterpreted symbols is presented. The algorithm is derived by generalizing the framework first presented in [13] for generating the congruence closure of ground equations over uninterpreted symbols. The key insight from [13]–flattening of subterms by introducing new constants and congruence closure on constants, is generalized by flattening mixed AC terms and purifying them by introducing new constants to stand for pure AC terms in every (single) AC symbol. The result of this transformation on a set of equations on mixed ground terms is a set of constant equations, a set of flat equations relating a nonconstant term in an uninterpreted symbol to a constant, and a set of equations on pure AC terms in a single AC symbol. Such decomposition and factoring enable using congruence closure algorithms for each of the subproblems independently, which propagate equalities on shared constants. Once the propagation of constant equalities stabilizes (reaches a fixed point), the result is (i) unique reduced canonical rewrite systems for each subproblem and finally, (ii) a unique reduced canonical rewrite system for the congruence closure of a finite set of ground equations over multiple AC symbols and uninterpreted symbols. The algorithms extend easily when AC symbols have additional properties such as idempotency, identity and nilpotency.

The modularity of the algorithms leads to easier and simpler correctness and termination proofs in contrast to those in [5, 22]. The complexity of the procedure is governed by the complexity of generating a canonical rewrite system for AC ground equations on constants.

The proposed algorithm is a direct generalization of Kapur's algorithm for the uninterpreted case, which has been shown to be efficiently integrated into SMT solvers including BarcelogicTools [26]. We believe that the AC congruence closure can also be effectively integrated into SMT solvers. Unlike other proposals, the proposed algorithm neither uses specialized AC compatible orderings on nonground terms nor extension rules often needed in AC/E completion algorithms and AC/E-unification, thus avoiding explosion of possible critical pairs for consideration.

A by-product of this new algorithm based on the proposed framework is a new way to view a Gröbner basis algorithm for polynomial ideals over integers, as a congruence closure algorithm over a commutative ring with unit, which is a congruence closure algorithm with two AC symbols $+$ and $*$, extended to consider the additional properties of $+$ and $*$.

References

1. Franz Baader and Deepak Kapur. Deciding the word problem for ground identities with commutative and extensional symbols. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning – 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, volume 12166 of *Lecture Notes in Computer Science*, pages 163–180. Springer, 2020. doi:10.1007/978-3-030-51074-9_10.
2. Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.
3. Leo Bachmair and Nachum Dershowitz. Critical pair criteria for completion. *J. Symb. Comput.*, 6(1):1–18, 1988. doi:10.1016/S0747-7171(88)80018-X.
4. Leo Bachmair, I. V. Ramakrishnan, Ashish Tiwari, and Laurent Vigneron. Congruence closure modulo associativity and commutativity. In Hélène Kirchner and Christophe Ringeissen, editors, *Frontiers of Combining Systems, Third International Workshop, FroCoS 2000, Nancy, France, March 22-24, 2000, Proceedings*, volume 1794 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 2000. doi:10.1007/10720084_16.
5. Leo Bachmair, Ashish Tiwari, and Laurent Vigneron. Abstract congruence closure. *J. Autom. Reason.*, 31(2):129–168, 2003. doi:10.1023/B:JARS.0000009518.26415.49.
6. A. Michael Ballantyne and Dallas Lankford. New decision algorithms for finitely presented commutative semigroups. *Computers and Mathematics Applications*, 7:159–165, 1981.
7. Thomas Becker, Volker Weispfenning, and Heinz Kredel. *Gröbner bases – a computational approach to commutative algebra*, volume 141 of *Graduate texts in mathematics*. Springer, 1993.
8. Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, 1980. doi:10.1145/322217.322228.
9. Abdelilah Kandri-Rody and Deepak Kapur. On relationship between buchberger's gröbner basis algorithm and the knuth-bendix completion procedure. Technical report no. ge-83crd286, General Electric Corporate Research and Development, Schenectady, NY, November 1983.
10. Abdelilah Kandri-Rody and Deepak Kapur. Computing a gröbner basis of a polynomial ideal over a euclidean domain. *J. Symb. Comput.*, 6(1):37–57, 1988. doi:10.1016/S0747-7171(88)80020-8.
11. Abdelilah Kandri-Rody, Deepak Kapur, and Paliath Narendran. An ideal-theoretic approach to work problems and unification problems over finitely presented commutative algebras. In Jean-Pierre Jouannaud, editor, *Rewriting Techniques and Applications, First International Conference, RTA-85, Dijon, France, May 20-22, 1985, Proceedings*, volume 202 of *Lecture Notes in Computer Science*, pages 345–364. Springer, 1985. doi:10.1007/3-540-15976-2_17.

12 Abdelilah Kandri-Rody, Deepak Kapur, and Franz Winkler. Knuth-bendix procedure and buchberger algorithm: A synthesis. In Gaston H. Gonnet, editor, *Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation, ISSAC '89, Portland, Oregon, USA, July 17-19, 1989*, pages 55–67. ACM, 1989. `doi:10.1145/74540.74548`.

13 Deepak Kapur. Shostak's congruence closure as completion. In Hubert Comon, editor, *Rewriting Techniques and Applications, 8th International Conference, RTA-97, Sitges, Spain, June 2-5, 1997, Proceedings*, volume 1232 of *Lecture Notes in Computer Science*, pages 23–37. Springer, 1997. `doi:10.1007/3-540-62950-5_59`.

14 Deepak Kapur. Conditional congruence closure over uninterpreted and interpreted symbols. *J. Syst. Sci. Complex.*, 32(1):317–355, 2019. `doi:10.1007/s11424-019-8377-8`.

15 Deepak Kapur. Weird gröbner bases: An application of associative-commutative congruence closure algorithm. Tech report under preparation, Department of Computer Science, University of New Mexico, May 2021.

16 Deepak Kapur and Yongyang Cai. An algorithm for computing a gröbner basis of a polynomial ideal over a ring with zero divisors. *Math. Comput. Sci.*, 2(4):601–634, 2009. `doi:10.1007/s11786-009-0072-z`.

17 Deepak Kapur, David R. Musser, and Paliath Narendran. Only prime superpositions need be considered in the knuth-bendix completion procedure. *J. Symb. Comput.*, 6(1):19–36, 1988. `doi:10.1016/S0747-7171(88)80019-1`.

18 Deepak Kapur and Paliath Narendran. Double-exponential complexity of computing a complete set of ac-unifiers. In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS '92), Santa Cruz, California, USA, June 22-25, 1992*, pages 11–21. IEEE Computer Society, 1992. `doi:10.1109/LICS.1992.185515`.

19 Deepak Kapur and Hantao Zhang. An overview of rewrite rule laboratory (rrl). *Computers and Mathematics Applications*, 29:91–114, 1995.

20 Donald Knuth and Peter Bendix. Simple word problems in universal algebras. In Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

21 Claude Marché. On ground ac-completion. In Ronald V. Book, editor, *Rewriting Techniques and Applications, 4th International Conference, RTA-91, Como, Italy, April 10-12, 1991, Proceedings*, volume 488 of *Lecture Notes in Computer Science*, pages 411–422. Springer, 1991. `doi:10.1007/3-540-53904-2_114`.

22 Claude Marché. Normalized rewriting: An alternative to rewriting modulo a set of equations. *J. Symb. Comput.*, 21(3):253–288, 1996. `doi:10.1006/jsco.1996.0011`.

23 Ernst W. Mayr and Albert Meyer. The complexity of the word problems for commutative semigroups and polynomial ideals. *Advances in Mathematics*, 46:305–439, 1982.

24 Paliath Narendran and Michaël Rusinowitch. Any gound associative-commutative theory has a finite canonical system. In Ronald V. Book, editor, *Rewriting Techniques and Applications, 4th International Conference, RTA-91, Como, Italy, April 10-12, 1991, Proceedings*, volume 488 of *Lecture Notes in Computer Science*, pages 423–434. Springer, 1991. `doi:10.1007/3-540-53904-2_115`.

25 Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980. `doi:10.1145/322186.322198`.

26 Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Inf. Comput.*, 205(4):557–580, 2007. `doi:10.1016/j.ic.2006.08.009`.

27 Gerald E. Peterson and Mark E. Stickel. Complete sets of reductions for some equational theories. *J. ACM*, 28(2):233–264, 1981. `doi:10.1145/322248.322251`.

28 Robert E. Shostak. An algorithm for reasoning about equality. *Commun. ACM*, 21(7):583–585, 1978. `doi:10.1145/359545.359570`.

29 Franz Winkler. *Canonical forms in the finitely presented algebras*. Ph.d. dissertation, U. of Paris 11, 1983.

30 Franz Winkler. *The Church-Rosser Property in Computer Algebra and Special Theorem Proving: An Investigation of Critical-Pair/Completion Algorithms.* Ph.d. dissertation, University of Linz, 1984.

31 Chee-Keng Yap. A new lower bound construction for commutative thue systems with aapplications. *J. Symb. Comput.*, 12(1):1–28, 1991. doi:10.1016/S0747-7171(08)80138-1.

32 Hantao Zhang. Implementing contextual rewriting. In Michaël Rusinowitch and Jean-Luc Rémy, editors, *Conditional Term Rewriting Systems, Third International Workshop, CTRS-92, Pont-à-Mousson, France, July 8-10, 1992, Proceedings*, volume 656 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 1992. doi:10.1007/3-540-56393-8_28.

A Appendix: Proofs

▶ **Lemma 5.** *An AC rewrite system R_S is* locally confluent *iff the critical pair:* $(f((AB - A_1) \cup A_2), f((AB - B_1) \cup B_2))$ *between every pair of distinct rules* $f(A_1) \to f(A_2), f(B_1) \to f(B_2)$ *is joinable, where* $AB = (A_1 \cup B_1) - (A_1 \cap B_1)$.

Proof. Consider a flat term $f(C)$ rewritten in two different ways in one step using not necessarily distinct rules: $f(A_1) \to f(A_2), f(B_1) \to f(B_2)$. The result of the rewrites is: $(f((C - A_1) \cup A_2), f((C - B_1) \cup B_2))$. Since $A_1 \subseteq C$ as well as $B_1 \subseteq C$, $AB \subseteq C$; let $D = C - AB$. The critical pair is then $(f(D \cup ((AB - A_1) \cup A_2))f(D \cup ((AB - B_1) \cup B_2)))$, all rules applicable to the critical pair to show its joinability, also apply, thus showing the joinability of the pair. The other direction is straightforward. The case of when at least one of the rules has a constant on its left side is trivially handled. ◀

▶ **Theorem 6.** *The algorithm* **1AC-Completion** *terminates, i.e., in Step 4, rules to R_f cannot be added infinitely often.*

Proof. Proof by Contradiction. A new rule $\hat{l} \to \hat{r}$ in Step 4 of the algorithm is added to R_f only if no other rule can reduce it, i.e, for every rule $l \to r \in R_f$, $\hat{l}$ and l are noncomparable in $\gg_D$. For R_f to be infinite, implying the nontermination of the algorithm means that R_f must include infinitely many noncomparable left sides in $\gg_D$, a contradiction to Dickson's Lemma. ◀

▶ **Theorem 8.** *Given a total ordering $\gg_f$ on f-monomials, there is a unique reduced canonical rewrite system associated with S_f.*

Proof. Proof by Contradiction. Suppose there are two distinct reduced canonical rewrite systems R_1 and R_2 associated with S_f for the same $\gg_f$. Pick the least rule $l \to r$ in $\gg_f$ on which R_1 and R_2 differ; wlog, let $l \to r \in R_1$. Given that R_2 is a canonical rewrite system for S_f and $l = R \in ACCC(S_f)$, l and r must reduce using R_2 implying that there is a rule $l' \to r' \in R_2$ such that $l \gg_D l'$; since R_1 has all the rules of R_2 smaller than $l \to r$, $l \to r$ can be reduced in R_1, contradicting the assumption that R_1 is reduced. If $l \to r' \in R_2$ where $r' \neq r$ but $r' \gg_f r$, then r' is not reduced implying that R_2 is not reduced. ◀

▶ **Lemma 5Idem.** *An AC rewrite system R_S with $f(x, x) = x$, is* locally confluent *iff (i) the critical pair:* $(f((AB - A_1) \cup A_2), f((AB - B_1) \cup B_2))$ *between every pair of distinct rules* $f(A_1) \to f(A_2), f(B_1) \to f(B_2)$ *is joinable, where* $AB = (A_1 \cup B_1) - (A_1 \cap B_1)$*, and (ii) for every rule* $f(M) \to f(N) \in R_S$ *and for every constant* $a \in M$*, the critical pair,* $(f(M), f(N \cup \{\{a\}\})$*, is joinable.*

Proof. Consider a flat term $f(C)$ rewritten in two different ways in one step using not necessarily distinct rules and/or $f(x, x) \to x$. There are three cases: (i) $f(C)$ is rewritten in two different ways in one step using $f(x, x) \to x$ to $f(C - \{\{a\}\})$ and $f(C - \{\{b\}\})$. After single step rewrites, the idempotent rule can be applied again on both sides giving $f(C - \{\{a, b\}\})$.

(ii) $f(C)$ is rewritten in two different ways, with one step using $f(x,x) \to x$ and another using $f(M) \to f(N)$. An application of the idempotent rule implies that C includes a constant a, say, at least twice; the result of one step rewriting is: $(f(C - \{\{a\}\}), f((C - M) \cup N)$. This implies there exists a multiset A such that $C = A \cup M \cup \{\ a\}\}$. The critical pair generated from $f(M) \to f(N)$ is $(f(M), f(N \cup \{\{a\}\})$. The rewrite steps used to show the joinability of $(f(M), f(N \cup \{\{a\}\})$ apply also on $(f(C - \{\{a\}\}), f((C - M) \cup N)$, showing joinability. The third case is the same as that of Lemma 5 and is omitted. ◀

Similar local confluence lemmas can be proved in case f is nilpotent, has unit and various combinations.

Derivation of a Virtual Machine For Four Variants of Delimited-Control Operators

Maika Fujii
Ochanomizu University, Tokyo, Japan

Kenichi Asai
Ochanomizu University, Tokyo, Japan

Abstract

This paper derives an abstract machine and a virtual machine for the λ-calculus with four variants of delimited-control operators: `shift/reset`, `control/prompt`, $\texttt{shift}_0/\texttt{reset}_0$, and $\texttt{control}_0/\texttt{prompt}_0$. Starting from Shan's definitional interpreter for the four operators, we successively apply various meaning-preserving transformations. Both trails of invocation contexts (needed for `control` and $\texttt{control}_0$) and metacontinuations (needed for $\texttt{shift}_0$ and $\texttt{control}_0$) are defunctionalized and eventually represented as a list of stack frames. The resulting virtual machine clearly models not only how the control operators and captured continuations behave but also when and which portion of stack frames is copied to the heap.

2012 ACM Subject Classification Theory of computation → Control primitives; Theory of computation → Lambda calculus; Theory of computation → Operational semantics; Theory of computation → Abstract machines; Software and its engineering → Virtual machines

Keywords and phrases delimited-control operators, functional derivation, CPS transformation, defunctionalization, abstract machine, virtual machine

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.16

Supplementary Material *Software (Source Code)*: `https://github.com/FujiiMaika/fscd21`
archived at `swh:1:dir:e523c86111370f0dce57a8b6c5506fcf7c35c1f1`

Funding *Kenichi Asai*: supported in part by JSPS KAKENHI under Grant No. JP18H03218.

Acknowledgements We are grateful to Youyou Cong and anonymous reviewers for their valuable comments and suggestions.

1 Introduction

Manipulation of control structure of a program is inevitable. In addition to the standard exception handling, more sophisticated manipulation of control using algebraic effects and handlers has been proposed [4, 25] and is becoming widely used [20]. To support such mechanisms in a compiler, one can either (i) transform the source program into continuation-passing style (CPS), or (ii) implement manipulation of control directly via the modification of a portion of a stack without transforming the program into CPS. There is extensive research comparing which approach (among more variants) is better in which circumstances [12].

However, for four variants of delimited-control operators, i.e., `shift` and `reset` [8, 9], `control` and `prompt` [13], $\texttt{shift}_0$ and $\texttt{reset}_0$ [23], and $\texttt{control}_0$ and $\texttt{prompt}_0$ [16], almost no low-level implementation has been considered. The only exceptions we are aware of are all on `shift/reset`: direct implementation of `shift/reset` in Scheme48 [15], in OchaCaml [22], and the derivation of a virtual machine for `shift/reset` [3]. Without proper low-level implementation strategies for all the four delimited-control operators, we cannot even discuss pros and cons of CPS vs. direct-style implementations for those operators. This omission could affect the low-level implementation strategies for algebraic effects and handlers, since they have a close connection with $\texttt{shift}_0$ and $\texttt{control}_0$ [14, 24].

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi; Article No. 16; pp. 16:1–16:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper, we derive an abstract machine and a virtual machine for the λ-calculus with four delimited-control operators. Starting from Shan's definitional interpreter [28], we successively apply various meaning-preserving transformations, following Danvy's recipe [2, 7]. The overall derivation is similar to our previous work [3] on deriving a virtual machine for `shift/reset`. However, handling of invocation contexts (needed for `control` and $\texttt{control}_0$) and metacontinuations (needed for $\texttt{shift}_0$ and $\texttt{control}_0$) is non-trivial: we need to have a trail of invocation contexts to be a tree structure to support concatenation of invocation contexts and have a metacontinuation to maintain a list of code pointers representing the contexts outside delimiters.

In summary, we make the following contributions in this paper:

- We present the first virtual machine that supports four delimited-control operators and that explains how they manipulate stacks.
- We show it is possible to apply Danvy's method of inter-deriving semantic artifacts to four delimited-control operators, giving another non-trivial example and widening its applicability.
- We clarify how trails and metacontinuations can be represented in a stack, suggesting a low-level implementation strategy for four delimited-control operators.

After introducing four delimited-control operators in the next section, we first show the definitional interpreter in Section 3. We then apply various program transformation to obtain a stack-based interpreter in Section 4, showing an abstract machine in passing. In Sections 5 and 6, we derive a compiler and a virtual machine. Related work is discussed in Section 7 and the paper concludes in Section 8. The appendix shows an example how a program is compiled to a list of instructions and executed on the virtual machine. The omitted OCaml code is available as supplementary material.

2 Four Delimited-Control Operators

Delimited-control operators enable us to capture the current continuation up to the enclosing delimiter and use it in the subsequent program. There are four variants of delimited-control operators: `shift` ($\mathcal{S}$) and `reset` [8, 9], `control` ($\mathcal{F}$) and `prompt` [13], $\texttt{shift}_0$ ($\mathcal{S}_0$) and $\texttt{reset}_0$ [23], and $\texttt{control}_0$ ($\mathcal{F}_0$) and $\texttt{prompt}_0$ [16]. Since the behavior of all the four delimiters (`reset`, `prompt`, $\texttt{reset}_0$, and $\texttt{control}_0$) are exactly the same, we use a uniform notation $\langle\rangle$ for them. The basic behavior of the four operators are to capture the current continuation up to the enclosing delimiter and execute their body. We describe their exact behavior below.

A `shift` expression, $\mathcal{S}c.\, e$, clears the current continuation up to the enclosing delimiter, binds it to c, and execute e. Thus, in $1 + \langle(\mathcal{S}c.\, 2 \times c\, 3) + 4\rangle$, the continuation $\langle[] + 4\rangle$ is cleared, bound to c, and $2 \times c\, 3$ is executed in `reset`. The original expression reduces to $1 + \langle 2 \times c\, 3\rangle$, giving the final result 15.

A `control` expression, $\mathcal{F}c.\, e$, differs from `shift` in that it does not insert a delimiter into the captured continuation. In $1 + \langle(\mathcal{F}c.\, 2 \times c\, 3) + 4\rangle$, c is bound to $[] + 4$ without surrounding `reset`. If the captured continuation contains another `control`, as in $1 + \langle(\mathcal{F}c.\, 2 \times c\, 3) + \mathcal{F}c'.\, 4\rangle$, c is bound to $[] + \mathcal{F}c'.\, 4$. The original expression reduces to $1 + \langle 2 \times (3 + \mathcal{F}c'.\, 4)\rangle$, where the second $\mathcal{F}$ captures (and discards) not just $3 + []$ but also the *invocation context* of c, namely $2 \times []$, giving the final result 5. Using $\mathcal{F}$, one can access the context in which the captured continuation is invoked. This is in contrast to the `shift` case: $1 + \langle(\mathcal{S}c.\, 2 \times c\, 3) + \mathcal{S}c'.\, 4\rangle$ reduces to $1 + \langle 2 \times \langle 3 + \mathcal{S}c'.\, 4\rangle\rangle$, giving the final result 9. Using more than one $\mathcal{F}$ in the same context, we can capture multiple invocation contexts.[1] To account for the invocation contexts of captured continuations, an interpreter for $\mathcal{F}$ must maintain a *trail* of continuations [5].

A $\texttt{shift}_0$ expression, $\mathcal{S}_0 c.\, e$, on the other hand, removes the original `reset` surrounding the $\texttt{shift}_0$ expression (but retains the `reset` around the captured continuation as in $\mathcal{S}$). By nesting $\mathcal{S}_0$, one can access the context outside the enclosing `reset`. For example, $\langle 1 + \langle(\mathcal{S}_0 c.\, \mathcal{S}_0 c'.\, 2 \times c'3) + 4\rangle\rangle$ reduces to $\langle 1 + (\mathcal{S}_0 c'.\, 2 \times c'3)\rangle$ where c is bound to $\langle[\,] + 4\rangle$ but is discarded. Note that there is no `reset` around $\mathcal{S}_0 c'.\, 2 \times c'3$. Thus, c' is bound to the context $\langle 1 + [\,]\rangle$, which was outside the original `reset`, giving the final result 8. This is in contrast to the `shift` case: $\langle 1 + \langle(\mathcal{S} c.\, \mathcal{S} c'.\, 2 \times c'3) + 4\rangle\rangle$ reduces to $\langle 1 + \langle(\mathcal{S} c'.\, 2 \times c'3)\rangle\rangle$. Now, c' is bound to an empty context $[\,]$, giving the final result 7. With more nested occurrences of $\mathcal{S}_0$, arbitrarily outer contexts can be captured. To account for hierarchical contexts, the interpreter for $\mathcal{S}_0$ must maintain a *metacontinuation* [23].

A $\texttt{control}_0$ expression, $\mathcal{F}_0 c.\, e$, has both the characteristics of $\mathcal{F}$ and $\mathcal{S}_0$: the captured continuation does not come with a surrounding `reset` and the original `reset` is removed. As such, the interpreter for $\mathcal{F}_0$ must maintain both a trail of continuations and a metacontinuation.

Shan [28] provides a detailed explanation on the difference between the four control operators, as well as an example where the choice of the four operators results in four different result values. Dyvbig, Peyton Jones, and Sabry [11] explain the four delimited-control operators in terms of different primitive control operators.

3 The Definitional Interpreter

Listing 1 shows the definitional interpreter for the λ-calculus extended with four delimited-control operators and the delimiter, written in OCaml. The interpreter is written in continuation-, trail-, and metacontinuation-passing style. Although the main interpreter function `f1` receives a trail and a metacontinuation explicitly, they do not play any roles for the pure λ-calculus terms. If we η-reduce them, the definition coincides with the standard continuation-passing style interpreter.

As in our previous work [3], an environment is represented as two lists, a list of variable names `xs` and a list of values `vs`, instead of an association list. This design comes from the goal of this work. Since we will decompose the interpreter into a compiler and a virtual machine, we separate an environment into the part that depends only on the input term and the part that depends on runtime values. The function `Env.offset` returns the offset of a variable within a given list.

In the interpreter, the current continuation and trail in the innermost surrounding delimiter are stored in the arguments `c` and `t` (of types `c` and `t`, respectively), while the continuations and trails outside the delimiter are stored in metacontinuation `m`, which is a list[2] of pairs of a continuation and a trail of each context. Thus, the context is delimited (in the `Reset (e)` case) by storing `c` and `t` to `m` and evaluating the body `e` in the initial continuation `idc` and the empty trail `TNil`.

To capture the current continuation and trail, one of four control operators is used. In all four cases, the current continuation `c` and trail `t` are captured, bound to `x`, and the body of the control operator is evaluated under appropriate settings.

---

[1] See [6] for the general case as well as other (typed) examples of the use of $\mathcal{F}$.

[2] We use `MNil` and `MCons` to construct metacontinuations. We cannot use `(c * t) list` as the definition of `m`, because the types `c` and `m` would then be circular.

Listing 1 The definitional interpreter.

```
(* syntax *)
type e = Var of string | Fun of string * e | App of e * e
       | Shift of string * e | Control of string * e
       | Shift0 of string * e | Control0 of string * e
       | Reset of e

type v = VFun of (v -> c -> t -> m -> v)      (* value *)
       | VContS of c * t | VContC of c * t
and  c = v -> t -> m -> v                      (* continuation *)
and  t = TNil | Trail of (v -> t -> m -> v)   (* trail *)
and  m = MNil | MCons of (c * t) * m          (* metacontinuation *)

(* initial continuation : v -> t -> m -> v *)
let idc v t m = match t with
    TNil -> (match m with
        MNil -> v
      | MCons((c,t),m) -> c v t m)
  | Trail(h) -> h v TNil m

(* cons : (v -> t -> m -> v) -> t -> t *)
let rec cons h t = match t with
    TNil -> Trail(h)
  | Trail(h') -> Trail(fun v t' m -> h v (cons h' t') m)

(* apnd : t -> t -> t *)
let apnd t0 t1 = match t0 with
    TNil -> t1
  | Trail(h) -> cons h t1

(* f1 : e -> string list -> v list -> c -> t -> m -> v *)
let rec f1 e xs vs c t m = match e with
    Var(x) -> c (List.nth vs (Env.offset x xs)) t m
  | Fun(x,e) ->
      c (VFun(fun v c' t' m' -> f1 e (x::xs) (v::vs) c' t' m')) t m
  | App(e0,e1) ->
      f1 e0 xs vs (fun v0 t0 m0 ->
      f1 e1 xs vs (fun v1 t1 m1 ->
      (match v0 with
         VFun(f) -> f v1 c t1 m1
       | VContS(c',t') -> c' v1 t' (MCons((c,t1),m1))
       | VContC(c',t') -> c' v1 (apnd t' (cons c t1)) m1)) t0 m0) t m
  | Shift(x,e) -> f1 e (x::xs) (VContS(c,t)::vs) idc TNil m
  | Control(x,e) -> f1 e (x::xs) (VContC(c,t)::vs) idc TNil m
  | Shift0(x,e) -> (match m with
      MCons((c0,t0),m0) -> f1 e (x::xs) (VContS(c,t)::vs) c0 t0 m0)
  | Control0(x, e) -> (match m with
      MCons((c0,t0),m0) -> f1 e (x::xs) (VContC(c,t)::vs) c0 t0 m0)
  | Reset(e) -> f1 e xs vs idc TNil (MCons((c,t),m))

(* f : e -> v *)
let f expr = f1 expr [] [] idc TNil MNil
```

- For `Shift (x, e)` and `Control (x, e)`, the body `e` is evaluated under the initial continuation and the empty trail. This reflects the fact that the original `reset` surrounding the control operator remains for these cases. Even if we use control operators within `e`, we cannot access the contexts outside `reset` because they reside in `m`.
- For `Shift0 (x, e)` and `Control0 (x, e)`, on the other hand, the body `e` is evaluated under the topmost continuation and trail stored in the metacontinuation `m`.[3] This reflects the fact that the original `reset` surrounding the control operator is removed for these cases. By using control operators within `e`, we can access the context outside the innermost `reset`.

The captured continuation and trail are packaged into `VContS` for `Shift (x, e)` and `Shift0 (x, e)` and into `VContC` for `Control (x, e)` and `Control0 (x, e)`. When `VContS` or `VContC` is applied (in the `App` case), it behaves differently depending on whether `reset` is present around the invocation.

- For `VContS (c', t')`, the continuation `c` and trail `t1` at the invocation time are pushed into metacontinuation `m1`. This reflects the fact that the invocation of a continuation captured by `Shift (x, e)` or `Shift0 (x, e)` is surrounded by `reset`. Even if we use control operators within `c'`, we cannot access `c` and `t1` because they reside in the metacontinuation.
- For `VContC (c', t')`, on the other hand, the continuation `c` and trail `t1` at the invocation time are concatenated to the current trail `t'`. This reflects the fact that the invocation of a continuation captured by `Control (x, e)` or `Control0 (x, e)` is *not* surrounded by `reset`; since the invocation-time continuation and trail are put into the trail, they can be captured by using control operators within `c'`.

Adding a continuation to a trail and appending two trails are realized by `cons` and `apnd`, respectively. A trail is either an empty trail `TNil` or a non-empty trail `Trail` holding a continuation, which represents functional composition of all the invocation contexts (continuations) encountered so far.

The interpreter is identical to Shan's interpreter [28] except for two points. First, Shan uses higher-order functions directly to represent captured continuations, while we use a defunctionalized form. We could have started from the higher-order functions; by applying defunctionalization to it, we obtain Listing 1. Second, Shan concatenates the captured continuation `c'` and trail `t'` with the continuation `c` and trail `t1` at the invocation time as `((cons c' t') v1 (cons c t1))`. By case analysis on `t'`, it is straightforward to verify that Shan's code is equivalent to `(c' v1 (apnd t' (cons c t1)))` which we adopt. The latter is also used by Biernacki, Danvy, and Millikin [5] and Kameyama and Yonezawa [19].

4 Stack Introduction

In this and next sections, we successively apply meaning-preserving program transformations to the definitional interpreter to obtain a compiler and a virtual machine. In this section, we introduce a stack into the interpreter by (1) defunctionalizing continuations (Section 4.1), (2) linearizing them into a list of frames (Section 4.2), and (3) separating static and dynamic data in the frames (Section 4.3). Along the way, we derive a stack-based abstract machine (Section 4.5).

[3] Metacontinuation `m` must be non-empty here. Otherwise, a pattern-match error is raised. (In the supplementary material, a more sensible error message "shift0/control0 is used without enclosing reset" is given.)

Listing 2 Type definition for defunctionalized interpreter.

```
type v = VFun of (v -> c -> t -> m -> v)
       | VContS of c * t | VContC of c * t
and  c = C0 | CApp0 of e * string list * v list * c | CApp1 of v * c
and  t = TNil | Trail of (v -> t -> m -> v)
and  m = MNil | MCons of (c * t) * m
```

Listing 3 Type definition for linearized interpreter.

```
type v = VFun of (v -> c -> t -> m -> v)
       | VContS of c * t | VContC of c * t
and  f = CApp0 of e * string list * v list | CApp1 of v
and  c = f list
and  t = TNil | Trail of (v -> t -> m -> v)
and  m = MNil | MCons of (c * t) * m
```

4.1 Defunctionalization

We first defunctionalize [26, 27] continuations in the definitional interpreter. In Listing 1, the type `c` is higher order. We turn it into a datatype as shown in Listing 2. The identity continuation is represented as `C0`, while two continuations in the `App` case are represented as `App0` and `App1` where the arguments represent free variables of the respective continuations. The resulting datatype essentially represents evaluation contexts.

We do *not* defunctionalize the argument of `VFun` at this point, because it is not necessary for stack introduction. This choice is arbitrary: we could defunctionalize it and the rest of derivations would go through without any problem. We will defunctionalize it later when we need to do so, to derive an abstract machine and a virtual machine.

We do *not* defunctionalize the argument of `Trail`, either. Even though the type of the argument of `Trail` is the same as `c`, defunctionalizing it together with `c` leads to tree-structured continuations. We can still obtain the same abstract machine and virtual machine, but by defunctionalizing it separately at a later stage, we can keep the definition of `c` to have a list structure (as in our previous work [3]) and postpone the introduction of a tree structure until Section 5.3.

We omit the standard definition of the defunctionalized interpreter due to the lack of space; see the supplementary material. We simply introduce a dispatch function for `c` and use it whenever a continuation is applied. The transformation is the standard defunctionalization and thus the resulting interpreter behaves the same as the definitional interpreter.

4.2 Linearizing Continuations

The type `c` in Listing 2 is isomorphic to a list where `C0` is an empty list and `CApp0` and `CApp1` are conses. Thus, we linearize continuations, i.e., we transform `c` into an OCaml list as shown in Listing 3. The type `c` is now a list of frames, where a frame `f` stores data that were previously held in `CApp0` and `CApp1`.

Obviously, the new interpreter (omitted) behaves the same as the previous one.

4.3 Introducing Stacks

Examining the type `f` in Listing 3, we notice that the constructors `CApp0` and `CApp1` contain both static (compile-time) and dynamic (run-time) data. Static data include the term `e` and the variable list `string list` in `CApp0`, which are fixed once the input program

Listing 4 Type definition for stack-based interpreter.

```
type v = VFun of (v -> c -> s -> t -> m -> v)
       | VContS of c * s * t | VContC of c * s * t
       | VEnv of v list
and  f = CApp0 of e * string list | CApp1
and  c = f list
and  s = v list
and  t = TNil | Trail of (v -> t -> m -> v)
and  m = MNil | MCons of (c * s * t) * m
```

Listing 5 Type definition for delinearized interpreter.

```
type v = VFun of (v -> c -> s -> t -> m -> v)
       | VContS of c * s * t | VContC of c * s * t
       | VEnv of v list
and  c = C0 | CApp0 of e * string list * c | CApp1 of c
and  s = v list
and  t = TNil | Trail of (v -> t -> m -> v)
and  m = MNil | MCons of (c * s * t) * m
```

is given. Dynamic data include `v list` in `CApp0` and `v` in `CApp1`, which are available only at run-time. Since our goal is to transform the interpreter into a compiler and a virtual machine, we separate these two types of data by introducing a stack.

Listing 4 shows the resulting data definition. The previous continuation `c` is split into a pair of a continuation `c` and a stack `s`. The former is a list of frames, where the frame `f` now keeps only the static data. The runtime data are kept in the stack, which is a list of values. Since the previous `CApp0` included `v list`, the value `v` is extended with `VEnv` to store the `v list` as a single value.[4] Since the new `c` (a list of frames) and `s` (a list of values) are obtained by splitting a single list (a list of `f` in Listing 3), they always have the same length. In the subsequent derivations, we keep this invariant throughout.

Because we only changed the representation of `c` locally, we immediately see that the new interpreter behaves the same as the previous one.

4.4 Delinearizing Continuations

The purpose of defunctionalization (Section 4.1) and linearization of continuations (Section 4.2) was to introduce a data stack. Now that we have introduced a data stack, we transform continuations back to the higher-order form via delinearization. In this section, we convert lists into constructors.

Listing 5 shows the resulting data definition. Here, only the static `f` is incorporated into `c`. The stack `s` remains as a list of values. Note that `c` contains only static data (in contrast to `c` in Listing 2 that contains both static and dynamic data). All the dynamic data are still carried around in `s`. As in Section 4.2, the old and new representations of `c` are isomorphic, and thus the new interpreter behaves the same as the previous one.

[4] The introduction of `VEnv` into `v` is arbitrary. Although we introduced it to emulate caller-save registers often found in the compiled code, a user cannot write a program that evaluates to `VEnv`. Instead, we could introduce a new type for stack items that consists of either a value or a list of values (`VEnv`). In the current paper, we followed our previous work [3] and included `VEnv` directly to `v`.

Figure 1 Abstract machine.

e	$\Rightarrow$	$\langle e,\ [],\ [],\ C_0,\ [],\ \mathit{TNil},\ []\rangle$
$\langle x,\ xs,\ vs,\ c,\ s,\ t,\ m\rangle$	$\Rightarrow$	$\langle c,\ \mathit{List.nth}\ vs\ (\mathit{offset}\ x\ xs),\ s,\ t,\ m\rangle$
$\langle \lambda x.e,\ xs,\ vs,\ c,\ s,\ t,\ m\rangle$	$\Rightarrow$	$\langle c,\ \mathit{VFun}(e,\ x,\ xs,\ vs),\ s,\ t,\ m\rangle$
$\langle e_0\ e_1,\ xs,\ vs,\ c,\ s,\ t,\ m\rangle$	$\Rightarrow$	$\langle e_0,\ xs,\ vs,\ \mathit{CApp}_0(e_1,\ xs,\ c),\ \mathit{VEnv}(vs) :: s,\ t,\ m\rangle$
$\langle \mathit{Shift}(x,\ e),\ xs,\ vs,\ c,\ s,\ t,\ m\rangle$	$\Rightarrow$	$\langle e,\ x :: xs,\ \mathit{VContS}(c,\ s,\ t) :: vs,\ C_0,\ [],\ \mathit{TNil},\ m\rangle$
$\langle \mathit{Control}(x,\ e),\ xs,\ vs,\ c,\ s,\ t,\ m\rangle$	$\Rightarrow$	$\langle e,\ x :: xs,\ \mathit{VContC}(c,\ s,\ t) :: vs,\ C_0,\ [],\ \mathit{TNil},\ m\rangle$
$\langle \mathit{Shift}_0(x,\ e),\ xs,\ vs,\ c,\ s,\ t,\ (c_0,\ s_0,\ t_0) :: m_0\rangle$	$\Rightarrow$	$\langle e,\ x :: xs,\ \mathit{VContS}(c,\ s,\ t) :: vs,\ c_0,\ s_0,\ t_0,\ m_0\rangle$
$\langle \mathit{Control}_0(x,\ e),\ xs,\ vs,\ c,\ s,\ t,\ (c_0,\ s_0,\ t_0) :: m_0\rangle$	$\Rightarrow$	$\langle e,\ x :: xs,\ \mathit{VContC}(c,\ s,\ t) :: vs,\ c_0,\ s_0,\ t_0,\ m_0\rangle$
$\langle \mathit{Reset}(e),\ xs,\ vs,\ c,\ s,\ t,\ m\rangle$	$\Rightarrow$	$\langle e,\ xs,\ vs,\ C_0,\ [],\ \mathit{TNil},\ (c,\ s,\ t) :: m\rangle$
$\langle C_0,\ v,\ [],\ \mathit{TNil},\ []\rangle$	$\Rightarrow$	v
$\langle C_0,\ v,\ [],\ \mathit{TNil},\ (c,\ s,\ t) :: m\rangle$	$\Rightarrow$	$\langle c,\ v,\ s,\ t,\ m\rangle$
$\langle C_0,\ v,\ [],\ \mathit{Trail}(h),\ m\rangle$	$\Rightarrow$	$\langle h,\ v,\ \mathit{TNil},\ m\rangle$
$\langle \mathit{CApp}_0(e,\ xs,\ c),\ v,\ \mathit{VEnv}(vs) :: s,\ t,\ m\rangle$	$\Rightarrow$	$\langle e,\ xs,\ vs,\ \mathit{CApp}_1(c),\ v :: s,\ t,\ m\rangle$
$\langle \mathit{CApp1}(c),\ v,\ \mathit{VFun}(e,\ x,\ xs,\ vs) :: s,\ t,\ m\rangle$	$\Rightarrow$	$\langle e,\ x :: xs,\ v :: vs,\ c,\ s,\ t,\ m\rangle$
$\langle \mathit{CApp}_1(c),\ v,\ \mathit{VContS}(c',\ s',\ t') :: s,\ t,\ m\rangle$	$\Rightarrow$	$\langle c',\ v,\ s',\ t',\ (c,\ s,\ t) :: m\rangle$
$\langle \mathit{CApp}_1(c),\ v,\ \mathit{VContC}(c',\ s',\ t') :: s,\ t,\ m\rangle$	$\Rightarrow$	$\langle c',\ v,\ s',\ \mathit{apnd}\ t'\ (\mathit{cons}\ (\mathit{Hold}\ (c,\ s))\ t),\ m\rangle$
$\langle \mathit{Hold}\ (c,\ s),\ v,\ t,\ m\rangle$	$\Rightarrow$	$\langle c,\ v,\ s,\ t,\ m\rangle$
$\langle \mathit{Append}\ (h,\ h'),\ v,\ t,\ m\rangle$	$\Rightarrow$	$\langle h,\ v,\ \mathit{cons}\ h'\ t,\ m\rangle$

4.5 Abstract Machine

In this section, we briefly describe the abstract machine that can be derived from the interpreter in Section 4.4. Since all the interpreters in this paper receive a continuation and a metacontinuation, all the calls to interpreter functions (such as `f1` and the dispatch function for continuations) are tail calls. As such, we can easily derive an abstract machine by simply regarding the arguments to interpreter functions as a state of the abstract machine. The derived abstract machine is shown in Figure 1. Although we omit the code for the interpreter, one can imagine how it looks like from the abstract machine. To extract the abstract machine, we further performed the following transformations:

- We defunctionalized the argument of `VFun`. A function is now represented as a closure. We will perform the same transformation later; see Section 5.3.
- We defunctionalized the argument of `Trail`. The `Trail` data are constructed in the two branches of `cons` (see Listing 1). The first one is represented as `Hold` that holds an invocation context; the second one as `Append` that appends two trails. We will perform the same transformation later; see Section 5.3 for details.
- Instead of `MNil` and `MCons`, we use standard lists for metacontinuations.

Because we have introduced a stack into the interpreter, we obtain a stack-based abstract machine. This is in contrast to the previous abstract machines [5, 11, 28] which do not carry a stack explicitly. The obtained abstract machine clearly describes the behavior of control operators. When one of the control operators is used, the current continuation c, stack s, and trail t are captured, put into a stack, and bound to x. Then, the body of the control operator is executed. For *Shift* and *Control*, the current continuation and trail are cleared, whereas for Shift_0 and $\mathit{Control}_0$, the ones in the metacontinuation are used. The `reset` operator pushes the current c, s, and t on the metacontinuation m, and initializes them.

When a continuation captured by *Shift* or Shift_0 is invoked, the current c, s, and t are pushed onto m and the captured state is reinstated. When a continuation captured by *Control* or $\mathit{Control}_0$ is invoked, on the other hand, t is extended by c and s (via *cons*), and the result is in turn extended by t' (via *apnd*).

4.6 Refunctionalizing Continuations

Finally, Listing 6 shows the refunctionalized interpreter where defunctionalized continuations are transformed back to higher-order functions. It is similar to the definitional interpreter in Listing 1, but passes around a stack. Typewise, all the occurrences of a continuation `c` in Listing 1 are replaced by pairs `c * s` of a continuation and a stack. Furthermore, the type `c` and the type of the argument of `VFun` are modified to receive a stack.

Compared to the definitional interpreter `f1` in Listing 1, the refunctionalized interpreter `f6` receives an additional stack argument `s`, and whenever it returns a value, a continuation `c` is applied to the value together with a stack `s`. We can also observe that the references to free variables in the definitional interpreter (`vs` and `v0` in the `App` case) are now realized by passing those values via the stack. We push those values at the recursive calls and pop them when the corresponding continuations are called. Since stacks are extracted from continuations and stacks have the same structure as (now refunctionalized) continuations, popping a value would never fail: popped values correspond to the dynamic arguments of `CApp0` and `CApp1`. This is the consequence of the invariant we keep between continuations and stacks. Similarly, `idc` corresponds to `C0`, which has no dynamic counterpart. Thus, the stack argument of `idc` (the second argument of `idc` in Listing 6) must be an empty stack.

The argument of `Trail` needs explanation. Since we have not defunctionalized the argument of `Trail` yet, we need type conversion to store a continuation `c` in a trail. See the first argument to `cons` in the `App` case. The continuation `c` is turned into `fun v t m -> c v s1 t m` with `s1` being a free variable. Later when we defunctionalize it, the stack `s1` will be extracted; see Section 5.3.

It is not straightforward to obtain the refunctionalized interpreter from the previous one. One has to verify that the previous interpreter is in defunctionalized form [10]. However, once it is obtained, it is simple to verify its correctness: by defunctionalizing the refunctionalized interpreter, we can obtain the previous one.

5 Deriving a Virtual Machine

In this section, we derive a virtual machine from the refunctionalized interpreter obtained in Section 4.6. We first combine arguments so that values are passed via a stack (Section 5.1). We then stage the interpreter into a compiler that operates on instructions represented as functions (Section 5.2). By defunctionalizing the instructions (Section 5.3) and linearizing instructions (Section 5.4) and stacks (Section 5.5), we obtain the virtual machine (Section 6).

5.1 Combining Arguments

In Listing 6, functions in `VFun` as well as continuations `c` receive both a value `v` and a stack `s`. In a low-level implementation, such as a virtual machine, we want to pass all the values via a stack rather than passing a value and a stack separately. Listing 7 shows the type definition of the result of such a transformation.

The argument `v` is removed from the argument of `VFun` and `c`. When we call such a function, we push `v` to the stack before the call. When the function is called, we pop `v` from the stack before the function body is executed. We do the same for the interpreter function: we remove the `vs` argument and push it on the stack. As a result, the type of the interpreter function `f7` after the transformation becomes as follows:

```
(* f7 : e -> string list -> c -> s -> t -> m -> v *)
```

Listing 6 Refunctionalized interpreter (`cons` and `apnd` are the same as in Listing 1).

```
type v = VFun of (v -> c -> s -> t -> m -> v)
       | VContS of c * s * t | VContC of c * s * t
       | VEnv of v list
and  c = v -> s -> t -> m -> v
and  s = v list
and  t = TNil | Trail of (v -> t -> m -> v)
and  m = MNil | MCons of (c * s * t) * m

(* initial continuation : v -> s -> t -> m -> v *)
let idc v [] t m = match t with
    TNil -> (match m with
        MNil -> v
      | MCons((c,s,t),m) -> c v s t m)
  | Trail(h) -> h v TNil m

(* f6 : e -> string list -> v list -> c -> s -> t -> m -> v *)
let rec f6 e xs vs c s t m = match e with
  | Var(x) -> c (List.nth vs (Env.offset x xs)) s t m
  | Fun(x,e) ->
      c (VFun(fun v c' s' t' m' -> f6 e (x::xs) (v::vs) c' s' t' m'))
        s t m
  | App(e0,e1) ->
      f6 e0 xs vs (fun v0 (VEnv(vs)::s0) t0 m0 ->
      f6 e1 xs vs (fun v1 (v0::s1) t1 m1 ->
      (match v0 with
         VFun(f) -> f v1 c s1 t1 m1
       | VContS(c',s',t') -> c' v1 s' t' (MCons((c,s1,t1),m1))
       | VContC(c',s',t') ->
         c' v1 s' (apnd t' (cons (fun v t m -> c v s1 t m) t1)) m1))
      (v0::s0) t0 m0) (VEnv(vs)::s) t m
  | Shift(x,e) -> f6 e (x::xs) (VContS(c,s,t)::vs) idc [] TNil m
  | Control(x,e) -> f6 e (x::xs) (VContC(c,s,t)::vs) idc [] TNil m
  | Shift0(x,e) -> (match m with
      MCons((c0,s0,t0),m0) ->
        f6 e (x::xs) (VContS(c,s,t)::vs) c0 s0 t0 m0)
  | Control0(x,e) -> (match m with
      MCons((c0,s0,t0),m0) ->
        f6 e (x::xs) (VContC(c,s,t)::vs) c0 s0 t0 m0)
  | Reset(e) -> f6 e xs vs idc [] TNil (MCons((c,s,t),m))

(* f : e -> v *)
let f expr = f6 expr [] [] idc [] TNil MNil
```

Listing 7 Type definition for interpreter with combined arguments.

```
type v = VFun of (c -> s -> t -> m -> v)
       | VContS of c * s * t | VContC of c * s * t
       | VEnv of v list
and  c = s -> t -> m -> v
and  s = v list
and  t = TNil | Trail of (v -> t -> m -> v)
and  m = MNil | MCons of (c * s * t) * m
```

Since we simply changed the way two arguments are passed locally, we immediately see that the new interpreter behaves the same as the previous one.

5.2 Introducing Combinators as Instructions

In this section, we extract a compiler from the interpreter. Looking at the type of `f7` in the previous section, we notice that the first two arguments are static and the rest of the arguments are dynamic. We first define the type `i` of instructions (in Listing 8) as the dynamic part of the interpreter, which represents the work to be done when dynamic data are received. We then regard the interpreter as a compiler that accepts two static data and returns an instruction. Listing 8 shows the result.

The interpreter function `f8`, or a compiler, processes only the static data: the input term `e` and a list of variable names `xs`. It then produces an instruction of type `i`, which performs the rest of the work when dynamic data are given.

For example, in the case of `Var (x)`, the compiler emits an instruction `access`, which, given dynamic data, returns the corresponding value in the environment. In the case of `App (e0, e1)`, we define `push_env`, `pop_env`, and `call`, and concatenate these instructions using `(>>)`. We employ the same technique as the previous work [3]: we store the return address `VK` (added to the definition of `v`) to the stack in `return` and retrieve it in `call`.

This interpreter behaves the same as the previous one, because if we inline all the instructions, we obtain the interpreter in the previous section.

5.3 Defunctionalizing Instructions

In this section, we defunctionalize the functional instructions introduced in the previous section into the ones that are closer to machine instructions. Specifically, we defunctionalize the argument of `VFun`, `i`, `c`, and the argument of `Trail`, separately, and change the representation of `m`. See Listing 9.

First, the argument of `VFun` (see `push_closure` and `call` in Listing 8) is defunctionalized to a closure. Second, the instruction `i` is defunctionalized. All the functional instructions are turned into constructors as shown in `i` in Listing 9. The corresponding dispatch function (omitted) is a virtual machine: given an instruction and the current dynamic state, it performs necessary operations. Observe how a virtual machine is naturally derived by defunctionalizing functional instructions. Note also that the instruction is *not* linear: it includes `ISeq` corresponding to `(>>)` and thus has a tree structure.

Third, `c` is defunctionalized. There are two cases that constitute the value of `c` in Listing 8: the identity continuation `idc`, which is closed, and the second argument to `i0` in `(>>)`, `fun s' t' m' -> i1 c s' t' m'`. Since the free variables of the latter are `i1` and `c`, we can represent `c` as a list of `i`, regarding the former as an empty list and the latter as cons list.

Fourth, the argument of `Trail` is defunctionalized and given a new type `h`. The `Trail` data are constructed in the two branches of `cons` (see Listing 1): its argument is either a continuation `h` or `fun v t' m -> h v (cons h' t') m` which has `h` and `h'` as free variables. They are represented as `Hold` and `Append` in Listing 9, respectively. Note that `h` has a tree structure. Finally, the metacontinuation `m` is turned into an OCaml list, as no circular dependency arises any more.[5]

Since all these changes are instances of defunctionalization and a simple local change of data representation, the behavior of the new interpreter is the same as the previous one.

[5] Unlike the definitional interpreter. See footnote 2.

Listing 8 Interpreter using combinators factored as instructions.

```
type v = VFun of (c -> s -> t -> m -> v)
       | VContS of c * s * t | VContC of c * s * t
       | VEnv of v list | VK of c
and  c = s -> t -> m -> v
and  s = v list
and  t = TNil | Trail of (v -> t -> m -> v)
and  m = MNil | MCons of (c * s * t) * m
type i = c -> s -> t -> m -> v

(* (>>) : i -> i -> i *)
let (>>) i0 i1 =
  fun c s t m -> i0 (fun s' t' m' -> i1 c s' t' m') s t m

(* instructions *)
let access n = fun c (VEnv(vs)::s) t m -> c ((List.nth vs n)::s) t m
let push_closure i = fun c (VEnv(vs)::s) t m ->
  c (VFun(fun c' (v::s') t' m' -> i c' (VEnv(v::vs)::s') t' m')::s)
    t m
let return = fun _ (v::VK(c)::s) t m -> c (v::s) t m
let push_env = fun c (VEnv(vs)::s) t m ->
  c (VEnv(vs)::VEnv(vs)::s) t m
let pop_env = fun c (v::VEnv(vs)::s) t m -> c (VEnv(vs)::v::s) t m
let call = fun c (v1::v0::s) t m -> match v0 with
    VFun(f) -> f idc (v1::VK(c)::s) t m
  | VContS(c',s',t') -> c' (v1::s') t' (MCons((c,s,t),m))
  | VContC(c',s',t') ->
    c' (v1::s') (apnd t' (cons (fun v t m -> c (v::s) t m) t)) m
let shift i = fun c (VEnv(vs)::s) t m ->
  i idc (VEnv(VContS(c,s,t)::vs)::[]) TNil m
let control i = fun c (VEnv(vs)::s) t m ->
  i idc (VEnv(VContC(c,s,t)::vs)::[]) TNil m
let shift0 i = fun c (VEnv(vs)::s) t (MCons((c0,s0,t0),m0)) ->
  i c0 (VEnv(VContS(c,s,t)::vs)::s0) t0 m0
let control0 i = fun c (VEnv(vs)::s) t (MCons((c0,s0,t0),m0)) ->
  i c0 (VEnv(VContC(c,s,t)::vs)::s0) t0 m0
let reset i = fun c (VEnv(vs)::s) t m ->
  i idc (VEnv(vs)::[]) TNil (MCons((c,s,t),m))

(* f8 : e -> string list -> i *)
let rec f8 e xs = match e with
    Var(x) -> access (Env.offset x xs)
  | Fun(x,e) -> push_closure ((f8 e (x::xs)) >> return)
  | App(e0,e1) ->
      push_env >> (f8 e0 xs) >> pop_env >> (f8 e1 xs) >> call
  | Shift(x,e) -> shift (f8 e (x::xs))
  | Control(x,e) -> control (f8 e (x::xs))
  | Shift0(x,e) -> shift0 (f8 e (x::xs))
  | Control0(x,e) -> control0 (f8 e (x::xs))
  | Reset(e) -> reset (f8 e xs)

(* f : e -> v *)
let f expr = f8 expr [] idc (VEnv([])::[]) TNil MNil
```

Listing 9 Type definition for interpreter with defunctionalized instructions and continuations.

```
type v = VFun of i * v list
       | VContS of c * s * t | VContC of c * s * t
       | VEnv of v list | VK of c
and  i = IAccess of int | IPush_closure of i | IReturn
       | IPush_env | IPop_env | ICall
       | IShift of i | IControl of i | IShift0 of i | IControl0 of i
       | IReset of i | ISeq of i * i
and  c = i list
and  s = v list
and  h = Hold of c * s | Append of h * h
and  t = TNil | Trail of h
type m = (c * s * t) list
```

Listing 10 The function `flat` to remove `ISeq`.

```
(* flat: i -> i list *)
let rec flat i = match i with
    IAccess (n) -> [IAccess (n)]
  | ...
  | ISeq (i0, i1) -> flat i0 @ flat i1
```

5.4 Linearizing Instructions

In the previous section, we used `ISeq` to combine two instructions. As such, an instruction had a tree structure. We can turn it into a linear list by flattening the tree into an OCaml list. With this transformation, `i` in `VFun` becomes `i list` (or equivalently, `c`) and `ISeq` is removed from `i`.

Although the transformation is intuitively clear, to show its correctness, we need to prove that the instructions form a monoid. Namely, the grouping of instructions does not matter as long as the order of instructions is preserved. We briefly sketch the proof. We first define a flattening function (Listing 10) that turns `i` into a list of `i`'s without `ISeq`. We can define similar functions (`flatV`, `flatC`, etc.) that flatten all the instructions appearing in given data (a value, a continuation, etc., respectively). We then prove the following equivalences:

- `flat (f9 e xs)` = `f10 e xs`, stating that the list of instructions generated by the new compiler is the same as flattening the instruction generated by the old compiler, and
- `flatV (run_i9 i c s t m)` = `run_c10 (flat i @ flatC c) (flatS s) (flatT t) (flatM m)`, stating that running `i` under `c` in the old virtual machine yields the same result as running the flattened instructions of `i` and `c` in the new virtual machine (or both do not terminate).

The former is proved by induction on the structure of `e` and the latter on the number of steps the old virtual machine takes. One has to be careful in the case when `i` is `ISeq`. Although the old virtual machine takes a step to execute it, there is no corresponding execution step in the new virtual machine, since `ISeq` is already flattened. Therefore, the termination behavior of the two virtual machines is different when the instruction list contains infinitely many `ISeq`'s: the former continues indefinitely executing `ISeq`'s while the latter terminates since all the `ISeq`'s are already flattened and removed. This does not happen, since all the instructions are finite.

Listing 11 Interpreter with linearized trails.

```
type v = VFun of c * v list | VContS of t | VContC of t
       | VEnv of v list | VK of c
and  i = IAccess of int | IPush_closure of c | IReturn
       | IPush_env | IPop_env | ICall
       | IShift of c | IControl of c | IShift0 of c | IControl0 of c
       | IReset of c
and  c = i list
and  s = v list
and  t = (c * s) list
type m = t list
```

Figure 2 Virtual machine.

$$
\begin{array}{rcl}
c & \Rightarrow & \langle c,\ [\mathit{VEnv}([])],\ [],\ []\rangle \\
\hline
\langle [],\ v :: [],\ [],\ []\rangle & \Rightarrow & v \\
\langle [],\ v :: [],\ [],\ ((c,\ s) :: t) :: m\rangle & \Rightarrow & \langle c,\ v :: s,\ t,\ m\rangle \\
\langle [],\ v :: [],\ (c,\ s) :: t,\ m\rangle & \Rightarrow & \langle c,\ v :: s,\ t,\ m\rangle \\
\langle \mathit{IAccess}(n) :: c,\ \mathit{VEnv}(vs) :: s,\ t,\ m\rangle & \Rightarrow & \langle c,\ (\mathit{List.nth}\ vs\ n) :: s,\ t,\ m\rangle \\
\langle \mathit{IPushClosure}(c') :: c,\ \mathit{VEnv}(vs) :: s,\ t,\ m\rangle & \Rightarrow & \langle c,\ \mathit{VFun}(c',\ vs) :: s,\ t,\ m\rangle \\
\langle \mathit{IReturn} :: \_,\ v :: \mathit{VK}(c) :: s,\ t,\ m\rangle & \Rightarrow & \langle c,\ v :: s,\ t,\ m\rangle \\
\langle \mathit{IPushEnv} :: c,\ \mathit{VEnv}(vs) :: s,\ t,\ m\rangle & \Rightarrow & \langle c,\ \mathit{VEnv}(vs) :: \mathit{VEnv}(vs) :: s,\ t,\ m\rangle \\
\langle \mathit{IPopEnv} :: c,\ v :: \mathit{VEnv}(vs) :: s,\ t,\ m\rangle & \Rightarrow & \langle c,\ \mathit{VEnv}(vs) :: v :: s,\ t,\ m\rangle \\
\langle \mathit{ICall} :: c,\ v :: \mathit{VFun}(c',\ vs) :: s,\ t,\ m\rangle & \Rightarrow & \langle c',\ \mathit{VEnv}(v :: vs) :: \mathit{VK}(c) :: s,\ t,\ m\rangle \\
\langle \mathit{ICall} :: c,\ v :: \mathit{VContS}((c',\ s') :: t') :: s,\ t,\ m\rangle & \Rightarrow & \langle c',\ v :: s',\ t',\ ((c,\ s) :: t) :: m\rangle \\
\langle \mathit{ICall} :: c,\ v :: \mathit{VContC}((c',\ s') :: t') :: s,\ t,\ m\rangle & \Rightarrow & \langle c',\ v :: s',\ t'\ @\ (c,\ s) :: t,\ m\rangle \\
\langle \mathit{IShift}(c') :: c,\ \mathit{VEnv}(vs) :: s,\ t,\ m\rangle & \Rightarrow & \langle c',\ \mathit{VEnv}(\mathit{VContS}((c,\ s) :: t) :: vs) :: [],\ [],\ m\rangle \\
\langle \mathit{IControl}(c') :: c,\ \mathit{VEnv}(vs) :: s,\ t,\ m\rangle & \Rightarrow & \langle c',\ \mathit{VEnv}(\mathit{VContC}((c,\ s) :: t) :: vs) :: [],\ [],\ m\rangle \\
\langle \mathit{IShift}_0(c') :: c,\ \mathit{VEnv}(vs) :: s,\ t,\ ((c_0,\ s_0) :: t_0) :: m_0\rangle & \Rightarrow & \langle c'\ @\ c_0,\ \mathit{VEnv}(\mathit{VContS}((c,\ s) :: t) :: vs) :: s_0,\ t_0,\ m_0\rangle \\
\langle \mathit{IControl}_0(c') :: c,\ \mathit{VEnv}(vs) :: s,\ t,\ ((c_0,\ s_0) :: t_0) :: m_0\rangle & \Rightarrow & \langle c'\ @\ c_0,\ \mathit{VEnv}(\mathit{VContC}((c,\ s) :: t) :: vs) :: s_0,\ t_0,\ m_0\rangle \\
\langle \mathit{IReset}(c') :: c,\ \mathit{VEnv}(vs) :: s,\ t,\ m\rangle & \Rightarrow & \langle c',\ \mathit{VEnv}(vs) :: [],\ [],\ ((c,\ s) :: t) :: m\rangle
\end{array}
$$

5.5 Linearizing Trails

Finally, we transform the type `t` of trails, which had a tree structure (Listing 9), into a linear list. By regarding `TNil` as an empty list, `Hold` as a singleton list consisting of `c * s`, and `Append` as a list append, we can represent `t` as a list of `c * s`. The resulting type definitions are shown in Listing 11. Now that `t` becomes `(c * s) list`, we change the type of `VContS` and `VContC` from `c * s * t` to `t` by piling up the `c * s` pair onto `t`. Similarly, `m` can be represented as `t list`.

To establish the correctness of this transformation, we need to show that the new virtual machine behaves the same as before:

```
flatV (run_c10 c s t m) = run_c11 c (flatS s) (flatT t) (flatM m)
```

where `flat` functions are defined similarly to the ones in the previous section to flatten the type of trails. The above equivalence is shown by induction on the number of execution steps the old virtual machine takes.

6 Virtual Machine

Figure 2 shows the state transition rules for the virtual machine obtained from the interpreter in the previous section. The main state consists of a tuple (c, s, t, m) of four elements: a continuation, a stack, a trail, and a metacontinuation. We show an example how a program is compiled to a list of instructions and executed on the virtual machine in the appendix.

The virtual machine succinctly models the low-level behavior of control operators. Just as in the abstract machine, when one of the control operators is used, the current continuation (or a pointer to an instruction) c, stack s, and trail t are captured and put into a stack. Then, the body of the control operator is executed. For *IShift* and *IControl*, the current continuation and trail are cleared, whereas for IShift_0 and $\mathit{IControl}_0$, the ones in the metacontinuation are used. The `reset` operator pushes the current c, s, and t on the metacontinuation m, and initializes them.

When a continuation captured by *IShift* or IShift_0 is invoked, the current c, s, and t are pushed onto m and the captured state is reinstated. When a continuation captured by *IControl* or $\mathit{IControl}_0$ is invoked, on the other hand, the current c and s are added to t to which the captured trail t' is appended.

Although we maintain s, t, and m separately in the virtual machine, we can represent them as a single stack. Remember that s is a list of values. Since t is a list of pairs of c and s, it has the form:

$$[(c, [v; \ldots; v]); \ldots; (c, [v; \ldots; v])]$$

Thus, if we represent c as a single value (e.g., using `VK`) pointing to the first instruction designated by c, and if we maintain the positions of c in t using pointers, we can represent t as a list of values. Furthermore, since m is a list of trails (a list of lists of pairs of c and s), it can be represented as a list of values, too, if we maintain pointers to each element of m.

If we represent s, t, and m as a single stack, we notice that we can sometimes avoid copying s and t. When c, s, and t are pushed to m in the rules for *IReset* and the *VContS* and *VContC* cases of *ICall*, the ordering of s, t, and m does not change. Thus, we can simply rearrange the pointers to the head of a stack, trail, and metacontinuation appropriately, without copying s and t. Similarly for s_0, t_0, and m_0 in the rules for IShift_0 and $\mathit{IControl}_0$. When do we have to copy s and t? It is when we use control operators or apply captured continuations. The s and t must be copied, in the former case to be stored in *VContS* or *VContC*, and in the latter case to use what was stored.

Finally, in the rules for IShift_0 and $\mathit{IControl}_0$, the body instructions c' of the control operators and the instructions c_0 in the metacontinuation are concatenated. This concatenation reflects the fact that the body of IShift_0 and $\mathit{IControl}_0$ has access to the context outside the current enclosing `reset`. (In the abstract machine, the concatenation was realized by executing the body under the continuation stored in the metacontinuation.) Implementation-wise, this suggests that we need to keep track of a list of pointers to these continuations, which is an interesting observation that has not been observed before.

7 Related Work

We are not aware of any work that derives a virtual machine for the four delimited-control operators other than `shift/reset`. Deriving a virtual machine for other language constructs includes Ager, Biernacki, Danvy, and Midtgaard's work [1] for λ-calculus (of various flavors) and Igarashi and Iwaki's work [18] for a staged language.

As for an abstract machine, Biernacki, Danvy, and Millikin [5] present abstract machines for the four delimited-control operators as definitional and derive a CPS interpreter for `control/prompt`. Shan [28] derives an abstract machine for `control/prompt` from the CPS interpreter for `control/prompt`. In both work, the derivation is done for `control/prompt` only. Their abstract machines are similar to ours but do not maintain a stack explicitly.

Dyvbig, Peyton Jones, and Sabry [11] show an abstract machine for primitive control operators that can implement four delimited-control operators with named prompts. Since they use their own primitive control operators, their CPS interpreter is quite different from ours. They do not use trails and represent concatenation of contexts using a metacontinuation, which is a list of continuations. Based on this abstract machine, Kiselyov [21] implements the control operators in OCaml by emulating the behavior of the abstract machine using OCaml's exception handling mechanism.

Hillerström, Lindley, and Atkey [17] show CPS translations and abstract machine semantics for algebraic effects and handlers. It would be interesting to see if the program transformation approach can be used in this setting, too.

8 Conclusion

In this paper, we have derived a compiler and a virtual machine for the four delimited-control operators from the definitional interpreter. The resulting virtual machine suggests a low-level implementation method for delimited continuations.

Although we focused on the behavior of the delimited-control operators, we also want to consider their type systems. We are currently trying to build a type system for the four delimited-control operators (the one for `control/prompt` is in [6]). Once we obtain a type system, we plan to implement the four delimited-control operators in OchaCaml [22] based on the virtual machine developed in this paper. That would form a solid foundation on which a different implementation of algebraic effects and handlers can be considered.

References

1. Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. *BRICS Report Series*, 03(14), 2003.
2. Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, pages 8–19. ACM, 2003. `doi:10.1145/888251.888254`.
3. Kenichi Asai and Arisa Kitani. Functional derivation of a virtual machine for delimited continuations. In Temur Kutsia, Wolfgang Schreiner, and Maribel Fernández, editors, *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 26-28, 2010, Hagenberg, Austria*, pages 87–98. ACM, 2010. `doi:10.1145/1836089.1836101`.
4. Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.*, 84(1):108–123, 2015. `doi:10.1016/j.jlamp.2014.02.001`.
5. Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. A dynamic continuation-passing style for dynamic delimited continuations. *ACM Trans. Program. Lang. Syst.*, 38(1):2:1–2:25, 2015. `doi:10.1145/2794078`.
6. Youyou Cong, Chiaki Ishio, Kaho Honda, and Kenichi Asai. A functional abstraction of typed invocation contexts. In Naoki Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*, volume 195 of *LIPIcs*, pages 12:1–12:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.FSCD.2021.12`.
7. Olivier Danvy. Defunctionalized interpreters for programming languages. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 131–142. ACM, 2008. `doi:10.1145/1411204.1411206`.
8. 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`.

9 Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Math. Struct. Comput. Sci.*, 2(4):361–391, 1992. doi:10.1017/S0960129500001535.

10 Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Sci. Comput. Program.*, 74(8):534–549, 2009. doi:10.1016/j.scico.2007.10.007.

11 R. Kent Dybvig, Simon L. Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *J. Funct. Program.*, 17(6):687–730, 2007. doi:10.1017/S0956796807006259.

12 Kavon Farvardin and John H. Reppy. From folklore to fact: comparing implementations of stacks and continuations. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 75–90. ACM, 2020. doi:10.1145/3385412.3385994.

13 Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and P. Mager, editors, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 180–190. ACM Press, 1988. doi:10.1145/73560.73576.

14 Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control. *Proc. ACM Program. Lang.*, 1(ICFP):13:1–13:29, 2017. doi:10.1145/3110257.

15 Martin Gasbichler and Michael Sperber. Final shift for call/cc: direct implementation of shift and reset. In Mitchell Wand and Simon L. Peyton Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, pages 271–282. ACM, 2002. doi:10.1145/581478.581504.

16 Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ml-like languages. In John Williams, editor, *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, pages 12–23. ACM, 1995. doi:10.1145/224164.224173.

17 Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect handlers via generalised continuations. *J. Funct. Program.*, 30:e5, 2020. doi:10.1017/S0956796820000040.

18 Atsushi Igarashi and Masashi Iwaki. Deriving compilers and virtual machines for a multi-level language. In Zhong Shao, editor, *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings*, volume 4807 of *Lecture Notes in Computer Science*, pages 206–221. Springer, 2007. doi:10.1007/978-3-540-76637-7_14.

19 Yukiyoshi Kameyama and Takuo Yonezawa. Typed dynamic control operators for delimited continuations. In Jacques Garrigue and Manuel V. Hermenegildo, editors, *Functional and Logic Programming, 9th International Symposium, FLOPS 2008, Ise, Japan, April 14-16, 2008. Proceedings*, volume 4989 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2008. doi:10.1007/978-3-540-78969-7_18.

20 Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA – September 25–27, 2013*, pages 145–158. ACM, 2013. doi:10.1145/2500365.2500590.

21 Oleg Kiselyov. Delimited control in ocaml, abstractly and concretely. *Theor. Comput. Sci.*, 435:56–76, 2012. doi:10.1016/j.tcs.2012.02.025.

22 Moe Masuko and Kenichi Asai. Caml light+ shift/reset= caml shift. *Theory and Practice of Delimited Continuations (TPDC 2011)*, pages 33–46, 2011.

23 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.

24 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, June 24-30, 2019, Dortmund, Germany*, volume 131 of *LIPIcs*, pages 30:1–30:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.FSCD.2019.30`.

25 Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009. `doi:10.1007/978-3-642-00590-9_7`.

26 John C. Reynolds. Definitional interpreters for higher-order programming languages. In John J. Donovan and Rosemary Shields, editors, *Proceedings of the ACM annual conference, ACM 1972, 1972, Volume 2*, pages 717–740. ACM, 1972. `doi:10.1145/800194.805852`.

27 John C. Reynolds. Definitional interpreters for higher-order programming languages. *High. Order Symb. Comput.*, 11(4):363–397, 1998. `doi:10.1023/A:1010027404223`.

28 Chung-chieh Shan. A static simulation of dynamic delimited control. *High. Order Symb. Comput.*, 20(4):371–401, 2007. `doi:10.1007/s10990-007-9010-4`.

A Example Execution

In this appendix, we show an example how the compiler and the virtual machine work. We use the control term in Section 2: $1 + \langle(\mathcal{F}c.\, 2 \times c\, 3) + \mathcal{F}c'.\, 4\rangle$. It is straightforward to support numbers and binary operators; see the supplementary material. State transition rules for the new instructions are summarized in Figure 3.

The list of instructions output by the compiler is:

$$[\mathit{IPushEnv}_1;\ \mathit{INum}(1);\ \mathit{IPopEnv}_1;\ \mathit{IReset}(c_1);\ \mathit{IOp}_1(+)]$$

where

$$\begin{array}{rcl} c_1 & = & [\mathit{IPushEnv}_2;\ \mathit{IControl}_1(c_2);\ \mathit{IPopEnv}_2;\ \mathit{IControl}_2(c_3);\ \mathit{IOp}_2(+)] \\ c_2 & = & [\mathit{IPushEnv}_3;\ \mathit{INum}(2);\ \mathit{IPopEnv}_3;\ \mathit{IPushEnv}_4;\ \mathit{IAccess}(0);\ \mathit{IPopEnv}_4; \\ & & \ \mathit{INum}(3);\ \mathit{ICall};\ \mathit{IOp}_3(*)] \\ c_3 & = & [\mathit{INum}(4)] \end{array}$$

We use subscripts to disambiguate instructions that appear more than once.

The list of instruction is executed as in Figure 4. We can observe that the trails $3 + []$ (i.e., $([\mathit{IOp}_2(+)], [\mathit{VNum}(3)])$) and $2 \times []$ (i.e., $([\mathit{IOp}_3(*)], [\mathit{VNum}(2)])$) are concatenated at the second invocation of *IControl* and are captured in vc_2.

$$\begin{array}{rcl} \langle \mathit{INum}(n) :: c,\ \mathit{VEnv}(vs) :: s,\ t,\ m\rangle & \Rightarrow & \langle c,\ \mathit{VNum}(n) :: s,\ t,\ m\rangle \\ \langle \mathit{IOp}(+) :: c,\ \mathit{VNum}(n_0) :: \mathit{VNum}(n_1) :: s,\ t,\ m\rangle & \Rightarrow & \langle c,\ \mathit{VNum}(n_0 + n_1) :: s,\ t,\ m\rangle \\ \langle \mathit{IOp}(*) :: c,\ \mathit{VNum}(n_0) :: \mathit{VNum}(n_1) :: s,\ t,\ m\rangle & \Rightarrow & \langle c,\ \mathit{VNum}(n_0 * n_1) :: s,\ t,\ m\rangle \end{array}$$

Figure 3 State transition rules for *INum* and *IOp*.

Instruction: $[\mathit{IPushEnv}_1;\ \mathit{INum}(1);\ \mathit{IPopEnv}_1;\ \mathit{IReset}(c_1);\ \mathit{IOp}_1(+)]$

$$
\begin{array}{rcl}
c_1 & = & [\mathit{IPushEnv}_2;\ \mathit{IControl}_1(c_2);\ \mathit{IPopEnv}_2;\ \mathit{IControl}_2(c_3);\ \mathit{IOp}_2(+)] \\
c_2 & = & [\mathit{IPushEnv}_3;\ \mathit{INum}(2);\ \mathit{IPopEnv}_3;\ \mathit{IPushEnv}_4;\ \mathit{IAccess}(0);\ \mathit{IPopEnv}_4;\ \mathit{INum}(3);\ \mathit{ICall};\ \mathit{IOp}_3(*)] \qquad c_3 \ = \ [\mathit{INum}(4)] \\
c_4 & = & [\mathit{IOp}_1(+)] \qquad c_5 \ = \ [\mathit{IPopEnv}_2;\ \mathit{IControl}_2(c_3);\ \mathit{IOp}_2(+)] \qquad c_6 \ = \ [\mathit{IOp}_3(*)] \qquad c_7 \ = \ [\mathit{IOp}_2(+)] \\
vc_1 & = & \mathit{VContC}((c_5, [\mathit{VEnv}([])]) :: []) \qquad vc_2 \ = \ \mathit{VContC}((c_7, [\mathit{VNum}(3)]) :: (c_6, [\mathit{VNum}(2)]) :: [])
\end{array}
$$

$$
\begin{array}{llrrrr}
 & \langle & \mathit{IPushEnv}_1 :: \ldots, & \mathit{VEnv}([]) :: [], & [], & []\rangle \\
\Rightarrow & \langle & \mathit{INum}(1) :: \ldots, & \mathit{VEnv}([]) :: \mathit{VEnv}([]) :: [], & [], & []\rangle \\
\Rightarrow & \langle & \mathit{IPopEnv}_1 :: \ldots, & \mathit{VNum}(1) :: \mathit{VEnv}([]) :: [], & [], & []\rangle \\
\Rightarrow & \langle & \mathit{IReset}(c_1) :: \ldots, & \mathit{VEnv}([]) :: \mathit{VNum}(1) :: [], & [], & []\rangle \\
\Rightarrow & \langle & \mathit{IPushEnv}_2 :: \ldots, & \mathit{VEnv}([]) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{IControl}_1(c_2) :: \ldots, & \mathit{VEnv}([]) :: \mathit{VEnv}([]) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{IPushEnv}_3 :: \ldots, & \mathit{VEnv}(vc_1 :: []) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{INum}(2) :: \ldots, & \mathit{VEnv}(vc_1 :: []) :: \mathit{VEnv}(vc_1 :: []) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{IPopEnv}_3 :: \ldots, & \mathit{VNum}(2) :: \mathit{VEnv}(vc_1 :: []) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{IPushEnv}_4 :: \ldots, & \mathit{VEnv}(vc_1 :: []) :: \mathit{VNum}(2) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{IAccess}(0) :: \ldots, & \mathit{VEnv}(vc_1 :: []) :: \mathit{VEnv}(vc_1 :: []) :: \mathit{VNum}(2) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{IPopEnv}_4 :: \ldots, & vc_1 :: \mathit{VEnv}(vc_1 :: []) :: \mathit{VNum}(2) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{INum}(3) :: \ldots, & \mathit{VEnv}(vc_1 :: []) :: vc_1 :: \mathit{VNum}(2) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{ICall} :: \ldots, & \mathit{VNum}(3) :: vc_1 :: \mathit{VNum}(2) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{IPopEnv}_2 :: \ldots, & \mathit{VNum}(3) :: \mathit{VEnv}([]) :: [], & (c_6, [\mathit{VNum}(2)]) :: [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{IControl}_2(c_3) :: \ldots, & \mathit{VEnv}([]) :: \mathit{VNum}(3) :: [], & (c_6, [\mathit{VNum}(2)]) :: [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{INum}(4) :: [], & \mathit{VEnv}(vc_2 :: []) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & [], & \mathit{VNum}(4) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{IOp}_1(+) :: [], & \mathit{VNum}(4) :: \mathit{VNum}(1) :: [], & [], & []\rangle \\
\Rightarrow & \langle & [], & \mathit{VNum}(5) :: [], & [], & []\rangle
\end{array}
$$

Figure 4 An example execution of $1 + \langle (\mathcal{F}c.\, 2 \times c\, 3) + \mathcal{F}c'.\, 4\rangle$ on the virtual machine.

Positional Injectivity for Innocent Strategies

Lison Blondeau-Patissier
Université Lyon, EnsL, UCBL, CNRS, LIP, F-69342, Lyon Cedex 07, France

Pierre Clairambault
Université Lyon, EnsL, UCBL, CNRS, LIP, F-69342, Lyon Cedex 07, France

Abstract

In asynchronous games, Melliès proved that innocent strategies are *positional*: their behaviour only depends on the position, not the temporal order used to reach it. This insightful result shaped our understanding of the link between dynamic (*i.e.* game) and static (*i.e.* relational) semantics.

In this paper, we investigate the positionality of innocent strategies in the traditional setting of Hyland-Ong-Nickau-Coquand pointer games. We show that though innocent strategies are not positional, total finite innocent strategies still enjoy a key consequence of positionality, namely *positional injectivity*: they are entirely determined by their positions. Unfortunately, this does not hold in general: we show a counter-example if finiteness and totality are lifted. For finite partial strategies we leave the problem open; we show however the partial result that two strategies with the same positions must have the same P-views of maximal length.

2012 ACM Subject Classification Theory of computation → Denotational semantics

Keywords and phrases Game Semantics, Innocence, Relational Semantics, Positionality

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.17

Related Version *Full Version*: https://arxiv.org/abs/2105.02485

Funding Work supported by the ANR project DyVerSe (ANR-19-CE48-0010-01); and by the Labex MiLyon (ANR-10-LABX-0070) of Université de Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007), operated by the French National Research Agency (ANR).

Acknowledgements We thank the reviewers, whose comments greatly helped improve the paper.

1 Introduction

Game semantics presents higher-order computation interactively as an exchange of tokens in a two-player game between Player (the program under study), and Opponent (its execution environment) [15, 1]. Game semantics has had a strong theoretical impact on denotational semantics, achieving full abstraction results for languages for which other tools struggle.

At the heart of Hyland and Ong's celebrated model [15] are *innocent strategies*, matching *pure* programs. They matter conceptually and technically: many full abstraction results rely on innocent strategies and their definability properties. Accordingly, innocence is perhaps the most studied notion on the foundational side of game semantics, with questions including categorical reconstructions [13], alternative definitions [16, 14], non-deterministic [18, 6], concurrent [7], or quantitative [17, 4] extensions. In particular, our modern understanding of innocence is shaped by Melliès' homotopy-theoretic reformulation in asynchronous games [16]. In this paper, Melliès also introduced an important result: innocent strategies are *positional*.

Positionality is an elementary notion on games on graphs: a strategy is positional if its behaviour only depends on the current node – the "position" – and not the path leading there. In standard game semantics there is, at first sight, no clear notion of position: plays are primitive, and it is not clear what is the ambient graph. In contrast, asynchronous games and relatives (*e.g.* concurrent games) admit a transparent notion of position: two plays reach the same position if they feature the same moves, though not necessarily in the same order. In investigating positionality, Melliès' motivation was to bridge standard play-based game semantics with more static, *relational*-like semantics [2, 12]. Indeed, points of the *web*

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi; Article No. 17; pp. 17:1–17:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

in relational semantics correspond to certain positions in game semantics. Positionality of innocent strategies entails that they are entirely defined by their positions (a property we shall call *positional injectivity*), so that collapsing game to relational semantics corresponds exactly to keeping only certain positions. See [8] for a recent account.

Now, traditional Hyland-Ong arena games are by no means disconnected from those developments: bridges with relational semantics were also investigated there, notably by Boudes [3]. There, points of the web match so-called *thick subtrees*, pomsets representing partial explorations of the arena with duplications. This provides *positions* for Hyland-Ong games. But then, are innocent strategies still positional? Though it came to us as a surprise, it is not hard to find a counter-example. So we focus on the key weakening of the question: are innocent strategies *positionally injective*? Our main result is positive, for *total finite* innocent strategies. We first link Hyland-Ong innocence with an alternative, causal formulation inspired from concurrent games [8], allowing a transparent link between a strategy and its positions. Drawing inspiration from the proof of injectivity of the relational model for MELL proof nets [10], we show how to track down duplications in certain well-engineered positions to recover a sufficient portion of the causal structure; and deduce positional injectivity. However, we show that in the general case (without *finiteness* and *totality*), positional injectivity fails. Finally, for finite (but not total) innocent strategies we show a partial result, namely that two strategies with the same positions have the same P-views of maximal length.

Tsukada and Ong [19] show an injective collapse from a category of innocent strategies onto the relational model. Their collapse is similar to ours, with an important distinction: they label moves in each play, coloring contiguous Opponent/Player pairs identically. Labels survive the collapse, allowing to read back causal links directly. This is possible because the web of atomic types is set to comprise countably many such labels – but then, the correspondence between positions and points of the web is lost. In contrast, our theorem requires us to prove injectivity directly, without such labeling.

In Section 2 we introduce the setting and state our main result. In Section 3 we reformulate the problem via a *causal* presentation of game semantics. In Section 4 we present the proof of positional injectivity for total finite innocent strategies. In Section 5, we show some partial results beyond total finite strategies. Finally, in Section 6, we conclude.

2 Innocent Strategies and Positions

2.1 Arenas and Constructions

We start this paper by giving a definition of *arenas*, which represent *types*.

▶ **Definition 1.** *An **arena** is $A = \langle |A|, \leq_A, \lambda_A \rangle$ where $\langle |A|, \leq_A \rangle$ is a partial order, and $\lambda_A : |A| \to \{-, +\}$ is a **polarity function**. Moreover, these data must satisfy:*

- finitary: *for all $a \in |A|$, $[a]_A = \{a' \in |A| \mid a' \leq_A a\}$ is finite,*
- forestial: *for all $a_1, a_2 \leq_A a$, then $a_1 \leq_A a_2$ or $a_2 \leq_A a_1$,*
- alternating: *for all $a_1 \rightarrowtriangle_A a_2$, then $\lambda_A(a_1) \neq \lambda_A(a_2)$,*
- negative: *for all $a \in \min(A) = \{a \in |A| \mid a \text{ minimal}\}$, $\lambda_A(a) = -$,*

where $a_1 \rightarrowtriangle_A a_2$ means $a_1 <_A a_2$ with no event strictly in between.

Though our notations differ superficially, our arenas are similar to [15]. They present observable computational events (on a given type) along with their causal dependencies: positive moves are due to Player / the program, and negative moves to Opponent / the

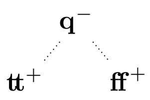


Figure 1 Arena **bool**.

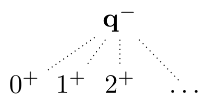


Figure 2 Arena **nat**.

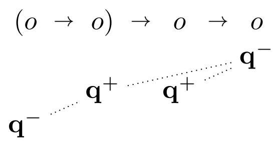


Figure 3 Arena $(o \Rightarrow o) \Rightarrow o \Rightarrow o$.

environment. We show in Figures 1 and 2, read from top to bottom, the representation of the datatypes **bool** and **nat** as arenas. Opponent initiates the execution with $\mathbf{q}^-$, annotated so as to indicate its polarity, and Player may respond any possible value, with a positive move.

We write 1 for the empty arena and o for the arena with exactly one (negative) move. More elaborate types involve matching constructions: the *product* and the *arrow*.

▶ **Definition 2.** *Consider A_1 and A_2 arenas. Then, we define $A_1 \parallel A_2$ as*

$$\begin{array}{rcl} |A_1 \parallel A_2| & = & (\{1\} \times |A_1|) \cup (\{2\} \times |A_2|) \\ (i,a) \leq_{A_1 \parallel A_2} (j,b) & \Leftrightarrow & i = j \quad \wedge \quad a \leq_{A_i} b \\ \lambda_{A_1 \parallel A_2}(i,a) & = & \lambda_{A_i}(a)\,, \end{array}$$

called their ***parallel composition*** *or* ***product****, and also written $A_1 \times A_2$.*

For any family $(A_i)_{i \in I}$ of arenas, this extends to $\prod_{i \in I} A_i$ in the obvious way. Any arena A decomposes (up to forest iso) as $A \cong \prod_{i \in I} A_i$ for some family $(A_i)_{i \in I}$ of arenas which are **well-opened**, *i.e.* with *exactly one* initial (*i.e. minimal*) move. We now define the *arrow*:

▶ **Definition 3.** *Consider A_1, A_2 arenas with A_2 well-opened. Then $A_1 \Rightarrow A_2$ has:*

$$\begin{array}{rcl} |A_1 \Rightarrow A_2| & = & (\{1\} \times |A_1|) \cup (\{2\} \times |A_2|) \\ (i,a) \leq_{A_1 \Rightarrow A_2} (j,b) & \Leftrightarrow & (i = j \wedge a \leq_{A_i} b) \vee (i = 2 \wedge a \in \min(A_2)) \\ \lambda_{A_1 \Rightarrow A_2}(i,a) & = & (-1)^i \cdot \lambda_{A_i}(a) \end{array}$$

This extends to all arenas with $A \Rightarrow \prod_{i \in I} B_i = \prod_{i \in I} A \Rightarrow B_i$ and $A \Rightarrow 1 = 1$.

We will mostly use $A \Rightarrow B$ for B well-opened. Figure 3 displays $(o \Rightarrow o) \Rightarrow o \Rightarrow o$, matching the simple type $(o \to o) \to o \to o$ with atomic type o – the position of moves follows a correspondence between those and atoms of the type. These arena constructions describe call-by-name computation: once Opponent initiates computation with $\mathbf{q}^-$, two Player moves become available. Player may call the second argument (terminating computation) or evaluate the first argument, which in turn allows Opponent to call its argument.

2.2 Plays and Strategies

In Hyland-Ong games, players are allowed to *backtrack*, and resume the play from any earlier stage. This is made formal by the notion of *pointing strings*:

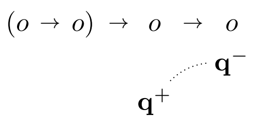
▶ **Definition 4.** *A* ***pointing string*** *over set Σ is a string $s \in \Sigma^*$, where each move may additionally come equipped with a* ***pointer*** *to an earlier move.*

We often write $s = s_1 \dots s_n$ for pointing strings, leaving pointers implicit.

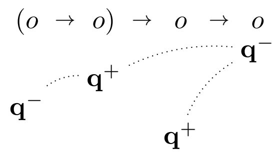
▶ **Definition 5.** *A* ***play*** *on arena A is a pointing string $s = s_1 \dots s_n$ over $|A|$ s.t.:*

rigid:	*If s_i points to s_j, then $s_j \rightarrowtriangle_A s_i$,*
alternating:	*for all $1 \leq i < n$, $\lambda_A(s_i) \neq \lambda_A(s_{i+1})$,*
legal:	*for all $1 \leq i \leq n$, either $s_i \in \min(A)$ or s_i has a pointer.*

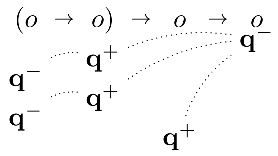
A play is ***well-opened*** *iff it has exactly one initial move. We write* $\mathsf{Plays}(A)$ *for the set of plays on A,* $\mathsf{Plays}^+(A)$ *for even-length plays, and* $\mathsf{Plays}_\bullet(A)$ *for well-opened plays.*



■ **Figure 4** $\lambda f^{o\to o}.\, \lambda x^o.\, x$.



■ **Figure 5** $\lambda f^{o\to o}.\, \lambda x^o.\, f\, x$.



■ **Figure 6** $\lambda f^{o\to o}.\, \lambda x^o.\, f\,(f\, x)$.

We write ε for the empty play, $\sqsubseteq$ for the prefix, and $\sqsubseteq^+$ if the smaller play has even length. Plays represent higher-order executions. Figures 4, 5 and 6 show plays on the arena of Figure 3; matching typical executions of the corresponding simply-typed λ-term. They are read from top to bottom, with pointers as dotted lines. As in Figure 3, the position of moves encodes their identity in the arena. Strategies, representing programs, are sets of plays:

▶ **Definition 6.** *A strategy $\sigma : A$ on arena A is a non-empty set $\sigma \subseteq \mathsf{Plays}^+(A)$ satisfying*

$$\begin{array}{rl} \text{prefix-closed:} & \forall s \in \sigma, \forall t \sqsubseteq^+ s, t \in \sigma\,, \\ \text{deterministic:} & \forall s \in \sigma,\ \ sab, sab' \in \sigma \implies sab = sab'\,. \end{array}$$

Implicit in the last clause is that sab and sab' also have the same pointers.

2.3 Visibility and Innocence

Innocence captures that the behaviour only depends on which program phrase currently has control. Intuitively, the "current program phrase" is captured by the *P-view*.

▶ **Definition 7.** *For any arena A, we set a partial function $\ulcorner - \urcorner : \mathsf{Plays}(A) \rightharpoonup \mathsf{Plays}(A)$ as:*

$$\begin{array}{rcll} \ulcorner si \urcorner &=& i & \text{if } i \in \min(A), \\ \ulcorner sn^- m^+ \urcorner &=& \ulcorner sn \urcorner m & \text{if the pointer of } m \text{ is in } \ulcorner sn \urcorner, \\ \ulcorner sn^+ t m^- \urcorner &=& \ulcorner sn \urcorner m & \text{if } m \text{ points to } n, \end{array}$$

undefined otherwise. In the last two cases, m keeps its pointer in the resulting play.
*If defined, $\ulcorner s \urcorner$ is the **P-view** of s. A play $s \in \mathsf{Plays}(A)$ is **visible** iff $\forall t \sqsubseteq s$, $\ulcorner t \urcorner$ is defined.*

We say that $s \in \mathsf{Plays}(A)$ is a **P-view** iff $\ulcorner s \urcorner = s$. A strategy $\sigma : A$ is **visible** iff any $s \in \sigma$ is visible. In that case, P-views are always well-defined, so that we may formulate:

▶ **Definition 8.** *A strategy $\sigma : A$ is **innocent** iff it is visible, and satisfies:*

innocence: *for all $sab, t \in \sigma$, if $ta \in \mathsf{Plays}(A)$ and $\ulcorner sa \urcorner = \ulcorner ta \urcorner$, then $tab \in \sigma$.*

where, in tab, b points "as in sab", i.e. *so as to ensure that $\ulcorner sab \urcorner = \ulcorner tab \urcorner$.*

An innocent $\sigma : A$ is determined by $\ulcorner\!\ulcorner \sigma \urcorner\!\urcorner = \{\ulcorner s \urcorner \mid s \in \sigma\}$, its *P-view forest*. Figures 4, 5 and 6 present P-views, each inducing an innocent strategy *via* the P-view forest obtained by even-length prefix closure. Likewise, Figures 7 and 8 induce strategies for the so-called simply-typed "Kierstead terms" $\lambda f^{(o\to o)\to o}.\, f\,(\lambda x^o.\, f\,(\lambda y^o.\, x))$ and $\lambda f^{(o\to o)\to o}.\, f\,(\lambda x^o.\, f\,(\lambda y^o.\, y))$. P-views are well-opened, so innocent strategies are determined by their set $\sigma_\bullet$ of well-opened plays.

Innocent strategies form a cartesian closed category Inn with as objects arenas, and morphisms from A to B the innocent strategies $\sigma : A \Rightarrow B$. Composing $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$ involves a "parallel interaction plus hiding" mechanism, which we omit [15].

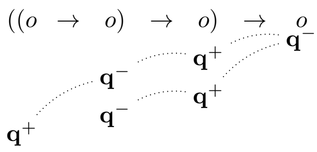


Figure 7 $K_x : ((o \to o) \to o) \to o$.

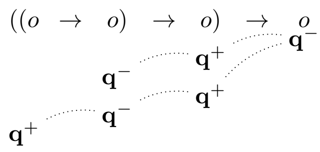


Figure 8 $K_y : ((o \to o) \to o) \to o$.

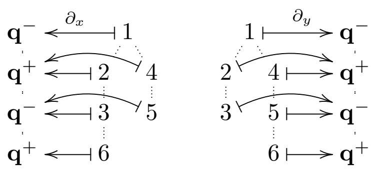


Figure 9 Deseq. K_x and K_y.

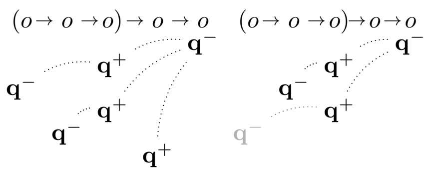


Figure 10 Non-positionality of innocence.

2.4 Positions

Boudes' "thick subtrees" [3], called *positions* in this paper, are the central concept informing the link between innocent game semantics and relational semantics. They are simply desequentialized plays, or in other words prefixes of the arena with duplications.

To introduce positions, our first stop is the following notion of *configuration*.

▶ **Definition 9.** *A **configuration** $x \in \mathscr{C}(A)$ of arena A is a tuple $x = \langle |x|, \leq_x, \partial_x \rangle$ such that $\langle |x|, \leq_x \rangle$ is a finite tree, and $\partial_x : |x| \to |A|$, the **display map**, is a labeling function s.t.:*

minimality-respecting: *for all $a \in |x|$, a is $\leq_x$-minimal iff $\partial_x(a)$ is $\leq_A$-minimal,*
causality-preserving: *for all $a_1, a_2 \in |x|$, if $a_1 \rightarrowtriangle_x a_2$ then $\partial_x(a_1) \rightarrowtriangle_A \partial_x(a_2)$.*

We call **events** the elements of $|x|$. Note $\langle |x|, \leq_x \rangle$ has exactly one minimal event, which suffices as innocent strategies are determined by well-opened plays. Configurations include:

▶ **Definition 10.** *The **desequentialization** $(\!| s |\!) \in \mathscr{C}(A)$ of $s = s_1 \dots s_n \in \mathsf{Plays}_\bullet(A)$ has $|(\!| s |\!)| = \{1, \dots, n\}$, $\partial_{(\!| s |\!)}(i) = s_i$, and $i \leq_{(\!| s |\!)} j$ if there is a chain of pointers from s_j to s_i in s.*

We show in Figure 9 the desequentialization of the maximal P-views of K_x and K_y from Figures 7 and 8. Extracting $(\!| s |\!)$ is a first step, we must then forget the identity of its events:

▶ **Definition 11.** *A bijection $\varphi : |x| \cong |y|$ is an **isomorphism** $\varphi : x \cong y$ iff it is*

arena-preserving: *for all $a \in |x|$, $\partial_y(\varphi(a)) = \partial_x(a)$,*
causality-respecting: *for all $a_1, a_2 \in |x|$, we have $a_1 \rightarrowtriangle_x a_2$ iff $\varphi(a_1) \rightarrowtriangle_y \varphi(a_2)$.*

*A **position** of A, written $\mathsf{x} \in (\!(A)\!)$, is an isomorphism class of configurations.*

If $s \in \mathsf{Plays}_\bullet(A)$, the **position** $(\!(s)\!) \in (\!(A)\!)$ is the isomorphism class of $(\!| s |\!)$.

We pause to consider the *positionality of innocent strategies* as mentioned in the introduction. Though it will only play a very minor role, we define *positional* strategies:

▶ **Definition 12.** *Consider $\sigma : A$ a strategy on A. We set the condition:*

positional: $\forall sab, t \in \sigma,\ ta' \in \mathsf{Plays}(A),\ (\!(sa)\!) = (\!(ta')\!) \implies \exists ta'b \in \sigma, (\!(sab)\!) = (\!(ta'b)\!)$.

Innocent strategies are not positional: Figure 10 displays (the two maximal P-views of) the innocent strategy for the λ-term $\lambda f^{o\to o\to o}.\, \lambda x^o.\, f\,(f \perp x)\,(f \perp \perp)$. On the right hand side, the last Opponent move is grayed out as an extension of a P-view triggering no response. After the fifth move the position is the same, contradicting positionality. In Melliès' asynchronous games [16], explicit copy indices help distinguish the two calls to f. The two plays no longer reach the same position, restoring positionality. But even in asynchronous games, if positions were quotiented by symmetry so as to match relational semantics, positionality would fail.

We turn to the weaker *positional injectivity.* If $\sigma : A$, its **positions** are those reached by well-opened plays, *i.e.* $\llparenthesis \sigma \rrparenthesis = \{ \llparenthesis s \rrparenthesis \mid s \in \sigma_\bullet \} \subseteq \llparenthesis A \rrparenthesis$. We may finally ask our main question:

▶ **Question 13** (Positional Injectivity). *If σ, τ are innocent and $\llparenthesis \sigma \rrparenthesis = \llparenthesis \tau \rrparenthesis$, do we have $\sigma = \tau$?*

2.5 Links with the Relational Model

To fully appreciate this question, it is informative to consider the link with the relational model. We start with the following observation concerning positions on the arrow arena.

▶ **Fact 14.** *Consider A and B arenas, and write $\mathcal{M}_f(X)$ for the **finite multisets** on X. Then, we have a bijection $\llparenthesis A \Rightarrow B \rrparenthesis \cong \mathcal{M}_f(\llparenthesis A \rrparenthesis) \times \llparenthesis B \rrparenthesis$.*

Recall [12] that the relational model forms a cartesian closed category $\mathsf{Rel}_!$ having *sets* as objects; and as morphisms from A to B the *relations* $R \subseteq \mathcal{M}_f(A) \times B$. Considering simple types generated from o and the arrow $A \to B$, and setting the relational interpretation of o as $[\![o]\!]_{\mathsf{Rel}_!} = \{\mathbf{q}\}$, then for any type A, there is a bijection $r_A : \llparenthesis [\![A]\!]_{\mathsf{Inn}} \rrparenthesis \cong [\![A]\!]_{\mathsf{Rel}_!}$.

▶ **Theorem 15.** *This extends to a functor $\llparenthesis - \rrparenthesis : \mathsf{Inn} \to \mathsf{Rel}_!$, which preserves the interpretation: for any term $M : A$ of the simply-typed λ-calculus, $r_A(\llparenthesis [\![M]\!]_{\mathsf{Inn}} \rrparenthesis) = [\![M]\!]_{\mathsf{Rel}_!}$.*

This *relational collapse* of innocent strategies has been studied extensively [3, 16, 19, 4, 9]. The inclusion $\subseteq$ is easy; the difficulty in proving $\supseteq$ is that game-semantic interaction is temporal: positions arising relationally might, in principle, fail to appear game-semantically because reproducing them yields a deadlock. For innocent strategies this does not happen: this may be proved through connections with syntax [3, 19] or semantically [4, 9].

In [19], Tsukada and Ong prove a similar collapse injective. This seems to answer Question 13 positively – but this is not so simple. The interpretation in $\mathsf{Rel}_!$ is parametrized by a set X for the ground type o. In [19], X is required to be countably infinite: this way one allocates one tag for each pair of chronologically contiguous O/P moves, encoding the *causal / axiom links.* In contrast, for Question 13 we are forced to interpret o with a singleton set $\{\mathbf{q}\}$, or lose the correspondence between points of the web and positions. We must reconstruct strategies directly from their desequentializations, with no help from labeling or coloring.

2.6 Main result

At first this seems desperate. In [19], an innocent strategy may already be reconstructed from the desequentialization of its P-views. But here, the two plays of Figures 7 and 8 yield the configurations of Figure 9, which are isomorphic – so give the same position. Nevertheless K_x and K_y *can* be distinguished, via their behaviour under replication. In both plays of Figure 11, we replay the move to which the deepest $\mathbf{q}^+$ points. This brings K_x and K_y to react differently, obtaining plays whose positions separate $\llparenthesis K_x \rrparenthesis$ and $\llparenthesis K_y \rrparenthesis$. So, by observing the behaviour of a strategy under replication, we can infer some temporal information.

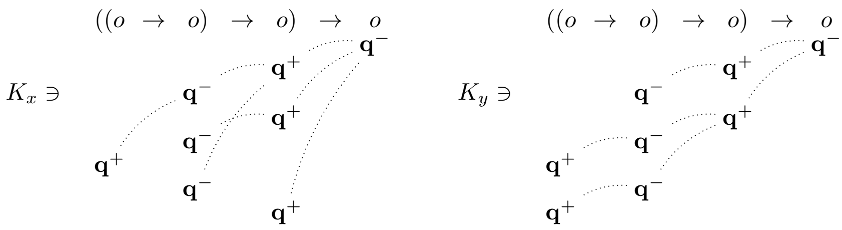


Figure 11 Plays yielding positions distinguishing K_x and K_y.

Most of the paper will be devoted to turning this idea into a proof. However, we have only been able to prove the result with the following additional restrictions on strategies.

▶ **Definition 16.** *For A an arena, we define conditions on innocent strategies $\sigma : A$ as:*

total: *for all $s \in \sigma$, if $sa \in \mathsf{Plays}(A)$ then there exists b such that $sab \in \sigma$,*
finite: *the set $\ulcorner\!\ulcorner \sigma \urcorner\!\urcorner = \{\ulcorner s \urcorner \mid s \in \sigma\}$ is finite.*

Total finite strategies are already well-known: on arenas interpreting simple types they exactly correspond to β-normal η-long normal forms of simply-typed λ-terms.

We now state our main result, **positional injectivity**:

▶ **Theorem 17.** *For any $\sigma, \tau : A$ innocent total finite, $\sigma = \tau$ iff $(\!|\sigma|\!) = (\!|\tau|\!)$.*

As observed in Section 2.1, all arenas decompose as $A = \prod_{i \in I} A_i$ with A_i well-opened. As $\times$ is a cartesian product in Inn, strategies $\sigma : A$ also decompose as $\sigma = \langle \sigma_i \mid i \in I \rangle$ with $\sigma_i : A_i$ for all $i \in I$. From innocence it follows that $(\!|\langle \sigma_i \mid i \in I \rangle|\!) \cong \sum_{i \in I} (\!|\sigma_i|\!)$, so it suffices to prove Theorem 17 for A well-opened. From now on, we consider all arenas well-opened.

3 Causal Presentation

Besides the behaviour of strategies under replication, plays also include the order, irrelevant for our purposes, in which branches are explored by Opponent. To isolate the effect of replication, we introduce a *causal* version of strategies inspired from *concurrent games* [5].

3.1 Augmentations

This formulation rests on the notion of *augmentations*. Intuitively those correspond to expanded trees of P-views, which enrich configurations with causal wiring from the strategy.

▶ **Definition 18.** *An **augmentation** on arena A is a tuple $\mathcal{q} = \langle |\mathcal{q}|, \leq_{(\!|\mathcal{q}|\!)}, \leq_{\mathcal{q}}, \partial_{\mathcal{q}} \rangle$, where $(\!|\mathcal{q}|\!) = \langle |\mathcal{q}|, \leq_{(\!|\mathcal{q}|\!)}, \partial_{\mathcal{q}} \rangle \in \mathscr{C}(A)$, and $\langle |\mathcal{q}|, \leq_{\mathcal{q}} \rangle$ is a tree satisfying:*

rule-abiding: *for all $a_1, a_2 \in |\mathcal{q}|$, if $a_1 \leq_{(\!|\mathcal{q}|\!)} a_2$, then $a_1 \leq_{\mathcal{q}} a_2$,*
courteous: *for all $a_1 \rightarrowtriangle_{\mathcal{q}} a_2$, if $\lambda(a_1) = +$ or $\lambda(a_2) = -$, then $a_1 \rightarrowtriangle_{(\!|\mathcal{q}|\!)} a_2$,*
deterministic: *for all $a^- \rightarrowtriangle_{\mathcal{q}} a_1^+$ and $a^- \rightarrowtriangle_{\mathcal{q}} a_2^+$, then $a_1 = a_2$,*

*we then write $\mathcal{q} \in \mathsf{Aug}(A)$, and call $(\!|\mathcal{q}|\!) \in \mathscr{C}(A)$ the **desequentialization** of $\mathcal{q}$.*

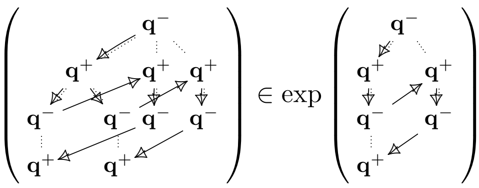


Figure 12 Causal K_x and its expansion.

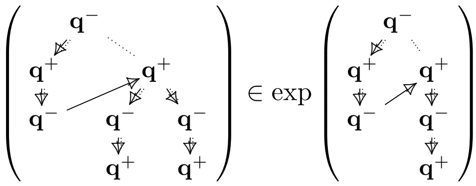


Figure 13 Causal K_y and its expansion.

Events of $|\mathscr{q}|$ inherit a polarity with $\lambda(a) = \lambda_A(\partial_{\mathscr{q}}(a))$. By *rule-abiding* and *courteous*, $\langle |\mathscr{q}|, \leq_{\mathscr{q}}\rangle$ and $\langle |\mathscr{q}|, \leq_{\llparenthesis \mathscr{q} \rrparenthesis}\rangle$ have the same minimal event $\mathsf{init}(\mathscr{q})$, called the **initial** event. If $a \in |\mathscr{q}|$ is not initial, there is a unique $a' \in |\mathscr{q}|$ such that $a' \rightarrowtriangle_{\mathscr{q}} a$, written $a' = \mathsf{pred}(a)$ and called the **predecessor** of a. Likewise, a non-initial $a \in |\mathscr{q}|$ also has a unique $a'' \in |\mathscr{q}|$ such that $a'' \rightarrowtriangle_{\llparenthesis \mathscr{q} \rrparenthesis} a$, written $a'' = \mathsf{just}(a)$ and called the **justifier** of a. By *courteous* and as immediate causality alternates in A (and hence in $\llparenthesis \mathscr{q} \rrparenthesis$), both $\mathsf{pred}(a)$ and $\mathsf{just}(a)$ have polarity opposite to a. They may not coincide, however from *courteous* they do for a negative.

Figures 12 and 13 show augmentations – though the corresponding definitions remain to be seen, those are the causal expansions of K_x and K_y matching the plays of Section 2.6. In such diagrams, immediate causality from the configuration appears as dotted lines, whereas that coming from the augmentation itself appears as $\rightarrowtriangle$. We set a few auxiliary conditions:

▶ **Definition 19.** *Let $\mathscr{q} \in \mathsf{Aug}(A)$ be an augmentation. We set the conditions:*

receptive:	*for all $a \in |\mathscr{q}|$, if $\partial_{\mathscr{q}}(a) \rightarrowtriangle_A b^-$, there is $a \rightarrowtriangle_{\mathscr{q}} b'$ such that $\partial_{\mathscr{q}}(b') = b$,*
$+$-covered:	*for all $a \in |\mathscr{q}|$ maximal in $\mathscr{q}$, we have $\lambda(a) = +$,*
$-$-linear:	*for all $a \rightarrowtriangle_{\mathscr{q}} a_1^-$, $a \rightarrowtriangle_{\mathscr{q}} a_2^-$, if $\partial_{\mathscr{q}}(a_1) = \partial_{\mathscr{q}}(a_2)$ then $a_1 = a_2$.*

We say that $\mathscr{q} \in \mathsf{Aug}(A)$ is **total** iff it is receptive and $+$-covered. We will also refer to receptive $-$-linear augmentations as **causal strategies**.

3.2 From Strategies to Causal Strategies

We may easily represent an innocent strategy as a causal strategy:

▶ **Proposition 20.** *For $\sigma : A$ finite innocent on A well-opened, we set components*

$$|\hat{\sigma}| = \{\ulcorner s \urcorner \mid s \in \sigma \wedge s \neq \varepsilon\} \cup \{\ulcorner sa \urcorner \mid s \in \sigma \wedge sa \in \mathsf{Plays}(A)\}\,,$$

$s \leq_{\hat{\sigma}} t$ iff $s \sqsubseteq t$, $sa \leq_{\llparenthesis \hat{\sigma} \rrparenthesis} satb$ iff there is a chain of justifiers from b to a, and $\partial_{\hat{\sigma}}(sa) = a$.
Then $\hat{\sigma} = \langle |\hat{\sigma}|, \leq_{\hat{\sigma}}, \leq_{\llparenthesis \hat{\sigma} \rrparenthesis}, \partial_{\hat{\sigma}}\rangle \in \mathsf{Aug}(A)$ is a causal strategy, and is total iff σ is total.

The proof is a straightforward verification. As for configurations, so as to forget the concrete identity of events we consider augmentations up to *isomorphism*:

▶ **Definition 21.** *A **morphism** $\varphi : \mathscr{q} \to \mathscr{p}$ is a function $\varphi : |\mathscr{q}| \to |\mathscr{p}|$ satisfying:*

arena-preserving:	*$\partial_{\mathscr{p}} \circ \varphi = \partial_{\mathscr{q}}$,*
causality-preserving:	*for all $a_1, a_2 \in |\mathscr{q}|$, if $a_1 \rightarrowtriangle_{\mathscr{q}} a_2$ then $\varphi(a_1) \rightarrowtriangle_{\mathscr{p}} \varphi(a_2)$,*
configuration-preserving:	*for all $a_1, a_2 \in |\mathscr{q}|$, if $a_1 \rightarrowtriangle_{\llparenthesis \mathscr{q} \rrparenthesis} a_2$ then $\varphi(a_1) \rightarrowtriangle_{\llparenthesis \mathscr{p} \rrparenthesis} \varphi(a_2)$.*

*An **isomorphism** is an invertible morphism – we then write $\varphi : \mathscr{q} \cong \mathscr{p}$.*

Note that by *arena-preserving*, φ must send $\mathsf{init}(\mathcal{q})$ to $\mathsf{init}(\mathcal{p})$.

The reader may check that the construction of Proposition 20 applied to K_x and K_y yields, up to isomorphism, the (small) augmentations of Figures 12 and 13. The next fact shows that augmentations are indeed an alternative presentation of innocent strategies.

▶ **Lemma 22.** *For any finite innocent strategies σ, τ on arena A, then $\sigma = \tau$ iff $\hat{\sigma} \cong \hat{\tau}$.*

Proof. Clearly, $\sigma = \tau$ implies $\hat{\sigma} = \hat{\tau}$. Conversely, assume $\varphi : \hat{\sigma} \cong \hat{\tau}$. Take $s = s_1 \dots s_n \in \ulcorner\sigma\urcorner$, and write $s_{\leq i} = s_1 \dots s_i$. Then we have a chain $s_{\leq 1} \twoheadrightarrow_{\hat{\sigma}} s_{\leq 2} \twoheadrightarrow_{\hat{\sigma}} \dots \twoheadrightarrow_{\hat{\sigma}} s_{\leq n-1} \twoheadrightarrow_{\hat{\sigma}} s$, transported through φ to $t_{\leq 1} \twoheadrightarrow_{\hat{\tau}} \dots \twoheadrightarrow_{\hat{\tau}} t$. By *arena-preserving*, $t_i = s_i$ for all $1 \leq i \leq n$. Finally by *configuration-preserving*, s and t have the same pointers, hence $s = t$ and $s \in \tau$. Symmetrically, any P-view $t \in \ulcorner\tau\urcorner$ is in σ, hence $\ulcorner\sigma\urcorner = \ulcorner\tau\urcorner$ and $\sigma = \tau$ by innocence. ◀

3.3 Expansions of Causal Strategies

Besides including representations of innocent strategies, augmentations can also represent their *expansions*, *i.e.* arbitrary plays, with Opponent's scheduling factored out.

▶ **Definition 23.** *Consider A an arena, and $\mathcal{p} \in \mathsf{Aug}(A)$ a causal strategy.*
An ***expansion*** *of $\mathcal{p}$, written $\mathcal{q} \in \exp(\mathcal{p})$, is $\mathcal{q} \in \mathsf{Aug}(A)$ such that:*

simulation: *there is a (necessarily unique) morphism $\varphi : \mathcal{q} \to \mathcal{p}$,*
+-obsessional: *for all $a^- \in |\mathcal{q}|$ and $\varphi(a^-) \twoheadrightarrow_{\mathcal{p}} b^+$, there is $a^- \twoheadrightarrow_{\mathcal{q}} a'$ s.t. $\varphi(a') = b^+$.*

The relationship between a causal strategy $\mathcal{p}$ and $\mathcal{q} \in \exp(\mathcal{p})$ is analogous to that between an arena A and a configuration $x \in \mathscr{C}(A)$: $\mathcal{q}$ explores a prefix of $\mathcal{p}$, possibly visiting the same branch many times. However, *determinism* ensures that only Opponent may cause duplications, and $+$-*obsessional* ensures that only Opponent may refuse to explore certain branches – if a Player move is available in $\mathcal{p}$, then it must appear in all corresponding branches of $\mathcal{q}$. Uniqueness of the morphism follows from $-$-linearity and determinism. Figures 12 and 13 show expansions of (the causal strategies corresponding to) K_x and K_y.

Now, we set $(\!|\mathcal{p}|\!) = \{(\!|\mathcal{q}|\!) \mid \mathcal{q} \in \exp(\mathcal{p})\}$ the **positions** of a causal strategy $\mathcal{p}$, where $(\!|\mathcal{q}|\!)$ is the isomorphism class of $(\!|\mathcal{q}|\!)$. By Lemma 22, any innocent $\sigma : A$ yields a causal strategy $\hat{\sigma} : A$, so this leaves us with the task to prove that the two notions of position coincide.

▶ **Proposition 24.** *For any total finite innocent strategy $\sigma : A$, we have $(\!|\sigma|\!) = (\!|\hat{\sigma}|\!)$.*

Proof. Any $\mathsf{x} \in (\!|\sigma|\!)$ is the isomorphism class of $(\!|s|\!)$ for $s = s_1 \dots s_n \in \sigma$. We build an expansion $\mathcal{q}(s) \in \exp(\hat{\sigma})$ as follows. Its configuration is $(\!|\mathcal{q}(s)|\!) = (\!|s|\!)$ (see Definition 10) with events $|\mathcal{q}(s)| = \{1, \dots, n\}$. Its causal order is $i \leq_{\mathcal{q}(s)} j$ iff $j \geq i$ and s_i is reached in the computation of $\ulcorner s_{\leq j}\urcorner$. To show that $\mathcal{q}(s) \in \exp(\hat{\sigma})$ we must provide a morphism $\varphi : \mathcal{q}(s) \to \hat{\sigma}$, which is simply $\varphi(i) = \ulcorner s_{\leq i}\urcorner$. So, $\mathsf{x} = (\!|\mathcal{q}(s)|\!) \in (\!|\hat{\sigma}|\!)$.

Reciprocally, take $\mathsf{x} \in (\!|\hat{\sigma}|\!)$, obtained as the isomorphism class of some $(\!|\mathcal{q}|\!)$, for $\mathcal{q} \in \exp(\hat{\sigma})$. From the totality of σ, $\mathcal{q}$ has maximal events all positive – it has exactly as many Player as Opponent events, and admits a linear extension $s = s_1 \dots s_n$ which is *alternating*, *i.e.* $\lambda(s_i) \neq \lambda(s_{i+1})$ for all $1 \leq i \leq n-1$. Besides, for any $1 \leq i \leq n$, $\ulcorner s_{\leq i}\urcorner$ (treating s as a play on arena $(\!|\mathcal{q}|\!)$) coincides with $[s_i]_{\mathcal{q}} = \{s \in |\mathcal{q}| \mid s \leq_{\mathcal{q}} s_i\}$, totally ordered by $\leq_{\mathcal{q}}$. So, writing $\partial_{\mathcal{q}}(s) = \partial_{\mathcal{q}}(s_1) \dots \partial_{\mathcal{q}}(s_n) \in \mathsf{Plays}(A)$ with pointers inherited from $(\!|\mathcal{q}|\!)$, $\ulcorner\partial_{\mathcal{q}}(s)_{\leq i}\urcorner \in \ulcorner\sigma\urcorner$, hence $\partial_{\mathcal{q}}(s) \in \sigma$ by innocence and $(\!|\partial_{\mathcal{q}}(s)|\!) \cong (\!|s|\!)$. Therefore, $(\!|\mathcal{q}|\!) = (\!|\partial_{\mathcal{q}}(s)|\!) \in (\!|\sigma|\!)$. ◀

The idea is that plays in σ are exactly linearizations of expansions of $\hat{\sigma}$. From a play we get an expansion by factoring out Opponent's scheduling, mimicking the construction of P-views while keeping duplicated branches separate. Reciprocally, an expansion allows

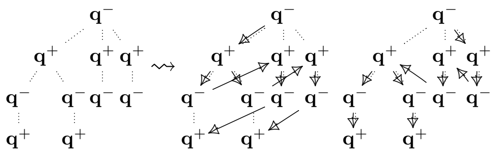


Figure 14 Non-unique causal explanation.

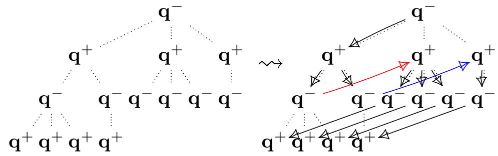


Figure 15 Unique causal explanation.

many (alternating) linearizations. For instance, the two plays of Section 2.6 are respectively linearizations of the expansions of Figures 12 and 13. This proposition fails if σ is not total, as expansions may then have trailing Opponent moves, preventing an alternating linearization.

Thanks to Proposition 24, we focus on positions reached by expansions of causal strategies.

4 Positional Injectivity

We now come to the main contribution of this paper, the proof of positional injectivity for total finite causal strategies. We start this section by introducing the proof idea.

4.1 Forks and Characteristic Expansions

Just from the static snapshot offered by positions, we must deduce the strategy.

Given $z \in \mathscr{C}(A)$, can we uniquely reconstruct its *causal explanation*, *i.e.* $\mathscr{q} \in \mathsf{Aug}(A)$ such that $z = ⦇\mathscr{q}⦈$? In general, there is no reason why $\mathscr{q}$ would be uniquely determined. Indeed, in Figure 14, we show on the left hand side the configuration z_1 underlying Figure 12 – up to iso it has exactly two causal explanations, shown on the right. The rightmost augmentation is not an expansion of K_x, so K_x is not the only strategy featuring (the isomorphism class of) z_1. However, we *can* find a position unique to K_x. Consider z_2 the configuration on the left hand side of Figure 15. The only possible augmentation (up to iso) yielding z_2 as a desequentialization appears on the right hand side (call it $\mathscr{q}$): every other attempt to guess causal wiring fails. In particular, the red and blue immediate causal links are forced by the cardinality of the subsequent duplications. But $\mathscr{q}$ is an expansion of the unique maximal branch of K_x – so it suffices to see z_2 in $⦇\sigma⦈$ to know that $\sigma = K_x$.

This suggests a proof idea: given $\mathscr{p}_1, \mathscr{p}_2 : A$ causal strategies with $⦇\mathscr{p}_1⦈ = ⦇\mathscr{p}_2⦈$, we devise a *characteristic expansion* of $\mathscr{p}_1$ with duplications chosen to make the causal structure essentially unique; meaning it must be an expansion of $\mathscr{p}_2$ as well. We do this by using:

▶ **Definition 25.** *A **fork** in $\mathscr{q} \in \mathsf{Aug}(A)$ is a maximal non-empty set $X \subseteq |\mathscr{q}|$ s.t.:*

negative: *for all $a \in X$, $\lambda(a) = -$,*
sibling: *$X = \{\mathsf{init}(\mathscr{q})\}$ or there is $b \in |\mathscr{q}|$ such that for all $a \in X$, $b \rightarrowtriangle_{\mathscr{q}} a$,*
identical: *for all $a_1, a_2 \in X$, $\partial_{\mathscr{q}}(a_1) = \partial_{\mathscr{q}}(a_2)$.*

We write $\mathsf{Fork}(\mathscr{q})$ for the set of forks in augmentation $\mathscr{q}$.

If $\mathscr{p}$ is a causal strategy, $\mathscr{q} \in \exp(\mathscr{p})$ and $X \in \mathsf{Fork}(\mathscr{q})$, the definition of expansions ensures that all Player moves caused by Opponent moves in X are copies. So if X has **cardinality** $\sharp X = n$, and if we find exactly one set of cardinality $\geq n$ of equivalent Player moves in $⦇\mathscr{q}⦈$, we may deduce that there is a causal link. For instance, in Figure 15, the causal successors for the fork of cardinality 3 may be found so. In general though, several

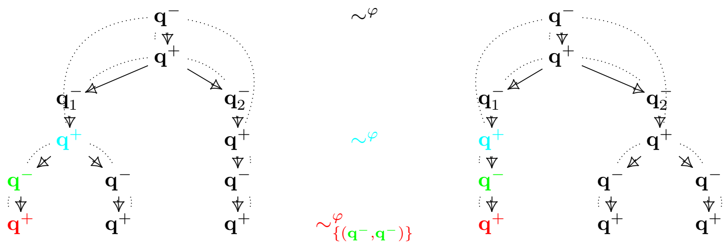


Figure 16 Distinct characteristic expansions reaching the same position.

Opponent moves may cause indistinguishable Player moves, so that the cardinality of a set Y of duplicated Player moves is the sum of the cardinalities of the predecessor forks. To allow us to identify these predecessor sets uniquely, the trick is to construct the expansion so that all forks have cardinality a distinct power of 2, making it so that the predecessor forks can be inferred from the binary decomposition of $\sharp Y$. This brings us to the following definition.

▶ **Definition 26.** *A* ***characteristic expansion*** *of $\mathscr{p}$ is $\mathscr{q} \in \exp(\mathscr{p})$ such that:*

injective:	*for $X, Y \in \mathsf{Fork}(\mathscr{q})$, if $\sharp X = \sharp Y$ then $X = Y$,*
well-powered:	*for all $X \in \mathsf{Fork}(\mathscr{q})$, there is $n \in \mathbb{N}$ such that $\sharp X = 2^n$,*
−-obsessional:	*for all $a^+ \in \lvert\mathscr{q}\rvert$, if $\partial_{\mathscr{q}}(a^+) \rightarrow_A b^-$, there is $a^+ \rightarrow_{\mathscr{q}} a'$ s.t. $\partial_{\mathscr{q}}(a') = b^-$.*

This only constrains causal links in $\mathscr{q}$ from positives to negatives, but by *courteous* those are in $\mathscr{q}$ iff they are in $(\!|\mathscr{q}|\!)$. So for $\mathscr{q} \in \exp(\mathscr{p})$, that it is a characteristic expansion is in fact a property of $(\!|\mathscr{q}|\!)$. Furthermore it is stable under iso so that if $(\!|\mathscr{p}_1|\!) = (\!|\mathscr{p}_2|\!)$, for $\mathscr{q}_1 \in \exp(\mathscr{p}_1)$ characteristic there must be $\mathscr{q}_2 \in \exp(\mathscr{p}_2)$ characteristic too such that $(\!|\mathscr{q}_1|\!) \cong (\!|\mathscr{q}_2|\!)$ – so it makes sense to restrict our attention to positions reached by characteristic expansions.

How different can be characteristic $\mathscr{q}_1 \in \exp(\mathscr{p}_1)$ and $\mathscr{q}_2 \in \exp(\mathscr{p}_2)$ s.t. $(\!|\mathscr{q}_1|\!) \cong (\!|\mathscr{q}_2|\!)$? A first guess is *isomorphic*, but that is off the mark; $\mathscr{q}_1$ and $\mathscr{q}_2$ have some degree of liberty in swapping forks around (as in Figure 16): they have the "same branches, but with possibly different multiplicity". A significant part of our endeavour has been to construct a relation between augmentations allowing such changes in multiplicity, while ensuring $\mathscr{p}_1 \cong \mathscr{p}_2$.

4.2 Bisimulations Across an Isomorphism

More than simply comparing augmentations, given $\mathscr{q}, \mathscr{p} \in \mathsf{Aug}(A)$, $a \in \lvert\mathscr{q}\rvert$, $b \in \lvert\mathscr{p}\rvert$, we shall need a a predicate $a \sim b$ expressing that a and b have the same causal follow-up, up to the multiplicity of duplications. In particular, a and b must have "the same pointer", but at first that makes no sense since a and b live in different ambient sets of events. So we also fix an isomorphism $\varphi : (\!|\mathscr{q}|\!) \cong (\!|\mathscr{p}|\!)$ providing the translation, and aim to define $a \sim^{\varphi} b$ parametrized by φ. We give some examples in Figure 16, where φ is any of the two possible isomorphisms, assuming $\mathbf{q}_1^-$ and $\mathbf{q}_2^-$ correspond to different moves of the arena.

This is defined via a bisimulation game: for instance, establishing that the roots are in relation requires us to first match the blue nodes. But as the bisimulation unfolds, requiring all pointers to match up to φ is too strong: the pointers of red moves do *not* match – but seen from $\mathbf{q}^+$ this is fine as the justifiers for the red moves are encountered at the same step of the bisimulation game from $\mathbf{q}^+$. So our actual predicate has form $a \sim^{\varphi}_{\Gamma} b$ for Γ a *context*, stating a correspondence between negative moves established in the bisimulation game so far:

▶ **Definition 27.** *A* ***context*** *between* $q, p \in \mathsf{Aug}(A)$ *is* $\Gamma : \mathrm{dom}(\Gamma) \cong \mathsf{cod}(\Gamma)$ *a bijection s.t.* $\mathrm{dom}(\Gamma) \subseteq |q|$, $\mathsf{cod}(\Gamma) \subseteq |p|$, $\lambda_q(\mathrm{dom}(\Gamma)) \subseteq \{-\}$, *and* $\forall a^- \in \mathrm{dom}(\Gamma)$, $\partial_q(a) = \partial_p(\Gamma(a))$.

We may now formulate a first notion of bisimulation across augmentations.

▶ **Definition 28.** *Consider* $q, p \in \mathsf{Aug}(A)$ *and an isomorphism* $\varphi : (\!|q|\!) \cong (\!|p|\!)$.
For $a \in |q|$, $b \in |p|$ *and* Γ *a context, we define a predicate* $a \sim^\varphi_\Gamma b$ *which holds if, firstly,*

(a) $\partial_q(a) = \partial_p(b)$ *and* $\Gamma \vdash (a, b)$
(b) *if* $\mathsf{just}(a^+) \in \mathrm{dom}(\Gamma)$, *then* $\mathsf{just}(b) \in \mathsf{cod}(\Gamma)$ *and* $\Gamma(\mathsf{just}(a)) = \mathsf{just}(b)$,
(c) *if* $\mathsf{just}(a^+) \notin \mathrm{dom}(\Gamma)$, *then* $\mathsf{just}(b) \notin \mathsf{cod}(\Gamma)$ *and* $\varphi(\mathsf{just}(a)) = \mathsf{just}(b)$,

where $\Gamma \vdash (a, b)$ *means that for all* $a' \in \mathrm{dom}(\Gamma)$, $\neg(a' >_q a)$ *and for all* $b' \in \mathsf{cod}(\Gamma)$, $\neg(b' >_p b)$*; and inductively, the following two bisimulation conditions hold:*

(1) *if* $a^+ \rightarrowtriangle_q a'$, *then there is* $b^+ \rightarrowtriangle_p b'$ *with* $a' \sim^\varphi_{\Gamma \cup \{(a',b')\}} b'$, *and symmetrically,*
(2) *if* $a^- \rightarrowtriangle_q a'$, *then there is* $b^- \rightarrowtriangle_p b'$ *with* $a' \sim^\varphi_\Gamma b'$, *and symmetrically.*

As $\Gamma \vdash (a, b)$ implies $a' \notin \mathrm{dom}(\Gamma)$ and $b' \notin \mathsf{cod}(\Gamma)$, $\Gamma \cup \{(a', b')\}$ remains a bijection.
Of particular interest is the case $a \sim^\varphi_\emptyset b$ over an empty context, written simply $a \sim^\varphi b$. From this, we deduce a relation between augmentations: we write $q \sim^\varphi p$ iff $\mathsf{init}(q) \sim^\varphi \mathsf{init}(p)$, for $q, p \in \mathsf{Aug}(A)$ and $\varphi : (\!|q|\!) \cong (\!|p|\!)$. Resuming the discussion at the end of Section 4.1: bisimulations allow us to express that two characteristic expansions with isomorphic configurations are "the same". More precisely, in due course we will be able to prove:

▶ **Proposition 29.** *Consider* $p_1, p_2 \in \mathsf{Aug}(A)$ *causal strategies,* $q_1 \in \exp(p_1)$ *and* $q_2 \in \exp(p_2)$ *characteristic expansions with an isomorphism* $\varphi : (\!|q_1|\!) \cong (\!|q_2|\!)$. *Then,* $q_1 \sim^\varphi q_2$.

The proof is the core of our injectivity argument, which we will cover in Section 4.5. For now, we focus on how to conclude from $q_1 \sim^\varphi q_2$ that we have $p_1 \cong p_2$.

4.3 Compositional Properties of Bisimulations

To achieve that, we exploit compositional properties of bisimulations. More precisely, we show that $q_i \in \exp(p_i)$ induces a bisimulation $q_i \sim p_i$, and find a way to compose

$$p_1 \sim q_1 \quad \sim^\varphi \quad q_2 \sim p_2 \tag{1}$$

to deduce $p_1 \sim p_2$ in a sense yet to be defined, and $p_1 \cong p_2$ will follow. We start with:

▶ **Lemma 30.** *Consider augmentations* $q, p, r \in \mathsf{Aug}(A)$, *isomorphisms* $\varphi : (\!|q|\!) \cong (\!|p|\!)$, $\psi : (\!|p|\!) \cong (\!|r|\!)$, *events* $a \in |q|$, $b \in |p|$, $c \in |r|$, *and contexts* Γ, Δ. *Then:*

reflexivity: $a \sim^{\mathrm{id}} a$,
transitivity: *if* $a \sim^\varphi_\Gamma b$ *and* $b \sim^\psi_\Delta c$ *with* $\mathsf{cod}(\Gamma) = \mathrm{dom}(\Delta)$, *then* $a \sim^{\psi \circ \varphi}_{\Delta \circ \Gamma} c$,
symmetry: *if* $a \sim^\varphi_\Gamma b$ *then* $b \sim^{\varphi^{-1}}_{\Gamma^{-1}} a$.

But in order to treat $q_i \in \exp(p_i)$ as a bisimulation between q_i and p_i, Definition 28 does not do the trick: we cannot expect there to be an iso between $(\!|q_i|\!)$ and $(\!|p_i|\!)$ as q_i has by construction many more events. We therefore introduce a variant of Definition 28:

▶ **Definition 31.** *Consider $q, p \in \mathsf{Aug}(A)$. For $a \in |q|$, $b \in |p|$, Γ, we have $a \sim_\Gamma b$ if*

(a) $\partial_q(a) = \partial_p(b)$ *and* $\Gamma \vdash (a, b)$,
(b) $\mathsf{just}(a^+) \in \mathrm{dom}(\Gamma)$ *and* $\Gamma(\mathsf{just}(a)) = \mathsf{just}(b)$,
(1) *if $a^+ \twoheadrightarrow_q a'$, then $b^+ \twoheadrightarrow_p b'$ with $a' \sim_{\Gamma \cup \{(a',b')\}} b'$, and symmetrically,*
(2) *if $a^- \twoheadrightarrow_q a'$, then $b^- \twoheadrightarrow_p b'$ with $a' \sim_\Gamma b'$, and symmetrically.*

This helps us relate q and p when $(\!|q|\!)$ and $(\!|p|\!)$ are not isomorphic: we set $q \sim p$ iff $\mathsf{init}(q) \sim_{\{(\mathsf{init}(q), \mathsf{init}(p))\}} \mathsf{init}(p)$. A variation of Lemma 30 shows $\sim$ is an equivalence, and:

▶ **Proposition 32.** *Consider A an arena, $p \in \mathsf{Aug}(A)$ a causal strategy, and $q \in \mathsf{Aug}(A)$. Then, q is a $-$-obsessional expansion of p iff $q \sim p$.*

Proof. *If.* We simply construct $\varphi : q \to p$ for all $a \in |q|$ by induction on $\leq_q$. The image is provided by bisimulation, its uniqueness by *determinism* and *$-$-linearity*.

Only if. For $\varphi : q \to p$ and $a \in |q|$, write $[a]_q^- = \{a' \in |q| \mid a' \leq_q a \;\&\; \lambda(a') = -\}$; it is totally ordered by $\leq_q$ as q is *forestial*. From the conditions on φ, it is direct that it induces an order-iso $[a]_q^- \cong [\varphi(a)]_p^-$, *i.e.* a context $\Gamma\langle a \rangle : [a]_q^- \cong [\varphi(a)]_p^-$. Then, we check that $a \sim_{\Gamma\langle a \rangle} \varphi(a)$ for all $a \in |q|$, using that φ is $+$-obsessional. We then apply this to $\mathsf{init}(q)$. ◀

This vindicates Definition 31. But for (1), we must compose two kinds of bisimulations, following Definitions 28 and 31. Fortunately, whenever both definitions apply, they coincide:

▶ **Lemma 33.** *Consider $q, p \in \mathsf{Aug}(A)$, and $\varphi : (\!|q|\!) \cong (\!|p|\!)$. Then, $q \sim^\varphi p$ iff $q \sim p$.*

Proof. *If.* Straightforward from Definitions 28 and 31: case *(c)* is never used.

Only if. We actually prove that for all $a \in |q|$, $b \in |p|$, for all context Γ which is *complete* in the sense that $[a]_q^- \subseteq \mathrm{dom}(\Gamma)$ and $[b]_p^- \subseteq \mathsf{cod}(\Gamma)$, if $a \sim_\Gamma^\varphi b$ then $a \sim_\Gamma b$. The proof is immediate by induction: the clause *(c)* is never used from the hypothesis that Γ is complete. Finally, we apply this to the roots of q, p with context $\{(\mathsf{init}(q), \mathsf{init}(p))\}$. ◀

Altogether, we have:

▶ **Proposition 34.** *Consider $p_1, p_2 \in \mathsf{Aug}(A)$ causal strategies, $q_1 \in \exp(p_1)$, $q_2 \in \exp(p_2)$ characteristic expansions with an iso $\varphi : (\!|q_1|\!) \cong (\!|q_2|\!)$. If $q_1 \sim^\varphi q_2$, then $p_1 \cong p_2$.*

Proof. By Lemma 33, $q_1 \sim q_2$. As characteristic expansions, q_1 and q_2 are $-$-obsessional, so by Proposition 32, $q_1 \sim p_1$ and $q_2 \sim p_2$. So $p_1 \sim q_1 \sim q_2 \sim p_2$ but $\sim$ is an equivalence, so $p_1 \sim p_2$. By Proposition 32, we have $\varphi : p_1 \to p_2$ and $\psi : p_2 \to p_1$ composing to $\psi \circ \varphi : p_1 \to p_1$. But by *$-$-linear* and *determinism* there is only one morphism from p_1 to itself, the identity, so $\psi \circ \varphi = \mathrm{id}$. Likewise $\varphi \circ \psi = \mathrm{id}$, hence $\varphi : p_1 \cong p_2$ as required. ◀

4.4 Clones

In Section 4.1, we introduced *characteristic expansions* which, via duplications with well-chosen cardinalities, constrain the causal structure. More precisely, if $q \in \exp(p)$ is characteristic, looking at a set of duplicated Player moves in $(\!|q|\!)$ of cardinality n as in Figure 17, decomposing $n = \sum_{i \in I} 2^i$, we can deduce that the causal predecessors of the $\mathbf{q}_j^+$'s are among the forks with cardinality 2^i for $i \in I$. But that is not enough: this does not tell us how to distribute the $\mathbf{q}_j^+$'s to the forks, and not all the choices will work: while the $\mathbf{q}_j^+$'s are copies, their respective causal follow-ups might differ. So the idea is simple: imagine that the causal follow-ups for the $\mathbf{q}_j^+$'s are already reconstructed. Then we may compare them using bisimulation, and replicate the same reasoning as above on bisimulation equivalence classes.

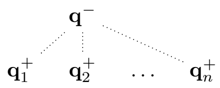


Figure 17 A set of copied Player moves.

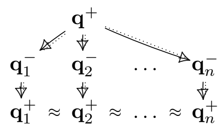


Figure 18 A set of clones switching pointers.

So we are left with the task of leveraging bisimulation to define an adequate equivalence relation on $|\mathscr{q}|$. This leads to the notion of *clones*, our last technical tool.

▶ **Definition 35.** *Consider $\mathscr{q}, \mathscr{p} \in \mathsf{Aug}(A)$, $\varphi : (\!|\mathscr{q}|\!) \cong (\!|\mathscr{p}|\!)$, and $a \in |\mathscr{q}|$, $b \in |\mathscr{p}|$.*

*We say that a and b are **clones** through φ, written $a \approx^\varphi b$, if there is a context Γ preserving pointers (*i.e. *for all $a' \in \mathrm{dom}(\Gamma)$, $\varphi(\mathsf{just}(a')) = \mathsf{just}(\Gamma(a'))$) such that $a \sim^\varphi_\Gamma b$.*

This allows a and b (and their follow-ups) to change their pointers through some unspecified Γ. Indeed, the picture painted by Figure 17 is limited: a fork might trigger Player moves with different pointers, as in Figure 18. As $a \approx^\varphi b$ quantifies existentially over contexts, compositional properties of clones are more challenging. Nevertheless, via a canonical form for contexts and leveraging Lemma 30, we show that $a \approx^{\mathrm{id}} a$, that $a \approx^\varphi b$ and $b \approx^\psi c$ imply $a \approx^{\psi\circ\varphi} c$, and that $a \approx^\varphi b$ implies $b \approx^{\varphi^{-1}} a$ whenever these typecheck – see Appendix A.2. Instantiating Definition 35 with $\mathscr{q} = \mathscr{p}$ and $\varphi = \mathrm{id}$, we get an equivalence relation $\approx$ on $|\mathscr{q}|$.

Moreover, we have the crucial property that forks generate clones (see Appendix A.2):

▶ **Lemma 36.** *Consider $\mathscr{q}$ a $-$-obsessional expansion of causal strategy $\mathscr{p}$ on arena A. Then, for all $a_1^-, a_2^- \in X \in \mathsf{Fork}(\mathscr{q})$, for all $a_1^- \rightarrowtriangle_{\mathscr{q}} b_1^+$ and $a_2^- \rightarrowtriangle_{\mathscr{q}} b_2^+$, $b_1 \approx b_2$.*

By Lemma 36, if a clone class includes a positive move, it also has all its cousins triggered by the same fork – so clone classes may be partitioned following forks:

▶ **Lemma 37.** *Let $\mathscr{q}$ be a characteristic expansion of causal strategy $\mathscr{p}$, and Y a clone class of positive events in $|\mathscr{q}|$, with $\sharp Y = \sum_{i\in I} 2^i$ for $I \subseteq \mathbb{N}$ finite. Then, for all $i \in \mathbb{N}$, $i \in I$ iff there is $X_i \in \mathsf{Fork}(\mathscr{q})$ with $\sharp X_i = 2^i$ and $a^- \in X_i$, $b^+ \in Y$ such that $a^- \rightarrowtriangle_{\mathscr{q}} b^+$.*

Proof. For any $i \in \mathbb{N}$, we write X_i the fork of $\mathscr{q}$ of cardinality 2^i, if it exists.

Consider the set $J := \{j \in \mathbb{N} \mid X_j \text{ exists}, \exists a \in X_j, \exists b \in Y, a \rightarrowtriangle_{\mathscr{q}} b\}$. Any $b \in Y$ is positive and so the unique (by determinism) successor of some negative event a. Moreover a appears in a fork X and by Lemma 36, all events of X are predecessors of events of Y. Hence, we have $Y = \biguplus_{j\in J} \mathsf{succ}(X_j)$, where the union is disjoint since $\mathscr{q}$ is forest-shaped. Therefore,

$$\sharp Y \;=\; \sum_{j\in J} \sharp\mathsf{succ}(X_j) \;=\; \sum_{j\in J} \sharp X_j \;=\; \sum_{j\in J} 2^j,$$

where the second equality is obtained by determinism. By uniqueness of the binary decomposition, $J = I$, which proves the lemma by definition of J. ◀

4.5 Positional Injectivity

We are finally in a position to prove the core of the injectivity argument.

▶ **Lemma 38** (Key lemma). *Consider $\mathscr{p}_1, \mathscr{p}_2 \in \mathsf{Aug}(A)$ causal strategies, $\mathscr{q}_1 \in \exp(\mathscr{p}_1)$ and $\mathscr{q}_2 \in \exp(\mathscr{p}_2)$ characteristic expansions, and $\varphi : (\!|\mathscr{q}_1|\!) \cong (\!|\mathscr{q}_2|\!)$. Then, $\forall a^+ \in |\mathscr{q}_1|, a \approx^\varphi \varphi(a)$.*

Proof. The **co-depth** of $a \in |q_i|$ is the maximal length k of $a = a_1 \rightarrow_{q_i} \dots \rightarrow_{q_i} a_k$ a causal chain in q_i. We show by induction on k the two symmetric properties:

(a) for all $a^+ \in |q_1|$ of co-depth $\leq k$, we have $a \approx^{\varphi} \varphi(a)$,

(b) for all $a^+ \in |q_2|$ of co-depth $\leq k$, we have $a \approx^{\varphi^{-1}} \varphi^{-1}(a)$.

Take $a^+ \in |q_1|$ of co-depth k. If a is maximal in q_1, so is $\varphi(a)$ in q_2 and $a \approx \varphi(a)$. Else, the successors of a partition as $G_1, \dots, G_n \subseteq \mathsf{Fork}(q_1)$, where $G_i = \{b^-_{i,1}, \dots, b^-_{i,2^{p_i}}\}$; likewise the successors of $\varphi(a)$ in q_2 are the forks $\varphi(G_i)$. For all $1 \leq i \leq n$ and $1 \leq j \leq 2^{p_i}$, we claim:

$$\text{for all } b_{i,j} \rightarrow_{q_1} c_{i,j}, \text{ there is } \varphi(b_{i,j}) \rightarrow_{q_2} d_{i,j} \text{ satisfying } c_{i,j} \approx^{\varphi} d_{i,j}. \tag{2}$$

Write $X = [c_{i,j}]_{\approx}$ the clone class of $c_{i,j}$ in q_1. It is easy to prove that the clone relation preserves co-depth, so it follows from the induction hypothesis and Lemma 46 that $\varphi(X)$ is a clone class in q_2. By Lemma 37, $\sharp X$ has 2^{p_i} in its binary decomposition – and as φ is a bijection, so does $\sharp(\varphi(X))$. So by Lemma 37, there is $\varphi(b_{i,j}) \in \varphi(G_i)$ and $d_{i,j} \in \varphi(X)$ such that $\varphi(b_{i,j}) \rightarrow_{q_2} d_{i,j}$. Since $\varphi(c_{i,j}), d_{i,j} \in \varphi(X)$ they are clones, so using $c_{i,j} \approx^{\varphi} \varphi(c_{i,j})$ by induction hypothesis, $c_{i,j} \approx^{\varphi} d_{i,j}$. Likewise, the mirror property of (2) also holds.

Deducing $a \approx^{\varphi} \varphi(a)$ requires some care: cloning is defined via a context, and the $c_{i,j} \approx^{\varphi} \varphi(c_{i,j})$ might not share the same. However, the contexts can be put into canonical forms that are shown to agree – Lemma 48 allows us to prove $a \approx^{\varphi} \varphi(a)$ from (2) and its mirror property. Finally, *(b)* is proved symmetrically. ◀

Now, consider $p_1, p_2, q_1, q_2, \varphi$ as in Proposition 29. If the q_i's are empty or singleton trees, there is nothing to prove. Otherwise q_i starts with $a_i^- \rightarrow_{q_i} b_i^+$ with a_i^- initial. But then $[b_i^+]_{\approx}$ is the only singleton clone class in q_i. As φ preserves clone classes, $\varphi(b_1^+) = b_2^+$. By Lemma 38, $b_1 \approx^{\varphi} b_2$. Thus $b_1 \sim^{\varphi} b_2$, so $a_1 \sim^{\varphi} a_2$ and $q_1 \sim^{\varphi} q_2$. This concludes the proof of Proposition 29. Putting everything together, we obtain:

▶ **Theorem 39.** *For $p_1, p_2 \in \mathsf{Aug}(A)$ causal strategies s.t. $⦇p_1⦈ = ⦇p_2⦈$, then $p_1 \cong p_2$.*

Proof. Consider $q_1 \in \exp(p_1)$ a characteristic expansion. By hypothesis, there must be $q_2 \in \exp(p_2)$ and $\varphi : ⦇q_1⦈ \cong ⦇q_2⦈$; necessarily q_2 is also a characteristic expansion of p_2. By Proposition 29, we have $q_1 \sim^{\varphi} q_2$. By Proposition 34, we have $p_1 \cong p_2$. ◀

Finally, Theorem 17 follows from Theorem 39, Proposition 24 and Lemma 22.

Theorem 17 only concerns *total* finite innocent strategies. In contrast, Theorem 39 requires no totality assumption: totality comes in not in the injectivity argument, but in Proposition 24 linking standard and causal strategies. Without totality, expansions of $\hat{\sigma}$ might not have as many Opponent as Player moves, and so may not be linearizable via alternating plays. Intuitively, in alternating plays Opponent may only explore converging parts of the strategy, whereas in the causal setting Opponent is free to explore simultaneously many branches, including divergences. Positional injectivity for *partial* finite innocent strategies may be studied causally by restricting to $+$-*covered* expansions, *i.e.* with only Player maximal events. But then we must also abandon $-$-*obsessionality* as Opponent moves leading to divergence will not be played, breaking our proof (Lemma 36 fails) in a way for which we see no fix.

5 Beyond Total Finite Strategies

Finally, we show some subtleties and partial results on generalizations of Theorem 17.

First, positional injectivity fails in general. Consider the infinitary terms $f : o \to o \to o \vdash T_1, T_2, L, R : o$ recursively defined as $T_1 = f\, T_2\, R, T_2 = f\, L\, T_1, L = f\, L\, \bot$ and $R = f\, \bot\, R$ in an infinitary simply-typed λ-calculus with divergence $\bot$. The corresponding strategies differ: their causal representations appear in Figures 19 and 20, infinite trees represented via loops.

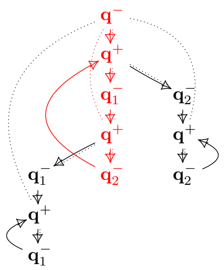


Figure 19 $[\![\lambda f^{o\to o\to o}.\, T_1]\!]$.

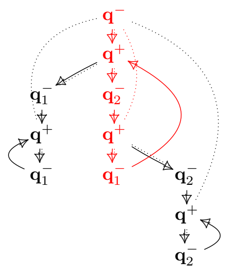


Figure 20 $[\![\lambda f^{o\to o\to o}.\, T_2]\!]$.

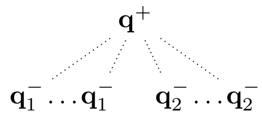


Figure 21 Bricks.

We consider positions reached by plays – or equivalently, by $+$-covered expansions of Figures 19 and 20. In fact, both strategies admit all *balanced* positions on $[\![(o \to o \to o) \to o]\!]$, *i.e.* with as many Opponent as Player moves. Ignoring the initial $\mathbf{q}^-$, a position is a multiset of **bricks** as in Figure 21, with $i \in \mathbb{N}$ occurrences of $\mathbf{q}_1^-$ and $j \in \mathbb{N}$ of $\mathbf{q}_2^-$. A brick with $i = j = 0$ is a **leaf**. The position is balanced if it has as many Opponent as Player moves.

Now, any position can be realized in $[\![\lambda f^{o\to o\to o}.\, T_i]\!]$ by first placing bricks with occurrences of both $\mathbf{q}_1^-$ and $\mathbf{q}_2^-$ greedily alongside the *spine*, shown in red in Figures 19 and 20. At each step, we continue from only one of the copies opened, leaving others dangling. If this gets stuck, apart from leaves we are left with only $\mathbf{q}_1^-$'s, or, only $\mathbf{q}_2^-$'s, but there is always a matching non-spine infinite branch available. Finally, leaves can always be placed as their number matches that of trailing negative moves by the balanced hypothesis.

We have $(\![[\![\lambda f^{o\to o\to o}.\, T_1]\!]]\!) = (\![[\![\lambda f^{o\to o\to o}.\, T_2]\!]]\!)$ as both strategies can realize *all* balanced positions on the arena $[\![o \to o \to o]\!]$, and *exactly* those: positional injectivity fails.

Positionality for *finite* innocent strategies remains open. We could only prove:

▶ **Theorem 40.** *Let $\sigma_1, \sigma_2 : A$ be finite innocent strategies with $(\![\sigma_1]\!) = (\![\sigma_2]\!)$. Then, σ_1 and σ_2 have the same P-views of maximal length.*

For the proof (see Appendix B), we assume σ_1 has a P-view s of maximal length n. We perform an expansion of s where each Opponent branching at co-depth $2d+1$ has arity $d+1$. By a combinatorial argument on trees, the only way to reassemble its nodes exhaustively in a tree with depth bounded by d is to rebuild exactly the same tree. Hence the tree is also in $\exp(\hat{\sigma_2})$, and $s \in \sigma_2$. This steers us into conjecturing that positional injectivity holds for partial finite innocent strategies, but our proof attempts have remained inconclusive.

6 Conclusion

Though innocent strategies in the Hyland-Ong sense are not positional, total finite innocent strategies satisfy *positional injectivity* – however, the property fails in general.

Beyond its foundational value, we believe this result may be helpful in the game semantics toolbox. Game semantics can be fiddly; in particular, proofs that two terms yield the same strategy are challenging to write in a concise yet rigorous manner. This owes a lot to the complexity of *composition*: proving that a play s is in $[\![M\,N]\!]$ involves constructing an "interaction witness" obtained from plays in $[\![M]\!]$ and $[\![N]\!]$ plus an adequate "zipping" of the two. Manipulations of plays with pointers are tricky and error-prone, and the link between plays and terms is obfuscated by the multi-layered interpretation.

In contrast, Theorem 17 lets us prove innocent strategies equal by comparing their positions. Now, constructing a position of $[\![MN]\!]$ simply involves exhibiting matching positions for $[\![M]\!]$ and $[\![N]\!]$. Side-stepping the interpretation, this can be presented as typing terms with positions or configurations – combining Section 2.5 and the link between relational semantics and non-idempotent intersection type systems [11]. For instance, in this way, finite definability, a basic result seldom presented in full formal details, boils down to typing the defined term with the same positions as the original strategy.

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 Samson Abramsky and Paul-André Melliès. Concurrent games and full completeness. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 431–442. IEEE Computer Society, 1999. doi:10.1109/LICS.1999.782638.

3 Pierre Boudes. Thick subtrees, games and experiments. In Pierre-Louis Curien, editor, *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*, volume 5608 of *Lecture Notes in Computer Science*, pages 65–79. Springer, 2009. doi:10.1007/978-3-642-02273-9_7.

4 Simon Castellan, Pierre Clairambault, Hugo Paquet, and Glynn Winskel. The concurrent game semantics of probabilistic PCF. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 215–224. ACM, 2018. doi:10.1145/3209108.3209187.

5 Simon Castellan, Pierre Clairambault, Silvain Rideau, and Glynn Winskel. Games and strategies as event structures. *Logical Methods in Computer Science*, 13(3), 2017. doi:10.23638/LMCS-13(3:35)2017.

6 Simon Castellan, Pierre Clairambault, and Glynn Winskel. Symmetry in concurrent games. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 28:1–28:10. ACM, 2014. doi:10.1145/2603088.2603141.

7 Simon Castellan, Pierre Clairambault, and Glynn Winskel. The parallel intensionally fully abstract games model of PCF. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, pages 232–243. IEEE Computer Society, 2015. doi:10.1109/LICS.2015.31.

8 Pierre Clairambault. A tale of additives and concurrency in game semantics. To appear, 2021.

9 Pierre Clairambault and Marc de Visme. Full abstraction for the quantum lambda-calculus. *Proc. ACM Program. Lang.*, 4(POPL):63:1–63:28, 2020. doi:10.1145/3371131.

10 Daniel de Carvalho. The relational model is injective for multiplicative exponential linear logic. In *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 - September 1, 2016, Marseille, France*, pages 41:1–41:19, 2016. doi:10.4230/LIPIcs.CSL.2016.41.

11 Daniel de Carvalho. Execution time of λ-terms via denotational semantics and intersection types. *Math. Struct. Comput. Sci.*, 28(7):1169–1203, 2018. doi:10.1017/S0960129516000396.

12 Thomas Ehrhard. The Scott model of linear logic is the extensional collapse of its relational model. *Theor. Comput. Sci.*, 424:20–45, 2012. doi:10.1016/j.tcs.2011.11.027.

13 Russell Harmer, Martin Hyland, and Paul-André Melliès. Categorical combinatorics for innocent strategies. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 379–388. IEEE Computer Society, 2007. doi:10.1109/LICS.2007.14.

14 Tom Hirschowitz and Damien Pous. Innocent strategies as presheaves and interactive equivalences for CCS. *Sci. Ann. Comput. Sci.*, 22(1):147–199, 2012. doi:10.7561/SACS.2012.1.147.

15 J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000. doi:10.1006/inco.2000.2917.
16 Paul-André Melliès. Asynchronous games 2: The true concurrency of innocence. *Theor. Comput. Sci.*, 358(2-3):200–228, 2006. doi:10.1016/j.tcs.2006.01.016.
17 Takeshi Tsukada and C.-H. Luke Ong. Innocent strategies are sheaves over plays – deterministic, non-deterministic and probabilistic innocence. *CoRR*, abs/1409.2764, 2014. arXiv:1409.2764.
18 Takeshi Tsukada and C.-H. Luke Ong. Nondeterminism in game semantics via sheaves. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, pages 220–231. IEEE Computer Society, 2015. doi:10.1109/LICS.2015.30.
19 Takeshi Tsukada and C.-H. Luke Ong. Plays as resource terms via non-idempotent intersection types. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 237–246. ACM, 2016. doi:10.1145/2933575.2934553.

A Positional Injectivity: Proofs from Section 4

In the sequel, A is a fixed arena. For any augmentation $\mathscr{q} \in \mathsf{Aug}(A)$ and event $a \in |\mathscr{q}|$, we define $\mathsf{succ}(a) := \{b \mid a \rightarrowtriangle_{\mathscr{q}} b\}$ the set of immediate successors of a in $\leq_{\mathscr{q}}$. We also define $\uparrow a$ the set of descendants of a, i.e. $\uparrow a := \{a' \mid a \leq_{\mathscr{q}} a'\}$.

A.1 Compositional Properties of Bisimulations (Section 4.3)

▶ **Lemma 41.** *Consider $\mathscr{q}, \mathscr{p} \in \mathsf{Aug}(A)$ where $\mathscr{p}$ is a causal strategy and $\mathscr{q} \in \exp(\mathscr{p})$ $-$-obsessional with $\varphi : \mathscr{q} \to \mathscr{p}$. Then $a \sim_{\Gamma\langle a\rangle} \varphi(a)$.*

Proof. Direct by induction on a. ◀

▶ **Lemma 42.** *Consider $\mathscr{q}, \mathscr{p} \in \mathsf{Aug}(A)$ with $\varphi : \mathscr{q} \cong \mathscr{p}$. Consider $a \sim^{\varphi}_{\Gamma} b$ for some Γ.*

Then for any $a' \in \uparrow a$, there exists $b' \in \uparrow b$ such that $a' \sim^{\varphi}_{\Gamma\cup\Delta} b'$ with Δ a context. Moreover, if $a \sim^{\varphi}_{\Gamma'} b$ for a context Γ', we also have $a' \sim^{\varphi}_{\Gamma'\cup\Delta} b'$.

Proof. The first part is immediate by Definition 28. Moreover, we can remark that Δ is exactly the negative moves between a and a', paired with the negative moves between b and b' (straightforward by induction). Finally, we prove the last part by induction on the *co-depth* of a (the maximal length k of $a = a_1 \rightarrowtriangle_{\mathscr{q}} a_2 \rightarrowtriangle_{\mathscr{q}} \dots \rightarrowtriangle_{\mathscr{q}} a_k$ a causal chain in $\mathscr{q}$). ◀

▶ **Definition 43.** *Consider $\mathscr{q}, \mathscr{p} \in \mathsf{Aug}(A)$, $\varphi : (\!|\mathscr{q}|\!) \cong (\!|\mathscr{p}|\!)$, $a \in |\mathscr{q}|$, $b \in |\mathscr{p}|$ with $a \sim^{\varphi}_{\Gamma} b$ for some context Γ. We define $\Gamma_{a,b}$ the **minimal context** for $a \sim^{\varphi}_{\Gamma} b$ as the restriction of Γ s.t.*

$$c \in \mathrm{dom}(\Gamma_{a,b}) \quad \Leftrightarrow \quad \begin{cases} \exists a' \in \uparrow a, \mathsf{just}(a') = c & \text{(a)} \\ \Gamma(c) \neq \varphi(c) & \text{(b)} \end{cases}$$

for all $c \in |\mathscr{q}|$,and symmetrically the mirror condition applies to any $d \in |\mathscr{p}|$.

▶ **Lemma 44.** *Consider $\mathscr{q}, \mathscr{p} \in \mathsf{Aug}(A)$ with $\varphi : (\!|\mathscr{q}|\!) \cong (\!|\mathscr{p}|\!)$. Consider $a \in |\mathscr{q}|$, $b \in |\mathscr{p}|$ and Γ, Γ' two contexts such that $a \sim^{\varphi}_{\Gamma} b$ and $a \sim^{\varphi}_{\Gamma'} b$.*

Then $\Gamma_{a,b} = \Gamma'_{a,b}$. Moreover, $\Gamma_{a,b}$ is the minimal (for inclusion) context s.t. $a \sim^{\varphi}_{\Gamma_{a,b}} b$.

Proof. The equality comes from Lemma 42 and the definition of $\Gamma_{a,b}$ and $\Gamma'_{a,b}$. By induction, $a \sim^{\varphi}_{\Gamma_{a,b}} b$, since we can safely remove from $\mathrm{dom}(\Gamma)$ all c that are never "used", *i.e.* such that there exists no $a' \in \uparrow a$ having c as pointer; and all c such that $\Gamma(c) = \varphi(c)$, because then we can use condition (c) of Definition 28 instead of condition (b). Finally, for any context Γ'' such that $a \sim^{\varphi}_{\Gamma''} b$, we have $\Gamma_{a,b} = \Gamma''_{a,b} \subseteq \Gamma''$, so $\Gamma_{a,b}$ is minimal for inclusion. ◀

This lemma allows us to write *the minimal context for a, b* without mentioning Γ.

A.2 Clones (Section 4.4)

For any a, b events of an augmentation $\mathscr{q}$, $a \approx b$ means $a \approx^{\mathrm{id}} b$.

▶ **Lemma 45.** *Consider $\mathscr{q}, \mathscr{p}, \mathscr{r} \in \mathsf{Aug}(A)$ with $\varphi : (\!|\mathscr{q}|\!) \cong (\!|\mathscr{p}|\!)$ and $\psi : (\!|\mathscr{p}|\!) \cong (\!|\mathscr{r}|\!)$. For any $a \in |\mathscr{q}|$, $b \in |\mathscr{p}|$, and $c \in |\mathscr{r}|$ such that $a \approx^{\varphi} b$ and $b \approx^{\psi} c$, we have $a \approx^{\psi\circ\varphi} c$.*

Proof. Consider Γ_1 and Γ_2 the minimal contexts such that $a \sim^{\varphi}_{\Gamma_1} b$ and $b \sim^{\psi}_{\Gamma_2} c$. If $\mathsf{cod}(\Gamma_1) = \mathrm{dom}(\Gamma_2)$, the result is immediate by Lemma 30. Otherwise, we complete them to:

$$\begin{aligned}
\Gamma'_1 &:= \Gamma_1 \cup \{(\varphi^{-1}(e'), e') \mid e' \in \mathrm{dom}(\Gamma_2), e' \notin \mathsf{cod}(\Gamma_1)\}\,, \\
\Gamma'_2 &:= \Gamma_2 \cup \{(e, \psi(e)) \mid e \in \mathsf{cod}(\Gamma_1), e \notin \mathrm{dom}(\Gamma_2)\}\,,
\end{aligned}$$

two pointer-preserving contexts. Then, we can prove that $a \sim^{\psi\circ\varphi}_{\Gamma'_2\circ\Gamma'_1} c$, so $a \approx^{\psi\circ\varphi} c$. ◀

This covers transitivity for the clone relation, with other equivalence properties direct:

▶ **Lemma 46.** *Consider $\mathscr{q}, \mathscr{p}, \mathscr{r} \in \mathsf{Aug}(A)$ augmentations, with $\varphi : (\!|\mathscr{q}|\!) \cong (\!|\mathscr{p}|\!)$ and $\psi : (\!|\mathscr{p}|\!) \cong (\!|\mathscr{r}|\!)$ two isomorphisms, and events $a \in |\mathscr{q}|$, $b \in |\mathscr{p}|$, $c \in |\mathscr{r}|$:*

reflexivity:	*$a \approx^{\mathrm{id}} a$,*
transitivity:	*if $a \approx^{\varphi} b$ and $b \approx^{\psi} c$, then $a \approx^{\psi\circ\varphi} c$,*
symmetry:	*if $a \approx^{\varphi} b$ then $b \approx^{\varphi^{-1}} a$.*

▶ **Lemma 36.** *Consider $\mathscr{q}$ a $-$-obsessional expansion of causal strategy $\mathscr{p}$ on arena A. Then, for all $a_1^-, a_2^- \in X \in \mathsf{Fork}(\mathscr{q})$, for all $a_1^- \rightarrow_{\mathscr{q}} b_1^+$ and $a_2^- \rightarrow_{\mathscr{q}} b_2^+$, $b_1 \approx b_2$.*

Proof. First, we prove that b_1 and b_2 are bisimilar. Since $\mathscr{q} \in \exp(\mathscr{p})$, there is $\varphi : \mathscr{q} \to \mathscr{p}$. By Lemma 41, $b_1 \sim_{\Gamma\langle b_1\rangle} \varphi(b_1)$ and $b_2 \sim_{\Gamma\langle b_2\rangle} \varphi(b_2)$. By $-$-linearity of $\mathscr{p}$, $\varphi(a_1) = \varphi(a_2)$, which implies $\varphi(b_1) = \varphi(b_2)$ by determinism. So $\mathsf{cod}(\Gamma\langle b_1\rangle) = \mathsf{cod}(\Gamma\langle b_2\rangle)$, and by Lemma 30, $b_1 \sim_{\Gamma\langle b_2\rangle^{-1}\circ\Gamma\langle b_1\rangle} b_2$. Finally, we verify that $\Gamma\langle b_2\rangle^{-1} \circ \Gamma\langle b_1\rangle$ preserves pointers. ◀

A.3 Positional Injectivity (Section 4.5)

In this section, we prove additional lemmas needed in the proof of Lemma 38.

▶ **Lemma 47.** *Consider $\mathscr{q} \in \mathsf{Aug}(A)$ and $a, b \in |\mathscr{q}|$ such that $a \approx b$. Then the minimal context for a and b is either empty or $\Gamma : \{c\} \cong \{d\}$ for some c, d.*

Proof. Assume, seeking a contradiction, that the minimal context Γ has at least two distinct elements $c_1, c_2 \in \mathrm{dom}(\Gamma)$. First, we can remark that since $a \approx b$, there exists Γ' a pointers-preserving context such that $a \sim_{\Gamma'} b$, and since $\Gamma \subseteq \Gamma'$, Γ also preserves pointers.

By condition *(a)* of Definition 43, $c_1 \leq_{\mathscr{q}} a$ and $c_2 \leq_{\mathscr{q}} a$. Therefore, $c_1 \leq_{\mathscr{q}} c_2$ or $c_2 \leq_{\mathscr{q}} c_1$ – assume *w.l.o.g.* it is the former. By courtesy, $\mathsf{just}(c_1) \leq_{\mathscr{q}} \mathsf{just}(c_2)$ as well. For the same reason, $\Gamma(c_1) \leq_{\mathscr{q}} \Gamma(c_2)$ or $\Gamma(c_2) \leq_{\mathscr{q}} \Gamma(c_1)$. If it is the latter, this entails that $\mathsf{just}(\Gamma(c_2)) \leq_{\mathscr{q}} \mathsf{just}(\Gamma(c_1))$ by courtesy; *i.e.*, since Γ preserves pointers, $\mathsf{just}(c_2) \leq_{\mathscr{q}} \mathsf{just}(c_1)$. So $\mathsf{just}(c_1) = \mathsf{just}(c_2)$, *i.e.* $\mathsf{pred}(c_1) = \mathsf{pred}(c_2)$ by courtesy. Because $c_1 \leq_{\mathscr{q}} c_2$ we have $c_1 = c_2$, contradiction.

So, $\Gamma(c_1) \leq_{\mathscr{q}} \Gamma(c_2)$, and $\Gamma(c_1) \neq \Gamma(c_2)$ by hypothesis. By courtesy, $\Gamma(c_1) \leq_{\mathscr{q}} \mathsf{just}(\Gamma(c_2))$. Likewise, $c_1 \leq_{\mathscr{q}} c_2$ entails $c_1 \leq_{\mathscr{q}} \mathsf{just}(c_2)$. Moreover, Γ preserves pointers, so $\mathsf{just}(c_2) = \mathsf{just}(\Gamma(c_2))$. Hence, we have both $\Gamma(c_1) \leq_{\mathscr{q}} \mathsf{just}(c_2)$ and $c_1 \leq_{\mathscr{q}} \mathsf{just}(c_2)$, so c_1 and $\Gamma(c_1)$ are comparable for $\leq_{\mathscr{q}}$ since $\mathscr{q}$ is a forest. But they are negative, so they have the same antecedent by courtesy. This implies $c_1 = \Gamma(c_1)$, contradicting *(b)* of Definition 43. ◀

Figure 22 Justifiers in $\mathcal{q}$ and $\mathcal{p}$.

▶ **Lemma 48.** *Consider* $\mathcal{q}, \mathcal{p} \in \mathsf{Aug}(A)$, $\varphi : (\!|\mathcal{q}|\!) \cong (\!|\mathcal{p}|\!)$*. Consider also* $a^+ \in |\mathcal{q}|$ *s.t.* $\mathsf{succ}(a) = \bigcup_{i \in I} G_i$*, where* $I \subseteq \mathbb{N}$ *and for* $i \in I$*,* $G_i = \{b_{i,1}, \dots, b_{i,2^i}\} \in \mathsf{Fork}(\mathcal{q})$ *with* $\sharp G_i = 2^i$*.*
Then we have $a \approx^\varphi \varphi(a)$*, provided the two conditions hold:*

$$\text{if } b_{i,j} \rightarrow_{\mathcal{q}} c_{i,j}, \text{ then } \varphi(b_{i,j}) \rightarrow_{\mathcal{p}} d_{i,j} \text{ and } c_{i,j} \approx^\varphi d_{i,j} \,, \tag{3}$$

$$\text{if } \varphi(b_{i,j}) \rightarrow_{\mathcal{p}} d_{i,j}, \text{ then } b_{i,j} \rightarrow_{\mathcal{q}} c_{i,j} \text{ and } c_{i,j} \approx^\varphi d_{i,j} \,. \tag{4}$$

Proof. For any $i \in I$, $1 \leq j \leq 2^i$, let $\Gamma_{i,j}$ be the minimal context for $b_{i,j}$ and $\varphi(b_{i,j})$. Such a context exists since either $b_{i,j}$ has no successors, and by (4) neither does $\varphi(b_{i,j})$, either $b_{i,j}$ has only one (by determinism) and $c_{i,j} \approx^\varphi d_{i,j}$ by (3). In both cases, $b_{i,j} \approx^\varphi \varphi(b_{i,j})$.

We wish to take the union of all $\Gamma_{i,j}$ as the context for a and $\varphi(a)$, but this is only possible if they are *compatible*. More precisely, we must ensure that for all $e \in \mathcal{q}, i, k \in I, 1 \leq j \leq 2^i$ and $1 \leq l \leq 2^k$, if there are $c'_{i,j} \in \uparrow b_{i,j}$ and $c'_{k,l} \in \uparrow b_{k,l}$ having both e as justifier, then their matching $d'_{i,j} \in \uparrow \varphi(b_{i,j})$ and $d'_{k,l} \in \uparrow \varphi(b_{k,l})$ also have the same justifier. This can only be a problem if e appears in $\mathrm{dom}(\Gamma_{i,j})$ or in $\mathrm{dom}(\Gamma_{k,l})$ as otherwise both justifiers are $\varphi(e)$.

For all i, j, $\Gamma_{i,j}$ has either one or zero element by Lemma 47. If all $\Gamma_{i,j}$ are empty, we can directly lift the clone relation to a. Otherwise, consider i, j s.t. $\Gamma_{i,j} : \{e_{i,j}\} \cong \{f_{i,j}\}$. From Definition 43, $e_{i,j} \in [b_{i,j}]^-_{\mathcal{q}}$ and $f_{i,j} \in [\varphi(b_{i,j})]^-_{\mathcal{p}}$. Actually we have $f_{i,j} \in [\varphi(a)]^-_{\mathcal{p}}$: indeed $f_{i,j} \neq \varphi(b_{i,j})$, since $e_{i,j}$ and $f_{i,j}$ have the same justifier through φ and the only $e \in [b_{i,j}]^-_{\mathcal{q}}$ s.t. $\varphi(\mathsf{just}(e)) = \mathsf{just}(\varphi(b_{i,j}))$ is $b_{i,j}$, which contradicts Definition 43.

Now, assume that for some k, l, there exists $c'_{k,l} \in \uparrow b_{k,l}$ s.t. $\mathsf{just}(c'_{k,l}) = e_{i,j}$. Since $b_{k,l} \approx^\varphi \varphi(b_{k,l})$, there is a matching $d'_{k,l} \in \uparrow \varphi(b_{k,l})$ s.t. $\varphi(\mathsf{just}(e_{i,j})) = \mathsf{just}(\mathsf{just}(d'_{k,l}))$. For $b_{i,j} \sim^\varphi_{\Gamma_{i,j}} \varphi(b_{i,j})$ and $b_{k,l} \sim^\varphi_{\Gamma_{k,l}} \varphi(b_{k,l})$ to be compatible, we need $\mathsf{just}(d'_{k,l}) = f_{i,j}$. But since $\Gamma_{i,j}$ preserves pointers, $\varphi(\mathsf{just}(e_{i,j})) = \mathsf{just}(f_{i,j})$. Putting both equalities together, we obtain $\mathsf{just}(\mathsf{just}(d'_{k,l})) = \mathsf{just}(f_{i,j})$, where $\mathsf{just}(d'_{k,l}) \in [d'_{k,l}]^-_{\mathcal{p}}$ and $f_{i,j} \in [\varphi(a)]^-_{\mathcal{p}}$. But $[\varphi(a)]^-_{\mathcal{p}} \subseteq [d'_{k,l}]^-_{\mathcal{p}}$, which is a fully ordered set for $\leq_{\mathcal{p}}$, so $\mathsf{just}(d'_{k,l})$ and $f_{i,j}$ are comparable. Moreover, they are negative, so by courtesy $\mathsf{just}(\mathsf{just}(d'_{k,l})) = \mathsf{just}(f_{i,j})$ iff $\mathsf{pred}(\mathsf{just}(d'_{k,l})) = \mathsf{pred}(f_{i,j})$, where pred is the predecessor for $\leq_{\mathcal{p}}$. Hence, $\mathsf{just}(d'_{k,l}) = f_{i,j}$ (see Figure 22, where $\rightarrow$ represents $\rightarrow_{\mathcal{q}}$, $\cdots$ represents $\rightarrow_{(\!|\mathcal{q}|\!)}$, and $\rightarrow$ represents $\leq_{\mathcal{q}}$ (and the same applies for $\mathcal{p}$)).

So all contexts $\Gamma_{i,j}$ are compatible. Writing $\Gamma = \cup_{i,j} \Gamma_{i,j}$ it follows that $b_{i,j} \sim^\varphi_\Gamma \varphi(b_{i,j})$ via a straightforward argument, which entails that $a \sim^\varphi_\Gamma \varphi(a)$ by two steps of the bisimulation game. This implies $a \approx^\varphi \varphi(a)$ since all $\Gamma_{i,j}$ preserve pointers. ◀

B Beyond Total Finite Strategies: Proofs from Section 5

We now give the proof of Theorem 40. Consider $\sigma_1, \sigma_2 : A$ finite (but not necessarily total) innocent strategies. If they are empty, there is nothing to prove. Otherwise, let $2n + 2$ be the length of s the longest P-view among them. *W.l.o.g.*, assume that $s \in \ulcorner \sigma_1 \urcorner$. Consider $\mathcal{p}_1$ the sub-augmentation of $\hat{\sigma_1}$ restricted to prefixes of s – it is a linear augmentation of length $2n + 2$, as shown on the right hand side of Figure 23. We build the **wide expansion**

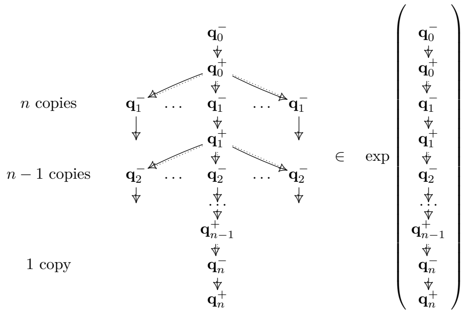


Figure 23 Wide expansion of a P-view.

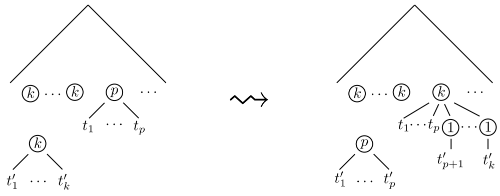


Figure 24 Rewriting trees.

$q_1 \in \exp(p_1)$ as shown in the left hand side of Figure 23: it is the unique $-$-obsessional and $+$-obsessional expansion of p_1 such that each fork of co-depth $2k$ has cardinality k (except for the initial move). So for any $1 \leq k \leq n$, they are $\frac{n!}{(n-k)!}$ copies of $\mathbf{q}_k^+$.

As $(\!|\sigma_1|\!) = (\!|\sigma_2|\!)$, Proposition 24 entails $(\!|\hat{\sigma_1}|\!) = (\!|\hat{\sigma_2}|\!)$. So there is $q_2 \in \exp(\hat{\sigma_2})$ along with some $\varphi : (\!|q_1|\!) \cong (\!|q_2|\!)$. By abuse of notation, we keep referring to events of $|q_2|$ with the same naming convention as in Figure 23, this is justified by φ. Then q_2 is a tree starting with $\mathbf{q}_0^-$. By courtesy it cannot break causal links from positives to negatives; so we may regard it as a tree whose nodes are the $\mathbf{q}_k^+$'s. For each $0 \leq k \leq n$, it has $n!/k!$ nodes of arity k (*arity* means the number of children in the tree) and by hypothesis its depth is bounded by $n+1$. The essence of the situation is captured by the following simplified setting:

Fix $n \in \mathbb{N}$. **Simple trees** are finite trees made of nodes (k) of arity k for $0 \leq k \leq n$. We set $T_0 =$ (0), and for $k > 0$, T_k is the tree with root (k) and k copies of T_{k-1} as children. If t is a simple tree, its **size** $\sharp t$ is its number of nodes, and its **depth** is the maximal number of nodes reached in a path. For instance, the depth of T_k is $k+1$ and its size is $\sharp T_k = k! \sum_{i=0}^{k} \frac{1}{i!}$.

Now, let us consider the set $\mathsf{Trees}(n)$ of simple trees of depth $\leq n+1$, and having, for $2 \leq k \leq n$, $\frac{n!}{k!}$ nodes (k), and arbitrarily many nodes (1) and (0). We prove:

▶ **Lemma 49.** *Let $t \in \mathsf{Trees}(n)$ of maximal size. Then, $t = T_n$.*

Proof. Seeking a contradiction, assume t is distinct from T_n. Consider a minimal node where they differ, *i.e.* closest to the root – say t has some (p) at the row corresponding to (k)'s in T_n. If $k = 0$ then $p > 0$ and this contradicts that the depth of t is less than n. So, $k \geq 1$. If $p > k$, then $p \geq 2$. But by minimality, t is the same as T_n for all rows closer to the root, so all (p) for $p > k$ are exhausted. Hence, $p < k$. If $k = 1$ and $p = 0$, then we may replace (p) with T_1, yielding $t' \in \mathsf{Trees}(n)$ of size strictly greater than $\sharp t$, contradicting maximality. Otherwise, $k \geq 2$. Then the number of nodes (k) is fixed, there are fewer of those on this row as for T_n, and they cannot occur on rows closer to the root. Therefore, there is an occurrence of (k) strictly deeper in t. We then perform the transformation as in Figure 24. This yields $t' \in \mathsf{Trees}(n)$. But $\sharp t' > \sharp t$, contradicting the maximality of t. ◀

Now, from q_2 we extract a simple tree $t(q_2) \in \mathsf{Trees}(n)$ as follows. For each $0 \leq k \leq n$, to each $\mathbf{q}_{n-k}^+$ we associate a node (k), with edges as in q_2. Because all P-views in σ_2 have length lesser or equal to $2n+2$ and $q_2 \in \exp(\hat{\sigma_2})$, $t(q_2)$ has depth $\leq n+1$. The constraints on the number of each node are ensured by the isomorphism $\varphi : (\!|q_1|\!) \cong (\!|q_2|\!)$. Therefore $t(q_2) \in \mathsf{Trees}(n)$, and by Lemma 49, $t(q_2) = T_n$.

This induces directly an isomorphism ψ between $(\mathscr{q}_1, \leq_{\mathscr{q}_1})$ and $(\mathscr{q}_2, \leq_{\mathscr{q}_2})$. We must still check that ψ preserves $\rightarrowtriangle_{(\!|\mathscr{q}_1|\!)}$, *i.e.* justification pointers. Assume $\mathbf{q}_j^- \rightarrowtriangle_{(\!|\mathscr{q}_1|\!)} \mathbf{q}_i^+$. Then, $\mathbf{q}_i^+$ has arity $n-i$, and $\mathsf{just}(\mathsf{just}(\mathbf{q}_i^+)) = \mathbf{q}_j^+$ of arity $n-j$. But then, by construction, it follows that for any move $a^+ \in |\mathscr{q}_1|$ of arity $n-i$, $\mathsf{just}(\mathsf{just}(a))$ has arity $n-j$. This is transported by the isomorphism φ, so this property also holds for $\mathscr{q}_2$. Now, consider $\psi(\mathbf{q}_i^+) \in |\mathscr{q}_2|$. Its justifier is some $b^- \in |\mathscr{q}_2|$ such that $\mathsf{just}(b^-)$ has arity $n-j$. But as arity is preserved by ψ, there is only one move with this property in the causal history of $\psi(\mathbf{q}_i^+)$, namely $\psi(\mathbf{q}_j^-)$. So, ψ preserves pointers. It also preserves the image in the arena: by construction of $\mathscr{q}_1$, all positive moves with the same arity have the same image, and all negative moves whose justifiers have the same arity also have the same image. Hence, the image only depends on the arity, which is a property of $(\!|\mathscr{q}_1|\!)$; and since $(\!|\mathscr{q}_1|\!)$ and $(\!|\mathscr{q}_2|\!)$ are isomorphic, the same holds for $\mathscr{q}_2$. Since ψ preserves arity and justifiers, it also preserves the image in the arena.

By construction, maximal branches of $\mathscr{q}_1$ have for image in the arena the chain of prefixes of s; by the iso it is also true for maximal branches of $\mathscr{q}_2$. Since $\mathscr{q}_2 \in \exp(\hat{\sigma_2})$, $s \in \ulcorner \hat{\sigma_2} \urcorner$.

Synthetic Undecidability of MSELL via FRACTRAN Mechanised in Coq

Dominique Larchey-Wendling
Université de Lorraine, CNRS, LORIA, Vandœuvre-lès-Nancy, France

Abstract

We present an alternate undecidability proof for entailment in (intuitionistic) multiplicative sub-exponential linear logic (MSELL). We contribute the result and its mechanised proof to the Coq library of synthetic undecidability. The result crucially relies on the undecidability of the halting problem for two counters Minsky machines, which we also hand out to the library. As a seed of undecidability, we start from FRACTRAN halting which we (many-one) reduce to Minsky machines termination by implementing Euclidean division using two counters only. We then give an alternate presentation of those two counters machines as sequent rules, where computation is performed by proof-search, and halting reduced to provability. We use this system called non-deterministic two counters Minsky machines to describe and compare both the legacy reduction to linear logic, and the more recent reduction to MSELL. In contrast with that former MSELL undecidability proof, our correctness argument for the reduction uses trivial phase semantics in place of a focused calculus.

2012 ACM Subject Classification Theory of computation → Models of computation; Theory of computation → Linear logic; Theory of computation → Type theory

Keywords and phrases Undecidability, computability theory, many-one reduction, Minsky machines, Fractran, sub-exponential linear logic, Coq

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.18

Supplementary Material *Software (Source Code)*:
`https://github.com/uds-psl/coq-library-undecidability/releases/tag/FSCD-2021`
archived at `swh:1:rev:4115398f10c42a41833036f8c4500f24233cc9a7`

Funding *Dominique Larchey-Wendling*: partially supported by the TICAMORE project (ANR grant 16-CE91-0002).

1 Introduction

In the late 80s, Lincoln et al. [17] gave a first proof of the undecidability of propositional linear logic (LL) via a many-one reduction from "and-branching two-counter machines without zero-test," a variant of Minsky machines extended with a *fork* instruction. The ability of LL to simulate the increment and decrement operations characteristic of Petri net operations was spotted very early and lead to paradigmatically characterise LL as a logic for counting resources. Critically, the exponential modality ! can be exploited to allow *unbounded reuse* of some specific resources like (Petri net) transitions or (Minsky machines) instructions.

To establish undecidability, one needed of course to go beyond Petri nets because those have a decidable reachability problem, a major result from the early 80s with a very involved proof still actively revisited nowadays [18, 15, 4, 16, 5]. As opposed to Minsky machines, Petri nets are not able to perform zero tests *combined* with a jump. Hence, the main idea of the reduction was to use forking to separate comparison with zero from jumping. In there, the additive conjunction of LL plays a central role:

$$\frac{\Sigma \vdash \alpha = 0 \qquad \Sigma \vdash \mathtt{jump}}{\Sigma \vdash \alpha = 0 \mathbin{\&} \mathtt{jump}}$$

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi; Article No. 18; pp. 18:1–18:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Indeed this right introduction rule *duplicates* the context Σ in the left and right sub-proofs which allows to delegate checking for emptiness in the left branch, and jumping in the right branch, the requirement of the two premises ensuring the correctness of the combination.

The same idea was then exploited to establish the undecidability of smaller fragments of LL [10, 11, 13]. In our own work [8], we gave the first mechanisation of the undecidability of the elementary fragment of LL in Coq, and hence ILL, based on this forking idea as well.

The multiplicative and exponential fragment (MELL) of linear logic lacks additive connectives, and is thus unable to duplicate the context. Arguably, the question of its decidability is the most important open conjecture (see e.g. [14]) in the context of LL, even with some claimed proof of decidability [1], later refuted [20]. The recent encoding of two counters Minsky machines in a fragment of LL lacking additives opened a new logical perspective on the MELL question [2]. Indeed, at the cost of a more complex modal structure, forking with & can be replaced with a constraint on modalities in the *promotion rule*. This extension of MELL is called multiplicative sub-exponential linear logic (MSELL).

In this paper, we mechanise this reduction from two counters Minsky machines to MSELL, following the encoding of [2]. However, we proceed in the intuitionistic version of the logic (two sided sequents with exactly one conclusion formula) that we call IMSELL. That fragment only involves the linear implication $\multimap$ and the modalities $!^m$ with $m \in \Lambda = \{a, b, \infty\}$, so it is short to describe. It is also convenient for comparing with our previous encoding in (elementary) intuitionistic LL [13, 8]. Schematically, we describe and mechanise the following many-one reduction chain, explained below:

$$\mathsf{FRACTRAN}_{\mathrm{reg}} \preceq \mathsf{MMA0}_2 \preceq \mathsf{MM}_{\mathrm{nd}} \preceq \mathsf{IMSELL}_\Lambda$$

Our work is based on and contributes to the Coq library of undecidability proofs; see [9] for a quick overview. As opposed to the legacy LL argument of forking, which can cope with Minsky machines using arbitrary many counters, the MSELL and IMSELL reductions rely on two counters machines in an essential way. Hence, we first had to implement the undecidability of the "halting on the zero state" problem for two counters Minsky machines, that we denote $\mathsf{MMA0}_2$; see Section 3. To establish this, we could follow the legacy reduction from many counters to just two by Minsky [19], that uses a Gödel coding of lists of natural numbers as essential trick. Following [12], we profit from the FRACTRAN language [3] that adequately abstracts away the Gödel coding phase, hence we establish the undecidability of $\mathsf{MMA0}_2$ by reducing from (regular) FRACTRAN halting instead, mainly by mechanising Euclidean division with two counters only.

In Section 4, we provide a sequent calculus style presentation of $\mathsf{MMA0}_2$, i.e. the instance $(\mathcal{M}, x, y)$ of $\mathsf{MMA0}_2$ is viewed as a sequent $\Sigma_\mathcal{M} /\!\!/_{\mathrm{n}}\, x \oplus y \vdash 1$, and the Minsky machine $\mathcal{M}$ starting at PC value 1 with register values (x, y) halts on the zero state if and only if the sequent $\Sigma_\mathcal{M} /\!\!/_{\mathrm{n}}\, x \oplus y \vdash 1$ has a derivation. We call this system and the associated problem non-deterministic two counters Minsky machines, denoted $\mathsf{MM}_{\mathrm{nd}}$. As $\mathsf{MM}_{\mathrm{nd}}$ is essentially a specialised proof theory for Minsky machines, reducing from it to logical entailment problems mainly consists in transformations of derivations. Hence Section 5, targeting IMSELL, can be understood from a proof theoretic perspective only. In there, we gives details of the reduction of two counters halting, explaining how the legacy fork trick for ILL is replaced by the modal constraints in the promotion rule of IMSELL, following [2]. Additionally, our proof of correctness of the reduction differs significantly: the former proof relies on the completeness of focused proof-search; we instead generalise our semantic argument [8], i.e. we prove and use the soundness of trivial phase semantics for IMSELL.

Our contributions in this work are the following. First, via a proof theoretic presentation of Minsky machines, a comparison of their encoding in ILL and in IMSELL, explaining precisely how and where forking is replaced with modalities. Then, a novel completeness proof of the IMSELL reduction based on the soundness of trivial phase semantics. On the implementation side, we provide the mechanized proof of the undecidability of two counters Minsky machines (with two different presentations), and of IMSELL. The Coq `8.13` code is available at

`https://github.com/uds-psl/coq-library-undecidability/tree/FSCD-2021`

and (sub-)section titles generally provide hyperlinks to the relevant source code. Our code extends the existing library with about 1800 loc, 1200 of which concern the reductions from FRACTRAN to MM_{nd}, and 600 more for the MM_{nd} to IMSELL reduction.

The paper describes the major steps of the implementation, in the language of type theory, but should be readable with only basic knowledge of it. We denote $\mathbb{P}$ (resp. $\mathbb{B}$ and $\mathbb{N}$) the type of propositions (resp. Booleans and natural numbers). We write $\mathbb{L}\,X$ for the type of *lists* over X, where $[\,]$ represents the empty list, $x :: l$ for the *cons* operation, $l +\!\!+\, l'$ for the *concatenation* of two lists, and $|l| : \mathbb{N}$ for the length of l. We write X^n for *vectors* $\vec{v}$ over type X with length $n : \mathbb{N}$, and $\mathbb{F}_n$ for the *finite type* with exactly n elements. Notations for lists are overloaded for vectors. Moreover, for $p : \mathbb{F}_n$ and $x : X$, we write $\vec{v}_p$ for the p-th component of $\vec{v} : X^n$ and $\vec{v}\,\{x/p\}$ when $\vec{v}$ is updated with x at component p. The (non-dependent) sum $A + B$ represents a computable/Boolean choice between an inhabitant of A or an inhabitant of B. In the case where A and B are propositions (i.e. of type $\mathbb{P}$), the sum $A + B : \texttt{Type}$ is stronger than the disjunction $A \vee B : \mathbb{P}$, because one cannot computably determine which of A or B holds in the later case. We also use the type-theoretic dependent sum $\Sigma_{x:A} B(x)$, denoted $\{x : A \mid B\,x\}$ in Coq,[1] inhabited by (Coq computable) values $x : A$ paired with a proof of $B\,x$.

The framework of synthetic computability [7] is based on the notion of many-one reduction. If $P : X \to \mathbb{P}$ is a predicate (on X) and $Q : Y \to \mathbb{P}$ is a predicate, we say that P *many-one reduces to* Q and write $P \preceq Q$ if there is a Coq function $f : X \to Y$ s.t. $\forall x : X,\ P\,x \leftrightarrow Q\,(f\,x)$, i.e. a *many-one reduction* from P to Q. Because we work in constructive (axiom-free) Coq, all definable functions are computable and thus the requirement of the computability of the reduction function f above can be discarded. If $P \preceq Q$ and P is undecidable then so is Q.

2 The FRACTRAN seed (files `FRACTRAN.v` and `fractran_utils.v`)

The FRACTRAN model of computation is very simple to describe. It was introduced by Conway [3] but its main idea, the Gödel coding of a list $[x_1; x_2; \ldots; x_n]$ of natural numbers as the number $p_1^{x_1} p_2^{x_2} \cdots p_n^{x_n}$, predates the introduction of FRACTRAN by several decades.

In the FRACTRAN formalism, programs are lists of formal fractions, i.e. terms Q of type $\mathbb{L}\,(\mathbb{N} \times \mathbb{N})$.[2] The state of a program is modelled as a natural number $x : \mathbb{N}$. A fraction p/q is *executable at state* x if $x{\cdot}p/q$ is a natural number (i.e. not a proper fraction) and in that case this is the new state. To allow FRACTRAN to discriminate, and b.t.w. turn it into a Turing complete model of computation, the first executable fraction in the list has to be picked up at each step of computation. The program Q *stops* when no fraction in the list is executable.

[1] or simply $\{x \mid B\,x\}$ when the type of x is guessable.
[2] For the moment, we can ignore the case of degenerate fractions like $p/0$.

Formally this prose translates in a straightforward *inductive definition*, not even involving the algebraic notion of fraction, and characterized by the two inductive rules below:

$$\frac{qy = px}{p/q :: Q \mathbin{/\!/_{\mathrm{F}}} x \succ y} \qquad \frac{q \nmid px \qquad Q \mathbin{/\!/_{\mathrm{F}}} x \succ y}{p/q :: Q \mathbin{/\!/_{\mathrm{F}}} x \succ y}$$

where $u \nmid v$ means u does not divide v, and $Q \mathbin{/\!/_{\mathrm{F}}} x \succ y$ reads as the FRACTRAN program Q *transforms state x into state y* in one step of computation. The computation is *terminated at x*, denoted $Q \mathbin{/\!/_{\mathrm{F}}} x \not\succ \star$, when there is no possibility to perform one step from x, and *termination from x*, denoted by $Q \mathbin{/\!/_{\mathrm{F}}} x \downarrow$, means there is exists a sequence of steps starting at state x at leading to the terminated state y. Formally, this gives us:

$$Q \mathbin{/\!/_{\mathrm{F}}} x \not\succ \star := \forall y, \neg(Q \mathbin{/\!/_{\mathrm{F}}} x \succ y) \quad \text{and} \quad Q \mathbin{/\!/_{\mathrm{F}}} x \downarrow := \exists y, (Q \mathbin{/\!/_{\mathrm{F}}} x \succ^* y \wedge Q \mathbin{/\!/_{\mathrm{F}}} y \not\succ \star)$$

There are some obvious quick remarks to make here: the empty program $Q = [\,]$ is terminated in any state; unless $Q = [\,]$, the state 0 is not terminated. The step relation is strongly decidable in the sense that one can discriminate between non-terminated and terminated states, and in the former case, computationally find a next state, expressed below using (Coq) dependent types:

▶ **Proposition 1.** *For any FRACTRAN program we have* $\forall x, \{y \mid Q \mathbin{/\!/_{\mathrm{F}}} x \succ y\} + (Q \mathbin{/\!/_{\mathrm{F}}} x \not\succ \star)$.

Proof. By structural induction on the list Q combined with Euclidean division. ◀

The dependent sum $\{y \mid Q \mathbin{/\!/_{\mathrm{F}}} x \succ y\}$ represents a (computable) state y together with a proof that y is next after x. The proposition $Q \mathbin{/\!/_{\mathrm{F}}} x \not\succ \star$ is for a proof that x is a terminated state. Finally, the outer sum $+$ represents a computable choice between the two alternatives.

Non-regular fractions like $0/0$ can make the computation non-deterministic; and non-proper fractions like $1/1$ or $6/2$ are always executable, implying that programs including such fractions have no terminating state. Non-deterministic step relations involves at least two different notions of termination, weak termination as defined above, and strong termination, when no infinite sequence of steps from x can exist. For our use of FRACTRAN, it does not matter because we only consider *regular FRACTRAN programs* where formal fractions $p/0$ are disallowed. Regular FRACTRAN is a universal model of computation, up to a Gödel encoding of natural numbers [3].[3]

▶ **Definition 2.** *A* $\mathsf{FRACTRAN}_{\mathrm{reg}}$ *instance is a pair composed of a list of regular formal fractions and a natural number, i.e. of type* $\{(Q, x) : \mathbb{L}\,(\mathbb{N} \times \mathbb{N}) \times \mathbb{N} \mid \forall p,\, p/0 \notin Q\}$, *and the question asked is whether* $Q \mathbin{/\!/_{\mathrm{F}}} x \downarrow$ *holds or not.*

Notice the use of a dependent sum in the type of instances where the predicate $\forall p,\ p/0 \notin Q$ acts as a guard against non-regular instances.

▶ **Theorem 3** (mechanized in [12]). *There is a many-one reduction from the Halting problem for single tape Turing machines to termination of regular FRACTRAN programs, i.e.* $\mathsf{Halt} \preceq \mathsf{FRACTRAN}_{\mathrm{reg}}$, *and thus* $\mathsf{FRACTRAN}_{\mathrm{reg}}$ *is undecidable.*

As a consequence, we can safely use $\mathsf{FRACTRAN}_{\mathrm{reg}}$ as our seed of undecidability for the chain of many-one reductions described in this paper.

---

[3] However, e.g. the function $n \mapsto 0$ cannot be *directly* represented by a FRACTRAN program where n would be the starting state leading, after finitely many steps of computation, to the 0 terminated state.

3 From FRACTRAN to two registers alternate Minsky machines

3.1 Alternate Minsky machines (files `MM.v` and `mma_defs.v`)

We describe alternate n counters (or registers) Minsky machines, where states are described as $(i, \vec{v}) : \mathbb{N} \times \mathbb{N}^n$. The number $i : \mathbb{N}$ is the current program counter (PC) value and the vector $\vec{v} : \mathbb{N}^n$ describes the n current values of the registers. When convenient, we also denote states as st, st_1... Instructions consist of either incrementing $\texttt{INC}_\mathrm{a}\, x$ a register by one, or decrementing $\texttt{DEC}_\mathrm{a}\, x\, j$ a register by one. Notice that when the register values 0, it is not possible to decrement it. So a conditional jump at j helps at discriminating between the zero and non-zero cases. Unless there is a conditional jump, the default behaviour after the register is updated is to jump to the next instruction at $\mathrm{PC} + 1$. In contrast with [8, 12] where the $\texttt{DEC}\, x\, j$ instruction jumps at j when $\vec{v}_x$ is empty, here in $\texttt{DEC}_\mathrm{a}\, x\, j$, the jump occurs when decrementing is possible, and this is the reason we call these machines alternate and suffix instructions with an "a" just as a reminder for this alternate semantics. Hence a single (atomic) step of computation is described by the following relation

$$\begin{array}{ll}
\texttt{INC}_\mathrm{a}\, x \mathbin{/\!\!/_\mathrm{a}} (i, \vec{v}) \succ \big(1{+}i, \vec{v}\{(1{+}u)/x\}\big) & \text{when } \vec{v}_x = u \\
\texttt{DEC}_\mathrm{a}\, x\, j \mathbin{/\!\!/_\mathrm{a}} (i, \vec{v}) \succ \big(j, \vec{v}\{u/x\}\big) & \text{when } \vec{v}_x = 1{+}u \\
\texttt{DEC}_\mathrm{a}\, x\, j \mathbin{/\!\!/_\mathrm{a}} (i, \vec{v}) \succ (1{+}i, \vec{v}) & \text{when } \vec{v}_x = 0
\end{array}$$

where $\sigma \mathbin{/\!\!/_\mathrm{a}} (i_1, \vec{v}_1) \succ (i_2, \vec{v}_2)$ reads as the MMA_n instruction σ at PC value i_1 transforms the state $(i_1, \vec{v}_1)$ into the state $(i_2, \vec{v}_2)$. Notice that this alternate semantics allows to implement a universal jump without needing an empty register, which will be critical when we will need to limit the number of registers to $n = 2$.

▶ **Proposition 4.** *The step relation for alternate Minsky machines is deterministic and total:*
1. *for any states st, st_1 and st_2, if $\sigma \mathbin{/\!\!/_\mathrm{a}} st \succ st_1$ and $\sigma \mathbin{/\!\!/_\mathrm{a}} st \succ st_2$ then $st_1 = st_2$;*
2. *for any state $(i_1, \vec{v}_1)$, one can compute a state $(i_2, \vec{v}_2)$ such that $\sigma \mathbin{/\!\!/_\mathrm{a}} (i_1, \vec{v}_1) \succ (i_2, \vec{v}_2)$.*

This means that starting from state $(i_1, \vec{v}_1)$, the instruction σ at PC value i_1 (provided there is one) changes the state in exactly one possible way, and the new state $(i_2, \vec{v}_2)$ is Coq-computable from the initial state $(i_1, \vec{v}_1)$. So the only way for such programs to terminate is to jump to a PC value which holds no instruction.

A program is pair $(i, P) : \mathbb{N} \times \mathbb{L}\,\mathsf{MMA}_n$ composed of the PC value of its first instruction and the sequence P of consecutive instructions of which is it composed. Informally, the program $(i, [\sigma_0; \ldots; \sigma_{m-1}])$ would be read as e.g. $i : \sigma_0;\ 1{+}i : \sigma_1;\ \ldots;\ m{-}1{+}i : \sigma_{m-1}$ using labelled instructions. We define the k-steps relation for a program (i, P) inductively with

$$\frac{}{(i, P) \mathbin{/\!\!/_\mathrm{a}} st \succ^0 st} \qquad \frac{i_1 = |L| + i \quad P = L \mathbin{+\!\!+} \sigma :: R \quad \sigma \mathbin{/\!\!/_\mathrm{a}} (i_1, \vec{v}_1) \succ st_2 \quad (i, P) \mathbin{/\!\!/_\mathrm{a}} st_2 \succ^k st_3}{(i, P) \mathbin{/\!\!/_\mathrm{a}} (i_1, \vec{v}_1) \succ^{1+k} st_3}$$

where the constraints $i_1 = |L| + i$ and $P = L \mathbin{+\!\!+} \sigma :: R$ impose that the instruction at PC value i_1 of (i, P) is σ. From its structure as lists of instructions, there is at most one instruction at a given PC value and thus, the k-steps relation is also deterministic. However, it can be non-total if a jump outside of the interval $[i, |P| - 1 + i]$ occurs, the lack of an instruction blocking the computation. We write $\texttt{out}\ j\ (i, P) := j < i \lor |P| + i \le j$ when there is no instruction at j in (i, P), and because of Proposition 4 (totality), blocked states are exactly those outside of the code, i.e. $\texttt{out}\ i_1\ (i, P) \leftrightarrow \forall st_2, \neg\, (i, P) \mathbin{/\!\!/_\mathrm{a}} (i_1, \vec{v}_1) \succ^1 st_2$.

We define the predicates of *computation*, of *progress*, of *output* and of *termination* as:

$$
\begin{array}{rll}
(i,P) /\!\!/_{\mathrm{a}}\, st_1 \succ^* st_2 & := \exists k, (i,P) /\!\!/_{\mathrm{a}}\, st_1 \succ^k st_2 & \text{(computation)} \\
(i,P) /\!\!/_{\mathrm{a}}\, st_1 \succ^+ st_2 & := \exists k > 0, (i,P) /\!\!/_{\mathrm{a}}\, st_1 \succ^k st_2 & \text{(progress)} \\
(i,P) /\!\!/_{\mathrm{a}}\, st_1 \rightsquigarrow (i_2, \vec{v}_2) & := (i,P) /\!\!/_{\mathrm{a}}\, st_1 \succ^* (i_2,\vec{v}_2) \wedge \mathtt{out}\ i_2\ (i,P) & \text{(output)} \\
(i,P) /\!\!/_{\mathrm{a}}\, st_1 \downarrow & := \exists st_2, (i,P) /\!\!/_{\mathrm{a}}\, st_1 \rightsquigarrow st_2 & \text{(termination)}
\end{array}
$$

output meaning that we have computed until we reach a state blocking the computation.

▶ **Definition 5.** *The problems* MMA_2 *and* $\mathsf{MMA0}_2$ *have the same instances: a pair* $(P,\vec{v})$ *where* P *is a list of* MMA_2 *instructions (starting at PC value 1) and the vector* $\vec{v} : \mathbb{N}^2$ *represents the initial values of the two registers.* MMA_2 *asks for termination, i.e.* $(1,P) /\!\!/_{\mathrm{a}}\, (1,\vec{v}) \downarrow$. $\mathsf{MMA0}_2$ *asks for termination on the zero state, i.e.* $(1,P) /\!\!/_{\mathrm{a}}\, (1,\vec{v}) \rightsquigarrow (0,[0;0])$.

We mention that there is substantial machinery for (alternate) Minsky machines, and more generally PC based state machines, in the Coq library of undecidable problem initially described in [8]. These tools enable *modular reasoning* in those models of computation.

3.2 A basic MMA_n library up to Euclidean division (file `mma_utils.v`)

We specify, implement and verify a small library to compute some basic operations with MMA_n. For this section, $n : \mathbb{N}$ is a fixed number of registers but all the below sub-programs involve at most two registers. In the coming statements, the vector $\vec{v} : \mathbb{N}^n$ is implicitly universally quantified over. The names $i,j,p,q : \mathbb{N}$ range over PC values, $k : \mathbb{N}$ over natural number constants, and the names $x,t,s,d : \mathbb{F}_n$ over registers indices.

Let us start with the easy simulations of an unconditional jump, the nullification of register x and the operation that adds k units to register x.

▶ **Proposition 6.** *For* $i,j : \mathbb{N}$ *and* $x : \mathbb{F}_n$ *we have* $(i, \mathtt{JUMP}_{\mathrm{a}}\, j\, x) /\!\!/_{\mathrm{a}}\, (i,\vec{v}) \succ^+ (j,\vec{v})$ *where* $\mathtt{JUMP}_{\mathrm{a}}\, j\, x := [\mathtt{INC}_{\mathrm{a}}\, x; \mathtt{DEC}_{\mathrm{a}}\, x\, j]$.

▶ **Proposition 7.** *For* $x : \mathbb{F}_n$ *and* $i : \mathbb{N}$*, we have* $(i, \mathtt{NULL}_{\mathrm{a}}\, x\, i) /\!\!/_{\mathrm{a}}\, (i,\vec{v}) \succ^+ (1+i, \vec{v}\{0/x\})$ *where* $\mathtt{NULL}_{\mathrm{a}}\, x\, i := [\mathtt{DEC}_{\mathrm{a}}\, x\, i]$.

▶ **Proposition 8.** *For* $i,k : \mathbb{N}$*,* $x : \mathbb{F}_n$*, we have* $(i, \mathtt{INCS}_{\mathrm{a}}\, x\, k) /\!\!/_{\mathrm{a}}\, (i,\vec{v}) \succ^* (k+i, \vec{v}\{(k+\vec{v}_x)/x\})$ *where* $\mathtt{INCS}_{\mathrm{a}}\, x\, k := [\mathtt{INC}_{\mathrm{a}}\, x; \dots; \mathtt{INC}_{\mathrm{a}}\, x]$ *is of length* $|\mathtt{INCS}_{\mathrm{a}}\, x\, k| = k$.

Then we simulate test for emptiness of register x, jumping to PC value p when x is empty, or else to the end of the sub-program otherwise. Registers are restored to their initial values when the sub-program is finished (assuming p points outside of its code).

▶ **Proposition 9.** *For* $x : \mathbb{F}_n$ *and* $p,i : \mathbb{N}$ *we have* $(i, \mathtt{EMPTY}_{\mathrm{a}}\, x\, p\, i) /\!\!/_{\mathrm{a}}\, (i,\vec{v}) \succ^+ (j,\vec{v})$ *where* $\mathtt{EMPTY}_{\mathrm{a}}\, x\, p\, i := [\mathtt{DEC}_{\mathrm{a}}\, x\, (3+i); \mathtt{JUMP}_{\mathrm{a}}\, p\, x; \mathtt{INC}_{\mathrm{a}}\, x]$*, and* $j := p$ *in case* $\vec{v}_x = 0$*, or else* $j := 4+i$ *in case* $\vec{v}_x \neq 0$.

Notice that this sub-program is of length $|\mathtt{EMPTY}_{\mathrm{a}}\, x\, p\, i| = 4$ (despite looking 3), because we abuse the list notation $[\dots;\dots;\dots]$ by allowing dots to be not only single instructions but also lists of instructions such as $\mathtt{JUMP}_{\mathrm{a}}\, p\, x$. Hence, $\mathtt{EMPTY}_{\mathrm{a}}\, x\, p\, i$ is formally defined as $\mathtt{DEC}_{\mathrm{a}}\, x\, (3+i) :: \mathtt{JUMP}_{\mathrm{a}}\, p\, x \mathbin{+\!\!+} \mathtt{INC}_{\mathrm{a}}\, x :: [\,]$ but we choose the friendly display for readability.

We now simulate the transfer of the contents of register s (for source) to d (for destination).

▶ **Proposition 10.** *For* $s \neq d : \mathbb{F}_n$ *and* $i : \mathbb{N}$ *we have* $(i, \mathtt{TRANSFER}_{\mathrm{a}}\, s\, d\, i) /\!\!/_{\mathrm{a}}\, (i,\vec{v}) \succ^+ (3+i, \vec{w})$ *where* $\mathtt{TRANSFER}_{\mathrm{a}}\, s\, d\, i := [\mathtt{INC}_{\mathrm{a}}\, d; \mathtt{DEC}_{\mathrm{a}}\, s\, i; \mathtt{DEC}_{\mathrm{a}}\, d\, (3+i)]$ *and* $\vec{w} := \vec{v}\{0/s\}\{(\vec{v}_s + \vec{v}_d)/d\}$.

We simulate multiplication of a register by a constant. The idea is similar to transfer but instead of transferring one for one, when one unit is removed from s, k units are added to d.

▶ **Proposition 11.** *For* $s \neq d : \mathbb{F}_n$, $k, i : \mathbb{N}$ *we have* $(i, \mathtt{MULT\_CST}_\mathrm{a}\, s\, d\, k\, i) /\!\!/_\mathrm{a} (i, \vec{v}) \succ^+ (j, \vec{w})$ *where* $\mathtt{MULT\_CST}_\mathrm{a}\, s\, d\, k\, i := [\mathtt{DEC}_\mathrm{a}\, s\, (3+i); \mathtt{JUMP}_\mathrm{a}\, (5+k+i)\, s; \mathtt{INCS}_\mathrm{a}\, d\, k; \mathtt{JUMP}_\mathrm{a}\, i\, s]$, $j := 5 + k + i$, $\vec{w} := \vec{v}\{0/s\}\{(k\vec{v}_s + \vec{v}_d)/d\}$, *and* $|\mathtt{MULT\_CST}_\mathrm{a}\, s\, d\, k\, i| = 5 + k$.

We simulate the minus k operation (with overflow management), jumping to PC value p when k units can be removed from register x, or else to PC value q when register x contains less than k units (overflow).

▶ **Proposition 12.** *For* $p, q, k, i : \mathbb{N}$ *and* $x : \mathbb{F}_n$, *we define*

$$\mathtt{DECS}_\mathrm{a}\, x\, p\, q\, k\, i := [\mathtt{DEC}_\mathrm{a}\, x\, (3+i); \mathtt{JUMP}_\mathrm{a}\, q\, x; \ldots; \mathtt{DEC}_\mathrm{a}\, x\, (3k+i); \mathtt{JUMP}_\mathrm{a}\, q\, x; \mathtt{JUMP}_\mathrm{a}\, p\, x]$$

where the pattern $\mathtt{DEC}_\mathrm{a}\, x\, (3u+i); \mathtt{JUMP}_\mathrm{a}\, q\, x$ *is repeated for* $u = 1, \ldots, k$.

Depending on the comparison between $\vec{v}_x$ *and* k, *we have the following:*
- *if* $\vec{v}_x < k$ *then* $(i, \mathtt{DECS}_\mathrm{a}\, x\, p\, q\, k\, i) /\!\!/_\mathrm{a} (i, \vec{v}) \succ^+ (q, \vec{v}\{0/x\})$;
- *if* $\vec{v}_x \geq k$ *then* $(i, \mathtt{DECS}_\mathrm{a}\, x\, p\, q\, k\, i) /\!\!/_\mathrm{a} (i, \vec{v}) \succ^+ (p, \vec{v}\{\vec{v}_x - k/x\})$.

Using an extra temporary register t, we implement a non-destructive minus k operation (with overflow management).

▶ **Proposition 13.** *For* $x \neq t : \mathbb{F}_n$ *and* $p, q, k, i : \mathbb{N}$, *we define*

$$\mathtt{DECS\_COPY}_\mathrm{a}\, x\, t\, p\, q\, k\, i := \left[\begin{array}{l} \mathtt{DEC}_\mathrm{a}\, x\, (4-1+i); \mathtt{JUMP}_\mathrm{a}\, q\, x; \mathtt{INC}_\mathrm{a}\, t; \\ \ldots \\ \mathtt{DEC}_\mathrm{a}\, x\, (4k-1+i); \mathtt{JUMP}_\mathrm{a}\, q\, x; \mathtt{INC}_\mathrm{a}\, t; \\ \mathtt{JUMP}_\mathrm{a}\, p\, x \end{array}\right]$$

where the pattern $\mathtt{DEC}_\mathrm{a}\, x\, (4u-1+i); \mathtt{JUMP}_\mathrm{a}\, q\, x; \mathtt{INC}_\mathrm{a}\, t$ *is repeated for* $u = 1, \ldots, k$.

Depending on the comparison between $\vec{v}_x$ *and* k, *we have the following:*
- *if* $\vec{v}_x < k$ *then* $(i, \mathtt{DECS\_COPY}_\mathrm{a}\, x\, t\, p\, q\, k\, i) /\!\!/_\mathrm{a} (i, \vec{v}) \succ^+ (q, \vec{v}\{0/x\}\{(\vec{v}_x + \vec{v}_t)/t\})$;
- *if* $\vec{v}_x \geq k$ *then* $(i, \mathtt{DECS\_COPY}_\mathrm{a}\, x\, t\, p\, q\, k\, i) /\!\!/_\mathrm{a} (i, \vec{v}) \succ^+ (p, \vec{v}\{(\vec{v}_x - k)/x\}\{(k + \vec{v}_t)/t\})$.

The length is $|\mathtt{DECS\_COPY}_\mathrm{a}\, x\, t\, p\, q\, k\, i| = 2 + 4k$.

Notice that the initial value of t has to be known if one wants to recover the initial value of x, e.g. if the initial value of t is 0 and x contains less than k units, then once the computation is finished, t contains a copy of the initial value of x.

We implement a non-destructive computation of a divisibility test of register x by a constant $k > 0$, using a spare register t to preserve the initial value of x.

▶ **Proposition 14.** *For* $x, t : \mathbb{F}_n$ *and* $p, q, k, i : \mathbb{N}$, *we define*

$$\mathtt{MOD\_CST}_\mathrm{a}\, x\, t\, p\, q\, k\, i := [\mathtt{EMPTY}_\mathrm{a}\, x\, p\, i; \mathtt{DECS\_COPY}_\mathrm{a}\, x\, t\, i\, q\, k\, (4+i)]$$

and we check the identity $|\mathtt{MOD\_CST}_\mathrm{a}\, x\, t\, p\, q\, k\, i| = 6 + 4k$. *Assuming* $x \neq t$ *and* $k > 0$, *we have* $(i, \mathtt{MOD\_CST}_\mathrm{a}\, x\, t\, p\, q\, k\, i) /\!\!/_\mathrm{a} (i, \vec{v}) \succ^+ (j, \vec{v}\{0/s\}\{(\vec{v}_x + \vec{v}_t)/t\})$ *where* $j := p$ *when* k *divides* $\vec{v}_x$, *and* $j := q$ *otherwise.*

We now implement division by a constant $k > 0$. It will only work when the contents of the input register s is a multiple of k and the quotient is then stored in d.

▶ **Proposition 15.** *For* $s, d : \mathbb{F}_n$ *and* $k, i : \mathbb{N}$, *we define*

$$\mathtt{DIV\_CST}_\mathrm{a}\, s\, d\, k\, i := [\mathtt{DECS}_\mathrm{a}\, s\, (2+3k+i)\, (5+3k+i)\, k\, i; \mathtt{INC}_\mathrm{a}\, d; \mathtt{JUMP}_\mathrm{a}\, i\, s]$$

and we check the identity $|\mathtt{DIV\_CST}_\mathrm{a}\, s\, d\, k\, i| = 5 + 3k$. *Assuming* $s \neq d$, $k > 0$ *and* $\vec{v}_s = ak$, *we have* $(i, \mathtt{DIV\_CST}_\mathrm{a}\, s\, d\, k\, i) /\!\!/_\mathrm{a} (i, \vec{v}) \succ^+ (5 + 3k + i, \vec{v}\{0/s\}\{(a + \vec{v}_d)/d\})$.

3.3 Compiling regular FRACTRAN programs (file `fractran_mma.v`)

We are now in position to compile regular FRACTRAN programs (with no $p/0$ fractions). We start with a sub-program for simulating the FRACTRAN step relation for one regular fraction p/q then we will chain those sub-programs.

We fix $n := 2$, and $s := 0 : \mathbb{F}_2$ and $d := 1 : \mathbb{F}_2$ are the two available registers for two counters alternate Minsky machines. Let use assume a regular fraction, i.e. $p, q : \mathbb{N}$ with $q \neq 0$, and $i, j : \mathbb{N}$ where i the starting PC value of the sub-program.

To help the readability of the following code, we decorate it with relevant labels (PC values), although those are not formally present in the mechanisation:

$$(i, \texttt{FRAC\_ONE}_{\mathrm{a}}\, p\, q\, i\, j) := \left[\begin{array}{ll} i_0\colon & \texttt{MULT\_CST}_{\mathrm{a}}\, s\, d\, p\, i_0; \\ i_1\colon & \texttt{MOD\_CST}_{\mathrm{a}}\, d\, s\, i_2\, i_5\, q\, i_1; \\ i_2\colon & \texttt{DIV\_CST}_{\mathrm{a}}\, s\, d\, q\, i_2; \\ i_3\colon & \texttt{TRANSFER}_{\mathrm{a}}\, d\, s\, i_3; \\ i_4\colon & \texttt{JUMP}_{\mathrm{a}}\, j\, d; \\ i_5\colon & \texttt{DIV\_CST}_{\mathrm{a}}\, s\, d\, p\, i_5; \\ i_6\colon & \texttt{TRANSFER}_{\mathrm{a}}\, d\, s\, i_6 \\ i_7\colon & \end{array}\right] \quad \text{where} \quad \begin{cases} i_0 := i, \\ i_1 := 5 + p + i_0, \\ i_2 := 6 + 4q + i_1, \\ i_3 := 5 + 3q + i_2, \\ i_4 := 3 + i_3, \\ i_5 := 2 + i_4, \\ i_6 := 5 + 3p + i_5, \\ i_7 := 3 + i_6 \end{cases}$$

▶ **Proposition 16.** *$|\texttt{FRAC\_ONE}_{\mathrm{a}}\, p\, q\, i\, j| = 29 + 4p + 7q$ and $i_7 = |\texttt{FRAC\_ONE}_{\mathrm{a}}\, p\, q\, i\, j| + i$.*

▶ **Proposition 17.** *If $qy = px$ then $(i, \texttt{FRAC\_ONE}_{\mathrm{a}}\, p\, q\, i\, j) /\!/_{\mathrm{a}} (i, [x; 0]) \succ^+ (j, [y; 0])$.*

▶ **Proposition 18.** *If $q \nmid px$ then $(i, \texttt{FRAC\_ONE}_{\mathrm{a}}\, p\, q\, i\, j) /\!/_{\mathrm{a}} (i, [x; 0]) \succ^+ (i_7, [x; 0])$.*

Proof. The proof of Proposition 17 (resp. 18) is sketched in Appendix A (resp. B). ◀

Hence $(i, \texttt{FRAC\_ONE}_{\mathrm{a}}\, p\, q\, i\, j)$ performs the multiplication of x by p/q if the result is a natural number, transferring the control to PC value j, or else, would the result be a proper fraction, the registers are globally unmodified and the PC is transferred at i_7, the end of this sub-program. Notice that the register d is assumed to be initially empty.

We now chain those sub-programs to simulate one step of a regular FRACTRAN program, encoding a list Q of fractions by structural recursion on Q:

$$\texttt{FRAC\_STEP}_{\mathrm{a}}\ j\ [\,]\ i := [\,] \qquad \texttt{FRAC\_STEP}_{\mathrm{a}}\ j\ (p/q :: Q)\ i := P +\!\!+ \texttt{FRAC\_STEP}_{\mathrm{a}}\ j\ Q\ (|P| + i)$$
$$\text{where } P := \texttt{FRAC\_ONE}_{\mathrm{a}}\, p\, q\, i\, j$$

▶ **Lemma 19.** *For any regular FRACTRAN program $Q : \mathbb{L}\,(\mathbb{N} \times \mathbb{N})$ and any $i, j, x, y : \mathbb{N}$, if $Q /\!/_{\mathrm{F}}\, x \succ y$ then $(i, \texttt{FRAC\_STEP}_{\mathrm{a}}\ j\ Q\ i) /\!/_{\mathrm{a}} (i, [x; 0]) \succ^+ (j, [y; 0])$.*

Proof. By induction on the predicate $Q /\!/_{\mathrm{F}}\, x \succ y$ using Propositions 17 and 18. ◀

▶ **Lemma 20.** *For any regular FRACTRAN program $Q : \mathbb{L}\,(\mathbb{N} \times \mathbb{N})$ and any $i, j, x : \mathbb{N}$, if $Q /\!/_{\mathrm{F}}\, x \not\succ \star$ then $(i, \texttt{FRAC\_STEP}_{\mathrm{a}}\ j\ Q\ i) /\!/_{\mathrm{a}} (i, [x; 0]) \succ^* (|\texttt{FRAC\_STEP}_{\mathrm{a}}\ j\ Q\ i| + i, [x; 0])$.*

Proof. By induction on Q using Proposition 18. ◀

The instance $\texttt{FRAC\_STEP}_{\mathrm{a}}\ 1\ Q\ 1$ starts at $i = 1$ and loops on itself ($j = 1$) until no fraction can be executed. In addition, we finish by nullifying s and then jump to PC value 0:

$$\texttt{FRAC\_MMA}_{\mathrm{a}}\, Q := \texttt{FRAC\_STEP}_{\mathrm{a}}\ 1\ Q\ 1 +\!\!+ \texttt{NULL}_{\mathrm{a}}\, s\, \big(|\texttt{FRAC\_STEP}_{\mathrm{a}}\ 1\ Q\ 1| + 1\big) +\!\!+ \texttt{JUMP}_{\mathrm{a}}\, 0\, s$$

$$\frac{}{\Sigma /\!/_{\mathrm{n}}\, 0 \oplus 0 \vdash p}\ \texttt{STOP}_{\mathrm{n}}\, p \in \Sigma$$

$$\frac{\Sigma /\!/_{\mathrm{n}}\, 1{+}a \oplus b \vdash q}{\Sigma /\!/_{\mathrm{n}}\, a \oplus b \vdash p}\ \texttt{INC}_{\mathrm{n}}\, \alpha\, p\, q \in \Sigma \qquad \frac{\Sigma /\!/_{\mathrm{n}}\, a \oplus b \vdash q}{\Sigma /\!/_{\mathrm{n}}\, 1{+}a \oplus b \vdash p}\ \texttt{DEC}_{\mathrm{n}}\, \alpha\, p\, q \in \Sigma \qquad \frac{\Sigma /\!/_{\mathrm{n}}\, 0 \oplus b \vdash q}{\Sigma /\!/_{\mathrm{n}}\, 0 \oplus b \vdash p}\ \texttt{ZERO}_{\mathrm{n}}\, \alpha\, p\, q \in \Sigma$$

$$\frac{\Sigma /\!/_{\mathrm{n}}\, a \oplus 1{+}b \vdash q}{\Sigma /\!/_{\mathrm{n}}\, a \oplus b \vdash p}\ \texttt{INC}_{\mathrm{n}}\, \beta\, p\, q \in \Sigma \qquad \frac{\Sigma /\!/_{\mathrm{n}}\, a \oplus b \vdash q}{\Sigma /\!/_{\mathrm{n}}\, a \oplus 1{+}b \vdash p}\ \texttt{DEC}_{\mathrm{n}}\, \beta\, p\, q \in \Sigma \qquad \frac{\Sigma /\!/_{\mathrm{n}}\, a \oplus 0 \vdash q}{\Sigma /\!/_{\mathrm{n}}\, a \oplus 0 \vdash p}\ \texttt{ZERO}_{\mathrm{n}}\, \beta\, p\, q \in \Sigma$$

Figure 1 The $\mathsf{S\text{-}MM}_{\mathrm{nd}}$ sequent style calculus for non-deterministic two counters Minsky machines.

▶ **Theorem 21.** *For any regular* FRACTRAN *program* $Q : \mathbb{L}\,(\mathbb{N} \times \mathbb{N})$ *and any* $x : \mathbb{N}$*, the three following properties are equivalent:*

1. $Q /\!/_{\mathrm{F}}\, x \downarrow$;
2. $(1, \texttt{FRAC\_MMA}_{\mathrm{a}}\, Q) /\!/_{\mathrm{a}}\, (1, [x; 0]) \rightsquigarrow (0, [0; 0])$;
3. $(1, \texttt{FRAC\_MMA}_{\mathrm{a}}\, Q) /\!/_{\mathrm{a}}\, (1, [x; 0]) \downarrow$.

Proof. A sketch of the proof can be found in Appendix C. ◀

▶ **Corollary 22.** $\mathsf{FRACTRAN}_{\mathrm{reg}} \preceq \mathsf{MMA}_2$ *and* $\mathsf{FRACTRAN}_{\mathrm{reg}} \preceq \mathsf{MMA0}_2$.

4 Minsky machine termination as provability

While the (heavy) alternate Minsky machines framework was useful to simulate FRACTRAN programs with two counter machines, using it as a seed for other reductions is not recommended. First, explaining the semantics and termination predicates requires many definitions, not necessarily obvious at first. Also, manipulating them without the tools for modular reasoning is quite difficult.

4.1 Non-deterministic two counters Minsky machines (file `ndMM2.v`)

For our reductions to linear logic, we replace MMA_2 with an equivalent model, much easier to describe and work with, where computations are performed by proof-search and termination matches the provability/derivability predicate. Reductions to entailment in logical systems will thus mainly consist in encoding derivations from one system to another. We call this model non-deterministic two counters Minsky machines and denote $\mathsf{MM}_{\mathrm{nd}}$.

We comment this logical presentation, sequent style, of Minsky machines. *$\mathsf{MM}_{\mathrm{nd}}$ instructions* are of the form $\texttt{STOP}_{\mathrm{n}}\, p \mid \texttt{INC}_{\mathrm{n}}\, x\, p\, q \mid \texttt{DEC}_{\mathrm{n}}\, x\, p\, q \mid \texttt{ZERO}_{\mathrm{n}}\, x\, p\, q$ where $x \in \{\alpha, \beta\}$ is a register index, either the first α or the second β,[4] and $p, q : \mathbb{N}$ are labels, here in type $\mathbb{N}$, but the definitions in this section are completely parametric in the type of labels. A *sequent of $\mathsf{MM}_{\mathrm{nd}}$* is of the form $\Sigma /\!/_{\mathrm{n}}\, a \oplus b \vdash p$ where Σ is a list of $\mathsf{MM}_{\mathrm{nd}}$ instructions viewed as a finite set, a and b of type $\mathbb{N}$ represent the values of the counters α and β respectively and p is the current label.

We define provability/derivability inductively by the rules the calculus $\mathsf{S\text{-}MM}_{\mathrm{nd}}$ in Fig. 1. Notice that since computation is simulated by proof-search, the initial state is the conclusion of a rule and it is transformed into the premise, when there is one. For example, the $\texttt{INC}_{\mathrm{n}}\, \_\, p\, q$ rule contains both the initial label and the jump-to label, hence it can only execute at label

[4] hence formally a Boolean value of type $\mathbb{B}$.

p. However, nothing prevents the simultaneous occurrence of another instruction $\texttt{INC}_\mathrm{n}\,\_\,p\,q'$ in Σ, and this could render proof-search non-deterministic, hence our choice of terminology. However, non-determinism is not relevant to the undecidability of the MM_nd.

Notice that it is common practice to represent the sequent and the derivability predicate of the sequent by the same denotation $\Sigma \mathbin{/\!/_\mathrm{n}} a \oplus b \vdash p$ which could lead to confusion. Usually, we qualify the notation with the "sequent" word to make it explicit. Unqualified or followed with "is derivable" means that the notation represents the $\mathsf{S\text{-}MM}_\mathrm{nd}$ derivability predicate.

▶ **Definition 23.** *A MM_nd problem instance is the data of a sequent $\Sigma \mathbin{/\!/_\mathrm{n}} a \oplus b \vdash p$, and the question is whether this sequent is derivable or not using the rules of $\mathsf{S\text{-}MM}_\mathrm{nd}$ (Fig. 1).*

Notice that $\texttt{ZERO}_\mathrm{n}\,x\,p\,q$ performs both a zero-test on x and if zero, a jump from p to q without changing registers. If we remove the $\texttt{ZERO}_\mathrm{n}\,\_\,p\,q$ rules, we get Petri nets reachability, more specifically VASS with states, which have a decidable reachability problem with non-elementary complexity [4], even non-primitive recursive according to [16, 5].

4.2 From $\mathsf{MMA0}_2$ to MM_nd (file `MMA2_to_ndMM2_ACCEPT.v`)

We give an alternate presentation of termination on zero for two registers Minsky machines, using the $\mathsf{S\text{-}MM}_\mathrm{nd}$ calculus of Section 4.1. Let us consider alternate Minsky machines MMA_2 with two counters, $s := 0 : \mathbb{F}_2$ and $d := 1 : \mathbb{F}_2$. We denote by $\alpha, \beta : \mathbb{B}$ the two registers of MM_nd instructions. We define the following encodings of single instructions and programs:

$$\begin{array}{lll}
\overline{(\cdot)} : \mathbb{F}_2 \to \mathbb{B} & \langle\cdot,\cdot\rangle : \mathbb{N} \to \mathsf{MMA}_2 \to \mathbb{L}\,\mathsf{MM}_\mathrm{nd} & \langle\!\langle\cdot,\cdot\rangle\!\rangle : \mathbb{N} \to \mathbb{L}\,\mathsf{MMA}_2 \to \mathbb{L}\,\mathsf{MM}_\mathrm{nd} \\
\overline{0} := \alpha & \langle i, \texttt{INC}_\mathrm{a}\,x\rangle := [\texttt{INC}_\mathrm{n}\,\overline{x}\,i\,(1{+}i)] & \langle\!\langle i, [\,]\rangle\!\rangle := [\,] \\
\overline{1} := \beta & \langle i, \texttt{DEC}_\mathrm{a}\,x\,j\rangle := [\texttt{DEC}_\mathrm{n}\,\overline{x}\,i\,j; \texttt{ZERO}_\mathrm{n}\,\overline{x}\,i\,(1{+}i)] & \langle\!\langle i, \sigma :: P\rangle\!\rangle := \langle i,\sigma\rangle \mathbin{+\!\!+} \langle\!\langle 1{+}i, P\rangle\!\rangle
\end{array}$$

▶ **Proposition 24.** *The encodings $\langle\cdot,\cdot\rangle$ and $\langle\!\langle\cdot,\cdot\rangle\!\rangle$ are sound:*
1. *assuming the inclusion $\langle i,\sigma\rangle \subseteq \Sigma$, if $\sigma \mathbin{/\!/_\mathrm{a}} (i,[a;b]) \succ (j,[a';b'])$ and $\Sigma \mathbin{/\!/_\mathrm{n}} a' \oplus b' \vdash j$ is derivable then so is $\Sigma \mathbin{/\!/_\mathrm{n}} a \oplus b \vdash i$;*
2. *assuming $\langle\!\langle 1,P\rangle\!\rangle \subseteq \Sigma$, if $(1,P) \mathbin{/\!/_\mathrm{a}} (i,[a;b]) \succ^1 (j,[a';b'])$ and $\Sigma \mathbin{/\!/_\mathrm{n}} a' \oplus b' \vdash j$ is derivable then so is $\Sigma \mathbin{/\!/_\mathrm{n}} a \oplus b \vdash i$.*

Proof. Item 1 is by case analysis on σ and item 2 follows from item 1. ◀

Let us now define $\Sigma_P := \texttt{STOP}_\mathrm{n}\,0 :: \langle\!\langle 1,P\rangle\!\rangle$ which constitutes the encoding of MMA_2 programs into MM_nd sequents. We establish its soundness.

▶ **Lemma 25.** *If $(1,P) \mathbin{/\!/_\mathrm{a}} (i,[a,b]) \succ^* (0,[0;0])$ then $\Sigma_P \mathbin{/\!/_\mathrm{n}} a \oplus b \vdash i$ is derivable.*

Proof. We have $\langle\!\langle 1,P\rangle\!\rangle \subseteq \Sigma_P$ by definition of Σ_P. Iterating Proposition 24 (item 2), we thus get $\Sigma_P \mathbin{/\!/_\mathrm{n}} 0 \oplus 0 \vdash 0 \;\to\; \Sigma_P \mathbin{/\!/_\mathrm{n}} a \oplus b \vdash i$. The derivability of $\Sigma_P \mathbin{/\!/_\mathrm{n}} 0 \oplus 0 \vdash 0$ follows from $\texttt{STOP}_\mathrm{n}\,0 \in \Sigma_P$ and the $\texttt{STOP}_\mathrm{n}\,0$ rule of $\mathsf{S\text{-}MM}_\mathrm{nd}$. ◀

▶ **Lemma 26.** *If $\Sigma_P \mathbin{/\!/_\mathrm{n}} a \oplus b \vdash i$ is derivable then $(1,P) \mathbin{/\!/_\mathrm{a}} (i,[a,b]) \succ^* (0,[0;0])$.*

Proof. The argument proceeds by structural induction on the derivation of $\Sigma_P \mathbin{/\!/_\mathrm{n}} a \oplus b \vdash i$, i.e. by analysing the structure of $\mathsf{S\text{-}MM}_\mathrm{nd}$ derivations. The following result is an essential ingredient in this case analysis: $c \in \langle\!\langle i,P\rangle\!\rangle \;\to\; \exists L\,\sigma\,R,\; P = L \mathbin{+\!\!+} \sigma :: R \,\wedge\, c \in \langle |L| + i, \sigma\rangle$. It allows to recover the MMA_2 instructions from which MM_nd instructions originate. ◀

▶ **Corollary 27.** $\mathsf{MMA0}_2 \preceq \mathsf{MM}_\mathrm{nd}$.

Proof. The reduction maps an instance $(P,[a;b])$ of $\mathsf{MMA0}_2$ to the sequent $\Sigma_P \mathbin{/\!/_\mathrm{n}} a \oplus b \vdash 1$. Lemmas 25 and 26 provide the equivalence between $(1,P) \mathbin{/\!/_\mathrm{a}} (1,[a,b]) \rightsquigarrow (0,[0;0])$ and the derivability of $\Sigma_P \mathbin{/\!/_\mathrm{n}} a \oplus b \vdash 1$, which ensures the correctness of the reduction. ◀

5 Undecidability of Sub-Exponential Linear Logic

Having established the undecidability of $\mathsf{MM}_{\mathrm{nd}}$ via $\mathsf{FRACTRAN}_{\mathrm{reg}}$ and $\mathsf{MMA0}_2$, we can now switch to undecidability in some fragments of linear logic and give a comparison between two different reductions. We introduce the intuitionistic version of sub-exponential linear logic [2] (IMSELL) and mechanise a many-one reduction from $\mathsf{MM}_{\mathrm{nd}}$ to entailment in IMSELL. Even if the former reduction [2] applies to classical sub-exponential linear logic with one sided sequents, our own reduction function is inspired from it. However, the completeness proof that we have mechanised largely differs since we avoid focused proofs (used to recover computations) and instead, adapt the trivial phase semantics argument [13, 8]. Additionally we precisely compare the reduction to ILL with the reduction to IMSELL by starting from the same $\mathsf{MM}_{\mathrm{nd}}$ seed, detailing what set of logical rules are used to simulate those machines.

5.1 The ILL and IMSELL fragments (files `ILL.v` and `IMSELL.v`)

We introduce two fragments/extensions of intuitionistic linear logic (ILL) that allow for a reduction from non-deterministic two counters Minsky machines.

The first fragment of the ILL logic we consider is composed of propositional formulæ build from two binary connectives, the *linear implication* $\multimap$ and *additive conjunction* $\&$, and one modality, *exponentiation* $!$. Logical variables come from (a copy of) the $\mathbb{N}$ type. Formally, the formulæ of ILL are of the form $A, B ::= X \mid A \multimap B \mid A \,\&\, B \mid !A$ where $X : \mathbb{N}$. To simplify, *we abusively call this fragment ILL*. By cut-elimination, the reduction discussed below also works for larger fragments containing more connectives like $\otimes$, $\oplus$, etc.

The sequents of ILL are intuitionistic, i.e. a pair (Γ, A) written $\Gamma \vdash A$ where Γ is a multiset of formulæ and A is a single formula. Multisets are just lists identified up-to permutations. If it is more convenient to work with lists, as we do in the Coq mechanization, then an *explicit permutation rule* is added to the sequent rules of the S-ILL calculus in Fig. 2.

The three leftmost rules are the identity (or axiom) rule stating that the sequent $A \vdash A$ has a trivial proof, and then the left- and right-introduction rules for the linear implication $\multimap$. The three rules middle-left are two left- and one right-introduction rules for the additive conjunction $\&$. The two middle-right rules are modal rules, on top, the *promotion* rule, and at bottom, the *dereliction* rule. Finally, on the right-hand-side are the *structural rules* for the $!$ modality, i.e. *weakening* on top and *contraction* at the bottom. Notice that specifically, linear logic does not allow for general weakening or contraction rules.

On the other hand, IMSELL is a purely multiplicative fragment but with several modalities, among them exponentials. The logic is parameterized with a fixed type Λ of *modalities* and a fixed sub-type $\mathcal{U} : \Lambda \to \mathbb{P}$ of *unbounded* modalities, also called exponentials. We follow the set theoretic syntax and write $u \in \mathcal{U}$ (instead of $\mathcal{U}\,u$) when u is unbounded. The formulæ of IMSELL_Λ are of the form $A, B ::= X \mid A \multimap B \mid !^m A$ where $X : \mathbb{N}$ and $m : \Lambda$. So compared to ILL, the additive $\&$ is missing whereas the modality $!$ becomes indexed as $!^m$ with m spanning over Λ. IMSELL_Λ sequents have the same structure $\Gamma \vdash A$ as those of ILL except that they are composed of IMSELL_Λ formulæ instead.

Before we describe the associated sequent calculus $\mathsf{S\text{-}IMSELL}_\Lambda$, we introduce supplementary structures on modalities: a pre-order $\preccurlyeq\, : \Lambda \to \Lambda \to \mathbb{P}$, i.e. a *reflexive* and *transitive* binary relation, such that $\mathcal{U}$ *is upward-closed for* $\preccurlyeq$, i.e. $u \preccurlyeq m$ and $u \in \mathcal{U}$ entail $m \in \mathcal{U}$ for any $m, u : \Lambda$. In the sequel, we will somehow abuse the notation and denote Λ both for the base type and the modal structure $(\Lambda, \mathcal{U}, \preccurlyeq)$ moreover assuming the pre-order and upward-closure properties.

$$\frac{}{A \vdash A} \quad \frac{A, \Gamma \vdash B}{\Gamma \vdash A \multimap B} \quad \frac{\Gamma \vdash A \quad B, \Delta \vdash C}{A \multimap B, \Gamma, \Delta \vdash C} \quad \Bigg| \quad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B} \quad \frac{A, \Gamma \vdash C}{A \,\&\, B, \Gamma \vdash C} \quad \frac{B, \Gamma \vdash C}{A \,\&\, B, \Gamma \vdash C} \quad \Bigg| \quad \frac{!\Gamma \vdash B}{!\Gamma \vdash !B} \quad \frac{A, \Gamma \vdash B}{!A, \Gamma \vdash B} \quad \Bigg| \quad \frac{\Gamma \vdash B}{!A, \Gamma \vdash B} \quad \frac{!A, !A, \Gamma \vdash B}{!A, \Gamma \vdash B}$$

Figure 2 The S-ILL sequent calculus.

$$\frac{}{A \vdash A} \quad \frac{A, \Gamma \vdash B}{\Gamma \vdash A \multimap B} \quad \frac{\Gamma \vdash A \quad B, \Delta \vdash C}{A \multimap B, \Gamma, \Delta \vdash C} \quad \Bigg\| \quad \frac{!^\star \Gamma \vdash B}{!^\star \Gamma \vdash !^m B}\ m \preccurlyeq \star \quad \frac{A, \Gamma \vdash B}{!^m A, \Gamma \vdash B} \quad \Bigg| \quad \frac{\Gamma \vdash B}{!^u A, \Gamma \vdash B}\ u \in \mathcal{U} \quad \frac{!^u A, !^u A, \Gamma \vdash B}{!^u A, \Gamma \vdash B}\ u \in \mathcal{U}$$

Figure 3 The S-IMSELL$_\Lambda$ sequent calculus with $(\Lambda, \mathcal{U}, \preccurlyeq)$.

In the sequent rules of the S-IMSELL$_\Lambda$ calculus of Fig. 3, the three leftmost rules are common with S-ILL, there is no rule for the additive conjunction & since it does belong to the fragment, and the modal rules have changed a bit. We skip over the two middle rules for the moment and consider the rightmost structural rules of weakening and contraction which generalise the corresponding rules of S-ILL, except that their use is limited to unbounded modalities ($u \in \mathcal{U}$). Back to the two middle rules, the bottom dereliction rule applies to every modality, so a direct generalisation of the corresponding rule of S-ILL. However, the promotion rule (reproduced below on the left)

$$\frac{!^\star \Gamma \vdash B}{!^\star \Gamma \vdash !^m B}\ m \preccurlyeq \star \quad \Bigg| \quad \frac{!^{k_1} A_1, \ldots, !^{k_n} A_n \vdash B}{!^{k_1} A_1, \ldots, !^{k_n} A_n \vdash !^m B}\ m \preccurlyeq k_1, \ldots, m \preccurlyeq k_n \quad \Bigg| \quad \frac{!^m \Gamma \vdash B}{!^m \Gamma \vdash !^m B}$$

is somehow more complicated and deserves further explanations. The $\star$ notation represents a multiset $k_1, \ldots, k_n$ of modalities and $!^\star \Gamma$ represents the multiset $!^{k_1} A_1, \ldots, !^{k_n} A_n$. The constraint $m \preccurlyeq \star$ imposes that m is lower than every modality in $\{k_1, \ldots, k_n\}$. Using these more explicit notations, we reframe it as in the above displayed middle rule. Finally, the (uniform) instance where $m = k_1 = \cdots = k_n$ (the rightmost above; the constraint $m \preccurlyeq \star$ holds by reflexivity), matches the promotion rule of S-ILL.

Considered independently, all modalities behave like ILL modalities, satisfying dereliction and promotion rules, while only unbounded modalities allow for contraction and weakening. However, depending on the relation $\preccurlyeq$, the promotion rule allows for *non-trivial interactions* between modalities. Given an unbounded modality $\infty \in \mathcal{U}$ and replacing ! with $!^\infty$, one can trivially embed the multiplicative fragment of ILL and recover intuitionistic multiplicative and exponential linear logic (IMELL), of which the (un)decidability of entailment is a notoriously difficult open problem [14, 20].

5.2 Embedding in S-ILL vs. S-IMSELL (file `ndMM2_IMSELL.v`)

For the reduction from MM$_{\text{nd}}$ to IMSELL$_\Lambda$ to work out properly, we need at least three modality $\{a, b, \infty\}$ where ∞ is the only unbounded modality ($\infty \in \mathcal{U}$ and $a, b \notin \mathcal{U}$), $a \preccurlyeq \infty$, $b \preccurlyeq \infty$ and a and b are incomparable, i.e. $a \not\preccurlyeq b$ and $b \not\preccurlyeq a$. As a consequence, ∞ is also strictly above a and b. From now on, we assume that Λ satisfies these requirements. The coming discussion can also be understood in the minimal case where $\Lambda_3 = \{a, b, \infty\}$ and we denote IMSELL$_3$ for either of these logics.

In this section, we review the encoding of $\mathsf{MM}_{\mathrm{nd}}$ sequents into both ILL and IMSELL_3, and explain how, while mostly similar, *they noticeably differ on how they handle zero tests combined with jumps*. Notice that the encoding targeting ILL can be adapted to n registers Minsky machines (as done in [8]), while in the case of IMSELL_3, working with two counters only is critically important to the construction.

Identifying the exponential ! with the unbounded modality $!^\infty$ allows to discuss IMELL, ILL and IMSELL_3 in a common syntactic framework, avoiding cumbersome notations for trivial embeddings. We show the derivability of the two following rules: generalised weakening and customised absorption.

▶ **Lemma 28.** *The two following rules are derivable in* IMELL*, and hence* ILL *and* IMSELL_3*:*

$$\frac{\Delta \vdash B}{!^\infty\Sigma, \Delta \vdash B} \qquad\qquad \frac{A, !^\infty\Sigma, \Delta \vdash B}{!^\infty\Sigma, \Delta \vdash B}\ A \in \Sigma$$

Proof. We obtain the left generalised weakening rule by repeating the weakening rule. For customised absorption, it is the combination of dereliction and contraction. ◀

These derived rules are essential tools for the reduction from $\mathsf{MM}_{\mathrm{nd}}$. Let us review the other tools. Recall that a $\mathsf{MM}_{\mathrm{nd}}$ sequent is of the form $\Sigma \mathbin{/\!\!/_{\mathrm{n}}} x \oplus y \vdash p$. We encode this with an IMSELL_3 (or ILL) sequent of the form $!^\infty\overline{\Sigma}, \Delta \vdash \overline{p}$ where $\Delta := x\overline{\alpha}, y\overline{\beta}$ encodes the pair $(x, y) : \mathbb{N} \times \mathbb{N}$, i.e. $\overline{\alpha}$ (resp. $\overline{\beta}$) is repeated x (resp. y) times. Hence increment and decrement operations on the values x/y naturally correspond to the multiset operations. We do not need to specify what formulæ are $\overline{\alpha}$ and $\overline{\beta}$ for the moment, but these will differ in the ILL case compared to the IMSELL_3 case. On the other hand, $\overline{p}$ or $\overline{q}$ will always be logical variables.

First, we show how to simulate the $\mathtt{INC}_{\mathrm{n}}\,\alpha\,p\,q$ rule of $\mathsf{S\text{-}MM}_{\mathrm{nd}}$:

$$\frac{\Sigma \mathbin{/\!\!/_{\mathrm{n}}} 1{+}x \oplus y \vdash q}{\Sigma \mathbin{/\!\!/_{\mathrm{n}}} x \oplus y \vdash p}\ \mathtt{INC}_{\mathrm{n}}\,\alpha\,p\,q \in \Sigma \quad\rightsquigarrow\quad \dfrac{\dfrac{\dfrac{!^\infty\overline{\Sigma}, \overline{\alpha}, \Delta \vdash \overline{q}}{!^\infty\overline{\Sigma}, \Delta \vdash \overline{\alpha} \multimap \overline{q}} \quad \dfrac{}{\overline{p} \vdash \overline{p}}}{(\overline{\alpha} \multimap \overline{q}) \multimap \overline{p}, !^\infty\overline{\Sigma}, \Delta \vdash \overline{p}}}{!^\infty\overline{\Sigma}, \Delta \vdash \overline{p}}\ (\overline{\alpha} \multimap \overline{q}) \multimap \overline{p} \in \overline{\Sigma}$$

and the $\mathtt{DEC}_{\mathrm{n}}\,\alpha\,p\,q$ rule of $\mathsf{S\text{-}MM}_{\mathrm{nd}}$:

$$\frac{\Sigma \mathbin{/\!\!/_{\mathrm{n}}} x \oplus y \vdash q}{\Sigma \mathbin{/\!\!/_{\mathrm{n}}} 1{+}x \oplus y \vdash p}\ \mathtt{DEC}_{\mathrm{n}}\,\alpha\,p\,q \in \Sigma \quad\rightsquigarrow\quad \dfrac{\dfrac{\dfrac{}{\overline{\alpha} \vdash \overline{\alpha}} \quad \dfrac{!^\infty\overline{\Sigma}, \Delta \vdash \overline{q} \quad \dfrac{}{\overline{p} \vdash \overline{p}}}{\overline{q} \multimap \overline{p}, !^\infty\overline{\Sigma}, \Delta \vdash \overline{p}}}{\overline{\alpha} \multimap (\overline{q} \multimap \overline{p}), !^\infty\overline{\Sigma}, \overline{\alpha}, \Delta \vdash \overline{p}}}{!^\infty\overline{\Sigma}, \overline{\alpha}, \Delta \vdash \overline{p}}\ \overline{\alpha} \multimap (\overline{q} \multimap \overline{p}) \in \overline{\Sigma}$$

Notice that we only use the customised absorption rule, the left- and right-introduction rules for $\multimap$ and the identity (axiom) rule hence simulating $\mathtt{INC}_{\mathrm{n}}\,\alpha\,p\,q$ and $\mathtt{DEC}_{\mathrm{n}}\,\alpha\,p\,q$ can be performed within the IMELL fragment.

The axiom rule $\mathtt{STOP}_{\mathrm{n}}\,p$ of $\mathsf{S\text{-}MM}_{\mathrm{nd}}$ (acceptance of $(0,0)$ at p) can also be simulated

$$\frac{}{\Sigma \mathbin{/\!\!/_{\mathrm{n}}} 0 \oplus 0 \vdash p}\ \mathtt{STOP}_{\mathrm{n}}\,p \in \Sigma \quad\rightsquigarrow\quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\overline{p} \vdash \overline{p}}}{\vdash \overline{p} \multimap \overline{p}} \quad \dfrac{}{\overline{p} \vdash \overline{p}}}{(\overline{p} \multimap \overline{p}) \multimap \overline{p} \vdash \overline{p}}}{(\overline{p} \multimap \overline{p}) \multimap \overline{p}, !^\infty\overline{\Sigma} \vdash \overline{p}}}{!^\infty\overline{\Sigma}, \emptyset \vdash \overline{p}}\ (\overline{p} \multimap \overline{p}) \multimap \overline{p} \in \overline{\Sigma}$$

using the customised absorption rule, then the generalised weakening rule, the left- and right-introduction rules for $\multimap$ and the identity rule. Hence an IMELL proof as well.

The remaining rules of $\mathsf{MM}_{\mathrm{nd}}$, that of e.g. $\mathtt{ZERO}_{\mathrm{n}}\,\alpha\,p\,q$, a zero test *combined with* a jump instruction, are the problematic rules to encode in the IMELL fragment. This can however be done in ILL and in IMSELL_3, but the techniques for the two fragments diverge precisely on these $\mathtt{ZERO}_{\mathrm{n}}\,\alpha\,p\,q$ instructions.

Let us first review[5] the (idea behind the) legacy encoding of Minsky machines into linear logic [17], mechanized for ILL in [8]. The idea is to *fork* the $\mathtt{ZERO}_{\mathrm{n}}\,\alpha\,p\,q$ simulation into a proof-search branch where only a zero test on α is performed, and in the other branch, only a jump to q is performed:

$$\dfrac{\Sigma /\!/_{\mathrm{n}}\, 0 \oplus y \vdash q}{\Sigma /\!/_{\mathrm{n}}\, 0 \oplus y \vdash p}\ \mathtt{ZERO}_{\mathrm{n}}\,\alpha\,p\,q \in \Sigma \qquad\rightsquigarrow\qquad \dfrac{\dfrac{\dfrac{\dfrac{\cdots}{!^\infty\overline{\Sigma}, y\overline{\beta} \vdash \underline{\alpha}} \quad !^\infty\overline{\Sigma}, y\overline{\beta} \vdash \overline{q}}{!^\infty\overline{\Sigma}, y\overline{\beta} \vdash \underline{\alpha} \,\&\, \overline{q}} \qquad \dfrac{}{\overline{p} \vdash \overline{p}}}{(\underline{\alpha} \,\&\, \overline{q}) \multimap \overline{p}, !^\infty\overline{\Sigma}, y\overline{\beta} \vdash \overline{p}}}{!^\infty\overline{\Sigma}, y\overline{\beta} \vdash \overline{p}}\ (\underline{\alpha} \,\&\, \overline{q}) \multimap \overline{p} \in \overline{\Sigma}$$

Notice that $\underline{\alpha}$ and $\underline{\beta}$ denote fresh logical variables. Critically for this encoding, the additive conjunction & is used to copy the context into the two premises, implementing a fork. The zero test on left sub-branch can however by performed in IMELL only:

$$\dfrac{\dfrac{}{\overline{\beta} \vdash \overline{\beta}} \quad \dfrac{\dfrac{\dfrac{!^\infty\overline{\Sigma}, \emptyset \vdash \underline{\alpha}}{\cdots}}{!^\infty\overline{\Sigma}, y\overline{\beta} \vdash \underline{\alpha}} \quad \dfrac{}{\underline{\alpha} \vdash \underline{\alpha}}}{\underline{\alpha} \multimap \underline{\alpha}, !^\infty\overline{\Sigma}, y\overline{\beta} \vdash \underline{\alpha}}}{\dfrac{\overline{\beta} \multimap (\underline{\alpha} \multimap \underline{\alpha}), !^\infty\overline{\Sigma}, \overline{\beta}, y\overline{\beta} \vdash \underline{\alpha}}{!^\infty\overline{\Sigma}, (1{+}y)\overline{\beta} \vdash \underline{\alpha}}\ \overline{\beta} \multimap (\underline{\alpha} \multimap \underline{\alpha}) \in \overline{\Sigma}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\underline{\alpha} \vdash \underline{\alpha}}}{\vdash \underline{\alpha} \multimap \underline{\alpha}} \quad \dfrac{}{\underline{\alpha} \vdash \underline{\alpha}}}{(\underline{\alpha} \multimap \underline{\alpha}) \multimap \underline{\alpha}, \emptyset \vdash \underline{\alpha}}}{(\underline{\alpha} \multimap \underline{\alpha}) \multimap \underline{\alpha}, !^\infty\overline{\Sigma}, \emptyset \vdash \underline{\alpha}}}{!^\infty\overline{\Sigma}, \emptyset \vdash \underline{\alpha}}\ (\underline{\alpha} \multimap \underline{\alpha}) \multimap \underline{\alpha} \in \overline{\Sigma}}{\cdots}$$

Notice that the dots above $!^\infty\overline{\Sigma}, y\overline{\beta} \vdash \underline{\alpha}$ mean repetition of the lower part of the proof until exhaustion of all the $\overline{\beta}$ from the context: this is implemented by an induction on y, and the base case where $y = 0$ corresponds to the upper part of the proof, starting at $!^\infty\overline{\Sigma}, \emptyset \vdash \underline{\alpha}$ and completed on the right hand side, simulating of a would be $\mathtt{STOP}_{\mathrm{n}}\,\underline{\alpha}$ instruction (see above).

We see that $\underline{\alpha}$ together with the formulæ $\overline{\beta} \multimap (\underline{\alpha} \multimap \underline{\alpha})$ and $(\underline{\alpha} \multimap \underline{\alpha}) \multimap \underline{\alpha}$ in $\overline{\Sigma}$ allow $\underline{\alpha}$ to perform the elimination of all the $\overline{\beta}$ from the context. However, $\underline{\alpha}$ will not allow the removal of any $\overline{\alpha}$ and hence, the zero test branch cannot be completed if Δ contains an occurrence of $\overline{\alpha}$, i.e. when $x \neq 0$. This encoding of the zero test using $\underline{\alpha}$, while it can already be performed in IMELL, is pertinent only for ILL because it is in combination with the fork in $(\underline{\alpha} \,\&\, \overline{q}) \multimap \overline{p}$ (see above) that it provides the ability to *conditionally jump on zero.*

Contrary to the ILL encoding, IMSELL_3 does not require (and cannot use) forking but instead uses sub-modalities to prevent jumping when the zero test fails. In that case, $\overline{\alpha}$ and $\overline{\beta}$ are not atomic formulæ anymore: they contain the bounded modalities $!^a$ and $!^b$, and we define $\overline{\alpha} := !^a\alpha_0$ and $\overline{\beta} := !^b\beta_0$ where α_0, β_0 are fresh variables. In the following encoding,

[5] here we only discuss the ILL case, i.e. we do not replicate the former ILL mechanisation [8] in the code.

$$\dfrac{\Sigma \mathbin{/\!/_{\mathrm{n}}} 0 \oplus y \vdash q}{\Sigma \mathbin{/\!/_{\mathrm{n}}} 0 \oplus y \vdash p}\ \mathtt{ZERO}_{\mathrm{n}}\,\alpha\,p\,q \in \Sigma \qquad \rightsquigarrow \qquad \dfrac{\dfrac{\dfrac{!^{\infty}\overline{\Sigma}, y\overline{\beta} \vdash \overline{q}}{!^{\infty}\overline{\Sigma}, y\overline{\beta} \vdash !^{b}\overline{q}} \quad \dfrac{}{\overline{p} \vdash \overline{p}}}{!^{b}\overline{q} \multimap \overline{p}, !^{\infty}\overline{\Sigma}, y\overline{\beta} \vdash \overline{p}}}{!^{\infty}\overline{\Sigma}, y\overline{\beta} \vdash \overline{p}}\ !^{b}\overline{q} \multimap \overline{p} \in \overline{\Sigma}$$

notice that the upper rule is an instance of the promotion rule of S-IMSELL$_3$. It is allowed because every formula on the left is prefixed either with the unbounded modality $!^{\infty}$ for those in $!^{\infty}\overline{\Sigma}$, or with the modality $!^{b}$ for those in $y\overline{\beta} = !^{b}\beta_0, \dots, !^{b}\beta_0$, and we have both $b \preccurlyeq \infty$ and $b \preccurlyeq b$. On the other hand, an occurrence of $\overline{\alpha} = !^{a}\alpha_0$ in the context, corresponding to a non-zero value of x, would prevent the application of the promotion rule ($b \not\preccurlyeq a$). This interaction of modalities in the promotion rule of IMSELL$_3$ is the key to simulate zero tests.

▶ **Definition 29.** *Let us define $\alpha_0 := 0$, $\beta_0 := 1$, $\overline{p} := 2 + p$, $\overline{\alpha} := !^{a}\alpha_0$ and $\overline{\beta} := !^{b}\beta_0$. We encode MM$_{\mathrm{nd}}$ instructions as:*

$$\overline{\mathtt{STOP}_{\mathrm{n}}\,p} := (\overline{p} \multimap \overline{p}) \multimap \overline{p}$$

$$\begin{array}{lll} \overline{\mathtt{INC}_{\mathrm{n}}\,\alpha\,p\,q} := (\overline{\alpha} \multimap \overline{q}) \multimap \overline{p} & \overline{\mathtt{DEC}_{\mathrm{n}}\,\alpha\,p\,q} := \overline{\alpha} \multimap (\overline{q} \multimap \overline{p}) & \overline{\mathtt{ZERO}_{\mathrm{n}}\,\alpha\,p\,q} := !^{b}\overline{q} \multimap \overline{p} \\ \overline{\mathtt{INC}_{\mathrm{n}}\,\beta\,p\,q} := (\overline{\beta} \multimap \overline{q}) \multimap \overline{p} & \overline{\mathtt{DEC}_{\mathrm{n}}\,\beta\,p\,q} := \overline{\beta} \multimap (\overline{q} \multimap \overline{p}) & \overline{\mathtt{ZERO}_{\mathrm{n}}\,\beta\,p\,q} := !^{a}\overline{q} \multimap \overline{p} \end{array}$$

and then map $\overline{(\cdot)}$ on the list Σ extensionally, i.e. $\overline{[\sigma_1; \dots; \sigma_n]} := \overline{\sigma_1}, \dots, \overline{\sigma_n}$.

▶ **Lemma 30.** *If $\Sigma \mathbin{/\!/_{\mathrm{n}}} x \oplus y \vdash p$ can be derived in S-MM$_{\mathrm{nd}}$ then the sequent $!^{\infty}\overline{\Sigma}, x\overline{\alpha}, y\overline{\beta} \vdash \overline{p}$ is provable in S-IMSELL$_3$.*

Proof. The argument proceeds by induction on the derivation of $\Sigma \mathbin{/\!/_{\mathrm{n}}} x \oplus y \vdash p$, combining the proof skeletons of the above discussion in a direct way. ◀

5.3 Trivial Phase semantics for IMSELL (file `imsell.v`)

We define trivial phase semantics for IMSELL$_{\Lambda}$ and show soundness w.r.t. the S-IMSELL$_{\Lambda}$ calculus. We start with a commutative monoid $(M, \bullet, \epsilon)$. Typically, for the completeness of our reduction, we will only need to use the semantics for $M = (\mathbb{N}^2, +, \vec{0})$, i.e. vectors of natural numbers of length 2, but the semantics works for any commutative monoid. For any $X, Y \subseteq M$, we define the *point-wise extension* by $X \bullet Y := \{x \bullet y \mid x \in X \wedge y \in Y\}$ and its *linear adjunct* as $X \multimap\!\!\bullet\, Y := \{k \in M \mid \{k\} \bullet X \subseteq Y\}$ providing a residuated monoidal structure on the subset type $M \to \mathbb{P}$.

To interpret the modal structure $(\Lambda, \mathcal{U}, \preccurlyeq)$, we further require for each modality $m \in \Lambda$, a subset $K_m \subseteq M$ i.e. a predicate $K_m : M \to \mathbb{P}$. We assume that the map $m \mapsto K_m$ is monotonically decreasing w.r.t. $\preccurlyeq$ (on the left below) and satisfies the three extra following rightmost axioms:

$$\forall m\,k,\ m \preccurlyeq k \to K_k \subseteq K_m \qquad \forall m,\ \epsilon \in K_m \qquad \forall m,\ K_m \bullet K_m \subseteq K_m \qquad \forall u \in \mathcal{U},\ K_u \subseteq \{\epsilon\}$$

Given any semantic interpretation $[\![\cdot]\!] \subseteq M$ of logical variables, we extend it inductively to IMSELL$_{\Lambda}$ sequents via *trivial phase semantics*:[6]

$$[\![A \multimap B]\!] := [\![A]\!] \multimap\!\!\bullet\, [\![B]\!] \qquad [\![!^{m}A]\!] := [\![A]\!] \cap K_m \qquad [\![A_1, \dots, A_n]\!] := [\![A_1]\!] \bullet \cdots \bullet [\![A_n]\!]$$

[6] The *trivial* qualifier refers to the use of the identity closure $\mathrm{cl}(X) = X$ in the interpretation of modalities, i.e. $[\![!^{m}A]\!] := [\![A]\!] \cap K_m$ instead of the more general $[\![!^{m}A]\!] := \mathrm{cl}([\![A]\!] \cap K_m)$ where $\mathrm{cl}(\cdot) : (M \to \mathbb{P}) \to (M \to \mathbb{P})$ is a stable closure operator. This also applies to the (implicit) multiplicative conjunction where $[\![A_1, \dots, A_n]\!] := [\![A_1]\!] \bullet \cdots \bullet [\![A_n]\!]$ instead of $[\![A_1, \dots, A_n]\!] := \mathrm{cl}([\![A_1]\!] \bullet \cdots \bullet [\![A_n]\!])$. Notice that trivial phase semantics is sound but *not complete* for IMELL, ILL and IMSELL$_{\Lambda}$; see [13] for details.

Notice that because we work with commutative monoids, the above semantic interpretation of lists is invariant under permutations, hence is suitable for multisets. An IMSELL_Λ sequent $\Gamma \vdash A$ is *valid in that interpretation* if $[\![\Gamma]\!] \subseteq [\![A]\!]$, or (equivalently) if $\epsilon \in [\![\Gamma]\!] \multimap\!\bullet\, [\![A]\!]$.

▶ **Theorem 31.** *Trivial phase semantics is sound: any sequent $\Gamma \vdash A$ provable in S-IMSELL_Λ must satisfy $\epsilon \in [\![\Gamma]\!] \multimap\!\bullet\, [\![A]\!]$ for any possible trivial phase semantics interpretation.*

Proof. We proceed by structural induction on the S-IMSELL_Λ derivation of $\Gamma \vdash A$. In the code, the proof is limited to the case where $M = (\mathbb{N}^n, +, \vec{0})$ for some $n : \mathbb{N}$. Compared to the soundness of trivial phase semantics for S-ILL [8], the only interesting new case is that of the promotion rule. In that case, we observe that $m \preccurlyeq k_1, \ldots, k_n$ implies $K_{k_1} \bullet \cdots \bullet K_{k_n} \subseteq K_m$. ◀

5.4 The completeness of the reduction (file `ndMM2_IMSELL.v`)

▶ **Lemma 32.** *If the sequent $!^\infty\overline{\Sigma}, x\overline{\alpha}, y\overline{\beta} \vdash \overline{p}$ is provable in S-IMSELL_3, then there is a derivation of $\Sigma /\!\!/_{\mathrm{n}}\, x \oplus y \vdash p$ in S-$\mathsf{MM}_{\mathrm{nd}}$.*

Proof. We use a soundness argument for trivial phase semantics in place of reasoning by induction on focused derivation in MSELL as done in [2]. We consider the monoid of vectors $M = (\mathbb{N}^2, +, [0;0])$ of length 2 of natural numbers. We define the following interpretation for modalities, $K_m\,[x;y] := (a \preccurlyeq m \to y = 0) \wedge (b \preccurlyeq m \to x = 0) \wedge (m \in \mathcal{U} \to x = 0 \wedge y = 0)$, and as a consequence, we can check that K_m satisfies the required axioms as well as $K_a = \big\{[x;0] \mid x \in \mathbb{N}\big\}$, $K_b = \big\{[0;y] \mid y \in \mathbb{N}\big\}$, and $K_\infty = \big\{[0;0]\big\}$. We interpret logical variables as:

$$[\![\alpha_0]\!] := \big\{[1;0]\big\} \quad \text{and} \quad [\![\beta_0]\!] := \big\{[0;1]\big\} \quad \text{and} \quad [\![\overline{p}]\!] = \big\{[x;y] \mid \Sigma /\!\!/_{\mathrm{n}}\, x \oplus y \vdash p\big\}$$

and thus we have $[\![\overline{\alpha}]\!] = [\![!^a\alpha_0]\!] = [\![\alpha_0]\!] \cap K_a = \big\{[1;0]\big\}$ and $[\![\overline{\beta}]\!] = \big\{[0;1]\big\}$. Consequently, we get $[\![x\overline{\alpha}, y\overline{\beta}]\!] = \big\{[x;y]\big\}$.

We verify that the interpretation of the IMSELL_3 encoding $\overline{\sigma}$ of $\mathsf{MM}_{\mathrm{nd}}$ instructions in Σ contains the zero vector, i.e. $\forall \sigma,\, \sigma \in \Sigma \to [0;0] \in [\![\overline{\sigma}]\!]$. For instance, let us consider the case $\sigma = \mathtt{ZERO}_{\mathrm{n}}\, \alpha\, p\, q$. Then $\overline{\sigma} = !^b\overline{q} \multimap \overline{p}$ is interpreted as $([\![\overline{q}]\!] \cap K_b) \multimap\!\bullet\, [\![\overline{p}]\!]$. Hence $[0;0] \in [\![\overline{\sigma}]\!] \leftrightarrow [\![\overline{q}]\!] \cap K_b \subseteq [\![\overline{p}]\!]$, i.e. for any $[x;y] : \mathbb{N}^2$, if $\Sigma /\!\!/_{\mathrm{n}}\, x \oplus y \vdash q$ and $x = 0$ then $\Sigma /\!\!/_{\mathrm{n}}\, x \oplus y \vdash p$ which is precisely the instance of rule $\mathtt{ZERO}_{\mathrm{n}}\, \alpha\, p\, q \in \Sigma$ of S-$\mathsf{MM}_{\mathrm{nd}}$.

From the previous observation, we deduce $[0;0] \in [\![!^\infty\Sigma]\!]$. Now let us consider a sequent $!^\infty\overline{\Sigma}, x\overline{\alpha}, y\overline{\beta} \vdash \overline{p}$ which is provable in S-IMSELL_3. By the soundness Theorem 31, we know that $[0;0] \in [\![!^\infty\Sigma, x\overline{\alpha}, y\overline{\beta}]\!] \multimap\!\bullet\, [\![\overline{p}]\!]$. Since $[x;y] = [0;0] + [x;y] \in [\![!^\infty\Sigma]\!] \bullet [\![x\overline{\alpha}, y\overline{\beta}]\!] = [\![!^\infty\Sigma, x\overline{\alpha}, y\overline{\beta}]\!]$, by the definition of $\multimap\!\bullet$ we deduce $[x;y] = [0;0] + [x;y] \in [\![\overline{p}]\!]$, and hence we conclude that $\Sigma /\!\!/_{\mathrm{n}}\, x \oplus y \vdash p$ holds. ◀

▶ **Theorem 33.** *Let $(\Lambda, \mathcal{U}, \preccurlyeq)$ contain three modalities a, b and ∞ such that $\infty \in \mathcal{U}$, $a, b \notin \mathcal{U}$, $a, b \preccurlyeq \infty$, $a \not\preccurlyeq b$ and $b \not\preccurlyeq a$. Then we have a reduction $\mathsf{MM}_{\mathrm{nd}} \preceq \mathsf{IMSELL}_\Lambda$, hence derivability in the S-IMSELL_Λ calculus is undecidable.*

6 Related works and Implementation remarks

While Theorem 33 gives us a mechanised synthetic proof of the undecidability of IMSELL_3, neither its statement nor the arguments deployed directly provide hints towards a solution to the question of the decidability IMELL/MELL. As in the original pen and paper proof [2], the two bounded modalities $!^a$ and $!^b$, and their interaction with the unbounded modality $!^\infty$ in the promotion rule, play an essential role in the simulation of conditional jumps of two counters Minsky machines. While the zero test can be implemented in MELL only, the provided implementation consumes its context and thus cannot conditionally branch at the same time, hence the fork used in the case of ILL [8].

However Theorem 33 does give indications that certain decidability arguments for MELL are bound to fail, e.g. those that would also apply to MSELL in general, or IMSELL_3 in particular. It is our understanding that the refutation [20] of the faulty proof attempt for the decidability of MELL [1] partly proceeds in showing how the claimed "proof" technique would easily generalise to MSELL. In the same vein, the lower bounds on the complexity of a would be decision procedure for MELL [14], and more recently the reachability problem for Petri nets themselves [4], indicate that a decision procedure for MELL must be of non-elementary complexity. The most recent investigations [16, 5] might very well confirm that this problem is Ackermann complete and hence not primitive recursive.

Considering formalisation issues, the growing Coq library of undecidability proofs [9] was of course of great help to this work. Indeed, at the time we decided to try to implement the undecidability of MSELL, the framework for certified programming with Minsky machines was already part of the library [8]. Hence, to get two counters Minsky machines, i.e. the seed of undecidability of the pen and paper proof [2], only a modest step from many counters machines was necessary and this was even alleviated by the results on the FRACTRAN language [12], factoring out the Gödel coding phase. In fact, we contributed the seed of two counters machines much ahead of the MSELL result, and in the meantime, this seed was used to establish to *undecidability uniform boundedness* for simple stack machines and then of the problem of *semi-unification* [6]. This illustrates a critical aspect of this undecidability framework: its extensive range of seed problems for plugging into it.

Indeed, there is an important issue to consider when proving undecidability by many-one reduction, by far the most used method in the field: even mechanised, your proof is only as strong as *the implementation* of your seed problem. Typical problems can exhibit subtleties that show up at the mechanisation level: for instance Turing machines are built on tapes, a potentially infinite structure of which it could be easy to corrupt the implementation. Choosing a seed already linked to the many-one equivalence class containing easy to describe problems such as e.g. the Post correspondence problem or FRACTRAN gives much more confidence that starting from an isolated seed, still to be mechanically checked undecidable.

Another aspect which is mostly overlooked in pen and paper proofs is the computability of the reduction function. The reason is that programming with low-level Turing complete models of computation is hard and painful, with encodings at every corner. To get a glimpse of the difficulty, think of a Turing machine working with logical formulas: because it only manipulates text written on tapes, it has to implement a syntax analyser, moreover proved correct. And only then can it start its real work. The general shortcut used in pen and paper proofs to avoid this kind of description is to speak about "algorithms" that manipulate high-level data-structures and rely on an informal and consensual understanding of what these are, hand-waving away the implementation issues completely.

In this regard, the synthetic computability framework allows, at the price of relying on the computability of Coq functions – e.g. by avoiding axioms, – to formally describe the reduction functions in a language strict enough to ensures their computability, but at the same time powerful enough to largely avoid complex encodings and hence get more natural correctness proofs following or inspired from pen and paper ones. Using a constructive framework like e.g. Coq or Agda is essential in that approach, because in classical frameworks, there is no direct way to automatically ensure the general computability of the defined (reduction) functions.

References

1 Katalin Bimbó. The decidability of the intensional fragment of classical linear logic. *Theoret. Comput. Sci.*, 597:1–17, 2015. `doi:10.1016/j.tcs.2015.06.019`.

2 Kaustuv Chaudhuri. Expressing additives using multiplicatives and subexponentials. *Math. Structures Comput. Sci.*, 28(5):651–666, 2018. `doi:10.1017/S0960129516000293`.

3 John H. Conway. *FRACTRAN: A Simple Universal Programming Language for Arithmetic*, pages 4–26. Springer New York, New York, NY, 1987.

4 Wojciech Czerwiński, Sławomir Lasota, Ranko Lazić, Jérôme Leroux, and Filip Mazowiecki. The reachability problem for Petri nets is not elementary. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2019, page 24–33, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3313276.3316369`.

5 Wojciech Czerwiński and Łukasz Orlikowski. Reachability in Vector Addition Systems is Ackermann-complete, 2021. `arXiv:2104.13866`.

6 Andrej Dudenhefner. Undecidability of Semi-Unification on a Napkin. In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, volume 167 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:16, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.FSCD.2020.9`.

7 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`.

8 Yannick Forster and Dominique Larchey-Wendling. Certified undecidability of intuitionistic linear logic via binary stack machines and Minsky machines. 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 104–117. ACM, 2019. `doi:10.1145/3293880.3294096`.

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 *CoqPL 2020*, New Orleans, LA, United States, 2020. URL: `https://github.com/uds-psl/coq-library-undecidability`.

10 Max Kanovich. Linear Logic as a Logic of Computations. *Ann. Pure Appl. Logic*, 67(1–3):183–212, 1994.

11 Max Kanovich. The direct simulation of Minsky machines in linear logic. In Jean-Yves Girard, Yves Lafont, and Laurent Regnier, editors, *Advances in Linear Logic*, volume 222 of *London Mathematical Society Lecture Note Series*, chapter 2, pages 123–145. Cambridge University Press, 1995.

12 Dominique Larchey-Wendling and Yannick Forster. Hilbert's Tenth Problem in Coq. In *4th International Conference on Formal Structures for Computation and Deduction*, volume 131 of *LIPIcs*, pages 27:1–27:20, February 2019.

13 Dominique Larchey-Wendling and Didier Galmiche. Nondeterministic Phase Semantics and the Undecidability of Boolean BI. *ACM Trans. Comput. Log.*, 14(1):6:1–6:41, 2013. `doi:10.1145/2422085.2422091`.

14 Ranko Lazić and Sylvain Schmitz. Non-Elementary Complexities for Branching VASS, MELL, and Extensions. *ACM Transactions on Computational Logic*, 16(3):20:1–20:30, 2015. `doi:10.1145/2733375`.

15 Jérôme Leroux and Sylvain Schmitz. Demystifying Reachability in Vector Addition Systems. In *LICS 2015: Proceedings of the 30th ACM/IEEE Symposium on Logic in Computer Science*, pages 56–67. IEEE, 2015. `doi:10.1109/LICS.2015.16`.

16 Jérôme Leroux. The Reachability Problem for Petri Nets is Not Primitive Recursive, 2021. `arXiv:2104.12695`.

17 Patrick Lincoln, John C. Mitchell, Andre Scedrov, and Natarajan Shankar. Decision problems for propositional linear logic. In *31st Annual Symposium on Foundations of Computer Science*, volume 2, pages 662–671. IEEE Computer Society, 1990. doi:10.1109/FSCS.1990.89588.

18 Ernst W. Mayr. An algorithm for the general Petri net reachability problem. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, page 238–246, New York, NY, USA, 1981. Association for Computing Machinery. doi:10.1145/800076.802477.

19 Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., 1967.

20 Lutz Straßburger. On the decision problem for MELL. *Theoret. Comput. Sci.*, 768:91–98, 2019.

A Proof (sketch) of Proposition 17

Let us use the denotation $\overline{s}$ (resp. $\overline{d}$) for the dynamic value of register s (resp. d). Hence, the contents of the registers is represented by the vector $[\overline{s};\overline{d}]$ of length 2 with initial value $[x;0]$. Also, the initial value of the PC is $i_0 = i$.

The sub-program $\texttt{MULT\_CST}_\mathrm{a}\,s\,d\,p\,i_0$ multiplies $\overline{s}$ with p and adds the result to the contents of d while emptying s, so the PC moves to i_1 and $\overline{s} = 0$ and $\overline{d} = px$. Then $\texttt{MOD\_CST}_\mathrm{a}\,d\,s\,i_2\,i_4\,q\,i_1$ tests the divisibility of $\overline{d}$ by q, which succeeds under the assumption $qy = px$. By Proposition 15, this transfers the control to i_2 and now $\overline{d} = 0$ and $\overline{s} = px$. Then $\texttt{DIV\_CST}_\mathrm{a}\,s\,d\,q\,i_2$ divides $\overline{s}$ with q while swapping the registers hence now $\overline{s} = 0$, $\overline{d} = y$ and the PC is at i_3. Then $\texttt{TRANSFER}_\mathrm{a}\,d\,s\,i_3$ swaps s with d hence now $\overline{s} = y$ and $\overline{d} = 0$ and PC is now i_4. Finally, $\texttt{JUMP}_\mathrm{a}\,j\,d$ transfers the control to j without altering the registers.

B Proof (sketch) of Proposition 18

As in the proof of Proposition 17, we reach the state where the PC is at i_1 and $\overline{s} = 0$ and $\overline{d} = px$. However now, $\texttt{MOD\_CST}_\mathrm{a}\,d\,s\,i_2\,i_4\,q\,i_1$ gives a negative answer to the divisibility of d by s hence according to Proposition 14, the control is transferred to i_5 while $\overline{s} = px$ and $\overline{d} = 0$. Then $\texttt{DIV\_CST}_\mathrm{a}\,s\,d\,p\,i_5$ divides the contents of s by p, reverting it to its initial value but there is a swap: PC is at i_6, $\overline{s} = 0$ and $\overline{d} = x$. Finally $\texttt{TRANSFER}_\mathrm{a}\,d\,s\,i_6$ swaps again and reverts the registers to their initial values $\overline{s} = x$ and $\overline{d} = 0$ while the PC moves to the end of the sub-program at i_7.

C Proof (sketch) of Theorem 21

The implication $2 \implies 3$ is trivial. We show $1 \implies 2$ and $3 \implies 1$.

Let us start with $1 \implies 2$. As an instance of Lemma 19, if $Q \,/\!\!/_\mathrm{F}\, x \succ y$ holds then we have $(1, \texttt{FRAC\_STEP}_\mathrm{a}\ 1\ Q\ 1) \,/\!\!/_\mathrm{a}\, (1, [x;0]) \succ^+ (1, [y;0])$. By transitivity, from $Q \,/\!\!/_\mathrm{F}\, x \succ^* y$ we can deduce $(1, \texttt{FRAC\_STEP}_\mathrm{a}\ 1\ Q\ 1) \,/\!\!/_\mathrm{a}\, (1, [x;0]) \succ^* (1, [y;0])$.

Now assuming $Q \,/\!\!/_\mathrm{F}\, x \downarrow$, we get some y such that $Q \,/\!\!/_\mathrm{F}\, x \succ^* y$ and $Q \,/\!\!/_\mathrm{F}\, y \not\succ \star$. Hence we have $(1, \texttt{FRAC\_STEP}_\mathrm{a}\ 1\ Q\ 1) \,/\!\!/_\mathrm{a}\, (1, [x;0]) \succ^* (1, [y;0])$. By Lemma 20, as $Q \,/\!\!/_\mathrm{F}\, y \not\succ \star$, we get $(1, \texttt{FRAC\_STEP}_\mathrm{a}\ 1\ Q\ 1) \,/\!\!/_\mathrm{a}\, (1, [y;0]) \succ^* (|\texttt{FRAC\_STEP}_\mathrm{a}\ 1\ Q\ 1| + 1, [y;0])$. We deduce

$$(1, \texttt{FRAC\_MMA}_\mathrm{a}\ Q) \,/\!\!/_\mathrm{a}\, (1, [x;0]) \succ^* (|\texttt{FRAC\_STEP}_\mathrm{a}\ 1\ Q\ 1| + 1, [y;0])$$

since $(1, \texttt{FRAC\_STEP}_\mathrm{a}\ 1\ Q\ 1)$ is a sub-program of $(1, \texttt{FRAC\_MMA}_\mathrm{a}\ Q)$. The nullifying code and the jump finish the computation and we get our proof that $(1, \texttt{FRAC\_MMA}_\mathrm{a}\ Q) \,/\!\!/_\mathrm{a}\, (1, [x;0]) \leadsto (0, [0;0])$ holds.

Let us now finish with $3 \implies 1$ and assume $(1, \texttt{FRAC\_MMA}_{\mathrm{a}}\, Q) /\!\!/_{\mathrm{a}} (1, [x; 0]) \downarrow$. We show that $Q /\!\!/_{\mathrm{F}} x \downarrow$. Because $(1, \texttt{FRAC\_STEP}_{\mathrm{a}}\ 1\ Q\ 1)$ is a sub-program of $(1, \texttt{FRAC\_MMA}_{\mathrm{a}}\, Q)$, we also have $(1, \texttt{FRAC\_STEP}_{\mathrm{a}}\ 1\ Q\ 1) /\!\!/_{\mathrm{a}} (1, [x; 0]) \downarrow$. Hence there is $k, j : \mathbb{N}$ and $\vec{v} : \mathbb{N}^2$ such that $(1, \texttt{FRAC\_STEP}_{\mathrm{a}}\ 1\ Q\ 1) /\!\!/_{\mathrm{a}} (1, [x; 0]) \succ^k (j, \vec{v})$ and $\texttt{out}\ j\ (1, \texttt{FRAC\_STEP}_{\mathrm{a}}\ 1\ Q\ 1)$. We prove $Q /\!\!/_{\mathrm{F}} x \downarrow$ by strong induction on k. By Proposition 1, one can decide between two possibilities:

- either $Q /\!\!/_{\mathrm{F}} x \not\succ \star$ in which case $Q /\!\!/_{\mathrm{F}} x \downarrow$ is obvious;
- or there is y such that $Q /\!\!/_{\mathrm{F}} x \succ y$. We deduce $(1, \texttt{FRAC\_STEP}_{\mathrm{a}}\ 1\ Q\ 1) /\!\!/_{\mathrm{a}} (1, [x; 0]) \succ^\delta (1, [y; 0])$ for some $\delta > 0$ by Lemma 19. Since the step relation is deterministic for Minsky machines, we have $(1, \texttt{FRAC\_STEP}_{\mathrm{a}}\ 1\ Q\ 1) /\!\!/_{\mathrm{a}} (1, [y; 0]) \succ^{k-\delta} (j, \vec{v})$ hence we can apply the induction hypothesis $(k - \delta < k)$ and we get $Q /\!\!/_{\mathrm{F}} y \downarrow$. Combining with $Q /\!\!/_{\mathrm{F}} x \succ y$, we conclude $Q /\!\!/_{\mathrm{F}} x \downarrow$.

An RPO-Based Ordering Modulo Permutation Equations and Its Applications to Rewrite Systems

Dohan Kim ✉
Clarkson University, Potsdam, NY, USA

Christopher Lynch ✉
Clarkson University, Potsdam, NY, USA

Abstract

Rewriting modulo equations has been researched for several decades but due to the lack of suitable orderings, there are some limitations to rewriting modulo permutation equations. Given a finite set of permutation equations E, we present a new RPO-based ordering modulo E using (permutation) group actions and their associated orbits. It is an E-compatible reduction ordering on terms with the subterm property and is E-total on ground terms. We also present a completion and ground completion method for rewriting modulo a finite set of permutation equations E using our ordering modulo E. We show that our ground completion modulo E always admits a finite ground convergent (modulo E) rewrite system, which allows us to obtain the decidability of the word problem of ground theories modulo E.

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

Keywords and phrases Recursive Path Ordering, Permutation Equation, Permutation Group, Rewrite System, Completion, Ground Completion

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.19

1 Introduction

Equations with permutations of variables occur frequently in mathematics and computer science. An equation is called a *permutation equation* [1] if it is of the form $f(x_1, \ldots, x_n) = f(x_{\rho(1)}, \ldots, x_{\rho(n)})$, where ρ is a permutation on $[n]$ (i.e. the set $\{1, \ldots, n\}$). A suitable ordering modulo permutation equations in the context of term rewriting has not been well-studied, although the modulo approach is natural for term rewriting with permutation equations. (For example, a simple permutation equation, such as $f(x, y) \approx f(y, x)$, cannot be oriented into a rewrite rule by well-founded orderings.) If there existed an E-compatible reduction ordering $\succ_E$ for a set of permutation equations E, then it can be used for the *extended rewrite system* for R modulo E, denoted by R, E [11,20]. (In this paper, an ordering modulo E and an E-compatible ordering are used interchangeably.) In particular, such an ordering $\succ_E$ provides a simple termination criterion for R, E, i.e., R, E is terminating if $l \succ_E r$ for all rules $l \to r \in R$ [11,20].

The *recursive path ordering* (RPO) [3,11,24] is one of the most well-known orderings for term rewriting and equational theorem proving. The main underlying idea of RPO is that, roughly speaking, two terms are first compared by their top symbols and the collections of their immediate subterms are recursively compared. Given a total precedence $\succ_{\mathcal{F}}$ on a finite set of function symbols $\mathcal{F}$,[1] the *recursive path ordering with status* [3,10,11,24,27] on $T(\mathcal{F}, \mathcal{X})$ is defined in such a way that $s \succ x$ if and only if $s \neq x$ and x is a variable in s, or else $s = f(s_1, \ldots, s_m) \succ g(t_1, \ldots, t_n) = t$ if and only if

[1] In this paper, we assume that a set of function symbols $\mathcal{F}$ in $T(\mathcal{F}, \mathcal{X})$ is finite and each function symbol in $\mathcal{F}$ has a fixed (bounded) arity. We also assume that a precedence $\succ_{\mathcal{F}}$ on $\mathcal{F}$ is total on $\mathcal{F}$.

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi; Article No. 19; pp. 19:1–19:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

(i) $s_i \succeq t$ for some $i \in [m]$, or
(ii) $f \succ_{\mathcal{F}} g$ and $s \succ t_i$ for all $i \in [n]$, or
(iii) $f = g \in Lex$ (and hence $m = n$), $<s_1, \ldots, s_m> \succ^{lex} <t_1, \ldots, t_m>$, and $s \succ t_i$ for all $i \in [m]$, or
(iv) $f = g \in Mul$ (and hence $m = n$), and $\{s_1, \ldots, s_m\} \succ^{mul} \{t_1, \ldots, t_m\}$,

where Lex (resp. Mul) denotes the set of function symbols with the lexicographic (resp. multiset) status, and $\succ^{lex}$ (resp. $\succ^{mul}$) denotes the lexicographic (resp. multiset) extension of $\succ$.

In [18, 26–28], RPO is adapted for an AC-compatible (resp. A-compatible) simplification ordering on terms that is AC-total (resp. A-total) on ground terms, where AC (resp. A) denotes the associative and commutative (resp. associativity) theory (cf. [23]). (There is also an RPO-like termination relation for a certain class of equations including associativity (see [8, 9] for details).) An RPO is also briefly described in Section 6.1 of [24] for an ordering modulo some simple permutation equations without providing a formal proof.[2] To our knowledge, an E-compatible simplification ordering on terms that is E-total on ground terms for any finite set of permutation equations E has not been studied in the literature.

Meanwhile, a completion procedure [5, 6, 20, 21] for a rewrite system provides a decision procedure for proving the validity of an equational theorem if the procedure generates a finite convergent rewrite system. A completion procedure was extended to a completion procedure modulo a set of equations E [6, 16, 25] for constructing a rewrite system that admits a unique normal form w.r.t. the congruence induced by E. In particular, ground completion modulo E for a ground rewrite systems R provides a decision procedure for the word problem of ground theories modulo E if it generates a finite convergent (modulo E) rewrite system.

In this paper, we present an RPO-based E-compatible simplification ordering $\succ_E$ on terms that is E-total on ground terms for a finite set of permutation equations E. Then we adapt the existing completion modulo a congruence approach to our completion modulo E procedure using the ordering $\succ_E$. We also present our ground completion modulo E and show that it always admits a finite ground convergent (modulo E) rewrite system for a finite set of permutation equations E.

2 Preliminaries

We assume that the reader has some familiarity with term rewriting [11, 20]. The definitions in this section can be found in [3–5, 11, 24, 27]. (For general references on RPOs, see Section 2.2 in [24], Section 5.4.2 in [3], Section 4 in [11], and [10].) In this paper, we usually denote variables by x, y, z, etc., constants by a, b, c, etc., function symbols by f, g, h, etc., and terms by r, s, t, etc., possibly with subscripts. We denote by $[n]$ the set $\{1, \ldots, n\}$.

We denote by $T(\mathcal{F}, \mathcal{X})$ the set of terms over a finite set of function symbols $\mathcal{F}$ and a denumerable set of variables $\mathcal{X}$. An *equation* is an expression $s \approx t$, where s and t are (first-order) terms built from $\mathcal{F}$ and $\mathcal{X}$. A *ground term* (resp. *ground equation*) is a term (resp. an equation) which does not contain any variable.

We write $s[u]$ if u is a subterm of s and denote by $s[t]_p$ the term that is obtained from s by replacing the subterm at position p of s by t.

[2] Our approach uses orbits discussed in the next section, which takes polynomial time for finding them [13]. Without using the group-theoretical approach, the problem of finding the corresponding equivalence classes using permutation equations may take exponential time if one uses traditional equational reasoning approaches [2].

An *equivalence* is a reflexive, transitive, and symmetric binary relation. An equivalence $\sim$ on terms is a *congruence* if $s \sim t$ implies $u[s]_p \sim u[t]_p$ for all terms s, t, u and positions p.

An *equational theory* is a set of equations. We denote by $\approx_E$ the least congruence on $T(\mathcal{F}, \mathcal{X})$ that is stable under substitutions and contains a set of equations E. If $s \approx_E t$ for two terms s and t, then s and t are E-*equivalent*.

A (strict) ordering $\succ$ on terms is an irreflexive and transitive relation on $T(\mathcal{F}, \mathcal{X})$.

An ordering $\succ$ on terms is *monotonic* if $s \succ t$ implies $u[s] \succ u[t]$ for all s, t, and non-empty contexts u. An ordering $\succ$ on terms is *stable under substitutions* if $s \succ t$ implies $s\sigma \succ t\sigma$ for all s, t, and substitutions σ.

An ordering $\succ$ on terms is a *rewrite ordering* if it is monotonic and stable under substitutions. A well-founded rewrite ordering is a *reduction ordering*.

An ordering $\succ$ on terms has the *subterm property* if $t[s]_p \succ s$ for all s, t, and $p \neq \lambda$. (We denote by λ the top position.) An ordering $\succ$ on terms is a *simplification ordering* if it is a rewrite ordering with the subterm property. (We do not need the *deletion property* [11] for a simplification ordering because we assume that each function symbol has a fixed bounded arity in this paper.)

An ordering $\succ$ on terms is *well-founded* if there is no infinite sequence $t_1 \succ t_2 \succ \cdots$.

An ordering $\succ$ on terms is E-*compatible* if $s' \approx_E s \succ t \approx_E t'$ implies $s' \succ t'$ for all s, s', t and t'. An ordering $\succ$ on ground terms is E-*total* if $s \not\approx_E t$ implies $s \succ t$ or $t \succ s$ for all ground terms s and t.

Given a rewrite system R and a set of equations E, the rewrite relation $\rightarrow_{R,E}$ on $T(\mathcal{F}, \mathcal{X})$ is defined by $s \rightarrow_{R,E} t$ if there is a non-variable position p in s, a rewrite rule $l \rightarrow r \in R$, and a substitution σ such that $s|_p \approx_E l\sigma$ and $t = s[r\sigma]_p$. (In this case, we may also write $s \rightarrow_{R,E}^{l \rightarrow r, \sigma} t$ or simply $s \rightarrow_{R,E}^{l \rightarrow r} t$.) The transitive and reflexive closure of $\rightarrow_{R,E}$ is denoted by $\xrightarrow{*}_{R,E}$. We say that a term t is a R, E-*normal form* if there is no term t' such that $t \rightarrow_{R,E} t'$.

The rewrite relation $\rightarrow_{R/E}$ on $T(\mathcal{F}, \mathcal{X})$ is defined by $s \rightarrow_{R/E} t$ if there are terms u and v such that $s \approx_E u$, $u \rightarrow_R v$, and $v \approx_E t$. We simply say the rewrite relation $\rightarrow_{R/E}$ (resp. $\rightarrow_{R,E}$) on $T(\mathcal{F}, \mathcal{X})$ as the rewrite relation R/E (resp. R, E).

The rewrite relation R, E is *Church-Rosser modulo* E if for all terms s and t with $s \xleftrightarrow{*}_{R \cup E} t$, there are terms u and v such that $s \xrightarrow{*}_{R,E} u \xleftrightarrow{*}_E v \xleftarrow{*}_{R,E} t$. The rewrite relation R, E is *convergent modulo* E if R, E is Church-Rosser modulo E and R/E is well-founded.

The substitution σ is *more general modulo* E on X than the substitution θ, denoted by $\sigma \leq_E^X \theta$, if there exists a substitution τ such that $x\theta \approx_E x\sigma\tau$ for all $x \in X$.

Let s and t be terms, and let V be the set of all variables occurring in s and t. Then s and t are E-*unifiable* if there exists a substitution σ, called an E-*unifier*, such that $s\sigma \approx_E t\sigma$. A set of E-unifiers of s and t is *complete*, denoted by $CSU_E(s, t)$, if for every E-unifier τ of s and t, there exists a substitution $\sigma \in CSU_E(s, t)$ such that $\sigma \leq_E^V \tau$. A complete set of E-unifiers of s and t is *minimal*, denoted by $\mu CSU_E(s, t)$, if for all σ and σ' in $CSU_E(s, t)$, $\sigma \leq_E^V \sigma'$ implies $\sigma = \sigma'$.

The *multiset extension of* $\approx_E$ is defined as the smallest relation $\approx_E^{mul}$ on multisets of terms such that $\emptyset \approx_E^{mul} \emptyset$ and $M \cup \{s\} \approx_E^{mul} M' \cup \{t\}$ if $s \approx_E t \wedge M \approx_E^{mul} M'$.

Let $\succ_e$ be an E-compatible ordering on terms. The *lexicographic extension* of $\succ_e$ w.r.t. $\approx_E$ is the relation $\succ_e^{lex}$ on n-tuples of terms defined by $<s_1, \ldots, s_n> \succ_e^{lex} <t_1, \ldots, t_n>$ if $s_1 \approx_E t_1, \ldots, s_{k-1} \approx_E t_{k-1}$ and $s_k \succ_e t_k$ for some $k \in [n]$. The *multiset extension of* $\succ_e$ w.r.t. $\approx_E$ is defined as the smallest ordering $\succ_e^{mul}$ on multisets of terms such that $M \cup \{s\} \succ_e^{mul} N \cup \{t_1, \ldots, t_n\}$ if $M \approx_E^{mul} N$ and $s \succ_e t_i$ for all $i \in [n]$.

▶ **Lemma 1.** *Let $\succ_e$ be an E-compatible ordering on terms.*

(i) *If $\succ_e$ is transitive, then both $\succ_e^{lex}$ and $\succ_e^{mul}$ are transitive.*

(ii) *If $M' \approx_E^{mul} M \succ_e^{mul} N \approx_E^{mul} N'$, then $M' \succ_e^{mul} N'$ for all multisets of terms M, M', N and N'.*

2.1 Leaf permutative equations and permutation groups

We will mainly use the notations and definitions of leaf permutative equations and permutation groups given in [2, 15].

An equation of the form $s \approx s'$ is *leaf permutative* [2] if s and s' are *linear terms* (i.e. no variable occurs twice in s and s') that have the same set of variables and are variants of each other. (Two terms are *variants* if they are instances of each other.) A set of leaf permutative equations $\{s_1 \approx t_1, \ldots, s_n \approx t_n\}$ is *uniform* if for all i and j, s_i and s_j are variants.

If $C[x_1, \ldots, x_n] \approx C[x_{\rho(1)}, \ldots, x_{\rho(n)}]$ is a leaf permutative equation for which all variables are indicated explicitly, then C is the *context* of this equation. We use variable naming in such a way that the left-hand side of each equation in a uniform set of leaf permutative equations has the same name of variables $x_1, \ldots, x_k$ from left to right.

If $e := C[x_1, \ldots, x_n] \approx C[x_{\rho(1)}, \ldots, x_{\rho(n)}]$ is a leaf permutative equation for which all variables are indicated explicitly, then ρ is the permutation of this equation. We denote by $\pi[e]$ the permutation of e. For example, ρ is the permutation of the leaf permutative equation $e' := f(g(x_1, x_2), x_3) \approx f(g(x_1, x_3), x_2)$ (i.e. $\pi[e'] = \rho$) with $\rho(1) = 1, \rho(2) = 3$, and $\rho(3) = 2$.

Let E be a uniform set of leaf permutative equations. Then $\Pi[E]$ is defined as $\Pi[E] := \{\pi[e] \,|\, e \in E\}$. The permutation group generated by $\Pi[E]$ is denoted by $<\Pi[E]>$.

▶ **Theorem 2** ([2, Theorem 1.4]). *Let E be a set of leaf permutative equations and let e be a leaf permutative equation such that $E \cup \{e\}$ is uniform. Then $E \models e$ if and only if $\pi[e] \in <\Pi[E]>$.*

▶ **Example 3.** Let $E = \{f(x_1, x_2, x_3, x_4) \approx f(x_2, x_1, x_3, x_4), f(x_1, x_2, x_3, x_4) \approx f(x_2, x_3, x_4, x_1)\}$. Then $\Pi[E]$ consists of two cycles $\{(1\,2), (1\,2\,3\,4)\}$. Since the two cycles $(1\,2)$ and $(1\,2\,3\,4)$ generate the symmetric group S_4, $<\Pi[E]>$ is S_4. Then $f(x_1, \ldots, x_4) \approx_E f(x_{\rho(1)}, \ldots, x_{\rho(4)})$ for any permutation $\rho \in S_4$ by Theorem 2.

Let G be a group with the identity element I. A (left) *action* of G on a set X is a function $G \times X \to X$ such that for all $x \in X$ and all $g_1, g_2 \in G$: (i) $Ix = x$, and (ii) $(g_1 g_2)x = g_1(g_2 x)$. When such an action is given, we say that G *acts* (left) on the set X, and X is a G*-set*.

Let X be a G-set. For $x_i, x_j \in X$, let $x_i \sim x_j$ if and only if there exists some $g \in G$ such that $gx_i = x_j$. Then, $\sim$ is an equivalence relation on X. The equivalence classes on X determined by $\sim$ are *orbits* of G on X.

▶ **Example 4.** Let $E = \{f(x_1, x_2, x_3, x_4) \approx f(x_2, x_1, x_3, x_4), f(x_1, x_2, x_3, x_4) \approx f(x_1, x_2, x_4, x_3)\}$. Then $\Pi[E]$ consists of two cycles $\{(1\,2), (3\,4)\}$. Let $<\Pi[E]>$ act on the set $X = \{x_1, x_2, x_3, x_4\}$ by $gx_i = x_{g(i)}$ for all $g \in <\Pi[E]>$. Then the orbits of $<\Pi[E]>$ on X are $\{x_1, x_2\}$ and $\{x_3, x_4\}$.

3 An ordering modulo a set of permutation equations

An equation of the form $f(x_1, \ldots, x_n) \approx f(x_{\rho(1)}, \ldots, x_{\rho(n)})$ is a *permutation equation* [1] if ρ is a permutation on $[n]$, which is a restricted form of a leaf permutative equation. In this section, given a set of permutation equations E, we provide an E-compatible simplification ordering on terms that is E-total on ground terms.

Let E be a finite set of permutation equations, where a permutation equation is a restricted form of a leaf permutative equation. Then E can be uniquely decomposed as $\bigcup_{i=1}^{n} E_i$ such that (i) each E_i is a finite set of permutation equations, and (ii) E_j and E_k with $j \neq k$ are disjoint such that if $s_j \approx t_j \in E_j$ and $s_k \approx t_k \in E_k$, then s_j and s_k do not have the same top symbol (and are not variants of each other). Since we assume that each function symbol has a fixed arity, each distinct function symbol occurring in E corresponds to a distinct E_i in E. We denote by $Eq(f)$ the corresponding equational theory with terms headed by such a function symbol f. We also denote by $\mathcal{F}_E$ the set of all function symbols occurring in E and by Lex the set of all other function symbols in $\mathcal{F}$ in $T(\mathcal{F}, \mathcal{X})$ so that $\mathcal{F}$ is split into $\mathcal{F}_E$ and Lex, i.e., $\mathcal{F} = \mathcal{F}_E \cup Lex$. (For comparison, given a total precedence $\succ_{\mathcal{F}}$ on $\mathcal{F}$, if $\mathcal{F}$ is simply $\mathcal{F} = Lex$, then the recursive path ordering $\succ$ (see the *lexicographic path ordering* (LPO) [11]) is total on ground terms, but not necessarily E-compatible on ground terms.)

Given $t = f(s_1, \dots, s_n)$ with $f(x_1, \dots, x_n) \approx f(x_{p(1)}, \dots, x_{p(n)}) \in E$ for some permutation p on $[n]$, let $<\Pi[Eq(f)]>$ act on the set $X = \{x_1, \dots, x_n\}$ by $\rho x_i = x_{\rho(i)}$ for all $\rho \in <\Pi[Eq(f)]>$. We denote each orbit of $<\Pi[Eq(f)]>$ on X by $O_k(f, E)$. (Here X is understood from $f \in \mathcal{F}_E$ and E.) By $Orbit_k(f, t)$ we denote that each x_i in $O_k(f, E)$ is substituted by s_i. (Note that $S = \{s_1, \dots, s_n\}$ can be a multiset, so we first let $<\Pi[Eq(f)]>$ act on the set $X = \{x_1, \dots, x_n\}$ instead of a (possibly) multiset $S = \{s_1, \dots, s_n\}$, and then replace each x_i in $O_k(f, E)$ with s_i in order to obtain $Orbit_k(f, t)$.) The number k in $O_k(f, E)$ is assigned (consecutively starting with 1) in a natural way such that if $k_i < k_j$ for $O_{k_i}(f, E)$ and $O_{k_j}(f, E)$, then $r_i < r_j$ for x_{r_i} and x_{r_j}, where x_{r_i} (resp. x_{r_j}) is the variable with the smallest index in $O_{k_i}(f, E)$ (resp. $O_{k_j}(f, E)$). For example, consider $E = \{f(x_1, x_2, x_3, x_4) \approx f(x_2, x_1, x_3, x_4), f(x_1, x_2, x_3, x_4) \approx f(x_1, x_2, x_4, x_3)\}$ (see Example 4) and $t = f(a, b, c, d)$. Then we have $O_1(f, E) = \{x_1, x_2\}$, $O_2(f, E) = \{x_3, x_4\}$, $Orbit_1(f, t) = \{a, b\}$, and $Orbit_2(f, t) = \{c, d\}$. Note that we only need to compute $O_k(f, E)$ once using $<\Pi[Eq(f)]>$. Then it is easy to obtain $Orbit_k(f, t)$ from $O_k(f, E)$ for any term t headed by $f \in \mathcal{F}_E$. In the following definition, we assume that a total precedence $\succ_{\mathcal{F}}$ on a finite set of function symbols $\mathcal{F}$ is given. We denote by $s \succeq_E t$ either $s \succ_E t$ or $s \approx_E t$.

▶ **Definition 5.** *Given a finite set of permutation equations E, let $s = f(s_1, \dots, s_m)$ and $t = g(t_1, \dots, t_n)$ be terms in $T(\mathcal{F}, \mathcal{X})$. Then $s \succ_E x$ if and only if x is a variable in s, or else $s \succ_E t$ if and only if*

(i) *$s_i \succeq_E t$ for some $i \in [m]$, or*
(ii) *$f \succ_{\mathcal{F}} g$ and $s \succ_E t_i$ for all $i \in [n]$, or*
(iii) *$f = g \in Lex$, $<s_1, \dots, s_m> \succ_E^{lex} <t_1, \dots, t_m>$, and $s \succ_E t_i$ for all $i \in [m]$, or*
(iv) *$f = g \in \mathcal{F}_E$ and there is some positive j such that $Orbit_1(f, s) \approx_E^{mul} Orbit_1(g, t), \dots,$ $Orbit_{j-1}(f, s) \approx_E^{mul} Orbit_{j-1}(g, t)$, $Orbit_j(f, s) \succ_E^{mul} Orbit_j(g, t)$, and $s \succ_E t_i$ for all $i \in [m]$.*

The following lemma directly follows from the definition of $Orbit_j(f, t)$ and $\approx_E$.

▶ **Lemma 6.** *Given a finite set of permutation equations E, let $s = f(s_1, \dots, s_n)$ and $t = f(t_1, \dots, t_n)$ be terms in $T(\mathcal{F}, \mathcal{X})$ with $f \in \mathcal{F}_E$. Then $s \approx_E t$ if and only if $Orbit_1(f, s) \approx_E^{mul} Orbit_1(f, t), \dots, Orbit_k(f, s) \approx_E^{mul} Orbit_k(f, t)$, where k is the number of orbits of $<\Pi[Eq(f)]>$ on $X = \{x_1, \dots, x_n\}$.*

▶ **Example 7.** Let $E = \{f(x_1, x_2, x_3, x_4) \approx f(x_2, x_1, x_3, x_4), f(x_1, x_2, x_3, x_4) \approx f(x_2, x_3, x_4, x_1)\}$ (see Example 3) and consider two terms $s = f(d, c, b, g(a))$ and $t = f(a, b, c, d)$ with $f \succ_{\mathcal{F}} g \succ_{\mathcal{F}} a \succ_{\mathcal{F}} b \succ_{\mathcal{F}} c \succ_{\mathcal{F}} d$. Then E is simply decomposed into $E = E_1$. We have $s \succ_E t$ by Case (iv), since $Orbit_1(f, s) = \{d, c, b, g(a)\} \succ_E^{mul} \{a, b, c, d\} = Orbit_1(f, t)$, $s \succ_E a$, $s \succ_E b$, $s \succ_E c$, and $s \succ_E d$. It is easy to verify that $\{d, c, b, g(a)\} \succ_E^{mul} \{a, b, c, d\}$ since $g(a) \succ_E a$ by Case (i). We leave it to the reader to verify that $s \succ_E a$, $s \succ_E b$, $s \succ_E c$, and $s \succ_E d$. (This is clear once we have the subterm property of $\succ_E$ (see Lemma 13).)

▶ **Example 8.** Let $E = \{f(x_1, x_2, x_3, x_4) \approx f(x_2, x_1, x_3, x_4), f(x_1, x_2, x_3, x_4) \approx f(x_1, x_2, x_4, x_3)\}$ (see Example 4) and consider two terms $s = f(x, a, c, a)$ and $t = f(a, x, b, c)$ with $f \succ_{\mathcal{F}} a \succ_{\mathcal{F}} b \succ_{\mathcal{F}} c$. Then E is simply decomposed into $E = E_1$. We have $s \succ_E t$ by

Case (iv), since $Orbit_1(f,s) = \{x,a\} \approx_E^{mul} \{a,x\} = Orbit_1(f,t)$, $Orbit_2(f,s) = \{c,a\} \succ_E^{mul} \{b,c\} = Orbit_2(f,t)$, $s \succ_E a$, $s \succ_E x$, $s \succ_E b$, and $s \succ_E c$. It is easy to verify that $\{c,a\} \succ_E^{mul} \{b,c\}$ since $a \succ_E b$. We leave it to the reader to verify that $s \succ_E a$, $s \succ_E x$, $s \succ_E b$, and $s \succ_E c$.

▶ **Example 9.** Let $E = \{f(x_1,x_2) \approx f(x_2,x_1), g(x_1,x_2,x_3) \approx g(x_2,x_1,x_3), g(x_1,x_2,x_3) \approx g(x_1,x_3,x_2)\}$ and consider two terms $s = h(f(a,g(b,a,x)),a)$ and $t = h(f(g(a,x,b),a),b)$ with $h \succ_{\mathcal{F}} f \succ_{\mathcal{F}} g \succ_{\mathcal{F}} a \succ_{\mathcal{F}} b$. Then E is decomposed into $E_1 \cup E_2$, where $E_1 = \{f(x_1,x_2) \approx f(x_2,x_1)\}$ and $E_2 = \{g(x_1,x_2,x_3) \approx g(x_2,x_1,x_3), g(x_1,x_2,x_3) \approx g(x_1,x_3,x_2)\}$. We have $s \succ_E t$ by Case (iii), since $f(a,g(b,a,x)) \approx_E f(g(a,x,b),a)$ by Lemma 6, $a \succ_E b$, $s \succ_E f(g(a,x,b),a)$, and $s \succ_E b$. We may verify that $s \succ_E f(g(a,x,b),a)$ by Case (i) and Lemma 6. We leave it to the reader to verify that $s \succ_E b$.

▶ **Example 10.** Let $E = \{f(x_1,x_2) \approx f(x_2,x_1), g(x_1,x_2,x_3) \approx g(x_2,x_1,x_3)\}$ and consider two terms $s = f(c,g(b,a,a))$ and $t = f(g(a,b,b),c)$ with $f \succ_{\mathcal{F}} g \succ_{\mathcal{F}} a \succ_{\mathcal{F}} b \succ_{\mathcal{F}} c$. Then E is decomposed into $E_1 \cup E_2$, where $E_1 = \{f(x_1,x_2) \approx f(x_2,x_1)\}$ and $E_2 = \{g(x_1,x_2,x_3) \approx g(x_2,x_1,x_3)\}$. We have $s \succ_E t$ by Case (iv), since $Orbit_1(f,s) = \{c,g(b,a,a)\} \succ_E^{mul} \{g(a,b,b),c\} = Orbit_1(f,t)$ by $g(b,a,a) \succ_E g(a,b,b)$, $s \succ_E g(a,b,b)$, and $s \succ_E c$. We may verify that $g(b,a,a) \succ_E g(a,b,b)$ by Case (iv), since $Orbit_1(g,g(b,a,a)) = \{b,a\} \approx_E^{mul} \{a,b\} = Orbit_1(g,g(a,b,b))$, $Orbit_2(g,g(b,a,a)) = \{a\} \succ_E^{mul} \{b\} = Orbit_2(g,g(a,b,b))$, $g(b,a,a) \succ_E a$, and $g(b,a,a) \succ_E b$. We leave it to the reader to verify that $s \succ_E g(a,b,b)$, $s \succ_E c$, $g(b,a,a) \succ_E a$, and $g(b,a,a) \succ_E b$.

In the following, we denote by $Vars(t)$ the set of variables occurring in t and by $top(t)$ the top symbol of t.

▶ **Lemma 11.** *$\succ_E$ is E-compatible.*

Proof. Let s, s', t, and t' be terms with $s' \approx_E s \succ_E t \approx_E t'$. We show that $s' \succ_E t'$. If t is a variable, then $s \neq t$ and $t \in Vars(s)$. We may infer that $t = t'$ and s' is not a variable. Since s' is not a variable with $Vars(s) = Vars(s')$, we have $s' \neq t'$ and $t' \in Vars(s')$, and thus $s' \succ_E t'$. Therefore, we assume that t is not a variable and let $s = f(s_1,\ldots,s_m)$ and $t = g(t_1,\ldots,t_n)$. We proceed by induction on $|s|+|t|$. (Note that we do not need to consider $s' \approx_E s$ (resp. $t \approx_E t'$) on the top position for the following 1 (resp. 2)).

1. If $s \succ_E t$ by Case (i), then we have $s_i \succeq_E t$ for some $i \in [m]$. Then s' is of the form $s' = f(s'_1,\ldots,s'_m)$ with $s_k \approx_E s'_{\rho(k)}$ for all $k \in [m]$ and some (permutation) $\rho \in S_m$. Since $s'_{\rho(i)} \succeq_E t'$ for some $i \in [m]$ by induction hypothesis, we have $s' \succ_E t'$ by Case (i).
2. If $s \succ_E t$ by Case (ii), then we have $f \succ_{\mathcal{F}} g$ and $s \succ_E t_i$ for all $i \in [n]$. Then t' is of the form $t' = g(t'_1,\ldots,t'_n)$ with $t_k \approx_E t'_{\pi(k)}$ for all $k \in [n]$ and some $\pi \in S_n$. Since $top(s') = f \succ_{\mathcal{F}} g = top(t')$ and $s' \succ_E t'_{\pi(i)}$ for all $i \in [n]$ by induction hypothesis, we have $s' \succ_E t'$ by Case (ii).
3. If $s \succ_E t$ by Case (iii), then we have $f = g \in Lex$, $<s_1,\ldots,s_m> \succ_E^{lex} <t_1,\ldots,t_m>$, and $s \succ_E t_i$ for all $i \in [m]$. Then s' is of the form $s' = f(s'_1,\ldots,s'_m)$ with $s_k \approx_E s'_k$ for all $k \in [m]$ and t' is of the form $t' = g(t'_1,\ldots,t'_m)$ with $t_k \approx_E t'_k$ for all $k \in [m]$. By induction hypothesis, we have $<s'_1,\ldots,s'_m> \succ_E^{lex} <t'_1,\ldots,t'_m>$ and $s' \succ_E t'_i$ for all $i \in [m]$, and thus we have $s' \succ_E t'$ by Case (iii).
4. If $s \succ_E t$ by Case (iv), then we have $f = g \in \mathcal{F}_E$, and there is some positive j such that $Orbit_1(f,s) \approx_E^{mul} Orbit_1(g,t),\ldots,Orbit_{j-1}(f,s) \approx_E^{mul} Orbit_{j-1}(g,t)$, $Orbit_j(f,s) \succ_E^{mul} Orbit_j(g,t)$, and $s \succ_E t_i$ for all $i \in [m]$. Then s' is of the form $s' = f(s'_1,\ldots,s'_m)$ with $s_k \approx_E s'_{\rho(k)}$ for all $k \in [m]$ and some $\rho \in <\Pi[Eq(f)]>$ and t' is of the form $t' = g(t'_1,\ldots,t'_m)$ with $t_k \approx_E t'_{\pi(k)}$ for all $k \in [m]$ and some $\pi \in <\Pi[Eq(g)]>$. By the definition

of $\approx_E^{mul}$, we have $Orbit_k(f, s') \approx_E^{mul} Orbit_k(f, s)$ and $Orbit_k(g, t) \approx_E^{mul} Orbit_k(g, t')$ for all $k \in [j-1]$, which implies that $Orbit_k(f, s') \approx_E^{mul} Orbit_k(g, t')$ for all $k \in [j-1]$. Furthermore, by induction hypothesis and Lemma 1(ii), we have $Orbit_j(f, s') \succ_E^{mul} Orbit_j(g, t')$ and $s' \succ_E t'_{\pi(i)}$ for all $i \in [m]$, and thus we have $s' \succ_E t'$ by Case (iv). (We may apply Lemma 1(ii) here because the induction hypothesis implies that $\succ_E$ is E-compatible for all terms r and u with $r \succ_E u$ and $|r| + |u| < |s| + |t|$.) ◀

▶ **Lemma 12.** *$\succ_E$ is transitive.*

Proof. Suppose that $r \succ_E s$ and $s \succ_E t$. Then r and s cannot be variables by Definition 5. Let $r = f(r_1, \dots, r_l)$ and $s = g(s_1, \dots, s_m)$. If t is a variable, then $t \in Vars(s)$. We leave it to the reader to verify that $t \in Vars(r)$ as well, which shows that $r \succ_E t$. Therefore, we assume that t is not a variable and let $t = h(t_1, \dots, t_n)$. We show that $r \succ_E t$ by induction on $|r| + |s| + |t|$.

1. If $r \succ_E s$ by Case (i), then $r_i \succeq_E s$ for some $i \in [l]$. By induction hypothesis and the E-compatibility of $\succ_E$, we have $r_i \succeq_E t$, and thus $r \succ_E t$ by Case (i).
2. If $s \succ_E t$ by Case (i) and $r \succ_E s$ by Case (ii), (iii), or (iv), then we have $r \succ_E s_i$ for all $i \in [m]$ and $s_j \succeq_E t$ for some $j \in [m]$. It follows that $r \succ_E s_j \succeq_E t$ for some $j \in [m]$, and thus $r \succ_E t$ by induction hypothesis and the E-compatibility of $\succ_E$.
3. If $r \succ_E s$ and $s \succ_E t$ by Case (ii), (iii), or (iv), then $f \succeq_{\mathcal{F}} h$ and $s \succ_E t_i$ for all $i \in [n]$.
   - 3.1. If $f \succ_{\mathcal{F}} h$, then we have $r \succ_E t_i$ for all $i \in [n]$ by induction hypothesis, and thus $r \succ_E t$ by Case (ii).
   - 3.2. If $f = g = h \in Lex$ with $r \succ_E s$ and $s \succ_E t$ by Case (iii), then we have $<r_1, \dots, r_l> \succ_E^{lex} <t_1, \dots, t_l>$ and $r \succ_E t_i$ for all $i \in [l]$ by induction hypothesis and Lemma 1(i) (using the E-compatibility of $\succ_E$), and thus $r \succ_E t$ by Case (iii).
   - 3.3. If $f = g = h \in \mathcal{F}_E$ with $r \succ_E s$ and $s \succ_E t$ by Case (iv), then there is some positive j such that $Orbit_1(f, r) \approx_E^{mul} Orbit_1(h, t), \dots, Orbit_{j-1}(f, r) \approx_E^{mul} Orbit_{j-1}(h, t)$, $Orbit_j(f, r) \succ_E^{mul} Orbit_j(h, t)$, and $r \succ_E t_i$ for all $i \in [l]$ by induction hypothesis and Lemma 1(i) and (ii) (using the E-compatibility of $\succ_E$), and thus $r \succ_E t$ by Case (iv). ◀

▶ **Lemma 13.** *$\succ_E$ has the subterm property.*

Proof. By the transitivity of $\succ_E$, it suffices to show that $s = f(\dots t \dots) \succ_E t$. If t is a variable, then we have $t \in Vars(s)$, and thus $s \succ_E t$. Therefore, we assume that t is not a variable. Since $t \succeq_E t$, we have $s \succ_E t$ by Case (i). ◀

▶ **Lemma 14.** *$\succ_E$ is irreflexive.*

Proof. Suppose, towards a contradiction, that there exists some t such that $t \succ_E t$. If t is a variable, then $t \succ_E t$ is not possible by Definition 5, which is a contradiction. Therefore, we assume that t is not a variable and let $t = f(t_1, \dots, t_n)$. We proceed by induction on $|t|$.

1. If $t \succ_E t$ by Case (i), then $t_i \succeq_E t$. On the other hand, we have $t \succ_E t_i$ by the subterm property of $\succ_E$. Then by the E-compatibility and transitivity of $\succ_E$, we have $t_i \succ_E t_i$, which is a contradiction by induction hypothesis.
2. If $t \succ_E t$ by Case (iii) or (iv), then there must exist some $i \in [n]$ such that $t_i \succ_E t_i$, which is a contradiction by induction hypothesis. (Note that $t \succ_E t$ by Case (ii) is not possible.) ◀

▶ **Lemma 15.** *$\succ_E$ is monotonic.*

Proof. Let $s \succ_E t$. We show that $r = f(s_1, \ldots, s_{i-1}, s, s_{i+1}, \ldots, s_n) \succ_E f(s_1, \ldots, s_{i-1}, t, s_{i+1}, \ldots, s_n) = u$, since the monotonicity of $\succ_E$ directly follows from this replacement property of $\succ_E$. By the subterm property of $\succ_E$, we have $r \succ_E s_j$ for all $j \in \{1, \ldots, i-1, i+1, \ldots, n\}$. By the subterm property and transitivity of $\succ_E$, we also have $r \succ_E t$.

If $f \in \mathit{Lex}$, then $r \succ_E u$ by Case (iii) because we have $s_1 \approx_E s_1, \ldots, s_{i-1} \approx_E s_{i-1}$ and $s \succ_E t$.

If $f \in \mathcal{F}_E$, then there is some positive j such that $s \in \mathit{Orbit}_j(f, r)$ and $t \in \mathit{Orbit}_j(f, u)$ and all other $\mathit{Orbit}_k(f, r)$ and $\mathit{Orbit}_k(f, u)$ are the same w.r.t. $\approx_E^{mul}$. Since $\mathit{Orbit}_j(f, r)$ and $\mathit{Orbit}_j(f, u)$ differ by only s and t, we have $\mathit{Orbit}_j(f, r) \succ_E^{mul} \mathit{Orbit}_j(f, u)$ by the definition of $\succ_E^{mul}$, and thus $r \succ_E u$ by Case (iv). ◀

▶ **Lemma 16.** *$\succ_E$ is stable under substitutions.*

Proof. Let $s = f(s_1, \ldots, s_m) \succ_E t$. If t is a variable, then $t \in \mathit{Vars}(s)$ and $t\sigma$ is a strict subterm of $s\sigma$ for all substitutions σ. By the subterm property of $\succ_E$, we have $s\sigma \succ_E t\sigma$. Therefore, we assume that t is not a variable and let $t = g(t_1, \ldots, t_n)$. We show that $s\sigma \succ_E t\sigma$ for all substitutions σ by induction on $|s| + |t|$.

1. If $s \succ_E t$ by Case (i), then $s_i \succeq_E t$ for some $i \in [m]$. By induction hypothesis and the stability under substitutions of $\approx_E$, we have $s_i\sigma \succeq_E t\sigma$, and thus $s\sigma \succ_E t\sigma$ by Case (i).
2. If $s \succ_E t$ by Case (ii), then $f \succ_{\mathcal{F}} g$ and $s \succ_E t_i$ for all $i \in [n]$. Since $\mathit{top}(s\sigma) = f \succ_{\mathcal{F}} g = \mathit{top}(t\sigma)$ and $s\sigma \succ_E t_i\sigma$ for all $i \in [n]$ by induction hypothesis, we have $s\sigma \succ_E t\sigma$ by Case (ii).
3. If $s \succ_E t$ by Case (iii), then $f = g \in \mathit{Lex}$, $<s_1, \ldots, s_m> \succ_E^{lex} <t_1, \ldots, t_m>$, and $s \succ_E t_i$ for all $i \in [m]$. Then we have $\mathit{top}(s\sigma) = f = g = \mathit{top}(t\sigma) \in \mathit{Lex}$, $<s_1\sigma, \ldots, s_m\sigma> \succ_E^{lex} <t_1\sigma, \ldots, t_m\sigma>$, and $s\sigma \succ_E t_i\sigma$ for all $i \in [m]$ by induction hypothesis and the stability under substitutions of $\approx_E$. Thus, $s\sigma \succ_E t\sigma$ by Case (iii).
4. If $s \succ_E t$ by Case (iv), then $f = g \in \mathcal{F}_E$, and there is some positive j such that $\mathit{Orbit}_1(f, s) \approx_E^{mul} \mathit{Orbit}_1(g, t), \ldots, \mathit{Orbit}_{j-1}(f, s) \approx_E^{mul} \mathit{Orbit}_{j-1}(g, t)$, $\mathit{Orbit}_j(f, s) \succ_E^{mul} \mathit{Orbit}_j(g, t)$, and $s \succ_E t_i$ for all $i \in [m]$. Then we have $\mathit{top}(s\sigma) = f = g = \mathit{top}(t\sigma) \in \mathcal{F}_E$ and there is some positive j such that $\mathit{Orbit}_1(f, s\sigma) \approx_E^{mul} \mathit{Orbit}_1(g, t\sigma), \ldots, \mathit{Orbit}_{j-1}(f, s\sigma) \approx_E^{mul} \mathit{Orbit}_{j-1}(g, t\sigma)$, $\mathit{Orbit}_j(f, s\sigma) \succ_E^{mul} \mathit{Orbit}_j(g, t\sigma)$, and $s\sigma \succ_E t_i\sigma$ for all $i \in [m]$ by induction hypothesis and the stability under substitutions of $\approx_E$. Thus, $s\sigma \succ_E t\sigma$ by Case (iv). ◀

▶ **Lemma 17.** *$\succ_E$ is E-total on ground terms.*

Proof. Let s and t be ground terms such that $s = f(s_1, \ldots, s_m)$ and $t = g(t_1, \ldots, t_n)$. We show that either $s \succ_E t$ or $t \succ_E s$ or $s \approx_E t$ by induction on $|s| + |t|$. In the following, for all s' and t' with $|s'| + |t'| < |s| + |t|$, we have either $s' \succ_E t'$ or $t' \succ_E s'$ or $s' \approx_E t'$ by induction hypothesis.

1. If $s_i \succeq_E t$ for some $i \in [m]$, then $s \succ_E t$ by Case (i).
2. Otherwise, if $t \succ_E s_i$ for all $i \in [m]$, then we consider the following subcases:
2.1. If $t_i \succeq_E s$ for some $i \in [n]$, then $t \succ_E s$ by Case (i).
2.2. Otherwise, if $s \succ_E t_i$ for all $i \in [n]$, then we consider the following subcases:
2.2.1. If $f \succ_{\mathcal{F}} g$, then $s \succ_E t$ by Case (ii).
2.2.2. If $g \succ_{\mathcal{F}} f$, then $t \succ_E s$ by Case (ii).
2.2.3. If $f = g$ (and hence $m = n$), then we consider the following subcases:
2.2.3.1. If $s_k \approx_E t_k$ for all $k \in [m]$, then $s \approx_E t$.

2.2.3.2. If $f = g \in Lex$ and $s_1 \approx_E t_1, \ldots, s_{j-1} \approx_E t_{j-1}$, and $s_j \succ_E t_j$ (resp. $t_j \succ_E s_j$) for some $j \in [m]$, then $s \succ_E t$ (resp. $t \succ_E s$) by Case (iii).

2.2.3.3. If $f = g \in \mathcal{F}_E$ and $Orbit_1(f,s) \approx_E^{mul} Orbit_1(g,t), \ldots, Orbit_{j-1}(f,s) \approx_E^{mul} Orbit_{j-1}(g,t)$, and $Orbit_j(f,s) \succ_E^{mul} Orbit_j(g,t)$ (resp. $Orbit_j(g,t) \succ_E^{mul} Orbit_j(f,s)$) for some positive j, then $s \succ_E t$ (resp. $t \succ_E s$) by Case (iv).

2.2.3.4. If $f = g \in \mathcal{F}_E$ and $Orbit_1(f,s) \approx_E^{mul} Orbit_1(g,t), \ldots, Orbit_k(f,s) \approx_E^{mul} Orbit_k(g,t)$, where k is the number of orbits of $<\Pi[Eq(f)]>$ on $X = \{x_1, \ldots, x_m\}$, then $s \approx_E t$.

Thus, we have either $s \succ_E t$ or $t \succ_E s$ or $s \approx_E t$ for each of the above cases by induction hypothesis. ◀

Lemmas 11–17 now amount to the following theorem.

▶ **Theorem 18.** *Let E be a finite set of permutation equations. Then $\succ_E$ is an E-compatible simplification ordering on terms and is E-total on ground terms.*

Since every simplification ordering on terms (i.e. $T(\mathcal{F}, \mathcal{X})$) is a reduction ordering [3, 24], we have the following corollary from Theorem 18. (Recall that $\mathcal{F}$ is finite in this paper.)

▶ **Corollary 19.** *Let E be a finite set of permutation equations. Then $\succ_E$ is an E-compatible reduction ordering on terms with the subterm property and is E-total on ground terms.*

Given a total precedence $\succ_\mathcal{F}$ on a finite set of function symbols $\mathcal{F}$ and two terms s and t, one can determine whether $s \succ_{rpo} t$ in time $O(n^2)$ (measured in $n = |s| + |t|$) using the dynamic programming approach [30, 31], where $\succ_{rpo}$ is the recursive path ordering with status. Given a finite set of permutation equations E and two terms s and t, one can also determine whether $s \approx_E t$ in time $O(n^2)$ (measured in $n = |s| + |t|$) using an additional table that can be constructed in polynomial time [1]. In the following theorem, we assume that this additional table and the orbits $O_k(f, E)$ for each $f \in \mathcal{F}_E$ are given for a (fixed) finite set of permutation equations E. Note that $O_k(f, E)$ can be computed only once in polynomial time [13] for each $f \in \mathcal{F}_E$. Once we have the orbits $O_k(f, E)$, it is easy to see that every $Orbit_k(f, t)$ can be immediately obtained for any term t headed by $f \in \mathcal{F}_E$. For the proof of the following theorem, we use the dynamic programming-like technique found in Section 5 of [17]. Recall that our ordering $\succ_E$ assumes a total precedence $\succ_\mathcal{F}$ on a finite set of function symbols $\mathcal{F}$.

▶ **Theorem 20.** *Given a finite set of permutation equations E, we can determine whether $s \succ_E t$ for two terms s and t in time $O(n^4)$ (measured in $n = |s| + |t|$).*

Proof. We construct a 2-dimensional array A of size $|s| \cdot |t|$ using a bottom-up approach. First, we assume that all subterms of s have already been compared to all subterms of t with the exception of s and t themselves. We also assume that the results are stored and easily accessible in A in such a way that if s_i is a subterm of s at position p and t_j is a subterm of t at position q with $p \neq \lambda$ or $q \neq \lambda$, then $A[p, q]$ indicates whether $s_i \approx_E t_j$, $s_i \succ_E t_j$, $t_j \succ_E s_i$, or s_i and t_j are incomparable.

Now we show that the time required to compare s and t, denoted by $TCOMP(s, t)$, takes $O(n^2)$ time using the above assumptions. We first test whether $s \approx_E t$ in $O(n^2)$ time. If $s \not\approx_E t$, then we proceed by case analysis in Definition 5. The straightforward comparisons of all s_i with t for Case (i), and s with all t_i for Case (ii) in the worst case using the existing entries of A takes $O(n)$ time. Similarly, it takes $O(n)$ time to compare s and t for Case (iii) using the existing entries of A. For Case (iv), since we already have the orbits $O_k(f, E)$, it takes at most $O(n)$ time to find every $Orbit_k(f,s)$ (and $Orbit_k(g,t)$ too). Then all s_i are compared to all t_j in the worst case using the existing entries of A, which takes $O(n^2)$ time. This shows that $TCOMP(s,t)$ takes $O(n^2)$ time.

Finally, it remains to sum up all possible $TCOMP(s_i,t_j)$ in a bottom-up way, where s_i is a subterm of s and t_j is a subterm of t. Since the number of subterms of s (resp. t) is bounded above by $O(|s|)$ (resp. $O(|t|)$), we have $\sum TCOMP(s_i,t_j) = O(|s|\cdot|t|\cdot TCOMP(s,t))$, where $TCOMP(s,t)$ takes $O(n^2)$ time. Thus, $s \succ_E t$ can be determined in time $O(n^4)$. ◀

4 Completion modulo a set of permutation equations

Knuth-Bendix completion [21] (or simply *completion*) is a technique using equations as rewrite rules and is used for solving the word problem for a finite set of equations. It is often parameterized by a reduction ordering to ensure that the resulting rewrite system terminates. If the procedure succeeds, then it yields a convergent rewrite system, which allows one to solve the word problem for a given finite set of equations. If the procedure encounters an unorientable equation w.r.t. a given reduction ordering, then it fails, i.e., the procedure cannot be continued.

A permutation equation (e.g. a commutativity equation) often cannot be oriented into a rewrite rule without losing the termination property, which causes the failure of the completion procedure. Therefore, it is natural to view permutation equations as *structural axioms* [5] (defining a congruence on terms) instead of viewing them as *simplifiers* (defining a terminating rewrite relation on terms). In this situation, we need to consider completion modulo E for a finite set of permutation equations E in order to construct a convergent (modulo E) rewrite system R, where normal forms w.r.t. R are unique up to the congruence induced by E. Here we are mainly concerned with the rewrite relation R,E instead of R/E because R/E tends to be less efficient than R,E [5]. We give an adapted version of completion modulo E in [5,6,20] for a finite set of permutation equations E using R,E in this section. We first give the necessary definitions used in completion modulo E. In the following, we denote by $\mathcal{FP}os(t)$ the set of non-variable positions of t.

▶ **Definition 21** ([5,20]). *Let R be a rewrite system and E be a finite set of equations.*

1. *A proof for $t \approx t'$ is a* rewrite proof modulo E *for R if for some t_1 and t_1', there is a proof of the form $t \xrightarrow{*}_{R,E} t_1 \xleftrightarrow{*}_E t_1' \xleftarrow{*}_{R,E} t'$.*
2. *A* peak *is a proof of the form $t_1 \leftarrow_R t \rightarrow_{R,E} t_2$ and a* cliff *is a proof of the form $t_1 \leftrightarrow_E t \rightarrow_{R,E} t_2$ or $t_1 \rightarrow_{R,E} t \leftrightarrow_E t_2$.*
3. *Given two rules $s \rightarrow t$ and $l \rightarrow r$ such that $Vars(s) \cap Vars(l) = \emptyset$ and $s|_p$ and l are E-unifiable at position p of $\mathcal{FP}os(s)$ with a minimal complete set of E-unifiers Ψ, the set $\{u \approx v \mid u = s[r]_p\sigma, v = t\sigma, \sigma \in \Psi\}$ is called a* set of E-critical pairs *of the rule $l \rightarrow r$ on $s \rightarrow t$ at position p of $\mathcal{FP}os(s)$.*
4. *The set of E-critical pairs between the rules in a rewrite system R is denoted by $CP_E(R)$. The set of E-critical pairs of the rules in R on the equations in E is denoted by $CP_E(R,E)$, where an equation $s \approx t \in E$ is considered as a rule $s \rightarrow t$ or $t \rightarrow s$.*

If R,E is Church-Rosser modulo E, then every peak or cliff (see Definition 21) can be replaced by a rewrite proof modulo E, where a proof is a rewrite proof modulo E if and only if it contains no peak or cliff [5,6]. (Note that non-overlap peaks (resp. cliffs) and variable overlap peaks (resp. cliffs) can always be replaced by rewrite proofs modulo E (see [5,6]).) Conversely, if R,E is not Church-Rosser modulo E and R/E is terminating, then there is some peak or cliff which cannot be replaced by a rewrite proof modulo E [5,6]. In completion

$$\text{ORIENT: } \frac{P \cup \{p \approx q\}; R}{P; R \cup \{p \to q\}} \quad \text{if } p \succ_E q.$$

$$\text{DEDUCE: } \frac{P; R}{P \cup \{p \approx q\}; R} \quad \text{if } p \approx q \in CP_E(R).$$

$$\text{SIMPLIFY: } \frac{P \cup \{p \approx q\}; R}{P \cup \{p' \approx q\}; R} \quad \text{if } p \to_{R,E} p'.$$

$$\text{DELETE: } \frac{P \cup \{p \approx q\}; R}{P; R} \quad \text{if } p \xleftrightarrow{*}_E q.$$

$$\text{COMPOSE: } \frac{P; R \cup \{l \to r\}}{P; R \cup \{l \to r'\}} \quad \text{if } r \to_{R,E} r'.$$

$$\text{COLLAPSE: } \frac{P; R \cup \{l \to r\}}{P \cup \{l' \approx r\}; R} \quad \text{if } l \to_{R,E}^{g \to d, \sigma} l' \text{ for } g \to d \in R \text{ and } l \to r >>_E g \to d.$$

Above, $\succ_E$ is our E-compatible reduction ordering on terms and $\sqsupset_E$ denotes a proper encompassment ordering modulo E, where E is a finite set of permutation equations.

Figure 1 Completion modulo a finite set of permutation equations E.

modulo E (or *extended completion* [5,6]), $CP_E(R)$ is used to eliminate peaks that are proper overlaps, while either $CP_E(R, E)$ or $EXT_E(R)$ in the following definition is used to eliminate cliffs that are proper overlaps (see [5,20]). We denote by $\overrightarrow{E}$ the set $\{s \to t, t \to s \mid s \approx t \in E\}$.

▶ **Definition 22** ([5,16]). *Let $l \to r \in R$ and $u \to v \in \overrightarrow{E}$ with $Vars(l) \cap Vars(u) = \emptyset$, such that some proper non-variable subterm $u|_p$ of u is E-unifiable with l. Then $u[l]_p \to u[r]_p$ is the* extended rule *of $l \to r$ w.r.t. E. The set of all extended rules in R w.r.t. E is denoted by $EXT_E(R)$.*

Observe that if E is a set of permutation equations, then $EXT_E(R)$ is the empty set for any rewrite system R because every proper subterm $u|_p$ of u in Definition 22 is a variable. Therefore, extended completion in [5,6] can be easily adapted for completion modulo a finite set of permutation equations E without taking $EXT_E(R)$ into account. Note that we do not need to compute $CP_E(R, E)$ either because cliffs that are proper overlaps do not occur with E, which is also the reason why $EXT_E(R)$ is empty.

The *proper encompassment ordering modulo E* [20] is defined in such a way that $l \sqsupset_E g$ if there is some substitution σ such that $l|_p \xleftrightarrow{*}_E g\sigma$ with $p \neq \lambda$, or $l \approx_E g\sigma$ and σ is not a renaming. In Figure 1, $>>_E$ is defined as follows: $l \to r >>_E g \to d$ if $l \sqsupset_E g$ or l and g are subsumption equivalent (w.r.t. $\approx_E$) and $r \succ_E d$ (see Section 18.3 and 18.4 in [20]).

In the remainder of this section, we denote by P a set of equations, R a set of rewrite rules, E a finite set of permutation equations, and by $\succ_E$ our E-compatible simplification ordering on terms. Now we write $P; R \vdash P'; R'$ to indicate that $P'; R'$ can be obtained from $P; R$ by application of an inference rule in Figure 1. A *derivation* is a sequence of states $P_0; R_0 \vdash P_1; R_1 \vdash \cdots$. Let $P_0; R_0 \vdash P_1; R_1 \vdash \cdots$ be a derivation. Then P_∞ denotes *the set of persisting equations* $\bigcup_i \bigcap_{j \geq i} P_j$. Similarly, R_∞ denotes *the set of persisting rules* $\bigcup_i \bigcap_{j \geq i} R_j$. A derivation is said to be *fair* [7] if any transition rule that is (continuously) enabled is

$$\text{ORIENT: } \frac{P \cup \{p \approx q\}; R}{P; R \cup \{p \to q\}} \quad \text{if } p \succ_E q.$$

$$\text{SIMPLIFY: } \frac{P \cup \{p \approx q\}; R}{P \cup \{p' \approx q\}; R} \quad \text{if } p \to_{R,E} p'.$$

$$\text{DELETE: } \frac{P \cup \{p \approx q\}; R}{P; R} \quad \text{if } p \overset{*}{\leftrightarrow}_E q.$$

$$\text{COMPOSE: } \frac{P; R \cup \{l \to r\}}{P; R \cup \{l \to r'\}} \quad \text{if } r \to_{R,E} r'.$$

$$\text{COLLAPSE: } \frac{P; R \cup \{l \to r\}}{P \cup \{l' \approx r\}; R} \quad \text{if } l \to_{R,E}^{g \to d} l' \text{ for } g \to d \in R, \text{ and if } l \overset{*}{\leftrightarrow}_E g, \text{ then } r \succ_E d.$$

Above, $\succ_E$ is our E-compatible total reduction ordering on ground terms with the subterm property for a finite set of permutation equations E.

Figure 2 Ground completion modulo a finite set of permutation equations E.

applied eventually. If a derivation $P_0; R_0 \vdash P_1; R_1 \vdash \cdots$ is fair and $P_\infty = \emptyset$ (i.e. *non-failing*), then $CP_E(R_\infty)$ is a subset of $\bigcup_k P_k$ [5]. Since a finite permutation theory E has a finite complete unification algorithm [1], and $\succ_E$ is E-compatible with the subterm property, the following theorem is a direct adaptation of Theorem 18.4 in [20] and Theorem 3.21 in [5].

▶ **Theorem 23.** *Let $P_0; R_0 \vdash P_1; R_1 \vdash \cdots$ be a fair derivation such that P_0 is a finite set of equations with $R_0 = \emptyset$, and $P_\infty = \emptyset$. Then R_∞, E is convergent modulo E.*

5 Ground completion modulo a set of permutation equations

It is known that the word problem of ground theories[3] modulo E is decidable by using ground completion modulo E for $E = AC$, $AC \cup U$ (unit), $AC \cup I$ (idempotent), AG (abelian group theory), and undecidable for $E = A$ (associativity), $AC \cup D$ (distributivity), and G (group theory) (see [22] for details). We show that our ground completion modulo a finite set of permutation equations E always admits a finite ground convergent (modulo E) rewrite system, allowing us to provide a decision procedure for the word problem of ground theories modulo E. In this section, we denote by P a set of ground equations, R a set of ground rewrite rules, E a finite set of permutation equations, and by $\succ_E$ our E-compatible simplification ordering on terms that is E-total on ground terms.

Note that the DEDUCE inference rule in Figure 1 is no longer needed for our ground completion modulo E in Figure 2 because the inference steps by DEDUCE can be replaced by other simplification inference steps, especially by COLLAPSE in Figure 2. Furthermore, an encompassment ordering modulo E in Figure 1 is also no longer needed for the COLLAPSE inference rule in Figure 2 for the ground case. We write $P; R \vdash P'; R'$ to indicate that $P'; R'$ can be obtained from $P; R$ by application of an inference rule in Figure 2.

[3] By a ground theory, we mean an equational theory defined by a *finite* set of ground equations throughout this paper.

▶ **Lemma 24.** *If $P; R \vdash P'; R'$, then the congruence relations $\xleftrightarrow{*}_{E\cup P\cup R}$ and $\xleftrightarrow{*}_{E\cup P'\cup R'}$ on $T(\mathcal{F})$ are the same.*

Proof. We consider each application of an inference rule τ for $P; R \vdash P'; R'$. If τ is ORIENT, SIMPLIFY, DELETE, or COMPOSE, then the conclusion can be easily verified. If τ is COLLAPSE, then let $R = R'' \cup \{l \to r\}$, $P' = P \cup \{l' \approx r\}$, and $R' = R''$. Since $(P\cup R)-(P'\cup R') = \{l \to r\}$, we need to show that $l \xleftrightarrow{*}_{E\cup P'\cup R'} r$. As $l \xleftrightarrow{*}_E \hat{l} \to_{R'}^{g\to d} l' \leftrightarrow_{P'} r$ for some $g \to d \in R''$, we have $l \xleftrightarrow{*}_{E\cup P'\cup R'} r$. Conversely, since $(P'\cup R')-(P\cup R) = \{l' \approx r\}$, we also need to show that $l' \xleftrightarrow{*}_{E\cup P\cup R} r$. As $l' \leftarrow_R^{g\to d} \hat{l} \xleftrightarrow{*}_E l \to_R r$ for some $g \to d \in R''$, we have $l' \xleftrightarrow{*}_{E\cup P\cup R} r$. Thus, the conclusion follows. ◀

▶ **Definition 25.** Let $s = s[u\sigma] \leftrightarrow s[v\sigma] = t$ be a proof step with the equation (or rule) $u \approx v \in E \cup P \cup R$. The *complexity* of this proof step is defined as follows:

(i) $(\{s\}, \bot, t)$ if $u \approx v \in E$
(ii) $(\{s,t\}, \bot, \bot)$ if $u \approx v \in P$
(iii) $(\{s\}, u, t)$ if $u \to v \in R$
(iv) $(\{t\}, v, s)$ if $v \to u \in R$

Complexities of proof steps are lexicographically compared by $\succ_E^{mul}$ in the first component, and $\succ_E$ in the second and the third component, where $\bot$ is a new constant symbol and is assumed to be minimal (w.r.t. $\succ_E$). The *complexity of a proof* is the multiset of the complexities of its proof steps [5, 7]. The ordering on proofs, denoted by $\succ_{\mathcal{C}}$, is the multiset extension of the ordering on the complexities of proof steps. Since the multiset/lexicographic extension of a well-founded ordering is still well-founded and $\succ_E$ is well-founded, we may infer that $\succ_{\mathcal{C}}$ is well-founded. By a *ground proof* in $E \cup P \cup R$ of an equation $s \approx t$ with $s, t \in T(\mathcal{F})$, we mean a sequence of proof steps such that $t_0 = s, t_n = t$ and for all $t_i \in T(\mathcal{F})$, $0 < i \le n$, one of $t_{i-1} \leftrightarrow_E t_i$, $t_{i-1} \leftrightarrow_P t_i$, $t_{i-1} \to_R t_i$, $t_{i-1} \leftarrow_R t_i$ holds.

▶ **Lemma 26.** *If $P; R \vdash P'; R'$, then for any ground proof ρ in $E \cup P \cup R$ of an equation $s \approx t$, there is a ground proof ρ' in $E \cup P' \cup R'$ of the equation $s \approx t$ such that $\rho \succeq_{\mathcal{C}} \rho'$.*

Proof. We show that each equation in $(P \cup R) - (P' \cup R')$ has a smaller proof (w.r.t. $\succ_{\mathcal{C}}$) in $E \cup P' \cup R'$ by considering each case for $P; R \vdash P'; R'$.

(i) ORIENT: The proof $p \leftrightarrow_P q$ is transformed to the proof $p \to_{R'} q$. Since $\{(\{p,q\}, \bot, \bot)\} \succ_{\mathcal{C}} \{(\{p\}, p, q)\}$, the newer proof $p \to_{R'} q$ is smaller (w.r.t. $\succ_{\mathcal{C}}$) than the proof $p \leftrightarrow_P q$.

(ii) SIMPLIFY: The proof $p \leftrightarrow_P q$ is transformed to the proof $p \xleftrightarrow{*}_E \hat{p} \to_{R'} p' \leftrightarrow_{P'} q$. The newer proof is smaller (w.r.t. $\succ_{\mathcal{C}}$) because $p \leftrightarrow_P q$ with the complexity $\{(\{p,q\}, \bot, \bot)\}$ is bigger (w.r.t. $\succ_{\mathcal{C}}$) than all proof steps in $p \xleftrightarrow{*}_E \hat{p}$, $\hat{p} \to_{R'} p'$ and $p' \leftrightarrow_{P'} q$ in the first component.

(iii) DELETE: The proof $p \leftrightarrow_P q$ is transformed to the proof $p \xleftrightarrow{*}_E q$. The proof $p \leftrightarrow_P q$ with the complexity $\{(\{p,q\}, \bot, \bot)\}$ is bigger (w.r.t. $\succ_{\mathcal{C}}$) than all proof steps in $p \xleftrightarrow{*}_E q$ in the first component.

(iv) COMPOSE: The proof $l \to_R r$ is transformed to the proof $l \to_{R'} r' \leftarrow_{R'} \hat{r} \xleftrightarrow{*}_E r$. The newer proof is smaller (w.r.t. $\succ_{\mathcal{C}}$) because $l \to_R r$ with the complexity $\{(\{l\}, l, r)\}$ is bigger (w.r.t. $\succ_{\mathcal{C}}$) than (a) the proof step in $l \to_{R'} r'$ in the third component, (b) the proof step $r' \leftarrow_{R'} \hat{r}$ in the first component, and (c) all proof steps in $\hat{r} \xleftrightarrow{*}_E r$ in the first component.

(v) COLLAPSE: The proof $l \to_R r$ is transformed to the proof $l \xleftrightarrow{*}_E \hat{l} \to_{R'}^{g\to d} l' \leftrightarrow_{P'} r$ for some $g \to d \in R'$. The newer proof is smaller (w.r.t. $\succ_{\mathcal{C}}$) because $l \to_R r$ with the complexity $\{(\{l\}, l, r)\}$ is bigger (w.r.t. $\succ_{\mathcal{C}}$) than (a) all proof steps in $l \xleftrightarrow{*}_E \hat{l}$ in the second component, (b) the proof step $\hat{l} \to_{R'}^{g\to d} l'$ in the second (resp. third) component if $l \not\xleftrightarrow{*}_E g$ (resp. $l \xleftrightarrow{*}_E g$), and (c) the proof step $l' \leftrightarrow_{P'} r$ in the first component. ◀

Note that if $P_0; R_0 \vdash P_1; R_1 \vdash \cdots$ is a fair derivation, then $P_\infty = \emptyset$ (i.e. non-failing) because $\succ_E$ is E-total on ground terms.

▶ **Theorem 27.** *Let $P_0; R_0 \vdash P_1; R_1 \vdash \cdots$ be a fair derivation such that P_0 is a finite set of ground equations with $R_0 = \emptyset$. Then the set of persisting rules R_∞ is finite and R_∞, E is ground convergent modulo E.*

Proof. Suppose that $P_0; R_0 \vdash P_1; R_1 \vdash \cdots$ is a fair derivation such that P_0 is a finite set of ground equations with $R_0 = \emptyset$. We first define a simple measure of a state $P_k; R_k$ as the multiset $\{\{\{s,t\}\} \mid s \approx t \in P_k\} \cup \{\{\{s\},\{t\}\} \mid s \to t \in R_k\}$ (cf. [7]). Two states are compared by these measures using the threefold multiset extension of $\succ_E$. It is easy to see that any application of an inference rule for a transition $P_k; R_k \vdash P_{k+1}; R_{k+1}$ reduces this measure. Since the multiset extension of a well-founded ordering is still well-founded and $\succ_E$ is well-founded, we may infer that any fair derivation starting from $P_0; R_0$ is finite. Therefore, R_∞ is finite with $P_\infty = \emptyset$. Since $l \succ_E r$ for all rules $l \to r \in R_\infty$, R_∞/E is also terminating.

Now it remains to show that R_∞, E is ground Church-Rosser modulo E. We show that all minimal (w.r.t. $\succ_{\mathcal{C}}$) proofs in $E \cup R_\infty$ are rewrite proofs modulo E.

Suppose that a proof is a minimal proof but not a rewrite proof modulo E. Then it should contain either a peak (or a cliff) that is a proper overlap (cf. [5]). (Note that every peak or cliff that is a non-overlap or a variable overlap can be replaced by a rewrite proof modulo E (see pp. 47–50 in [5]), which is smaller (w.r.t. $\succ_{\mathcal{C}}$) than the original peak or cliff, so this is not the case.)

Now consider such a peak $t_1 \leftarrow_{R_\infty} t \rightarrow_{R_\infty,E} t_2$ that is a proper overlap. (Since $EXT_E(R)$ is empty, we do not need to consider a cliff that is a proper overlap.) By the *Extended Critical Pair Lemma* [6,16], it can be replaced by a proof $t_1 \overset{*}{\leftrightarrow}_E t' \leftrightarrow_{CP_E(R_\infty)} t'' \overset{*}{\leftrightarrow}_E t_2$. Since $CP_E(R_\infty) \subseteq \bigcup_k P_k$ by fairness of the derivation, there is a ground proof $t_1 \overset{*}{\leftrightarrow}_E t' \leftrightarrow_{P_k} t'' \overset{*}{\leftrightarrow}_E t_2$ for some k. We name this proof as ρ. We see that the ground proof ρ in $E \cup P_k$ is strictly smaller (w.r.t. $\succ_{\mathcal{C}}$) than the original peak $t_1 \leftarrow_{R_\infty} t \rightarrow_{R_\infty,E} t_2$. Since $P_\infty = \emptyset$, there is a ground proof ρ' in $E \cup R_\infty$ such that $\rho \succeq_{\mathcal{C}} \rho'$ by Lemma 26. Now we may infer that ρ' is strictly smaller (w.r.t. $\succ_{\mathcal{C}}$) than the original peak $t_1 \leftarrow_{R_\infty} t \rightarrow_{R_\infty,E} t_2$, which is the required contradiction. ◀

By Theorem 27, the rewrite system R_∞ constructed from a fair derivation $P_0; R_0 \vdash P_1; R_1 \vdash \cdots$ may serve as a decision procedure for the word problem of ground theories P_0 modulo E.

▶ **Corollary 28.** *Given a finite set of permutation equations E, the word problem of ground theories modulo E is decidable.*

The following example is a variant of the *reachability problem* [32] modulo a finite set of permutation equations E.

▶ **Example 29.** Consider the following set of permutation equations:

$$\begin{aligned} E = \{ & f(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8,x_9,x_{10}) \approx f(x_2,x_1,x_3,x_4,x_5,x_6,x_7,x_8,x_9,x_{10}),\\ & f(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8,x_9,x_{10}) \approx f(x_2,x_3,x_4,x_5,x_1,x_6,x_7,x_8,x_9,x_{10}),\\ & f(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8,x_9,x_{10}) \approx f(x_1,x_2,x_3,x_4,x_5,x_7,x_6,x_8,x_9,x_{10}),\\ & f(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8,x_9,x_{10}) \approx f(x_1,x_2,x_3,x_4,x_5,x_7,x_8,x_9,x_{10},x_6)\}. \end{aligned}$$

In this example, we may view each variable x_i as a vertex in a graph with ten vertices, where each vertex will be assigned to one of three colors: blue (b), red (r), and white (w). Therefore, each ground term $f(c_1,c_2,c_3,c_4,c_5,c_6,c_7,c_8,c_9,c_{10})$

with $c_i = b, r$, or w represents a certain coloring of this graph. There is a transition function with a function symbol $g \notin \mathcal{F}_E$, which transforms one coloring to another coloring of the graph. We assign the precedence as $g \succ_{\mathcal{F}} f \succ_{\mathcal{F}} b \succ_{\mathcal{F}} r \succ_{\mathcal{F}} w$. We see that $\prod[E] = \{(1\,2), (1\,2\,3\,4\,5), (6\,7), (6\,7\,8\,9\,10)\}$, which means that $f(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}) \approx_E f(x_{\rho(1)}, x_{\rho(2)}, x_{\rho(3)}, x_{\rho(4)}, x_{\rho(5)}, x_6, x_7, x_8, x_9, x_{10})$ for any permutation ρ on the set $\{1, 2, 3, 4, 5\}$ and $f(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}) \approx_E f(x_1, x_2, x_3, x_4, x_5, x_{\pi(6)}, x_{\pi(7)}, x_{\pi(8)}, x_{\pi(9)}, x_{\pi(10)})$ for any permutation π on the set $\{6, 7, 8, 9, 10\}$ (see Thereom 2). Therefore, ten vertices are partitioned into two equivalence classes. We may view them as two components, i.e. $\{x_1, x_2, x_3, x_4, x_5\}$ and $\{x_6, x_7, x_8, x_9, x_{10}\}$, where the order of a coloring does not matter in each component. For example, $f(r, r, b, b, b, w, w, b, b, b) \approx_E f(b, b, r, b, r, w, b, b, b, w)$. We start with the following set of ground equations:[4]

1. $g(f(b, b, b, b, b, b, b, b, b, b)) \approx f(r, b, b, b, b, b, b, b, b, b)$
2. $g(f(b, b, r, b, b, b, b, b, b, b)) \approx f(r, b, b, b, b, r, b, b, b, b)$
3. $f(r, b, b, b, b, b, b, b, b, b) \approx f(w, b, b, b, b, b, b, b, b, b)$
4. $f(r, b, b, b, b, r, b, b, b, b) \approx f(w, b, b, b, b, w, b, b, b, b)$
5. $g(f(w, b, b, b, b, w, b, b, b, b)) \approx f(w, w, b, b, b, w, w, b, b, b)$
6. $f(w, w, b, b, b, w, w, b, b, b) \approx f(r, r, b, b, b, r, r, b, b, b))$
7. $g(f(r, b, b, b, r, r, b, b, b, r)) \approx f(r, r, r, r, r, r, r, r, r, r)$

The problem is to determine if there is some i such that $g^i(f(b, b, b, b, b, b, b, b, b, b)) = f(r, r, r, r, r, r, r, r, r, r)$, where $f(b, b, b, b, b, b, b, b, b, b)$ is the initial state and $f(r, r, r, r, r, r, r, r, r, r)$ is the target state. (Here $g^i(t)$ denotes that the function symbol g is applied to term $g^{i-1}(t)$ with $g^0(t)$ denoting t.) Now ground completion modulo E works (roughly) as follows:

$1(a)$.	$g(f(b, b, b, b, b, b, b, b, b, b)) \rightarrow f(r, b, b, b, b, b, b, b, b, b)$	ORIENT 1
$2(a)$.	$g(f(b, b, r, b, b, b, b, b, b, b)) \rightarrow f(r, b, b, b, b, r, b, b, b, b)$	ORIENT 2
$3(a)$.	$f(r, b, b, b, b, b, b, b, b, b) \rightarrow f(w, b, b, b, b, b, b, b, b, b)$	ORIENT 3
$1(b)$.	$g(f(b, b, b, b, b, b, b, b, b, b)) \rightarrow f(w, b, b, b, b, b, b, b, b, b)$	COMPOSE 1(a), 3(a)
$2(b)$.	$g(f(w, b, b, b, b, b, b, b, b, b)) \approx f(r, b, b, b, b, r, b, b, b, b)$	COLLAPSE 2(a), 3(a)
$2(c)$.	$g(f(w, b, b, b, b, b, b, b, b, b)) \rightarrow f(r, b, b, b, b, r, b, b, b, b)$	ORIENT 2(b)
$4(a)$.	$f(r, b, b, b, b, r, b, b, b, b) \rightarrow f(w, b, b, b, b, w, b, b, b, b)$	ORIENT 4
$2(d)$.	$g(f(w, b, b, b, b, b, b, b, b, b)) \rightarrow f(w, b, b, b, b, w, b, b, b, b)$	COMPOSE 2(c), 4(a)
$5(a)$.	$g(f(w, b, b, b, b, w, b, b, b, b)) \rightarrow f(w, w, b, b, b, w, w, b, b, b)$	ORIENT 5
$6(a)$.	$f(r, r, b, b, b, r, r, b, b, b)) \rightarrow f(w, w, b, b, b, w, w, b, b, b)$	ORIENT 6
$7(a)$.	$g(f(w, w, b, b, b, w, w, b, b, b)) \approx f(r, r, r, r, r, r, r, r, r, r)$	SIMPLIFY 6(a), 7
$7(b)$.	$g(f(w, w, b, b, b, w, w, b, b, b)) \rightarrow f(r, r, r, r, r, r, r, r, r, r)$	ORIENT 7(a)

We eventually obtain the ground convergent (modulo E) rewrite system R_∞ (with $P_\infty = \emptyset$), which consists of the rewrite rules $1(b), 2(d), 3(a), 4(a), 5(a), 6(a)$, and $7(b)$. (It is easy to see that the remaining rules $1(a), 2(a)$ and $2(c)$, and the remaining equations $2(b)$ and $7(a)$ are not persistent.) Now we see that $g^4(f(b, b, b, b, b, b, b, b, b, b)) \rightarrow_{R_\infty, E} g^3(f(w, b, b, b, b, b, b, b, b, b)) \rightarrow_{R_\infty, E} g^2(f(w, b, b, b, b, w, b, b, b, b)) \rightarrow_{R_\infty, E} g(f(w, w, b, b, b, w, w, b, b, b)) \rightarrow_{R_\infty, E} f(r, r, r, r, r, r, r, r, r, r)$. Therefore, we may interpret that $f(r, r, r, r, r, r, r,$

[4] We may consider the additional state transitions using a transformation function with symbol g, or partition vertices in a different way with a different number of vertices using a different set of permutation equations.

$r, r, r)$ is reachable from $f(b, b, b, b, b, b, b, b, b, b)$ by means of iterative applications of the state transition function with symbol g. Note that if $g(f(c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}))$ is a normal form w.r.t. R_∞, E, then we may also interpret that $f(c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10})$ is a fixed state (or a stable state) and cannot be further transformed to another state by an application of the state transition function with symbol g.

6 Conclusion

We have presented an RPO-based E-compatible simplification ordering $\succ_E$ on terms that is E-total on ground terms for a finite set of permutation equations E. Since permutation groups naturally arise in sets of permutation equations, we have used permutation group theory for $\succ_E$, especially permutation group actions and their associated orbits. Our ordering is simple and can be adapted from the standard RPO widely used for rewrite systems and theorem proving. Also, the computation of orbits in permutation groups can be done efficiently using the existing permutation group algorithms [29] and software tools (e.g. GAP [12]). We have shown that given two terms s and t, we can determine whether $s \succ_E t$ in polynomial time.

Our ordering $\succ_E$ provides a simple termination criterion for R, E (resp. R/E), that is, R, E (resp. R/E) is terminating if $l \succ_E r$ for all rules $l \to r \in R$. We have used $\succ_E$ for a completion and ground completion procedure for R, E. Furthermore, our ground completion modulo E always terminates with a finite ground convergent (modulo E) rewrite system, which allows us to provide a decision procedure for the word problem of ground theories modulo E. (It is also an interesting question whether other ground completion approaches and formalisms (e.g. the *abstract completion* of [14]) can be extended for ground completion modulo E for a finite set of permutation equations E using $\succ_E$.)

Since permutations and combinations are widely used in mathematics and many fields of science including computer science, developing applications of term rewriting and equational theorem proving [19] with built-in permutation equations is one of the promising future directions of the research discussed in this paper. For example, one may consider reachability problems modulo E and its applications to hardware and software verification using our ordering and rewriting modulo E approach for a finite set of permutation equations E.

References

1 Jürgen Avenhaus. Efficient Algorithms for Computing Modulo Permutation Theories. In David Basin and Michaël Rusinowitch, editors, *Automated Reasoning – IJCAR 2004, Cork, Ireland, July 4–8*, pages 415–429, Berlin, Heidelberg, 2004. Springer.

2 Jürgen Avenhaus and David A. Plaisted. General algorithms for permutations in equational inference. *Journal of Automated Reasoning*, 26(3):223–268, 2001.

3 Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, Cambridge, UK, 1998.

4 Franz Baader and Wayne Snyder. Unification Theory. In *Handbook of Automated Reasoning*, Volume I, chapter 8, pages 445–532. Elsevier, Amsterdam, 2001.

5 Leo Bachmair. *Canonical Equational Proofs*. Birkhäuser, Boston, 1991.

6 Leo Bachmair and Nachum Dershowitz. Completion for rewriting modulo a congruence. *Theoretical Computer Science*, 67(2):173–201, 1989.

7 Leo Bachmair, Ashish Tiwari, and Laurent Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*, 31(2):129–168, 2003.

8 Frédéric Blanqui. Rewriting Modulo in Deduction Modulo. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications*, pages 395–409, Berlin, Heidelberg, 2003. Springer.

9 Frédéric Blanqui. Termination of rewrite relations on λ-terms based on Girard's notion of reducibility. *Theoretical Computer Science*, 611:50–86, 2016.

10 Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.
11 Nachum Dershowitz and David A. Plaisted. Rewriting. In *Handbook of Automated Reasoning*, Volume I, chapter 9, pages 535–610. Elsevier, Amsterdam, 2001.
12 GAP Group. *Groups, Algorithms, Programming, Version 4.8*, 2016. http://www.gap-system.org.
13 Sumanta Ghosh and Piyush P. Kurur. *Permutation Groups and the Graph Isomorphism Problem*, pages 183–202. Springer, Cham, 2014.
14 Nao Hirokawa, Aart Middeldorp, Christian Sternagel, and Sarah Winkler. Abstract Completion, Formalized. *Logical Methods in Computer Science*, 15:19:1–19:42, 2019.
15 Thomas W. Hungerford. *Algebra*. Springer, New York, NY, 1980.
16 Jean-Pierre Jouannaud and Hélène Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal on Computing*, 15(4):1155–1194, 1986.
17 Deepak Kapur, Paliath Narendran, and G. Sivakumar. A path ordering for proving termination of term rewriting systems. In Hartmut Ehrig, Christiane Floyd, Maurice Nivat, and James Thatcher, editors, *Mathematical Foundations of Software Development*, pages 173–187, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.
18 Deepak Kapur and G. Sivakumar. Proving Associative-Commutative Termination Using RPO-Compatible Orderings. In R. Caferra and Gernot Salzer, editors, *Automated Deduction in Classical and Non-Classical Logics*, pages 39–61, Berlin, Heidelberg, 2000. Springer.
19 Dohan Kim and Christopher Lynch. Equational Theorem Proving Modulo. In *Automated Deduction – CADE 28: The 28th International Conference on Automated Deduction, Carnegie Mellon University, Pittsburgh, PA (Virtual Conference), July 11-16, to appear*. Springer, 2021.
20 Claude Kirchner and Helene Kirchner. Rewriting, Solving, Proving, 1999. Preliminary version of a book: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.144.5349`.
21 Donald. E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pages 342–376. Springer, Berlin, Heidelberg, 1983.
22 Claude Marché. Normalized rewriting: an alternative to rewriting modulo a set of equations. *Journal of Symbolic Computation*, 21(3):253–288, 1996.
23 Paliath Narendran and Michaël Rusinowitch. Any ground associative-commutative theory has a finite canonical system. In R. V. Book, editor, *Rewriting Techniques and Applications*, pages 423–434, Berlin, Heidelberg, 1991. Springer.
24 Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In *Handbook of Automated Reasoning*, Volume I, chapter 7, pages 371–443. Elsevier, Amsterdam, 2001.
25 Gerald E Peterson and Mark E Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM (JACM)*, 28(2):233–264, 1981.
26 Albert Rubio. Theorem Proving modulo Associativity. In H. Kleine Büning, editor, *Computer Science Logic*, pages 452–467, Berlin, Heidelberg, 1996. Springer.
27 Albert Rubio. A Fully Syntactic AC-RPO. *Inf. Comput.*, 178(2):515–533, 2002.
28 Albert Rubio and Robert Nieuwenhuis. A total AC-compatible ordering based on RPO. *Theoretical Computer Science*, 142(2):209–227, 1995.
29 Charles C Sims. *Computation with finitely presented groups*, volume Vol. 48. Cambridge University Press, Cambridge, UK, 1994.
30 Wayne Snyder. On the complexity of recursive path orderings. *Information Processing Letters*, 46(5):257–262, 1993.
31 Joachim Steinbach. On the complexity of simplification orderings. Technical Report Technical Report SR-93-18 (SFB), SEKI University of Kaiserslautern, 1993.
32 Christian Sternagel and Akihisa Yamada. Reachability analysis for termination and confluence of rewriting. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 262–278, Cham, 2019. Springer International Publishing.

Some Axioms for Mathematics

Frédéric Blanqui
Université Paris-Saclay, ENS Paris-Saclay, LMF, CNRS, Inria, France

Gilles Dowek
Université Paris-Saclay, ENS Paris-Saclay, LMF, CNRS, Inria, France

Émilie Grienenberger
Université Paris-Saclay, ENS Paris-Saclay, LMF, CNRS, Inria, France

Gabriel Hondet
Université Paris-Saclay, ENS Paris-Saclay, LMF, CNRS, Inria, France

François Thiré
Nomadic Labs, Paris, France

Abstract

The $\lambda\Pi$-calculus modulo theory is a logical framework in which many logical systems can be expressed as theories. We present such a theory, the theory $\mathcal{U}$, where proofs of several logical systems can be expressed. Moreover, we identify a sub-theory of $\mathcal{U}$ corresponding to each of these systems, and prove that, when a proof in $\mathcal{U}$ uses only symbols of a sub-theory, then it is a proof in that sub-theory.

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

Keywords and phrases logical framework, axiomatic theory, dependent types, rewriting, interoperabilty

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.20

Acknowledgements The authors want to thank Michael Färber, César Muñoz, Thiago Felicissimo, and Makarius Wenzel for helpful remarks on a first version of this paper.

1 Introduction

The $\lambda\Pi$-calculus modulo theory ($\lambda\Pi/\equiv$) [13], implemented in the system Dedukti [3, 29], is a logical framework, that is a framework to define theories. It generalizes some previously proposed frameworks: Predicate logic [28], λ-Prolog [32], Isabelle [34], the Edinburgh logical framework [27], also called the $\lambda\Pi$-calculus, Deduction modulo theory [17, 18], Pure type systems [6, 39], and Ecumenical logic [36, 16, 35, 25]. It is thus an extension of Predicate logic that provides the possibility for all symbols to bind variables, a syntax for proof-terms, a notion of computation, a notion of proof reduction for axiomatic theories, and the possibility to express both constructive and classical proofs.

$\lambda\Pi/\equiv$ enables to express all theories that can be expressed in Predicate logic, such as geometry, arithmetic, and set theory, but also Simple type theory [10] and the Calculus of constructions [12], that are less easy to define in Predicate logic.

We present a theory in $\lambda\Pi/\equiv$, the theory $\mathcal{U}$, where all proofs of Minimal, Constructive, and Ecumenical predicate logic; Minimal, Constructive, and Ecumenical simple type theory; Simple type theory with predicate subtyping, prenex predicative polymorphism, or both; the Calculus of constructions, and the Calculus of constructions with prenex predicative polymorphism can be expressed. This theory is therefore a candidate for a universal theory, where proofs developed in implementations of Classical predicate logic (such as automated theorem proving systems, SMT solvers, etc.), Classical simple type theory (such as HOL 4, HOL Light, Isabelle/HOL, etc.), the Calculus of constructions (such as Coq, Matita, Lean, etc.), and Simple type theory with predicate subtyping and prenex polymorphism (such as PVS), can be expressed.

Moreover, the proofs of the theory $\mathcal{U}$ can be classified as proofs in Minimal predicate logic, Constructive Predicate logic, etc. just by identifying the axioms they use, akin to proofs in geometry that can be classified as proofs in Euclidean, hyperbolic, elliptic, neutral, etc. geometries. More precisely, we identify sub-theories of the theory $\mathcal{U}$ that correspond to each of these theories, and we prove that when a proof in $\mathcal{U}$ uses only symbols of a sub-theory, then it is a proof in that sub-theory.

In Section 2, we recall the definition of $\lambda\Pi/\equiv$ and of a theory. In Section 3, we introduce the theory $\mathcal{U}$ step by step. In Section 4, we provide a general theorem on sub-theories in $\lambda\Pi/\equiv$, and prove that every fragment of $\mathcal{U}$, including $\mathcal{U}$ itself, is indeed a theory, that is, it is defined by a confluent and type-preserving rewriting systems. Finally, in Section 5, we detail the sub-theories of $\mathcal{U}$ that correspond to the above mentioned systems.

2 The $\lambda\Pi$-calculus modulo theory

$\lambda\Pi/\equiv$ is an extension of the Edinburgh logical framework [27] with a primitive notion of computation defined with rewriting rules [14, 38].

The terms are those of the Edinburgh logical framework

$$t, u = c \mid x \mid \texttt{TYPE} \mid \texttt{KIND} \mid \Pi x : t, u \mid \lambda x : t, u \mid t\ u$$

where c belongs to a finite or infinite set of constants $\mathcal{C}$ and x to an infinite set $\mathcal{V}$ of variables. The terms TYPE and KIND are called sorts. The term $\Pi x : t, u$ is called a product. It is dependent if the variable x occurs free in u. Otherwise, it is simply written $t \to u$. Terms are also often written A, B, etc. The set of constants of a term t is written $\mathit{const}(t)$.

A rewriting rule is a pair of terms $\ell \hookrightarrow r$, such that $\ell = c\ \ell_1\ \ldots\ \ell_n$, where c is a constant. If $\mathcal{R}$ is a set of rewriting rules, we write $\hookrightarrow_{\mathcal{R}}$ for the smallest relation closed by term constructors and substitution containing $\mathcal{R}$, $\hookrightarrow_\beta$ for the usual β-reduction, $\hookrightarrow_{\beta\mathcal{R}}$ for $\hookrightarrow_\beta \cup \hookrightarrow_{\mathcal{R}}$, and $\equiv_{\beta\mathcal{R}}$ for the smallest equivalence relation containing $\hookrightarrow_{\beta\mathcal{R}}$.

The typing rules of $\lambda\Pi/\equiv$ are given in Figure 1. The difference with the rules of the Edinburgh logical framework is that, in the rule (conv), types are identified modulo $\equiv_{\beta\mathcal{R}}$ instead of just $\equiv_\beta$. In a typing judgement $\Gamma \vdash_{\Sigma,\mathcal{R}} t : A$, the term t is given the type A with respect to three parameters: a signature Σ that assigns a type to the constants of t, a context Γ that assigns a type to the free variables of t, and a set of rewriting rules $\mathcal{R}$. A context Γ is a list of declarations $x_1 : B_1, \ldots, x_m : B_m$ formed with a variable and a term. A signature Σ is a list of declarations $c_1 : A_1, \ldots, c_n : A_n$ formed with a constant and a closed term, that is a term term with no free variables. This is why the rule (const) requires no context for typing A. We write $|\Sigma|$ for the set $\{c_1, \ldots, c_n\}$, and $\Lambda(\Sigma)$ for the set of terms t such that $\mathit{const}(t) \subseteq |\Sigma|$. We say that a rewriting rule $\ell \hookrightarrow r$ is in $\Lambda(\Sigma)$ if ℓ and r are, and a context $x_1 : B_1, \ldots, x_m : B_m$ is in $\Lambda(\Sigma)$ if $B_1, \ldots, B_m$ are. It is often convenient to group constant declarations and rules into small clusters, called "axioms".

A relation $\hookrightarrow$ preserves typing in $\Sigma, \mathcal{R}$ if, for all contexts Γ and terms t, u and A of $\Lambda(\Sigma)$, if $\Gamma \vdash_{\Sigma,\mathcal{R}} t : A$ and $t \hookrightarrow u$, then $\Gamma \vdash_{\Sigma,\mathcal{R}} u : A$. The relation $\hookrightarrow_\beta$ preserves typing as soon as $\hookrightarrow_{\beta\mathcal{R}}$ is confluent (see for instance [7]) for, in this case, the product is injective modulo $\equiv_{\beta\mathcal{R}}$: $\Pi x : A, B \equiv_{\beta\mathcal{R}} \Pi x : A', B'$ iff $A \equiv_{\beta\mathcal{R}} A'$ and $B \equiv_{\beta\mathcal{R}} B'$. The relation $\hookrightarrow_{\mathcal{R}}$ preserves typing if every rewriting rule $\ell \hookrightarrow r$ preserves typing, that is: for all contexts Γ, substitutions θ and terms A of $\Lambda(\Sigma)$, if $\Gamma \vdash_{\Sigma,\mathcal{R}} \theta l : A$ then $\Gamma \vdash_{\Sigma,\mathcal{R}} \theta r : A$.

$$\frac{}{\vdash_{\Sigma,\mathcal{R}} [\,]\ \text{well-formed}}\ \text{(empty)}$$

$$\frac{\Gamma \vdash_{\Sigma,\mathcal{R}} A : s}{\vdash_{\Sigma,\mathcal{R}} \Gamma, x : A\ \text{well-formed}}\ \text{(decl)}$$

$$\frac{\vdash_{\Sigma,\mathcal{R}} \Gamma\ \text{well-formed}}{\Gamma \vdash_{\Sigma,\mathcal{R}} \mathtt{TYPE} : \mathtt{KIND}}\ \text{(sort)}$$

$$\frac{\vdash_{\Sigma,\mathcal{R}} \Gamma\ \text{well-formed} \quad \vdash_{\Sigma,\mathcal{R}} A : s}{\Gamma \vdash_{\Sigma,\mathcal{R}} c : A}\ \text{(const)} \quad c : A \in \Sigma$$

$$\frac{\vdash_{\Sigma,\mathcal{R}} \Gamma\ \text{well-formed}}{\Gamma \vdash_{\Sigma,\mathcal{R}} x : A}\ \text{(var)} \quad x : A \in \Gamma$$

$$\frac{\Gamma \vdash_{\Sigma,\mathcal{R}} A : \mathtt{TYPE} \quad \Gamma, x : A \vdash_{\Sigma,\mathcal{R}} B : s}{\Gamma \vdash_{\Sigma,\mathcal{R}} \Pi x : A, B : s}\ \text{(prod)}$$

$$\frac{\Gamma \vdash_{\Sigma,\mathcal{R}} A : \mathtt{TYPE} \quad \Gamma, x : A \vdash_{\Sigma,\mathcal{R}} B : s \quad \Gamma, x : A \vdash_{\Sigma,\mathcal{R}} t : B}{\Gamma \vdash_{\Sigma,\mathcal{R}} \lambda x : A, t : \Pi x : A, B}\ \text{(abs)}$$

$$\frac{\Gamma \vdash_{\Sigma,\mathcal{R}} t : \Pi x : A, B \quad \Gamma \vdash_{\Sigma,\mathcal{R}} u : A}{\Gamma \vdash_{\Sigma,\mathcal{R}} t\ u : (u/x)B}\ \text{(app)}$$

$$\frac{\Gamma \vdash_{\Sigma,\mathcal{R}} t : A \quad \Gamma \vdash_{\Sigma,\mathcal{R}} B : s}{\Gamma \vdash_{\Sigma,\mathcal{R}} t : B}\ \text{(conv)} \quad A \equiv_{\beta\mathcal{R}} B$$

Figure 1 Typing rules of $\lambda\Pi/\equiv$ with signature Σ and rewriting rules $\mathcal{R}$.

Although typing is defined with arbitrary signatures Σ and sets of rewriting rules $\mathcal{R}$, we are only interested in sets $\mathcal{R}$ verifying some confluence and type-preservation properties.

▶ **Definition 1** (System, theory)**.** *A system is a pair $\Sigma, \mathcal{R}$ such that each rule of $\mathcal{R}$ is in $\Lambda(\Sigma)$. It is a theory if $\hookrightarrow_{\beta\mathcal{R}}$ is confluent on $\Lambda(\Sigma)$, and every rule of $\mathcal{R}$ preserves typing in $\Sigma, \mathcal{R}$.*

Therefore, in a theory, $\hookrightarrow_{\beta\mathcal{R}}$ preserves typing since $\hookrightarrow_{\beta}$ preserves typing (for $\hookrightarrow_{\beta\mathcal{R}}$ is confluent) and $\hookrightarrow_{\mathcal{R}}$ preserves typing (for every rule preserves typing). We recall two other basic properties of $\lambda\Pi/\equiv$ we will use in Theorem 7:

▶ **Lemma 2.** *If $\Gamma \vdash_{\Sigma,\mathcal{R}} t : A$, then either $A = \mathtt{KIND}$ or $\Gamma \vdash_{\Sigma,\mathcal{R}} A : s$ for some sort s.*
If $\Gamma \vdash_{\Sigma,\mathcal{R}} \Pi x : A, B : s$, then $\Gamma \vdash_{\Sigma,\mathcal{R}} A : \mathtt{TYPE}$.

3 The theory $\mathcal{U}$

Object-terms

The notions of term, proposition, and proof are not primitive in $\lambda\Pi/\equiv$. The first axioms of the theory $\mathcal{U}$ introduce these notions. We first define a notion analogous to the Predicate logic notion of term, to express the objects the theory speaks about, such as the natural numbers. As all expressions in $\lambda\Pi/\equiv$ are called "terms", we shall call these expressions "object-terms", to distinguish them from the other terms.

The easiest way to build the notion of object-term in $\lambda\Pi/\equiv$ would be to declare a constant I of type $\mathtt{TYPE}$ and constants of type $I \to \ldots \to I \to I$ for the function symbols, for instance a constant 0 of type I and a constant *succ* of type $I \to I$. The object-terms, for instance

$(succ\ (succ\ 0))$ and $(succ\ x)$, would then just be $\lambda\Pi/\equiv$ terms of type I and, in an object-term, the variables would be $\lambda\Pi/\equiv$ variables of type I. If we wanted to have object-terms of several sorts, like in Many-sorted predicate logic, we could just declare several constants I_1, I_2, ..., I_n of type TYPE. But these sorts would be mixed with the other terms of type TYPE, which we will introduce later. Instead, we declare a constant *Set* of type TYPE, a constant ι of type *Set*, and a constant *El* to embed the terms of type *Set* into terms of type TYPE

$$Set : \texttt{TYPE} \qquad (Set\text{-decl})$$
$$\iota : Set \qquad (\iota\text{-decl})$$
$$El : Set \to \texttt{TYPE} \qquad (El\text{-decl})$$

so that the symbol I can be replaced with the term $El\ \iota$. If we want to have object-terms of several sorts, we declare several constants ι_1, ι_2, etc. of type *Set*. The types of object-terms then have the form $El\ A$ and are distinguished among the other terms of type TYPE.

Assigning the type $Set \to \texttt{TYPE}$ to the constant *El* uses the fact that $\lambda\Pi/\equiv$ supports dependent types.

Propositions

Just like $\lambda\Pi/\equiv$ does not contain a primitive notion of object-term, it does not contain a primitive notion of proposition, but tools to define this notion. To do so, in the theory $\mathcal{U}$, we declare a constant *Prop* of type TYPE

$$Prop : \texttt{TYPE} \qquad (Prop\text{-decl})$$

and predicate symbols are then just constants of type $El\ \iota \to \ldots \to El\ \iota \to Prop$. Propositions are then $\lambda\Pi/\equiv$ terms of type *Prop*.

Implication

In the theory $\mathcal{U}$, we then declare a constant for implication

$$\Rightarrow : Prop \to Prop \to Prop \quad \text{(written infix)} \qquad (\Rightarrow\text{-decl})$$

Proofs

Predicate logic defines a language for terms and propositions, but proofs have to be defined in a second step, for instance as derivations in natural deduction, sequent calculus, etc. These derivations, like object-terms and propositions, are trees. Therefore, they can be represented as $\lambda\Pi/\equiv$ terms.

Using the Brouwer-Heyting-Kolmogorov interpretation, a proof of the proposition $A \Rightarrow B$ should be a $\lambda\Pi/\equiv$ term expressing a function mapping proofs of A to proofs of B. Then, using the Curry-de Bruijn-Howard correspondence, the type of this term should be the proposition $A \Rightarrow B$ itself. But, this is not possible in the theory $\mathcal{U}$ yet, as the proposition $A \Rightarrow B$ has the type *Prop*, and not the type TYPE. So we introduce an embedding *Prf* of propositions into types, mapping each proposition A to the type $Prf\ A$ of its proofs

$$Prf : Prop \to \texttt{TYPE} \qquad (Prf\text{-decl})$$

Note that this embedding is not surjective. In particular *Set*, $El\ \iota$, and *Prop* are not types of proofs. So, there are more types than propositions, and propositions and types are not fully identified.

According to the Brouwer-Heyting-Kolmogorov interpretation, a proof of $A \Rightarrow A$ is a $\lambda\Pi/\equiv$ term expressing a function mapping proofs of A to proofs of A. In particular, the identity function $\lambda x : \mathit{Prf}\ A, x$ mapping each proof of A to itself is a proof of $A \Rightarrow A$. According to the Curry-de Bruijn-Howard correspondence, this term should have the type $\mathit{Prf}\ (A \Rightarrow A)$, but it has the type $\mathit{Prf}\ A \to \mathit{Prf}\ A$. So, the types $\mathit{Prf}\ (A \Rightarrow A)$ and $\mathit{Prf}\ A \to \mathit{Prf}\ A$ must be identified. To do so, we use the fact that $\lambda\Pi/\equiv$ allows the declaration of rewriting rules, so that $\mathit{Prf}\ (A \Rightarrow A)$ rewrites to $\mathit{Prf}\ A \to \mathit{Prf}\ A$

$$\mathit{Prf}\ (x \Rightarrow y) \hookrightarrow \mathit{Prf}\ x \to \mathit{Prf}\ y \qquad (\Rightarrow\text{-red})$$

In the theory $\mathcal{U}$, the Brouwer-Heyting-Kolmogorov interpretation of proofs for implication is made explicit: it is the rule ($\Rightarrow$-red).

Universal quantification

Unlike implication, the universal quantifier binds a variable. Thus, we express the proposition $\forall z\ A$ as the proposition $\forall\ (\lambda z : \mathit{El}\ \iota, A)$ [10, 32, 34, 27], yielding the type $(\mathit{El}\ \iota \to \mathit{Prop}) \to \mathit{Prop}$ for the constant $\forall$ itself. But, we want to allow quantification over variables of any type $\mathit{El}\ B$, for B of type Set. Thus, we generalize this type to

$$\forall : \Pi x : \mathit{Set}, (\mathit{El}\ x \to \mathit{Prop}) \to \mathit{Prop} \qquad (\forall\text{-decl})$$

and we write $\forall\ \iota\ (\lambda z : \mathit{El}\ \iota, A)$ for the proposition $\forall z\ A$.

Just like for the implication, we declare a rewriting rule expressing that the type of the proofs of the proposition $\forall\ x\ p$ is the type of functions mapping each z of type $\mathit{El}\ x$ to a proof of $p\ z$

$$\mathit{Prf}\ (\forall\ x\ p) \hookrightarrow \Pi z : \mathit{El}\ x, \mathit{Prf}\ (p\ z) \qquad (\forall\text{-red})$$

Again, the Brouwer-Heyting-Kolmogorov interpretation of proofs for the universal quantifier is made explicit: it is this rule ($\forall$-red).

Other constructive connectives and quantifiers

We define the other connectives and quantifiers, *à la* Russell, for instance $\mathit{Prf}\ (x \wedge y)$ as $\Pi z : \mathit{Prop}, (\mathit{Prf}\ x \to \mathit{Prf}\ y \to \mathit{Prf}\ z) \to \mathit{Prf}\ z$. In this definition, we do not use the quantifier $\forall$ of the theory $\mathcal{U}$ (so far, in the theory $\mathcal{U}$, we can quantify over the type $\mathit{El}\ \iota$, but not over the type Prop), but the quantifier Π of the logical framework $\lambda\Pi/\equiv$ itself.

Remark that, *per se*, the quantification on the variable z of type Prop is predicative, as the term $\Pi z : \mathit{Prop}, (\mathit{Prf}\ x \to \mathit{Prf}\ y \to \mathit{Prf}\ z) \to \mathit{Prf}\ z$ has type `TYPE` and not Prop. But, the rule rewriting $\mathit{Prf}\ (x \wedge y)$ to $\Pi z : \mathit{Prop}, (\mathit{Prf}\ x \to \mathit{Prf}\ y \to \mathit{Prf}\ z) \to \mathit{Prf}\ z$ introduces some impredicativity, as $x \wedge y$ of type Prop is "defined" as the inverse image, for the embedding Prf, of the type $\Pi z : \mathit{Prop}, (\mathit{Prf}\ x \to \mathit{Prf}\ y \to \mathit{Prf}\ z) \to \mathit{Prf}\ z$, that contains a quantification on a variable of type Prop

$$\top : \mathit{Prop} \qquad (\top\text{-decl})$$
$$\mathit{Prf}\ \top \hookrightarrow \Pi z : \mathit{Prop}, \mathit{Prf}\ z \to \mathit{Prf}\ z \qquad (\top\text{-red})$$
$$\bot : \mathit{Prop} \qquad (\bot\text{-decl})$$
$$\mathit{Prf}\ \bot \hookrightarrow \Pi z : \mathit{Prop}, \mathit{Prf}\ z \qquad (\bot\text{-red})$$
$$\neg : \mathit{Prop} \to \mathit{Prop} \qquad (\neg\text{-decl})$$
$$\mathit{Prf}\ (\neg\, x) \hookrightarrow \mathit{Prf}\ x \to \Pi z : \mathit{Prop}, \mathit{Prf}\ z \qquad (\neg\text{-red})$$

$$\wedge : Prop \to Prop \to Prop \quad \text{(written infix)} \qquad (\wedge\text{-decl})$$
$$Prf\ (x \wedge y) \hookrightarrow \Pi z : Prop, (Prf\ x \to Prf\ y \to Prf\ z) \to Prf\ z \qquad (\wedge\text{-red})$$
$$\vee : Prop \to Prop \to Prop \quad \text{(written infix)} \qquad (\vee\text{-decl})$$
$$Prf\ (x \vee y) \hookrightarrow \Pi z : Prop, (Prf\ x \to Prf\ z) \to (Prf\ y \to Prf\ z) \to Prf\ z \qquad (\vee\text{-red})$$
$$\exists : \Pi a : Set, (El\ a \to Prop) \to Prop \qquad (\exists\text{-decl})$$
$$Prf\ (\exists\ a\ p) \hookrightarrow \Pi z : Prop, (\Pi x : El\ a, Prf\ (p\ x) \to Prf\ z) \to Prf\ z \qquad (\exists\text{-red})$$

Infinity

Now that we have the symbols $\top$ and $\bot$, we can express that the type $El\ \iota$ is infinite, that is, that there exists a non-surjective injection from this type to itself. We call this non-surjective injection *succ*. To express its injectivity, we introduce its left inverse *pred*. To express its non-surjectivity, we introduce an element 0, that is not in its image *positive* [19]. This choice of notation enables the definition of natural numbers as some elements of type $El\ \iota$

$$0 : El\ \iota \qquad (0\text{-decl})$$
$$succ : El\ \iota \to El\ \iota \qquad (succ\text{-decl})$$
$$pred : El\ \iota \to El\ \iota \qquad (pred\text{-decl})$$
$$pred\ 0 \hookrightarrow 0 \qquad (pred\text{-red1})$$
$$pred\ (succ\ x) \hookrightarrow x \qquad (pred\text{-red2})$$
$$positive : El\ \iota \to Prop \qquad (positive\text{-decl})$$
$$positive\ 0 \hookrightarrow \bot \qquad (positive\text{-red1})$$
$$positive\ (succ\ x) \hookrightarrow \top \qquad (positive\text{-red2})$$

Classical connectives and quantifiers

The disjunction in constructive logic and in classical logic are governed by different deduction rules, thus they have a different meaning, and they should be expressed with different symbols, for instance $\vee$ for the constructive disjunction and $\vee_c$ for the classical one, just like, in classical logic, we use two different symbols for the inclusive disjunction and the exclusive one. These constructive and classical disjunctions need not belong to different languages, but they can coexist in the same Ecumenical one [36, 16, 35, 25].

Many Ecumenical logics consider the constructive connectives and quantifiers as primitive and attempt to define the classical ones from them, using the negative translation as a definition. In the theory $\mathcal{U}$, we have chosen to define the classical connectives and quantifiers as in [1], for instance $A \vee_c B$ as $(\neg\neg A) \vee (\neg\neg B)$. Using these definitions, the proposition $(P \wedge_c Q) \Rightarrow_c P$ is $(\neg\neg((\neg\neg P) \wedge (\neg\neg Q))) \Rightarrow (\neg\neg P)$, which is not exactly the negative translation $\neg\neg((\neg\neg((\neg\neg P) \wedge (\neg\neg Q))) \Rightarrow (\neg\neg P))$ of $(P \wedge Q) \Rightarrow P$, as the double negation at the root of the proposition is missing. As we already have a distinction between the proposition A and the type $Prf\ A$ of its proofs, we can just include this double negation into the constant *Prf*, introducing a classical version Prf_c of this constant

$$Prf_c : Prop \to \mathtt{TYPE} \qquad (Prf_c\text{-decl})$$
$$Prf_c \hookrightarrow \lambda x : Prop, Prf\ (\neg\,\neg\,x) \qquad (Prf_c\text{-red})$$
$$\Rightarrow_c : Prop \to Prop \to Prop \quad \text{(written infix)} \qquad (\Rightarrow_c\text{-decl})$$
$$\Rightarrow_c \hookrightarrow \lambda x : Prop, \lambda y : Prop, (\neg\,\neg\,x) \Rightarrow (\neg\,\neg\,y) \qquad (\Rightarrow_c\text{-red})$$
$$\wedge_c : Prop \to Prop \to Prop \quad \text{(written infix)} \qquad (\wedge_c\text{-decl})$$
$$\wedge_c \hookrightarrow \lambda x : Prop, \lambda y : Prop, (\neg\,\neg\,x) \wedge (\neg\,\neg\,y) \qquad (\wedge_c\text{-red})$$

$$\vee_c : Prop \to Prop \to Prop \quad \text{(written infix)} \qquad (\vee_c\text{-decl})$$
$$\vee_c \hookrightarrow \lambda x : Prop, \lambda y : Prop, (\neg\neg x) \vee (\neg\neg y) \qquad (\vee_c\text{-red})$$
$$\forall_c : \Pi a : Set, (El\ a \to Prop) \to Prop \qquad (\forall_c\text{-decl})$$
$$\forall_c \hookrightarrow \lambda a : Set, \lambda p : (El\ a \to Prop), \forall\ a\ (\lambda x : El\ a, \neg\neg(p\ x)) \qquad (\forall_c\text{-red})$$
$$\exists_c : \Pi a : Set, (El\ a \to Prop) \to Prop \qquad (\exists_c\text{-decl})$$
$$\exists_c \hookrightarrow \lambda a : Set, \lambda p : (El\ a \to Prop), \exists\ a\ (\lambda x : El\ a, \neg\neg(p\ x)) \qquad (\exists_c\text{-red})$$

Note that $\top_c$ and $\bot_c$ are $\top$ and $\bot$, by definition. Note also that $\neg\neg\neg A$ is equivalent to $\neg A$, so we do not need to duplicate negation either.

Propositions as objects

So far, we have mainly reconstructed the Predicate logic notions of object-term, proposition, and proof. We can now turn to two notions coming from Simple type theory: propositions as objects and functionality.

Simple type theory can be expressed in Predicate logic and Predicate logic is a restriction of Simple type theory, allowing quantification on variables of type ι only. So, once we have reconstructed Predicate logic, we can either define Simple type theory as a theory in Predicate logic or as an extension of Predicate logic. In the theory $\mathcal{U}$, we choose the second option, which leads to a simpler expression of Simple type theory, avoiding the stacking of two encodings. Simple type theory is thus expressed by adding two axioms on top of Predicate logic: one for propositions as objects and one for functionality.

Let us start with propositions as objects. So far, the term ι is the only closed term of type Set. So, we can only quantify over the variables of type $El\ \iota$. In particular, we cannot quantify over propositions. To do so, we just need to declare a constant o of type Set and a rule identifying $El\ o$ and $Prop$

$$o : Set \qquad (o\text{-decl})$$
$$El\ o \hookrightarrow Prop \qquad (o\text{-red})$$

Note that just like there are no terms of type ι, but terms, such as 0, which have type $El\ \iota$, there are no terms of type o, but terms, such as $\top$, that have type $El\ o$, that is $Prop$.

Applying the constant $\forall$ to the constant o, we obtain a term of type $(El\ o \to Prop) \to Prop$, that is $(Prop \to Prop) \to Prop$, and we can express the proposition $\forall p\ (p \Rightarrow p)$ as $\forall\ o\ (\lambda p : Prop, p \Rightarrow p)$. The type $Prf\ (\forall\ o\ (\lambda p : Prop, p \Rightarrow p))$ of the proofs of this proposition rewrites to $\Pi p : Prop, Prf\ p \to Prf\ p$. So, the term $\lambda p : Prop, \lambda x : Prf\ p, x$ is a proof of this proposition.

Functionality

Besides ι and o, we introduce more types in the theory, for functions and sets. To do so, we declare a constant $\leadsto$ and a rewriting rule

$$\leadsto : Set \to Set \to Set \quad \text{(written infix)} \qquad (\leadsto\text{-decl})$$
$$El\ (x \leadsto y) \hookrightarrow El\ x \to El\ y \qquad (\leadsto\text{-red})$$

For instance, these rules enable the construction of the $\lambda\Pi/\equiv$ term $\iota \leadsto \iota$ of type Set that expresses the simple type $\iota \to \iota$. The $\lambda\Pi/\equiv$ term $El\ (\iota \leadsto \iota)$ of type TYPE rewrites to $El\ \iota \to El\ \iota$. The simply typed term $\lambda x : \iota, x$ of type $\iota \to \iota$ is then expressed as the term $\lambda x : El\ \iota, x$ of type $El\ \iota \to El\ \iota$ that is $El\ (\iota \leadsto \iota)$.

Dependent function types

The axiom ($\leadsto$) enables us to give simple types to the object-terms expressing functions. We can also give them dependent types, with the dependent versions of this axiom

$$\leadsto_d : \Pi x : Set, (El\ x \to Set) \to Set \quad \text{(written infix)} \tag{$\leadsto_d$-decl}$$
$$El\ (x \leadsto_d y) \hookrightarrow \Pi z : El\ x, El\ (y\ z) \tag{$\leadsto_d$-red}$$

Note that, if we apply the constant $\leadsto_d$ to a term t and a term $\lambda z : El\ t, u$, where the variable z does not occur in u, then $El\ (t \leadsto_d \lambda z : El\ t, u)$ rewrites to $El\ t \to El\ u$, just like $El\ (t \leadsto u)$. Thus, the constant $\leadsto_d$ is useful only if we can build a term $\lambda z : El\ t, u$ where the variable z occurs in u. With the symbols we have introduced so far, this is not possible. Just like we have a constant ι of type Set, we could add a constant *array* of type $El\ \iota \to Set$ such that *array* n is the type of arrays of length n. We could then construct the term $(\iota \leadsto_d \lambda x : El\ \iota, array\ x)$ of type Set and the type $El\ (\iota \leadsto_d \lambda x : El\ \iota, array\ x)$ that rewrites to $\Pi x : El\ \iota, El\ (array\ x)$, would be the type of functions mapping a natural number n to an array of length n. So, this symbol $\leadsto_d$ becomes useful, only if we add such a constant *array*, object-level dependent types, or the symbols π or *psub* below.

Dependent implication

In the same way, we can add a dependent implication, where, in the proposition $A \Rightarrow B$, the proof of A may occur in B

$$\Rightarrow_d : \Pi x : Prop, (Prf\ x \to Prop) \to Prop \quad \text{(written infix)} \tag{$\Rightarrow_d$-decl}$$
$$Prf\ (x \Rightarrow_d y) \hookrightarrow \Pi z : Prf\ x, Prf\ (y\ z) \tag{$\Rightarrow_d$-red}$$

Proofs in object-terms

To construct an object-term, we sometimes want to apply a function symbol to other object-terms and also to proofs. For instance, we may want to apply the Euclidean division *div* to two numbers t and u and to a proof that u is positive. To be able to so, we introduce another constant π and the corresponding rewriting rule

$$\pi : \Pi x : Prop, (Prf\ x \to Set) \to Set \tag{π-decl}$$
$$El\ (\pi\ x\ y) \hookrightarrow \Pi z : Prf\ x, El\ (y\ z) \tag{π-red}$$

This way, we can give, to the constant *div*, the type

$$El\ (\iota \leadsto \iota \leadsto_d \lambda y : El\ \iota, \pi\ (positive\ y)\ (\lambda z : Prf\ (positive\ y), \iota))$$

If we also have a constant eq_ι of type $El\ (\iota \leadsto \iota \leadsto o)$, we can then express the proposition

$$positive\ y \Rightarrow_d \lambda p : Prf\ (positive\ y), eq_\iota\ (div\ x\ y\ p)\ (div\ x\ y\ p)$$

usually written $y > 0 \Rightarrow x/y = x/y$. The proposition $x/y = x/y$ is well-formed, but it contains an implicit free variable p, for a proof of $y > 0$. This variable is bound by the implication, that needs therefore to be a dependent implication.

Proof irrelevance

If p and q are two non convertible proofs of the proposition $\mathit{positive}\ 2$, the terms $\mathit{div}\ 7\ 2\ p$ and $\mathit{div}\ 7\ 2\ q$ are not convertible. As a consequence, even if we had a reflexivity axiom for the aforementioned equality eq_ι, the proposition

$$eq_\iota\ (\mathit{div}\ 7\ 2\ p)\ (\mathit{div}\ 7\ 2\ q)$$

would not be provable.

To make these terms convertible, we embed the theory into an extended one, that contains another constant

$$\mathit{div}^\dagger : \mathit{El}\ (\iota \rightsquigarrow \iota \rightsquigarrow \iota)$$

and a rule

$$\mathit{div}\ x\ y\ p \ \hookrightarrow\ \mathit{div}^\dagger\ x\ y$$

and we define convertibility in this extended theory. This way, the terms $\mathit{div}\ 7\ 2\ p$ and $\mathit{div}\ 7\ 2\ q$ are convertible, as they both reduce to $\mathit{div}^\dagger\ 7\ 2$.

Note that, in the extended theory, the constant $\mathit{div}^\dagger$ enables the construction of the erroneous term $\mathit{div}^\dagger\ 1\ 0$. But the extended theory is only used to define the convertibility in the restricted one and this term is not a term of the restricted theory. It is not even the reduct of a term of the form $\mathit{div}\ 1\ 0\ r$ [20, 9].

Dependent pairs and predicate subtyping

Instead of declaring a constant div that takes three arguments: a number t, a number u, and a proof p that u is positive, we can declare a constant that takes two arguments: a number t and a pair $\mathit{pair}\ \iota\ \mathit{positive}\ u\ p$ formed with a number u and a proof p that u is positive.

The type of the pair $\mathit{pair}\ \iota\ \mathit{positive}\ u\ p$ is written $\mathit{psub}\ \iota\ \mathit{positive}$, or informally $\{x : \iota \mid \mathit{positive}\ x\}$. It can be called "the type of positive numbers". It is a subtype of the type of natural numbers defined with the predicate $\mathit{positive}$. Therefore, the symbol psub introduces predicate subtyping. We thus declare a constant psub and a constant pair

$$\mathit{psub} : \Pi t : \mathit{Set}, (\mathit{El}\ t \to \mathit{Prop}) \to \mathit{Set} \qquad (\mathit{psub}\text{-decl})$$
$$\mathit{pair} : \Pi t : \mathit{Set}, \Pi p : \mathit{El}\ t \to \mathit{Prop}, \Pi m : \mathit{El}\ t, \mathit{Prf}\ (p\ m) \to \mathit{El}\ (\mathit{psub}\ t\ p) \qquad (\mathit{pair}\text{-decl})$$

This way, instead of giving the type $\mathit{El}\ (\iota \rightsquigarrow \iota \rightsquigarrow_d \lambda y : \mathit{Prf}\ (\mathit{positive}\ y), \iota)$ to the constant div, we can give it the type $\mathit{El}\ (\iota \rightsquigarrow \mathit{psub}\ \iota\ \mathit{positive} \rightsquigarrow \iota)$.

To avoid introducing a new positive number $\mathit{pair}\ \iota\ \mathit{positive}\ 3\ p$ with each proof p that 3 is positive, we make this symbol pair proof irrelevant [20, 9] by introducing a symbol $\mathit{pair}^\dagger$ and a rewriting rule that discards the proof

$$\mathit{pair}^\dagger : \Pi t : \mathit{Set}, \Pi p : \mathit{El}\ t \to \mathit{Prop}, \mathit{El}\ t \to \mathit{El}\ (\mathit{psub}\ t\ p) \qquad (\mathit{pair}^\dagger\text{-decl})$$
$$\mathit{pair}\ t\ p\ m\ h \ \hookrightarrow\ \mathit{pair}^\dagger\ t\ p\ m \qquad (\mathit{pair}\text{-red})$$

This declaration and this rewriting rule are not part of the theory $\mathcal{U}$ but of the theory $\mathcal{U}^\dagger$ used to define the conversion on the terms of $\mathcal{U}$.

Finally, we declare the projections *fst* and *snd* together with an associated rewriting rule

$$fst : \Pi t : Set, \Pi p : El\ t \to Prop, El\ (psub\ t\ p) \to El\ t \qquad (fst\text{-decl})$$
$$fst\ t\ p\ (pair^{\dagger}\ t'\ p'\ m) \hookrightarrow m \qquad (fst\text{-red})$$
$$snd : \Pi t : Set, \Pi p : El\ t \to Prop, \Pi m : El\ (psub\ t\ p), Prf\ (p\ (fst\ t\ p\ m)) \qquad (snd\text{-decl})$$

Prenex predicative type quantification in types

Using the symbols of the theory $\mathcal{U}$ introduced so far, the symbol for equality of elements of type ι is eq_ι of type $El\ (\iota \rightsquigarrow \iota \rightsquigarrow o)$. This equality symbol is not polymorphic. Indeed, it cannot be used to express the equality of, for example, functions of type $\iota \rightsquigarrow \iota$. This motivates the introduction of object-level polymorphism [24, 37]. However extending Simple type theory with object-level polymorphism makes it inconsistent [30, 11], and similarly it makes the theory $\mathcal{U}$ inconsistent. So, object-level polymorphism in $\mathcal{U}$ is restricted to prenex polymorphism. To do so, we introduce a new constant *Scheme* of type `TYPE`, a constant *Els* to embed the terms of type *Scheme* into terms of type `TYPE`, a constant $\uparrow$ to embed the terms of type *Set* into terms of type *Scheme* and a rule connecting these embeddings

$$Scheme : \mathtt{TYPE} \qquad (Scheme\text{-decl})$$
$$Els : Scheme \to \mathtt{TYPE} \qquad (Els\text{-decl})$$
$$\uparrow : Set \to Scheme \qquad (\uparrow\text{-decl})$$
$$Els\ (\uparrow\ x) \hookrightarrow El\ x \qquad (\uparrow\text{-red})$$

We then introduce a quantifier for the variables of type *Set* in the terms of type *Scheme* and the associated rewriting rule

$$\forall : (Set \to Scheme) \to Scheme \qquad (\forall\text{-decl})$$
$$Els\ (\forall\ p) \hookrightarrow \Pi x : Set, Els\ (p\ x) \qquad (\forall\text{-red})$$

This way, we can give the polymorphic type $Els\ (\forall\ (\lambda A : Set, \uparrow (A \rightsquigarrow A \rightsquigarrow o)))$ to the equality *eq*. In the same way, the type of the identity function is $Els\ (\forall\ (\lambda A : Set, \uparrow (A \rightsquigarrow A)))$. It rewrites to $\Pi A : Set, El\ A \to El\ A$. Therefore, it is inhabited by the term $\lambda A : Set, \lambda x : El\ A, x$.

Prenex predicative type quantification in propositions

When we express the reflexivity of the polymorphic equality, we need also to quantify over a type variable, but now in a proposition. To be able to do so, we introduce another quantifier and its associated rewriting rule

$$\forall\!\!\!\!\forall : (Set \to Prop) \to Prop \qquad (\forall\!\!\!\!\forall\text{-decl})$$
$$Prf\ (\forall\!\!\!\!\forall\ p) \hookrightarrow \Pi x : Set, Prf\ (p\ x) \qquad (\forall\!\!\!\!\forall\text{-red})$$

This way, the reflexivity of equality can be expressed as $(\forall\!\!\!\!\forall\ (\lambda A : Set, \forall\ A\ (\lambda x : El\ A, eq\ A\ x\ x)))$.

The theory $\mathcal{U}$: bringing everything together

The theory $\mathcal{U}$ is formed with the 38 axioms with a black bar at the beginning of the line: (Set), (El), (ι), ($Prop$), (Prf), ($\Rightarrow$), ($\forall$), ($\top$), ($\bot$), ($\neg$), ($\wedge$), ($\vee$), ($\exists$), (Prf_c), ($\Rightarrow_c$), ($\wedge_c$), ($\vee_c$), ($\forall_c$), ($\exists_c$), (o), ($\rightsquigarrow$), ($\rightsquigarrow_d$), ($\Rightarrow_d$), (π), (0), ($succ$), ($pred$), ($positive$), ($psub$), ($pair$), ($pair^{\dagger}$), (fst), (snd), ($Scheme$), (Els), ($\uparrow$), ($\forall$), ($\forall\!\!\!\!\forall$). Note that, strictly speaking, the declaration ($pair^{\dagger}$-decl) and the rule ($pair$-red) are not part of the theory $\mathcal{U}$, but of its extension $\mathcal{U}^{\dagger}$ used to define the conversion on the terms of $\mathcal{U}$. Among these axioms, 12 only have a constant declaration, 24 have a constant declaration and one rewriting rule, and 2 have a constant declaration and two rewriting rules. So $\Sigma_{\mathcal{U}}$ contains 38 declarations and $\mathcal{R}_{\mathcal{U}}$ 28 rules.

This large number of axioms is explained by the fact that $\lambda\Pi/\equiv$ is a weaker framework than Predicate logic. The 19 first axioms are needed just to construct notions that are primitive in Predicate logic: terms, propositions, with their 13 constructive and classical connectives and quantifiers, and proofs. So the theory $\mathcal{U}$ is just 19 axioms on top of the definition of Predicate logic.

It is also explained by the fact that axioms are more atomic than in Predicate logic, for instance 4 axioms: (0), (*succ*), (*pred*), and (*positive*) are needed to express "the" axiom of infinity, 5 (*psub*), (*pair*), ($pair^{\dagger}$), (*fst*), and (*snd*) to express predicate subtyping, and 5 (*Scheme*), (*Els*), ($\uparrow$), ($\forall$), and ($\not\forall$) to express prenex polymorphism. The 5 remaining axioms express propositions as objects (o), various forms of functionality ($\rightsquigarrow$), ($\rightsquigarrow_d$), and (π), and dependent implication ($\Rightarrow_d$).

4 Sub-theories

Not all proofs require all these axioms. Many proofs can be expressed in sub-theories built by bringing together some of the axioms of $\mathcal{U}$, but not all.

Given subsets $\Sigma_{\mathcal{S}}$ of $\Sigma_{\mathcal{U}}$ and $\mathcal{R}_{\mathcal{S}}$ of $\mathcal{R}_{\mathcal{U}}$, we would like to be sure that a proof in $\mathcal{U}$, using only constants in $\Sigma_{\mathcal{S}}$, is a proof in $\Sigma_{\mathcal{S}}, \mathcal{R}_{\mathcal{S}}$. Such a result is trivial in Predicate logic: for instance, a proof in ZFC which does not use the axiom of choice is a proof in ZF, but it is less straightforward in $\lambda\Pi/\equiv$, because $\Sigma_{\mathcal{S}}, \mathcal{R}_{\mathcal{S}}$ might not be a theory. So we should not consider any pair $\Sigma_{\mathcal{S}}, \mathcal{R}_{\mathcal{S}}$. For instance, as *Set* occurs in the type of *El*, if we want *El* in $\Sigma_{\mathcal{S}}$, we must take *Set* as well. In the same way, as *positive* (*succ* x) rewrites to $\top$, if we want (*positive*) and (*succ*) in $\Sigma_{\mathcal{S}}$, we must include $\top$ in $\Sigma_{\mathcal{S}}$ and the rule rewriting *positive* (*succ* x) to $\top$ in $\mathcal{R}_{\mathcal{S}}$.

This leads to a definition of a notion of sub-theory and to prove that, if $\Sigma_1, \mathcal{R}_1$ is a sub-theory of a theory $\Sigma_0, \mathcal{R}_0$, Γ, t and A are in $\Lambda(\Sigma_1)$, and $\Gamma \vdash_{\Sigma_0,\mathcal{R}_0} t : A$, then $\Gamma \vdash_{\Sigma_1,\mathcal{R}_1} t : A$.

This property implies that, if π is a proof of A in $\mathcal{U}$ and both A and π are in $\Lambda(\Sigma_1)$, then π is a proof of A in $\Sigma_1, \mathcal{R}_1$, but it does not imply that if A is in $\Lambda(\Sigma_1)$ and A has a proof in $\mathcal{U}$, then it has a proof in $\Sigma_1, \mathcal{R}_1$.

4.1 Fragments

▶ **Definition 3** (Fragment). *A signature Σ_1 is included in a signature Σ_0, $\Sigma_1 \subseteq \Sigma_0$, if each declaration $c : A$ of Σ_1 is a declaration of Σ_0.*

A system $\Sigma_1, \mathcal{R}_1$ is a fragment of a system $\Sigma_0, \mathcal{R}_0$, if the following conditions are satisfied:

- $\Sigma_1 \subseteq \Sigma_0$ *and* $\mathcal{R}_1 \subseteq \mathcal{R}_0$;
- *for all* $(c : A) \in \Sigma_1$, $const(A) \subseteq |\Sigma_1|$;
- *for all* $\ell \hookrightarrow r \in \mathcal{R}_0$, *if* $const(\ell) \subseteq |\Sigma_1|$, *then* $const(r) \subseteq |\Sigma_1|$ *and* $\ell \hookrightarrow r \in \mathcal{R}_1$.

We write $\vdash_i$ for $\vdash_{\Sigma_i,\mathcal{R}_i}$, $\hookrightarrow_i$ for $\hookrightarrow_{\beta\mathcal{R}_i}$, and $\equiv_i$ for $\equiv_{\beta\mathcal{R}_i}$.

▶ **Lemma 4** (Preservation of reduction). *If $\Sigma_1, \mathcal{R}_1$ is a fragment of $\Sigma_0, \mathcal{R}_0$, $t \in \Lambda(\Sigma_1)$ and $t \hookrightarrow_0 u$, then $t \hookrightarrow_1 u$ and $u \in \Lambda(\Sigma_1)$.*

Proof. By induction on the position where the rule is applied. We only detail the case of a top reduction, the other cases easily following by induction hypothesis.

So, let $\ell \hookrightarrow r$ be the rule used to rewrite t in u and θ such that $t = \theta\ell$ and $u = \theta r$. As $t \in \Lambda(\Sigma_1)$, we have $\ell \in \Lambda(\Sigma_1)$ and, for all x free in ℓ, $\theta x \in \Lambda(\Sigma_1)$. Thus, as $\Sigma_1, \mathcal{R}_1$ is a fragment of $\Sigma_0, \mathcal{R}_0$, $r \in \Lambda(\Sigma_1)$ and $\ell \hookrightarrow r \in \mathcal{R}_1$. Therefore $t \hookrightarrow_1 u$ and $u = \theta r \in \Lambda(\Sigma_1)$. ◀

▶ **Lemma 5** (Preservation of confluence). *Every fragment of a confluent system is confluent.*

Proof. Let $\Sigma_1, \mathcal{R}_1$ be a fragment of a confluent system $\Sigma_0, \mathcal{R}_0$. We prove that $\hookrightarrow_1$ is confluent on $\Lambda(\Sigma_1)$. Assume that $t, u, v \in \Lambda(\Sigma_1)$, $t \hookrightarrow_1^* u$ and $t \hookrightarrow_1^* v$. Since $|\Sigma_1| \subseteq |\Sigma_0|$, we have $t, u, v \in \Lambda(\Sigma_0)$. Since $\mathcal{R}_1 \subseteq \mathcal{R}_0$, we have $t \hookrightarrow_0^* u$ and $t \hookrightarrow_0^* v$. By confluence of $\hookrightarrow_0$ on $\Lambda(\Sigma_0)$, there exists a w in $\Lambda(\Sigma_0)$ such that $u \hookrightarrow_0^* w$ and $v \hookrightarrow_0^* w$. Since $u, v \in \Lambda(\Sigma_1)$, by Lemma 4, $w \in \Lambda(\Sigma_1)$, $u \hookrightarrow_1^* w$ and $v \hookrightarrow_1^* w$. ◀

▶ **Definition 6** (Sub-theory). *A system $\Sigma_1, \mathcal{R}_1$ is a sub-theory of a theory $\Sigma_0, \mathcal{R}_0$, if $\Sigma_1, \mathcal{R}_1$ is a fragment of $\Sigma_0, \mathcal{R}_0$ and it is a theory. As we already know that $\mathcal{R}_1$ is confluent, this amounts to say that each rule of $\mathcal{R}_1$ preserves typing in $\Sigma_1, \mathcal{R}_1$.*

4.2 The fragment theorem

▶ **Theorem 7.** *Let $\Sigma_0, \mathcal{R}_0$ be a confluent system and $\Sigma_1, \mathcal{R}_1$ be a fragment of $\Sigma_0, \mathcal{R}_0$ that preserves typing. If the judgement $\Gamma \vdash_0 t : D$ is derivable, $\Gamma \in \Lambda(\Sigma_1)$ and $t \in \Lambda(\Sigma_1)$, then there exists $D' \in \Lambda(\Sigma_1)$ such that $D \hookrightarrow_0^* D'$ and the judgement $\Gamma \vdash_1 t : D'$ is derivable.*

Proof. By induction on the derivation. The important cases are (abs), (app), and (conv). The other cases are a simple application of the induction hypothesis.

- If the last rule of the derivation is

$$\frac{\Gamma \vdash_0 A : \texttt{TYPE} \quad \Gamma, x : A \vdash_0 B : s \quad \Gamma, x : A \vdash_0 t : B}{\Gamma \vdash_0 \lambda x : A, t : \Pi x : A, B} \text{(abs)}$$

  as Γ, A, and t are in $\Lambda(\Sigma_1)$, by induction hypothesis, there exists A' in $\Lambda(\Sigma_1)$ such that $\texttt{TYPE} \hookrightarrow_0^* A'$ and $\Gamma \vdash_1 A : A'$ is derivable, and there exists B' in $\Lambda(\Sigma_1)$ such that $B \hookrightarrow_0^* B'$ and $\Gamma, x : A \vdash_1 t : B'$ is derivable. As $\texttt{TYPE}$ is a sort, $A' = \texttt{TYPE}$. Therefore, $\Gamma \vdash_1 A : \texttt{TYPE}$ is derivable.
  As B is typable and every subterm of a typable term is typable, $\texttt{KIND}$ does not occur in B. As $B \hookrightarrow_0^* B'$ and no rule contains $\texttt{KIND}$, $\texttt{KIND}$ does not occur in B' as well. Hence, $B' \neq \texttt{KIND}$. By Lemma 2, as $\Gamma, x : A \vdash_1 t : B'$ is derivable and $B' \neq \texttt{KIND}$, there exists a sort s' such that $\Gamma, x : A \vdash_1 B' : s'$ is derivable.
  Thus, by the rule (abs), $\Gamma \vdash_1 \lambda x : A, t : \Pi x : A, B'$ is derivable. So there is $D' = \Pi x : A, B'$ in $\Lambda(\Sigma_1)$ such that $\Pi x : A, B \hookrightarrow_0^* D'$ and $\Gamma \vdash_1 \lambda x : A, t : D'$ is derivable.
- If the last rule of the derivation is

$$\frac{\Gamma \vdash_0 t : \Pi x : A, B \quad \Gamma \vdash_0 u : A}{\Gamma \vdash_0 t\ u : (u/x)B} \text{(app)}$$

  as Γ, t, and u are in $\Lambda(\Sigma_1)$, by induction hypothesis, there exist C and A_2 in $\Lambda(\Sigma_1)$, such that $\Pi x : A, B \hookrightarrow_0^* C$, $\Gamma \vdash_1 t : C$ is derivable, $A \hookrightarrow_0^* A_2$, and $\Gamma \vdash_1 u : A_2$ is derivable. As $\Pi x : A, B \hookrightarrow_0^* C$ and rewriting rules are of the form $(c\ l_1 \ldots l_n \hookrightarrow r)$, there exist A_1 and B_1 in $\Lambda(\Sigma_1)$ such that $C = \Pi x : A_1, B_1$, $A \hookrightarrow_0^* A_1$, and $B \hookrightarrow_0^* B_1$. By confluence of $\hookrightarrow_0$, there exists A' such that $A_1 \hookrightarrow_0^* A'$ and $A_2 \hookrightarrow_0^* A'$. By Lemma 4, as $A_1 \in \Lambda(\Sigma_1)$ and $A_1 \hookrightarrow_0^* A'$, we have $A' \in \Lambda(\Sigma_1)$ and $A_1 \hookrightarrow_1^* A'$. In a similar way, as $A_2 \in \Lambda(\Sigma_1)$ and $A_2 \hookrightarrow_0^* A'$, we have $A_2 \hookrightarrow_1^* A'$. By Lemma 2, as $\Gamma \vdash_1 t : \Pi x : A_1, B_1$ is derivable and $\Pi x : A_1, B_1 \neq \texttt{KIND}$, there exists a sort s such that $\Gamma \vdash_1 \Pi x : A_1, B_1 : s$ is derivable. Thus, by Lemma 2, $\Gamma \vdash_1 A_1 : \texttt{TYPE}$ is derivable.

As $\Gamma \vdash_1 \Pi x : A_1, B_1 : s$, $\Pi x : A_1, B_1 \hookrightarrow_1^* \Pi x : A', B_1$, and $\Sigma_1, \mathcal{R}_1$ preserves typing, $\Gamma \vdash_1 \Pi x : A', B_1 : s$ is derivable. In a similar way, as $\Gamma \vdash_1 A_1 : \texttt{TYPE}$ is derivable, and $A_1 \hookrightarrow_1^* A'$, $\Gamma \vdash_1 A' : \texttt{TYPE}$ is derivable. Therefore, by the rule (conv), $\Gamma \vdash_1 t : \Pi x : A', B_1$ and $\Gamma \vdash_1 u : A'$ are derivable. Therefore, by the rule (app), $\Gamma \vdash_1 t\ u : (u/x)B_1$ is derivable. So there exists $D' = (u/x)B_1$ in $\Lambda(\Sigma_1)$, such that $(u/x)B \hookrightarrow_0^* D'$ and $\Gamma \vdash_1 t\ u : D'$ is derivable.

- If the last rule of the derivation is

$$\frac{\Gamma \vdash_0 t : A \quad \Gamma \vdash_0 B : s}{\Gamma \vdash_0 t : B}\ (\text{conv}) \quad A \equiv_{\beta\mathcal{R}_0} B$$

as Γ and t are in $\Lambda(\Sigma_1)$, by induction hypothesis, there exists A' in $\Lambda(\Sigma_1)$ such that $A \hookrightarrow_0^* A'$ and $\Gamma \vdash_1 t : A'$ is derivable. By confluence of $\hookrightarrow_0$, there exists C such that $A' \hookrightarrow_0^* C$ and $B \hookrightarrow_0^* C$. As $A' \in \Lambda(\Sigma_1)$ and $A' \hookrightarrow_0^* C$ we have, by Lemma 4, $C \in \Lambda(\Sigma_1)$ and $A' \hookrightarrow_1^* C$.
As B is typable and every subterm of a typable term is typable, $\texttt{KIND}$ does not occur in B. As $B \hookrightarrow_0^* C$ and no rule contains $\texttt{KIND}$, $\texttt{KIND}$ does not occur in C as well. Thus $C \neq \texttt{KIND}$. As $A' \hookrightarrow_0^* C$, $A' \neq \texttt{KIND}$. By Lemma 2, as $\Gamma \vdash_1 t : A'$ and $A' \neq \texttt{KIND}$, there exists a sort s' such that $\Gamma \vdash_1 A' : s'$ is derivable. Thus, as $A' \hookrightarrow_1^* C$, and $\Sigma_1, \mathcal{R}_1$ preserves typing, $\Gamma \vdash_1 C : s'$ is derivable. a As $\Gamma \vdash_1 t : A'$ and $\Gamma \vdash_1 C : s'$ are derivable and $A' \hookrightarrow_1 C$, by the rule (conv), $\Gamma \vdash_1 t : C$ is derivable. Thus there exists $D' = C$ in $\Lambda(\Sigma_1)$ such that $\Gamma \vdash_1 t : D'$ is derivable and $B \hookrightarrow_0^* D'$. ◀

▶ **Corollary 8.** *Let $\Sigma_0, \mathcal{R}_0$ be a confluent system, $\Sigma_1, \mathcal{R}_1$ be a fragment of $\Sigma_0, \mathcal{R}_0$ that preserves typing. If $\Gamma \vdash_0 t : D$, $\Gamma \in \Lambda(\Sigma_1)$, $t \in \Lambda(\Sigma_1)$, and $D \in \Lambda(\Sigma_1)$, then $\Gamma \vdash_1 t : D$.*

In particular, if $\Sigma_0, \mathcal{R}_0$ is a theory, $\Sigma_1, \mathcal{R}_1$ be a sub-theory of $\Sigma_0, \mathcal{R}_0$, $\Gamma \vdash_0 t : D$, $\Gamma \in \Lambda(\Sigma_1)$, $t \in \Lambda(\Sigma_1)$, and $D \in \Lambda(\Sigma_1)$, then $\Gamma \vdash_1 t : D$.

Proof. There is a $D' \in \Lambda(\Sigma_1)$ such that $D \hookrightarrow_0^* D'$ and $\Gamma \vdash_1 t : D'$. As $D \in \Lambda(\Sigma_1)$ and $D \hookrightarrow_0^* D'$. By Lemma 4 we have $D \hookrightarrow_1^* D'$, and we conclude with the rule (conv). ◀

▶ **Theorem 9** (Sub-theories of $\mathcal{U}$)**.** *Every fragment $\Sigma_1, \mathcal{R}_1$ of $\mathcal{U}$ (including $\mathcal{U}$ itself) is a theory, that is, is confluent and preserves typing.*

Proof. The relation $\hookrightarrow_{\beta\mathcal{R}_\mathcal{U}}$ is confluent on $\Lambda(\Sigma_\mathcal{U})$ since it is an orthogonal combinatory reduction system [31]. Hence, after the fragment theorem, it is sufficient to prove that every rule of $\mathcal{R}_\mathcal{U}$ preserves typing in any fragment $\Sigma_1, \mathcal{R}_1$ containing the symbols of the rule.

To this end, we will use the criterion described in [8, Theorem 19] which consists in computing the equations that must be satisfied for a rule left-hand side to be typable, which are system-independent, and then check that the right-hand side has the same type modulo these equations in the desired system: for all rules $l \hookrightarrow r \in \Lambda(\Sigma_1)$, sets of equations $\mathcal{E}$ and terms T, if the inferred type of l is T, the typability constraints of l are $\mathcal{E}$, and r has type type T in the system $\Lambda(\Sigma_1)$ whose conversion relation $\equiv_{\beta\mathcal{R}\mathcal{E}}$ has been enriched with $\mathcal{E}$, then $l \hookrightarrow r$ preserves typing in $\Lambda(\Sigma_1)$.

This criterion can easily be checked for all the rules but (*pred*-red2) and (*fst*-red) because, except in those two cases, the left-hand side and the right-hand side have the same type.

In (*pred*-red2), $\mathit{pred}\ (\mathit{succ}\ x) \hookrightarrow x$, the left-hand side has type $\mathit{El}\ \iota$ if the equation $\mathit{type}(x) = \mathit{El}\ \iota$ is satisfied. Modulo this equation, the right-hand side has type $\mathit{El}\ \iota$ in any fragment containing the symbols of the rule.

In (*fst*-red), $\mathit{fst}\ t\ p\ (\mathit{pair}^\dagger\ t'\ p'\ m) \hookrightarrow m$, the left-hand side has type $\mathit{El}\ t$ if $\mathit{type}(t) = \mathit{Set}$, $\mathit{type}(p) = \mathit{El}\ t \to \mathit{Prop}$, $\mathit{El}\ (\mathit{psub}\ t'\ p') = \mathit{El}\ (\mathit{psub}\ t\ p)$, $\mathit{type}(t') = \mathit{Set}$, $\mathit{type}(p') = \mathit{El}\ t' \to \mathit{Prop}$, and $\mathit{type}(m) = \mathit{El}\ t'$. But, in $\mathcal{U}$, there is no rule of the form $\mathit{El}\ (\mathit{psub}\ t\ p) \hookrightarrow r$. Hence,

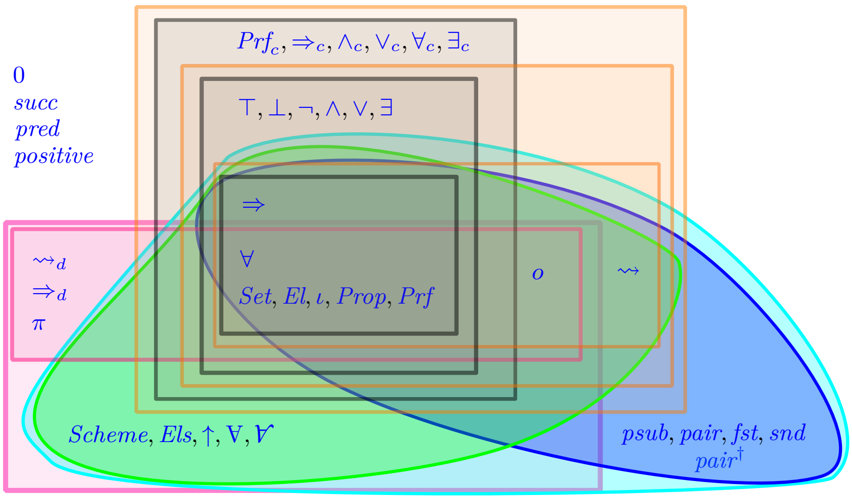


Figure 2 The wind rose. In black: Minimal, Constructive, and Ecumenical predicate logic. In orange: Minimal, Constructive, and Ecumenical simple type theory. In green: Simple type theory with prenex polymorphism. In blue: Simple type theory with predicate subtyping. In cyan: Simple type theory with predicate subtyping and prenex polymorphism. In pink: the Calculus of constructions with a constant ι, without and with prenex polymorphism.

by confluence, the equation $El\ (psub\ t'\ p') = El\ (psub\ t\ p)$ is equivalent to the equations $t' = t$ and $p' = p$. Therefore, the right-hand side is of type $El\ t$ in every fragment of $\mathcal{U}$ containing the symbols of the rule. ◀

5 Examples of sub-theories of the theory $\mathcal{U}$

We finally identify 13 sub-theories of the theory $\mathcal{U}$, that correspond to known theories. For each of these sub-theories $\Sigma_\mathcal{S}, \mathcal{R}_\mathcal{S}$, according to the Corollary 8, if Γ, t, and A are in $\Lambda(\Sigma_\mathcal{S})$, and $\Gamma \vdash_{\Sigma_\mathcal{U}, \mathcal{R}_\mathcal{U}} t : A$, then $\Gamma \vdash_{\mathcal{R}_\mathcal{S}, \Sigma_\mathcal{S}} t : A$.

Minimal predicate logic. The 7 axioms (Set), (El), (ι), ($Prop$), (Prf), ($\Rightarrow$), and ($\forall$) define Minimal predicate logic. This theory can be proven equivalent to more common formulations of Minimal predicate logic. As Minimal predicate logic is itself a logical framework, it must be complemented with more axioms, such as the axioms of geometry, arithmetic, etc.

Constructive predicate logic. The 13 axioms (Set), (El), (ι), ($Prop$), (Prf), ($\Rightarrow$), ($\forall$), ($\top$), ($\bot$), ($\neg$), ($\wedge$), ($\vee$), and ($\exists$) define Constructive predicate logic. This theory can be proven equivalent to more common formulations of Constructive predicate logic [15, 3].

Ecumenical predicate logic. The 19 axioms (Set), (El), (ι), ($Prop$), (Prf), ($\Rightarrow$), ($\forall$), ($\top$), ($\bot$), ($\neg$), ($\wedge$), ($\vee$), ($\exists$), (Prf_c), ($\Rightarrow_c$), ($\wedge_c$), ($\vee_c$), ($\forall_c$), and ($\exists_c$) define Ecumenical predicate logic. This theory can be proven equivalent to more common formulations of Ecumenical predicate logic [26]. Note that classical predicate logic is not a sub-theory of the theory $\mathcal{U}$, because the classical connectives and quantifiers depend on the constructive ones. Yet, it is known that if a proposition contains only classical connectives and quantifiers, it is provable in Ecumenical predicate logic if and only if it is provable in classical predicate logic.

Minimal simple type theory. The 9 axioms (*Set*), (ι), (*El*), (*Prop*), (*Prf*), ($\Rightarrow$), ($\forall$), (o), and ($\leadsto$) define Minimal simple type theory. And this theory can be proven equivalent to more common formulations of Minimal simple type theory [2, 3]. We could save the declaration (*Prop*-decl) and the rule (o-red) by replacing everywhere *Prop* with *El* o[3]. However, by removing (*Prop*-decl) and (o-red), this theory does not construct Simple type theory as an extension of Minimal predicate logic.

Constructive simple type theory. The 15 axioms (*Set*), (*El*), (ι), (*Prop*), (*Prf*), ($\Rightarrow$), ($\forall$), ($\top$), ($\bot$), ($\neg$), ($\wedge$), ($\vee$), ($\exists$), (o) and ($\leadsto$) define Constructive simple type theory.

Ecumenical simple type theory. The 21 axioms (*Set*), (*El*), (ι), (*Prop*), (*Prf*), ($\Rightarrow$), ($\forall$), ($\top$), ($\bot$), ($\neg$), ($\wedge$), ($\vee$), ($\exists$), (Prf_c), ($\Rightarrow_c$), ($\wedge_c$), ($\vee_c$), ($\forall_c$), ($\exists_c$), (o) and ($\leadsto$) define Ecumenical simple type theory. And this theory can be proven equivalent to more common formulations of Ecumenical simple type theory [26].

Simple type theory with predicate subtyping. Adding to the 9 axioms of Minimal simple type theory the 5 axioms of predicate subtyping yields Minimal simple type theory with predicate subtyping, formed with the 14 axioms (*Set*), (ι), (*El*), (*Prop*), (*Prf*), ($\Rightarrow$), ($\forall$), (o), ($\leadsto$), (*psub*), (*pair*), ($pair^\dagger$), (*fst*), and (*snd*). This theory can be proven equivalent to more common formulations of Minimal simple type theory with predicate subtyping [23, 9]. Such formulations like PVS [33] often use predicate subtyping implicitly to provide a lighter syntax without (*pair*), ($pair^\dagger$), (*fst*) nor (*snd*) but at the expense of losing uniqueness of type and making type-checking undecidable. In these cases, terms generally do not hold the proofs needed to be of a sub-type, which provides proof irrelevance. Our implementation of proof irrelevance of Section 3 Page 9 extends the conversion in order to ignore these proofs.

Simple type theory with prenex predicative polymorphism. Adding to Minimal simple type theory the 5 axioms of prenex predicative polymorphism yields Simple type theory with prenex predicative polymorphism (STT$\forall$) [40, 41] formed with the 14 axioms (*Set*), (*El*), (ι), (*Prop*), (*Prf*), ($\Rightarrow$), ($\forall$), (o), ($\leadsto$), (*Scheme*), (*Els*), ($\uparrow$), ($\mathbb{V}$), and ($\dot{\forall}$).

Simple type theory with predicate subtyping and prenex polymorphism. Adding to the 9 axioms of Simple type theory both the 5 axioms of predicate subtyping and the 5 axioms of prenex polymorphism yields a sub-theory with 19 axioms which is a subsystem of PVS [33] handling both predicate subtyping and prenex polymorphism.

The Calculus of constructions. The 9 axioms (*Set*), (*El*), (*Prop*), (*Prf*), ($\Rightarrow_d$), ($\forall$), (o), ($\leadsto_d$), and (π) define the Calculus of constructions. This is the usual expression of the Calculus of constructions in $\lambda\Pi/\equiv$ [13, 3] except that we write *Prop* for U_*, *Prf* for ε_*, *Set* for $U_\Box$, *El* for $\varepsilon_\Box$, o for $\dot{*}$, $\Rightarrow_d$ for $\dot{\Pi}_{\langle *,*,*\rangle}$, $\forall$ for $\dot{\Pi}_{\langle \Box,*,*\rangle}$, π for $\dot{\Pi}_{\langle *,\Box,\Box\rangle}$, and $\leadsto_d$ for $\dot{\Pi}_{\langle \Box,\Box,\Box\rangle}$. As $\Rightarrow_d$ is $\dot{\Pi}_{\langle *,*,*\rangle}$, $\forall$ is $\dot{\Pi}_{\langle \Box,*,*\rangle}$, π is $\dot{\Pi}_{\langle *,\Box,\Box\rangle}$, and $\leadsto_d$ is $\dot{\Pi}_{\langle \Box,\Box,\Box\rangle}$, using the terminology of Barendregt's λ-cube [4], the axiom ($\forall$) expresses polymorphism, the axiom (π) dependent types, and the axiom ($\leadsto_d$) type constructors. Note that these constants have similar types.

So if Γ is a context and A is a term A in the Calculus of constructions then A is inhabited in Γ in the Calculus of constructions if and only if the translation of A in $\lambda\Pi/\equiv$ is inhabited in the translation of Γ in $\lambda\Pi/\equiv$ [13, 3]. In the translation of Γ in $\lambda\Pi/\equiv$, variables have a $\lambda\Pi/\equiv$ type of the form *Prf* u or *El* u, and none of them can have the type *Set*. But, in $\lambda\Pi/\equiv$, nothing prevents from declaring a variable of type *Set*. So, the formulation of the Calculus of constructions in $\lambda\Pi/\equiv$ is in fact a conservative extension of the original formulation of the Calculus of constructions, where the judgement $x : Set \vdash x : Set$ can be derived. Allowing the declaration of variables of type Set in the Calculus of constructions usually requires to add a sort $\triangle$ and an axiom $\Box : \triangle$ [22]. This is not needed here.

The Calculus of constructions with a type ι. Adding the axiom (ι) to the Calculus of constructions yields a sub-theory with the 10 axioms (Set), (El), (ι), $(Prop)$, (Prf), $(\Rightarrow_d)$, $(\forall)$, (o), $(\leadsto_d)$, and (π). It corresponds to the Calculus of constructions with an extra constant ι of type $\Box$. Adding a constant of type Set in $\lambda\Pi/\equiv$, like adding variables of type Set does not require to introduce an extra sort $\triangle$.

Some developments in the Calculus of constructions choose to declare the types of mathematical objects such as ι, nat, etc. in $*$, that would correspond to $\iota : Prop$, fully identifying types and propositions. We did not make this choice in the theory $\mathcal{U}$, because, then, the type ι of the constant 0 has type $*$ and the type $\iota \to *$ of the constant $positive$ has type $\Box$, while, in Simple type theory, both ι and $\iota \to o$ are simple types. So the expression of the simple type $\iota \to o$ requires type constructors and not dependent types. Dependent types, the constant π, are thus marginalized to type functions mapping proofs to terms.

In the Calculus of constructions with a constant ι of type $\Box$, there are no dependent types and no polymorphism at the object level, the latter leading to an inconsistent system [30, 11]. There are no object-level dependent types in the theory $\mathcal{U}$, that is the type $El\ \iota \to Set$ of the symbol $array$ is not equivalent to a term of the form $\varepsilon_\triangle\ A$, but such dependent types could be added. Polymorphism is discussed below.

The Minimal sub-theory. Adding the axioms $(\Rightarrow)$ and $(\leadsto)$ yields a sub-theory with the 12 axioms (Set), (El), (ι), $(Prop)$, (Prf), $(\Rightarrow)$, $(\forall)$, (o), $(\leadsto)$, $(\leadsto_d)$, $(\Rightarrow_d)$, and (π) called the "Minimal sub-theory" of the theory $\mathcal{U}$. It contains both the 10 axioms of the Calculus of constructions and the 9 axioms of Minimal simple type theory. It is a formulation of the Calculus of constructions where dependent and non dependent arrows are distinguished. A proof expressed in the Calculus of constructions can be expressed in this theory. In a proof, every symbol $\leadsto_d$ or $\Rightarrow_d$ that uses a dummy dependency can be replaced with a symbol $\leadsto$ or $\Rightarrow$. Every proof that does not use $\leadsto_d$, $\Rightarrow_d$ and π, can be expressed in Minimal simple type theory.

The Calculus of constructions with prenex predicative polymorphism. Adding the 5 axioms of prenex predicative polymorphism to the 10 axioms of the Calculus of constructions with a constant ι yields a sub-theory formed with the 15 axioms (Set), (El), (ι), $(Prop)$, (Prf), $(\Rightarrow_d)$, $(\forall)$, (o), $(\leadsto_d)$, (π), $(Scheme)$, (Els), $(\uparrow)$, $(\forall)$, and $(\dot\forall)$ defining the Calculus of constructions with prenex predicative polymorphism. It is a cumulative type system [5], containing four sorts $*$, $\Box$, $\triangle$ and $\diamond$, with $* : \Box$, $\Box : \triangle$, and $\Box \preceq \diamond$, and besides the rules $\langle *, *, * \rangle$, $\langle *, \Box, \Box \rangle$, $\langle \Box, *, * \rangle$, $\langle \Box, \Box, \Box \rangle$, a rule $\langle \triangle, \diamond, \diamond \rangle$ to quantify over a variable of type $\Box$ in a scheme and a rule $\langle \triangle, *, * \rangle$ to quantify over $\Box$ in a proposition [41].

6 Conclusion

The theory $\mathcal{U}$ is thus a candidate for a universal theory where proofs developed in various proof systems: HOL Light, Isabelle/HOL, HOL 4, Coq, Matita, Lean, PVS, etc. can be expressed. This theory can be complemented with other axioms to handle inductive types, co-inductive types, universes, etc. [2, 41, 21].

Each proof expressed in the theory $\mathcal{U}$ can use a sub-theory of the theory $\mathcal{U}$, as if the other axioms did not exist: the classical connectives do not impact the constructive ones, propositions as objects and functionality do not impact predicate logic, dependent types and predicate subtyping do not impact simple types, etc.

The proofs in the theory $\mathcal{U}$ can be classified according to the axioms they use, independently of the system they have been developed in. Finally, some proofs using classical connectives and quantifiers, propositions as objects, functionality, dependent types, or predicate subtyping may be translated into smaller fragments and used in systems different from the ones they have been developed in, making the theory $\mathcal{U}$ a tool to improve the interoperability between proof systems.

References

1 L. Allali and O. Hermant. Semantic A-translation and super-consistency entail classical cut elimination. *CoRR*, abs/1401.0998, 2014. `arXiv:1401.0998`.

2 A. Assaf. *A framework for defining computational higher-order logics*. PhD thesis, École polytechnique, 2015.

3 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. Manuscript, 2016.

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, pages 117–309. Oxford University Press, 1992.

5 B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université Paris 7, France, 1999.

6 S. Berardi. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube. Manuscript, 1988.

7 F. Blanqui. *Type theory and rewriting*. PhD thesis, Université Paris-Sud, France, 2001. URL: `http://hal.inria.fr/inria-00105525`.

8 F. Blanqui. Type Safety of Rewrite Rules in Dependent Types. In *Proceedings of the 5th International Conference on Formal Structures for Computation and Deduction*, Leibniz International Proceedings in Informatics 167, 2020. `doi:10.4230/LIPIcs.FSCD.2020.13`.

9 F. Blanqui and G. Hondet. Encoding of predicate subtyping and proof irrelevance in the $\lambda\pi$-calculus modulo theory. In *Proceedings of the 26th International Conference on Types for Proofs and Programs*, Leibniz International Proceedings in Informatics 188, 2021.

10 A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

11 Th. Coquand. An analysis of Girard's paradox. Technical Report RR-0531, Inria, 1986. URL: `https://hal.inria.fr/inria-00076023`.

12 Th. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2):95–120, 1988.

13 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. `doi:10.1007/978-3-540-73228-0_9`.

14 N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science. Volume B: Formal Models and Semantics*, chapter 6, pages 243–320. North-Holland, 1990.

15 A. Dorra. équivalence de curry-howard entre le $\lambda\Pi$ calcul et la logique intuitionniste. Internship report, 2010.

16 G. Dowek. On the definition of the classical connectives and quantifiers. In E.H. Haeusler, W. de Campos Sanz, and B. Lopes, editors, *Why is this a Proof?, Festschrift for Luiz Carlos Pereira*. College Publications, 2015.

17 G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31:33–72, 2003. doi:10.1023/A:1027357912519.

18 G. Dowek and B. Werner. Proof normalization modulo. *Journal of Symbolic Logic*, 68(4):1289–1316, 2003. doi:10.2178/jsl/1067620188.

19 G. Dowek and B. Werner. Arithmetic as a theory modulo. In Jürgen Giesl, editor, *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 423–437. Springer, 2005.

20 Gaspard Férey and François Thiré. Proof Irrelevance in LambdaPi Modulo Theory. https://eutypes.cs.ru.nl/eutypes_pmwiki/uploads/Main/books-of-abstracts-TYPES2019.pdf, 2019.

21 G. Genestier. *Dependently-Typed Termination and Embedding of Extensional Universe-Polymorphic Type Theory using Rewriting*. PhD thesis, Université Paris-Saclay, 2020.

22 H. Geuvers. The Calculus of Constructions and Higher Order Logic. In Ph. de Groote, editor, *The Curry-Howard isomorphism*, volume 8 of *Cahiers du Centre de logique*, pages 139–191. Université catholique de Louvain, 1995.

23 F. Gilbert. *Extending higher-order logic with predicate subtyping: Application to PVS. (Extension de la logique d'ordre supérieur avec le sous-typage par prédicats)*. PhD thesis, Sorbonne Paris Cité, France, 2018.

24 J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université de Paris VII, 1972.

25 É. Grienenberger. A logical system for an Ecumenical formalization of mathematics, 2019. Manuscript.

26 É. Grienenberger. Expressing Ecumenical systems in the lambda-pi-calculus modulo theory, 2021. In preparation.

27 R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. doi:10.1145/138027.138060.

28 D. Hilbert and W. Ackermann. *Grundzüge der theoretischen Logik*. Springer-Verlag, 1928.

29 G. Hondet and F. Blanqui. The New Rewriting Engine of Dedukti. In *Proceedings of the 5th International Conference on Formal Structures for Computation and Deduction*, Leibniz International Proceedings in Informatics 167, 2020. doi:10.4230/LIPIcs.FSCD.2020.35.

30 A. J. C. Hurkens. A simplification of Girard's paradox. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Typed Lambda Calculi and Applications*, pages 266–278, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

31 J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993. doi:10.1016/0304-3975(93)90091-7.

32 G. Nadathur and D. Miller. An overview of lambda-prolog. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*, pages 810–827, 1988.

33 Sam Owre and Natarajan Shankar. *The Formal Semantics of PVS*. SRI International, SRI International, Computer Science Laboratory, Menlo Park CA 94025 USA, 1997. URL: http://pvs.csl.sri.com/doc/semantics.pdf.

34 L.C. Paulson. Isabelle: The next 700 theorem provers. *CoRR*, cs.LO/9301106, 1993. arXiv:cs.LO/9301106.

35 L.C. Pereira and R.O. Rodriguez. Normalization, soundness and completeness for the propositional fragment of Prawitz'Ecumenical system. *Revista Portuguesa de Filosofia*, 73(3-4):1153–1168, 2017.

36 D. Prawitz. Classical versus intuitionistic logic. In E.H. Haeusler, W. de Campos Sanz, and B. Lopes, editors, *Why is this a Proof?, Festschrift for Luiz Carlos Pereira*. College Publications, 2015.

37 J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, pages 408–425. Springer, 1974.

38 TeReSe. *Term rewriting systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

39 J. Terlouw. Een nadere bewijstheoretische analyse van GSTT's. Manuscript, 1989.

40 F. Thiré. Sharing a Library between Proof Assistants: Reaching out to the HOL Family. In Frédéric Blanqui and Giselle Reis, editors, *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP@FSCD 2018, Oxford, UK, 7th July 2018*, volume 274 of *EPTCS*, pages 57–71, 2018.

41 F. Thiré. *Interoperability between proof systems using the Dedukti logical framework*. PhD thesis, Université Paris-Saclay, France, 2020.

Non-Deterministic Functions as Non-Deterministic Processes

Joseph W. N. Paulus
University of Groningen, The Netherlands

Daniele Nantes-Sobrinho
University of Brasília, Brazil

Jorge A. Pérez
University of Groningen, The Netherlands
CWI, Amsterdam, The Netherlands

Abstract

We study encodings of the λ-calculus into the π-calculus in the unexplored case of calculi with *non-determinism* and *failures*. On the sequential side, we consider $\lambda^{\lightning}_{\oplus}$, a new non-deterministic calculus in which intersection types control resources (terms); on the concurrent side, we consider $\mathsf{s}\pi$, a π-calculus in which non-determinism and failure rest upon a Curry-Howard correspondence between linear logic and session types. We present a typed encoding of $\lambda^{\lightning}_{\oplus}$ into $\mathsf{s}\pi$ and establish its correctness. Our encoding precisely explains the interplay of non-deterministic and fail-prone evaluation in $\lambda^{\lightning}_{\oplus}$ via typed processes in $\mathsf{s}\pi$. In particular, it shows how failures in sequential evaluation (absence/excess of resources) can be neatly codified as interaction protocols.

2012 ACM Subject Classification Theory of computation → Type structures; Theory of computation → Process calculi

Keywords and phrases Resource calculi, π-calculus, intersection types, session types, linear logic

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.21

Related Version Online appendix with omitted proofs and further examples:
Full Version: `https://arxiv.org/abs/2104.14759` [22]

Funding Paulus and Pérez have been partially supported by the Dutch Research Council (NWO) under project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software).

Acknowledgements We are grateful to the anonymous reviewers for their careful reading and constructive remarks.

1 Introduction

Milner's seminal work on encodings of the λ-calculus into the π-calculus [18] explains how *interaction* in π subsumes *evaluation* in λ. It opened a research strand on formal connections between sequential and concurrent calculi, covering untyped and typed regimes (see, e.g., [23, 4, 1, 25, 16, 26]). This paper extends this line of work by tackling a hitherto unexplored angle, namely encodability of calculi in which computation is *non-deterministic* and may be subject to *failures* – two relevant features in sequential and concurrent programming models.

We focus on *typed* calculi and study how non-determinism and failures interact with *resource-aware* computation. In sequential calculi, *non-idempotent intersection types* [2] offer one fruitful perspective at resource-awareness. Because non-idempotency distinguishes between types σ and $\sigma \wedge \sigma$, this class of intersection types can "count" different resources and enforce quantitative guarantees. In concurrent calculi, resource-awareness has been much studied using *linear types*. Linearity ensures that process actions occur exactly once, which is key to enforce protocol correctness. To our knowledge, connections between calculi adopting these two distinct views of resource-awareness via types are still to be established. We aim to develop such connections by relating models of sequential and concurrent computation.

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi; Article No. 21; pp. 21:1–21:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

On the sequential side, we introduce $\lambda^{\lightning}_{\oplus}$: a λ-calculus with resources, non-determinism, and failures, which distills key elements from λ-calculi studied in [3, 21] (§ 2). Evaluation in $\lambda^{\lightning}_{\oplus}$ considers *bags* of resources, and determines alternative executions governed by non-determinism. Failure results from a lack or excess of resources (terms), and is captured by the term $\mathtt{fail}^{\widetilde{x}}$ (for some variables $\widetilde{x}$). Non-determinism is *non-collapsing*: given M and N with reductions $M \longrightarrow M'$ and $N \longrightarrow N'$, the non-deterministic sum $M + N$ reduces to $M' + N'$. (Under a *collapsing* view, as in, e.g., [8], $M + N$ reduces to either M or N.)

On the concurrent side, we consider $\mathsf{s}\pi$: a session π-calculus with (non-collapsing) non-determinism and failure proposed in [6] (§ 3). Processes in $\mathsf{s}\pi$ are disciplined by *session types* that specify the protocols that the channels of a process must respect. Exploiting linearity, session types ensure absence of communication errors and stuck processes; $\mathsf{s}\pi$ rests upon a Curry-Howard correspondence between session types and (classical) linear logic extended with two modalities that express *non-deterministic protocols* that may succeed or fail.

Contributions. This paper presents the following contributions:

1. **The resource calculus $\boldsymbol{\lambda^{\lightning}_{\oplus}}$**, a new calculus that distills the distinguishing elements from previous resource calculi [4, 21], while offering an explicit treatment of failures in a setting with non-collapsing non-determinism. Using intersection types, we define well-typed (fail-free) expressions and well-formed (fail-prone) expressions in $\lambda^{\lightning}_{\oplus}$ (see below).
2. **An encoding of $\boldsymbol{\lambda^{\lightning}_{\oplus}}$ into $\boldsymbol{\mathsf{s}\pi}$**, proven correct following established criteria [11, 17] (§ 4). These criteria attest to an encoding's quality; we consider *type preservation*, *operational correspondence*, *success sensitiveness*, and *compositionality*. Thanks to these correctness properties, our encoding precisely describes how typed interaction protocols can codify sequential evaluation in which the absence and excess of resources may lead to failures.

These contributions entail different challenges. The first is bridging the different mechanisms for resource-awareness involved (intersection types in $\lambda^{\lightning}_{\oplus}$, session types in $\mathsf{s}\pi$). A direct encoding of $\lambda^{\lightning}_{\oplus}$ into $\mathsf{s}\pi$ is far from obvious, as multiple occurrences of a variable in $\lambda^{\lightning}_{\oplus}$ must be accommodated into the linear setting of $\mathsf{s}\pi$. To overcome this, we introduce $\widehat{\lambda}^{\lightning}_{\oplus}$: a variant of $\lambda^{\lightning}_{\oplus}$ with *sharing* [13, 10]. This way, we "atomize" occurrences of the same variable, thus simplifying the task of encoding $\lambda^{\lightning}_{\oplus}$ expressions into $\mathsf{s}\pi$ processes.

Another challenge is framing failures (undesirable computations) in $\lambda^{\lightning}_{\oplus}$ as well-typed $\mathsf{s}\pi$ processes. We define *well-formed* $\lambda^{\lightning}_{\oplus}$ expressions, which can lead to failure, in two stages. First, we consider $\lambda_{\oplus}$, the sub-language of $\lambda^{\lightning}_{\oplus}$ without $\mathtt{fail}^{\widetilde{x}}$. We give an intersection type system for $\lambda_{\oplus}$ to regulate fail-free evaluation. Well-formed expressions are defined on top of well-typed $\lambda_{\oplus}$ expressions. We show that $\mathsf{s}\pi$ can correctly encode the fail-free $\lambda_{\oplus}$ but, much more interestingly, also well-formed $\lambda^{\lightning}_{\oplus}$ expressions, which are fail-prone by definition.

Discussion about our approach and results, and comparisons with related works is in § 5.

2 $\lambda^{\lightning}_{\oplus}$: A λ-calculus with Non-Determinism and Failure

The syntax of $\lambda^{\lightning}_{\oplus}$ combines elements from calculi studied by Boudol and Laneve [4] and by Pagani and Ronchi della Rocca [21]. We use $x, y, \ldots$ to range over the set of *variables*. We write $\widetilde{x}$ to denote the sequence of pairwise distinct variables $x_1, \ldots, x_k$, for some $k \geq 0$. We write $|\widetilde{x}|$ to denote the length of $\widetilde{x}$.

▶ **Definition 1** (Syntax of $\lambda^{\lightning}_{\oplus}$). *The $\lambda^{\lightning}_{\oplus}$ calculus is defined by the following grammar:*

$$\begin{array}{llcl} \textit{(Terms)} & M, N, L & ::= & x \mid \lambda x.M \mid (M\ B) \mid M\langle\!\langle B/x \rangle\!\rangle \mid \mathtt{fail}^{\widetilde{x}} \\ \textit{(Bags)} & A, B & ::= & \mathtt{1} \mid \wr M \wr \mid A \cdot B \\ \textit{(Expressions)} & \mathbb{M}, \mathbb{N}, \mathbb{L} & ::= & M \mid \mathbb{M} + \mathbb{N} \end{array}$$

We have three syntactic categories: *terms* (in functional position); *bags* (in argument position), which denote multisets of resources; and *expressions*, which are finite formal sums that represent possible results of a computation. Terms are unary expressions: they can be variables, abstractions, and applications. Following [3, 4], the *explicit substitution* of a bag B for a variable x, written $\langle\!\langle B/x \rangle\!\rangle$, is also a term. The term $\mathtt{fail}^{\widetilde{x}}$ results from a reduction in which there is a lack or excess of resources to be substituted, where $\widetilde{x}$ denotes a multiset of free variables that are encapsulated within failure.

The empty bag is denoted $\mathtt{1}$. The bag enclosing the term M is $\wr M \wr$. The concatenation of bags B_1 and B_2 is $B_1 \cdot B_2$; this is a commutative and associative operation, where $\mathtt{1}$ is the identity. We treat expressions as *sums*, and use notations such as $\sum_i^n N_i$ for them. Sums are associative and commutative; reordering of the terms in a sum is performed silently.

▶ **Notation 2** (Expressions). *Notation $N \in \mathbb{M}$ denotes that N is part of the sum denoted by $\mathbb{M}$. Similarly, we write $N_i \in B$ to denote that N_i occurs in the bag B, and $B \setminus N_i$ to denote the bag that is obtained by removing one occurrence of the term N_i from B.*

Full details on the reduction semantics and typing system for $\lambda^{\lightning}_{\oplus}$ can be found in the appendix and [22].

A Resource Calculus With Sharing

We define a variant of $\lambda^{\lightning}_{\oplus}$ with sharing variables, dubbed $\widehat{\lambda}^{\lightning}_{\oplus}$, inspired by the work by Gundersen et al. [13] and Ghilezan et al. [10]. In § 4 we shall use $\widehat{\lambda}^{\lightning}_{\oplus}$ as intermediate language in our encoding of $\lambda^{\lightning}_{\oplus}$ into $\mathsf{s}\pi$.

The syntax of $\widehat{\lambda}^{\lightning}_{\oplus}$ only modifies the syntax of $\lambda^{\lightning}_{\oplus}$-terms, which is defined by the grammar below; the syntax of bags B and expressions $\mathbb{M}$ is as in Def. 1.

$$\begin{array}{lll} \text{(Terms)} & M, N, L ::= & x \mid \lambda x.(M[\widetilde{x} \leftarrow x]) \mid (M\ B) \mid M\langle\!| N/x |\!\rangle \mid \mathtt{fail}^{\widetilde{x}} \\ & & \mid M[\widetilde{x} \leftarrow x] \mid (M[\widetilde{x} \leftarrow x])\langle\!\langle B/x \rangle\!\rangle \end{array}$$

We consider the *sharing construct* $M[\widetilde{x} \leftarrow x]$ and the *explicit linear substitution* $M\langle\!| N/x |\!\rangle$. The term $M[\widetilde{x} \leftarrow x]$ defines the sharing of variables $\widetilde{x}$ occurring in M using x. We shall refer to x as *sharing variable* and to $\widetilde{x}$ as *shared variables*. A variable is only allowed to appear once in a term. Notice that $\widetilde{x}$ can be empty: $M[\leftarrow x]$ expresses that x does not share any variables in M. As in $\lambda^{\lightning}_{\oplus}$, the term $\mathtt{fail}^{\widetilde{x}}$ explicitly accounts for failed attempts at substituting the variables $\widetilde{x}$, due to an excess or lack of resources. There is a difference with respect to $\lambda^{\lightning}_{\oplus}$: in the term $\mathtt{fail}^{\widetilde{x}}$, $\widetilde{x}$ denotes a set (rather than a multiset) of variables, which may include shared variables.

In $M[\widetilde{x} \leftarrow x]$ we require that (i) every $x_i \in \widetilde{x}$ must occur exactly once in M and that (ii) x_i is not a sharing variable. The occurrence of x_i can appear within the fail term $\mathtt{fail}^{\widetilde{y}}$, if $x_i \in \widetilde{y}$. In the explicit linear substitution $M\langle\!| N/x |\!\rangle$, we require: (i) the variable x has to occur in M; (ii) x cannot be a sharing variable; and (iii) x cannot be in an explicit linear substitution occurring in M. For instance, $M'\langle\!| L/x |\!\rangle\langle\!| N/x |\!\rangle$ is not a valid term in $\widehat{\lambda}^{\lightning}_{\oplus}$.

To define the reduction semantics of $\widehat{\lambda}^{\lightning}_{\oplus}$, we require some auxiliary notions: the free variables of an expression/term, the head of a term, and linear head substitution.

$$\begin{array}{lll} \mathsf{fv}(x) = \{x\} & \mathsf{fv}(\mathtt{fail}^{\widetilde{x}}) = \{\widetilde{x}\} & \mathsf{fv}(\wr M \wr) = \mathsf{fv}(M) \\ \mathsf{fv}(B_1 \cdot B_2) = \mathsf{fv}(B_1) \cup \mathsf{fv}(B_2) & \mathsf{fv}(M\ B) = \mathsf{fv}(M) \cup \mathsf{fv}(B) & \mathsf{fv}(1) = \emptyset \end{array}$$

$$\begin{array}{ll} \mathsf{fv}(M \langle\!| N/x |\!\rangle) = (\mathsf{fv}(M) \setminus \{x\}) \cup \mathsf{fv}(N) & \mathsf{fv}(M[\widetilde{x} \leftarrow x]) = (\mathsf{fv}(M) \setminus \{\widetilde{x}\}) \cup \{x\} \\ \mathsf{fv}(\lambda x.(M[\widetilde{x} \leftarrow x])) = \mathsf{fv}(M[\widetilde{x} \leftarrow x]) \setminus \{x\} & \mathsf{fv}(\mathbb{M} + \mathbb{N}) = \mathsf{fv}(\mathbb{M}) \cup \mathsf{fv}(\mathbb{N}) \end{array}$$

$$\mathsf{fv}((M[\widetilde{x} \leftarrow x])\langle\!\langle B/x \rangle\!\rangle) = (\mathsf{fv}(M[\widetilde{x} \leftarrow x]) \setminus \{x\}) \cup \mathsf{fv}(B)$$

Figure 1 Free variables for $\widehat{\lambda}^{\lightning}_{\oplus}$.

▶ **Definition 3** (Free Variables). *The set of free variables of a term, bag and expressions in $\widehat{\lambda}^{\lightning}_{\oplus}$, is defined in Fig. 1. As usual, a term M is* closed *if $\mathsf{fv}(M) = \emptyset$.*

▶ **Notation 4.** *We write $\mathsf{PER}(B)$ to denote the set of all permutations of bag B. Also, $B_i(n)$ denotes the n-th term in the (permuted) B_i. We define $\mathsf{size}(B)$ to denote the number of terms in bag B. That is, $\mathsf{size}(1) = 0$ and $\mathsf{size}(\wr M \wr \cdot B) = 1 + \mathsf{size}(B)$.*

▶ **Definition 5** (Head). *The head of a term M, denoted $\mathsf{head}(M)$, is defined inductively:*

$$\begin{array}{ll} \mathsf{head}(x) = x & \mathsf{head}(\lambda x.(M[\widetilde{x} \leftarrow x])) = \lambda x.(M[\widetilde{x} \leftarrow x]) \\ \mathsf{head}(M\ B) = \mathsf{head}(M) & \mathsf{head}(M \langle\!| N/x |\!\rangle) = \mathsf{head}(M) \\ \mathsf{head}(\mathtt{fail}^{\widetilde{x}}) = \mathtt{fail}^{\widetilde{x}} & \end{array}$$

$$\mathsf{head}(M[\widetilde{x} \leftarrow x]) = \begin{cases} x & \text{If } \mathsf{head}(M) = y \text{ and } y \in \widetilde{x} \\ \mathsf{head}(M) & \text{Otherwise} \end{cases}$$

$$\mathsf{head}((M[\widetilde{x} \leftarrow x])\langle\!\langle B/x \rangle\!\rangle) = \begin{cases} \mathtt{fail}^{\emptyset} & \text{If } |\widetilde{x}| \neq \mathsf{size}(B) \\ \mathsf{head}(M[\widetilde{x} \leftarrow x]) & \text{Otherwise} \end{cases}$$

▶ **Definition 6** (Linear Head Substitution). *Given a term M with $\mathsf{head}(M) = x$, the linear substitution of a term N for x in M, written $M\{\!|N/x|\!\}$ is inductively defined as:*

$$\begin{array}{rcll} x\{\!|N/x|\!\} &=& N & \\ (M\ B)\{\!|N/x|\!\} &=& (M\{\!|N/x|\!\})\ B & \\ (M \langle\!| L/y |\!\rangle)\{\!|N/x|\!\} &=& (M\{\!|N/x|\!\})\ \langle\!| L/y |\!\rangle & x \neq y \\ ((M[\widetilde{y} \leftarrow y])\langle\!\langle B/y \rangle\!\rangle)\{\!|N/x|\!\} &=& (M[\widetilde{y} \leftarrow y]\{\!|N/x|\!\})\ \langle\!\langle B/y \rangle\!\rangle & x \neq y \\ (M[\widetilde{y} \leftarrow y])\{\!|N/x|\!\} &=& (M\{\!|N/x|\!\})[\widetilde{y} \leftarrow y] & x \neq y \end{array}$$

We now define contexts for terms and expressions in $\widehat{\lambda}^{\lightning}_{\oplus}$. Term contexts involve an explicit linear substitution, rather than an explicit substitution: this is due to the reduction strategy we have chosen to adopt, as we always wish to evaluate explicit substitutions first. Expression contexts can be seen as sums with holes. We assume that the terms that fill in the holes respect the conditions on explicit linear substitutions (i.e., variables appear in a term only once, shared variables must occur in the context).

▶ **Definition 7** (Term and Expression Contexts in $\widehat{\lambda}^{\lightning}_{\oplus}$). *Let $[\cdot]$ denote a hole. Contexts for terms and expressions are defined by the following grammar:*

$$\begin{array}{lcl} C[\cdot], C'[\cdot] &::=& ([\cdot])B \mid ([\cdot])\langle\!| N/x |\!\rangle \mid ([\cdot])[\widetilde{x} \leftarrow x] \mid ([\cdot])[\leftarrow x]\langle\!\langle 1/x \rangle\!\rangle \\ D[\cdot], D'[\cdot] &::=& M + [\cdot] \mid [\cdot] + M \end{array}$$

The substitution of a hole with term M in a context $C[\cdot]$, denoted $C[M]$, must be a $\widehat{\lambda}^{\lightning}_{\oplus}$-term.

$$[\texttt{RS:Beta}]\ \frac{}{(\lambda x.M[\widetilde{x} \leftarrow x])B \longrightarrow M[\widetilde{x} \leftarrow x]\langle\!\langle B/x \rangle\!\rangle}$$

$$[\texttt{RS:Ex-Sub}]\ \frac{B = \wr M_1 \int \cdots \wr M_k \int \qquad k \geq 1 \qquad M \neq \texttt{fail}^{\widetilde{y}}}{M[x_1,\ldots,x_k \leftarrow x]\langle\!\langle B/x \rangle\!\rangle \longrightarrow \sum_{B_i \in \mathsf{PER}(B)} M\langle\!| B_i(1)/x_1 |\!\rangle \cdots \langle\!| B_i(k)/x_k |\!\rangle}$$

$$[\texttt{RS:Lin-Fetch}]\ \frac{\mathsf{head}(M) = x}{M\langle\!| N/x |\!\rangle \longrightarrow M\{\!| N/x |\!\}}$$

$$[\texttt{RS:Fail}]\ \frac{k \neq \mathsf{size}(B) \qquad \widetilde{y} = (\mathsf{fv}(M) \setminus \{x_1,\ldots,x_k\}) \cup \mathsf{fv}(B)}{M[x_1,\ldots,x_k \leftarrow x]\langle\!\langle B/x \rangle\!\rangle \longrightarrow \sum_{\mathsf{PER}(B)} \texttt{fail}^{\widetilde{y}}}$$

$$[\texttt{RS:Cons}_1]\ \frac{\widetilde{y} = \mathsf{fv}(B)}{\texttt{fail}^{\widetilde{x}}\ B \longrightarrow \sum_{\mathsf{PER}(B)} \texttt{fail}^{\widetilde{x}\cup\widetilde{y}}} \qquad [\texttt{RS:Cons}_2]\ \frac{\mathsf{size}(B) = k \quad k + |\widetilde{x}| \neq 0 \quad \widetilde{z} = \mathsf{fv}(B)}{(\texttt{fail}^{\widetilde{x}\cup\widetilde{y}}[\widetilde{x} \leftarrow x])\langle\!\langle B/x \rangle\!\rangle \longrightarrow \sum_{\mathsf{PER}(B)} \texttt{fail}^{\widetilde{y}\cup\widetilde{z}}}$$

$$[\texttt{RS:Cons}_3]\ \frac{\widetilde{z} = \mathsf{fv}(N)}{\texttt{fail}^{\widetilde{y}\cup x}\langle\!| N/x |\!\rangle \longrightarrow \texttt{fail}^{\widetilde{y}\cup\widetilde{z}}}$$

$$[\texttt{RS:TCont}]\ \frac{M \longrightarrow M_1' + \cdots + M_k'}{C[M] \longrightarrow C[M_1'] + \cdots + C[M_k']} \qquad [\texttt{RS:ECont}]\ \frac{\mathbb{M} \longrightarrow \mathbb{M}'}{D[\mathbb{M}] \longrightarrow D[\mathbb{M}']}$$

Figure 2 Reduction rules for $\widehat{\lambda}^{\sharp}_{\oplus}$.

This way, e.g., the hole in context $C[\cdot] = ([\cdot])\langle\!| N/x |\!\rangle$ cannot be filled with y, since $C[y] = (y)\langle\!| N/x |\!\rangle$ is not a well-defined term. Indeed, $M\langle\!| N/x |\!\rangle$ requires that x occurs exactly once within M. Similarly, we cannot fill the hole with $\texttt{fail}^{z}$ with $z \neq x$, since $C[\texttt{fail}^{z}] = (\texttt{fail}^{z})\langle\!| N/x |\!\rangle$ is also not a well-defined term, for the same reason.

Reduction Semantics

The reduction relation $\longrightarrow$ operates lazily on expressions; it is defined by the rules in Fig. 2. A β-reduction in $\widehat{\lambda}^{\sharp}_{\oplus}$ results into an explicit substitution $\langle\!\langle B/x \rangle\!\rangle$, which then evolves into a linear head substitution $\{\!| N/x |\!\}$ (with $N \in B$). Reduction in $\widehat{\lambda}^{\sharp}_{\oplus}$ introduces an intermediate step whereby the explicit substitution expands into a sum of terms involving explicit linear substitutions $\langle\!| N/x |\!\rangle$, which are the ones to reduce into a linear head substitution. In the case there is a mismatch between the size of B and the number of shared variables to be substituted, the term reduces to failure.

More specifically, Rule [RS:Beta] is standard and results into an explicit substitution. Rule [RS:Ex-Sub] applies when the size k of the bag coincides with the length of $\widetilde{x} = x_1,\ldots,x_k$. Intuitively, this rule "distributes" an explicit substitution into a sum of terms involving explicit linear substitutions; it considers all possible permutations of the elements in the bag among all shared variables. Rule [RS:Lin-Fetch] specifies the evaluation of a term with an explicit linear substitution into a linear head substitution.

There are three rules reduce to the failure term: their objective is to accumulate all (free) variables involved in failed reductions. Accordingly, Rule [RS:Fail] formalizes failure in the evaluation of an explicit substitution $M[\widetilde{x} \leftarrow x]\langle\!\langle B/x \rangle\!\rangle$, which occurs if there is a mismatch between the resources (terms) present in B and the number of occurrences of

$$
\begin{array}{rl}
M[\leftarrow x]\langle\!\langle 1/x\rangle\!\rangle \succeq_\lambda M & \\
MB\langle\!| N/x |\!\rangle \equiv_\lambda (M\langle\!| N/x|\!\rangle)B & \text{with } x \notin \mathsf{fv}(B) \\
M\langle\!| N_2/y|\!\rangle\langle\!| N_1/x|\!\rangle \equiv_\lambda M\langle\!| N_1/x|\!\rangle\langle\!| N_2/y|\!\rangle & \text{with } x \notin \mathsf{fv}(N_2) \\
MA[\widetilde{x} \leftarrow x]\langle\!\langle B/x\rangle\!\rangle \equiv_\lambda (M[\widetilde{x} \leftarrow x]\langle\!\langle B/x\rangle\!\rangle)A & \text{with } x_i \in \widetilde{x} \Rightarrow x_i \notin \mathsf{fv}(A) \\
M[\widetilde{y} \leftarrow y]\langle\!\langle A/y\rangle\!\rangle[\widetilde{x} \leftarrow x]\langle\!\langle B/x\rangle\!\rangle \succeq_\lambda & \\
(M[\widetilde{x} \leftarrow x]\langle\!\langle B/x\rangle\!\rangle)[\widetilde{y} \leftarrow y]\langle\!\langle A/y\rangle\!\rangle & \text{with } x_i \in \widetilde{x} \Rightarrow x_i \notin \mathsf{fv}(A) \\
C[M] \succeq_\lambda C[M'] & \text{with } M \succeq_\lambda M' \\
D[\mathbb{M}] \succeq_\lambda D[\mathbb{M}'] & \text{with } \mathbb{M} \succeq_\lambda \mathbb{M}'
\end{array}
$$

Figure 3 Precongruence in $\widehat{\lambda}^{\lightning}_{\oplus}$.

x to be substituted. The resulting failure term preserves all free variables in M and B within its attached set $\widetilde{y}$. Rules [RS:Cons$_1$] and [RS:Cons$_2$] describe reductions that lazily consume the failure term, when a term has $\mathtt{fail}^{\widetilde{x}}$ at its head position. The former rule consumes bags attached to it whilst preserving all its free variables. Finally, Rule [RS:Cons$_3$] accumulates into the failure term the free variables involved in an explicit linear substitution. The contextual rules [RS:TCont] and [RS:Econt] are standard.

▶ **Notation 8.** *As standard, $\longrightarrow$ denotes one step reduction; $\longrightarrow^+$ and $\longrightarrow^*$ denote the transitive and the reflexive-transitive closure of $\longrightarrow$, respectively. We write $\mathbb{N} \longrightarrow_{[\mathtt{R}]} \mathbb{M}$ to denote that* [R] *is the last (non-contextual) rule used in inferring the step from $\mathbb{N}$ to $\mathbb{M}$.*

▶ **Example 9.** We show how a term can reduce using Rule [RS:Cons$_2$].

$$
\begin{array}{l}
(\lambda x.x_1[x_1 \leftarrow x])\wr \mathtt{fail}^{\emptyset}[\leftarrow y]\langle\!\langle \wr N \wr/y\rangle\!\rangle \wr \longrightarrow_{[\mathtt{RS:Beta}]} x_1[x_1 \leftarrow x]\langle\!\langle \wr \mathtt{fail}^{\emptyset}[\leftarrow y]\langle\!\langle \wr N\wr/y\rangle\!\rangle \wr/x\rangle\!\rangle \\
\longrightarrow_{[\mathtt{RS:Ex\text{-}Sub}]} x_1\langle\!| \mathtt{fail}^{\emptyset}[\leftarrow y]\langle\!\langle \wr N\wr/y\rangle\!\rangle/x_1|\!\rangle \longrightarrow_{[\mathtt{RS:Lin\text{-}Fetch}]} \mathtt{fail}^{\emptyset}[\leftarrow y]\langle\!\langle \wr N\wr/y\rangle\!\rangle \longrightarrow_{[\mathtt{RS:Cons_2}]} \mathtt{fail}^{\mathsf{fv}(N)}
\end{array}
$$

⌟

Notice that the left-hand sides of the reduction rules in $\widehat{\lambda}^{\lightning}_{\oplus}$ do not interfere with each other. Reduction in $\widehat{\lambda}^{\lightning}_{\oplus}$ satisfies a *diamond property*; see [22].

A Precongruence

Fig. 3 defines a precongruence for $\widehat{\lambda}^{\lightning}_{\oplus}$ on terms and expressions, denoted $\succeq_\lambda$. We write $M \equiv_\lambda M'$ whenever both $M \succeq_\lambda M'$ and $M' \succeq_\lambda M$ hold.

▶ **Example 10.** We illustrate the precongruence in case of failure:

$$
\begin{array}{l}
(\lambda x.x_1[x_1 \leftarrow x])\wr \mathtt{fail}^{\emptyset}[\leftarrow y]\langle\!\langle 1/y\rangle\!\rangle \wr \longrightarrow_{[\mathtt{RS:Beta}]} x_1[x_1 \leftarrow x]\langle\!\langle \wr \mathtt{fail}^{\emptyset}[\leftarrow y]\langle\!\langle 1/y\rangle\!\rangle \wr/x\rangle\!\rangle \\
\longrightarrow_{[\mathtt{RS:Ex\text{-}Sub}]} x_1\langle\!| \mathtt{fail}^{\emptyset}[\leftarrow y]\langle\!\langle 1/y\rangle\!\rangle/x_1|\!\rangle \longrightarrow_{[\mathtt{RS:Lin\text{-}Fetch}]} \mathtt{fail}^{\emptyset}[\leftarrow y]\langle\!\langle 1/y\rangle\!\rangle \succeq_\lambda \mathtt{fail}^{\emptyset}
\end{array}
$$

In the last step, Rule [RS:Cons$_2$] cannot be applied: y is sharing with no shared variables and the explicit substitution involves the bag 1. ⌟

▶ **Example 11.** We illustrate how Rule [RS:Fail] can introduce $\mathtt{fail}^{\widetilde{x}}$ into a term. It also shows how Rule [RS:Cons$_3$] consumes an explicit linear substitution:

$$
\begin{array}{l}
x_1[\leftarrow y]\langle\!\langle \wr N\wr/y\rangle\!\rangle[x_1 \leftarrow x]\langle\!\langle \wr M\wr/x\rangle\!\rangle \longrightarrow_{[\mathtt{RS:Ex\text{-}Sub}]} x_1[\leftarrow y]\langle\!\langle \wr N\wr/y\rangle\!\rangle\langle\!| M/x_1|\!\rangle \\
\longrightarrow_{[\mathtt{RS:Fail}]} \mathtt{fail}^{\{x_1\}\cup\mathsf{fv}(N)}\langle\!| M/x_1|\!\rangle \longrightarrow_{[\mathtt{RS:Cons_3}]} \mathtt{fail}^{\mathsf{fv}(M)\cup\mathsf{fv}(N)}
\end{array}
$$

⌟

Intersection Types

We define a type system for $\widehat{\lambda}^{\lightning}_{\oplus}$ based on non-idempotent intersection types, similar to the one defined by Bucciarelli et al. in [5]. Intersection types allow us to reason about types of resources in bags but also about every occurrence of a variable. That is, non-idempotent intersection types enable us to distinguish expressions not only by measuring the size of a bag but also by counting the number of times a variable occurs within a term.

▶ **Definition 12** (Types for $\widehat{\lambda}^{\lightning}_{\oplus}$). *We define* strict *and* multiset types *by the grammar:*

$$\textit{(Strict)} \quad \sigma, \tau, \delta ::= \mathbf{unit} \mid \pi \to \sigma \qquad\qquad \textit{(Multiset)} \quad \pi, \zeta ::= \bigwedge_{i\in I} \sigma_i \mid \omega$$

A strict type can be the unit type **unit** or a functional type $\pi \to \sigma$, where π is a multiset type and σ is a strict type. Multiset types can be either the empty type ω or an intersection of strict types $\bigwedge_{i\in I} \sigma_i$, with I non-empty. The operator $\wedge$ is commutative, associative, and non-idempotent, that is, $\sigma \wedge \sigma \neq \sigma$. The empty type is the type of the empty bag and acts as the identity element to $\wedge$.

Type assignments range over $\Gamma, \Delta, \ldots$ and have the form $\Gamma, x : \sigma$, assigning the empty type to all but a finite number of variables. Multiple occurrences of a variable can occur within an assignment; they are assigned only strict types. For instance, $x : \tau \to \tau, x : \tau$ is a valid type assignment: it means that x can be of both type $\tau \to \tau$ and τ. The multiset of variables in Γ is denoted as $\mathtt{dom}(\Gamma)$. *Type judgements* are of the form $\Gamma \vdash \mathbb{M} : \sigma$, where Γ consists of variable type assignments, and $\mathbb{M} : \sigma$ means that $\mathbb{M}$ has type σ. We write $\vdash \mathbb{M} : \sigma$ to denote $\emptyset \vdash \mathbb{M} : \sigma$.

▶ **Notation 13.** *Given $k \geq 0$, we shall write σ^k to stand for $\sigma \wedge \cdots \wedge \sigma$ (k times, if $k > 0$) or for ω (if $k = 0$). Similarly, we write $\hat{x} : \sigma^k$ to stand for $x : \sigma, \cdots, x : \sigma$ (k times, if $k > 0$) or for $x : \omega$ (if $k = 0$).*

We define *well-formed* $\widehat{\lambda}^{\lightning}_{\oplus}$ expressions, in two stages. We first consider the type system given in Fig. 4 for $\widehat{\lambda}_{\oplus}$, the sub-calculus of $\widehat{\lambda}^{\lightning}_{\oplus}$ *without* the failure term $\mathtt{fail}^{\widetilde{x}}$. Then, we define well-formed expressions for the full language $\widehat{\lambda}^{\lightning}_{\oplus}$ via Def. 14 (see below).

We first discuss selected rules of the type system for $\widehat{\lambda}_{\oplus}$, which takes into account the sharing construct $M[\widetilde{x} \leftarrow x]$. Rule [$\mathtt{TS{:}var}$] is standard. Rule [$\mathtt{TS{:}1}$] assigns the empty bag $\mathtt{1}$ the empty type ω. The weakening rule [$\mathtt{TS{:}weak}$] deals with $k = 0$, typing the term $M[\leftarrow x]$, when there are no occurrences of x in M, as long as M is typable. Rule [$\mathtt{TS{:}abs\text{-}sh}$] is as expected: it requires that the sharing variable is assigned the k-fold intersection type σ^k (Not. 13). Rule [$\mathtt{TS{:}app}$] is standard, requiring a match on the multiset type π. Rule [$\mathtt{TS{:}bag}$] types the concatenation of bags. Rule [$\mathtt{TS{:}ex\text{-}lin\text{-}sub}$] supports explicit linear substitutions. Rule [$\mathtt{TS{:}ex\text{-}sub}$] types explicit substitutions where a bag must consist of both the same type and length of the shared variable it is being substituted for. Rule [$\mathtt{TS{:}sum}$] types the sum of two expressions of the same type. Rule [$\mathtt{TS{:}share}$] requires that the shared variables $x_1, \ldots, x_k$ have the same type as the sharing variable x, for $k \neq 0$.

On top of this type system for $\widehat{\lambda}_{\oplus}$, we define well-formed expressions: $\lambda^{\lightning}_{\oplus}$-terms whose computation may lead to failure.

▶ **Definition 14** (Well-formedness in $\widehat{\lambda}^{\lightning}_{\oplus}$). *An expression $\mathbb{M}$ is well formed if there exist Γ and τ such that $\Gamma \models \mathbb{M} : \tau$ is entailed via the rules in Fig. 5.*

Rules [$\mathtt{FS{:}wf\text{-}expr}$] and [$\mathtt{FS{:}wf\text{-}bag}$] guarantee that every well-typed expression and bag, respectively, is well-formed. Since our language is expressive enough to account for failing computations, we include rules for checking the structure of these ill-behaved terms – terms

$$[\texttt{TS:var}]\ \frac{}{x:\sigma \vdash x:\sigma} \qquad [\texttt{TS:1}]\ \frac{}{\vdash \mathtt{1}:\omega} \qquad [\texttt{TS:weak}]\ \frac{\Delta \vdash M:\tau}{\Delta, x:\omega \vdash M[\leftarrow x]:\tau}$$

$$[\texttt{TS:abs-sh}]\ \frac{\Delta, x:\sigma^k \vdash M[\widetilde{x} \leftarrow x]:\tau}{\Delta \vdash \lambda x.(M[\widetilde{x} \leftarrow x]):\sigma^k \to \tau} \qquad [\texttt{TS:app}]\ \frac{\Gamma \vdash M:\pi \to \tau \qquad \Delta \vdash B:\pi}{\Gamma, \Delta \vdash M\ B:\tau}$$

$$[\texttt{TS:bag}]\ \frac{\Gamma \vdash M:\sigma \qquad \Delta \vdash B:\sigma^k}{\Gamma, \Delta \vdash \wr M \wr \cdot B:\sigma^{k+1}} \qquad [\texttt{TS:ex-lin-sub}]\ \frac{\Delta \vdash N:\sigma \qquad \Gamma, x:\sigma \vdash M:\tau}{\Gamma, \Delta \vdash M\langle\!|N/x|\!\rangle:\tau}$$

$$[\texttt{TS:ex-sub}]\ \frac{\Delta \vdash B:\pi \qquad \Gamma, x:\pi \vdash M[\widetilde{x} \leftarrow x]:\tau}{\Gamma, \Delta \vdash M[\widetilde{x} \leftarrow x]\langle\!\langle B/x\rangle\!\rangle:\tau} \qquad [\texttt{TS:sum}]\ \frac{\Gamma \vdash \mathbb{M}:\sigma \qquad \Gamma \vdash \mathbb{N}:\sigma}{\Gamma \vdash \mathbb{M}+\mathbb{N}:\sigma}$$

$$[\texttt{TS:share}]\ \frac{\Delta, x_1:\sigma, \cdots, x_k:\sigma \vdash M:\tau \qquad x \notin \mathtt{dom}(\Delta) \qquad k \neq 0}{\Delta, x:\sigma^k \vdash M[x_1, \cdots, x_k \leftarrow x]:\tau}$$

Figure 4 Typing rules for $\widehat{\lambda}_{\oplus}$.

$$[\texttt{FS:wf-expr}]\ \frac{\Gamma \vdash \mathbb{M}:\tau}{\Gamma \models \mathbb{M}:\tau} \qquad [\texttt{FS:wf-bag}]\ \frac{\Gamma \vdash B:\pi}{\Gamma \models B:\pi} \qquad [\texttt{FS:weak}]\ \frac{\Gamma \models M:\tau}{\Gamma, x:\omega \models M[\leftarrow x]:\tau}$$

$$[\texttt{FS:abs-sh}]\ \frac{\Gamma, x:\sigma^k \models M[\widetilde{x} \leftarrow x]:\tau \quad x \notin \mathtt{dom}(\Gamma)}{\Gamma \models \lambda x.(M[\widetilde{x} \leftarrow x]):\sigma^k \to \tau} \qquad [\texttt{FS:fail}]\ \frac{\mathtt{dom}(\Gamma) = \widetilde{x}}{\Gamma \models \mathtt{fail}^{\widetilde{x}}:\tau}$$

$$[\texttt{FS:app}]\ \frac{\Gamma \models M:\sigma^j \to \tau \quad \Delta \models B:\sigma^k}{\Gamma, \Delta \models M\ B:\tau} \qquad [\texttt{FS:bag}]\ \frac{\Gamma \models M:\sigma \quad \Delta \models B:\sigma^k}{\Gamma, \Delta \models \wr M \wr \cdot B:\sigma^{k+1}}$$

$$[\texttt{FS:ex-lin-sub}]\ \frac{\Gamma, x:\sigma \models M:\tau \quad \Delta \models N:\sigma}{\Gamma, \Delta \models M\langle\!|N/x|\!\rangle:\tau} \qquad [\texttt{FS:sum}]\ \frac{\Gamma \models \mathbb{M}:\sigma \quad \Gamma \models \mathbb{N}:\sigma}{\Gamma \models \mathbb{M}+\mathbb{N}:\sigma}$$

$$[\texttt{FS:ex-sub}]\ \frac{\Gamma, x:\sigma^k \models M[\widetilde{x} \leftarrow x]:\tau \qquad \Delta \models B:\sigma^j}{\Gamma, \Delta \models M[\widetilde{x} \leftarrow x]\langle\!\langle B/x\rangle\!\rangle:\tau}$$

$$[\texttt{FS:share}]\ \frac{\Gamma, x_1:\sigma, \cdots, x_k:\sigma \models M:\tau \quad x \notin \mathtt{dom}(\Gamma) \quad k \neq 0}{\Gamma, x:\sigma^k \models M[x_1, \cdots, x_k \leftarrow x]:\tau}$$

Figure 5 Well-formedness rules for $\widehat{\lambda}^{\lightning}_{\oplus}$.

that can be well-formed, but not typable. For instance, Rules [FS:ex-sub] and [FS:app] differ from similar typing rules in Fig. 4: the size of the bags (as declared in their types) is no longer required to match. Also, Rule [FS:fail] has no analogue in the type system: we allow the failure term $\mathtt{fail}^{\widetilde{x}}$ to be well-formed with any type, provided that the context contains the types of the variables in $\widetilde{x}$. The other rules are self-explanatory.

Well-formed expressions satisfy subject reduction (SR); the proof is standard (cf. [22]).

▶ **Theorem 15** (SR in $\widehat{\lambda}^{\lightning}_{\oplus}$). *If* $\Gamma \models \mathbb{M}:\tau$ *and* $\mathbb{M} \longrightarrow \mathbb{M}'$ *then* $\Gamma \models \mathbb{M}':\tau$.

3 sπ: A Session-Typed π-Calculus

The π-calculus [19] is a model of concurrency in which *processes* interact via *names* (or *channels*) to exchange values, which can be themselves names. Here we overview sπ, introduced by Caires and Pérez in [6], in which *session types* [14, 15] ensure that the two endpoints of a channel perform matching actions: when one endpoint sends, the other receives; when an endpoint closes, the other closes too. Following [7, 27], sπ defines a Curry-Howard correspondence between session types and a linear logic with two dual modalities ($\&A$ and $\oplus A$), which define *non-deterministic* sessions. In sπ, cut elimination corresponds to process communication, proofs correspond to processes, and propositions correspond to session types.

Syntax and Semantics

We use $x, y, z, w \ldots$ to denote names implementing the *(session) endpoints* of protocols specified by session types. We consider the sub-language of [6] without labeled choices and replication, which is actually sufficient to encode $\widehat{\lambda}^{\lightning}_{\oplus}$.

▶ **Definition 16** (Processes). *The syntax of* sπ *processes is given by the grammar:*

$$\begin{aligned} P, Q ::= {} & \overline{x}(y).P \mid x(y).P \mid x.\overline{\texttt{close}} \mid x.\texttt{close}; P \mid [x \leftrightarrow y] \mid (P \mid Q) \mid (\nu x)P \mid \mathbf{0} \\ & \mid x.\overline{\texttt{some}}; P \mid x.\overline{\texttt{none}} \mid x.\texttt{some}_{(w_1,\cdots,w_n)}; P \mid P \oplus Q \end{aligned}$$

In the first line, an output process $\overline{x}(y).P$ sends a fresh name y along session x and then continues as P. An input process $x(y).P$ receives a name z along x and then continues as $P\{z/y\}$, which denotes the capture-avoiding substitution of z for y in P. Processes $x.\overline{\texttt{close}}$ and $x.\texttt{close}; P$ denote complementary actions for closing session x. The forwarder process $[x \leftrightarrow y]$ denotes a bi-directional link between sessions x and y. Process $P \mid Q$ denotes the parallel execution of P and Q. Process $(\nu x)P$ denotes the process P in which name x has been restricted, i.e., x is kept private to P. $\mathbf{0}$ is the inactive process.

The constructs in the second line introduce non-deterministic sessions which, intuitively, *may* provide a session protocol *or* fail.

- Process $x.\overline{\texttt{some}}; P$ confirms that the session on x will execute and continues as P. Process $x.\overline{\texttt{none}}$ signals the failure of implementing the session on x.
- Process $x.\texttt{some}_{(w_1,\cdots,w_n)}; P$ specifies a dependency on a non-deterministic session x. This process can either (i) synchronize with an action $x.\overline{\texttt{some}}$ and continue as P, or (ii) synchronize with an action $x.\overline{\texttt{none}}$, discard P, and propagate the failure on x to $(w_1, \cdots, w_n)$, which are sessions implemented in P. When x is the only session implemented in P, the tuple of dependencies is empty and so we write simply $x.\texttt{some}; P$.
- $P \oplus Q$ denotes a *non-deterministic choice* between P and Q. We shall often write $\bigoplus_{i \in I} P_i$ to stand for $P_1 \oplus \cdots \oplus P_n$.

In $(\nu y)P$ and $x(y).P$ the distinguished occurrence of name y is binding, with scope P. The set of free names of P is denoted by $\mathit{fn}(P)$.

The *reduction semantics* of sπ specifies the computations that a process performs on its own (Fig. 6). It relies on *structural congruence*, denoted $\equiv$, which expresses basic identities on the structure of processes and the non-collapsing nature of non-determinism (cf. [22]).

In Fig. 6, the first reduction rule formalizes communication, which concerns bound names only (internal mobility): name y is bound in both $\overline{x}(y).Q$ and $x(y).P$. The reduction rule for the forwarder process leads to a name substitution. The reduction rule for closing a session is self-explanatory, as is the rule in which prefix $x.\overline{\texttt{some}}$ confirms the availability of a non-deterministic session. When the non-deterministic session is not available, prefix $x.\overline{\texttt{none}}$

$$\begin{array}{rcl}
\overline{x}(y).Q \mid x(y).P & \longrightarrow & (\nu y)(Q \mid P) \\
(\nu x)([x \leftrightarrow y] \mid P) & \longrightarrow & P\{y/x\} \quad (x \neq y) \\
x.\overline{\texttt{close}} \mid x.\texttt{close}; P & \longrightarrow & P \\
x.\overline{\texttt{some}}; P \mid x.\texttt{some}_{(w_1,\cdots,w_n)}; Q & \longrightarrow & P \mid Q \\
x.\overline{\texttt{none}} \mid x.\texttt{some}_{(w_1,\cdots,w_n)}; Q & \longrightarrow & w_1.\overline{\texttt{none}} \mid \cdots \mid w_n.\overline{\texttt{none}}
\end{array}$$

$$\begin{array}{ll}
P \equiv P' \wedge P' \longrightarrow Q' \wedge Q' \equiv Q \Rightarrow P \longrightarrow Q & Q \longrightarrow Q' \Rightarrow P \mid Q \longrightarrow P \mid Q' \\
P \longrightarrow Q \Rightarrow (\nu y)P \longrightarrow (\nu y)Q & Q \longrightarrow Q' \Rightarrow P \oplus Q \longrightarrow P \oplus Q'
\end{array}$$

Figure 6 Reduction for $\mathsf{s}\pi$.

triggers this failure to all dependent sessions $w_1, \ldots, w_n$; this may in turn trigger further failures (i.e., on sessions that depend on $w_1, \ldots, w_n$). Reduction is closed under structural congruence. The remaining rules define contextual reduction with respect to restriction, parallel composition, and non-deterministic choice.

Type System

We introduce the session types that govern process behavior:

▶ **Definition 17** (Session Types). *Session types are given by*

$$A, B ::= \bot \mid \mathbf{1} \mid A \otimes B \mid A \parr B \mid \&A \mid \oplus A$$

Types are assigned to names: an *assignment* $x : A$ enforces the use of name x according to the protocol specified by A. The multiplicative units $\bot$ and $\mathbf{1}$ are used to type terminated (closed) endpoints. We use $A \otimes B$ to type a name that first outputs a name of type A before proceeding as specified by B. Similarly, $A \parr B$ types a name that first inputs a name of type A before proceeding as specified by B. Then we have the two modalities introduced in [6]. We use $\&A$ as the type of a (non-deterministic) session that *may produce* a behavior of type A. Dually, $\oplus A$ denotes the type of a session that *may consume* a behavior of type A.

The two endpoints of a session should be *dual* to ensure absence of communication errors. The dual of a type A is denoted $\overline{A}$. Duality corresponds to negation $(\cdot)^\perp$ in linear logic [6]:

▶ **Definition 18** (Duality). *The duality relation on types is given by:*

$$\overline{\mathbf{1}} = \bot \qquad \overline{\bot} = \mathbf{1} \qquad \overline{A \otimes B} = \overline{A} \parr \overline{B} \qquad \overline{A \parr B} = \overline{A} \otimes \overline{B} \qquad \overline{\oplus A} = \&\overline{A} \qquad \overline{\&A} = \oplus\overline{A}$$

Typing judgments are of the form $P \vdash \Delta$, where P is a process and Δ is a linear context of assignments of types to names. The empty context is denoted "$\cdot$". We write $\&\Delta$ to denote that all assignments in Δ have a non-deterministic type, i.e., $\Delta = w_1{:}\&A_1, \ldots, w_n{:}\&A_n$, for some $A_1, \ldots, A_n$. The typing judgment $P \vdash \Delta$ corresponds to the logical sequent $\vdash \Delta$ for classical linear logic, which can be recovered by erasing processes and name assignments.

Typing rules for processes correspond to proof rules in the logic; Fig. 7 gives a selection (see [6] and [22] for a full account). Rule [$\texttt{Tid}$] interprets the identity axiom using the forwarder process. Rules [$\texttt{T}\mathbf{1}$] and [$\texttt{T}\bot$] type the constructs for session termination. Rules [$\texttt{T}\otimes$] and [$\texttt{T}\parr$] type output and input of a name along a session, respectively. The last four rules are used to type constructs for non-determinism and failure. Rules [$\texttt{T}\&^{\texttt{x}}_{\texttt{d}}$] and [$\texttt{T}\&^{\texttt{x}}$] introduce a session of type $\&A$, which may produce a behavior of type A: while the former rule covers the case in which $x : A$ is available, the latter rule formalizes the case in which $x : A$ is not available (i.e., a failure). Given a sequence of names $\widetilde{w} = w_1, \ldots, w_n$, Rule [$\texttt{T}\oplus^{\texttt{x}}_{\widetilde{\texttt{w}}}$] accounts

$$[\mathtt{Tid}]\ \frac{}{[x \leftrightarrow y] \vdash x{:}A, y{:}\overline{A}} \qquad [\mathtt{T1}]\ \frac{}{x.\overline{\mathtt{close}} \vdash x : \mathbf{1}} \qquad [\mathtt{T}\bot]\ \frac{P \vdash \Delta}{x.\mathtt{close}; P \vdash x{:}\bot, \Delta}$$

$$[\mathtt{T}\otimes]\ \frac{P \vdash \Delta, y : A \quad Q \vdash \Delta', x : B}{\overline{x}(y).(P \mid Q) \vdash \Delta, \Delta', x : A \otimes B} \qquad [\mathtt{T}\parr]\ \frac{P \vdash \Delta, y : C, x : D}{x(y).P \vdash \Delta, x : C \parr D} \qquad [\mathtt{T}\&^{\mathtt{x}}]\ \frac{}{x.\overline{\mathtt{none}} \vdash x : \& A}$$

$$[\mathtt{T}\oplus^{\mathtt{x}}_{\widetilde{\mathtt{w}}}]\ \frac{P \vdash \widetilde{w} : \&\Delta, x : A}{x.\mathtt{some}_{\widetilde{w}}; P \vdash \widetilde{w}{:}\&\Delta, x{:} \oplus A} \qquad [\mathtt{T}\&^{\mathtt{x}}_{\mathtt{d}}]\ \frac{P \vdash \Delta, x : A}{x.\overline{\mathtt{some}}; P \vdash \Delta, x : \& A} \qquad [\mathtt{T}\&]\ \frac{P \vdash \&\Delta \qquad Q \vdash \&\Delta}{P \oplus Q \vdash \&\Delta}$$

Figure 7 Selected typing rules for $\mathsf{s}\pi$.

for the possibility of not being able to consume the session $x : A$ by considering sessions different from x as potentially not available. Finally, Rule $[\mathtt{T}\&]$ expresses non-deterministic choice of processes P and Q that implement non-deterministic behaviors only.

The type system enjoys type preservation, a result that follows directly from the cut elimination property in the underlying logic; it ensures that the observable interface of a system is invariant under reduction. The type system also ensures other properties for well-typed processes (e.g. global progress and confluence); see [6] for details.

▶ **Theorem 19** (Type Preservation [6]). *If $P \vdash \Delta$ and $P \longrightarrow Q$ then $Q \vdash \Delta$.*

4 The Encoding

To encode $\lambda^{\lightning}_{\oplus}$ into $\mathsf{s}\pi$, we first define the encoding $(\!|\cdot|\!)^{\circ}$ from well-formed expressions in $\lambda^{\lightning}_{\oplus}$ to well-formed expressions in $\widehat{\lambda}^{\lightning}_{\oplus}$. Then, the encoding $[\![\cdot]\!]^{\lightning}_{u}$ (for a name u) translates well-formed expressions in $\widehat{\lambda}^{\lightning}_{\oplus}$ to well-typed processes in $\mathsf{s}\pi$. We first discuss the encodability criteria.

4.1 Encodability Criteria

We follow most of the criteria in [11], a widely studied abstract framework for establishing the *quality* of encodings. A *language* $\mathcal{L}$ is a pair: a set of terms and a reduction semantics $\longrightarrow$ on terms (with reflexive, transitive closure denoted $\xrightarrow{*}$). A correct encoding translates terms of a source language $\mathcal{L}_1$ into terms of a target language $\mathcal{L}_2$ by respecting certain criteria. The criteria in [11] concern *untyped* languages; because we treat *typed* languages, we follow [17] in requiring that encodings preserve typability.

▶ **Definition 20** (Correct Encoding). *Let $\mathcal{L}_1 = (\mathcal{M}, \longrightarrow_1)$ and $\mathcal{L}_2 = (\mathcal{P}, \longrightarrow_2)$ be two languages. We use $M, M', \ldots$ and $P, P', \ldots$ to range over elements in $\mathcal{M}$ and $\mathcal{P}$. Also, let $\approx_2$ be a behavioral equivalence on terms in $\mathcal{P}$. We say that a translation $[\![\cdot]\!] : \mathcal{M} \to \mathcal{P}$ is a* correct encoding *if it satisfies the following criteria:*

1. *Type preservation: For every well-typed M, it holds that $[\![M]\!]$ is well-typed.*
2. *Operational Completeness: For every M, M' such that $M \xrightarrow{*}_1 M'$, it holds that $[\![M]\!] \xrightarrow{*}_2 \approx_2 [\![M']\!]$.*
3. *Operational Soundness: For every M and P such that $[\![M]\!] \xrightarrow{*}_2 P$, there exists an M' such that $M \longrightarrow^*_1 M'$ and $P \xrightarrow{*}_2 \approx_2 [\![M']\!]$.*
4. *Success Sensitiveness: For every M, it holds that $M\checkmark_1$ if and only if $[\![M]\!]\checkmark_2$, where $\checkmark_1$ and $\checkmark_2$ denote a success predicate in $\mathcal{M}$ and $\mathcal{P}$, respectively.*

Besides these semantic criteria, we also consider *compositionality*, a syntactic criterion that requires that a composite source term is encoded as the combination of the encodings of its sub-terms. Operational completeness formalizes how reduction steps of a source term are mimicked by its corresponding encoding in the target language; $\approx_2$ conveniently abstracts away from target terms useful in the translation but which are not meaningful in comparisons. Operational soundness concerns the opposite direction: it formalizes the correspondence between (i) the reductions of a target term obtained via the translation and (ii) the reductions of the corresponding source term. The role of $\approx_2$ can be explained as in completeness. Success sensitiveness complements completeness and soundness, which concern reductions and therefore do not contain information about observable behaviors. The so-called success predicates $\checkmark_1$ and $\checkmark_2$ serve as a minimal notion of *observables*; the criterion then says that observability of success of a source term implies observability of success in the corresponding target term, and viceversa. Finally, type preservation is self-explanatory.

We choose not to use *full abstraction* as a correctness criterion. As argued in [12], full abstraction is not an informative criterion when it comes to an encoding's quality.

4.2 First Step: From $\lambda^{\lightning}_{\oplus}$ into $\widehat{\lambda}^{\lightning}_{\oplus}$

We define an encoding $(\!|-|\!)^{\circ}$ from $\lambda^{\lightning}_{\oplus}$ into $\widehat{\lambda}^{\lightning}_{\oplus}$ and prove it is correct. The encoding, defined for well-formed terms in $\lambda^{\lightning}_{\oplus}$ (cf. Def. 42 in App. A.1), relies on an intermediate encoding $(\!|\cdot|\!)^{\bullet}$ on closed $\lambda^{\lightning}_{\oplus}$-terms.

We introduce some notation. Given a term M such that $\#(x, M) = k$ and a sequence of pairwise distinct fresh variables $\widetilde{x} = x_1, \dots, x_k$ we write $M\langle \widetilde{x}/x \rangle$ or $M\langle x_1, \cdots, x_n/x \rangle$ to stand for $M\langle x_1/x \rangle \cdots \langle x_k/x \rangle$. That is, $M\langle \widetilde{x}/x \rangle$ denotes a simultaneous linear substitution whereby each distinct occurrence of x in M is replaced by a distinct $x_i \in \widetilde{x}$. Notice that each x_i has the same type as x. We use (simultaneous) linear substitutions to force all bound variables in $\lambda^{\lightning}_{\oplus}$ to become shared variables in $\widehat{\lambda}^{\lightning}_{\oplus}$.

▶ **Definition 21** (From $\lambda^{\lightning}_{\oplus}$ to $\widehat{\lambda}^{\lightning}_{\oplus}$). *Let $M \in \lambda^{\lightning}_{\oplus}$. Suppose $\Gamma \models M : \tau$, with $\mathtt{dom}(\Gamma) = \mathsf{fv}(M) = \{x_1, \cdots, x_k\}$ and $\#(x_i, M) = j_i$. We define $(\!|M|\!)^{\circ}$ as*

$$(\!|M|\!)^{\circ} = (\!|M\langle \widetilde{x_1}/x_1 \rangle \cdots \langle \widetilde{x_k}/x_k \rangle|\!)^{\bullet}[\widetilde{x_1} \leftarrow x_1] \cdots [\widetilde{x_k} \leftarrow x_k]$$

where $\widetilde{x_i} = x_{i_1}, \cdots, x_{i_{j_i}}$ and the encoding $(\!|\cdot|\!)^{\bullet} : \lambda^{\lightning}_{\oplus} \to \widehat{\lambda}^{\lightning}_{\oplus}$ is defined in Fig. 8 on closed $\lambda^{\lightning}_{\oplus}$-terms. The encoding $(\!|\cdot|\!)^{\circ}$ extends homomorphically to expressions.

The encoding $(\!|-|\!)^{\circ}$ "atomizes" occurrences of variables: it converts n occurrences of a variable x in a term into n distinct variables $x_1, \dots, x_n$. The sharing construct coordinates the occurrences of these variables by constraining each to occur exactly once within a term. We proceed in two stages. First, we share all free variables using $(\!|-|\!)^{\circ}$: this ensures that free variables are replaced by bound shared variables. Second, we apply the encoding $(\!|-|\!)^{\bullet}$ on the corresponding closed term. Two cases of Fig. 8 are noteworthy. In $(\!|\lambda x.M|\!)^{\bullet}$, the occurrences of x are replaced with fresh shared variables that only occur once within in M. The definition of $(\!|M\langle\!\langle B/x \rangle\!\rangle|\!)^{\bullet}$ considers two possibilities. If the bag being encoded is non-empty and the explicit substitution would not lead to failure (the number of occurrences of x and the size of the bag coincide) then we encode the explicit substitution as a sum of explicit linear substitutions. Otherwise, the explicit substitution will lead to a failure, and the encoding proceeds inductively. As we will see, doing this will enable a tight operational correspondence result with $\mathsf{s}\pi$.

$$(\!|x|\!)^\bullet = x \quad (\!|\wr M \wr \cdot B|\!)^\bullet = \wr (\!|M|\!)^\bullet \wr \cdot (\!|B|\!)^\bullet \quad (\!|\mathtt{fail}^{\widetilde{x}}|\!)^\bullet = \mathtt{fail}^{\widetilde{x}} \quad (\!|M\ B|\!)^\bullet = (\!|M|\!)^\bullet\ (\!|B|\!)^\bullet$$

$$(\!|\mathtt{1}|\!)^\bullet = \mathtt{1} (\!|\lambda x.M|\!)^\bullet = \lambda x.((\!|M\langle \widetilde{x}/x\rangle|\!)^\bullet[\widetilde{x} \leftarrow x]) \quad \#(x, M) = n, \text{ each } x_i \text{ is fresh}$$

$$(\!|M\langle\!\langle B/x\rangle\!\rangle|\!)^\bullet = \begin{cases} \displaystyle\sum_{B_i \in \mathsf{PER}((\!|B|\!)^\bullet)} (\!|M\langle \widetilde{x}/x\rangle|\!)^\bullet \langle\!| B_i(1)/x_1 |\!\rangle \cdots \langle\!| B_i(k)/x_k |\!\rangle & \#(x, M) = \mathsf{size}(B) = k \geq 1 \\ (\!|M\langle x_1. \cdots, x_k/x\rangle|\!)^\bullet[\widetilde{x} \leftarrow x]\langle\!\langle (\!|B|\!)^\bullet/x\rangle\!\rangle & \text{otherwise, } \#(x, M) = k \geq 0 \end{cases}$$

Figure 8 Auxiliary Encoding: $\lambda^{\lightning}_{\oplus}$ into $\widehat{\lambda}^{\lightning}_{\oplus}$.

▶ **Example 22.** Consider the $\lambda^{\lightning}_{\oplus}$ term $y\langle\!\langle B/x\rangle\!\rangle$, with $\mathsf{fv}(B) = \emptyset$ and $y \neq x$. Its encoding into $\widehat{\lambda}^{\lightning}_{\oplus}$ is $(\!|y\langle\!\langle B/x\rangle\!\rangle|\!)^\circ = (\!|y_0\langle\!\langle B/x\rangle\!\rangle|\!)^\bullet[y_0 \leftarrow y] = y_0[\leftarrow x]\langle\!\langle (\!|B|\!)^\bullet/x\rangle\!\rangle[y_0 \leftarrow y]$. Notice that the encoding induces (empty) sharing on x, even if x does not occur in the term y. ⌟

We consider correctness (Def. 20) for $(\!|\cdot|\!)^\circ$. Our encoding is in "two-levels", because $(\!|\cdot|\!)^\circ$ it is defined in terms of $(\!|\cdot|\!)^\bullet$. As such, it satisfies a weak form of compositionality [11]. In [22] we have established the following:

▶ **Theorem 23** (Correctness for $(\!|\cdot|\!)^\circ$). *The encoding $(\!|\cdot|\!)^\circ$ is type preserving, operationally complete, operationally sound, and success sensitive.*

4.3 Second Step: From $\widehat{\lambda}^{\lightning}_{\oplus}$ to $\mathsf{s}\pi$

We now define our encoding of $\widehat{\lambda}^{\lightning}_{\oplus}$ into $\mathsf{s}\pi$, and establish its correctness.

▶ **Definition 24** (From $\widehat{\lambda}^{\lightning}_{\oplus}$ into $\mathsf{s}\pi$: Expressions). *Let u be a name. The encoding $[\![\cdot]\!]^{\lightning}_u : \widehat{\lambda}^{\lightning}_{\oplus} \to \mathsf{s}\pi$ is defined in Fig. 9.*

As usual in encodings of λ into π, we use a name u to provide the behaviour of the encoded expression. Here u is a non-deterministic session: the encoded expression can be available or not; this is signaled by prefixes $u.\overline{\mathtt{some}}$ and $u.\overline{\mathtt{none}}$, respectively. Notice that every (free) variable x in a $\widehat{\lambda}^{\lightning}_{\oplus}$ expression becomes a name x in its corresponding $\mathsf{s}\pi$ process.

We discuss the most interesting aspects of the translation in Fig. 9. The term $M\,B$ is encoded into a non-deterministic sum: this models the fact that application involves a choice in the order in which the elements of the bag are substituted. The encoding of $M\langle\!| N/x |\!\rangle$ is the parallel composition of the translations of M and N. We need to wait for confirmation of a behaviour along the variable that is being substituted. The encoding of $M[x_1, \cdots, x_n \leftarrow x]$ first confirms the availability of the behavior along x. Then it sends a dummy variable y_i, which is used to collapse the process in the case of a failed reduction. Subsequently, for each shared variable, the encoding receives a name, which will act as an occurrence of the shared variable. At the end, we use $x.\overline{\mathtt{none}}$ to signal that there is no further information to send over. The encoding of $\wr M \wr \cdot B$ synchronises with the encoding of $M[x_1, \cdots, x_n \leftarrow x]$, just discussed. The name y_i is used to trigger a failure in the computation if there is a lack of elements in the encoding of bag. The encoding of $\mathtt{fail}^{x_1,\cdots,x_k}$ simply triggers failure on u and on each of $x_1, \cdots, x_k$. The encoding of $[\![\mathbb{M} + \mathbb{N}]\!]^{\lightning}_u$ homomorphically preserves non-determinism.

$$\llbracket x \rrbracket^{\lightning}_{u} = x.\overline{\mathtt{some}}; [x \leftrightarrow u]$$

$$\llbracket \lambda x.M[\widetilde{x} \leftarrow x] \rrbracket^{\lightning}_{u} = u.\overline{\mathtt{some}}; u(x).\llbracket M[\widetilde{x} \leftarrow x] \rrbracket^{\lightning}_{u}$$

$$\llbracket MB \rrbracket^{\lightning}_{u} = \bigoplus_{B_i \in \mathsf{PER}(B)} (\nu v)(\llbracket M \rrbracket^{\lightning}_{v} \mid v.\mathtt{some}_{u,\mathsf{fv}(B)}; \overline{v}(x).([v \leftrightarrow u] \mid \llbracket B_i \rrbracket^{\lightning}_{x}))$$

$$\llbracket M[\widetilde{x} \leftarrow x]\langle\!\langle B/x \rangle\!\rangle \rrbracket^{\lightning}_{u} = \bigoplus_{B_i \in \mathsf{PER}(B)} (\nu x)(\llbracket M[\widetilde{x} \leftarrow x] \rrbracket^{\lightning}_{u} \mid \llbracket B_i \rrbracket^{\lightning}_{x})$$

$$\llbracket M \langle\!| N/x |\!\rangle \rrbracket^{\lightning}_{u} = (\nu x)(\llbracket M \rrbracket^{\lightning}_{u} \mid x.\mathtt{some}_{\mathsf{fv}(N)}; \llbracket N \rrbracket^{\lightning}_{x})$$

$$\llbracket M[\leftarrow x] \rrbracket^{\lightning}_{u} = x.\overline{\mathtt{some}}.\overline{x}(y_i).(y_i.\mathtt{some}_{u,\mathsf{fv}(M)}; y_i.\mathtt{close}; \llbracket M \rrbracket^{\lightning}_{u} \mid x.\overline{\mathtt{none}})$$

$$\begin{aligned}\llbracket M[x_1, \cdots, x_n \leftarrow x] \rrbracket^{\lightning}_{u} = \\ & x.\overline{\mathtt{some}}.\overline{x}(y_1).(y_1.\mathtt{some}_{\emptyset}; y_1.\mathtt{close}; \mathbf{0} \\ & \mid x.\overline{\mathtt{some}}; x.\mathtt{some}_{u,(\mathsf{fv}(M)\setminus x_1,\cdots,x_n)}; x(x_1).\cdots \\ & .x.\overline{\mathtt{some}}.\overline{x}(y_n).(y_n.\mathtt{some}_{\emptyset}; y_n.\mathtt{close}; \mathbf{0} \mid x.\overline{\mathtt{some}}; x.\mathtt{some}_{u,(\mathsf{fv}(M)\setminus x_n)}; x(x_n) \\ & .x.\overline{\mathtt{some}}; \overline{x}(y_{n+1}).(y_{n+1}.\mathtt{some}_{u,\mathsf{fv}(M)}; y_{n+1}.\mathtt{close}; \llbracket M \rrbracket^{\lightning}_{u} \mid x.\overline{\mathtt{none}})\ ) \cdots)\end{aligned}$$

$$\llbracket \mathtt{fail}^{x_1,\cdots,x_k} \rrbracket^{\lightning}_{u} = u.\overline{\mathtt{none}} \mid x_1.\overline{\mathtt{none}} \mid \cdots \mid x_k.\overline{\mathtt{none}}$$

$$\llbracket 1 \rrbracket^{\lightning}_{x} = x.\mathtt{some}_{\emptyset}; x(y_n).(y_n.\overline{\mathtt{some}}; y_n.\overline{\mathtt{close}} \mid x.\mathtt{some}_{\emptyset}; x.\overline{\mathtt{none}})$$

$$\begin{aligned}\llbracket \wr M \wr \cdot B \rrbracket^{\lightning}_{x} = \ & x.\mathtt{some}_{\mathsf{fv}(\wr M \wr \cdot B)}; x(y_i).x.\mathtt{some}_{y_i,\mathsf{fv}(\wr M \wr \cdot B)}; x.\overline{\mathtt{some}}; \overline{x}(x_i) \\ & .(x_i.\mathtt{some}_{\mathsf{fv}(M)}; \llbracket M \rrbracket^{\lightning}_{x_i} \mid \llbracket B \rrbracket^{\lightning}_{x} \mid y_i.\overline{\mathtt{none}})\end{aligned}$$

$$\llbracket \mathbb{M} + \mathbb{N} \rrbracket^{\lightning}_{u} = \llbracket \mathbb{M} \rrbracket^{\lightning}_{u} \oplus \llbracket \mathbb{N} \rrbracket^{\lightning}_{u}$$

Figure 9 Encoding $\widehat{\lambda}^{\lightning}_{\oplus}$ expressions into $\mathsf{s}\pi$ processes.

▶ **Example 25.** We illustrate $\llbracket \cdot \rrbracket^{\lightning}_{u}$ in Fig. 9 by encoding the $\widehat{\lambda}^{\lightning}_{\oplus}$-terms $N[\leftarrow x]\langle\!\langle \wr M \wr / x \rangle\!\rangle$ and $\mathtt{fail}^{\mathsf{fv}(N)\cup\mathsf{fv}(M)}$, where M, N are closed well-formed $\widehat{\lambda}^{\lightning}_{\oplus}$-terms (i.e. $\mathsf{fv}(N) = \mathsf{fv}(M) = \emptyset$):

$$\begin{aligned}\llbracket N[\leftarrow x]\langle\!\langle \wr M \wr / x \rangle\!\rangle \rrbracket^{\lightning}_{u} &= (\nu x)(\llbracket N[\leftarrow x] \rrbracket^{\lightning}_{u} \mid \llbracket \wr M \wr \rrbracket^{\lightning}_{x}) \\ &= (\nu x)(x.\overline{\mathtt{some}}.\overline{x}(y_i).(y_i.\mathtt{some}_{u}; y_i.\mathtt{close}; \llbracket N \rrbracket^{\lightning}_{u} \mid x.\overline{\mathtt{none}}) \mid \\ &\qquad x.\mathtt{some}_{\emptyset}; x(y_i).x.\mathtt{some}_{y_i}; x.\overline{\mathtt{some}}; \overline{x}(x_i) \\ &\qquad\quad .(x_i.\mathtt{some}_{\emptyset}; \llbracket M \rrbracket^{\lightning}_{x_i} \mid \llbracket 1 \rrbracket^{\lightning}_{x} \mid y_i.\overline{\mathtt{none}})) \\ \llbracket \mathtt{fail}^{\mathsf{fv}(N)\cup\mathsf{fv}(M)} \rrbracket^{\lightning}_{u} &= u.\overline{\mathtt{none}}\end{aligned}$$

◀

We now encode intersection types (for $\lambda^{\lightning}_{\oplus}$ and $\widehat{\lambda}^{\lightning}_{\oplus}$) into session types (for $\mathsf{s}\pi$):

▶ **Definition 26** (From $\widehat{\lambda}^{\lightning}_{\oplus}$ into $\mathsf{s}\pi$: Types)**.** *The translation $\llbracket \cdot \rrbracket^{\lightning}$ on types is defined in Fig. 10. Let Γ be an assignment defined as $\Gamma = \underline{x_1 : \sigma_1, \cdots, x_m : \sigma_k}, \underline{v_1 : \pi_1, \cdots, v_n} : \pi_n$. We define $\llbracket \Gamma \rrbracket^{\lightning}$ as $x_1 : \&\overline{\llbracket \sigma_1 \rrbracket^{\lightning}}, \cdots, x_k : \&\overline{\llbracket \sigma_k \rrbracket^{\lightning}}, v_1 : \&\llbracket \pi_1 \rrbracket^{\lightning}_{(\sigma,i_1)}, \cdots, v_n : \&\llbracket \pi_n \rrbracket^{\lightning}_{(\sigma,i_n)}$.*

The encoding of types captures our use of non-deterministic session protocols (typed with "&") to represent non-deterministic and fail-prone evaluation in $\widehat{\lambda}_{\oplus}$. Notice that the encoding of the multiset type π depends on two arguments (a strict type σ and a number $i \geq 0$) which are left unspecified above. This is crucial to represent mismatches in $\widehat{\lambda}^{\lightning}_{\oplus}$ (i.e., sources of failures) as typable processes in $\mathsf{s}\pi$. For instance, in Fig. 5, Rule [FS:app] admits a mismatch between $\sigma^{j} \to \tau$ and σ^{k}, for it allows $j \neq k$. In our proof of type preservation, these two arguments are instantiated appropriately, enabling typability as session-typed processes.

$$
\begin{aligned}
[\![\mathbf{unit}]\!]^{\lightning} &= \&\mathbf{1} \\
[\![\pi \to \tau]\!]^{\lightning} &= \&((\overline{[\![\pi]\!]^{\lightning}_{(\sigma,i)}}) \mathbin{\bindnasrepma} [\![\tau]\!]^{\lightning}) \quad \text{(for some strict type } \sigma \text{, with } i \geq 0) \\
[\![\sigma \wedge \pi]\!]^{\lightning}_{(\sigma,i)} &= \overline{\&((\oplus\bot) \otimes (\& \oplus ((\&\overline{[\![\sigma]\!]^{\lightning}}) \mathbin{\bindnasrepma} (\overline{[\![\pi]\!]^{\lightning}_{(\sigma,i)}}))))} \\
&= \oplus((\&\mathbf{1}) \mathbin{\bindnasrepma} (\oplus\&((\oplus[\![\sigma]\!]^{\lightning}) \otimes ([\![\pi]\!]^{\lightning}_{(\sigma,i)}))))\\
[\![\omega]\!]^{\lightning}_{(\sigma,i)} &= \begin{cases} \overline{\&((\oplus\bot) \otimes (\& \oplus \bot))} & \text{if } i = 0 \\ \overline{\&((\oplus\bot) \otimes (\& \oplus ((\&\overline{[\![\sigma]\!]^{\lightning}}) \mathbin{\bindnasrepma} (\overline{[\![\omega]\!]^{\lightning}_{(\sigma,i-1)}}))))} & \text{if } i > 0 \end{cases}
\end{aligned}
$$

Figure 10 Encoding types for $\widehat{\lambda}^{\lightning}_{\oplus}$ as session types.

With our encodings of expressions and types in place, we can now encode judgments:

▶ **Definition 27** (Encoding Judgments). *If* $\Gamma \models \mathbb{M} : \tau$ *then* $[\![\mathbb{M}]\!]^{\lightning}_u \vdash [\![\Gamma]\!]^{\lightning}, u : [\![\tau]\!]^{\lightning}$.

We are now ready to consider correctness for $[\![\cdot]\!]^{\lightning}$, as in Def. 20. First, the compositionality property follows directly from Fig. 9. We now state the remaining properties in Def. 20, which we have established in [22]. First, type preservation:

▶ **Theorem 28** (Type Preservation for $[\![\cdot]\!]^{\lightning}_u$). *Let B and $\mathbb{M}$ be a bag and an expression.*

1. *If* $\Gamma \models B : \pi$ *then* $[\![B]\!]^{\lightning}_u \models [\![\Gamma]\!]^{\lightning}, u : [\![\pi]\!]^{\lightning}_{(\sigma,i)}$, *for some strict type σ and some i.*
2. *If* $\Gamma \models \mathbb{M} : \tau$ *then* $[\![\mathbb{M}]\!]^{\lightning}_u \models [\![\Gamma]\!]^{\lightning}, u : [\![\tau]\!]^{\lightning}$.

We now consider operational completeness. Because $\widehat{\lambda}^{\lightning}_{\oplus}$ satisfies the diamond property, it suffices to consider completeness based on a single reduction step ($\mathbb{N} \longrightarrow \mathbb{M}$):

▶ **Theorem 29** (Operational Completeness). *Let $\mathbb{N}$ and $\mathbb{M}$ be well-formed $\widehat{\lambda}^{\lightning}_{\oplus}$ closed expressions. If $\mathbb{N} \longrightarrow \mathbb{M}$ then there exists Q such that $[\![\mathbb{N}]\!]^{\lightning}_u \longrightarrow^* Q = [\![\mathbb{M}]\!]^{\lightning}_u$.*

▶ **Example 30** (Cont. Example 25). Since M and N are well-formed we can verify, by applying rules in Fig. 5 that, $N[\leftarrow x]\langle\!\langle \wr M \wr / x \rangle\!\rangle$ and $\mathtt{fail}^{\mathsf{fv}(N)\cup\mathsf{fv}(M)}$ are well-formed. Notice that $N[\leftarrow x]\langle\!\langle \wr M \wr / x \rangle\!\rangle \longrightarrow_{[\mathtt{RS:Fail}]} \mathtt{fail}^{\mathsf{fv}(N)\cup\mathsf{fv}(M)}$. The encoding of the lhs reduces to encoding of the rhs via the reduction rules of $\mathsf{s}\pi$ (Fig. 6) as $[\![N[\leftarrow x]\langle\!\langle \wr M \wr / x \rangle\!\rangle]\!]^{\lightning}_u \longrightarrow^* [\![\mathtt{fail}^{\mathsf{fv}(N)\cup\mathsf{fv}(M)}]\!]^{\lightning}_u$. The complete example with the reduction steps can be found in [22]. ⌟

In soundness we use the precongruence $\succeq_\lambda$ (Fig. 3). We write $N \longrightarrow_{\succeq_\lambda} N'$ iff $N \succeq_\lambda N_1 \longrightarrow N_2 \succeq_\lambda N'$, for some N_1, N_2. The reflexive, transitive closure of $\longrightarrow_{\succeq_\lambda}$ is $\longrightarrow^*_{\succeq_\lambda}$.

▶ **Theorem 31** (Operational Soundness). *Let $\mathbb{N}$ be a well-formed, closed $\widehat{\lambda}^{\lightning}_{\oplus}$ expression. If $[\![\mathbb{N}]\!]^{\lightning}_u \longrightarrow^* Q$ then $Q \longrightarrow^* Q'$, $\mathbb{N} \longrightarrow^*_{\succeq_\lambda} \mathbb{N}'$ and $[\![\mathbb{N}']\!]^{\lightning}_u = Q'$, for some $Q', \mathbb{N}'$.*

Finally, we consider success sensitiveness. This requires extending $\widehat{\lambda}^{\lightning}_{\oplus}$ and $\mathsf{s}\pi$ with success predicates. In $\mathsf{s}\pi$, we say that P is unguarded if it does not occur behind a prefix.

▶ **Definition 32** (Success in $\widehat{\lambda}^{\lightning}_{\oplus}$). *We extend the syntax of terms for $\widehat{\lambda}^{\lightning}_{\oplus}$ with the $\checkmark$ construct. We define $\mathbb{M} \Downarrow_{\checkmark}$ iff there exist $M_1, \cdots, M_k$ such that $\mathbb{M} \longrightarrow^* M_1 + \cdots + M_k$ and* $\mathsf{head}(M'_j) = \checkmark$, *for some $j \in \{1, \ldots, k\}$ and term M'_j such that $M_j \succeq_\lambda M'_j$.*

▶ **Definition 33** (Success in $\mathsf{s}\pi$). *We extend the syntax of $\mathsf{s}\pi$ processes with the $\checkmark$ construct, which we assume well typed. We define $P \Downarrow_{\checkmark}$ to hold whenever there exists a P' such that $P \longrightarrow^* P'$ and P' contains an unguarded occurrence of $\checkmark$.*

We now extend Def. 24 by decreeing $[\![\checkmark]\!]^{\lightning}_u = \checkmark$. We finally have:

▶ **Theorem 34** (Success Sensitivity). *Let $\mathbb{M}$ be a well-formed, closed $\widehat{\lambda}^{\lightning}_{\oplus}$ expression. Then $\mathbb{M} \Downarrow_{\checkmark}$ iff $[\![\mathbb{M}]\!]^{\lightning}_u \Downarrow_{\checkmark}$.*

5 Discussion

Summary. We developed a correct encoding of $\lambda^{\lightning}_{\oplus}$, a new resource λ-calculus in which expressions feature non-determinism and explicit failure, into $\mathsf{s}\pi$, a session-typed π-calculus in which behavior is non-deterministically available: a protocol may perform as stipulated but also fail. Our encodability result is obtained by appealing to $\widehat{\lambda}^{\lightning}_{\oplus}$, an intermediate language with sharing constructs that simplifies the treatment of variables in expressions. To our knowledge, we are the first to relate typed λ-calculi and typed π-calculi encompassing non-determinism and explicit failures, while connecting intersection types and session types, two different mechanisms for resource-awareness in sequential and concurrent settings, respectively.

Design of $\lambda^{\lightning}_{\oplus}$ (and $\widehat{\lambda}^{\lightning}_{\oplus}$). The design of the sequential calculus $\lambda^{\lightning}_{\oplus}$ has been influenced by the typed mechanisms for non-determinism and failure in the concurrent calculus $\mathsf{s}\pi$. As $\mathsf{s}\pi$ stands on rather solid logical foundations (via the Curry-Howard correspondence between linear logic and session types [7, 27, 6]), $\lambda^{\lightning}_{\oplus}$ defines a logically motivated addition to resource λ-calculi in the literature; see, e.g., [3, 4, 21]. Major similarities between $\lambda^{\lightning}_{\oplus}$ and these existing languages include: as in [4], our semantics performs lazy evaluation and linear substitution on the head variable; as in [21], our reductions lead to non-deterministic sums. A distinctive feature of $\lambda^{\lightning}_{\oplus}$ is its lazy treatment of failures, via the dedicated term $\mathtt{fail}^{\widetilde{x}}$. In contrast, in [3, 4, 21] there is no dedicated term to represent failure. The non-collapsing semantics for non-determinism is another distinctive feature of $\lambda^{\lightning}_{\oplus}$.

Our design for $\widehat{\lambda}^{\lightning}_{\oplus}$ has been informed by the λ-calculi with sharing introduced in [13] and studied in [10]. Also, our translation from $\lambda^{\lightning}_{\oplus}$ into $\widehat{\lambda}^{\lightning}_{\oplus}$ borrows insights from the translations presented in [13]. Notice that the calculi in [13, 10] do not consider explicit failure nor non-determinism. We distinguish between *well-typed* and *well-formed* expressions: this allows us to make fail-prone evaluation in $\lambda^{\lightning}_{\oplus}$ explicit. It is interesting that explicit failures can be elegantly encoded as protocols in $\mathsf{s}\pi$– this way, we make the most out of $\mathsf{s}\pi$'s expressivity.

Related Works. A source of inspiration for our work is the work by Boudol and Laneve [4]. As far as we know, this is the only prior study that connects λ and π from a resource-oriented perspective, via an encoding of a λ-calculus with multiplicities into a π-calculus without sums. The goal of [4] is different from ours, as they study the discriminating power of semantics for λ as induced by encodings into π. In contrast, we study how typability delineates the encodability of resource-awareness across sequential and concurrent realms. Notice that the calculi in [4] are untyped, whereas we consider typed calculi and our encodings preserve typability. As a result, the encoding in [4] is conceptually different from ours; remarkably, our encoding of $\widehat{\lambda}^{\lightning}_{\oplus}$ into $\mathsf{s}\pi$ respects linearity and homomorphically translates sums.

There are some similarities between $\lambda^{\lightning}_{\oplus}$ and the differential λ-calculus, introduced in [9]. Both express non-deterministic choice via sums and use linear head reduction for evaluation. In particular, our fetch rule, which consumes non-deterministically elements from a bag, is related to the derivation (which has similarities with substitution) of a differential term. However, the focus of [9] is not on typability nor encodings to process calculi; instead they relate the Taylor series of analysis to the linear head reduction of λ-calculus.

Prior works have studied encodings of typed λ-calculi into typed π-calculi; see, e.g., [23, 4, 24, 1, 16, 20, 26]. None of these works consider non-determinism and failures; the one exception is the encoding in [6], which involves a λ-calculus with exceptions and failures (but without non-determinism due to bags, as in $\lambda^{\lightning}_{\oplus}$) for which no (reduction) semantics is given. As a result, the encoding in [6] is different from ours, and only preserves typability: important semantic properties such as operational completeness, operational soundness, and success sensitivity are not considered in [6].

Ongoing and Future Work. In $\lambda^{\lightning}_{\oplus}$ bags have *linear* resources, which are used exactly once. In ongoing work, we have established that our approach to encodability in $\mathsf{s}\pi$ extends to the case in which bags contain both linear and *unrestricted* resources, as in [21]. Handling such an extension of $\lambda^{\lightning}_{\oplus}$ requires the full typed process framework in [6], with replicated processes and labeled choices (which were not needed to encode $\lambda^{\lightning}_{\oplus}$).

The approach and results developed here enable us to tackle open questions that go beyond the scope of this work. First, we wish to explore whether our correct encoding can be defined in a setting with *collapsing* non-determinism. Second, we plan to investigate formal results of relative expressiveness that connect $\lambda^{\lightning}_{\oplus}$ and the resource calculi in [4, 21].

References

1 Martin Berger, Kohei Honda, and Nobuko Yoshida. Genericity and the pi-calculus. In Andrew D. Gordon, editor, *Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS 2003 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2620 of *Lecture Notes in Computer Science*, pages 103–119. Springer, 2003. `doi:10.1007/3-540-36576-1_7`.

2 Viviana Bono and Mariangiola Dezani-Ciancaglini. A tale of intersection types. In Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller, editors, *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, pages 7–20. ACM, 2020. `doi:10.1145/3373718.3394733`.

3 Gérard Boudol. The lambda-calculus with multiplicities (abstract). In Eike Best, editor, *CONCUR '93, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 1–6. Springer, 1993. `doi:10.1007/3-540-57208-2_1`.

4 Gérard Boudol and Cosimo Laneve. lambda-calculus, multiplicities, and the pi-calculus. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 659–690, 2000.

5 Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. Non-idempotent intersection types for the lambda-calculus. *Logic Journal of the IGPL*, 25(4):431–464, 2017.

6 Luís Caires and Jorge A. Pérez. Linearity, control effects, and behavioral types. In Hongseok Yang, editor, *Programming Languages and Systems – 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 229–259. Springer, 2017. `doi:10.1007/978-3-662-54434-1_9`.

7 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR 2010 – Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31–September 3, 2010. Proceedings*, pages 222–236, 2010. `doi:10.1007/978-3-642-15375-4_16`.

8 Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Adolfo Piperno. Filter models for a parallel and non deterministic lambda-calculus. In Andrzej M. Borzyszkowski and Stefan Sokolowski, editors, *Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93, Gdansk, Poland, August 30–September 3, 1993, Proceedings*, volume 711 of *Lecture Notes in Computer Science*, pages 403–412. Springer, 1993. `doi:10.1007/3-540-57182-5_32`.

9 Thomas Ehrhard and Laurent Regnier. The differential lambda-calculus. *Theor. Comput. Sci.*, 309(1-3):1–41, 2003. `doi:10.1016/S0304-3975(03)00392-X`.

10 Silvia Ghilezan, Jelena Ivetic, Pierre Lescanne, and Silvia Likavec. Intersection types for the resource control lambda calculi. In *Theoretical Aspects of Computing – ICTAC 2011 – 8th International Colloquium, Johannesburg, South Africa, August 31–September 2, 2011. Proceedings*, pages 116–134, 2011. `doi:10.1007/978-3-642-23283-1_10`.

11 Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.*, 208(9):1031–1053, 2010. `doi:10.1016/j.ic.2010.05.002`.

12 Daniele Gorla and Uwe Nestmann. Full abstraction for expressiveness: history, myths and facts. *Math. Struct. Comput. Sci.*, 26(4):639–654, 2016. doi:10.1017/S0960129514000279.

13 Tom Gundersen, Willem Heijltjes, and Michel Parigot. Atomic lambda calculus: A typed lambda-calculus with explicit sharing. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 311–320, 2013. doi:10.1109/LICS.2013.37.

14 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

15 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems – ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28–April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

16 Kohei Honda, Nobuko Yoshida, and Martin Berger. Process types as a descriptive tool for interaction – control and the pi-calculus. 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 1–20. Springer, 2014. doi:10.1007/978-3-319-08918-8_1.

17 Dimitrios Kouzapas, Jorge A. Pérez, and Nobuko Yoshida. On the relative expressiveness of higher-order session processes. *Inf. Comput.*, 268, 2019. doi:10.1016/j.ic.2019.06.002.

18 Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992. doi:10.1017/S0960129500001407.

19 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992. doi:10.1016/0890-5401(92)90008-4.

20 Dominic A. Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20–22, 2016*, pages 568–581. ACM, 2016. doi:10.1145/2837614.2837634.

21 Michele Pagani and Simona Ronchi Della Rocca. Solvability in resource lambda-calculus. In C.-H. Luke Ong, editor, *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6014 of *Lecture Notes in Computer Science*, pages 358–373. Springer, 2010. doi:10.1007/978-3-642-12032-9_25.

22 Joseph Paulus, Daniele Nantes-Sobrinho, and Jorge A. Pérez. Non-Deterministic Functions as Non-Deterministic Processes (Extended Version). *CoRR*, abs/2104.14759, 2021. arXiv:2104.14759.

23 Davide Sangiorgi. From lambda to pi; or, rediscovering continuations. *Math. Struct. Comput. Sci.*, 9(4):367–401, 1999. URL: http://journals.cambridge.org/action/displayAbstract?aid=44843.

24 Davide Sangiorgi and David Walker. *The Pi-Calculus – a theory of mobile processes.* Cambridge University Press, 2001.

25 Bernardo Toninho, Luís Caires, and Frank Pfenning. Functions as session-typed processes. In Lars Birkedal, editor, *Foundations of Software Science and Computational Structures – 15th International Conference, FOSSACS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24–April 1, 2012. Proceedings*, volume 7213 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 2012. doi:10.1007/978-3-642-28729-9_23.

26 Bernardo Toninho and Nobuko Yoshida. On polymorphic sessions and functions – A tale of two (fully abstract) encodings. In Amal Ahmed, editor, *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 827–855. Springer, 2018. doi:10.1007/978-3-319-89884-1_29.

27 Philip Wadler. Propositions as sessions. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 273–286. ACM, 2012. doi:10.1145/2364527.2364568.

A Appendix

A.1 Omitted Syntactic and Semantic Notations for $\lambda^{\lightning}_{\oplus}$

Auxiliary Notions. In $\lambda^{\lightning}_{\oplus}$, a β-reduction induces an explicit substitution of a bag B for a variable x, denoted $\langle\!\langle B/x \rangle\!\rangle$. This explicit substitution is then expanded into a sum of terms, each of which features a *linear head substitution* $\{\!|N_i/x|\!\}$, where N_i is a term in B; the bag $B \setminus N_i$ is kept in an explicit substitution. In case there is a mismatch between the number of occurrences of the variable to be substituted and the number of resources available, then the reduction leads to the failure term. The reduction rules in Fig. 12 rest upon some auxiliary notions.

▶ **Definition 35** (Set and Multiset of Free Variables). *The set of free variables of a term, bag, and expression, is defined in Fig. 11. We use* $\mathsf{mfv}(M)$ *or* $\mathsf{mfv}(B)$ *to denote a multiset of free variables, defined similarly. We sometimes treat the sequence* $\widetilde{x}$ *as a (multi)set. We write* $\widetilde{x} \uplus \widetilde{y}$ *to denote the multiset union of* $\widetilde{x}$ *and* $\widetilde{y}$ *and* $\widetilde{x} \setminus y$ *to express that every occurrence of* y *is removed from* $\widetilde{x}$. *As usual, a term* M *is* closed *if* $\mathsf{fv}(M) = \emptyset$ *(and similarly for expressions).*

$$\mathsf{fv}(x) = \{x\} \quad \mathsf{fv}(\wr M \wr) = \mathsf{fv}(M) \quad \mathsf{fv}(\lambda x.M) = \mathsf{fv}(M)\setminus\{x\} \quad \mathsf{fv}(M\ B) = \mathsf{fv}(M) \cup \mathsf{fv}(B)$$
$$\mathsf{fv}(1) = \emptyset \qquad \mathsf{fv}(B_1 \cdot B_2) = \mathsf{fv}(B_1) \cup \mathsf{fv}(B_2) \qquad \mathsf{fv}(\mathbb{M} + \mathbb{N}) = \mathsf{fv}(\mathbb{M}) \cup \mathsf{fv}(\mathbb{N})$$
$$\mathsf{fv}(M\langle\!\langle B/x \rangle\!\rangle) = (\mathsf{fv}(M) \setminus \{x\}) \cup \mathsf{fv}(B) \qquad \mathsf{fv}(\mathtt{fail}^{x_1,\cdots,x_n}) = \{x_1, \cdots, x_n\}$$

Figure 11 Free variables for $\lambda^{\lightning}_{\oplus}$.

▶ **Notation 36.** $\#(x, M)$ *denotes the number of (free) occurrences of* x *in* M. *Similarly, we write* $\#(x, \widetilde{y})$ *to denote the number of occurrences of* x *in the multiset* $\widetilde{y}$.

▶ **Definition 37** (Head). *Given a term* M, *we define* $\mathsf{head}(M)$ *inductively as:*

$$\mathsf{head}(x) = x \qquad \mathsf{head}(\lambda x.M) = \lambda x.M \qquad \mathsf{head}(M\ B) = \mathsf{head}(M)$$
$$\mathsf{head}(\mathtt{fail}^{\widetilde{x}}) = \mathtt{fail}^{\widetilde{x}}$$
$$\mathsf{head}(M\langle\!\langle B/x \rangle\!\rangle) = \begin{cases} \mathsf{head}(M) & \textit{if } \#(x, M) = \mathsf{size}(B) \\ \mathtt{fail}^{\emptyset} & \textit{otherwise} \end{cases}$$

▶ **Definition 38** (Linear Head Substitution). *Let* M *be a term such that* $\mathsf{head}(M) = x$. *The* linear head substitution *of a term* N *for* x, *denoted* $\{\!|N/x|\!\}$, *is defined as:*

$$x\{\!|N/x|\!\} = N \qquad (M\ B)\{\!|N/x|\!\} = (M\{\!|N/x|\!\})\ B$$
$$(M\ \langle\!\langle B/y \rangle\!\rangle)\{\!|N/x|\!\} = (M\{\!|N/x|\!\})\ \langle\!\langle B/y \rangle\!\rangle \qquad \textit{where } x \neq y$$

$$[\texttt{R : Beta}]\ \frac{}{(\lambda x.M)B \longrightarrow M\ \langle\!\langle B/x \rangle\!\rangle} \qquad [\texttt{R : Fail}]\ \frac{\#(x,M) \neq \mathsf{size}(B) \quad \widetilde{y} = (\mathsf{mfv}(M) \setminus x) \uplus \mathsf{mfv}(B)}{M\ \langle\!\langle B/x \rangle\!\rangle \longrightarrow \sum_{\mathsf{PER}(B)} \mathtt{fail}^{\widetilde{y}}}$$

$$[\texttt{R : Fetch}]\ \frac{\mathsf{head}(M) = x \qquad B = \wr N_1 \wr \cdot \dots \cdot \wr N_k \wr\ ,\ k \geq 1 \qquad \#(x,M) = k}{M\ \langle\!\langle B/x \rangle\!\rangle \longrightarrow M\{\!|N_1/x|\!\}\langle\!\langle (B \setminus N_1)/x \rangle\!\rangle + \dots + M\{\!|N_k/x|\!\}\langle\!\langle (B \setminus N_k)/x \rangle\!\rangle}$$

$$[\texttt{R : Cons}_1]\ \frac{\widetilde{y} = \mathsf{mfv}(B)}{\mathtt{fail}^{\widetilde{x}}\ B \longrightarrow \sum_{\mathsf{PER}(B)} \mathtt{fail}^{\widetilde{x} \uplus \widetilde{y}}} \qquad [\texttt{R : Cons}_2]\ \frac{\mathsf{size}(B) = k \qquad \#(z,\widetilde{x}) + k \neq 0 \qquad \widetilde{y} = \mathsf{mfv}(B)}{\mathtt{fail}^{\widetilde{x}}\ \langle\!\langle B/z \rangle\!\rangle \longrightarrow \sum_{\mathsf{PER}(B)} \mathtt{fail}^{(\widetilde{x} \setminus z) \uplus \widetilde{y}}}$$

$$[\texttt{R : TCont}]\ \frac{M \longrightarrow M_1' + \dots + M_k'}{C[M] \longrightarrow C[M_1'] + \dots + C[M_k']} \qquad [\texttt{R : ECont}]\ \frac{\mathbb{M} \longrightarrow \mathbb{M}'}{D[\mathbb{M}] \longrightarrow D[\mathbb{M}']}$$

Figure 12 Reduction rules for $\lambda^{\lightning}_{\oplus}$.

Finally, we define contexts for terms and expressions and convenient notations:

▶ **Definition 39** (Term and Expression Contexts). *Contexts for terms (CTerm) and expressions (CExpr) are defined by the following grammar:*

$$\textit{(CTerm)}\ C[\cdot], C'[\cdot] ::= ([\cdot])B \mid ([\cdot])\langle\!\langle B/x \rangle\!\rangle \qquad \textit{(CExpr)}\ D[\cdot], D'[\cdot] ::= M + [\cdot] \mid [\cdot] + M$$

Reduction for $\lambda^{\lightning}_{\oplus}$. The reduction relation $\longrightarrow$ operates lazily on expressions; it is defined by the rules in Fig. 12. Rule [$\texttt{R : Beta}$] is standard and admits a bag (possibly empty) as parameter. Rule [$\texttt{R : Fetch}$] transforms a term into an expression: it opens up an explicit substitution into a sum of terms with linear head substitutions, each denoting the partial evaluation of an element from the bag. Hence, the size of the bag will determine the number of summands in the resulting expression.

Three rules reduce to the failure term: their objective is to accumulate all (free) variables involved in failed reductions. Accordingly, Rule [$\texttt{R : Fail}$] formalizes failure in the evaluation of an explicit substitution $M\ \langle\!\langle B/x \rangle\!\rangle$, which occurs if there is a mismatch between the resources (terms) present in B and the number of occurrences of x to be substituted. The resulting failure preserves all free variables in M and B within its attached multiset $\widetilde{y}$. Rules [$\texttt{R : Cons}_1$] and [$\texttt{R : Cons}_2$] describe reductions that lazily consume the failure term, when a term has $\mathtt{fail}^{\widetilde{x}}$ at its head position. The former rule consumes bags attached to it whilst preserving all its free variables. The latter rule is similar but for the case of explicit substitutions; its second premise ensures that either (i) the bag in the substitution is not empty or (ii) the number of occurrences of x in the current multiset of accumulated variables is not zero. When both (i) and (ii) hold, we apply a precongruence rule (cf. [22]), rather than reduction.

Finally, Rule [$\texttt{R : TCont}$] describes the reduction of sub-terms within an expression; in this rule, summations are expanded outside of term contexts. Rule [$\texttt{R : ECont}$] says that reduction of expressions is closed by expression contexts.

▶ **Example 40.** Let $M = (\lambda x.x\wr x\wr y\int\int)\ B$, with $B = \wr z_1\int \cdot \wr z_2\int \cdot \wr z_1\int$. We have:

$$M \longrightarrow_{[\mathtt{R:Beta}]} x\wr x\wr y\int\int\langle\langle \wr z_1\int \cdot \wr z_2\int \cdot \wr z_1\int / x\rangle\rangle \longrightarrow_{[\mathtt{R:Fail}]} \sum_{\mathsf{PER}(B)} \mathtt{fail}^{y,z_1,z_2,z_1}$$

The number of occurrences of x in the term obtained after β-reduction (2) does not match the size of the bag (3). Therefore, the reduction leads to failure.

Notice that the left-hand sides of the reduction rules in $\lambda^{\lightning}_{\oplus}$ do not interfere with each other. Therefore, reduction in $\lambda^{\lightning}_{\oplus}$ satisfies a *diamond property*:

▶ **Proposition 41** (Diamond Property for $\lambda^{\lightning}_{\oplus}$)**.** *For all* $\mathbb{N}$, $\mathbb{N}_1$, $\mathbb{N}_2$ *in* $\lambda^{\lightning}_{\oplus}$ *s.t.* $\mathbb{N} \longrightarrow \mathbb{N}_1$, $\mathbb{N} \longrightarrow \mathbb{N}_2$ *with* $\mathbb{N}_1 \neq \mathbb{N}_2$ *then* $\exists \mathbb{M}$ *s.t.* $\mathbb{N}_1 \longrightarrow \mathbb{M}$, $\mathbb{N}_2 \longrightarrow \mathbb{M}$.

Proof. We give a short argument to convince the reader of this. Notice that an expression can only perform a choice of reduction steps when it is a nondeterministic sum of terms in which multiple terms can perform independent reductions. For simplicity sake we will only consider an expression $\mathbb{N}$ that consist of two terms where $\mathbb{N} = N + M$. We also have that $N \longrightarrow N'$ and $M \longrightarrow M'$. Then we let $\mathbb{N}_1 = N' + M$ and $\mathbb{N}_2 = N + M'$ by the $[\mathtt{R:ECont}]$ rules. Finally we prove that $\mathbb{M}$ exists by letting $\mathbb{M} = N' + M'$ ◀

Non-Idempotent Intersection Types. The type system for $\lambda^{\lightning}_{\oplus}$ is based on non-idempotent intersection types. The grammar of strict and multiset types, the notions of typing assignments and judgements are the same as in Section 2.

We define *well-formed* $\lambda^{\lightning}_{\oplus}$ expressions, in two stages. We first define a type system for the sub-language $\lambda_{\oplus}$, given in Fig. 13, using the types of Def. 12. Then, we define well-formed expressions for the full language $\lambda^{\lightning}_{\oplus}$, via Def. 42 (see below).

We first discuss selected rules of the type system for $\lambda_{\oplus}$ in Fig. 13. Rule [T:var] is standard. Rule [T:1] assigns the empty bag 1 the empty type ω. Rule [T:weak] introduces a useful weakening principle. Rule [T:app] is standard, requiring a match on the multiset type π. Rule [T:ex-sub] types explicit substitutions where a bag must consist of both the same type and size of the variable it is being substituted for. On top of this type system for $\lambda_{\oplus}$, we define well-formed expressions:

▶ **Definition 42** (Well-formed $\lambda^{\lightning}_{\oplus}$ expressions)**.** *An expression* $\mathbb{M}$ *is well-formed if there exist* Γ *and* τ *such that* $\Gamma \models \mathbb{M} : \tau$ *is entailed via the rules in Fig. 14.*

In Fig. 14, Rules [F:wf-expr] and [F:wf-bag] allow well-typed terms and bags to be well-formed. Rules [F:abs], [F:bag], and [F:sum] are as in the type system for $\lambda_{\oplus}$, but extended to the system of well-formed expressions. Rules [F:ex-sub] and [F:app] differ from similar typing rules as the size of the bags (as declared in their types) is no longer required to match. Finally, Rule [F:fail] has no analogue in the type system: we allow the failure term $\mathtt{fail}^{\widetilde{x}}$ to be well-formed with any type, provided that the context contains the types of the variables in $\widetilde{x}$.

Well-formed expressions satisfy subject reduction (SR); see [22] for a proof.

▶ **Theorem 43** (SR in $\lambda^{\lightning}_{\oplus}$)**.** *If* $\Gamma \models \mathbb{M} : \tau$ *and* $\mathbb{M} \longrightarrow \mathbb{M}'$ *then* $\Gamma \models \mathbb{M}' : \tau$.

Clearly, the set of *well-typed* expressions is strictly included in the set of *well-formed* expressions. Take $M = x\langle\langle \wr N_1\int \cdot \wr N_2\int / x\rangle\rangle$ where both N_1 and N_2 are well-typed. It is easy to see that M is well-formed. However, M is not well-typed.

▶ **Example 44.** The following example illustrates an expression which is not well-formed:

$$\lambda x.x\wr\lambda y.y\int \cdot \wr\lambda z.z_1\wr z_1\wr z_2\int\int\int$$

This is due to the bag being composed of two terms of different types.

$$[\mathtt{T:var}]\ \frac{}{x:\sigma\vdash x:\sigma}\qquad [\mathtt{T:1}]\ \frac{}{\vdash 1:\omega}\qquad [\mathtt{T:weak}]\ \frac{\Gamma\vdash M:\sigma\qquad x\notin \mathtt{dom}(\Gamma)}{\Gamma, x:\omega\vdash M:\sigma}$$

$$[\mathtt{T:abs}]\ \frac{\Gamma,\hat{x}:\sigma^k\vdash M:\tau\quad x\notin \mathtt{dom}(\Gamma)}{\Gamma\vdash \lambda x.M:\sigma^k\to\tau}\qquad [\mathtt{T:app}]\ \frac{\Gamma\vdash M:\pi\to\tau\qquad \Delta\vdash B:\pi}{\Gamma,\Delta\vdash M\ B:\tau}$$

$$[\mathtt{T:bag}]\ \frac{\Gamma\vdash M:\sigma\qquad \Delta\vdash B:\sigma^k}{\Gamma,\Delta\vdash \wr M\int\cdot B:\sigma^{k+1}}\qquad [\mathtt{T:sum}]\ \frac{\Gamma\vdash \mathbb{M}:\sigma\qquad \Gamma\vdash\mathbb{N}:\sigma}{\Gamma\vdash\mathbb{M}+\mathbb{N}:\sigma}$$

$$[\mathtt{T:ex\text{-}sub}]\ \frac{\Gamma,\hat{x}:\sigma^k\vdash M:\tau\qquad \Delta\vdash B:\sigma^k}{\Gamma,\Delta\vdash M\langle\!\langle B/x\rangle\!\rangle:\tau}$$

Figure 13 Typing rules for the sub-language $\lambda_\oplus$ (i.e., $\lambda_\oplus^{\lightning}$ without the failure term).

$$[\mathtt{F:wf\text{-}expr}]\ \frac{\Gamma\vdash\mathbb{M}:\tau}{\Gamma\models\mathbb{M}:\tau}\qquad [\mathtt{F:wf\text{-}bag}]\ \frac{\Gamma\vdash B:\pi}{\Gamma\models B:\pi}\qquad [\mathtt{F:weak}]\ \frac{\Delta\models M:\tau}{\Delta, x:\omega\models M:\tau}$$

$$[\mathtt{F:abs}]\ \frac{\Gamma,\hat{x}:\sigma^n\models M:\tau\quad x\notin\mathtt{dom}(\Gamma)}{\Gamma\models\lambda x.M:\sigma^n\to\tau}\qquad [\mathtt{F:bag}]\ \frac{\Gamma\models M:\sigma\qquad \Delta\models B:\sigma^k}{\Gamma,\Delta\models\wr M\int\cdot B:\sigma^{k+1}}$$

$$[\mathtt{F:sum}]\ \frac{\Gamma\models\mathbb{M}:\sigma\qquad\Gamma\models\mathbb{N}:\sigma}{\Gamma\models\mathbb{M}+\mathbb{N}:\sigma}\qquad [\mathtt{F:fail}]\ \frac{\mathtt{dom}(\Gamma)=\widetilde{x}}{\Gamma\models\mathtt{fail}^{\widetilde{x}}:\tau}$$

$$[\mathtt{F:ex\text{-}sub}]\ \frac{\Gamma,\hat{x}:\sigma^k\models M:\tau\qquad \Delta\models B:\sigma^j\quad k,j\geq 0}{\Gamma,\Delta\models M\langle\!\langle B/x\rangle\!\rangle:\tau}$$

$$[\mathtt{F:app}]\ \frac{\Gamma\models M:\sigma^j\to\tau\qquad \Delta\models B:\sigma^k\qquad k,j\geq 0}{\Gamma,\Delta\models M\ B:\tau}$$

Figure 14 Well-formedness rules for the full language $\lambda_\oplus^{\lightning}$.

Type-Theoretic Constructions of the Final Coalgebra of the Finite Powerset Functor

Niccolò Veltri
Department of Software Science, Tallinn University of Technology, Estonia

Abstract

The finite powerset functor is a construct frequently employed for the specification of nondeterministic transition systems as coalgebras. The final coalgebra of the finite powerset functor, whose elements characterize the dynamical behavior of transition systems, is a well-understood object which enjoys many equivalent presentations in set-theoretic foundations based on classical logic.

In this paper, we discuss various constructions of the final coalgebra of the finite powerset functor in constructive type theory, and we formalize our results in the Cubical Agda proof assistant. Using setoids, the final coalgebra of the finite powerset functor can be defined from the final coalgebra of the list functor. Using types instead of setoids, as it is common in homotopy type theory, one can specify the finite powerset datatype as a higher inductive type and define its final coalgebra as a coinductive type. Another construction is obtained by quotienting the final coalgebra of the list functor, but the proof of finality requires the assumption of the axiom of choice. We conclude the paper with an analysis of a classical construction by James Worrell, and show that its adaptation to our constructive setting requires the presence of classical axioms such as countable choice and the lesser limited principle of omniscience.

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

Keywords and phrases type theory, finite powerset, final coalgebra, Cubical Agda

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.22

Supplementary Material *Software (Source Code)*: `https://github.com/niccoloveltri/final-pfin`

Funding This work was supported by the Estonian Research Council grant PSG659 and by the ESF funded Estonian IT Academy research measure (project 2014-2020.4.05.19-0001).

Acknowledgements We thank Henning Basold, Tarmo Uustalu, Andrea Vezzosi and Niels van der Weide for valuable discussions.

1 Introduction

The powerset functor, delivering the set of subsets of a given set, plays a fundamental role in the behavioral analysis of nondeterministic systems [26], which include process calculi such as Milner's calculus of communicating systems [23] and π-calculus [24]. A nondeterministic system is determined by a function $c : S \to \mathsf{P}\ S$, called a coalgebra, from a set of states S to the set $\mathsf{P}\ S$ of subsets of S. The function c associates to each state $x : S$ a set of new states $c\ x$ reachable from x, so it represents the transition relation of an unlabelled transition system. Adding labels to transitions is easy, just consider coalgebras of the form $c : S \to \mathsf{P}\ (A \times S)$ or $c : S \to (A \to \mathsf{P}\ S)$ instead, where A is a set of labels. In many applications, the set of reachable states is known to be finite, so the powerset functor P can be replaced by the finite powerset functor Pfin delivering only the set of finite subsets.

The behavior of a finitely nondeterministic system starting from a given initial state is fully captured by the final coalgebra of Pfin. Elements of the final coalgebra are execution traces obtained by iteratively running the coalgebra function modelling the system on the

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi; Article No. 22; pp. 22:1–22:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

initial state. The resulting traces are possibly infinite trees with finite unordered branching. Several formal constructions of the final coalgebra of Pfin and other finitary set functors exist in the literature, developed using various different techniques [6, 2, 32, 33, 3]. Adámek et al. collect and compare all these characterizations in their recent book draft [4, Chapter 4]. All these constructions take place in set theory, and reasoning is based on classical logic.

In this work we present various definitions of the final coalgebra of the finite powerset functor in constructive type theory, which have all been formalized in the Cubical Agda proof assistant [30]. Cubical Agda is an implementation of cubical type theory [10], which in turn is a particular presentation of homotopy type theory with support for univalence and higher inductive types (HITs). The choice of Cubical Agda as our foundational setting, over other proof assistants based on Martin-Löf type theory or the calculus of constructions such as plain Agda, Coq or Lean, lays in the fact that both univalence and HITs play an important role for both encoding and reasoning with the finite powerset datatype in homotopy type theory [17]. In our development we also take advantage of Cubical Agda's support for coinductive types [30].

First, we study the construction of the finite powerset as a setoid [7], i.e. a pair of a carrier type and an equivalence relation on the carrier. This is inspired by Danielsson's setoid of finite multisubsets [13]. The final coalgebra of the finite powerset in this setting arises as a setoid with the final coalgebra of the List functor as carrier, whose elements are non-wellfounded trees with finite ordered branching. The equivalence relation on the latter type relates trees that differ only in the order and multiplicity of their subtrees.

Working with setoids, therefore associating a specific equality relation to each type and ensuring that all constructions respect this relation, is not in the spirit of homotopy type theory, where the spotlight is on the notion of propositional equality, also called path equality in this setting. We then consider Frumin et al.'s presentation of the finite powerset datatype as a HIT, Pfin A, formally delivering the free join semilattice on A [17]. It is well-known that coinductive types can be employed for the construction of M-types, i.e. final coalgebras of polynomial functors. We show that coinductive types can be used in a similar way for defining the final Pfin-coalgebra. This construction works since in Cubical Agda HITs are implemented as usual inductive types, in which higher path constructors depend on additional interval names and satisfy two matching conditions on endpoints [11, 9]. In other words, HITs are part of the grammar of strictly positive types and as such they are allowed to appear in the domain type of destructors of coinductive types.

An alternative construction of the final coalgebra of the finite powerset functor (as a type) is obtainable by performing a quotient operation on the final setoid coalgebra, i.e. quotienting the final List-coalgebra by the equivalence relation relating trees containing the same subtrees, possibly in different order and with different multiplicity. This construction is possible in homotopy type theory due to the existence of a set quotient operation definable as a HIT [27]. We show that the resulting quotient type is indeed a fixpoint of Pfin, but the proof of its finality requires the assumption of the full axiom of choice.

The last part of the paper is devoted to the analysis of a classical set-theoretic construction of the final Pfin-coalgebra by James Worrell [33]. It is well known that the chain of iterated applications of Pfin on the singleton set does not stabilize after ω steps [2]. This is in antithesis with the case of polynomial functors, whose final coalgebras (a.k.a. M-types in type theory) always arise as ω-limits, a fact that can also be proved in homotopy type theory [5]. Worrell showed that the final Pfin-coalgebra can be obtained by iterating applications of Pfin for extra ω steps, i.e. as the $(\omega + \omega)$-limit of the chain. Elements of the ω-limit are represented by non-wellfounded trees with unordered but possibly infinite branching, while the $(\omega + \omega)$-limit corresponds to the subset of these trees with finite branching at all levels. We study Worrell's construction in our constructive setting and show that the $(\omega + \omega)$-limit is indeed the final Pfin-coalgebra modulo the assumption of classical principles such as axiom of countable choice and the lesser limited principle of omniscience (LLPO). Notably, Worrell's iterated construction is inherently classical: the injectivity of the canonical Pfin-algebra on the ω-limit is equivalent to LLPO. In particular, it is impossible to prove that the $(\omega + \omega)$-limit is a subset of the ω-limit, as in Worrell's construction, without the assumption of LLPO.

All the material presented in the paper have been formalized in the Cubical Agda proof assistant. The code is freely available at `https://github.com/niccoloveltri/final-pfin`.

2 Type Theory and Cubical Agda

Our work takes place in homotopy type theory (HoTT) [27]. Practically, we formalize our constructions in Cubical Agda [30]. This is an implementation of cubical type theory [10], a particular flavor of HoTT with support for univalence, function extensionality and higher inductive types. What follows is a brief description of basic notions employed in our work. More details on programming in Cubical Agda can be found in Vezzosi et al.'s paper [30].

A few words on notation. We write Type for the universe of small types. We use Agda notation for dependent function types $(x : A) \to B\ x$, where B is a type family of type $A \to \mathsf{Type}$. Implicit arguments of functions are enclosed in curly brackets. We write $=_{\mathrm{df}}$ for definitional equality and we denote judgemental equality by $\equiv$. We reserve the use of the equality symbol $=$ for path equality. Given an element of a dependent sum type $\sum x : A.\ B\ x$, we denote its two projections by fst and snd. The unit type is 1 with unique inhabitant tt, the empty type is $\bot$. The type of Boolean values is Bool with elements true and false, and the binary sum of types A and B is $A + B$. The type of natural numbers is $\mathbb{N}$ with constructors zero and suc, the type of lists with entries in A is $\mathsf{List}\ A$ with constructors $[\,]$ and $(::)$. The unique function from a type A into the unit type is called $! : A \to 1$.

2.1 Univalence, Path Types, Higher Inductive Types

In cubical type theory, and therefore Cubical Agda, *univalence* is a theorem stating that equality of types corresponds to equivalence. A function $f : A \to B$ is an *equivalence* if it has contractible fibers, i.e. if the preimage of any element in B under f is a singleton type. Any function underlying a type isomorphism defines an equivalence. Writing $A \simeq B$ for the type of equivalences between A and B, univalence states that the canonical function of type $A = B \to A \simeq B$ is an equivalence. In particular, there is a function $\mathsf{ua} : A \simeq B \to A = B$ which turns equivalences into equalities. From any proof of equality built as $\mathsf{ua}\ e$ we need to be able to extract the equivalence e, so the representation of equality needs to accommodate such information. Cubical type theory takes inspiration from the cubical interpretation of HoTT [10] and represents equalities as paths, i.e. functions out of an interval object.

In Cubical Agda there is a primitive interval type $\mathbb{I}$ required to be a De Morgan algebra with two endpoints i_0 and i_1. This is used in the implementation of the primitive type $\mathsf{Path}\ A\ \ x\ y$ of path equalities between elements $x : A$ and $y : A$, which we always denote by $x = y$. A path type is similar to a function type with domain $\mathbb{I}$: an element $p : x = y$ is eliminated by application to an interval element $r : \mathbb{I}$, returning $p\ r : A$. Unlike a function type, this application can compute even when p is unknown by using the endpoints x and y stored in the type: $p\ \mathsf{i}_0$ reduces to x, while $p\ \mathsf{i}_1$ reduces to y. The introduction of a path is done via lambda abstraction $\lambda\, i : \mathbb{I}.\, t : x = y$, but this causes the extra requirement to match the endpoints: $t[\mathsf{i}_0/i] \equiv x$ and $t[\mathsf{i}_1/i] \equiv y$.

The identification of equalities with special functions from an interval type allows the provability of the *function extensionality* principle, stating that pointwise equal functions are equal. The proof consists of simply swapping the order of the two input arguments:

$$\mathsf{funExt} : \{f\ g : A \to B\} \to ((x : A) \to f\ x = g\ x) \to f = g$$
$$\mathsf{funExt}\ p\ i\ x =_{\mathrm{df}} p\ x\ i$$

A characteristic feature of homotopy type theory, together with Voevodsky's univalence, is the presence of higher inductive types (HITs) [27]. A HIT is a type whose constructors inductively generate both its elements and its (higher) paths. We introduce three HITs: propositional truncation, set quotient and finite powerset (the latter in Section 4.1).

First, we introduce three classes of types: the *contractible types*, which have a unique inhabitant up to path equality, the *(mere) propositions*, for which any two elements are path equal, and the *sets*, whose path types are propositions. The collections of propositions is called $\mathsf{hProp} =_{\mathrm{df}} \sum A : \mathsf{Type}.\, \mathsf{isProp}\ A$. We follow the informal convention of identifying a proposition with its underlying type (i.e. its first projection).

$$\mathsf{isContr}\ A =_{\mathrm{df}} \textstyle\sum x : A.\ (y : A) \to x = y$$
$$\mathsf{isProp}\ A =_{\mathrm{df}} (x\ y : A) \to x = y$$
$$\mathsf{isSet}\ A =_{\mathrm{df}} (x\ y : A) \to \mathsf{isProp}\ (x = y)$$

The *propositional truncation* of a type A is the HIT generated by the following constructors:

$$\frac{x : A}{|x| : \|A\|} \qquad \frac{x, y : \|A\|}{\mathsf{squash}\ x\ y : x = y}$$

$\|A\|$ is the proposition associated to type A, in which all elements of A have been unified thanks to the path constructor squash. Using propositional truncation, we can define an uninformative *existential quantifier* $\exists x : A.\ B\ x =_{\mathrm{df}} \| \sum x : A.\ B\ x\|$.

The *set quotient* of a type A by a (proof-relevant) relation $R : A \to A \to \mathsf{Type}$ is the HIT generated by the following constructors:

$$\frac{x : A}{[x] : A/R} \qquad \frac{x, y : A \quad r : R\ x\ y}{\mathsf{eq/}\ r : x = y} \qquad \frac{x, y : A/R \quad p, q : x = y}{\mathsf{squash/}\ p\ q : p = q}$$

The element $[x]$ is the R-equivalence class of x, while the path constructor $\mathsf{eq/}$ states that R-related elements have path equal equivalence classes. The 2-path constructor $\mathsf{squash/}$ forces A/R to be a set.

HITs are supported in cubical type theory [11] and have been implemented in Cubical Agda, where they can be introduced using the syntax of inductive types. Path constructors are considered as point constructors depending on extra interval names and satisfying the required matching conditions on endpoints. Functions out of HITs can be defined via pattern matching, where now the user has to deal with the extra cases of higher path constructors. For example, propositional truncation is a functor, and its action on functions is defined as

$$\mathsf{map}_{\|}: (A \to B) \to \|A\| \to \|B\|$$
$$\mathsf{map}_{\|}\ f\ |x| =_{\mathrm{df}} |f\ x|$$
$$\mathsf{map}_{\|}\ f\ (\mathsf{squash}\ x\ y\ i) =_{\mathrm{df}} \mathsf{squash}\ (\mathsf{map}_{\|}\ f\ x)(\mathsf{map}_{\|}\ f\ y)\ i$$

2.2 Coinductive Types

Agda has native support for coinductive types specified by strictly positive functors, and this support has been extended to Cubical Agda as well. As an example, which will be employed in the successive sections, consider the type Tree consisting of finitely-branching

```
record Tree : Type where
  coinductive
  field
    subtrees_L : List Tree
```

```
record TreeB (t u : Tree) : Type where
  coinductive
  field
    subtreesB_L : List‾ TreeB (subtrees_L t) (subtrees_L u)
```

Figure 1 Agda definitions of infinite trees and tree bisimilarity.

non-wellfounded trees defined as the final coalgebra of the List functor. In Agda, the latter is encoded as a coinductive record with one destructor $\mathsf{subtrees_L}$, returning the subtrees of the root, see the left code in Figure 1. The type Tree, together with the destructor $\mathsf{subtrees_L}$, is a coalgebra of the List functor. Elements of coinductive types are characterized by the result of the application of destructors, which means that an element of type Tree is specified by the list of its subtrees. This is dual to the construction of elements of inductive types in terms of constructors. For example, the infinite binary tree is corecursively defined as: $\mathsf{subtrees_L}\ \mathsf{binTree} =_{\mathrm{df}} \mathsf{binTree} :: \mathsf{binTree} :: [\,]$.

An important advantage of working in Cubical Agda is the possibility to prove the *coinduction principle* [30]. For the type of trees, this states that tree bisimilarity is equivalent to path equality. Bisimilarity can be defined as a coinductive relation on trees, and as such it can be encoded in Agda as a coinductive record, see the right code in Figure 1. In the codomain of the destructor $\mathsf{subtreesB_L}$ we employ the *lifting* of a type family $R : A \to B \to \mathsf{Type}$ to lists, inductively generated by two constructors:

$$\frac{}{[\,] : \overline{\mathsf{List}}\ R\ [\,]\ [\,]} \qquad \frac{p : R\ a\ b \quad r : \overline{\mathsf{List}}\ R\ l\ m}{p :: r : \overline{\mathsf{List}}\ R\ (a :: l)\ (b :: m)} \tag{1}$$

The proof of the coinduction principle $\mathsf{bisim_L}$ fundamentally employs copatterns [1] and lambda abstraction of interval variables, i.e. the introduction rule of path types. The coinduction principle $\mathsf{bisim_L}$ is simultaneously constructed with an auxiliary proof $\mathsf{bisim'_L}$, stating that ($\overline{\mathsf{List}}$ TreeB)-related lists of trees are path equal.

$$\begin{array}{l} \mathsf{bisim_L} : \{t\,u : \mathsf{Tree}\} \to \mathsf{TreeB}\,t\,u \to t = u \\ \mathsf{subtrees_L}\ (\mathsf{bisim_L}\ b\,i) =_{\mathrm{df}} \mathsf{bisim'_L}\ (\mathsf{subtreesB_L}\ b)\ i \end{array} \qquad \begin{array}{l} \mathsf{bisim'_L} : \{l\,m : \mathsf{List\ Tree}\} \to \overline{\mathsf{List}}\ \mathsf{TreeB}\,l\,m \to l = m \\ \mathsf{bisim'_L}\ [\,] \qquad i =_{\mathrm{df}} [\,] \\ \mathsf{bisim'_L}\ (b :: r)\ i =_{\mathrm{df}} \mathsf{bisim_L}\ b\,i :: \mathsf{bisim'_L}\ r\,i \end{array}$$

The productivity, i.e. well-definiteness, of the function $\mathsf{bisim_L}$ is guaranteed by the presence of list constructors [ ] and :: at top level in the definition of $\mathsf{bisim'_L}$. More generally, corecursively defined terms are accepted as valid by Agda's productivity checker only when recursive calls appear directly under the application of a constructor. This syntactic restriction, while indeed sufficient for ensuring the productivity of corecursive definitions, makes programming and reasoning with coinductive types in Agda quite cumbersome. For example, the following construction of the unique coalgebra morphism from the carrier of a coalgebra $c : X \to \mathsf{List}\ X$ to Tree is not accepted in Agda ($\mathsf{map_{List}}$ is the action on functions of the List functor):

$$\begin{array}{l} \mathsf{anaTree} : (c : X \to \mathsf{List}\ X) \to X \to \mathsf{Tree} \\ \mathsf{subtrees_L}\ (\mathsf{anaTree}\ c\ x) =_{\mathrm{df}} \mathsf{map_{List}}\ (\mathsf{anaTree}\ c)\ (c\ x) \end{array} \tag{2}$$

For this reason, in our code we parameterize our coinductive types with *sizes*, to ease the productivity checking of corecursive definitions [18, 14]. For example, the function anaTree is accepted by Agda if the type of trees is decorated with size information. Notice that we use sized types for mere practical convenience: we believe possible, with some extra work, to massage the corecursive definitions in our Agda code and obtain equivalent characterizations able to overtake Agda's syntactic productivity checker. In the paper, all mentions to sizes have been removed.

Since HITs are implemented in Agda as a particular kind of inductive types, Cubical Agda also allows the construction of coinductive types specified by functors defined as HITs. This is a somehow experimental feature: the existence of such objects would need to be verified in the cubical set model, which we leave to future work. We will show an example of such a coinductive type in Section 4.2.

3 The Finite Powerset and Its Final Coalgebra as a Setoid

Here we introduce the finite powerset construction as a setoid and study its final coalgebra.

A *setoid* [7], or *Bishop set*, is a pair (A, R) consisting of a type A and a proof-irrelevant (i.e. valued in propositions) equivalence relation R on A. We write Setoid for the type of setoids and, given $S : \mathsf{Setoid}$, we write $\mathsf{carr}\ S$ and $\mathsf{eqr}\ S$ for the carrier and the equivalence relation of S, respectively. A *setoid morphism* between setoids (A, R) and (B, S) is a function $f : A \to B$ which is compatible with the equivalence relations: for all $x, y : A$, if $R\ x\ y$ then $S\ (f\ x)\ (f\ y)$. We write $\mathsf{SetoidMor}\ S\ T$ for the type of setoid morphisms between setoids S and T, and, given $h : \mathsf{SetoidMor}\ S\ T$, we write $\mathsf{fun}\ h : \mathsf{carr}\ S \to \mathsf{carr}\ T$ for its underlying function. Setoids and their morphisms form a category SETOID, but this is not the framework typically employed as a foundational setting for constructive mathematics, since in this category equality of morphisms is given by path equality, not equivalence relation. Bishop-style constructive mathematics is instead developed in SETOIDREL, which is the category SETOID enriched in the category of sets and equivalence relations [19]. In this setting, two setoid morphisms f and g between setoids (A, R) and (B, S) are considered equal whenever, for all $x : A$, $S\ (f\ x)\ (g\ x)$.

In SETOIDREL, given an endofunctor F with action on setoid morphisms map_F (satisfying the functor laws up to the appropriate equivalence relation), the types of F-coalgebras and F-coalgebra morphisms between two F-coalgebras (S, s) and (T, t) are defined as follows:

$$
\begin{aligned}
&\mathsf{Coalg_s}\ F =_{\mathrm{df}} \textstyle\sum S : \mathsf{Setoid}.\ \mathsf{SetoidMor}\ S\ (F\ S)\\
&\mathsf{CoalgMor_s}\ F\ (S, s)\ (T, t) =_{\mathrm{df}}\\
&\qquad \textstyle\sum h : \mathsf{SetoidMor}\ S\ T.\ (x : \mathsf{carr}\ S) \to \mathsf{eqr}\ T\ (\mathsf{fun}\ t\ (\mathsf{fun}\ h\ x))\ (\mathsf{fun}\ (\mathsf{map_F}\ h)\ (\mathsf{fun}\ s\ x))
\end{aligned}
\tag{3}
$$

A coalgebra in SETOIDREL is final if there exists a unique coalgebra morphism from any other coalgebra, up to equivalence relation:

$$
\mathsf{Final_s}\ F =_{\mathrm{df}} \textstyle\sum C : \mathsf{Coalg_s}\ F.\ (D : \mathsf{Coalg_s}\ F) \to \mathsf{isContr_s}\ (\mathsf{CoalgMor_s}\ F\ C\ D) \tag{4}
$$

where elements of $\mathsf{isContr_s}\ (\mathsf{CoalgMor_s}\ F\ C\ D)$ are pairs consisting of a coalgebra morphism h and, for any other coalgebra morphism h', a proof that h and h' are equivalent as setoid morphisms.

```
record TreeR (t u : Tree) : Type where
  coinductive
  field
    subtreesR : List^ TreeR (subtrees_L t) (subtrees_L u)
```

Figure 2 Agda definition of the coinductive closure of the relator $\widehat{\mathsf{List}}$.

3.1 The Setoid of Finite Subsets

Given a setoid (A, R), its setoid of finite subsets is defined as $\mathsf{Pfin_s}\ (A, R) =_{\mathrm{df}} (\mathsf{List}\ A, \widehat{\mathsf{List}}\ R)$, where $\widehat{\mathsf{List}}$ is a lifting of List to relations, alternative to the lifting given in (1). $\widehat{\mathsf{List}}$ is sometimes called a *relator* and plays an important role in the study of applicative bisimilarity for functional programming languages with nondeterministic choice [21]. Given a type family $R : A \to B \to \mathsf{Type}$, the type family $\widehat{\mathsf{List}}\ R : \mathsf{List}\ A \to \mathsf{List}\ B \to \mathsf{Type}$ is defined as

$$\begin{array}{ll}\widehat{\mathsf{List}}\ R\ l\ m =_{\mathrm{df}} & ((x : A) \to x \in_{\mathsf{L}} l \to \exists y : B.\ y \in_{\mathsf{L}} m \times R\ x\ y) \\ & \times \\ & ((y : B) \to y \in_{\mathsf{L}} m \to \exists x : A.\ x \in_{\mathsf{L}} l \times R\ y\ x)\end{array} \tag{5}$$

So two lists are related by $\widehat{\mathsf{List}}\ R$ when each element of a list is R-related to an element of the other list. The type family $\in_{\mathsf{L}}$ is the inductive (proof-relevant) membership relation on lists, the subscript L distinguishes this to the membership relation on the type Pfin introduced in Section 4.1. $\mathsf{Pfin_s}$ is an endofunctor on SetoidRel. Its action on setoid morphisms $\mathsf{map}_{\mathsf{Pfin_s}} : \mathsf{SetoidMor}\ S\ T \to \mathsf{SetoidMor}\ (\mathsf{Pfin_s}\ S)\ (\mathsf{Pfin_s}\ T)$ has underlying function $\mathsf{map}_{\mathsf{List}}$.

Notice the presence of existential quantifications $\exists$ in the definition of $\widehat{\mathsf{List}}$. If we were to replace them with $\sum$, we would obtain a setoid of finite multisubsets instead, as the one considered by Danielsson [13].

3.2 The Final Coalgebra

The final coalgebra of the final powerset functor in SetoidRel can be constructed using coinductive types. Consider the coinductive relation of Figure 2 obtained by replacing the lifting $\overline{\mathsf{List}}$ with the lifting $\widehat{\mathsf{List}}$ in the destructor of the tree bisimilarity relation TreeB in Figure 1. Two trees are related by TreeR if, for each subtree of one tree, there merely exists a TreeR-related subtree of the other tree. The setoid $\nu\mathsf{Pfin_s} =_{\mathrm{df}} (\mathsf{Tree}, \mathsf{TreeR})$ is a $\mathsf{Pfin_s}$-coalgebra:

$$\begin{array}{l}\mathsf{coalg_s} : \mathsf{SetoidMor}\ \nu\mathsf{Pfin_s}\ (\mathsf{Pfin_s}\ \nu\mathsf{Pfin_s}) \\ \mathsf{coalg_s} = (\mathsf{subtrees_L}, \mathsf{subtreesR})\end{array}$$

▶ **Theorem 1.** *The* $\mathsf{Pfin_s}$*-coalgebra* $(\nu\mathsf{Pfin_s}, \mathsf{coalg_s})$ *is final in* SetoidRel*.*

Proof. We only show the existence of a coalgebra morphism into $(\nu\mathsf{Pfin_s}, \mathsf{coalg_s})$. Given another $\mathsf{Pfin_s}$-coalgebra (S, s), there is a setoid morphism h from S to $(\mathsf{Tree}, \mathsf{TreeR})$ with underlying function $\mathsf{anaTree}\ (\mathsf{fun}\ s)$.

This function is compatible with equivalence relations. Assume given $x, y : \mathsf{carr}\ S$ such that $\mathsf{eqr}\ S\ x\ y$. We prove $\mathsf{TreeR}\ (\mathsf{anaTree}\ (\mathsf{fun}\ s)\ x)\ (\mathsf{anaTree}\ (\mathsf{fun}\ s)\ y)$. This is a coinductive type, so we proceed by applying the destructor of TreeR and we are left to show that $\mathsf{subtrees_L}\ (\mathsf{anaTree}\ (\mathsf{fun}\ s)\ x)$ is $(\widehat{\mathsf{List}}\ \mathsf{TreeR})$-related to $\mathsf{subtrees_L}\ (\mathsf{anaTree}\ (\mathsf{fun}\ s)\ y)$. The definition of the lifting $\widehat{\mathsf{List}}$ in (5) is symmetric, so it is sufficient to prove the following lemma (in which we unfold the definition of $\mathsf{anaTree}$ as in (2)):

$$
\begin{aligned}
&(x' : \mathsf{Tree}) \to x' \in_{\mathsf{L}} \mathsf{map}_{\mathsf{List}}\ (\mathsf{anaTree}\ (\mathsf{fun}\ s))\ (\mathsf{fun}\ s\ x)\\
&\qquad\qquad \to \exists y' : \mathsf{Tree}.\ y' \in_{\mathsf{L}} \mathsf{map}_{\mathsf{List}}\ (\mathsf{anaTree}\ (\mathsf{fun}\ s))\ (\mathsf{fun}\ s\ y) \times \mathsf{TreeR}\ x'\ y'
\end{aligned}
\tag{6}
$$

Given a tree x' as in the hypotheses of (6), it is possible to construct another tree x'' such that $x'' \in_{\mathsf{L}} \mathsf{fun}\ s\ x$ and $\mathsf{anaTree}\ (\mathsf{fun}\ s)\ x'' = x'$. Remember that s is a setoid morphism and $\mathsf{eqr}\ S\ x\ y$ holds by assumption. This implies that, for each element in the list $\mathsf{fun}\ s\ x$, there exists a $(\mathsf{eqr}\ S)$-related element in the list $\mathsf{fun}\ s\ y$. Since $x'' \in_{\mathsf{L}} \mathsf{fun}\ s\ x$, we obtain a tree y'' such that $y'' \in \mathsf{fun}\ s\ y$ and $\mathsf{eqr}\ S\ x''\ y''$. The required tree y' in the goal of (6) is defined as $\mathsf{anaTree}\ (\mathsf{fun}\ s)\ y''$. The proof of y' being TreeR-related to x' is given by the corecursive hypothesis applied to arguments x'',y'' and the proof of $\mathsf{eqr}\ S\ x''\ y''$. ◀

Differently from the case of polynomial functors, the final $\mathsf{Pfin_s}$-coalgebra cannot be constructed as an ω-limit in SetoidRel. In fact, the ω-limit of the sequence obtained by iterated application of $\mathsf{Pfin_s}$ on the unit setoid is not a fixpoint of $\mathsf{Pfin_s}$. This is in analogy with the situation in classical set theory described by Adámek and Koubek [2], for which we give a constructive account in Section 5.

It is possible to prove a version of Theorem 1 for Setoid instead of SetoidRel: $(\nu\mathsf{Pfin_s}, \mathsf{coalg_s})$ is also the final $\mathsf{Pfin_s}$-coalgebra in Setoid, where one first needs to appropriately adapt the definitions of coalgebra morphism and final coalgebra in (3) and (4) to Setoid.

4 The Finite Powerset and Its Final Coalgebra as a Type

We now abandon the setoid setting and work with types as primary object instead of setoids, as typically done in HoTT. Given an endofunctor $F : \mathsf{Type} \to \mathsf{Type}$ with action on functions $\mathsf{map_F}$, the types of F-coalgebras and F-coalgebra morphisms between two F-coalgebras (A, a) and (B, b) are defined as follows:

$$
\begin{aligned}
&\mathsf{Coalg}\ F =_{\mathrm{df}} \textstyle\sum A : \mathsf{Type}.\ A \to F\ A\\
&\mathsf{CoalgMor}\ F\ (A, a)\ (B, b) =_{\mathrm{df}} \textstyle\sum f : A \to B.\ (x : A) \to b\ (f\ x) = \mathsf{map_F}\ f\ (a\ x)
\end{aligned}
\tag{7}
$$

In this setting, a coalgebra is final if there exists a unique (up to path equality) coalgebra morphism to any other coalgebra.

$$
\mathsf{Final}\ F =_{\mathrm{df}} \textstyle\sum C : \mathsf{Coalg}\ F.\ (D : \mathsf{Coalg}\ F) \to \mathsf{isContr}\ (\mathsf{CoalgMor}\ F\ C\ D) \tag{8}
$$

The definitions in (7) and (8) are the same of Ahrens et al. [5], which they only consider in the case of F being a polynomial functor specified by a signature. The coinductive type Tree of Section 2.2 is the final List-coalgebra, with the function $\mathsf{anaTree}$ of (2) as unique mediating coalgebra morphism.

4.1 The Type of Finite Subsets

The action of the finite powerset functor on a type A returns the set of all finite subsets of A. Following Frumin et al. [17], the finite powerset functor can be encoded as a higher inductive type in two equivalent ways: as a set quotient of lists or as the term algebra of the theory of join semilattices.

As a Set Quotient. The set of finite subsets can be defined as a set quotient of the type of lists: $\mathsf{Pfin_q}\ A =_{\mathrm{df}} \mathsf{List}\ A/\mathsf{SameEls}$. The subscript q indicates that this type is a set quotient. The relation $\mathsf{SameEls}$, as the name suggests, relates lists containing the same elements, and it is given by the relator $\widehat{\mathsf{List}}$ applied to path equality on A, i.e. $\mathsf{SameEls} =_{\mathrm{df}} \widehat{\mathsf{List}}\ (=)$.

As the Free Join Semilattice. The set of finite subsets can also be defined as the *free join semilattice* on a given type A. A join semilattice is a partially ordered set $(X, \leq)$ with a bottom element and a binary join operation. Join semilattices admit an equational presentation as algebraic theories, from which the following higher inductive type can be extrapolated:

$$\frac{}{\varnothing : \mathsf{Pfin}\ A} \qquad \frac{a : A}{\eta\ a : \mathsf{Pfin}\ A} \qquad \frac{x, y : \mathsf{Pfin}\ A}{x \cup y : \mathsf{Pfin}\ A}$$

$$\frac{x : \mathsf{Pfin}\ A}{\mathsf{nr}\ x : x \cup \varnothing = x} \qquad \frac{x, y, z : \mathsf{Pfin}\ A}{\mathsf{assoc}\ x\ y\ z : (x \cup y) \cup z = x \cup (y \cup z)} \qquad \frac{x, y : \mathsf{Pfin}\ A}{\mathsf{comm}\ x\ y : x \cup y = y \cup x}$$

$$\frac{x : \mathsf{Pfin}\ A}{\mathsf{idem}\ x : x \cup x = x} \qquad \frac{x, y : \mathsf{Pfin}\ A \quad p, q : x = y}{\mathsf{squashPfin}\ p\ q : p = q}$$

The type $\mathsf{Pfin}\ A$ is a join semilattice, with empty subset $\varnothing$ as bottom element and binary union $\cup$ as join operation. The partial order can be recovered in the usual way: $x \leq y =_{\mathrm{df}} (x \cup y) = y$. The 1-path constructors mimic the equational theory of join semilattices, while the 2-path constructor $\mathsf{squashPfin}$ forces $\mathsf{Pfin}\ A$ to be a set. The constructor η embeds A into $\mathsf{Pfin}\ A$ and represents the singleton subset operation. The elimination principle of $\mathsf{Pfin}\ A$ corresponds to the universal property of $\mathsf{Pfin}\ A$ as the free join semilattice on A.

The membership relation $\in$ is defined by induction on the finite subset in input.

$$\begin{array}{lll}
\multicolumn{3}{l}{\in\ : A \to \mathsf{Pfin}\ A \to \mathsf{hProp}} \\
a \in \varnothing & =_{\mathrm{df}} & \bot \\
a \in \eta\ b & =_{\mathrm{df}} & \|a = b\| \\
a \in x \cup y & =_{\mathrm{df}} & \|a \in x + a \in y\|
\end{array}$$

The omitted cases for the higher constructors are dealt with using univalence. Moreover the right-hand-sides only contain the types underlying the propositions, the proof terms showing that these satisfy the predicate isProp have been omitted. The subset relation is given by $x \subseteq y =_{\mathrm{df}} (a : A) \to a \in x \to a \in y$, which is equivalent to the order relation $\leq$ defined above.

Given a type family $R : A \to B \to \mathsf{Type}$, its *lifting* to Pfin is the type family $\overline{\mathsf{Pfin}}\ R : \mathsf{Pfin}\ A \to \mathsf{Pfin}\ B \to \mathsf{Type}$ defined as

$$\begin{array}{ll}
\overline{\mathsf{Pfin}}\ R\ s\ t =_{\mathrm{df}} & ((x : A) \to x \in s \to \exists y : B.\ y \in t \times R\ x\ y) \\
& \times \\
& ((y : B) \to y \in t \to \exists x : A.\ x \in s \times R\ y\ x)
\end{array}$$

For all relations R, it is possible to show that $\overline{\mathsf{Pfin}}\ R$ is a congruence, which means that we are able to construct elements of the following types:

$$\overline{\mathsf{Pfin}}\ R\ \varnothing\ \varnothing \qquad \overline{\mathsf{Pfin}}\ R\ s_1\ t_1 \to \overline{\mathsf{Pfin}}\ R\ s_2\ t_2 \to \overline{\mathsf{Pfin}}\ R\ (s_1 \cup s_2)\ (t_1 \cup t_2) \tag{9}$$

When R is path equality, $\overline{\mathsf{Pfin}}\ (=)$ corresponds to extensional equality of finite subsets, i.e. $\overline{\mathsf{Pfin}}\ (=)\ s\ t \simeq (s \subseteq t \times t \subseteq s)$. Since the subset relation is antisymmetric, we have that $\overline{\mathsf{Pfin}}\ (=)\ s\ t \simeq (s = t)$. We call $\mathsf{toPfinEq} : \overline{\mathsf{Pfin}}\ (=)\ s\ t \to (s = t)$ the left-to-right function underlying this equivalence.

The two types $\mathsf{Pfin}\ A$ and $\mathsf{Pfin_q}\ A$ are provably equivalent [17].

```
record νPfin : Type where
  coinductive
  field
    subtrees_P : Pfin νPfin
```

```
record νPfinB (t u : νPfin) : Type where
  coinductive
  field
    subtreesB_P : ‾Pfin‾ νPfinB (subtrees_P t) (subtrees_P u)
```

Figure 3 Agda definitions of coinductive final coalgebra of Pfin and its bisimilarity relation.

4.2 The Final Coalgebra

As a Coinductive Type. As mentioned in the last paragraph of Section 2.2, Cubical Agda allows the construction of coinductive types specified by functors using HITs in their definition. When such functor is the finite powerset functor Pfin, this construction is given by the record type on the left in Figure 3. Notice that νPfin is a set: Pfin A is a set for all types A, so in particular Pfin νPfin is a set, and the latter is isomorphic to νPfin.

In Figure 3, the coinductive relation on the right is the notion of bisimilarity associated to the infinite trees in νPfin. The coinduction principle for νPfin is derivable by copattern matching and path introduction as follows:

$$
\begin{aligned}
&\mathsf{bisim_P} : \{t\ u : \nu\mathsf{Pfin}\} \to \nu\mathsf{PfinB}\ t\ u \to t = u\\
&\mathsf{subtrees_P}\ (\mathsf{bisim_P}\ b\ i) =_{\mathrm{df}}\\
&\quad \mathsf{toPfinEq}\ (\lambda x\, m.\ \mathsf{map}_{\|}(\lambda(x',m',b').\ (x',m',\mathsf{bisim_P}\ b'))\ (\mathsf{fst}\ (\mathsf{subtreesB_P}\ b)\ x\ m),\\
&\qquad\qquad\quad \lambda x\, m.\ \mathsf{map}_{\|}(\lambda(x',m',b').\ (x',m',\mathsf{bisim_P}\ b'))\ (\mathsf{snd}\ (\mathsf{subtreesB_P}\ b)\ x\ m))\\
&\qquad\qquad\ \ i
\end{aligned}
$$

▶ **Theorem 2.** *νPfin is the final coalgebra of Pfin in the sense of (8).*

Proof. The construction of the mediating coalgebra morphism between a Pfin-coalgebra (A, a) and νPfin, whose coalgebra is the destructor $\mathsf{subtrees_P}$, is analogous to the one in (2):

$$
\begin{aligned}
&\mathsf{anaPfin} : (c : X \to \mathsf{Pfin}\ X) \to X \to \nu\mathsf{Pfin}\\
&\mathsf{subtrees_P}\ (\mathsf{anaPfin}\ c\ x) =_{\mathrm{df}} \mathsf{map_{Pfin}}\ (\mathsf{anaPfin}\ c)\ (c\ x)
\end{aligned}
\tag{10}
$$

Now assume given another coalgebra morphism $f : X \to \nu$Pfin. We prove simultaneously the two following lemmata, and conclude uniqueness using the coinduction principle of νPfin.

$$
\begin{aligned}
&\mathsf{anaPfinUniq} : (x : X) \to \nu\mathsf{PfinB}\ (f\ x)\ (\mathsf{anaPfin}\ c\ x)\\
&\mathsf{anaPfinUniqR} : (s : \mathsf{Pfin}\ X) \to \overline{\mathsf{Pfin}}\ \nu\mathsf{PfinB}\ (\mathsf{map_{Pfin}}\ f\ s)\ (\mathsf{map_{Pfin}}\ (\mathsf{anaPfin}\ c)\ s)
\end{aligned}
$$

The first lemma is proved by corecursion, so after an application of the destructor of νPfinB (and after unfolding the definition of anaPfin in (10)), we are left to construct an element of type $\overline{\mathsf{Pfin}}$ νPfinB ($\mathsf{subtrees_P}$ $(f\ x)$) ($\mathsf{map_{Pfin}}$ (anaPfin c) $(c\ x)$). Since f is a coalgebra morphism, we can substitute $\mathsf{subtrees_P}$ $(f\ x)$ for $\mathsf{map_{Pfin}}$ f $(c\ x)$ in the latter, and we return anaPfinUniqR $(c\ x)$ as the inhabitant of the type resulting from the substitution.

The second lemma is proved by induction on s. Since the return type is a proposition, we only need to deal with the three cases of the point constructors.

- Case $s \equiv \varnothing$. We are done by the left result in (9).
- Case $s \equiv \eta\ z$. Our goal reduces to $\overline{\mathsf{Pfin}}$ νPfinB $(\eta\ (f\ z))$ $(\eta$ (anaPfin $c\ z$)). We construct the first argument of this product type, the second argument is defined analogously. Assume given $x : \nu$Pfin and $p : x \in \eta\ (f\ z)$, i.e. a truncated equality proof $p : \|x = f\ z\|$. We need to show that there merely exists $y : \nu$Pfin such that $y \in \eta$ (anaPfin $c\ z$), i.e. $\|y =$ anaPfin $c\ z\|$, and νPfinB $x\ y$ holds. Take $y =_{\mathrm{df}}$ anaPfin $c\ z$, and derive νPfinB $x\ y$ by first rewriting x to $f\ z$ using p (we can remove the propositional truncation in p since the return type is a proposition as well) and subsequently applying anaPfinUniq to z.

- Case $s \equiv s_1 \cup s_2$. We apply the right result in (9) and we conclude by invoking inductive hypotheses $\mathsf{anaPfinUniqR}\ s_1$ and $\mathsf{anaPfinUniqR}\ s_2$. ◀

As a Set Quotient. Alternatively, we could quotient the type Tree of finitely ordered branching trees by the equivalence relation TreeR introduced in Figure 2. The resulting type is a fixpoint of $\mathsf{Pfin_q}$.

▶ **Theorem 3.** *The type* $\mathsf{Tree}/\mathsf{TreeR}$ *is equivalent to* $\mathsf{Pfin_q}\ (\mathsf{Tree}/\mathsf{TreeR})$

Proof. We only discuss the construction of the functions underlying the equivalence. A function $f : \mathsf{Tree}/\mathsf{TreeR} \to \mathsf{Pfin_q}\ (\mathsf{Tree}/\mathsf{TreeR})$ is defined by pattern matching (we only show the case of the point constructor): $f\ [t] =_{\mathrm{df}} [\mathsf{map_{List}}\ (\lambda x.\, [x])\ (\mathsf{subtrees_L}\ t)]$.

A function $g : \mathsf{Pfin_q}\ (\mathsf{Tree}/\mathsf{TreeR}) \to \mathsf{Tree}/\mathsf{TreeR}$ is also defined by pattern matching. Notice that this is equivalent to construct a function $g' : \mathsf{List}\ (\mathsf{Tree}/\mathsf{TreeR}) \to \mathsf{Tree}/\mathsf{TreeR}$ which is compatible with the relation $\mathsf{SameEls}$. Since the type $\mathsf{List}\ (\mathsf{Tree}/\mathsf{TreeR})$ is equivalent to $\mathsf{List}\ \mathsf{Tree}/\overline{\mathsf{List}}\ \mathsf{TreeR}$, it is sufficient to define a function $g'' : \mathsf{List}\ \mathsf{Tree}/\overline{\mathsf{List}}\ \mathsf{TreeR} \to \mathsf{Tree}/\mathsf{TreeR}$ satisfying an adjusted compatibility condition. This is given by pattern matching (again, we omit the cases of the path constructors): $g''\ [l] =_{\mathrm{df}} [\mathsf{subtrees_L^{-1}}\ l]$, where $\mathsf{subtrees_L^{-1}}$ is the inverse of the destructor $\mathsf{subtrees_L}$. ◀

Proving finality of the coalgebra underlying the equivalence of Theorem 3 seems to require the assumption of the full *axiom of choice*. This is constructively problematic, since in HoTT the axiom of choice implies the law of excluded middle [27]. We employ an alternative formulation of the axiom of choice, provably equivalent to the usual one. First, consider two types A, B and a type family $R : B \to B \to \mathsf{Type}$. Let $\overline{\mathsf{Fun}}\ R$ be the lifting of the type family R to the function space $A \to B$, i.e. given $f, g : A \to B$, define $\overline{\mathsf{Fun}}\ R\ f\ g =_{\mathrm{df}} (x : A) \to R\ (f\ x)\ (g\ x)$. It is possible to define a function $\theta_R : (A \to B)/\overline{\mathsf{Fun}}\ R \to A \to B/R$ by pattern matching on the first argument. The existence of a section for θ_R, for all type families R, is an equivalent phrasing of the axiom of choice (see e.g. [28] for a proof of this equivalence):

$$\begin{aligned} &\mathsf{AC} =_{\mathrm{df}} \{A, B : \mathsf{Type}\}\ (R : B \to B \to \mathsf{Type}) \\ &\quad \to \exists \psi_R : (A \to B/R) \to (A \to B)/\overline{\mathsf{Fun}}\ R.\ (x : (A \to B)/\overline{\mathsf{Fun}}\ R) \to \theta_R\ (\psi_R\ x) = x \end{aligned} \tag{11}$$

▶ **Theorem 4.** *Assuming axiom of choice, the type* $\mathsf{Tree}/\mathsf{TreeR}$ *is the final coalgebra of* $\mathsf{Pfin_q}$.

Proof. We only discuss the construction of the mediating coalgebra morphism. We are asked to construct a function $\mathsf{anaPfin_q} : (c : X \to \mathsf{Pfin_q}\ X) \to X \to \mathsf{Tree}/\mathsf{TreeR}$. This can be obtained from the function $\mathsf{anaPfin_q'} : (c : X \to \mathsf{Pfin_q}\ X) \to \mathsf{Pfin_q}\ X \to \mathsf{Tree}/\mathsf{TreeR}$ by precomposition with the coalgebra c. In turn, this can be obtained from the function $\mathsf{anaPfin_q''} : (X \to \mathsf{List}\ X)/\overline{\mathsf{Fun}}\ \mathsf{SameEls} \to \mathsf{List}\ X/\mathsf{SameEls} \to \mathsf{Tree}/\mathsf{TreeR}$ by precomposition with the section ψ_{SameEls}. The latter is definable by pattern matching on both arguments: $\mathsf{anaPfin_q''}\ [c]\ [l] =_{\mathrm{df}} [\mathsf{anaTree}\ c\ l]$. The missing cases in the definition have been omitted. ◀

Without the assumption of the axiom of choice, one gets stuck in the construction of $\mathsf{anaPfin_q}$. In fact, the mediating coalgebra morphism may call the coalgebra $c : X \to \mathsf{Pfin_q}\ X$ an arbitrarily large number of times, and, since we are given no information on the cardinality of X, each application of c may happen on a different input $x : X$. This implies that generally the recursion principle of set quotients would need to be invoked the same large number of times, and this could only be achieved by assuming the full axiom of choice.

5 Analysis of Worrell's Classical Set-Theoretic Construction

In classical set theory there are many constructions of the final coalgebra of the finite powerset functor. See Adámek et al.'s collection and comparison of all these characterizations [4]. In this section we scrutinize a construction due to James Worrell as an $(\omega + \omega)$-limit [33]. Worrell's general construction of final coalgebras of finitary functors as $(\omega + \omega)$-limits can be seen as a generalization of the traditional construction of final coalgebras of polynomial functors as ω-limits. In the same spirit, one can consider our attempt at internalizing Worrell's construction in type theory, here in the special case of the final powerset functor, as a generalization of Ahrens et al.'s internalization of the construction of M-types in homotopy type theory [5]. We will see that a sprinkle of classical logic is needed for Worrell's construction to work in our constructive setting.

Worrell's construction starts by considering the ω-limit of the sequence

$$1 \xleftarrow{\ !\ } \mathsf{Pfin}\ 1 \xleftarrow{\mathsf{map}_{\mathsf{Pfin}}\ !} \mathsf{Pfin}^2\ 1 \xleftarrow{\mathsf{map}^2_{\mathsf{Pfin}}\ !} \dots \tag{12}$$

which in type theory can be encoded as the following dependent sum:

$$\mathsf{V}_\omega =_{\mathrm{df}} \sum x : (n : \mathbb{N}) \to \mathsf{Pfin}^n\ 1.\ (n : \mathbb{N}) \to \mathsf{map}^n_{\mathsf{Pfin}}\ !\ (x\ (\mathsf{suc}\ n)) = x\ n$$

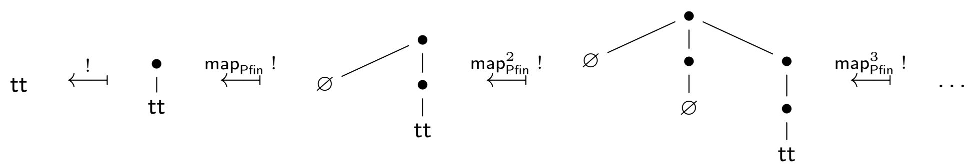
Here $\mathsf{Pfin}^n\ A$ is the n-iterated application of Pfin to type A, i.e. $\mathsf{Pfin}^{\mathsf{zero}}\ A =_{\mathrm{df}} A$ and $\mathsf{Pfin}^{\mathsf{suc}\ n} =_{\mathrm{df}} \mathsf{Pfin}\ (\mathsf{Pfin}^n\ A)$. Similarly, $\mathsf{map}^n_{\mathsf{Pfin}}$ is the n-iterated application of $\mathsf{map}_{\mathsf{Pfin}}$. Let ℓ_n be the function mapping an element of V_ω to its nth approximation in $\mathsf{Pfin}^n\ 1$, i.e. $\ell_n\ x =_{\mathrm{df}} \mathsf{fst}\ x\ n$. A function $\mathsf{alg}_{\mathsf{V}_\omega}$ from $\mathsf{Pfin}\ \mathsf{V}_\omega$ to V_ω can be constructed as follows, basically using the universal property of the ω-limit (we use copatterns and we only show the definition of the first projection): $\mathsf{fst}\ (\mathsf{alg}_{\mathsf{V}_\omega}\ s)\ n =_{\mathrm{df}} \mathsf{map}^n_{\mathsf{Pfin}}\ !\ (\mathsf{map}_{\mathsf{Pfin}}\ \ell_n\ s)$. As noticed by Adámek and Koubek [2], V_ω is not the final coalgebra of Pfin. This is because V_ω is not a fixpoint of Pfin, as the canonical algebra function $\mathsf{alg}_{\mathsf{V}_\omega}$ is not an isomorphism.

▶ **Proposition 5.** *The function* $\mathsf{alg}_{\mathsf{V}_\omega} : \mathsf{Pfin}\ \mathsf{V}_\omega \to \mathsf{V}_\omega$ *is not surjective.*

Proof. Consider the sequence

$$\begin{array}{ll}
\mathsf{growing} : (n : \mathbb{N}) \to \mathsf{Pfin}^n 1 & \\
\mathsf{growing\ zero} & =_{\mathrm{df}} \mathsf{tt} \\
\mathsf{growing\ (suc\ zero)} & =_{\mathrm{df}} \eta\ \mathsf{tt} \\
\mathsf{growing}\ (\mathsf{suc}\ (\mathsf{suc}\ n)) & =_{\mathrm{df}} \eta\ \varnothing \cup \mathsf{map}_{\mathsf{Pfin}}\ \eta\ (\mathsf{growing}\ (\mathsf{suc}\ n))
\end{array}$$

corresponding pictorially to the following element of V_ω:



The top-level branching of the sequence $\mathsf{growing}$ is, as the name suggests, growing. It is possible to show that it is absurd to assume that $\mathsf{growing}$ is in the image of $\mathsf{alg}_{\mathsf{V}_\omega}$. ◀

Elements of type V_ω represent non-wellfounded trees with unordered branching (as opposed to elements of type Tree, in which branching is ordered). The element $\mathsf{growing}$ introduced in the proof of Proposition 5 shows that these trees generally do not have finite branching, even if all their finite approximations do. So V_ω cannot possibly be a fixpoint of Pfin, and, in particular, it cannot be its final coalgebra.

While the sequence in (12) does not stabilize in ω steps, Worrell shows that, in classical set theory, it stabilized after $\omega + \omega$ steps. To this end, he considers the ω-limit $\mathsf{V}_{\omega+\omega}$ of the sequence

$$\mathsf{V}_\omega \xleftarrow{\mathsf{alg}_{\mathsf{V}_\omega}} \mathsf{Pfin}\ \mathsf{V}_\omega \xleftarrow{\mathsf{map}_{\mathsf{Pfin}}\ \mathsf{alg}_{\mathsf{V}_\omega}} \mathsf{Pfin}^2\ \mathsf{V}_\omega \xleftarrow{\mathsf{map}^2_{\mathsf{Pfin}}\ \mathsf{alg}_{\mathsf{V}_\omega}} \dots \tag{13}$$

which in type theory corresponds to the dependent sum

$$\mathsf{V}_{\omega+\omega} =_{\mathrm{df}} \sum x : (n : \mathbb{N}) \to \mathsf{Pfin}^n\ \mathsf{V}_\omega.\ (n : \mathbb{N}) \to \mathsf{map}^n_{\mathsf{Pfin}}\ \mathsf{alg}_{\mathsf{V}_\omega}\ (x\ (\mathsf{suc}\ n)) = x\ n$$

and proves that $\mathsf{V}_{\omega+\omega}$ is the final coalgebra of Pfin. A fundamental ingredient in his proof is the fact that the function $\mathsf{alg}_{\mathsf{V}_\omega}$ is injective (even more, classically it is a split monomorphism), so that $\mathsf{Pfin}\ \mathsf{V}_\omega$ can be characterized as the subset of V_ω consisting of all the trees in which the top-level branching is finite. Consequently, $\mathsf{Pfin}^2\ \mathsf{V}_\omega$ consists of all trees in which the first two levels of branching are finite. The limit $\mathsf{V}_{\omega+\omega}$ can then be characterized as the subset of V_ω consisting of trees with finite branching at all levels.

In our constructive setting, the injectivity of $\mathsf{alg}_{\mathsf{V}_\omega}$ is not provable. In fact, under the assumption of the axiom of countable choice, injectivity of $\mathsf{alg}_{\mathsf{V}_\omega}$ is equivalent to the *lesser limited principle of omniscience (LLPO)*:

$$\begin{aligned}\mathsf{LLPO} =_{\mathrm{df}}\ & (a : \mathbb{N} \to \mathsf{Bool}) \to \mathsf{isProp}\ (\textstyle\sum n : \mathbb{N}.\ a\ n = \mathsf{true}) \\ & \to \|((n : \mathbb{N}) \to \mathsf{isEven}\ n \to a\ n = \mathsf{false}) + ((n : \mathbb{N}) \to \mathsf{isOdd}\ n \to a\ n = \mathsf{false})\|\end{aligned}$$

LLPO states that, if a Boolean stream a contains at most one occurrence of value true, then either all its even positions contain false or all its odd positions contain false. The axiom of countable choice is just AC in (11) with type A fixed to be $\mathbb{N}$, but we prefer to have a different (and more standard) equivalent formulation of countable choice that is directly applicable in the forthcoming constructions:

$$\mathsf{AC}_{\mathbb{N}} : (P : \mathbb{N} \to \mathsf{Type}) \to ((n : \mathbb{N}) \to \|P\ n\|) \to \|(n : \mathbb{N}) \to P\ n\|$$

Proving that the injectvity of $\mathsf{alg}_{\mathsf{V}_\omega}$ implies LLPO does not require countable choice. The proof is obtained as an adaptation of the proof of equivalence between LLPO and the completeness of finite sets of real numbers in Bishop-style constructive mathematics [22].

▶ **Theorem 6.** *From the injectivity of* $\mathsf{alg}_{\mathsf{V}_\omega}$ *we can construct the following term:*

$$\begin{aligned}&\mathsf{complete} : \{x\ y_1\ y_2 : \mathsf{V}_\omega\}\ (z : \mathbb{N} \to \mathsf{V}_\omega) \\ &\quad \to (p : (n : \mathbb{N}) \to z\ n = y_1 + z\ n = y_2)\ (q : (n : \mathbb{N}) \to \ell_n\ x = \ell_n\ (z\ n)) \\ &\quad \to x \in \eta\ y_1 \cup \eta\ y_2\end{aligned} \tag{14}$$

Proof. Assume given a sequence $z : \mathbb{N} \to \mathsf{V}_\omega$ with proof terms p and q as in the type above. Define two elements of $\mathsf{Pfin}\ \mathsf{V}_\omega$ as follows: $t =_{\mathrm{df}} \eta\ y_1 \cup \eta\ y_2$ and $s =_{\mathrm{df}} \eta\ x \cup t$. In order to prove $\mathsf{complete}\ z\ p\ q$, it is enough to show that $\mathsf{alg}_{\mathsf{V}_\omega}$ maps s and t to path equal elements, since then the injectivity of $\mathsf{alg}_{\mathsf{V}_\omega}$ would imply $s = t$ and therefore also $x \in t$. Proving $\mathsf{alg}_{\mathsf{V}_\omega}\ s = \mathsf{alg}_{\mathsf{V}_\omega}\ t$ is equivalent to show $\ell_n\ (\mathsf{alg}_{\mathsf{V}_\omega}\ s) = \ell_n\ (\mathsf{alg}_{\mathsf{V}_\omega}\ t)$ for all $n : \mathbb{N}$, which, unfolding the definition of $\mathsf{alg}_{\mathsf{V}_\omega}$, is also equivalent to show $\mathsf{map}_{\mathsf{Pfin}}\ \ell_n\ s = \mathsf{map}_{\mathsf{Pfin}}\ \ell_n\ t$ for all $n : \mathbb{N}$. Assuming $n : \mathbb{N}$, we invoke the antisymmetry of the subset relation and we are left to show $\mathsf{map}_{\mathsf{Pfin}}\ \ell_n\ s \subseteq \mathsf{map}_{\mathsf{Pfin}}\ \ell_n\ t$ (the other direction is trivial since $t \subseteq s$). Unfolding the definition of s, the only interesting case to prove is $\ell_n\ x \in \mathsf{map}_{\mathsf{Pfin}}\ \ell_n\ t$. The proof proceeds by case analysis on $p\ n$. If $z\ n = y_1$, then $\ell_n\ x = \ell_n\ (z\ n) = \ell_n\ y_1$, where the first path equality is given by $q\ n$, so $\ell_n\ x \in \mathsf{map}_{\mathsf{Pfin}}\ \ell_n\ t$. The case of $z\ n = y_2$ is analogous. ◀

Intuitively, $\mathsf{complete}$ corresponds to the completeness of two-element subsets of V_ω wrt. the pseudometric $d(x, y) =_{\mathrm{df}} \inf\{2^{-n} \mid n : \mathbb{N},\ \ell_n\ x = \ell_n\ y\}$ [4].

The proof of the next theorem requires the introduction of some auxiliary definitions. First, $\mathsf{long} : \mathsf{V}_\omega$ corresponds to the infinite tree in which each node has exactly one subtree. We only show the construction of the first projection (the definition uses copatterns).

$$\begin{aligned}
&\mathsf{fst\ long\ zero} &&=_{\mathrm{df}} \mathsf{tt}\\
&\mathsf{fst\ long}\ (\mathsf{suc}\ n) &&=_{\mathrm{df}} \eta\ (\mathsf{fst\ long}\ n)
\end{aligned}$$

Given a sequence $a : \mathbb{N} \to \mathsf{Bool}$, one can also define a variant $\mathsf{long?}\ a$ of long, which is the same as long if a contains only value false, but its height stop growing if there is $n : \mathbb{N}$ such that $a\ n$ is the first true in a. In the latter case, $\mathsf{long?}\ a$ is a finite tree with height n, so that $\mathsf{fst\ long}\ (\mathsf{suc}\ n)$ is not equal to $\mathsf{fst}\ (\mathsf{long?}\ a)\ (\mathsf{suc}\ n)$.

$$\begin{aligned}
&\mathsf{fst}\ (\mathsf{long?}\ a)\ \mathsf{zero} &&=_{\mathrm{df}} \mathsf{tt}\\
&\mathsf{fst}\ (\mathsf{long?}\ a)\ (\mathsf{suc}\ n) &&=_{\mathrm{df}} \mathsf{if}\ a\ \mathsf{zero\ then}\ \varnothing\ \mathsf{else}\ \eta\ (\mathsf{fst}\ (\mathsf{long?}\ (a \circ \mathsf{suc}))\ n)
\end{aligned}$$

Lastly, given a sequence $a : \mathbb{N} \to \mathsf{Bool}$, a type A with two elements $x, y : A$ and a Boolean b, we can define a sequence $\mathsf{seq}\ a\ x\ y\ b : \mathbb{N} \to A$ as follows:

$$\begin{aligned}
&\mathsf{seq}\ a\ x\ y\ b\ \mathsf{zero} &&=_{\mathrm{df}} \mathsf{if}\ a\ \mathsf{zero\ and}\ b\ \mathsf{then}\ y\ \mathsf{else}\ x\\
&\mathsf{seq}\ a\ x\ y\ b\ (\mathsf{suc}\ n) &&=_{\mathrm{df}} \mathsf{if}\ a\ \mathsf{zero\ and}\ b\ \mathsf{then}\ y\ \mathsf{else\ seq}\ (a \circ \mathsf{suc})\ x\ y\ (\mathsf{not}\ b)\ n
\end{aligned}$$

The rationale behind the construction of the latter sequence, in the case when b is true, goes as follows: $\mathsf{seq}\ a\ x\ y\ \mathsf{true}\ n$ returns y if there exists an even number $k : \mathbb{N}$ with $k \leq n$ such that $a\ k = \mathsf{true}$ and $a\ j = \mathsf{false}$ for all $j < k$; in all other cases $\mathsf{seq}\ a\ x\ y\ \mathsf{true}\ n$ returns x.

▶ **Theorem 7.** *The existence of a term* $\mathsf{complete}$ *as in (14) implies* LLPO*.*

Proof. Let $a : \mathbb{N} \to \mathsf{Bool}$ be a sequence with at most one occurrence of value true. Define $y_1 =_{\mathrm{df}} \mathsf{long}$, $y_2 =_{\mathrm{df}} \mathsf{long?}\ a$ and $z =_{\mathrm{df}} \mathsf{seq}\ a\ y_1\ y_2\ \mathsf{true}$. Take x to be the diagonal of z, i.e. $\mathsf{fst}\ x\ n =_{\mathrm{df}} \ell_n\ z\ n$, which can in fact be proved to be an element of V_ω. Clearly each entry in z is either y_1 or y_2, therefore all the hypotheses in the type in (14) are satisfied. Applying $\mathsf{complete}$ to these hypotheses gives $x \in \eta\ y_1 \cup \eta\ y_2$. Invoking the recursion principle of propositional truncation on the resulting proof term, which we are allowed to use since the return type of LLPO is a proposition, gives us either $x = y_1$ or $x = y_2$. Assume $x = y_1$, we show $a\ n = \mathsf{false}$ for all even numbers $n : \mathbb{N}$. Suppose $a\ n = \mathsf{true}$ for a certain even number n. Since $a\ n = \mathsf{true}$, and this is the only true in a, we know that $z\ (\mathsf{suc}\ n) \equiv \mathsf{seq}\ a\ y_1\ y_2\ \mathsf{true}\ (\mathsf{suc}\ n) = y_2$ which in turn implies $\ell_{\mathsf{suc}\ n}\ x = \ell_{\mathsf{suc}\ n}\ (z\ (\mathsf{suc}\ n)) = \ell_{\mathsf{suc}\ n}\ y_2$. By assumption $\ell_{\mathsf{suc}\ n}\ x = \ell_{\mathsf{suc}\ n}\ y_1$, therefore by path composition and path inversion we get $\ell_{\mathsf{suc}\ n}\ y_1 = \ell_{\mathsf{suc}\ n}\ y_2$, i.e. $\mathsf{fst\ long}\ (\mathsf{suc}\ n) = \mathsf{fst}\ (\mathsf{long?}\ a)\ (\mathsf{suc}\ n)$, which is impossible since $a\ n$ is true. So $a\ n$ must be false for all even n. Analogously one can prove that $x = y_2$ implies $a\ n = \mathsf{false}$ for all odd numbers $n : \mathbb{N}$, therefore concluding the derivation of LLPO. ◀

Patching together Theorems 6 and 7 shows that the injectivity of $\mathsf{alg}_{\mathsf{V}_\omega}$ implies LLPO.

▶ **Corollary 8.** *The injectivity of* $\mathsf{alg}_{\mathsf{V}_\omega}$ *implies* LLPO*.*

This displays the non-constructive nature of the injectivity of $\mathsf{alg}_{\mathsf{V}_\omega}$. The reverse implication also holds, which we have proved assuming the axiom of countable choice. We refrain from proving this in the paper, but the interested reader can find all the details in the Agda code.

▶ **Theorem 9.** *Assuming countable choice,* LLPO *implies the injectivity of* $\mathsf{alg}_{\mathsf{V}_\omega}$*.*

One can also modify the proofs of Corollary 8 and Theorem 9 to show that LLPO is also equivalent to the injectivity of the function $\ell_\omega : \mathsf{V}_{\omega+\omega} \to \mathsf{V}_\omega$ given by $\ell_\omega\ x =_{\mathrm{df}} \mathsf{fst}\ x\ \mathsf{zero}$. This demonstrates that Worrell's construction of the final coalgebra of Pfin as a subset of the limit V_ω is not achievable without the assumption of a certain amount of classical logic in the metatheory.

▶ **Theorem 10.**
1. *The injectivity of ℓ_ω implies* LLPO*.*
2. *Assuming countable choice,* LLPO *implies the injectivity of ℓ_ω.*

Having the injectivity of $\mathsf{alg}_{\mathsf{V}_\omega}$ at hand, the construction of a coalgebra structure on $\mathsf{V}_{\omega+\omega}$ and the proof of its finality morally follow Worrell's description [33].

▶ **Theorem 11.** *Assuming the axiom of countable choice and the injectvity of* $\mathsf{alg}_{\mathsf{V}_\omega}$*,* $\mathsf{V}_{\omega+\omega}$ *is a* Pfin*-coalgebra which is final.*

Proof. The meat of the proof lays in the construction of a function of type $\mathsf{V}_{\omega+\omega} \to \mathsf{Pfin}\ \mathsf{V}_{\omega+\omega}$. We show that the latter comes from an equivalence $\mathsf{V}_{\omega+\omega} \simeq \mathsf{Pfin}\ \mathsf{V}_{\omega+\omega}$ which is constructed in several steps. First define a family of functions $u : (n : \mathbb{N}) \to \mathsf{Pfin}^n\ \mathsf{V}_\omega \to \mathsf{V}_\omega$ by recursion: $u\ \mathsf{zero}\ x =_{\mathrm{df}} x$ and $u\ (\mathsf{suc}\ n)\ x =_{\mathrm{df}} u\ n\ (\mathsf{map}^n_{\mathsf{Pfin}}\ \mathsf{alg}_{\mathsf{V}_\omega}\ x)$. It is possible to prove that $\mathsf{V}_{\omega+\omega}$ is equivalent to the wide pullback $\bigcap u$ of the family of functions u. In general, given a family of functions $f : (n : \mathbb{N}) \to A\ n \to C$, its *wide pullback* is defined in type theory as

$$\bigcap f \ =_{\mathrm{df}} \sum x : (n : \mathbb{N}) \to A\ n.\ (n : \mathbb{N}) \to f\ (\mathsf{suc}\ n)\ (x\ (\mathsf{suc}\ n)) = f\ \mathsf{zero}\ (x\ \mathsf{zero})$$

We use the intersection symbol, and we refer to this pullback as *intersection*, since all the families of functions f that we consider have $f\ n$ injective, for all $n : \mathbb{N}$. In particular, each function $u\ n$ is injective, which can be proved by induction on n using the assumption that $\mathsf{alg}_{\mathsf{V}_\omega}$ is injective. This implies the existence of an equivalence $\mathsf{eqv}_1 : \mathsf{Pfin}\ \mathsf{V}_{\omega+\omega} \simeq \mathsf{Pfin}\ (\bigcap u)$.

In an analogous manner, one can prove that the intersection of the family $\mathsf{map}_{\mathsf{Pfin}} \circ u : (n : \mathbb{N}) \to \mathsf{Pfin}^{\mathsf{suc}\ n}\ \mathsf{V}_\omega \to \mathsf{Pfin}\ \mathsf{V}_\omega$ is equivalent to the ω-limit of the shifted sequence

$$\mathsf{Pfin}\ \mathsf{V}_\omega \xleftarrow{\mathsf{map}_{\mathsf{Pfin}}\ \mathsf{alg}_{\mathsf{V}_\omega}} \mathsf{Pfin}^2\ \mathsf{V}_\omega \xleftarrow{\mathsf{map}^2_{\mathsf{Pfin}}\ \mathsf{alg}_{\mathsf{V}_\omega}} \mathsf{Pfin}^3\ \mathsf{V}_\omega \xleftarrow{\mathsf{map}^3_{\mathsf{Pfin}}\ \mathsf{alg}_{\mathsf{V}_\omega}} \cdots$$

It is well-known that the ω-limit of the shifted sequence is equivalent to the ω-limit of the original sequence in (13), i.e. $\mathsf{V}_{\omega+\omega}$. We obtain an equivalence $\mathsf{eqv}_3 : \bigcap(\mathsf{map}_{\mathsf{Pfin}} \circ u) \simeq \mathsf{V}_{\omega+\omega}$.

It is also possible to show, using the axiom of countable choice, that Pfin preserves intersections: given a generic family of injective functions $f : (n : \mathbb{N}) \to A\ n \to C$, the following equivalence exists: $\mathsf{eqv}_2 : \mathsf{Pfin}\ (\bigcap f) \simeq \bigcap(\mathsf{map}_{\mathsf{Pfin}} \circ f)$.

By composing equivalences $\mathsf{eqv}_1, \mathsf{eqv}_2$ and eqv_3, we obtain the desired equivalence showing that $\mathsf{V}_{\omega+\omega}$ is a fixpoint of Pfin. A Pfin-coalgebra for $\mathsf{V}_{\omega+\omega}$ is extracted as the function of type $\mathsf{V}_{\omega+\omega} \to \mathsf{Pfin}\ \mathsf{V}_{\omega+\omega}$ underlying this equivalence. It is possible to continue following Worrell's proof and show that this coalgebra is indeed final. ◀

6 Conclusions and Future Work

In this paper we discussed various presentations of the final coalgebra of the finite powerset functor in Cubical Agda: (i) as a setoid, (ii) as a coinductive type, (iii) as a set quotient and (iv) as a subset of an ω-limit. Construction (iii) requires the presence of the axiom of choice in the proof of finality, while construction (iv) corresponds to the classical construction of the final coalgebra as a $(\omega + \omega)$-limit by Worrell, which can be performed in our setting prior the assumption of countable choice and LLPO. For these reasons, we believe the best choice to be number (*ii*), i.e. the coinductive type $\nu\mathsf{Pfin}$ of Section 4.2, since it does not require the assumption of classical principles such as choice or LLPO, and it does not force the user to employ setoids instead of types.

The work presented in this paper is motivated by our will to certify programming language semantics in proof assistants. We are specifically thinking about languages with nondeterministic or concurrent behavior. In previous work [29], we presented a fully abstract denotational model of the early π-calculus, mechanized in Guarded Cubical Agda. We believe possible to port these result to Cubical Agda using the presentations of the final Pfin-coalgebra of Section 4.2. Such an attempt would employ Cubical Agda's coinductive types instead of the guarded recursive types of Guarded Cubical Agda.

We wish to study the more general construction of final coalgebras of finitary functors in type theory. Frumin et al.'s functor Pfin captures a particular notion of finite type, known as Kuratowski finiteness: a type A is finite iff there exists a pair consisting of $x : \mathsf{Pfin}\ A$ and a proof that $(a : A) \to a \in x$. But in type theory, and more generally in constructive mathematics, there exist many more inequivalent formulations of finiteness [12, 25, 15, 16, 17]. We plan to investigate final coalgebras of finitary functors using these various formulations. In particular, we wonder if an alternative notion of finiteness in the specification of the finite powerset functor would make Worrell's proof go through without the assumption of additional classical principles. A large class of finitary functors should be definable via the syntax for set truncated HITs developed by Basold, Geuvers and van der Weide [8, 31].

The construction of the final coalgebra given in Section 4.2 used a higher inductive type in the domain of a coinductive type destructor. This definition is allowed in Cubical Agda, and it is intuitively justified by the treatment of HITs in cubical type theory as inductive types with constructors possibly depending on extra interval variables [11, 9]. We leave to future work a formal construction of the final coalgebra of the finite powerset and other finitary functors in the cubical set model [10]. Inspiration could be drawn from the recent model of clocked cubical type theory of Kristensen et al. [20], where HITs are shown to commute on the nose with limits modelling the notion of clock quantification.

References

1 Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. In Roberto Giacobazzi and Radhia Cousot, editors, *Proc. of 40th Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'13*, pages 27–38. ACM, 2013. doi:10.1145/2429069.2429075.

2 Jirí Adámek and Václav Koubek. On the greatest fixed point of a set functor. *Theoretical Computer Science*, 150(1):57–75, 1995. doi:10.1016/0304-3975(95)00011-K.

3 Jirí Adámek, Paul Blain Levy, Stefan Milius, Lawrence S. Moss, and Lurdes Sousa. On final coalgebras of power-set functors and saturated trees. *Applied Categorical Structures*, 23(4):609–641, 2015. doi:10.1007/s10485-014-9372-9.

4 Jiří Adámek, Stefan Milius, and Lawrence S. Moss. Initial algebras, terminal coalgebras, and the theory of fixed points of functors. Draft book, available from http://www.stefan-milius.eu, 2021.

5 Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. Non-wellfounded trees in homotopy type theory. In Thorsten Altenkirch, editor, *Proc. of 13th Int. Conf. on Typed Lambda Calculi and Applications, TLCA'15*, volume 38 of *Leibniz International Proceedings in Informatics*, pages 17–30. Schloss Dagstuhl, 2015. doi:10.4230/LIPIcs.TLCA.2015.17.

6 Michael Barr. Terminal coalgebras in well-founded set theory. *Theoretical Computer Science*, 114(2):299–315, 1993. doi:10.1016/0304-3975(93)90076-6.

7 Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293, 2003. doi:10.1017/S0956796802004501.

8 Henning Basold, Herman Geuvers, and Niels van der Weide. Higher inductive types in programming. *Journal of Universal Computer Science*, 23(1):63–88, 2017. doi:10.3217/jucs-023-01-0063.

9 Evan Cavallo and Robert Harper. Higher inductive types in cubical computational type theory. *PACMPL*, 3(POPL):1:1–1:27, 2019. doi:10.1145/3290314.

10 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In Tarmo Uustalu, editor, *Proc. of 21st Int. Conf. on Types for Proofs and Programs, TYPES'15*, volume 69 of *Leibniz International Proceedings in Informatics*, pages 5:1–5:34. Schloss Dagstuhl, 2015. doi:10.4230/LIPIcs.TYPES.2015.5.

11 Thierry Coquand, Simon Huber, and Anders Mörtberg. On higher inductive types in cubical type theory. In Anuj Dawar and Erich Grädel, editors, *Proc. of the 33rd Ann. ACM/IEEE Symp. on Logic in Computer Science, LICS'18*, pages 255–264. ACM, 2018. doi:10.1145/3209108.3209197.

12 Thierry Coquand and Arnaud Spiwack. Constructively finite? In Laureano Lambàn, Ana Romero, and Julio Rubio, editors, *Scientific Contributions in Honor of Mirian Andrés Gómez*, pages 217–230. Universidad de La Rioja, 2010.

13 Nils Anders Danielsson. Bag equivalence via a proof-relevant membership relation. In Lennart Beringer and Amy P. Felty, editors, *Proc. of 3rd Int. Conf. on Interactive Theorem Proving, ITP'12*, volume 7406 of *Lecture Notes in Computer Science*, pages 149–165. Springer, 2012. doi:10.1007/978-3-642-32347-8_11.

14 Nils Anders Danielsson. Up-to techniques using sized types. *PACMPL*, 2(POPL):43:1–43:28, 2018. doi:10.1145/3158131.

15 Denis Firsov and Tarmo Uustalu. Dependently typed programming with finite sets. In Patrick Bahr and Sebastian Erdweg, editors, *Proc. of 11th ACM SIGPLAN Workshop on Generic Programming, WGP'15*, pages 33–44. ACM, 2015. doi:10.1145/2808098.2808102.

16 Denis Firsov, Tarmo Uustalu, and Niccolò Veltri. Variations on Noetherianness. In Robert Atkey and Neelakantan R. Krishnaswami, editors, *Proc. of 6th Wksh. on Mathematically Structured Functional Programming, MSFP'16*, volume 207 of *Electronic Proceedings in Theoretical Computer Science*, pages 76–88, 2016. doi:10.4204/EPTCS.207.4.

17 Dan Frumin, Herman Geuvers, Léon Gondelman, and Niels van der Weide. Finite sets in homotopy type theory. In *Proc. of 7th ACM SIGPLAN Int. Conf. on Certified Programs and Proofs, CPP'18*, pages 201–214. ACM, 2018. doi:10.1145/3167085.

18 John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proc. of 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'96*, pages 410–423, 1996. doi:10.1145/237721.240882.

19 Yoshiki Kinoshita and John Power. Category theoretic structure of setoids. *Theoretical Computer Science*, 546:145–163, 2014. doi:10.1016/j.tcs.2014.03.006.

20 Magnus Baunsgaard Kristensen, Rasmus Ejlers Møgelberg, and Andrea Vezzosi. A model of clocked cubical type theory, 2021. arXiv:2102.01969.

21 Paul Blain Levy. Similarity quotients as final coalgebras. In Martin Hofmann, editor, *Proc. of 14th Int. Conf on Foundations of Software Science and Computational Structures, FoSSaCS'11*, volume 6604 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2011. doi:10.1007/978-3-642-19805-2_3.

22 Mark Mandelkern. Constructively complete finite sets. *Mathematical Logic Quarterly*, 34(2):97–103, 1988. doi:10.1002/malq.19880340202.

23 Robin Milner. *A calculus of communicating systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. doi:10.1007/3-540-10235-3.

24 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992. doi:10.1016/0890-5401(92)90008-4.

25 Erik Parmann. Investigating streamless sets. In Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau, editors, *Proc. of 20th Int. Conf. on Types for Proofs and Programs, TYPES'14*, volume 39 of *Leibniz International Proceedings in Informatics*, pages 187–201. Schloss Dagstuhl, 2014. doi:10.4230/LIPIcs.TYPES.2014.187.

26 Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000. doi:10.1016/S0304-3975(00)00056-6.

27 The Univalent Foundations Program. *Homotopy type theory: Univalent foundations of mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

28 Niccolò Veltri. *A type-theoretical study of nontermination.* PhD thesis, Tallinn University of Technology, 2017. URL: https://digi.lib.ttu.ee/i/?7631.

29 Niccolò Veltri and Andrea Vezzosi. Formalizing π-calculus in Guarded Cubical Agda. In Jasmin Blanchette and Catalin Hritcu, editors, *Proc. of 9th ACM SIGPLAN Int. Conf. on Certified Programs and Proofs, CPP'20*, pages 270–283. ACM, 2020. doi:10.1145/3372885.3373814.

30 Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *PACMPL*, 3(ICFP):87:1–87:29, 2019. doi:10.1145/3341691.

31 Niels van der Weide and Herman Geuvers. The construction of set-truncated higher inductive types. In Barbara König, editor, *Proc. of 35th Int. Conf. on Mathematical Foundations of Programming Semantics, MFPS'19*, volume 347 of *Electronic Notes in Theoretical Computer Science*, pages 261–280. Elsevier, 2019. doi:10.1016/j.entcs.2019.09.014.

32 James Worrell. Terminal sequences for accessible endofunctors. In Bart Jacobs and Jan J. M. M. Rutten, editors, *Proc. of 2nd Int. Wksh. on Coalgebraic Methods in Computer Science, CMCS'99*, volume 19 of *Electronic Notes in Theoretical Computer Science*, pages 24–38. Elsevier, 1999. doi:10.1016/S1571-0661(05)80267-1.

33 James Worrell. On the final sequence of a finitary set functor. *Theoretical Computer Science*, 338(1-3):184–199, 2005. doi:10.1016/j.tcs.2004.12.009.

Resource Transition Systems and Full Abstraction for Linear Higher-Order Effectful Programs

Ugo Dal Lago
University of Bologna, Italy
INRIA Sophia Antipolis, France

Francesco Gavazzo
University of Bologna, Italy
INRIA Sophia Antipolis, France

Abstract

We investigate program equivalence for linear higher-order (sequential) languages endowed with primitives for computational effects. More specifically, we study operationally-based notions of program equivalence for a *linear* λ-calculus with *explicit copying* and *algebraic effects à la* Plotkin and Power. Such a calculus makes explicit the interaction between copying and linearity, which are *intensional* aspects of computation, with effects, which are, instead, *extensional*. We review some of the notions of equivalences for linear calculi proposed in the literature and show their limitations when applied to effectful calculi where copying is a first-class citizen. We then introduce *resource transition systems*, namely transition systems whose states are built over tuples of programs representing the available resources, as an operational semantics accounting for both intensional and extensional interactive behaviours of programs. Our main result is a *sound and complete* characterization of contextual equivalence as *trace equivalence* defined on top of resource transition systems.

2012 ACM Subject Classification Theory of computation → Operational semantics

Keywords and phrases algebraic effects, linearity, program equivalence, full abstraction

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.23

Related Version *Full Version*: `https://arxiv.org/abs/2106.12849`

Funding The authors are supported by the ERC Consolidator Grant DLV-818616 DIAPASoN.

1 Introduction

This work aims to study operationally-based equivalences for higher-order sequential programming languages enjoying three main features, which we are going to explain: *algebraic effects*, *linearity*, and *explicit copying*.

Algebraic Effects. Since the early days of programming language semantics, the study of computational effects, i.e. those aspects of computations that go beyond the pure process of computing, has been of paramount importance. Starting with the seminal work by Moggi [49, 50], modelling and understanding computational effects in terms of monads [43] has been a standard practice in the denotational semantics of higher-order sequential languages. More recently, Plotkin and Power [60, 57, 58] have extended the analysis of computational effects in terms of monads to *operational semantics*, introducing the theory of *algebraic effects*. Accordingly, computational effects are produced by effect-triggering operations whose behaviour is, in essence, algebraic. Examples of such operations are nondeterministic and probabilistic choices, primitives for I/O, primitives for reading and writing from a global store, and many others. The operational analysis of computational effects in terms of algebraic operations also gave new insights not only on the operational semantics of effectful programming languages but also on their theories of equality, this way leading to the development of, e.g., effectful logical relations [36, 12], effectful applicative and normal form/open bisimulation [21, 19], and logic-based equivalences [67, 46].

Linearity and Copying. The analysis of effectful computations in terms of monads and algebraic effects is, in its very essence, *extensional*: ultimately, a program represents a function from inputs to monadic outputs. However, when reasoning about computational effects, also *intensional* aspects of programs may be relevant. In particular, *linearity* [34, 69, 8] (and its quantitative refinements [33, 32, 14, 4, 23]) has been recognised as a fundamental tool to reason about computational effects [28, 48], as witnessed by a number of programming languages, such as Clean [55], Rust [47], Granule [52], and Linear Haskell [9], which explicitly rely on linearity to structure and manage effects. Indeed, the interaction between linearity, copying, and computational effects deeply influences program equivalence: there are effectful programs that cannot be discriminated without allowing the environment to copy them, and thus program transformations which are *sound* if linearity is guaranteed, but *unsound* in presence of copying.

A simple, yet instructive example of such a transformation, which we will carefully examine in the next section, is given by distributivity of λ-abstraction over probabilistic choice operators: $\lambda x.(e \oplus f) \simeq (\lambda x.e) \oplus (\lambda x.f)$. This transformation is well-known to be unsound for "classical" call-by-value probabilistic languages [16]. However, it is sound if the programs involved cannot be copied [27, 26]. What, instead, we expect to be unsound is the transformation $!(e \oplus f) \simeq !e \oplus !f$, where the operator ! (bang) is the usual linear logic exponential modality making terms under its scope copyable and erasable. It is thus natural to ask if, and to what extent, the aforementioned notions of effectful program equivalence can be extended to *linear* languages with *explicit copying*.

Our Contribution. In this paper we introduce *resource transition systems* as an intensional, resource-sensitive operational semantics for linear languages with algebraic operations and explicit copying. Resource transition systems combine standard *extensional* properties of effectful computations with linearity and copying, whose nature is, instead, *intensional*. We model the former using monads – as one does for ordinary effectful semantics – and the latter by shifting from program-based transition systems to *tuple-based* transition systems, as one does in environmental bisimulation [62, 44]. Indeed, a resource transition system can be thought of as an ordinary transition system whose states are built over tuples of copyable programs and linear values representing the available resources produced by a program while interacting with the external environment. Another possible way to look at resource transition systems is as an interactive semantics defined on top of the so-called storage model [68]. We then define and study trace equivalence on resource transition systems. Our main result states that trace equivalence is *sound* and *complete* for contextual equivalence. To the best of the authors' knowledge, this is the first full abstraction result for a linear λ-calculus with arbitrary algebraic effects and explicit copying.

Outline This paper is structured as follows. After an informal introduction to program equivalence for effectful linear languages (Section 2), Section 3 recalls some background notions on monads and algebraic operations. Section 4 introduces our vehicle calculus and its operational semantics. Resource-sensitive resource transition systems and their associated notions of equivalence are given in Section 5. Due to space constraints, several details have been omitted. The interested reader can find them in the extended version of the present paper [20].

2 Effects, Linearity, and Program Equivalence

In this section, we give a gentle introduction to program equivalence in presence of linearity, explicit copying, and effects. In this work, we are concerned with *operationally-based* equivalences, example of those being contextual and CIU equivalences [51, 45], logical relations [61, 56, 66] and, bisimulation-based equivalences [1, 40, 41, 62]. Moreover, among operationally-based equivalences, we seek for lightweight ones, by which we mean equivalences which are as easy to use as possible (otherwise, contextual equivalence would be enough). Accordingly, we do not consider equivalences in the spirit of logical relations – which usually require heavy techniques such as biorthogonality [54] and step-indexing [3] when applied to calculi in which recursion is present, either at the level of types or at the level of terms. Instead, we focus on *first-order* equivalences [44], viz. notions of trace equivalence and bisimilarity.

Our running examples in this paper are the already mentioned distributivity of (lambda) abstraction and bang over (fair) probabilistic choice in probabilistic call-by-value λ-calculi [24, 18, 27]:

$$\lambda x.(e \oplus f) \simeq (\lambda x.e) \oplus (\lambda x.f) \tag{λ-dist}$$

$$!(e \oplus f) \simeq\ !e \oplus !f \tag{!-dist}$$

It is well-known [16] that in call-by-value probabilistic languages, lambda abstraction does not distribute over probabilistic choice. In a linear setting, however, we see that any resource-sensitive notion of program equivalence $\simeq$ should actually validate the equivalence (λ-dist) but not (!-dist). Why? Let us look at the transition systems describing the (interactive) behaviour (Figure 1) of the programs involved in (λ-dist), where we write $[\![e]\!]$ for the result of the evaluation of an expression e. One way to understand the failure of the equivalence (λ-dist)

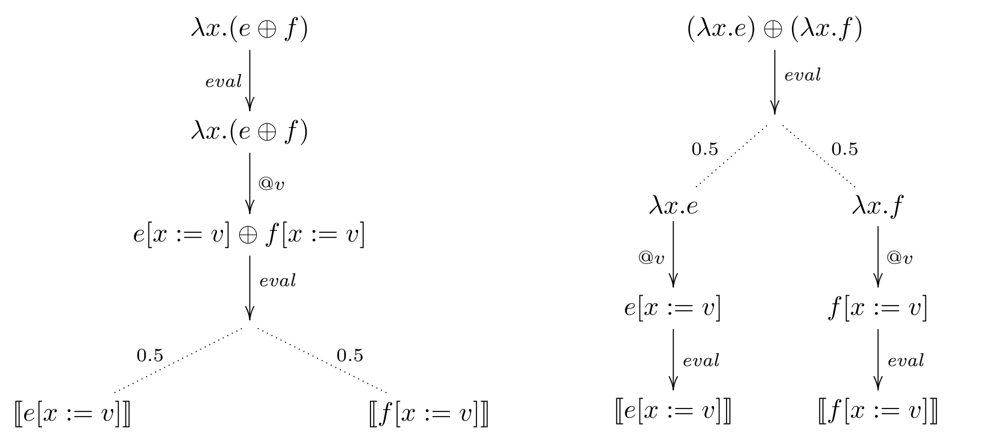


Figure 1 Interactive behaviour of $\lambda x.(e \oplus f)$ and $(\lambda x.e) \oplus (\lambda x.f)$.

in *classical* (i.e. resource-agnostic) languages is that several notions of probabilistic program equivalence (such as probabilistic contextual equivalence [24], applicative bisimilarity [16, 24], and logical relations [13]) are sensitive to branching. However, sensitivity to branching does not quite feel like the crux of the failure of distributivity of abstraction over choice in classical languages. In fact, what we see is that $\lambda x.(e \oplus f)$ waits for an input, and then resolves the probabilistic choice. Dually, $(\lambda x.e) \oplus (\lambda x.f)$ first resolves the choice, and then waits for an input. As a consequence, if we evaluate these programs, $\lambda x.(e \oplus f)$ essentially does nothing, whereas $(\lambda x.e) \oplus (\lambda x.f)$ probabilistically chooses if continuing with either $\lambda x.e$ or $\lambda x.f$. At this point, there is a crucial difference between the programs obtained: $\lambda x.(e \oplus f)$ still has to resolve the probabilistic choice. If we were allowed to use it twice by passing it an argument v – this way resolving the choice twice – then we could observe a (probabilistic) behaviour different from both the one of $\lambda x.e$ and of $\lambda x.f$. Indeed, assuming $f[x := v]$ to diverge and $e[x := v]$ to converge (with probability 1), then, we would converge (to $e[x := v]$) with probability 0.25, in the former case, and with probability 0.5, in the latter case. To observe such a behaviour, however, it is crucial to *copy* $\lambda x.(e \oplus f)$. Otherwise, we could only interact with it by passing it an argument only *once*, this way validating (λ-dist).

Summing up, to invalidate (λ-dist) one has to be able to *copy* the results of the evaluation of the programs involved. This observation suggests that the deep reason why (λ-dist) fails relies on the copying capabilities of the calculus [63]. If the calculus at hand is linear (and thus offers no copying capability), we should then expect (λ-dist) to hold, while $!\lambda x.(e \oplus f) \simeq !(\lambda x.e) \oplus !(\lambda x.f)$ (and thus ultimately (!-dist)) to fail. This agrees with a recent result by Deng and Zhang [27, 26], who observed that if a calculus does not have copying capabilities, then contextual equivalence (which is *a fortiori* linear) validates (λ-dist). More generally, Deng and Zhang showed that *linear contextual equivalence*, i.e. contextual equivalence where contexts test their arguments linearly (viz. exactly once), coincides with *linear trace equivalence* in probabilistic languages.

But what about (!-dist)? Unfortunately, linear trace equivalence has been designed for linear languages *without* copying, only. Moreover, straightforward extensions of linear trace equivalence to languages with copying would actually validate (!-dist), trace equivalence being insensitive to branching. The situation does not change much if one looks at different forms of equivalence, such as Bierman's applicative bisimilarity [10]. Such equivalences usually invalidate (!-dist), but they all invalidate (λ-dist), too. We interpret all of this as a symptom of the lack of intensional structure in the aforementioned notions of equivalence. Ultimately, this can be traced back to the very operational semantics of the calculus, which is meant to be an abstract description of the input-output behaviour of programs, but gives no insight into their *intensional* structure, i.e. linearity and copying in our case [68].

We propose to overcome this deficiency by giving calculi a *resource-sensitive* operational semantics on top of which notions of program equivalence accounting for both intensional and extensional aspects of programs can be naturally defined. We do so by shifting from program-based transition systems to transition systems whose states are *tuples* $(\Gamma; \Delta)$, where Γ is a sequence of *non-linear* (hence copyable) programs and Δ is a sequence of *linear* values, as states. Accordingly, fixed a tuple $(\Gamma; \Delta)$ and a program e, we evaluate e, say obtaining a value v, and add v to the linear environment Δ, this way describing the *extensional* behaviour of the program. There are two *intensional* actions we can make on tuples. If Δ contains a value of the form $!e$, then we can remove $!e$ from Δ and add e to Γ. Dually, once we have a program e in Γ, we can decide to evaluate it – and thus to possibly produce a new linear value – *without* removing it from Γ, this way reflecting its non-linear nature. Finally, we can interact with a value $\lambda x.f$ by passing it an argument built using programs in Γ and values in Δ. As the latter are linear, we will then remove them from Δ.

We conclude this section by remarking that although here we have focused on probabilistic languages, a similar analysis can be made for languages exhibiting different kinds of effects, such as input-output behaviours as well as combinations of effects (e.g. probabilistic nondeterminism and global stores).

3 Preliminaries: Monads and Algebraic Effects

Starting with the seminal work by Moggi [49, 50], *monads* have become a standard formalism to model and study computational effects in higher-order sequential languages. Instead of working with monads, we opt for the equivalent notion of a *Kleisli triple* [43]. Additionally, instead of defining monads on arbitrary categories, we tacitly restrict our analysis to the category of sets and functions.

▶ **Definition 1.** *A* Kleisli triple *is triple* $(T, \eta, \gg\!=)$ *consisting of a map associating to any set* X *a set* $T(X)$*, a set-indexed family of functions* $\eta_X : X \to T(X)$*, and a map* $\gg\!=$*, called* bind*, associating to each function* $f : X \to T(Y)$ *a function* $\gg\!=f : T(X) \to T(Y)$*. Additionally, these data must obey the following laws, for* f *and* g *functions with appropriate (co)domains:*

$$\gg\!=\eta = id; \qquad \gg\!=f \circ \eta = f; \qquad \gg\!=g \circ \gg\!=f = \gg\!=(\gg\!=g \circ f).$$

Following standard practice, we write $m \gg\!= f$ *for* $\gg\!=f(m)$*.*

The computational interpretation behind Kleisli triples is the following: if A is a set (or type) of values, then $T(A)$ represent the set of computations returning values in A. Accordingly, for each set A there is a function $\eta_A : A \to T(A)$ that regards a value $a \in A$ as a trivial computation returning a (and producing no effect). The map η corresponds to the programming constructor **return**. Similarly, $\mu \gg\!= f$ is the *sequential composition* of a computation $\mu \in T(A)$ with a function $f : A \to T(B)$, and corresponds to the sequencing constructor **let** $x = -$ **in** $-$. Following this interpretation, we can read the identities in Definition 1 as stipulating that η indeed produces no effect, and that sequencing is associative.

Monads alone are not enough to produce actual effectful computations, as they only provide primitives to produce trivial effects (via the map η) and to (sequentially) compose them (via binding). For this reason, we endow monads T with (finitary) operations, i.e. with set-indexed families of functions $\mathbf{op}_X : T(X)^n \to T(X)$, where $n \in \mathbb{N}$ is the arity of the operation **op**.

▶ **Example 2.** Here are examples of monads modeling some of the computational effects discussed in Section 1. Further examples, such as global stores and exceptions can be found in, e.g., [49, 70].

1. We model possibly divergent computations using the maybe monad $\mathcal{M}(X) \triangleq X + \{\uparrow\}$. An element in $\mathcal{M}(A)$ is either an element $a \in A$ (meaning that we have a terminating computation returning a), or the element $\uparrow$ (meaning that the computation diverges). Given $a \in A$, the map η_A simply (left) injects a in $\mathcal{M}(A)$, whereas $\gg\!=f$ sends a terminating computation returning a to $f(a)$, and divergence to divergence:

   $$\mathbf{inr}\ (a) \gg\!= f \triangleq f(a); \qquad \mathbf{inr}\ (\uparrow) \gg\!= f \triangleq \mathbf{inr}\ (\uparrow).$$

   As non-termination is an intrinsic feature of complete programming languages, we do not consider explicit operations to produce divergence.
2. We model probabilistic computations using the (discrete) subdistribution monad $\mathcal{D}$. Recall that a discrete subdistribution over a *countable* set X is a function $\mu : X \to [0, 1]$ such that $\sum_x \mu(x) \leq 1$. An element element $\mu \in \mathcal{D}(A)$ gives for any $a \in A$ the probability $\mu(a)$ of returning a. Notice that working with subdistribution we can easily model divergent computations [25]. Given $a \in A$, $\eta_A(a)$ is the Dirac distribution on a (mapping a to 1 and all other elements to 0), whereas for $\mu \in \mathcal{D}(A)$ and $f : A \to \mathcal{D}(B)$ we define $(\mu \gg\!= f)(b) \triangleq \sum_a \mu(a) \cdot f(a)(b)$. Finally, we generate probabilistic computations using a binary fair probabilistic choice operation $\oplus$ thus defined: $(\mu \oplus \nu)(x) \triangleq 0.5 \cdot \mu(x) + 0.5 \cdot \nu(x)$.

3. We model computations with output using the output monad $\mathcal{O}(X) \triangleq O^\infty \times (X + \{\uparrow\})$, where O^∞ is the set of finite and infinite strings over a fixed output alphabet O and $\uparrow$ is a special symbol denoting divergence. An element of $\mathcal{O}(A)$ is either a pair $(o, \mathbf{inl}\ a)$, with $a \in A$, or a pair $(o, \mathbf{inr}\ \uparrow)$. The former case denotes convergence to a outputting o (in which case o is a *finite* string), whereas the former denotes divergence outputting o (in which case o can be either finite or *infinite*). Given $a \in A$, the pair $(\varepsilon, \mathbf{inr}\ a)$ represents the trivial computation that returns a and outputs nothing (ε denotes the empty string). Further, sequential composition of computations is defined using string concatenation as follows, where $f(a) = (o', x)$:

$$(o, \mathbf{inr}\ \uparrow) \gg\!= f \triangleq (o, \mathbf{inr}\ \uparrow); \qquad\qquad (o, \mathbf{inl}\ a) \gg\!= f \triangleq (oo', x).$$

Finally, we produce outputs using (a O-indexed family of) unary operations $\mathbf{print}_c$ mapping (o, x) to (co, x).

4. We model computations with input using the input monad $\mathcal{I}(X) = \mu\alpha.(X + \{\uparrow\}) + \alpha^I$, where I is an input alphabet (for simplicity, we take $I = \{\mathit{true}, \mathit{false}\}$). An element in $\mathcal{I}(A)$ is a binary tree whose leaves are labeled either by elements in A or by the divergent symbol $\uparrow$. The trivial computation returning a is the single leaf labeled by a, whereas given a tree $t \in \mathcal{I}(A)$ and a map $f : A \to \mathcal{I}(B)$, the tree $t \gg\!= f$ is defined by replacing the leaves of t labeled by elements $a \in A$ with $f(a)$. Finally, we consider a binary input operation whereby $\mathbf{read}(t_{\mathit{true}}, t_{\mathit{false}})$ is the tree whose left child is t_{true} and whose right child is t_{false}.

We restrict our analysis to monads T preserving weak pullbacks, and thus preserving injections. As a consequence, if $i : A \hookrightarrow X$ is the subset inclusion map, then $T(i) : T(A) \hookrightarrow T(X)$ is an injection, which can be regarded as monadic inclusion. Intuitively, given an element $\mu \in T(X)$, we think about the *smallest* set $i : A \hookrightarrow X$ such that $\mu \in T(A)$ as the *support* of μ, and denote such a set as $\mathsf{supp}(\mu)$. Of course, in general the support of an element μ may not exist and therefore we restrict our analysis to monads coming with a notion of *countable* support.

▶ **Definition 3.** *We say that a monad is* countable *if for any set X and any element $\mu \in T(X)$, there exists the smallest countable set $i : Y \hookrightarrow X$, denoted by $\mathsf{supp}(\mu)$, such that $\mu \in T(Y)$ (i.e. there exists $\nu \in T(Y)$ such that $\mu = T(i)(\nu)$).*

All monads in Example 2 are countable (for instance, the subdistribution monad $\mathcal{D}$ is countable by definition). An example of a non-countable monad is the powerset monad $\mathcal{P}$. Nonetheless, since we will apply monads to countable sets only (viz. sets of λ-terms and variations thereof), we can regard $\mathcal{P}$ to be countable by taking its countable restriction.

3.1 Algebraic Effects

Following Example 2, let us consider a probabilistic program $e \triangleq E[e_1 \oplus e_2]$, where E is an evaluation context. The operational behaviour of e is to fairly choose $e_i \in \{e_1, e_2\}$, and then execute $E[e_i]$. That is, $E[e_1 \oplus e_2]$ evaluates to $E[e_1]$ (resp. $E[e_2]$) with probability 0.5. But that is exactly the behaviour of $E[e_1] \oplus E[e_2]$, so that we have the program equivalence $E[e_1 \oplus e_2] \equiv E[e_1] \oplus E[e_2]$. It does not take much to realize that a similar equivalence holds for all operations in Example 2. Semantically, operations justifying these equivalences are known as *algebraic operations* [58, 59].

▶ **Definition 4.** *An n-ary (set-indexed family of) operation(s) $\mathbf{op}_X : T(X)^n \to T(X)$ is an* algebraic operation *on T, if for all X, Y, $f : X \to T(Y)$, and $\mu_1, \ldots, \mu_n \in T(X)$, we have:*

$$(\mathbf{op}_X(\mu_1, \ldots, \mu_n)) \gg= f = \mathbf{op}_Y(\mu_1 \gg= f, \ldots, \mu_n \gg= f).$$

Using algebraic operations we can model a large class of effects, including those of Example 2, pure nondeterminism (using the powerset monad and set-theoretic union as binary nondeterminism choice), imperative computations (using the global states monad and operations for reading and updating stores), as well as combinations thereof [35].

3.2 Continuity

Another feature shared by all monads in Example 2 is that they all endow sets $T(X)$ with an ω-complete pointed partial order (ω-cppo, for short) structure making $\gg=$ strict, monotone, and continuous in both arguments, and algebraic operations monotone and continuous in all arguments. This property has been formalized in [21] as Σ-*continuity*.

▶ **Definition 5.** *Let T be a monad and Σ be a set of algebraic operations on T. We say that T is* Σ-continuous *if for any set X, $T(X)$ carries an ω-cppo structure such that $\gg=$ is strict, monotone, and continuous in both arguments, and (algebraic) operations in Σ are monotone and continuous in all arguments.*

▶ **Example 6.**

1. The maybe monad is $\emptyset$-continuous, with $\mathcal{M}(X)$ endowed with the flat order.
2. The subdistribution monad is $\{\oplus\}$-continuous, with subdistributions ordered pointwise (i.e. $\mu \leq \nu$ if and only if $\mu(x) \leq \nu(x)$, for any $x \in X$).
3. Let $\Sigma \triangleq \{\mathbf{print}_c \mid c \in O\}$. Then, the output monad is Σ-continuous, with $\mathcal{O}(A)$ endowed with the order: $(o, x) \sqsubseteq (o', x')$ if and only if either $x = \mathbf{inr} \uparrow$ and $o \sqsubseteq o'$ or $x = \mathbf{inl}\ a = x'$ and $o = o'$.
4. The input monad is $\{\mathbf{read}\}$-continuous with respect to the standard tree ordering.

4 A Linear Calculus with Algebraic Effects

In this section, we introduce a core *linear* call-by-value calculus with *algebraic operations* and *explicit copying* and its *resource-agnostic* operational semantics. The syntax of the calculus is parametric with respect to a signature Σ of operation symbols (notation $\mathbf{op} \in \Sigma$), whereas its dynamics relies on a Σ-continuous monad T, which we assume to be fixed.

4.1 Syntax

Our vehicle calculus is a linear refinement of fine-grain call-by-value [42], which we call $\Lambda^!$. The syntax of $\Lambda^!$ is given by two syntactic classes, *values* (notation $v, w, \ldots$) and *computations* (notation $e, f, \ldots$), which are thus defined:

$$v ::= x \mid \lambda x.e \mid !e$$

$$e ::= a \mid \mathbf{val}\ v \mid vv \mid \mathbf{let}\ x = e\ \mathbf{in}\ e \mid \mathbf{op}(e, \ldots, e) \mid \mathbf{let}\ !a = v\ \mathbf{in}\ e.$$

The letter x denotes a *linear* variable, and thus acts as a placeholder for a *value* which has to be used exactly once. Dually, the letter a denotes a *non-linear* variable, and thus acts as a placeholder for a *computation* which can be used *ad libitum*.

Following the fine-grain discipline, we require computations to be explicitly sequenced by means of the **let** $x = -$ **in** $-$ constructor. The latter comes in two flavors: in the first case, we deal with expressions of the form **let** $x = e$ **in** f, where x is a *linear* variable in f (and thus used once). The intuitive semantics of such an expression is to evaluate e, and then bind the result of the evaluation to x in f. As x is linear in f, the result of e cannot be copied. In the second case, we deal with expressions of the form **let** $!a = v$ **in** f, where a is a *non-linear* variable in f (and thus it can be used as will). As we are going to see, for such an expression to be meaningful, we need v to be a banged computation $!e$. The intuitive semantics of such an expression is thus to "unbang" $!e$, and then bind e to a in f, this way enabling f to copy e at will.

When the distinction between values and computations is not relevant, we generically refer to *terms*, and denote them as $t, s, \ldots$. We adopt standard syntactic conventions as in [5]. In particular, we work with terms modulo renaming of bound variables, and denote by $t[x := v]$ (resp. $t[a := e]$) the result of capture-avoiding substitution of the value v (resp. computation e) for the variable x (resp. a) in t.

4.2 Statics

The syntax of $\Lambda^!$ allows one to write undesired programs, such as programs having runtime errors (e.g. $(!e)v$) and programs that should be forbidden by any reasonable type system (such as $(\mathbf{val}\ !e) \oplus (\mathbf{val}\ \lambda x.f)$). To overcome this problem, we follow [18] and endow $\Lambda^!$ with a simply-typed system with recursive types, using the system in, e.g., [6]. Types are defined by the following grammar:

$$\sigma ::= \mathsf{x} \ \big|\ !\sigma \ \big|\ \sigma \multimap \sigma \ \big|\ \mu\mathsf{x}.\sigma \multimap \sigma \ \big|\ \mu\mathsf{x}.!\sigma$$

where x is a type variable. Types are defined up to equality, as defined in Figure 2, where $\sigma[\tau/\mathsf{x}]$ denotes the substitution of τ for all the (free) occurrences of x in σ. In the third rule in Figure 2, we require ρ to be productive in x, meaning that each free occurrence of x in ρ is under the scope of either $\multimap$ or $!$.

$$\frac{}{\mu\mathsf{x}.\sigma \multimap \tau = \sigma[\mu\mathsf{x}.\sigma \multimap \tau/\mathsf{x}] \multimap \tau[\mu\mathsf{x}.\sigma \multimap \tau/\mathsf{x}]} \qquad \frac{}{\mu\mathsf{x}.!\sigma = !\sigma[\mu\mathsf{x}.!\sigma/\mathsf{x}]} \qquad \frac{\sigma = \rho[\sigma/\mathsf{x}] \quad \tau = \rho[\tau/\mathsf{x}]}{\sigma = \tau}$$

Figure 2 Type Equality.

In order to define the collection of well-typed expressions, we consider sequents $\Sigma \mid \Omega \vdash^{\mathsf{v}} v : \sigma$ and $\Sigma \mid \Omega \vdash^{\Lambda} e : \sigma$, where Ω is a linear environment, i.e. a set without repetitions of the form $x_1 : \sigma_1, \ldots, x_n : \sigma_n$, and Σ is a *non-linear* environment, i.e. a set without repetitions of the form $a_1 : \tau_1, \ldots, a_n : \tau_n$. Rules for derivable sequents are given in Figure 3. We write $\mathcal{V}_\sigma$ and Λ_σ for the collection of closed values and computations of type σ, respectively. We write $\mathcal{V}$ and Λ when types are not relevant.

▶ Remark 7 (Notational Convention). In order to facilitate the communication of the main ideas behind this work and to lighten the (quite heavy) notation we will employ in the next sections, we avoid to mention types (and ignore them in the notation) whenever possible. Nonetheless, the reader should keep in mind that from now on we work with typable terms only. We refer to such an assumption as the *type assumption*.

$$\frac{}{\Sigma \mid x : \sigma \vdash^{\mathsf{v}} x : \sigma} \quad \frac{}{a : \sigma, \Sigma \mid \emptyset \vdash^{\Lambda} a : \sigma} \quad \frac{\Sigma \mid x : \sigma, \Omega \vdash^{\Lambda} e : \tau}{\Sigma \mid \Omega \vdash^{\mathsf{v}} \lambda x.e : \sigma \multimap \tau} \quad \frac{\Sigma \mid \Omega \vdash^{\mathsf{v}} v : \sigma}{\Sigma \mid \Omega \vdash^{\Lambda} \mathbf{val}\ v : \sigma}$$

$$\frac{\Sigma \mid \Omega \vdash^{\mathsf{v}} v : \sigma \multimap \tau \quad \Sigma \mid \Omega' \vdash^{\mathsf{v}} w : \sigma}{\Sigma \mid \Omega, \Omega' \vdash^{\Lambda} vw : \tau} \quad \frac{\Sigma \mid \emptyset \vdash^{\Lambda} e : \sigma}{\Sigma \mid \emptyset \vdash^{\mathsf{v}} !e : !\sigma} \quad \frac{\Sigma \mid \Omega \vdash^{\mathsf{v}} v : !\sigma \quad \Sigma, a : \sigma \mid \Omega' \vdash^{\Lambda} e : \tau}{\Sigma \mid \Omega, \Omega' \vdash^{\Lambda} \mathbf{let}\ !a = v\ \mathbf{in}\ e : \tau}$$

$$\frac{\Sigma \mid \Omega \vdash^{\Lambda} e : \sigma \quad \Sigma \mid \Omega', x : \sigma \vdash^{\Lambda} f : \tau}{\Sigma \mid \Omega, \Omega' \vdash^{\Lambda} \mathbf{let}\ x = e\ \mathbf{in}\ f : \tau} \quad \frac{\Sigma \mid \Omega \vdash^{\Lambda} e_1 : \sigma \quad \dots \quad \Sigma \mid \Omega \vdash^{\Lambda} e_n : \sigma}{\Sigma \mid \Omega \vdash^{\Lambda} \mathbf{op}(e_1, \dots, e_n) : \sigma}$$

Figure 3 Statics of $\Lambda^!$.

4.3 Dynamics

The dynamic semantics of $\Lambda^!$ associates to any *closed computation* e of type σ a monadic element in $T(\mathcal{V}_\sigma)$. The dynamics of $\Lambda^!$ is defined in Figure 4 by means of an $\mathbb{N}$-indexed family of evaluation functions mapping a *closed* computation $e \in \Lambda_\sigma$ to an element $[\![e]\!]^{\Lambda}_k \in T(\mathcal{V}_\sigma)$, where we stipulate $[\![e]\!]^{\Lambda}_0 \triangleq \bot$. Since $([\![e]\!]^{\Lambda}_k)_{k\geq 0}$ forms an ω-chain in $T(\mathcal{V})$, we define $[\![e]\!]^{\Lambda} \triangleq \bigsqcup_{k\geq 0}[\![e]\!]^{\Lambda}_k$. Notice that thanks to the type assumption, we ignore programs causing runtime errors. Finally, we lift $[\![-]\!]^{\Lambda}$ to monadic computations, i.e. to elements $\xi \in T(\Lambda)$ by setting $[\![\xi]\!]^{\Lambda^*} \triangleq \xi \gg= (e \to [\![e]\!]^{\Lambda})$ (and similarity for $[\![-]\!]^{\Lambda}_k$).

$$\begin{aligned}
[\![\mathbf{val}\ v]\!]^{\Lambda}_{k+1} &\triangleq \eta(v)\\
[\![(\lambda x.e)v]\!]^{\Lambda}_{k+1} &\triangleq [\![e[x := v]]\!]^{\Lambda}_k\\
[\![\mathbf{let}\ x = e\ \mathbf{in}\ f]\!]^{\Lambda}_{k+1} &\triangleq [\![e]\!]^{\Lambda}_k \gg= (v \to [\![f[x := v]]\!]^{\Lambda}_k)\\
[\![\mathbf{let}\ !a = !e\ \mathbf{in}\ f]\!]^{\Lambda}_{k+1} &\triangleq [\![f[a := e]]\!]^{\Lambda}_k\\
[\![\mathbf{op}(e_1, \dots, e_n)]\!]^{\Lambda}_{k+1} &\triangleq [\![\mathbf{op}]\!]([\![e_1]\!]^{\Lambda}_k, \dots, [\![e_n]\!]^{\Lambda}_k)
\end{aligned}$$

Figure 4 Operational Semantics of $\Lambda^!$.

4.4 Observational Equivalence

In order to compare $\Lambda^!$-terms, we introduce the notion of *contextual equivalence* [51]. To do so, we follow [67, 22] and postulate that once an observer executes a program, she can only observe the effects produced by the evaluation of the program. For instance, in a pure (resp. probabilistic) calculus one observes pure (resp. the probability of) convergence. Following this postulate, we define an observation function $\mathtt{obs}^{\Lambda^*} : T(\mathcal{V}) \to T(1)$ as $T(!_{\mathcal{V}})$, where $1 = \{*\}$ is the one-element set and $!_{\mathcal{V}} : \mathcal{V} \to 1$ is the terminal arrow. As a consequence, we see that $\mathtt{obs}^{\Lambda^*}$ is strict and continuous, so that we have, e.g., $\mathtt{obs}^{\Lambda^*}(\bigsqcup_k \xi_k) = \bigsqcup_k \mathtt{obs}^{\Lambda^*}(\xi_k)$.

▶ **Example 8.** Notice that $T(1)$ indeed describes the observations one usually works with in concrete calculi. For instance, $\mathcal{D}(1) \cong [0, 1]$, so that $\mathtt{obs}^{\Lambda^*}([\![e]\!])$ gives the probability of convergence of e, and $\mathcal{M}(1) \cong \{\bot, \top\}$, so that $\mathtt{obs}^{\Lambda^*}([\![e]\!]) = \top$ if and only if e converges.

In order to define contextual equivalence, we need to introduce the notion of a $\Lambda^!$-context. The latter is simply a $\Lambda^!$-term with a single *linear* hole $[-]$ acting as a placeholder for a computation (we regard a value v as the computation **val** v). We do not give an explicit definition of contexts, the latter being standard.

▶ **Definition 9.** *Define contextual equivalence* $\equiv^{\mathtt{ctx}}$ *as follows:*

$$v \equiv^{\mathtt{ctx}} w \iff \mathbf{val}\ v \equiv^{\mathtt{ctx}} \mathbf{val}\ w \qquad e \equiv^{\mathtt{ctx}} f \iff \forall C.\ \mathtt{obs}^{\Lambda^*}\ [\![C[e]]\!] = \mathtt{obs}^{\Lambda^*}\ [\![C[f]]\!].$$

The universal quantification over contexts guarantees $\equiv^{\mathtt{ctx}}$ to be a congruence relation. However, it also makes $\equiv^{\mathtt{ctx}}$ difficult to be used in practice. We overcome this deficiency by characterising contextual equivalence as a suitable notion of trace equivalence.

5 Resource-Sensitive Semantics and Program Equivalence

The operational semantics of Section 4.3 is *resource-agnostic*, meaning that linearity *de facto* plays no role in the definition of the dynamics of a program. To overcome this deficiency, we endow $\Lambda^!$ with a resource-sensitive operational semantics: we give the latter by means of a suitable transition systems, which we dub resource transition systems. *Resource transition systems* (RTSs, for short) provide an operational semantics for $\Lambda^!$-programs accounting for both their intensional and extensional behaviour. Those are defined as first-order transition systems in the spirit of [44], and generalise the Markov chains of [18].

5.1 Auxiliary Notions

In order to properly handle resources, it is useful to introduce some notation on sequences. Let S, S' be sequences over objects $s_1, s_2, \ldots$. Unless ambiguous, we denote the concatenation of S and S' as S, S'. Moreover, for $S = s_1, \ldots, s_k$ we denote by $|S| = k$ the length of S, and write $S[s]_i$, with $i \in \{1, \ldots, k+1\}$, for the sequence obtained by inserting s in S at position i, i.e. the sequence $s_1, \ldots, s_{i-1}, s, s_i, \ldots, s_k$ of length $k+1$. Given a sequence $S = s_1, \ldots, s_k$, we will form new sequences out of it by taking elements in S at given positions. If $\bar{c} = c_1, \ldots, c_n$ is a sequence with elements in $\{1, \ldots, k\}$ without repetitions, then we write $S_{\bar{c}}$ for the sequence $s_{c_1}, \ldots, s_{c_n}$, and $S \ominus \bar{c}$ for the sequence obtained from S by removing elements in positions $c_1, \ldots, c_n$. In order to preserve the order of S, we often consider sequences $\bar{c} = (c_1 < \cdots < c_n)$ with $c_i \in \{1, \ldots, k\}$. We call such sequences valid for S (although we should say valid for $|S|$).

System $\mathcal{K}$

The resource-sensitive operational semantics of $\Lambda^!$ is given by the RTS $\mathcal{K}$. Following [44], $\mathcal{K}$-states are defined as *configurations* $(\Gamma; \Theta)$, i.e. pairs of sequences of terms, where Γ is a (finite) sequence of (closed) computations and Θ is a (finite) sequence of (closed) terms in which only the last one need not be a value. To facilitate our analysis, we write $(\Gamma; \Delta; e)$ if $\Theta = \Delta, e$, with Δ finite sequence of closed values and $e \in \Lambda$. Otherwise, we write $(\Gamma; \Delta)$, with Δ as above.

In a configuration $(\Gamma; \Delta; e)$ (and similarly in $(\Gamma; \Delta)$), Γ represents the non-linear resources available, which are (closed) computations: the environment can freely duplicate and evaluate them, as well as use them *ad libitum* to build arguments to be passed as input to other programs. Once a resource in Γ has been used, it *remains* in Γ, this way reflecting its non-linear nature. Dually, Δ represents the linear resources available, which are closed values. Values in Δ being closed, they are either abstractions or banged computations. In the latter case, the environment can take a value $!e$, unbang it, and put e in Γ. In the former case, the environment can pass to a value $\lambda x.f$ an input argument made out of a context C (provided by the very environment) using values and computations in Γ, Δ. Since resources in Δ are linear, once they are used by C, they must be removed from Δ. Finally, the program e is the tested program. The environment can only evaluate it, possibly producing effects and values (linear resources). Once a linear resource v has been produced, it is put in Δ.

The calculus $\Lambda^!$ being typed, it is convenient to extend the notion of a type to configurations by defining a configuration type (notation $\alpha, \beta, \dots$) as a pair of sequences $(\sigma_1, \dots, \sigma_n; \tau_1, \dots, \tau_m)$ of ordinary types. We say that a configuration $K = (\Gamma; \Theta)$ has type $\alpha = (\sigma_1, \dots, \sigma_n; \tau_1, \dots, \tau_m)$ (and write $\vdash K : \alpha$) if each computation e_i at position i in Γ has type σ_i, and each term t_i at position i in Θ has type τ_i.

Notice that configuration types almost completely describe the structure of configurations. However, they do not allow one to see whether the last argument in the second component Θ of a configuration $(\Gamma; \Theta)$ is a value (so that the type will be inhabitated by configurations of the form $(\Gamma; \Delta)$) or a computation (so that the type will be inhabitated by configurations of the form $(\Gamma; \Delta; e)$). To avoid this issue, we add a special label to the last type τ_m of the second component of a configuration type, this way specifying whether τ_m refers to a value or to a computation.

We denote by $\mathcal{C}_\alpha$ the collection of configurations of type α. Notice that if $K, L \in \mathcal{C}_\alpha$, then they have the same structure. In particular, terms in K and L at the same position have the same type and belong to the same syntactic class. As usual, following the type assumption, we will omit configuration types whenever possible.

States of $\mathcal{K}$ are thus (typable) configurations, whereas its dynamics is based on three kind of actions: *evaluation*, *duplication*, and *resource-based application*, which are *extensional*, *intensional*, and *mixed extensional-intensional* actions, respectively. Formally, we consider transitions from (typable) configurations, i.e. elements in $\bigcup_\alpha \mathcal{C}_\alpha$ to monadic configurations in $\bigcup_\alpha T(\mathcal{C}_\alpha)$, i.e. monadic configurations κ such that all configurations in the support of κ have the same type. This ensures that all configurations in $\mathsf{supp}(\kappa)$ can make the same actions. As usual, such a property follows by typing, hence by the type assumption. We now spell out the main ideas behind the dynamics of $\mathcal{K}$.

- Given a configuration $(\Gamma; \Delta; e)$, the environment simply evaluates e. That is, we have the transition:

$$(\Gamma; \Delta; e) \xrightarrow{eval} [\![e]\!] \gg\!= (v \to \eta(\Gamma; \Delta, v)).$$

- Given a configuration of the form $(\Gamma; \Delta[!e]_l)$, the environment adds e to the non-linear environment, and removes $!e$ from the linear one. We thus have the transition:

$$(\Gamma; \Delta[!e]_l) \xrightarrow{?_l} \eta(\Gamma, e; \Delta).$$

- In a configuration of the form $(\Gamma[e]_l; \Delta)$, the environment has the non-linear resource e at its disposal, which can be duplicated (and eventually evaluated via an *eval* action). We model such a behaviour as the following transition (notice that e is not removed from $\Gamma[e]_l$):

$$(\Gamma[e]_l; \Delta) \xrightarrow{!_l} \eta(\Gamma[e]_l; \Delta; e).$$

- For the last action, namely resource-based application, we consider open terms as playing the role of contexts. An open term is simply a term $\Sigma \mid \Omega \vdash t$. We refer to an open term $a_1, \dots, a_n \mid x_1, \dots, x_m \vdash t$ as a (n, m)-(value/computation) context, depending on whether t is a value or a computation. Given sequences $\Gamma = e_1, \dots, e_n$, $\Delta = v_1, \dots, v_m$, we write $t[\Gamma, \Delta]$ for the substitution of variables in t with the corresponding elements in Γ, Δ. As usual, following the type-assumption we assume types of variables to match types of the substituted terms. Given sequences $\bar{\imath}$, $\bar{\jmath}$ of length n, m valid for Γ, Δ, respectively, we can build a new (closed) term out of Γ, Δ and a (n,m)-context t as $t[\Gamma_{\bar{\imath}}, \Delta_{\bar{\jmath}}]$. Since resources in Δ are linear, the construction of $t[\Gamma_{\bar{\imath}}, \Delta_{\bar{\jmath}}]$ affects Δ, this way leaving only resources $\Delta \ominus \bar{\jmath}$ available. We formalise this behaviour as the transition:

$$\frac{t\ (n,m)\text{-value context} \quad |\bar{\imath}| = n, |\bar{\jmath}| = m \quad \bar{\imath}, \bar{\jmath} \text{ valid for } \Gamma, \Delta}{(\Gamma; \Delta[\lambda x.f]_l) \xrightarrow{(\bar{\imath},\bar{\jmath},l,t)} \eta(\Gamma; \Delta \ominus \bar{\jmath}; f[x := t[\Gamma_{\bar{\imath}}, \Delta_{\bar{\jmath}}]])}$$

▶ **Definition 10.** *System $\mathcal{K}$ is the (resource) transition system having typable configurations as states, actions*

$$\{eval, ?_l, !_l, , (\bar{\imath}, \bar{\jmath}, l, t), \alpha \mid l \in \mathbb{N}, t\ (n,m)\text{-}value\ context, |\bar{\imath}| = n, |\bar{\jmath}| = m\}$$

where α ranges over configuration types, and dynamics defined by the transition rules in Figure 5, where we employ the notation of previous discussion.

$$(\Gamma; \Delta; e) \xrightarrow{eval} [\![e]\!] \gg\!= v \to \eta(\Gamma; \Delta, v) \qquad (\Gamma; \Delta[!e]_l) \xrightarrow{?_l} \eta(\Gamma, e; \Delta).$$

$$(\Gamma[e]_l; \Delta) \xrightarrow{!_l} \eta(\Gamma[e]_l; \Delta; e) \qquad (\Gamma; \Delta[\lambda x.f]_l) \xrightarrow{(\bar{\imath},\bar{\jmath},l,t)} \eta(\Gamma; \Delta \ominus \bar{\jmath}; f[x := t[\Gamma_{\bar{\imath}}, \Delta_{\bar{\jmath}}]])$$

Figure 5 Transition rules for $\mathcal{K}$.

▶ Remark 11. Notice that given $K \in \mathcal{C}_\alpha$, K can always make a α-transition, this way making its type visible. Additionally, we see that the transition structure of $\mathcal{K}$ is *type-driven*. That is, given a configuration $K \in \mathcal{C}_\alpha$ and a $\mathcal{K}$-action ℓ, α and ℓ alone determine whether K can make an ℓ-transition. Moreover, if that is the case, then there is a unique κ such that $K \xrightarrow{\ell} \kappa$. Besides, $\kappa \in T(\mathcal{C}_\beta)$ for some configuration type β which is *uniquely* determined by ℓ and α. That is, there is a *partial* function b from configuration types and actions such that if $\mathsf{b}(\alpha, \ell)$ is defined and $K \in \mathcal{C}_\alpha$, then $K \xrightarrow{\ell} \kappa$ with $\kappa \in T(\mathcal{C}_{\mathsf{b}(\alpha,\ell)})$. From now on, we write $\mathsf{b}(\alpha, \ell) = \beta$ to mean that $\mathsf{b}(\alpha, \ell)$ is defined and equal β. As a consequence, we have the rule:

$$K \in \mathcal{C}_\alpha \wedge \mathsf{b}(\alpha, \ell) = \beta \implies \exists! \kappa \in T(\mathcal{C}_\beta).\ K \xrightarrow{\ell} \kappa.$$

Having defined system $\mathcal{K}$, there are at least two natural ways to compare its states. The first one is by means of *bisimilarity*, which can be defined in a standard way [21]. Unfortunately, bisimilarity being sensitive to branching, it is bound not to work well for our purposes, as already extensively discussed. The second natural way to compare $\mathcal{K}$-states is by means of *trace equivalence* which, contrary to bisimilarity, is not sensitive to branching, and thus qualifies as a suitable candidate program equivalence for our purposes.

▶ **Definition 12.** *A $\mathcal{K}$-trace (just trace) is a finite sequence of $\mathcal{K}$-actions. That is, a trace* $\mathtt{t}$ *is either the empty sequence (denoted by ε), or a sequence of the form $\ell \cdot \mathtt{u}$, where ℓ is a $\mathcal{K}$-action and* $\mathtt{u}$ *a trace.*

We are interested in observing the behaviour of $\mathcal{K}$-states on those traces that are coherent with their type. Therefore, given a $\mathcal{K}$-state K, we define the set $Tr(K)$ of its traces by stipulating that $\varepsilon \in Tr(K)$, for any K, and that $\ell \cdot \mathtt{u} \in Tr(K)$ whenever $K \xrightarrow{\ell} \kappa$, for some monadic configuration κ, and $\mathtt{u} \in Tr(L)$, for any $L \in \mathsf{supp}(\kappa)$. Notice that the latter clause is meaningful, since $Tr(K)$ is actually determined by the type of K (rather than by K itself), and if $K \xrightarrow{\ell} \kappa$, then all configurations in the support of κ have the same type.

Now, given a $\mathcal{K}$-state K, and a trace $\mathtt{t} \in Tr(K)$, the observable behaviour of K on $\mathtt{t}$ is the element in $T(1)$ computed using the map $\mathtt{st}$ thus defined:

$$\mathtt{st}(K, \varepsilon) \triangleq \eta(*); \qquad \mathtt{st}(K, \ell \cdot \mathtt{u}) \triangleq \kappa \gg\!= (L \to \mathtt{st}(L, \mathtt{u})) \text{ where } K \xrightarrow{\ell} \kappa.$$

▶ **Example 13.** Let us consider the (sub)distribution monad $\mathcal{D}$, and let K be a configuration. Recall that $\mathcal{D}(1) \cong [0,1]$, and notice that $\mathtt{st}(K, \varepsilon) = 1$. Suppose now $K \xrightarrow{eval} \sum_{i \in n} p_i \cdot L_i$. Then, we see that $\mathtt{st}(K, eval \cdot \mathtt{u}) = \sum_{i \in n} p_i \cdot \mathtt{st}(L_i, \mathtt{u}) \in [0,1]$, meaning that $\mathtt{st}(K, \mathtt{t})$ gives the probability that K passes the trace $\mathtt{t}$.

▶ **Definition 14.** *The relation $\simeq^{\mathcal{K}}_{\mathrm{Tr}}$ on $\mathcal{K}$-states is thus defined:*

$$K \simeq^{\mathcal{K}}_{\mathrm{Tr}} L \iff Tr(K) = Tr(L) \wedge \forall \mathtt{t} \in Tr(K).\ \mathtt{st}(K, \mathtt{t}) = \mathtt{st}(L, \mathtt{t})$$

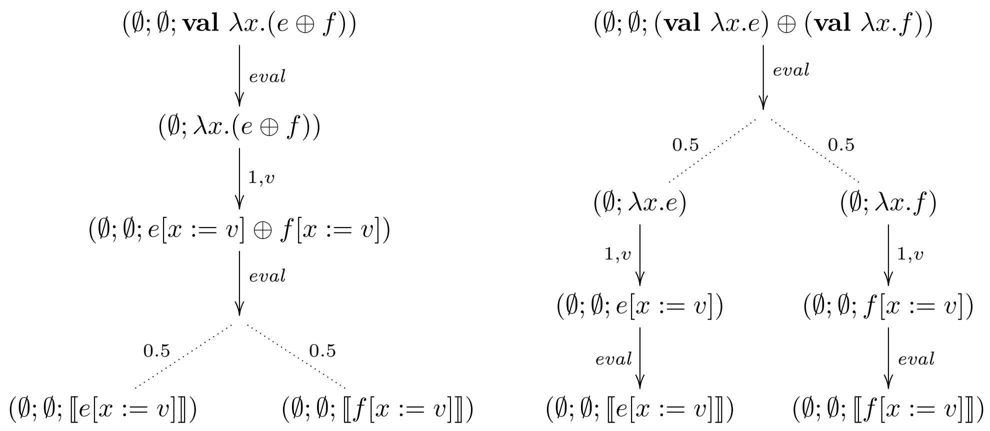
We extend the action of $\simeq^{\mathcal{K}}_{\mathrm{Tr}}$ to $\Lambda^{!}$-terms by regarding a computation e as the configuration $(\emptyset; \emptyset; e)$, and a value v as the computation $\mathbf{val}\ v$. We denote the resulting notion $\simeq^{\Lambda}_{\mathrm{Tr}}$.

Having added $\simeq^{\mathcal{K}}_{\mathrm{Tr}}$ to our arsenal of operational techniques, it is time to investigate its structural properties and its relationship with contextual equivalence. Before doing so, however, we take a fresh look at our running example.

▶ **Example 15.** Let us use the machinery developed so far to review our introductory examples. First, we show

$$\mathbf{val}\ \lambda x.(e \oplus f) \simeq^{\Lambda}_{\mathrm{Tr}} (\mathbf{val}\ \lambda x.e) \oplus (\mathbf{val}\ \lambda x.f).$$

Let us call g the former program, and h the latter. To see that $g \simeq^{\Lambda}_{\mathrm{Tr}} h$, we simply observe that $Tr(\emptyset; \emptyset; g) = Tr(\emptyset; \emptyset; h)$ and that for any $\mathtt{t} \in Tr(g)$, the probability that $(\emptyset; \emptyset; g)$ passes $\mathtt{t}$ coincides with the one of $(\emptyset; \emptyset; h)$. All of this can be easily observed by inspecting the following transition systems.



In light of Theorem 17, we can then conclude $g \equiv^{\mathrm{ctx}} h$. Next, we prove that such an equivalence is only linear: $\mathbf{val}\ !(e \oplus f) \not\equiv^{\mathrm{ctx}} (\mathbf{val}\ !e) \oplus (\mathbf{val}\ !f)$. For that, it is sufficient to instantiate e and f as the identity program $\mathbf{val}\ (\lambda x.\mathbf{val}\ x)$ and the purely divergent program Ω, respectively, and to take the context C defined as $\mathbf{let}\ x = [-]\ \mathbf{in}\ \mathbf{let}\ !a = x\ \mathbf{in}\ (a; a; \mathbf{val}\ v)$, where v is closed value, and $e; f$ denotes trivial sequencing. Indeed, what C does is to evaluate its input and then test the result thus obtained *twice*.

5.2 Full Abstraction of Trace Equivalence

In this section, we outline the proof of *full abstraction* of trace equivalence for contextual equivalence. Our proof of full abstraction builds upon the technique given by Deng and Zhang [27] and Crubillé and Dal Lago [18] to prove similar full abstraction results for trace equivalences and metrics, respectively. Due to the large amount of technicalities, the full proof of full abstraction of trace equivalence goes beyond the scope of this paper, so that here we only outline its main points (see [20] for details). Let us begin by showing that trace equivalence is *sound* for contextual equivalence.

▶ **Proposition 16.** $\simeq^{\Lambda}_{\mathtt{Tr}} \subseteq \equiv^{\mathtt{ctx}}$.

To prove Proposition 16, we have to show that if $e \simeq^{\Lambda}_{\mathtt{Tr}} f$, then we have $\mathtt{obs}^{\Lambda^*}[\![C[e]]\!]^{\Lambda} = \mathtt{obs}^{\Lambda^*}[\![C[e]]\!]^{\Lambda}$, for any context C. Our proof proceeds by progressively building systems with increasingly more complex state spaces, but with finer dynamics. We summarise our strategy in the following diagram.

$$\begin{tikzcd}
\Lambda \arrow[rrr, "{C[-]}"] \arrow[d, hook] & & & \Lambda^* \arrow[r, "\mathtt{obs}^{\Lambda^*}"] & T1 \\
\mathcal{K} \arrow[r, hook] & \mathcal{K}^* \arrow[r, "{C[-]}"] & \mathcal{F} \arrow[r, hook] & \mathcal{F}^* \arrow[u, "\mathtt{push}"] \arrow[ur, "\mathtt{obs}^{\mathcal{F}^*}"'] &
\end{tikzcd}$$

Since $\simeq^{\Lambda}_{\mathtt{Tr}}$ is defined in terms of $\simeq^{\mathcal{K}}_{\mathtt{Tr}}$, we consider configurations – $\mathcal{K}$-states – and contexts for them, where a context for a $\mathcal{K}$-state K is just a standard multiple-holes context whose holes have to be filled with with terms in K. The first step of our strategy is the *determinization* of $\mathcal{K}$. This is achieved by lifting the state space of $\mathcal{K}$ from configurations to monadic configurations. The dynamics of $\mathcal{K}$ is then lifted relying on the (strong) monad structure of T in a standard way [22]. We call the resulting system $\mathcal{K}^*$. The advantage of working with $\mathcal{K}^*$ is that $\mathcal{K}^*$-bisimilarity and $\mathcal{K}^*$-trace equivalence coincide, $\mathcal{K}^*$ being deterministic. In general, most of the transition systems we rely on can be ultimately described as systems $\mathcal{S} = (X, \delta)$ made of a state space X and a dynamics $\delta : X \to T(X)^A$, for some set A of actions. The determinization of $\mathcal{S}$, which we usually denote by $\mathcal{S}^*$, has $T(X)$ as state space and dynamics $\delta^* : T(X) \to T(X)^A$ defined as the strong Kleisli extension of δ (modulo (un)currying).

Having determinized $\mathcal{K}$, we reach a situation where we have to study the computational behaviour of a monadic configuration κ – i.e. a $\mathcal{K}^*$-state – and a context C for the configurations in the support of κ. To do so, we build a further system, called $\mathcal{F}$, whose states are pairs $C : \kappa$ made of a monadic configuration κ and a context C for it. The dynamics of $\mathcal{F}$ is given by an evaluation function which, when applied to a $\mathcal{F}$-state $C : \kappa$, gives the same result of evaluating the *monadic computation* $C[\kappa] \in T(\Lambda)$, where $C[\kappa] = \kappa \gg\!= (K \to \eta(C[K]))$. Such a dynamics explicitly separates the computational steps acting on C only from those making C and κ interact. This feature is crucial, as it shows that any interaction between C and κ corresponds to a $\mathcal{K}^*$-action, so that equivalent $\mathcal{K}^*$-states will have the same $\mathcal{F}$-dynamics when paired with the same context. That gives us a finer analysis of the computational behaviour of the compound monadic computation $C[\kappa]$, and ultimately of a compound computation $C[e]$. As we did for $\mathcal{K}$, it is actually convenient to determinise $\mathcal{F}$. We call the resulting system $\mathcal{F}^*$. Finally, from $\mathcal{F}^*$ we can come back to $T(\Lambda)$ using the map $\mathtt{push} : \mathcal{F}^* \to T(\Lambda)$ defined by $\mathtt{push}(\xi) \triangleq \xi \gg\!= (C : \kappa \mapsto C[\kappa])$. We summarize the systems introduced so far in the following table.

System	$\mathcal{K}$	$\mathcal{K}^*$	$\mathcal{F}$	$\mathcal{F}^*$
States	Configurations K	Monadic configurations κ	Pairs $C : \kappa$	Monadic pairs
Dynamics	Definition 10	Kleisli lifting of $\mathcal{K}$	$[\![C[\kappa]]\!]^*$	Kleisli lifting of $\mathcal{F}$

What remains to be clarified is how relations between computations can be transformed into relations on the aforementioned systems. The answer to this question is given by the following *lax*[1] commutative diagram:

$$\begin{tikzcd}
\Lambda \arrow[r, hook] \arrow[d, "\simeq^{\Lambda}_{\mathrm{Tr}}"'] & \mathcal{K} \arrow[r, hook] \arrow[d, "\simeq^{\mathcal{K}}_{\mathrm{Tr}}"'] & \mathcal{K}^* \arrow[r, "C[-]"] \arrow[d, "\simeq^{\mathcal{K}^*}_{\mathrm{Tr}}"'] & \mathcal{F} \arrow[r, hook] \arrow[d, "\mathtt{C}(\simeq^{\mathcal{K}^*}_{\mathrm{Tr}})"'] & \mathcal{F}^* \arrow[r, "\mathtt{obs}^{\mathcal{F}^*}"] \arrow[d, "\mathtt{BC}(\simeq^{\mathcal{K}^*}_{\mathrm{Tr}})"'] & T1 \arrow[d, "="'] \\
\Lambda \arrow[r, hook] & \mathcal{K} \arrow[r, hook] & \mathcal{K}^* \arrow[r, "C[-]"'] & \mathcal{F} \arrow[r, hook] & \mathcal{F}^* \arrow[r, "\mathtt{obs}^{\mathcal{F}^*}"'] & T1
\end{tikzcd}$$

Here, $\mathtt{C}(R)$ denotes the contextual closure of R, whereas $\mathtt{B}(R)$ is the Barr extension of R [7, 38]. Finally, the map $\mathtt{obs}^{\mathcal{F}^*}$ is obtained postcomposing the observation map `obs` with `push`. Let us now move to full abstraction.

▶ **Theorem 17.** $\equiv^{\mathsf{ctx}} = \simeq^{\Lambda}_{\mathrm{Tr}}$.

To prove Theorem 17 it is sufficient to show $\equiv^{\mathsf{ctx}} \subseteq \simeq^{\Lambda}_{\mathrm{Tr}}$. The latter is proved by noticing that any $\mathcal{K}$-action can be encoded as a context. The encoding of $\mathcal{K}$-actions as contexts is essentially the same one of the one given by Crubillé and Dal Lago [18].

6 Conclusion and Future Work

In this paper, we have introduced resource transition systems as an operational account of both intensional and extensional behaviours of linear effectful programs with explicit copying. On top of resource transition systems, we have defined trace equivalence and showed that the latter is fully abstract for contextual equivalence.

Although the present paper focuses on linearity (and effects), the authors believe that resource transition systems can be extended to deal with finer notions of context dependence such as *structural coeffects* [53, 29, 14, 52]. To do so, one should modify resource transition systems by considering sequences of terms indexed by elements of a resource algebra (the latter being a preordered semiring), and let transitions update resources. Thus, for instance, from a sequence $(\Gamma, \langle e \rangle_{r+1}, \Delta)$, meaning that e is available according to the resource $r+1$, we have a transition to $(\Gamma, \langle e \rangle_r, \Delta; e)$. The authors also believe that resource transition systems can be used to generalise Crubillé and Dal Lago probabilistic program metric to arbitrary algebraic effects. To do so, one would simply replace ordinary relations with relations taking values over quantales [30, 31]. In the same direction, it would be interesting to study whether resource transition systems give fully abstract equivalences in presence of *continuous*, rather than discrete, probability (applicative bisimilarity, for instance, has been proved to be sound but not fully abstract on higher-order calculi with sampling from continuous distributions [39]).

Finally, as a long term future work, the authors would like to study whether the ideas presented in this paper can be adapted to deal with quantum languages [64, 65], where the interaction between linearity and effects plays a central role. In fact, although we have not discussed tensor product types (which play a crucial role in a quantum setting), it is not hard to see that resource transition systems can be extended to deal with such types [17].

[1] Each square gives a set-theoretic inclusion. For instance, the leftmost square states that $\simeq^{\Lambda}_{\mathrm{Tr}} \subseteq \simeq^{\mathcal{K}}_{\mathrm{Tr}}$.

6.1 Related Work

This is not the first work on operationally-based notions of program equivalence for linear calculi. In particular, notions of equivalences have been defined by means of logical relations by Bierman, Pitts, and Russo [11], of applicative bisimilarity by Bierman [10] and Crole[2] [15], of trace equivalence by Deng and Zhang [27, 26], as well as of a number of possible worlds-indexed equivalences (e.g. [2, 37]). As already remarked, one of the advantages of resource transition systems (and their associated trace equivalence) compared, e.g., with logical relations, is that they they provide a *first-order* account of program equality.

Among first-order notions of program equivalence, Bierman's applicative bisimilarity plays a prominent role. The latter is a lightweight extensional equivalence extending Abramsky's applicative bisimilarity [1] to a *pure* linear λ-calculus with explicit copying. Bierman's applicative bisimilarity can be readily extended to calculi with algebraic effects along the lines of [21], this way obtaining a notion of equivalence invalidating (!-dist). However, such a notion of bisimilarity stipulates that two programs $!e$ and $!f$ are bisimilar if and only if e and f are, this way making bisimilarity insensitive to linearity, and thus invalidating (λ-dist) as well.[3]

Deng and Zhang's linear trace equivalence has been designed to study the interaction of linearity and (both pure and probabilistic) nondeterminism. The latter equivalence, in fact, validates (λ-dist). However, linear trace equivalence does not deal with (explicit) copying: even worse, natural extensions of such notions to languages with copying result in equivalences validating (!-dist). Crubillé and Dal Lago [18] solved that problem by introducing a tuple-based applicative bisimilarity for a calculus with probabilistic nondeterminism and explicit copying. Our notion of a resource transition system can be seen as a generalisation of the Markov chain underlying tuple based applicative bisimilarity to arbitrary algebraic effects.

References

1 Samson Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–117. Addison Wesley, 1990.

2 Amal Ahmed, Matthew Fluet, and Greg Morrisett. L^3: A linear language with locations. *Fundam. Informaticae*, 77(4):397–449, 2007.

3 Andrew W. Appel and Daddiv A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001.

4 Robert Atkey. Syntax and semantics of quantitative type theory. In *Proc. of LICS 2018*, pages 56–65, 2018.

5 Hendrik P. Barendregt. *The lambda calculus: its syntax and semantics.* Studies in logic and the foundations of mathematics. North-Holland, 1984.

6 Hendrik P. Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types.* Perspectives in logic. Cambridge University Press, 2013.

7 Michael Barr. Relational algebras. *Lect. Notes Math.*, 137:39–55, 1970.

8 Nick Benton and Philip Wadler. Linear logic, monads and the lambda calculus. In *Proc. of LICS 1996*, pages 420–431, 1996.

9 Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *PACMPL*, 2(POPL):5:1–5:29, 2018.

[2] Crole's applicative bisimilarity, however, does not deal with copying.
[3] Besides, notice that bisimilarity being sensitive to branching, it naturally invalidates (λ-dist).

10 Gavin M. Bierman. Program equivalence in a linear functional language. *J. Funct. Program.*, 10(2):167–190, 2000.

11 Gavin M. Bierman, Andrew M. Pitts, and Claudio V. Russo. Operational properties of lily, a polymorphic linear lambda calculus with recursion. *Electr. Notes Theor. Comput. Sci.*, 41(3):70–88, 2000.

12 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.

13 Ales Bizjak and Lars Birkedal. Step-indexed logical relations for probability. In *Proc. of FOSSACS 2015*, pages 279–294, 2015.

14 Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A core quantitative coeffect calculus. In *Proc. of ESOP 2014*, pages 351–370, 2014.

15 Roy L. Crole. Completeness of bisimilarity for contextual equivalence in linear theories. *Logic Journal of the IGPL*, 9(1):27–51, 2001.

16 Raphaëlle Crubillé and Ugo Dal Lago. On probabilistic applicative bisimulation and call-by-value lambda-calculi. In *Proc. of ESOP 2014*, pages 209–228, 2014.

17 Raphaëlle Crubillé and Ugo Dal Lago. Metric reasoning about lambda-terms: The affine case. In *Proc. of LICS 2015*, pages 633–644, 2015.

18 Raphaëlle Crubillé and Ugo Dal Lago. Metric reasoning about lambda-terms: The general case. In *Proc. of ESOP 2017*, pages 341–367, 2017.

19 Ugo Dal Lago and Francesco Gavazzo. Effectful normal form bisimulation. In *Proc. of ESOP 2019*, pages 263–292, 2019.

20 Ugo Dal Lago and Francesco Gavazzo. Resource transition systems and full abstraction for linear higher-order effectful programs (extended version), 2021. `arXiv:2106.12849`.

21 Ugo Dal Lago, Francesco Gavazzo, and Paul Blain Levy. Effectful applicative bisimilarity: Monads, relators, and howe's method. In *Proc. of LICS 2017*, pages 1–12, 2017.

22 Ugo Dal Lago, Francesco Gavazzo, and Ryo Tanaka. Effectful applicative similarity for call-by-name lambda calculi. *Theor. Comput. Sci.*, 813:234–247, 2020.

23 Ugo Dal Lago and Martin Hofmann. Bounded linear logic, revisited. In *Proc. of TLCA 2009*, pages 80–94, 2009.

24 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.

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

26 Yuxin Deng and Yuan Feng. Bisimulations for probabilistic linear lambda calculi. In *Proc. of TASE 2017*, pages 1–8, 2017.

27 Yuxin Deng and Yu Zhang. Program equivalence in linear contexts. *Theor. Comput. Sci.*, 585:71–90, 2015.

28 Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. Linear-use CPS translations in the enriched effect calculus. *Logical Methods in Computer Science*, 8(4), 2012.

29 Marco Gaboardi, Shin-ya Katsumata, Dominic A. Orchard, Flavien Breuvart, and Tarmo Uustalu. Combining effects and coeffects via grading. In *Proc. of ICFP 2016*, pages 476–489, 2016.

30 Francesco Gavazzo. Quantitative behavioural reasoning for higher-order effectful programs: Applicative distances. In *Proc. of LICS 2018*, pages 452–461, 2018.

31 Francesco Gavazzo. *Coinductive Equivalences and Metrics for Higher-order Languages with Algebraic Effects.* PhD thesis, University of Bologna, Italy, 2019. URL: `http://amsdottorato.unibo.it/9075/`.

32 Dan R. Ghica and Alex I. Smith. Bounded linear types in a resource semiring. In *Proc. of ESOP 2014*, pages 331–350, 2014.

33 J-Y. Girard, A. Scedrov, and P.J. Scott. Bounded linear logic: A modular approach to polynomial-time computability. *Theor. Comput. Sci.*, 97:1–66, 1992.

34 Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

35 Martin Hyland, Gordon D. Plotkin, and John Power. Combining effects: Sum and tensor. *Theor. Comput. Sci.*, 357(1-3):70–99, 2006.

36 Patricia Johann, Alex Simpson, and Janis Voigtländer. A generic operational metatheory for algebraic effects. In *Proc. of LICS 2010*, pages 209–218. IEEE Computer Society, 2010.

37 Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. Integrating linear and dependent types. In *Proc. of POPL 2015*, pages 17–30, 2015.

38 Alexander Kurz and Jiri Velebil. Relation lifting, a survey. *J. Log. Algebr. Meth. Program.*, 85(4):475–499, 2016.

39 Ugo Dal Lago and Francesco Gavazzo. On bisimilarity in lambda calculi with continuous probabilistic choice. In *Proc. of MFPS 2019*, pages 121–141, 2019.

40 Søren B. Lassen. Bisimulation in untyped lambda calculus: Böhm trees and bisimulation up to context. *Electr. Notes Theor. Comput. Sci.*, 20:346–374, 1999.

41 Søren B. Lassen. Eager normal form bisimulation. In *Proceedings of LICS 2005*, pages 345–354, 2005.

42 Paul B. Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Inf. Comput.*, 185(2):182–210, 2003.

43 Saunders MacLane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

44 Jean-Marie Madiot, Damien Pous, and Davide Sangiorgi. Bisimulations up-to: Beyond first-order transition systems. In *Proc. of CONCUR 2014*, pages 93–108, 2014.

45 Ian A. Mason and Carolyn L. Talcott. Equivalence in functional languages with effects. *J. Funct. Program.*, 1(3):287–327, 1991.

46 Cristina Matache and Sam Staton. A sound and complete logic for algebraic effects. In *Proc. of FOSSACS 2019*, pages 382–399, 2019.

47 Nicholas D. Matsakis and Felix S. Klock II. The rust language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, pages 103–104, 2014.

48 Rasmus Ejlers Møgelberg and Sam Staton. Linear usage of state. *Logical Methods in Computer Science*, 10(1), 2014.

49 Eugenio Moggi. Computational lambda-calculus and monads. In *Proc. of LICS 1989*, pages 14–23. IEEE Computer Society, 1989.

50 Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

51 J. Morris. *Lambda Calculus Models of Programming Languages.* PhD thesis, MIT, 1969.

52 Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.*, 3(ICFP):110:1–110:30, 2019.

53 Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. Coeffects: a calculus of context-dependent computation. In *Proc. of ICFP 2014*, pages 123–135, 2014.

54 Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10(3):321–359, 2000.

55 Marinus J. Plasmeijer. CLEAN: a programming environment based on term graph rewriting. *Electr. Notes Theor. Comput. Sci.*, 2:215–221, 1995.

56 Gordon Plotkin. Lambda-definability and logical relations. Technical Report SAI-RM-4, School of A.I., University of Edinburgh, 1973.

57 Gordon D. Plotkin and John Power. Adequacy for algebraic effects. In *Proc. of FOSSACS 2001*, pages 1–24, 2001.

58 Gordon D. Plotkin and John Power. Semantics for algebraic operations. *Electr. Notes Theor. Comput. Sci.*, 45:332–345, 2001.

59 Gordon D. Plotkin and John Power. Notions of computation determine monads. In *Proc. of FOSSACS 2002*, pages 342–356, 2002.

60 Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

61 John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

62 Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. *ACM Trans. Program. Lang. Syst.*, 33(1):5:1–5:69, 2011.

63 Davide Sangiorgi and Valeria Vignudelli. Environmental bisimulations for probabilistic higher-order languages. In *Proceedings of POPL 2016*, pages 595–607, 2016.

64 Peter Selinger and Benoît Valiron. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science*, 16(3):527–552, 2006.

65 Peter Selinger and Benoît Valiron. A linear-non-linear model for a computational call-by-value lambda calculus (extended abstract). In *Proc. of FOSSACS 2008*, pages 81–96, 2008.

66 Kurt Sieber. Reasoning about sequential functions via logical relations. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Applications of Categories in Computer Science*, volume 177 of *London Mathematical Society Lecture Note Series*, pages 258–269. Cambridge University Press, 1992.

67 Alex Simpson and Niels Voorneveld. Behavioural equivalence via modalities for algebraic effects. In *Proc. of ESOP 2018*, pages 300–326, 2018.

68 David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theor. Comput. Sci.*, 227(1-2):231–248, 1999.

69 Philip Wadler. Linear types can change the world! In *Programming concepts and methods, 1990*, page 561, 1990.

70 Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, pages 24–52, 1995.

Z

Syntax-Free Developments

Vincent van Oostrom
Universität Innsbruck, Austria

Abstract

We present the Z-property and instantiate it to various rewrite systems: associativity, positive braids, self-distributivity, the lambda-calculus, lambda-calculi with explicit substitutions, orthogonal TRSs, ... The Z-property is proven equivalent to Takahashi's angle property by means of a syntax-free notion of development. We show that several classical consequences of having developments such as confluence, normalisation, and recurrence, can be regained in a syntax-free way, and investigate how the notion corresponds to the classical syntactic notion of development in term rewriting.

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

Keywords and phrases rewrite system, confluence, normalisation, recurrence

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.24

Acknowledgements Patrick Dehornoy introduced me to the main themes presented here, and indeed this paper was always intended to be a joint one. His work continues to be an inspiration. I want to thank Bertram Felgenhauer, Julian Nagele, and Christian Sternagel for discussions on their Isabelle formalisations of the Z-property.

Dedicated to Patrick Dehornoy

1 Introduction

Confluence of rewrite systems is discussed in order-theoretic terms on the first page of [25]. It expresses the existence of an *upper bound*[1] for *pairs* of objects having a common lower bound, in the quasi-order obtained by the reflexive–transitive closure of a rewrite system. Qualifying confluence proof-methods from this order-theoretic perspective, Newman's Lemma is seen to construct the *greatest* upper bound (the normal form) and the Tait–Martin-Löf (TML) method [4] the *least* upper bound [21, 38].[2] The Z-property, depicted in Fig. 1 and formally defined in the preliminaries, introduced here is based on constructing an upper bound for *sets* of objects having a common single-step lower bound. The choice of upper bound is arbitrary but should be *monotonic*; increasing the single-step lower-bound should increase the constructed upper bound. In complexity, establishing *some* upper bound is often much shorter and simpler than getting a *tight* upper bound. The choice offered by the Z-property enables the same for proving confluence, as we illustrate in Sect. 3.

Skolemising the existence of upper bounds gives rise to a function $\bullet$[3] mapping each object a to the chosen upper bound $a^\bullet$ of objects b such that $a \to b$, i.e. having a as single-step lower bound. Accordingly, we define the many-step rewrite *strategy* $\multimap\!\!\!\bullet\!\!\to$ to rewrite a into $a^\bullet$. For instance, taking as upper bound of a term t the term $t^\bullet$ obtained by a complete *development* of the *full* set of redexes in t, $\multimap\!\!\!\bullet\!\!\to$ is known as the *Gross–Knuth/full substitution* strategy in the λ-calculus/term rewriting [4, 38]. Based on $\bullet$, the classical notion of a

[1] [25] employs the reverse order, so speaks of existence of lower bounds.
[2] Newman leaves studing least upper bounds *for later* [25, p. 223] but we didn't find later work by him on this. TML in fact gives least upper bounds only *up to* permutation equivalence [21, 38].
[3] We will speak of the *bullet* function with the suggestion $\multimap\!\!\!\bullet\!\!\to$ is bullet-fast; cf. Sect. 4.1.

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi; Article No. 24; pp. 24:1–24:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

development [5, 4, 38] can be given a *syntax-free* definition as $a \xrightarrow{\circ} b$ if $a \twoheadrightarrow b \twoheadrightarrow a^\bullet$; that is, a *develops* to b if b is *between* a and $a^\bullet$; with our notations suggesting that $\xrightarrow{\circ}$ is a development that is not as *full* as $\xrightarrow{\bullet}$ is. In Sect. 4 we first show that if the Z-property holds then several results (on confluence, normalisation, and recurrence) can be obtained in a syntax-free way, i.e. in terms of $\xrightarrow{\bullet}$ and $\xrightarrow{\circ}$. Next we investigate for term rewrite systems in how far our syntax-free definition of developments corresponds or can be made to correspond to the traditional syntactic definition, and show they correspond in the absence of *syntactic accidents*.

▶ Remark 1. Thinking of reduction steps and reductions to normal form as *small* respectively *big* step semantics, $\xrightarrow{\bullet}$ can be seen as a *medium* step semantics; although $\xrightarrow{\bullet}$-steps need not directly yield a normal form, they are monotonic. This may be suitable in a setting where for a step $a \to b$, the semantics of b should be *greater* than that of a, i.e. approximate better.

2 Preliminaries

We define our key notions for abstract rewriting with which we assume basic familiarity [38].

▶ **Definition 2.** *A* rewrite *system is a system comprising a set of* objects*, a set of (*rewrite*)* steps*, and functions* src, tgt *mapping a step to its* source*,* target *object. Two steps are called* co-initial *if they have the same sources,* co-final *if they have the same targets, and* composable *if the target of the former is the source of the latter. The corresponding pair of steps is then called, respectively, a* peak*, a* valley*, and* consecutive*.*

▶ Remark 3. We follow [25] in taking steps as first-class citizens of rewrite *systems* and speak of a rewrite *relation* (only) if there is at most one step between any two objects.

We use arrow-like notations to denote rewrite systems and their steps, let $a, b, \ldots$ range over objects, and $\phi, \psi, \ldots$ over steps. Sources and targets naturally extend to peaks, valleys, and consecutive steps; e.g., the source of a peak is the common source of its steps and its target is its pair of targets.

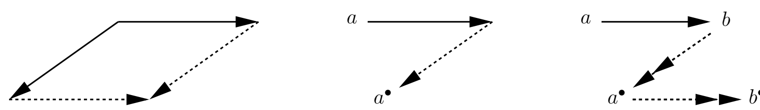


Figure 1 Diamond, angle, and Z-property for bullet function $\bullet$; named after the diagram shapes.

▶ **Definition 4.** *A rewrite system $\to$ has the (see Fig. 1):*
- diamond *property if for every peak there is a composable valley;*
- angle *property if there is map $\bullet$ such that $b \to a^\bullet$ for every a and step $a \to b$; and*
- Z *property if there is a map $\bullet$ such that $b \twoheadrightarrow a^\bullet \twoheadrightarrow b^\bullet$ for every a and step $a \to b$.*

where $\twoheadrightarrow$ denotes reduction*, finite (possibly empty) composition of steps. A map $\bullet$ is* extensive *if $a \twoheadrightarrow a^\bullet$ for all a, and induces a rewrite system $\xrightarrow{\bullet}$ having the same objects as $\to$ and steps $a \xrightarrow{\bullet} a^\bullet$ for all a not in $\to$-normal form.*

▶ Remark 5. The diamond and angle properties are relatively standard in rewriting, see e.g. [38, Def. 1.1.8]; our angle property is the Skolemisation of the *tri*angle property there. We obtained the Z-property in 2007 by abstracting Dehornoy's proof-method for showing confluence of self-distributivity [6] with preliminary results distributed and presented at diverse venues, e.g. [7, 31]. It has been introduced both before, for the λ-calculus, in [18, Ex. 4.1] and after in [16]. In the meantime it has been formalised [9] and applied, e.g. [23, 11].

Two angles make a diamond, but the angle property is stronger than the diamond property. If the Z-property holds $\bullet$ is *monotonic* on reductions: if $a \twoheadrightarrow b$ then $a^\bullet \twoheadrightarrow b^\bullet$ (by induction).

▶ **Example 6.** Less-than $<$ on $\mathbb{Z}$ has the diamond but not the angle property for lack of upper bounds of infinite sets of numbers. Note that the predecessor relation on $\mathbb{Z}$ does have the angle property, despite inducing the same quasi-order as $<$.

The following simple but key result was the starting point of our investigations on the Z-property. It hinges on a syntax-free definition of the classical notion of *development* [4, 38].

▶ **Definition 7.** *For rewrite system $\to$ and map $\bullet$ on its objects, the* $\bullet$-development *rewrite system $\mathrel{\circ\!\!\!\rightarrow}$ has the objects of $\to$ and a step $a \mathrel{\circ\!\!\!\rightarrow} b$ for each pair of $\to$-reductions $a \twoheadrightarrow b \twoheadrightarrow a^\bullet$.*

One may think of b as being *between* a and $a^\bullet$ and of $\mathrel{\circ\!\!\!\rightarrow}$ as comprising prefixes or left-divisors of $\mathrel{\bullet\!\!\!\rightarrow}$ w.r.t. composition (for sources not in normal form).

▶ **Lemma 8.** *Let $\to$ be a rewrite system.*

- *$\to$ has the Z-property iff some $\to'$ such that $\to \subseteq \to' \subseteq \twoheadrightarrow$ has the angle property;*[4]
- *if $\to$ has the Z-property for $\bullet$, then it has the Z-property for some extensive $\star$; and*
- *$\to$ has the Z-property for an extensive $\bullet$ iff some rewrite system $\to'$ such that $\to \subseteq \to' \subseteq \twoheadrightarrow$ has the angle property and $a \to' a^\bullet$ for all a.*

Proof. We only provide a detailed proof of the first, main, item.

- we show both directions taking the same bullet function $\bullet$.
  For the if-direction, assume $\to'$ has the angle property, $\to \subseteq \to' \subseteq \twoheadrightarrow$, and suppose $a \to b$. Then by $\to \subseteq \to'$ and the angle property for $a \to' b$ we have $b \to' a^\bullet$, hence $a^\bullet \to' b^\bullet$ by applying the angle property again. Two angles make a Z; using $\to' \subseteq \twoheadrightarrow$ twice, we conclude to $b \twoheadrightarrow a^\bullet$ and $a^\bullet \twoheadrightarrow b^\bullet$.
  For the only–if-direction, assume $\to$ has the Z-property. Consider the $\bullet$-development rewrite system $\mathrel{\circ\!\!\!\rightarrow}$. To show $\mathrel{\circ\!\!\!\rightarrow}$ has the angle property, suppose $a \mathrel{\circ\!\!\!\rightarrow} b$. By definition $a \twoheadrightarrow b \twoheadrightarrow a^\bullet$. Combining $b \twoheadrightarrow a^\bullet$ with $a^\bullet \twoheadrightarrow b^\bullet$, which follows from $a \twoheadrightarrow b$ by monotonicity of $\bullet$, yields $b \mathrel{\circ\!\!\!\rightarrow} a^\bullet$ by definition of $\mathrel{\circ\!\!\!\rightarrow}$, showing the angle property. That the first inclusion in $\to \subseteq \mathrel{\circ\!\!\!\rightarrow} \subseteq \twoheadrightarrow$ holds follows from that $a \to b$ entails $b \twoheadrightarrow a^\bullet$ by the Z-property hence by definition $a \mathrel{\circ\!\!\!\rightarrow} b$, and that the second inclusion holds from that $a \mathrel{\circ\!\!\!\rightarrow} b$ unfolds to $a \twoheadrightarrow b \twoheadrightarrow a^\bullet$.
- one checks that defining $\star$ to be $\bullet$ updated to map each object that is *not* the source of some step to itself, works; and
- one checks the additional conditions on either side in the first item. The if-direction is trivial since $\to' \subseteq \twoheadrightarrow$ by assumption. ◀

Adjoining being extensive to the angle property in Fig. 1 gives rise to a triangle, i.e. the second and third items reconcile both names of the property.

Although the intuition is that $\bullet$-developments correspond to developments, the former, by being defined in a syntax-free way, are more liberal (we will look into this in Sect. 4.4) as shown by:

[4] The inclusions are relation inclusions, i.e. concern the rewrite relation underlying the rewrite systems.

▶ **Example 9.** The rewrite system $a_i \to a_{i+1 \bmod 4}$ has the Z-property for the function $\bullet$ mapping a_i to $a_{i+1 \bmod 4}$ because $\to$ is deterministic. Classically there are only two developments from a_0 namely to itself, the empty development, and to a_1. However, because $\to$ is cyclic there are more $\bullet$-developments, e.g. $a_0 \twoheadrightarrow\!\!\!\!\circ\ a_2$ (since $a_0 \twoheadrightarrow a_2 \twoheadrightarrow a_1 = a_0^\bullet$).

3 Examples of the Z-property

We present (non-)examples of rewrite systems having the Z-property with a focus on the diversity of the examples and the similarity of the proofs. We give proofs in as far as they could serve as blue-prints of proofs of the Z-property for related calculi. We proceed from abstract to more concrete rewrite systems.

3.1 Abstract

We investigate for some known confluence criteria for (abstract) rewrite systems [3, 38] whether or not they entail the Z-property. We assume $\to$ is a rewrite system. In the previous section we have already seen a characterisation of the Z-property via the angle property. That the Z-property holds for *deterministic* (if $a \to b$ and $a \to c$, then $a = b$) systems by mapping to *the next object* was exemplified in Ex. 9.

▶ **Lemma 10.** *If $\to$ is deterministic, then it has the Z-property.*

In case a rewrite system is terminating mapping to the *greatest* object works.

▶ **Lemma 11.** *If $\to$ is terminating, then $\to$ has the Z-property iff $\to$ is locally confluent.*

Proof. Suppose $\to$ is locally confluent and terminating. Let $\bullet$ be the *normal form* function mapping each object to its $\to$-normal form, This is well-defined: the normal form exists by termination and is unique as local confluence entails confluence by Newman's Lemma. Thus we conclude to the Z-property since if $a \to b$ then $b \twoheadrightarrow a^\bullet = b^\bullet$. Vice versa, if $\to$ has the Z-property for $\bullet$ then $a^\bullet$ is a common reduct to all b such that $a \to b$. ◀

Ex. 6 shows there are confluent rewrite systems $\to$ that do not *have* the Z-property but *admit* it in that there is a rewrite system $\to'$ presenting the same quasi-order, i.e. $\twoheadrightarrow = \twoheadrightarrow'$, that *does* have the Z-property: $<$ does not *have* the Z-property but *admits* it as it is the reflexive–transitive closure of the predecessor relation that does have the Z-property (by being deterministic).[5] By the first item of Lem. 8 a rewrite system admits the Z-property iff it admits the angle property, using for the only–if-direction that $\to \subseteq \to' \subseteq \twoheadrightarrow$ entails $\to$ and $\to'$ present the same quasi-order. But there are confluent rewrite systems not admitting either.

▶ **Example 12.** Consider the confluent rewrite system[6] given by $a \to b_i \to c_i \to c_{i+1}$ for $i \in \mathbb{N}$, and suppose $\to'$ were some presentation of it having the Z-property. Observe that then $a \to' b_i$ for $i \in \mathbb{N}$, since there are no objects between a and b_i in $\to$, but there is no common upper bound to all b_i in $\to$, so neither there is one in $\to'$.

▶ Remark 13. Bullet functions for the Z-property may be *incomparable* (comparing their images bulletwise by $\twoheadrightarrow$), but are preserved under composition allowing arbitrary *speed-up*.

[5] If a rewrite system has the Z-property, then so does its so-called *transitive reduction*, but not necessarily the other way around. However note that $<$ admits the Z-property even on $\mathbb{R}$, e.g. by restricting to pairs of reals having distance at most 1, despite that $<$ then has no transitive reduction.

[6] The rewrite system is a variation on the rewrite systems visualised in [12, Fig. 2].

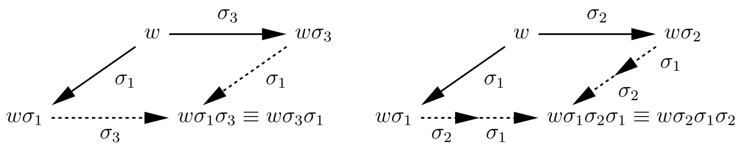


Figure 2 Local confluence diagrams for positive braids.

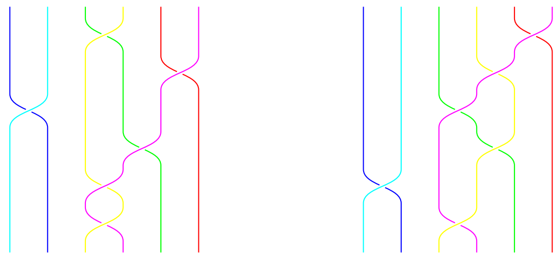


Figure 3 Isotopic braids, $\sigma_3\sigma_5\sigma_1\sigma_4\sigma_3\sigma_3 \equiv \sigma_5\sigma_4\sigma_3\sigma_4\sigma_1\sigma_3$, deformable into one another.

3.2 Positive braids

Positive braids have the Z-property [6] or equivalently the angle property [34],[38, Sect. 8.9].

▶ **Definition 14.** *The rewrite system $\mathcal{B}^+$ of (*positive*) *braids *on ℓ* strands *has:*

- *as objects* braids, *words over the* Artin *generators σ_i for $1 \leq i < \ell$, modulo*

$$\sigma_i\sigma_j\sigma_i = \sigma_j\sigma_i\sigma_j \qquad \text{if } |i-j| = 1 \tag{1}$$

$$\sigma_i\sigma_j = \sigma_j\sigma_i \qquad \text{if } |i-j| > 1 \tag{2}$$

- *steps $w \to w\sigma_i$ for any braid w and $1 \leq i < \ell$.*

The equivalence generated by (1) and (2) is denoted by $\equiv$. The rewrite system $\mathcal{B}^+$ is locally confluent as illustrated in Figure 2: any pair of distinct generators σ_i, σ_j either is *too far apart* (2) like σ_1 and σ_3 on the left, or *too close together* (1) like σ_1 and σ_2 on the right. See Figure 3 for two words representing the same positive braid on 6 strands. Extending a braid by a *full* swap, crossing all strands over another as represented by the *Garside* word, works, the intuition being that is the *least* way to extend all *single* steps. The proof is short and by straightforward inductions.

▶ **Lemma 15.** *$\mathcal{B}^+$ has the Z-property for the map that suffixes the Garside word.*

Proof. The bullet function $\bullet$ suffixing the Garside word is formally defined by $w^\bullet := wG_\ell$, where, starting crossing from the left, the Garside word may be inductively defined by $G_0 := \varepsilon$ and if $n > 0$, then $G_n := G_{n\dot-1}\sigma_{\langle n,1\rangle}$ with $\sigma_{\langle i,j\rangle} := \sigma_{i\dot-1}\dots\sigma_j$ crossing the ith strand over $i \dot- j$ strands to its left. The key property of G_ℓ is that it is a so-called Garside *element* as each generator is both a left and right divisor of it. More specifically, we claim that for all $1 \leq i < n$ there exists a braid G_n^i such that (cf. Ex. 16)

$$\sigma_i G_n^i \equiv G_n \equiv G_n^i \sigma_{n\dot-i} \tag{3}$$

From the claim we conclude to the Z-property, since for a step $w \to w\sigma_i$ then $w\sigma_i \twoheadrightarrow w\sigma_i G_n^i \equiv wG_n \to wG_n\sigma_{n\dot-i} \equiv w\sigma_i G_n^i \sigma_{n\dot-i} \equiv w\sigma_i G_n$.

It remains to prove the claim (a well-known fact). The intuition for G_n^i is that it is the *residual* of G_n after σ_i, i.e. what remains to be done of a *full* swap after swapping i. Formally, it may be inductively defined by $G_n^{n\dot{-}1} := G_{n\dot{-}1}\sigma_{\langle n,2\rangle}$ and $G_n^i := G_{n\dot{-}1}^i\sigma_{\langle n,1\rangle}$ otherwise. Accordingly, we show (3) by induction on n, with trivial base case, and cases on whether or not $i = n \dot{-} 1$:

$$\begin{array}{rcl} \sigma_i G_n^i & = & \sigma_i G_{n\dot{-}1}^i \sigma_{\langle n,1\rangle} \\ & \equiv_{IH} & G_{n\dot{-}1}\sigma_{\langle n,1\rangle} \\ & \equiv_{IH} & G_{n\dot{-}1}^i \sigma_{n\dot{-}1\dot{-}i}\sigma_{\langle n,1\rangle} \\ & \equiv_{(ii)} & G_{n\dot{-}1}^i \sigma_{\langle n,1\rangle}\sigma_{n\dot{-}i} \\ & = & G_n^i \sigma_{n\dot{-}i} \end{array} \qquad \begin{array}{rcl} \sigma_{n\dot{-}1} G_n^{n\dot{-}1} & = & \sigma_{n\dot{-}1} G_{n\dot{-}2}\sigma_{\langle n\dot{-}1,1\rangle}\sigma_{\langle n,2\rangle} \\ & \equiv_{(i)} & G_{n\dot{-}2}\sigma_{\langle n,1\rangle}\sigma_{\langle n,2\rangle} \\ & \equiv_{(iii)} & G_{n\dot{-}2}\sigma_{\langle n\dot{-}1,1\rangle}\sigma_{\langle n,1\rangle} \\ & = & G_{n\dot{-}1}\sigma_{\langle n,1\rangle} \\ & = & G_n^{n\dot{-}1}\sigma_1 \end{array}$$

where (i) follows by (2); $\sigma_{n\dot{-}1}$ and $G_{n\dot{-}2}$ commute, i.e. $\sigma_{n\dot{-}1}G_{n\dot{-}2} \equiv G_{n\dot{-}2}\sigma_{n\dot{-}1}$, as their generators are *too far apart*, (ii) holds since for all $i \dot{-} 1 > k \geq j$:

$$\begin{array}{rlll} \sigma_k\sigma_{\langle i,j\rangle} & \equiv_{(2)} & \sigma_{\langle i,k+2\rangle}\sigma_k\sigma_{k+1}\sigma_k\sigma_{\langle k,j\rangle} & \\ & \equiv_{(1)} & \sigma_{\langle i,k+2\rangle}\sigma_{k+1}\sigma_k\sigma_{k+1}\sigma_{\langle k,j\rangle} & \equiv_{(2)} \quad \sigma_{\langle i,j\rangle}\sigma_{k+1} \end{array}$$

and (iii) follows from (ii) by induction on $\sigma_{\langle n,2\rangle}$. ◀

▶ **Example 16.** To see that (3) holds for $i := 2$ and $n := 4$, we first compute $G_4^2 := \sigma_1\sigma_2\sigma_3\sigma_2\sigma_1$ and $G_4 := \sigma_1\sigma_2\sigma_1\sigma_3\sigma_2\sigma_1$, and then verify $\underline{\sigma_2\sigma_1\sigma_2}\sigma_3\sigma_2\sigma_1 \equiv_{(1)} \sigma_1\sigma_2\underline{\sigma_1\sigma_3}\sigma_2\sigma_1 \equiv_{(2)} \sigma_1\sigma_2\sigma_3\underline{\sigma_1\sigma_2\sigma_1} \equiv_{(1)} \sigma_1\sigma_2\sigma_3\sigma_2\sigma_1\sigma_2$.

3.3 First-order terms

In this section we consider TRSs, i.e. first-order term rewrite systems [3, 38]. We show the Z-property holds for orthogonal TRSs for the *full development* and the *full superdevelopment* functions, for weakly orthogonal TRSs by the *maximal multistep* map, for *associativity* by an inductive *normal form* function, and extending that, for *self-distributivity* by the *full distribution* function. Our presentation suggests the commonality between the proofs the Z-property holds. We assume $\mathcal{T}$ is a TRS and $\to_{\mathcal{T}}$ or simply $\to$ to be its underlying rewrite system on terms $t, s, r, \ldots$. Each bullet function $\bullet$ on terms defined below is assumed to be pointwise extended to vectors of terms $\vec{t}, \vec{s}, \ldots$ and substitutions $\sigma, \tau, \ldots$. We first observe that as a corollary to Lem. 11 and Huet's Critical Pair Lemma we immediately have:

▶ **Corollary 17.** *A terminating TRS has the Z-property iff all its critical pairs are joinable.*

3.3.1 Orthogonal

We show *orthogonal* TRSs, i.e. left-linear and non-overlapping, have the Z-property.

▶ **Example 18.** The classical example of an orthogonal TRS is Combinatory Logic (CL). It has a binary symbol @ and constants K, S, I and rules, written in full on the left and *applicatively* [38, Sect. 3.3.5] on the right (making @ implicit, infix, and associate to the left):

$$\begin{array}{rcl} @(I,x) & \to & x \\ @(@(K,x),y) & \to & x \\ @(@(@(S,x),y),z) & \to & @(@(x,z),@(y,z)) \end{array} \qquad \begin{array}{rcl} Ix & \to & x \\ Kxy & \to & x \\ Sxyz & \to & xz(yz) \end{array}$$

For orthogonal TRSs mapping a term to the result of contracting all redexes works, the intuition being again that it is the *least* way of extending all single steps. This amounts to an inductive definition of the *full substitution* or *maximal multistep* strategy [38, Def. 9.3.18].

▶ **Definition 19.** *For an orthogonal TRS,* full development $\bullet$ *is inductively defined by*

$$
\begin{aligned}
x^\bullet &:= x \\
f(\vec{t})^\bullet &:= r^\sigma \quad \textit{if } f(\vec{t}) \textit{ is a redex and } f(\vec{t}^\bullet) = \ell^\sigma \textit{ for some rule } \ell \to r \textit{ and substitution } \sigma \\
&:= f(\vec{t}^\bullet) \quad \textit{otherwise}
\end{aligned}
$$

▶ **Example 20.** In CL $(I(Ix))^\bullet = x$ and $(IIx)^\bullet = Ix$ contracting II but not the *created* Ix.

▶ Remark 21. By orthogonality, if for some *redex* t there is a reduction without head-steps $t \twoheadrightarrow \ell^\tau$ for lhs of a rule ℓ and substitution τ, then $t = \ell^\sigma$ for some substitution σ such that $\sigma \twoheadrightarrow \tau$. Vice versa, if we have such reduction $\ell^\tau \twoheadrightarrow t$ for some *term* t, then $t = \ell^\sigma$ and $\tau \twoheadrightarrow \sigma$.

▶ **Lemma 22.**

(Extensive) $t \twoheadrightarrow t^\bullet$ *for all terms* t*;*
(Rhs) $t^{(\sigma^\bullet)} \twoheadrightarrow (t^\sigma)^\bullet$ *for terms* t*, substitutions* σ*;* $t^{(\sigma^\bullet)} = (t^\sigma)^\bullet$ *if* t *is a proper subterm of a lhs;*
(Z) $\to$ *has the Z-property for the full development function.*

Proof.

(Extensive) By induction on t. If t is a variable x, then $t^\bullet = x$ and we conclude by reflexivity of $\twoheadrightarrow$. Otherwise t has shape $f(\vec{t})$ and $\vec{t} \twoheadrightarrow \vec{t^\bullet}$ by the IH and transitivity, so $f(\vec{t}) \twoheadrightarrow f(\vec{t^\bullet})$. If the third clause applies we immediately conclude. Otherwise, $f(\vec{t}^\bullet) = \ell^\sigma$ and $t^\bullet = r^\sigma$ for symbol f, terms $\vec{t}$, rule $\ell \to r$ and substitution σ, and we append $\ell^\sigma \to r^\sigma$;

(Rhs) We show the first by induction on t. If t is some variable x, then both sides are equal to $\sigma(x)^\bullet$. Otherwise, $t = f(\vec{t})$ for some symbol f and terms $\vec{t}$, and $\vec{t}^{(\sigma^\bullet)} \twoheadrightarrow (\vec{t^\sigma})^\bullet$ by the IH, hence $f(\vec{t})^{(\sigma^\bullet)} \twoheadrightarrow f((\vec{t^\sigma})^\bullet)$. If the third clause applies to $f(\vec{t^\sigma})$ then we conclude, and otherwise we append a corresponding final root step to the reduction. For the second, note we have the stronger $f((\vec{t^\sigma})^\bullet) = f(\vec{t})^{(\sigma^\bullet)}$ in the induction step, so the second clause never applies as this is not an instance of a lhs by assumption on t and orthogonality;

(Z) We show for the full-development function $\bullet$, that $s \twoheadrightarrow t^\bullet \twoheadrightarrow s^\bullet$ for all steps $t \to s$ by induction on t. The case that t is a single variable being impossible, as variables cannot be rewritten due to the assumption that lhss of rules are not single variables, assume t has shape $f(\vec{t})$ for symbol f and terms $\vec{t}$ and distinguish cases on the clause of $\bullet$.
Suppose the second clause applies, i.e. $f(\vec{t}^\bullet) = \ell^\tau$ for some rule $\ell \to r$ and $t^\bullet = r^\tau$ for symbol f, terms $\vec{t}$, rule $\ell \to r$ and substitution τ. Distinguish cases on the step $t \to s$.

- If the step is a head step, then it must have shape $t = \ell^\sigma \to r^\sigma = s$ for the *same* rule $\ell \to r$ and some substitution σ such that $\sigma^\bullet = \tau$, by Rem. 21 and (Rhs) as $t = f(\vec{t}) \twoheadrightarrow f(\vec{t}^\bullet)$ by (Extensive). Then (Z) holds by $r^\sigma \twoheadrightarrow r^\tau = (\ell^\sigma)^\bullet = r^{(\sigma^\bullet)} \twoheadrightarrow (r^\sigma)^\bullet$ using (Extensive) for σ for the first reduction and (Rhs) for the second; and
- If the step is not a head step, then $s = f(\vec{s})$ for some $\vec{s}$ equal to $\vec{t}$ except for some i for which $t_i \to s_i$, for which by the IH $s_i \twoheadrightarrow t_i^\bullet \twoheadrightarrow s_i^\bullet$. From that, Rem. 21 and (Extensive) $\ell^\tau = f(\vec{t}^\bullet) \twoheadrightarrow f(\vec{s}^\bullet) = \ell^\sigma \to r^\sigma = s^\bullet$ for some substitution σ with $\tau \twoheadrightarrow \sigma$. Using that for the second reduction, and the IH and (Extensive) for the first, (Z) holds by $f(\vec{s}) \twoheadrightarrow f(\vec{t}^\bullet) = \ell^\tau \to r^\tau = f(\vec{t})^\bullet \twoheadrightarrow r^\sigma = s^\bullet = f(\vec{s})^\bullet$.

Suppose the third clause applies, so $t^\bullet = f(\vec{t}^\bullet)$. Then the step cannot be a head step (otherwise $f(\vec{t}^\bullet)$ would be a redex) and $s = f(\vec{s})$ for some $\vec{s}$ equal to $\vec{t}$ except for some i for which $t_i \to s_i$, for which by the IH $s_i \twoheadrightarrow t_i^\bullet \twoheadrightarrow s_i^\bullet$. Then (Z) holds by using the IH and (Extensive) on $\vec{t}$ for both reductions in $f(\vec{s}) \twoheadrightarrow f(\vec{t}^\bullet) = f(\vec{t})^\bullet \twoheadrightarrow f(\vec{s}^\bullet)$, to which a further head step must be appended in case the second clause applies to s to yield $s^\bullet$. ◀

In the proof of the lemma the condition $f(\vec{t})$ *is a redex* in the second clause of Def. 19 was never used. Indeed, dropping it preserves the proof. We dub the resulting function the full *super*development function as it relates to the full development function as Aczel's proof of confluence [2, 26] relates to the Tait–Martin-Löf proof [4]; see [35] for a discussion. Full superdevelopments also contract all *upward created* [17] redexes.

▶ **Definition 23.** *Replacing* redex *by* term *in Def. 19 gives the full* super*development function.*

▶ **Lemma 24.** $\to$ *has the Z-property for the full superdevelopment function.*

▶ **Example 25.** Compared to Ex. 20 again $(I(Ix))^\bullet = x$ but now $(IIx)^\bullet = x$ by also allowing to contract the *upward created* redex Ix. That CL has the Z-property is formalised in [9].

For *simply typed* CL we now already have seen 3 distinct functions witnessing the Z-property, in order of increasing(ly lax) upperbounds: full-development, full-superdevelopment, and normal form (Lem. 11 applies as simply typed CL is terminating).

3.3.2 Weakly orthogonal

We show *weakly* orthogonal TRSs [3, 38], having left-linear rules whose critical peaks $s \leftarrow t \to r$ are *trivial*, i.e. $s = r$, have the Z-property.

▶ **Example 26.** The TRS with rules $p(s(x)) \to x$ and $s(p(x)) \to x$ is weakly orthogonal.

▶ **Definition 27.** *For a weakly orthogonal TRS, the* maximal multistep *map* $\bullet$ *is inductively defined simultaneously with its* maximal context max *by*

$$\begin{array}{rcl@{\qquad}rcll} x^\bullet & := & x & \max(x) & := & \Box & \\ f(\vec{t})^\bullet & := & r^\sigma & \max(f(\vec{t})) & := & \Box & \textit{if } P \\ & := & f(\vec{t}^\bullet) & & := & f(\max(\vec{t})) & \textit{otherwise} \end{array}$$

where P *asks* $f(\vec{t}) = \ell^\sigma$ *for some substitution* σ*, rule* $\ell \to r$ *such that* ℓ *is a prefix of* $f(\max(\vec{t}))$.

▶ **Example 28.** For the predecessor–successor TRS of Ex. 26 letting $t := p(s(x))$ and $s := p(s(p(x)))$, we have $t^\bullet = x$ and $\max(t) = \Box$, respectively $s^\bullet = p(x)$ and $\max(s) = p(\Box)$.

The full development function being ambiguous[7] for weakly orthogonal TRSs, is resolved by the maximal multistep map by adhering to an *inside–out* strategy. The intuition for $\max(t)$ is that it comprises the *context of all maximal redexes selected for contraction by* $\bullet$, and the intuition for $\bullet$ is that it tries to find *any* lhs that is contained in that context, i.e. does not have overlap with any of the already selected redexes in its arguments. As a consequence, in P the condition ℓ *is a prefix of* $f(\max(\vec{t}))$ is always satisfied for TRSs that are orthogonal and for those the maximal multistep and full development functions coincide.

▶ **Lemma 29.** $\to$ *has the Z-property for the maximal multistep function.*

Proof. Since the Z-property is equivalent to the angle property, Lem. 8, this follows from the *maximal multistep* function having the angle property [38, Thm. 8.8.27], noting Def. 27 is a rephrasing of the notion going under the same name in the proof of that theorem. ◀

---

[7] Different maximal sets of non-overlapping redexes may exist and result in different terms. E.g. the other redexes overlap the underlined one in $p(\underline{s(p(s(x)))})$ hence the latter is maximal, but so are the other 2.

▶ Remark 30. Proceeding *outside–in* instead of *inside–out*, in a naïve way cannot work. It does not yield a bullet function having the Z-property as exemplified by the TRS with rules $c(x) \to x$, $f(f(x)) \to f(x)$ and $g(f(f(f(x)))) \to g(f(f(x)))$. We have $t \to s$ for $t := g(f(f(c(f(f(x))))))$ and $s := g(f(f(f(f(x)))))$ by contracting the c-redex, but the Z-property (monotonicity) fails for a naïve outside–in bullet function $\star$, as we do *not* have $t^\star = g(f(f(x))) \twoheadrightarrow g(f(f(f(x)))) = s^\star$. This can be overcome [8, Lem. 7.10][8], even effectively so [8, Cor. 7.27], by discarding *Takahashi configurations* [38, Prop. 9.3.5], [14, Rem. 4.38].

3.3.3 Associativity

From the above one might have the impression that the Z-property only holds for confluent TRSs that are orthogonal or closely associated to such. This is not the case.

▶ **Example 31.** The term rewrite system for *associativity* (to the right) has as single rule:

$$@(@(x,y),z) \;\to\; @(x,@(y,z)) \qquad\qquad xyz \;\to\; x(yz)$$

written on the left in standard notation and applicatively (cf. Ex. 18) on the right.

As is well-known associativity is terminating and locally confluent as its one and only critical pair is joinable. Hence it has the Z-property by Cor. 17. Here we give a direct inductive definition of the *normal form* function, cf. Rem. 1, to show that one *can* proceed similarly to the (weakly) orthogonal case, and to prepare for the case of self-distributivity below.

▶ **Definition 32.** *We give an inductive definition of the* normal form *function* $\bullet$ *depending on an auxiliary* grafting *function* $t\langle r]$ *(we assume grafting binds stronger than the implicit* @*)*

$$\begin{array}{rcl@{\qquad}rcl} x\langle r] & := & xr & x^\bullet & := & x \\ (ts)\langle r] & := & ts\langle r] & (ts)^\bullet & := & t^\bullet\langle s^\bullet] \end{array}$$

The idea is that $t\langle r]$ grafts the second argument r to the right tip of the first argument t.

▶ **Example 33.** $(xy)^\bullet = x^\bullet\langle y^\bullet] = xy$, so $(xyz)^\bullet = (xy)\langle z] = x(yz)$ and $(xyzw)^\bullet = x(y(zw))$.

Note $\bullet$ indeed only has normal forms in its image and these are preserved by grafting. The second example shows associativity can be viewed as performing an elementary case of grafting. How grafting and the normal form function interact with rewriting is captured by the following two lemmata, all of whose items are proven by induction on terms.[9]

▶ **Lemma 34.**

(Sequentialisation) $ts \twoheadrightarrow t\langle s]$*, for all terms* t, s*;*
(Compatible) $t\langle s] \twoheadrightarrow t'\langle s']$*, if* $t \twoheadrightarrow t'$ *and* $s \twoheadrightarrow s'$*; and*
(Substitution) $t\langle s]\langle r] = t\langle s\langle r]]$*, for all terms* t, s, r*.*

▶ **Lemma 35.**

(Extensive) $t \twoheadrightarrow t^\bullet$*, for all terms* t*;*
(Rhs) $t^\bullet(s^\bullet r^\bullet) \twoheadrightarrow (tsr)^\bullet$*, for all terms* t, s, r*;*
(Z) $\to$ *has the Z-property for the normal form function* $\bullet$*.*

---

[8] As shown there, this extends to infinitary rewriting, for non-collapsing TRSs.
[9] See Appendix A to check that the proofs of the two lemmata are indeed by straightforward inductions.

▶ Remark 36. Def. 32 effectively encodes a normalising *strategy*. *A priori* this entails neither termination of $\to$ nor uniqueness of the computed normal form.[10] The latter only follows by the monotonicity part of the Z-property for $\bullet$. Turning things around, *because* $\bullet$ maps to normal forms, (Extensive) and monotonicity would have sufficed to establish the Z-property, as then $t \to s$ entails $s \twoheadrightarrow s^\bullet = t^\bullet$, but that would break the analogy with other proofs here.

3.3.4 Self-distributivity

Dehornoy's proof that self-distributivity has the Z-property [6] fits in the above mould.

▶ **Example 37.** The *self-distributivity* TRS has the (applicative) rule $xyz \to xz(yz)$.

Self-distributivity is non-terminating as its lhs can be embedded in its rhs, and is locally confluent as its one and only critical peak is joinable. Both its equational and rewrite theories are highly non-trivial; the book [6] is entirely devoted to them and still much more is to say.

▶ **Example 38.** Self-distributivity has any ACI-operation (e.g., logical $\wedge$ or $\vee$) as model, as well as interpreting the binary operation as *taking the middle* between points in $\mathbb{R}^2$. The *Substitution Lemma* of the λ-calculus (cf. [32, Thm. 5]) yields an instance of self-distributivity. Self-distributivity is obtained by "forgetting" the S in the CL rule for S, or alternatively (and giving more insight) by "enriching" the rhs of the associativity rule with another copy of z.

▶ **Definition 39.** *We give an inductive definition of the* full *distribution function* $\bullet$ *[6, Def. V.3.7] depending on the* uniform *distribution* $t[s]$ *of* s *over* t *[6, Def. V.3.4].*

$$\begin{array}{rcl@{\qquad}rcl} x[s] & := & xs & x^\bullet & := & x \\ (tr)[s] & := & t[s]r[s] & (ts)^\bullet & := & t^\bullet[s^\bullet] \end{array}$$

Uniform distribution grafts the 2nd argument uniformly to all leafs $t[s] = t^{[x_1,x_2,\ldots:=x_1s,x_2s,\ldots]}$. The following key lemmata, obtained by structuring [6, Lem. V.3.6,10–12] in the same way as was done for associativity above, are again proven by straightforward induction on terms.[9]

▶ **Lemma 40.**

(Sequentialisation) $ts \twoheadrightarrow t[s]$*, for all terms* t, s*;*
(Compatible) $t[s] \twoheadrightarrow t'[s']$*, if* $t \twoheadrightarrow t'$ *and* $s \twoheadrightarrow s'$*; and*
(Substitution) $t[s][r] \twoheadrightarrow t[r][s[r]]$*, for all terms* t, s, r*.*

▶ **Lemma 41.**

(Extensive) $t \twoheadrightarrow t^\bullet$*, for all terms* t*; and*
(Z) $\to$ *has the Z-property for the full distribution function* $\bullet$*.*

3.4 The lambda-calculus

The $\lambda\beta$-calculus and the $\lambda\beta\eta$-calculus [4] being prime examples of *orthogonal* respectively *weakly orthogonal* higher-order term rewrite systems [20, 27], it is natural that the full development and full superdevelopment functions for orthogonal TRSs, and the maximal multistep map for weakly orthogonal TRSs should lift. They do. As the Z-property for the full development function is known [18]/[16] and for the full superdevelopment function was formalised [22, 9], we will be satisfied with giving the definitions and proof structure.

[10] But in fact it can be shown to do so, by choosing appropriate weights in *random descent* [33].

▶ **Definition 42.** *The* full development *function* $\bullet$ *is inductively [37, p. 121] defined by:*

$$\begin{array}{rcll} x^\bullet & := & x & \\ (\lambda x.M)^\bullet & := & \lambda x.M^\bullet & \\ (MN)^\bullet & := & M'^{[x:=N']} & \text{if } MN \text{ is a redex and } M^\bullet N^\bullet = (\lambda x.M')N' \\ & := & M^\bullet N^\bullet & \text{otherwise} \end{array}$$

The full super*development function is obtained by dropping the condition MN* is a redex *from the third clause (or replacing it by MN* is a term*; cf. Def. 19 and the text below Lem. 22).*

▶ **Example 43.** Taking $I := \lambda x.x$ in Ex. 20 gives full (super)developments as for CL.

Assuming α-equivalence, congruence of β-reduction, the Substitution Lemma [4, Lem. 2.1.16], and compatibility of β-reduction with substitution [4, Sect. 3.1], and coherence of β-reduction with abstraction, we successively show:

▶ **Lemma 44.**

(Extensive) $M \twoheadrightarrow M^\bullet$*, for all λ-terms M;*
(Rhs) $M^{(\sigma^\bullet)} \twoheadrightarrow (M^\sigma)^\bullet$ *for λ-terms M, substitutions σ; and*
(Z) $\to_\beta$ *has the Z-property for the full (super)development function* $\bullet$*.*

▶ Remark 45. It would be interesting to see whether one could have a *single* formalised statement and proof for the Z-property for both full developments and full superdevelopments.

▶ Remark 46. Our *inside–out* definition of the maximal multistep map for weakly orthogonal TRSs straightforwardly extends to all weakly orthogonal higher-order term rewrite systems, and the Z-property still holds (in [29] we established the angle property), which immediately yields the same for the $\lambda\beta\eta$-calculus. Although the *outside–in* construction on [37, p. 121, (F8*)] does yield the Z-property for the $\lambda\beta\eta$-calculus,[11] it fails to do so for weakly orthogonal higher-order term rewrite systems in general; monotonicity fails for the TRS in Rem. 30.

▶ Remark 47. We do not know whether there is a generalisation of the full superdevelopment function to the $\lambda\beta\eta$-calculus. A problem is illustrated by the following example taken from [27, Rem. 3.4.24]. We have the co-initial full and non-full superdevelopments:

$$(\lambda x.\underline{(\lambda y.yx)I})z \to_\beta (\underline{\lambda x.Ix})z \to_\eta \underline{Iz} \to_\beta z \qquad \underline{(\lambda x.(\lambda y.yx)I)z} \to_\beta (\lambda y.yz)I$$

but to reduce the target of the latter to that of the former requires *two* superdevelopments.

▶ **Example 48.** The λ-calculus with explicit substitutions $\lambda\sigma$ [1] has the Z-property on closed terms. This is witnessed by the composition of first the function mapping a term to its $\to'$-normal form where $\to'$ denotes σ reduction, and next the full development function $\bullet$ contracting all *Beta*-redexes (Beta on its own is orthogonal). The proof is given in Fig. 4, where black ordinary arrows denote *Beta*-reductions, blue open arrows $\to'$-reductions, $\overline{t}$ the $\to'$-normal form of t, and $t^\bullet$ the result of subsequently applying the full-development function. For the result to hold, it suffices that

(Γ) $\to'$ is confluent and terminating [38, Exercise 3.6.3(i)];
(Δ) $\multimap\!\!\to$ has the triangle property for $\bullet$; and
(E) single $\multimap\!\!\to$-steps commute with $\to'$-reduction [38, Exercise 3.6.3(iii)].

▶ **Example 49.** We do not know whether Mints' λ-calculus with *restricted* η-expansion (such that no β-redexes are created) has the Z-property. The restriction hampers monotonicity.

[11] It coincides with the maximal multistep function since redex-*clusters* are *chains* [14, Defs. 4.31,4.47].

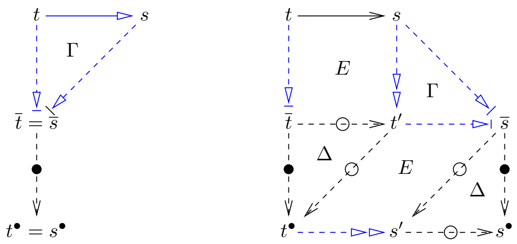


Figure 4 $\lambda\sigma$ has the Z-property.

4 Syntax-free developments

We first show in Sects. 4.1–4.3 that several classical rewrite results that are known for the classical syntactic notion of development[12] in term rewriting [38] and the λ-calculus [4] carry over to our syntax-free notion $-\!\circ\!\!\rightarrow$ of $\bullet$-development (Def. 7) defined for a bullet function $\bullet$ witnessing the Z-property. The diagrammatic proofs are obtained by *pasting with Zs*. Next, we investigate in Sect. 4.4 for the special case of orthogonal TRSs, under what conditions the syntactic and syntax-free notions of development coincide. Throughout we assume $\rightarrow$ has the Z-property for $\bullet$.

4.1 Hyper-Cofinality

We show $-\!\bullet\!\!\rightarrow$ is a best possible many-step strategy for $\rightarrow$ in that it is *hyper-cofinal* [38, Sect. 9.1.1]; in order-theoretic terms: starting from object a and always eventually performing a $-\!\bullet\!\!\rightarrow$-step eventually will yield a result greater than b, for any b greater than a. Observe first that $-\!\bullet\!\!\rightarrow$ *is* a many-step strategy since if $a -\!\bullet\!\!\rightarrow a^\bullet$ then by Def. 4 a is not in $\rightarrow$-normal form, so there is some step $a \rightarrow b$ from which we conclude to $a \twoheadrightarrow a^\bullet$ by the Z-property.

▶ **Theorem 50.** $-\!\bullet\!\!\rightarrow$ *is* hyper-cofinal *for* $\rightarrow$.

Proof. It suffices to show that, for a given step $a \rightarrow b$ and *maximal [38, below Def. 1.1.13] reduction* γ *of* $\rightarrow$,$-\!\bullet\!\!\rightarrow$*-steps which always eventually contains a* $-\!\bullet\!\!\rightarrow$*-step*, there is another such reduction δ from b eventually coinciding with it. By maximality, γ either ends in a

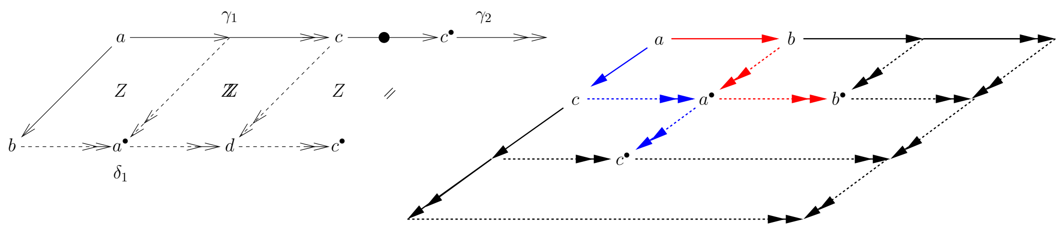


Figure 5 Hyper-cofinality of $-\!\bullet\!\!\rightarrow$ (left) and confluence of $\rightarrow$ (right), by tiling with Zs.

normal form c, or by the assumption ("always eventually") decomposes into a $\rightarrow$-reduction $\gamma_1 : a \twoheadrightarrow c$, followed by $c -\!\bullet\!\!\rightarrow c^\bullet$ followed by another such reduction γ_2 from $c^\bullet$ (see Fig. 5). Induction on the length of $a \twoheadrightarrow c$ and monotonicity of $\bullet$ give a d between c and $c^\bullet$ such that $\delta_1 : b \twoheadrightarrow d$. If c is a normal form, $c = d$ and we set $\delta := \delta_1$, else we compose δ from δ_1, $d \twoheadrightarrow c^\bullet$ and γ_2. ◀

[12] Developments go all the way back to *sequences of contractions on the parts* in [5], for the λI-calculus.

As a consequence [38, Sect. 9.1] $-\!\!\bullet\!\!\to$ is a *hyper-normalising* strategy, i.e. if an object reduces to a normal form then always eventually performing a $-\!\!\bullet\!\!\to$-step will reach it. For the λ-calculus $-\!\!\bullet\!\!\to$ is (weak-)head-normalising, since (weak-)head-normal forms are closed under reduction; Normalisation of $-\!\!\bullet\!\!\to$, i.e. of Gross–Knuth-reduction, was already noted in [18, Ex. 4.1].

4.2 Confluence

▶ **Lemma 51.** *$\to$ is confluent.*

Proof. Confluence can be established in several ways. We present three.

- By tiling the plane with Zs as displayed in Fig. 5 (formally by the Strip Lemma and [38, Prop. 1.1.10]). In Fig. 5 we have high-lighted the Zs for $a \to b$ and $a \to c$ in red and blue;
- Via Lem. 8, the angle property for $-\!\!\circ\!\!\to$ and [38, Prop. 1.1.11]; and
- Via Thm. 50, cofinality of $-\!\!\bullet\!\!\to$ and [38, Thm. 1.2.3(iv)]:[13]

$$\begin{aligned}
\twoheadleftarrow\cdot\twoheadrightarrow &\subseteq \twoheadrightarrow\cdot\twoheadleftarrow\!\!\bullet\!\!-\cdot-\!\!\bullet\!\!\twoheadrightarrow\cdot\twoheadleftarrow && \text{as } \twoheadrightarrow\subseteq -\!\!\bullet\!\!\twoheadrightarrow\cdot\twoheadleftarrow \text{ by cofinality of } -\!\!\bullet\!\!\to\\
&\subseteq \twoheadrightarrow\cdot-\!\!\bullet\!\!\twoheadrightarrow\cdot\twoheadleftarrow\!\!\bullet\!\!-\cdot\twoheadleftarrow && \text{as } -\!\!\bullet\!\!\to \text{ is deterministic hence confluent}\\
&\subseteq \twoheadrightarrow\cdot\twoheadrightarrow\cdot\twoheadleftarrow\cdot\twoheadleftarrow && \text{as } -\!\!\bullet\!\!\to \text{ is a many-step } \to\text{-strategy}\\
&\subseteq \twoheadrightarrow\cdot\twoheadleftarrow && \text{by transitivity of } \twoheadrightarrow
\end{aligned}$$ ◀

Since confluence is defined as the diamond property of the induced quasi-order, we have as a corollary that any rewrite system *admitting* the Z-property (Sect. 3.1) is confluent.

▶ Remark 52. Choosing an appropriate bullet function (cf. Sect. 1) can lead to remarkably short proofs of confluence via the Z-property. To wit, the confluence proofs for positive braids (by full swaps), self-distributivity (by full distribution),[14] and for orthogonal TRSs and the λ-calculus (by full superdevelopments)[15] are the shortest ones we know, in the same informal sense of "shortest" as was used by Takahashi on [37, p. 121] when she stated the proof of confluence of $\lambda\beta$ via the angle property was "perhaps the shortest". However, the proof via the Z-property is (a bit) shorter [22].

▶ Remark 53. Takahashi's confluence proof method [37, Sect. 1] for the λ-calculus can be viewed as being based on the angle property for developments. Although the Z and angle properties are equivalent (Lem. 8), her method is slightly more involved, conceptually and technically, as it involves (inductively) defining *both* the bullet function *and* developments (called $*$ respectively *parallel reduction* in [37]). Our approach does away with the latter; our $\bullet$-developments are *derived* from $\bullet$ in a syntax-free way; beware though that developments and $\bullet$-developments in general differ, cf. Sect. 4.4.

4.3 Recurrence

[36, Proposition 1] characterises the *recurrent* terms in CL (see Ex. 18) in terms of Gross–Knuth reduction. We recast this in a syntax-free way for $\to$ having the Z-property.

▶ **Definition 54.** *An object a is* $\to$-recurrent *if $a \to b$ entails $b \to a$ for all b. An object is* recurrent *if it is $\twoheadrightarrow$-recurrent.*

▶ **Proposition 55.** *If $\bullet$ is extensive, then a is recurrent iff $a^\bullet \twoheadrightarrow a$.*

---

[13] This generalises half of Staples' confluence method [38, Exercise 1.3.9].

[14] Confluence of self-distributivity is non-trivial. Currently no tool can prove it automatically; see problem 126 of `http://cops.uibk.ac.at/results/?y=2020-full-run&c=TRS`.

[15] Full developments involve a useless test for being a redex (Def. 42).

Proof. For the if-direction we show for all n, for all b, if $a \to^n b$ then $b \twoheadrightarrow a$, by induction on n. In the base case $a = b$ and we conclude by reflexivity of $\twoheadrightarrow$. In the induction step, we have $a \to^n c \to b$ for some object c, so $c \twoheadrightarrow a$ by the IH for $a \to^n c$. We conclude by composing $b \twoheadrightarrow c^\bullet$, which holds by the Z-property for $c \to b$, with $c^\bullet \twoheadrightarrow a^\bullet$, which holds by monotonicity of $\bullet$ for $c \twoheadrightarrow a$, and with $a^\bullet \twoheadrightarrow a$, which holds by assumption, to $b \twoheadrightarrow a$ as desired.

For the only–if-direction, we have $a \twoheadrightarrow a^\bullet$ by the assumption that $\bullet$ is extensive, hence $a^\bullet \twoheadrightarrow a$ by the assumption that a is recurrent, as desired. ◀

▶ Remark 56. This result was used and formalised by Felgenhauer for a study of fixed-point combinators in CL [10]. E.g., although it is simple to *see* $SII(SII)$ is recurrent, how to *prove* it in a simple way? By Proposition 55 it suffices to show that the result of a Gross–Knuth step reduces to it, i.e. that $I(SII)(I(SII)) \twoheadrightarrow SII(SII)$, which *is* simple to check.

4.4 Syntactic developments in orthogonal term rewriting

We investigate for orthogonal TRSs (cf. Sect. 3.3.1) the correspondence between the classical *syntactic* definition of a development and the *syntax-free* definition of $\bullet$-development (Def. 7) arising from taking as bullet function $\bullet$ the full development function that maps a term to the result of contracting all redex-patterns in it (Def. 19). This section is based on *permutation equivalence* via *residual* theory originating with [13], as presented in [38, Chs. 8 and 9]. We restrict to investigating the, non-trivial, correspondence for orthogonal TRSs hoping it can serve as a stepping stone for the same for more complex cases such as self-distributivity and the λ-calculus.

We first expand on the discrepancy between the syntactic and the syntax-free notions as observed in Ex. 9 (a *non-terminating* orthogonal TRS). Our first observation is that $\bullet$-developments are more encompassing than developments due to what are called *syntactic accidents* [17, p. 34], i.e. due to reductions yielding the same result despite not doing the same work, not being permutation equivalent. We show absence of syntactic accidents suffices.

▶ **Example 57.** For the *erasing* TRS with rules $a \to b \to c$ and $f(x) \to d$, we have $f(a)^\bullet := d$ and there is a $\bullet$-development from $f(a)$ to $f(c)$, but no such development. For the *collapsing* TRS with rules $g(x) \to h(x)$, $h(x) \to i(x)$ and $i(x) \to x$, we have $i(h(g(a)))^\bullet := i(h(a))$ and there is a $\bullet$-development from $i(h(g(a)))$ to $i(h(i(a)))$, but no such development.

▶ **Proposition 58.** *For orthogonal, terminating, non-collapsing, and non-erasing TRSs, developments and $\bullet$-developments coincide.*

Proof. We claim the assumptions guarantee the absence of syntactical accidents: if γ, δ are reductions from t to s then they are permutation equivalent $\gamma \simeq \delta$.[16] From the claim it follows that if $\gamma : t \twoheadrightarrow t^\bullet$ and $\delta : t \twoheadrightarrow s$ for some $\epsilon : s \twoheadrightarrow t^\bullet$, then $\gamma \simeq \delta \cdot \epsilon$. Therefore, decomposing δ as $\delta_1 \cdot \phi \cdot \delta_2$ for some step $\phi : t' \to s'$, we have $\gamma/\delta_1 : t' \twoheadrightarrow s$ and $\phi \lesssim \gamma/\delta_1$, which by non-erasingness entails that ϕ is among the redex-patterns in γ/δ_1.[17] Since this holds for each step, δ is a development of the set of all redex-patterns in t. The other implication follows from that every development from t can be completed into a complete development to $t^\bullet$.

[16] We employ the *projection* equivalence notation $\simeq$ from [38, Def. 8.7.21]. We freely employ results from that chapter, e.g. that permutation and projection equivalence coincide for orthogonal TRSs.

[17] This fails for erasing systems. For instance, the step $f(a) \to f(s)$ is not a development of the step $f(a) \to c$ in the TRS with rules $a \to b$ and $f(x) \to c$.

It remains to prove the claim, which we prove by contradiction assuming $\gamma \not\simeq \delta$. By residual theory, the peak γ, δ (where both have the same target, say u, by accident) can be completed by a valley comprising $\gamma' := \delta/\gamma$ and $\delta' := \gamma/\delta$ such that $\gamma \cdot \gamma' \simeq \delta \cdot \delta'$. At least one of γ', δ' must be non-empty, as otherwise γ, δ would be projection equivalent. But then the other must be non-empty as well, since otherwise we would have a reduction cycle on u contradicting the assumed termination. To see that $\gamma' \not\simeq \delta'$ note we may assume that γ, δ are standard, where a reduction is *standard* [13] if for each step in it the position of the contracted redex-pattern is in the redex-pattern of the first step after and left–outer of it [38, Definition 8.5.40]. W.l.o.g. we may assume γ, δ differ in their first steps and at least one of them contains a head-step, say γ contains head-step ϕ. Then δ doesn't, as otherwise their first steps would not differ by [15, Lemma 1]. We conclude $\gamma/\delta \not\simeq \delta/\gamma$ since the former contains a head-step as projection of a reduction γ containing a head-step over a reduction δ containing none, and the latter contains no head-step as projection of δ containing none over another reduction γ using the assumption that rules are non-collapsing. Applying the construction again, to the peak γ', δ' (where both have the same target again by accident) yields a valley comprising $\gamma'' := \delta'/\gamma'$ and $\delta'' := \gamma'/\delta'$ such that $\gamma' \cdot \gamma'' \simeq \delta' \cdot \delta''$ but $\gamma'' \not\simeq \delta''$. Repeating arbitrarily often yields an infinite reduction from t, contradicting termination. ◀

The three conditions in Prop. 58 are rather restrictive. We employ *labelling* [38, Sect. 8.4] to *turn* an arbitrary orthogonal term rewrite system into one satisfying them, and recover the result. We separate this into two phases, first turning a TRS into a *non-erasing* one by means of *memorising* the erased arguments,[18] and next lifting to a TRS that is also *terminating* and *non-collapsing* by means of the Hyland–Wadsworth labelling [38, Sect. 8.4.4].

▶ **Definition 59.** *The TRS* with memory $[\mathcal{T}]$ *of a TRS* $\mathcal{T}$ *has*

- *as signature the signature of* $\mathcal{T}$ *extended with a binary symbol* $[\,,\,]$;
- *as rules* $\varrho_{\hat{\ell}} : \hat{\ell} \to [r, \vec{x}]$ *for some* $\mathcal{T}$*-rule* $\varrho : \ell \to r$*, where* $\hat{\ell}$ *is such that projecting all occurrences of* $[\,,\,]$ *in it on their first argument yields* ℓ*, but these are not at the root, do not have a variable as first argument, and all have fresh variables (uniquely determined by their position) as second arguments. Here* $\vec{x}$ *is the list (unique for* $\hat{\ell}$*) of all variables in* $\hat{\ell}$ *not in* r*,* $[t]$ *denotes* t*, and* $[t, x\vec{y}]$ *denotes* $[t, [x, \vec{y}]]$.

▶ **Example 60.** The TRS with memory for the rules $f(a) \to b$ and $f(x) \to b$, yields infinitely many rules $f(a) \to b$, $f([a, x]) \to [b, x]$, $f([[a, y], x]) \to [b, xy], \ldots$ for the first rule, and the single rule $f(x) \to [b, x]$ for the second one.

▶ **Lemma 61.** *If* $\mathcal{T}$ *is orthogonal, then* $[\mathcal{T}]$ *is orthogonal and non-erasing. The identity map induces a* rewrite labelling *[38, Def. 8.4.5(ii)] of* $\mathcal{T}$ *into* $[\mathcal{T}]$.

▶ **Example 62.** Memorising overcomes erasingness. With memory $f(a)^\bullet := [d, b]$ for the first TRS in Ex. 57, so the $\bullet$-developments from $f(a)$ are the initial prefixes of $f(a) \to f(b) \to [d, b]$ and $f(a) \to [d, a] \to [d, b]$. There is now *no* $\bullet$-development from $f(a)$ to $f(c)$.

To overcome also non-termination and (as a side-effect) collapsingness, we employ the Hyland–Wadsworth labelling [17, 4, 19, 38] $\mathcal{T}^\omega$ of a TRS $\mathcal{T}$. The idea of that labelling is to approximate arbitrary (possibly infinite) $\mathcal{T}$-reductions with arbitrary precision, where precision is measured via the causal length of reductions. Technically, this is achieved by labelling edges[19] in terms with their *creation depth* (a natural number) in such a way that any unlabelled reduction can be lifted to one having some *bounded* creation depth n, and such that the corresponding subsystem $\mathcal{T}^n$ of $\mathcal{T}^\omega$ is *terminating* and confluent.

[18] A technique going back to Nederpelt's *scars* [24, p. 90].

[19] To make sure that every redex-*pattern* contains at least one edge, we replace any function symbol f with a pair f'–f with f' a fresh unary function symbol.

▶ **Definition 63.** *The* Hyland–Wadsworth *(HW) labelling of a TRS $\mathcal{T}$ is the TRS $\mathcal{T}^\omega$*

- *having as signature all natural numbers (*labels*) and for every f of $\mathcal{T}$ both f and a fresh copy f' of it, with all symbols not in $\mathcal{T}$ having arity 1;*
- *having as rules $\varrho_{\hat{\ell}} : \hat{\ell} \to (r')^n$ for every rule $\varrho : \ell \to r$, where $\hat{\ell}$ is such that between any two non-labels there is at least one label, n is the maximum value of all labels in $\hat{\ell}$ plus one, and removing all yields ℓ', where priming and natural-number-labelling are defined by:*

$$\begin{array}{rclcrcl} x' & := & x & \qquad & x^n & := & n(x) \\ f(\vec{t})' & := & f'(f(\vec{t'})) & & g(\vec{s})^n & := & n(g(\vec{s^n})) \end{array}$$

$\mathcal{T}^n$ is the restriction of $\mathcal{T}^\omega$ to rules whose lhss have labels $< n$.

▶ **Example 64.** We illustrate the saturation process of the HW-labelling on a rule with a single-function-symbol left-hand side (cf. footnote 19), The Hyland–Wadsworth labelling of the TRS with rule $f(x) \to x$ has the infinitely many rules $f'(0(f(x))) \to 1(x)$, $f'(1(f(x))) \to 2(x)$, $\dots$, $f'(0(0(f(x)))) \to 1(x)$, $f'(1(0(f(x)))) \to 2(x)$, $f'(0(1(f(x)))) \to 2(x)$, $\dots$. Note the original rule was collapsing, but its HW-labellings are not.

Hyland–Wadsworth labelling preserves orthogonality and is sound in that reductions can be lifted, however with ever increasing labels so bounding them yields termination.

▶ **Lemma 65.**
- *$\mathcal{T}^\omega$ and $\mathcal{T}^n$ are (left/right) linear and/or orthogonal iff $\mathcal{T}$ is;*
- *mapping every term t to $(t')^0$ gives a rewrite labelling of $\mathcal{T}$; and*
- *the restriction $\mathcal{T}^n$ of $\mathcal{T}^\omega$ to lhs with labels $< n$ is terminating [19].*

▶ **Example 66.** To see how the HW-labelling avoids syntactical accidents for collapsing rules consider the reduction $f(f(x)) \to f(x)$ for rule $f(x) \to x$. It lifts differently depending on which redex-pattern is contracted:

$$0(f'(0(f(0(f'(0(f(0(x))))))))) \to 0(1(0(f'(0(f(0(x)))))))$$

$$0(f'(0(f(0(f'(0(f(0(x))))))))) \to 0(f'(0(f(0(1(0(x)))))))$$

Along the lines of the proof of Prop. 58 we show *all* syntactical accidents are avoided. The lemma expresses an *invertibility* property (cf. [38, Thm. 8.4.20]): given the target term of a $\mathcal{T}^\omega$ reduction, the reduction can be reconstructed *up to permutation equivalence.*

▶ **Lemma 67.** *If γ, δ are co-initial and co-final $\mathcal{T}^\omega$ reductions, then $\gamma \simeq \delta$.*

▶ **Theorem 68.** *Developments and $\bullet$-developments coincide in $[\mathcal{T}]^\omega$, if $\mathcal{T}$ is orthogonal.*

Proof. Since $\mathcal{T}$ is orthogonal by assumption, so is $[\mathcal{T}]$ by Lemma 61. Therefore, by Lemma 67: if $\gamma, \delta : a \twoheadrightarrow b$ are $[\mathcal{T}]^\omega$-reductions then $\gamma \simeq \delta$. It follows that if $\gamma : t \multimap\!\!\!\twoheadrightarrow t^\bullet$ and $\delta : t \twoheadrightarrow s$ for some $\epsilon : s \twoheadrightarrow t^\bullet$, then $\gamma \simeq \delta \cdot \epsilon$. Therefore, decomposing δ as $\delta_1 \cdot \phi \cdot \delta_2$ for some step $\phi : t' \to s'$, we have $\gamma/\delta_1 : t' \multimap\!\!\!\twoheadrightarrow s$ and $\phi \lesssim \gamma/\delta_1$, which by non-erasingness of $[\mathcal{T}]$ hence of $[\mathcal{T}]^\omega$ entails that ϕ is among the redex-patterns in γ/δ_1. Since this holds for each step, δ is a development of the set of all redex-patterns in t. The other implication follows from that every development from t can be completed into a complete development to $t^\bullet$. ◀

5 Conclusion

We have presented the Z-property and illustrated its flexibility, showing it applies to various rewrite systems to yield short proofs for classical results such as confluence and normalisation. Their proofs are based on a syntax-free version of the classical notion of development. We hope and expect more results can be factored in this way. We showed it coincides for orthogonal TRSs with the syntactic notion of development if syntactical accidents are absent (Prop. 58, Lem. 67) and hope that this invertibility result and its novel proof method extend to more complex systems, e.g. λ-calculus or self-distributivity.

References

1 M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In F.E. Allen, editor, *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 31–46. ACM Press, 1990. doi:10.1145/96709.96712.

2 P. Aczel. A general Church–Rosser theorem, 1978. corrections http://www.ens-lyon.fr/LIP/REWRITING/MISC/AGRT_corrections.pdf. URL: http://www.ens-lyon.fr/LIP/REWRITING/MISC/AGeneralChurch-RosserTheorem.pdf.

3 F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

4 H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 2nd revised edition, 1984.

5 A. Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39:472–482, 1936. doi:10.2307/1989762.

6 P. Dehornoy. *Braids and Self-Distributivity*, volume 192 of *Progress in Mathematics*. Birkhäuser, 2000.

7 P. Dehornoy and V. van Oostrom. Z; proving confluence by monotonic single-step upperbound functions. In *Logical Models of Reasoning and Computation (LMRC-08), Moscow*, 2008. URL: http://cl-informatik.uibk.ac.at/users/vincent/research/publication/talk/lmrc060508.pdf.

8 J. Endrullis, C. Grabmayer, D. Hendriks, J.W. Klop, and V. van Oostrom. Infinitary term rewriting for weakly orthogonal systems: Properties and counterexamples. *Logical Methods in Computer Science*, 10(2), 2014. doi:10.2168/LMCS-10(2:7)2014.

9 B. Felgenhauer, J. Nagele, V. van Oostrom, and C. Sternagel. The Z property. *Arch. Formal Proofs*, 2016, 2016. URL: https://www.isa-afp.org/entries/Rewriting_Z.shtml.

10 B. Felgenhaurer. Personal communication, 2017.

11 Y. Honda, K. Nakazawa, and K. Fujita. Confluence proofs of lambda-mu-calculi by Z theorem. *Studia Logica*, 2021. doi:10.1007/s11225-020-09931-0.

12 G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, 1980. doi:10.1145/322217.322230.

13 G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems, Part I + II. In J.L. Lassez and G.D. Plotkin, editors, *Computational Logic – Essays in Honor of Alan Robinson*, pages 395–443, Cambridge MA, 1991. MIT Press. Update of: Call-by-need computations in non-ambiguous linear term rewriting systems, 1979.

14 J. Ketema, J.W. Klop, and V. van Oostrom. Vicious circles in rewriting systems. Technical Report E0427, Centrum voor Wiskunde en Informatica, December 2004. URL: https://ir.cwi.nl/pub/11022/11022D.pdf.

15 J.W. Klop, V. van Oostrom, and F. van Raamsdonk. Reduction strategies and acyclicity. In H. Comon-Lundh, C. Kirchner, and H. Kirchner, editors, *Rewriting, Computation and Proof, Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday*, volume 4600 of *Lecture Notes in Computer Science*, pages 89–112. Springer, 2007. doi:10.1007/978-3-540-73147-4_5.

16 Y. Komori, N. Matsuda, and F. Yamakawa. A simplified proof of the Church–Rosser theorem. *Studia Logica: An International Journal for Symbolic Logic*, 102(1):175–183, 2014. doi:10.1007/s11225-013-9470-y.

17 J.-J. Lévy. *Réductions correctes et optimales dans le λ-calcul*. Thèse de doctorat d'état, Université Paris VII, 1978. URL: http://pauillac.inria.fr/~levy/pubs/78phd.pdf.

18 R. Loader. Notes on simply typed lambda calculus. ECS-LFCS- 98-381, Laboratory for Foundations of Computer Science, The University of Edinburgh, 1998. URL: http://www.lfcs.inf.ed.ac.uk/reports/98/ECS-LFCS-98-381/.

19 L. Maranget. *La stratégie paresseuse*. Thèse de doctorat, Université Paris 7, 1992.

20 R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theor. Comput. Sci.*, 19(1):3–29, 1998. `doi:10.1016/S0304-3975(97)00143-6`.

21 P.-A. Melliès. Axiomatic rewriting theory VI residual theory revisited. In S. Tison, editor, *Rewriting Techniques and Applications, 13th International Conference, RTA 2002, Copenhagen, Denmark, July 22–24, 2002, Proceedings*, volume 2378 of *Lecture Notes in Computer Science*, pages 24–50. Springer, 2002. `doi:10.1007/3-540-45610-4_4`.

22 J. Nagele, V. van Oostrom, and C Sternagel. A short mechanized proof of the Church–Rosser theorem by the Z-property for the $\lambda\beta$-calculus in nominal Isabelle. In *5th International Workshop on Confluence, IWC 2016, Obergurgl, Austria, September 8–9, 2016, Online Proceedings*, 1016. URL: `http://www.csl.sri.com/users/tiwari/iwc2016/iwc2016.pdf`.

23 K. Nakazawa and K. Fujita. Compositional Z: Confluence proofs for permutative conversion. *Studia Logica: An International Journal for Symbolic Logic*, 104(6):1205–1224, 2016. `doi:10.1007/s11225-016-9673-0`.

24 R.P. Nederpelt. *Strong Normalization in a Typed Lambda Calculus with Lambda Structured Types*. PhD thesis, Technische Hogeschool Eindhoven, June 1973.

25 M.H.A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43:223–243, 1942. `doi:10.2307/2269299`.

26 T. Nipkow. Orthogonal higher-order rewrite systems are confluent. In M. Bezem and J.F. Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 306–317. Springer, 1993. `doi:10.1007/BFb0037114`.

27 V. van Oostrom. *Confluence for Abstract and Higher-Order Rewriting*. PhD thesis, Vrije Universiteit, Amsterdam, March 1994. URL: `https://research.vu.nl/files/62846778/complete%20dissertation.pdf`.

28 V. van Oostrom. Finite family developments. In H. Comon, editor, *Rewriting Techniques and Applications, 8th International Conference, RTA-97, Sitges, Spain, June 2-5, 1997, Proceedings*, volume 1232 of *Lecture Notes in Computer Science*, pages 308–322. Springer, 1997. `doi:10.1007/3-540-62950-5_80`.

29 V. van Oostrom. Reduce to the max, 1999. Unpublished manuscript. URL: `http://cl-informatik.uibk.ac.at/users/vincent/research/publication/pdf/max.pdf`.

30 V. van Oostrom. Random descent. In *RTA*, volume 4533 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 2007. `doi:10.1007/978-3-540-73449-9_24`.

31 V. van Oostrom. Abstract rewriting. In A. Middeldorp, editor, *3rd International School on Rewriting, ISR 2008, Obergurgl, Austria, July 21–26, 2008*, 2008. Z-property in part 2 of the slides. URL: `http://cl-informatik.uibk.ac.at/isr-2008/html/b.4.html`.

32 V. van Oostrom. Confluence by decreasing diagrams; converted. In A. 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 306–320. Springer, 2008. `doi:10.1007/978-3-540-70590-1_21`.

33 V. van Oostrom and Y. Toyama. Normalisation by Random Descent. In *FSCD*, volume 52 of *LIPIcs*, pages 32:1–32:18. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. `doi:10.4230/LIPIcs.FSCD.2016.32`.

34 V. van Oostrom and Y. Venema. Term rewriting systems I and II. In *10th European Summer School in Logic, Language and Information, ESSLLI 98, Saarbrücken, Germany, August 17—28, 1998*, 2008. Course notes on braids in part I of the course. URL: `http://cl-informatik.uibk.ac.at/users/vincent/research/publication/pdf/braids.pdf`.

35 F. van Raamsdonk. *Confluence and Normalisation for Higher-Order Rewriting*. PhD thesis, Vrije Universiteit Amsterdam, 1996. URL: `https://research.vu.nl/files/62847150/complete%20dissertation.pdf`.

36 R. Statman. There is no hyperrecurrent s,k combinator. Research Report 91-1332, Department of Mathematics, Carnegie Mellon University, Pittsburg, PA 15213, June 1991. URL: `http://shelf2.library.cmu.edu/Tech/53922203.pdf`.

37 M. Takahashi. Parallel reductions in λ-calculus. *Information and Computation*, 118:120–127, 1995. `doi:10.1006/inco.1995.1057`.

38 Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

A Proofs omitted from the main text

Proofs of second and third items of Lem. 8.

- Assume $\rightarrow$ has the Z-property for bullet function $\bullet$. Define $\star$ to be $\bullet$ updated to map each object that is not the source of some step, to itself.
  To see that $\star$ is extensive, we distinguish cases on whether a is the source of some step or not. If it is, say $a \rightarrow b$, then $b \twoheadrightarrow a^\bullet \twoheadrightarrow b^\bullet$ by the Z-property for $\bullet$. Hence $a \twoheadrightarrow a^\bullet = a^\star$ by composition and definition of $\star$. If it is not, then $a \twoheadrightarrow a^\star = a$ by reflexivity and definition of $\star$.
  To see that $\rightarrow$ has the Z-property for $\star$, suppose $a \rightarrow b$. By the Z-property for $\bullet$ and by definition of $\star$, then $b \twoheadrightarrow a^\bullet = a^\star \twoheadrightarrow b^\bullet$. The result follows if, as we claim, $b^\bullet = b^\star$. That follows by noting that, by definition of $\star$, the only way in which $b^\bullet = b^\star$ could fail to hold, is if b were not the source of *some* step. But then the above reduction collapses to $b = a^\bullet = a^\star = b^\bullet$ and we conclude since $b = b^\star$.
- We only check the additional conditions on either side w.r.t. the first item.
  For the only–if-direction, suppose $\rightarrow$ has the Z-property for an extensive $\bullet$. To show $a \multimap a^\bullet$, distinguish cases on whether there is some $\rightarrow$-step from a or not. If there is, say $a \rightarrow b$ then by the Z-property, $a \twoheadrightarrow b^\bullet \twoheadrightarrow a^\bullet$. If there is no $\rightarrow$-step from a, then extensivity of $\bullet$ entails $a = a^\bullet$. In either case, $a \twoheadrightarrow a^\bullet \twoheadrightarrow a^\bullet$ by reflexivity of $\twoheadrightarrow$, so $a \multimap a^\bullet$ by definition of $\multimap$. ◀

Proof of Lem. 34.

(Sequentialisation) The proof is by induction on t. If t is a variable, then $ts = t\langle s]$ and we conclude by reflexivity. Otherwise, t has shape t_1t_2 and we conclude using the IH to $ts = t_1t_2s \rightarrow t_1(t_2s) \twoheadrightarrow t_1t_2\langle s] = t\langle s]$ from which the statement follows by transitivity.

(Compatible) We show the stronger fact that single steps in either t or s are preserved, by induction on t, which suffices by transitivity of $\twoheadrightarrow$. If t is a variable x, then $s \rightarrow s'$ and $t\langle s] = xs \rightarrow xs' = t\langle s']$ by compatibility of reduction. If $t = t_1t_2$, we distinguish cases on where the step takes place:

- If the step takes place at the root of t, then $t = t_{11}t_{12}t_2 \rightarrow t_{11}(t_{12}t_2) = t'$ and we conclude by unfolding the definition of right-substitution twice on both sides to $t\langle s] = t_{11}t_{12}t_2\langle s] \rightarrow t_{11}(t_{12}t_2\langle s]) = t'\langle s]$;
- If the step takes place in t_1, then $t\langle s] = t_1t_2\langle s] \rightarrow t_1't_2\langle s] = t'\langle s]$ by compatibility of reduction;
- If the step takes place in t_2, then $t\langle s] = t_1t_2\langle s] \rightarrow t_1t_2'\langle s] = t'\langle s]$ by the IH and compatibility of reduction;
- If the step takes place in s, then $t\langle s] = t_1t_2\langle s] \rightarrow t_1t_2\langle s'] = t\langle s']$ by the IH and compatibility of reduction.

(Substitution) The statement is shown by induction on t. If t is a variable, say x then $t\langle s]\langle r] = xs\langle r] = t\langle s\langle r]]$ by unfolding the definition of right-substitution. If t has shape t_1t_2, then $t\langle s]\langle r] = t_1t_2\langle s]\langle r] = t_1t_2\langle s\langle r]] = t\langle s\langle r]]$ by unfolding the definition of right-substitution and the IH. ◀

Proof of Lem. 35.

(Extensive) By induction on t. If t is a variable, then $t = t^\bullet$ and we conclude by reflexivity of $\twoheadrightarrow$. Otherwise t has shape t_1t_2, and we conclude by (Sequentialisation), the IH twice, (Compatible), and definition to $t_1t_2 \twoheadrightarrow t_1\langle t_2] \twoheadrightarrow t_1^\bullet\langle t_2^\bullet] = (t_1t_2)^\bullet$;

(Rhs) By (Sequentialization) twice and (Substitution) we conclude $t^\bullet(s^\bullet r^\bullet) \twoheadrightarrow t^\bullet\langle s^\bullet\langle r^\bullet]] = t^\bullet\langle s^\bullet]\langle r^\bullet] = (tsr)^\bullet$;

(Z) As $\bullet$ maps to normal forms, we show a strengthening of the Z-property, $s \twoheadrightarrow t^\bullet = s^\bullet$, for all steps $t \to s$, by induction and cases on t.

If t is a variable, then the statement holds vacuously since the term then does not allow any step. Otherwise, t has shape t_1t_2 and we distinguish cases on the position of the step.

- If the step takes place at the root, then $t = (t_{11}t_{12})t_2 \to t_{11}(t_{12}t_2) = s$, and we conclude using (extensive), (Rhs), the definition, and (Substitution) to $t_{11}(t_{12}t_2) \twoheadrightarrow t_{11}^\bullet(t_{12}^\bullet t_2^\bullet) \twoheadrightarrow (t_{11}t_{12}t_2)^\bullet = t_{11}^\bullet\langle t_{12}^\bullet]\langle t_2^\bullet] = t_{11}^\bullet\langle t_{12}^\bullet\langle t_2^\bullet]]$;
- If the step takes place in t_1, say $t_1 \to s_1$, then we conclude using the IH, (Extensive), (Sequentialisation) and definition to $s_1t_2 \twoheadrightarrow t_1^\bullet t_2^\bullet \twoheadrightarrow (t_1t_2)^\bullet = t_1^\bullet\langle t_2^\bullet] = s_1^\bullet\langle t_2^\bullet] = (s_1t_2)^\bullet$.
- If the step takes place in t_2 we proceed as in the previous item. ◀

Proof of Lem. 40. Both items can we proven by induction on t or via the alternative definition of uniform distribution by means of substitution as given in the main text. We give samples of both:

(Sequentialisation) For variables $xs = x[s]$, and for applications $t_1t_2s \to t_1s(t_2s) \twoheadrightarrow t_1[s]t_2[s] = (t_1t_2)[s]$, as $t_is \twoheadrightarrow t_i[s]$ by the IH;

(Compatible) $t[s] = t^\sigma$ for the substitution σ mapping x to xs, and $t'[s'] = t'^{\sigma'}$ for σ' mapping x to xs'. Hence if $t \twoheadrightarrow t'$ and $s \twoheadrightarrow s'$ then $\sigma \twoheadrightarrow \sigma'$, hence $t^\sigma \twoheadrightarrow t'^{\sigma'}$ by compatibility of rewriting with substitution; and

(Substitution) For variables $x[s][r] = (xs)[r] = xrs[r] \twoheadrightarrow x[r][s[r]]$ by Sequentialisation twice, and for applications $(t_1t_2)[s][r] = t_1[s][r]t_2[s][r] \twoheadrightarrow t_1[r][s[r]]t_2[r][s[r]] = (t_1t_2)[r][s[r]]$ by the induction hypothesis twice. ◀

Proof of Lem. 41. The items are proven by induction on t.

(Extensive) For variables $x = x^\bullet$, and for applications $ts \twoheadrightarrow t[s] \twoheadrightarrow t^\bullet[s^\bullet] = (ts)^\bullet$ by (Sequentialisation) first and then (Compatible) using the IH twice;

(Z) We distinguish cases on whether the step is a head step or not.

- Suppose the step is a head step, so has shape $tsr \to tr(sr)$. Then $tr(sr) \twoheadrightarrow t[r]s[r] = (ts)[r] \twoheadrightarrow (ts)^\bullet[r^\bullet] = (tsr)^\bullet$ by (Sequentialisation) and (Extensive), twice. Monotonicity of $\bullet$ holds by $(tsr)^\bullet = t^\bullet[s^\bullet][r^\bullet] \twoheadrightarrow t^\bullet[r^\bullet][s^\bullet[r^\bullet]] = (tr(sr))^\bullet$ using (Substitution).
- If $t_1t_2 \to s_2s_2$ because $t_i \to s_i$ and $t_{3-i} = s_{3-i}$ for some $i \in \{1,2\}$, then $s_j \twoheadrightarrow t_j^\bullet \twoheadrightarrow s_j^\bullet$ for $j \in \{1,2\}$, either by the IH, or (Extensive) and reflexivity. Using that, (Sequentialisation), and (Substitution) $s_1s_2 \twoheadrightarrow s_1[s_2] \twoheadrightarrow t_1^\bullet[t_2^\bullet] = (t_1t_2)^\bullet \twoheadrightarrow s_1^\bullet[s_2^\bullet] = (s_1s_2)^\bullet$. ◀

Proof of Lem. 61. Orthogonality is preserved since brackets are only inserted between original function symbols, so overlapping $[\mathcal{T}]$-redex-patterns are mapped to overlapping $\mathcal{T}$-patterns by projecting brackets on their first arguments. That $[\mathcal{T}]$ is non-erasing holds per construction.[20]

The second part holds per construction of saturating left-hand sides of rules with memory. ◀

[20] if $\mathcal{T}$ is orthogonal and *right-linear*, then $[\mathcal{T}]$ is linear, so has random descent [30]: all reductions to a normal form have the same length.

Proof of Lem. 65. For the first item first note that its only–if-direction requires $n > 0$ as otherwise $\mathcal{T}^n$ has *no* rules. Then, all (priming, labelling) operations for obtaining the rules of $\mathcal{T}^\omega$ from those of $\mathcal{T}$ are linear (only unary function symbols are added/removed) and redex-patterns overlapping in $\mathcal{T}^\omega$ still do so after removing labels and collapsing f'–f-pairs to f. $\mathcal{T}^n$ being a sub-system of $\mathcal{T}^\omega$ the properties are preserved.

The second item holds per construction of the rules with both left- and right-hand sides being of shape t' in which labels are inserted, for some t. Note that we also have the structural properties that reachable terms have at least one label between any two non-labels and removing all labels yields a term of shape s' for some s.

Maranget [19] shows termination in the third item is a consequence of RPO, for the greater–than relation on labels, which is well-founded by the assumption that labels $< n$. Instead of basing ourselves on RPO, we can also give a direct inductive proof of termination in the style of van Daalen [17, 4]. In particular, we specialise the higher-order approach of [28] to first-order term rewriting. The proof is based on the so-called *RHS* lemma [28, Lemma 8][21] stating that a term rewrite system is terminating iff r^σ is terminating for every rhs r of a rule and terminating substitution σ. The only–if-direction of the RHS-lemma being trivial, to see the if-direction holds note that if there were a non-terminating term then there would be one of minimal size which then would have shape $f(\vec{t})$ with all $\vec{t}$ terminating by minimality. Hence an infinite reduction from it would have shape $f(\vec{t}) \twoheadrightarrow f(\vec{s}) = \ell^\sigma \to r^\sigma \twoheadrightarrow \dots$ for some rule $\ell \to r$, substitution σ, and terms $\vec{s}$ such that $t_i \twoheadrightarrow s_i$ for all i. This is impossible as r^σ is terminating by assumption since σ is terminating as it assigns subterms of the $\vec{s}$ to variables[22] and each s_i is terminating as reduct of t_i.

To establish the assumption of the RHS lemma for $\mathcal{T}^n$ we prove the more general claim[23] that $(t^m)^\sigma$ *is terminating for every* $m \leq n$*, term* t *over (primed) symbols in* $\mathcal{T}$*, and terminating substitution* σ. This suffices as per construction of $\mathcal{T}^n$ rhss of rules have this shape since labels in lhss are $< n$.[24] The proof of the claim is by induction on the pair (m, t) ordered by, in lexicographic order, the greater-than-or-equal order and the subterm order, and by distinguishing cases on the shape of t.

- If t is a variable, then $(x^m)^\sigma := m(x^\sigma)$ and we conclude by the assumption that σ is terminating, since the head symbol m is not affected by any step per construction of $\mathcal{T}^\omega$; labels occur in lhss only between (possibly primed) $\mathcal{T}$-symbols.
- Otherwise t has shape $f(\vec{t})$ for some (possibly primed) $\mathcal{T}$-symbol. Since each $(t_i^m)^\sigma$ is terminating by the IH, which applies by a decrease in the second component of the pair, a hypothetical infinite reduction from $(t^m)^\sigma$ must then contain a *head*-step, i.e. have shape

  $$m(f((\vec{t^m})^\sigma)) \twoheadrightarrow m(f(\vec{s})) = m(\ell^\tau) \to m((r^k)^\tau) \twoheadrightarrow \dots$$

  for some $\mathcal{T}^n$ rule of shape $\ell \to r^k$ with k the maximum of the labels in ℓ plus one, substitution τ and r a term over (possibly primed) $\mathcal{T}$-symbols, and terms $\vec{s}$ such that $(t_i^m)^\sigma \twoheadrightarrow s_i$ for each i. This is impossible as $(r^k)^\tau$ is terminating by the IH, which applies by a decrease in the first component of the pair: $m < k$ because $(t_i^m)^\sigma \twoheadrightarrow s_i$ guarantees

[21] Despite being intuitive and easy to prove the right-hand side lemma *is* informative: it would already fail for first-order TRSs if left-hand sides of rules were allowed to be single variables, consider the "rule" $x \to x$, and for higher-order TRSs it would fail if non-pattern-lhss were allowed [28].
[22] Here we use that left-hand sides of term rewrite rules are not single variables.
[23] To enable induction on terms; rhss of rules are not closed (as rhss!) under subterms in general.
[24] Although *terms* of $\mathcal{T}^n$ may contain labels $> n$, these need not be taken into consideration here. They have been "filtered-out" already by means of the RHS lemma so to speak, since labels $> n$ do not occur in the *rules* of $\mathcal{T}^n$.

that m is the head symbol of each s_i, and per construction of $\mathcal{T}^\omega$ the lhs of any rule applicable to $f(\vec{s})$ contains the labels directly below f; in fact f must be a (unary) primed symbol having a corresponding unprimed symbol (in $\mathcal{T}$) below it. That τ is terminating follows from that it assigns subterms of the s_i to variables, which are reducts of the $(t_i^m)^\sigma$. ◀

Proof of Lem. 67. We proceed as in the proof of Proposition 58, here stressing the similarity of structure and referring to that proof for details. We show that for all natural numbers n, for all $\mathcal{T}^n$ reductions $\gamma, \delta : t \twoheadrightarrow s$ we have $\gamma \simeq \delta$ by induction on t ordered by the union of $\leftarrow$ and the sub-term relation, well-founded since $\mathcal{T}^n$ is terminating. This suffices since any pair of $\mathcal{T}^\omega$-reductions is a pair of $\mathcal{T}^n$-reductions (take n greater than all labels occurring in the redex-patterns contracted in γ, δ), and $\mathcal{T}^n$-projection equivalence entails $\mathcal{T}^\omega$-projection equivalence.

Suppose γ, δ were minimal such that $\gamma \not\simeq \delta$. By residual theory, the peak γ, δ can be completed by a valley comprising $\gamma' := \delta/\gamma$ and $\delta' := \gamma/\delta$ such that $\gamma \cdot \gamma' \simeq \delta \cdot \delta'$. By assumption, at least one of γ', δ' must be non-empty. We may assume that γ, δ are standard, and by minimality that they don't have the same first step (one may be empty), and at least one of them, say w.l.o.g. γ, contains a head step. Since the system is orthogonal, [15, Lemma 1] yields then that δ does not contain a head step. Hence $\gamma/\delta \not\simeq \delta/\gamma$ since γ/δ contains a head step as projection of a reduction γ containing a head step over a reduction δ containing none, and δ/γ contains no head step as projection of δ containing none over another reduction γ, using that $\mathcal{T}^n$-rules are non-collapsing. Their sources being $\to^+$-reachable from t, $\gamma/\delta, \delta/\gamma$ contradicts minimality of γ, δ. ◀

Recursion and Sequentiality in Categories of Sheaves

Cristina Matache
University of Oxford, UK

Sean Moss
University of Oxford, UK

Sam Staton
University of Oxford, UK

Abstract

We present a fully abstract model of a call-by-value language with higher-order functions, recursion and natural numbers, as an exponential ideal in a topos. Our model is inspired by the fully abstract models of O'Hearn, Riecke and Sandholm, and Marz and Streicher. In contrast with semantics based on cpo's, we treat recursion as just one feature in a model built by combining a choice of modular components.

2012 ACM Subject Classification Theory of computation → Denotational semantics; Theory of computation → Categorical semantics

Keywords and phrases Denotational semantics, Full abstraction, Recursion, Sheaf toposes, CPOs

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.25

Funding *Cristina Matache*: Research supported by an EPSRC studentship and Balliol College and Clarendon Fund scholarships.
Sean Moss: Research supported by a Junior Research Fellowship at University College, Oxford.
Sam Staton: Research supported by a Royal Society University Research Fellowship and the ERC BLAST grant.

1 Introduction

This paper is about building denotational models of programming languages with recursion by using categories of sheaves. The naive idea of denotational semantics is to interpret every type A as a set of values $[\![A]\!]$, every typing context Γ as a set of environments $[\![\Gamma]\!]$, and every term $\Gamma \vdash t : A$ as a partial function $[\![t]\!] : [\![\Gamma]\!] \rightharpoonup [\![A]\!]$, so that composing terms corresponds to composing functions. A more general approach says that a "denotational model" is a category with enough structure, such as a category of sets, so that we regard $[\![\Gamma]\!]$ and $[\![A]\!]$ as objects of that category, and $[\![t]\!]$ as a morphism. In our work here, we work in various categories of sheaves, so that $[\![\Gamma]\!]$ and $[\![A]\!]$ are sheaves, which is not far from the naive set-theoretic idea because categories of sheaves are often regarded as models of intuitionistic set theory. As we will explain, each category of sheaves is captured by a small site, and by combining or comparing sites we can combine and compare different denotational models of programming languages.

We illustrate this by combining sites to give a fully abstract model of a call-by-value PCF. Full abstraction means that two terms t, u are interpreted as equal functions ($[\![t]\!] = [\![u]\!]$) if and only if they are contextually equivalent. In PCF, which is a simple functional language, the main challenge for full abstraction is to capture the fact that PCF is sequential, in that it does not have any primitives for parallelism.

Our model is inspired by earlier models that were *not* explicitly sheaf-theoretic [36, 39, 46]. Our fully abstract model is built by combining many different sites which include one for recursion and that happen to include sites that will turn out to give full definability with truncated natural numbers. Overall, this truncated full definability can be used to prove full abstraction of the model.

Although the focus of this paper is on a simple PCF-like language, a broader agenda is to combine this analysis of recursion and sequentiality with recent sheaf-based models for other phenomena, including concurrency (e.g. [2]), differentiable programming [42, 18], probabilistic programming [16], quantum programming [27] and homotopy type theory [1]. The broader context, then, is to use sheaf-based constructions as a principled approach to building sophisticated models of increasingly elaborate languages.

If the reader is familiar with synthetic domain theory, they may regard the contribution of this paper as an account of full abstraction in that tradition: at a high level we are merging the sheaf model of [14] with the Kripke model of [36], via [9]. We give a survey of synthetic domain theory in §8.2.

We now introduce the key ideas of our paper: to consider a general theory of "normal" models of PCF (§1.1) and then to build a fully abstract one by combining certain sites (§1.2).

1.1 Normal models of PCF

The key general definition of our paper is that of "normal model" (Definition 4.1). This has three components: a sheaf category; it has a well-behaved notion of partial function; and it supports recursion. We now discuss these three components. We motivate with the example of the extended vertical natural numbers: the linear order $\mathrm{V} = \{0 \leq 1 \leq \cdots \leq n \leq \ldots \infty\}$. It is informally an interpretation of the ML datatype `datatype v = succ of (unit -> v)`, or `data V = Succ V` in Haskell, and it is widely regarded as a source of recursion (e.g. [6]).

Sheaf categories. We interpret types of the language as sheaves and terms as natural transformations between them. Following our motivating example, a *(concrete) v-set* is a set X together with a given set $C_X \subseteq [\mathrm{V} \to X]$ of chains with endpoints; these should be closed under pre-composition with Scott-continuous functions of V and contain all constant functions. For example, any cpo X can be regarded as a concrete v-set where the chains are the chains in X with their limits. The concrete v-sets form the (concrete) sheaves on the one object category $\mathbb{V}$ whose morphisms are Scott-continuous functions $\mathrm{V} \to \mathrm{V}$ (§5). It is helpful to bear in mind two views of this category, or any category of sheaves:

- The external view is that the sheaves comprise sets with infinitary logical relations (of arity V). The invariance property has the flavour of a Kripke structure, so they are similar to Kripke logical relations.
- The internal view is that the category of sheaves is a model of intuitionistic set theory, with a special object V for which all functions $\mathrm{V} \to \mathrm{V}$ are continuous.

Partial functions with semidecidable domains. Our programming language contains functions that might not terminate, and so programs correspond to partial functions. Intuitively, we should only consider partial functions with a semidecidable domain. We formalize this by requiring that a normal model have a specified sheaf Δ of "semidecidable truth values" (§3, Definition 3.1). For example, in concrete v-sets we pick $\Delta = \{0 \leq 1\}$ with $C_\Delta \subseteq [\mathrm{V} \to \Delta]$ the characteristic functions of infinite or empty up-sets. In general, a choice of object Δ induces a "lifting" monad L. So we can program with partial functions $X \to L(Y)$ using Moggi's monadic metalanguage [32].

Recursion via orthogonality. Among the v-sets, there is a canonical sheaf V, but actually we can construct an analogous sheaf $\bar{\omega}$ in any sheaf category with a semidecidable truth object Δ, by taking a limit of a chain (§2.1). We can also define a *non*-extended vertical natural numbers sheaf ω by taking a *co*limit of a chain; in v-sets this is the set $\{0 \leq 1 \leq \dots\}$ without an endpoint, with chains all the eventually constant ones.
Our language has recursion, and we interpret recursive definitions in a sheaf A by using Tarski's fixed point theorem, by building a chain and taking its formal limit. This can be done in a canonical way when A is *complete*, which we define in terms of *orthogonality*. The conditions says that the morphism $A^{\bar{\omega}} \to A^{\omega}$ induced by $\omega \subseteq \bar{\omega}$ is an isomorphism: intuitively, every chain has a canonical upper bound (§2.2). We give a recipe for showing that A is complete for the interpretation of any type (§3.1).

Recall that cpo's can be regarded as v-sets,. The constructions of product, function cpo, and lifting are all preserved by the inclusion functor hence the interpretation in v-sets is equivalent to the usual one in cpo's. The point is that we can now follow the same kind of interpretation in any sheaf category with this structure, and we can combine our site $\mathbb{V}$ with other sites, as we now explain.

1.2 Combining sites and full abstraction

In §6, we build a sheaf category that is a normal model for our variant of PCF, that we show to be fully abstract in Theorem 7.7. Our argument is based on full definability: every morphism has a syntactic counterpart.

Our construction in §7 is non-syntactic, but by way of motivation we first consider a site built from the syntax of PCF. First, let us define a syntactic "semidecidable subset" of a type τ to be a definable function $s : \tau \to \mathsf{unit}$, i.e. it will either terminate or diverge. Now we temporarily define a category $\mathcal{S}yn$ where the objects are pairs (τ, s) of a type τ and a semidecidable property. A morphism $f : (\tau, s) \to (\tau', s')$ is a definable function $f : \tau \to \tau'$ such that $s = \big(\lambda x.\, f(x); ()\big)$ and $f = \big(\lambda x.\, s(x); \mathsf{let}\, y = f(x)\, \mathsf{in}\, s'(y); y\big)$. In other words, the morphisms of this category should be regarded as total maps on their given domains.

The presheaf category $[\mathcal{S}yn^{\mathsf{op}}, \mathsf{Set}]$ nearly satisfies all the requirements of a normal model, and since the Yoneda embedding $\mathcal{S}yn \to [\mathcal{S}yn^{\mathsf{op}}, \mathsf{Set}]$ is always full and faithful, we almost have a model with full definability. There are two obstacles which we will explain how to bypass: the natural numbers are not preserved by the Yoneda embedding, and we would prefer a non-syntactic model. To resolve these issues we also need machinery for combining concrete sites.

Natural numbers objects and truncated definability. In a non-trivial sheaf category there are uncountably many morphisms $\mathbb{N} \to \mathbb{N}$. This is arguably a good thing, in that we can reason set-theoretically, but it means that we cannot have full definability because the syntax is countable. We follow Milner [31] in considering, for each n, a version of PCF where any natural number $> n$ triggers divergence. For this truncated language, it is possible to impose a sheaf condition on the site $\mathcal{S}yn$ so that the Yoneda embedding $\mathcal{S}yn \to \mathsf{Sh}(\mathcal{S}yn)$ preserves the structure of the language. Now, by combining sites for all possible n, together with $\mathbb{V}$ to include recursion, we end up with sufficient definability.

Non-syntactic models. To avoid using the syntax of PCF in the definition of the model, we consider a broader semantic class of sites that we can show include ones with truncated full definability. We assemble this broad class of sites by using a general method (§6.3) based on a semantic structure for sequentiality called "structural systems of partitions" [30, 46].

Combining sites and concreteness. PCF satisfies the context lemma, which is to say that the meaning of a term with free variables can be determined by substituting closed values for those variables. In a categorical semantics, since the terminal object interprets the empty context, the context lemma indicates that we are working with categories $\mathcal{E}$ that are *concrete* in the sense that the hom-functor $\mathcal{E}(1, -) : \mathcal{E} \to \mathsf{Set}$ is faithful: in effect, we are working with a category of sets and functions.

Sheaf categories are not concrete in general. In fact, in future work we intend to use non-concrete sheaf categories to address non-well-pointed phenomena in semantics [24]. But to model PCF, we need to ensure that when we combine sites we preserve concreteness. To this end we introduce a notion of sum for concrete sites, and show that it is a way of building normal models (§6.4). Moreover, as we show, there are structure preserving functors out of this sum (Proposition 6.12).

In summary, we build our fully abstract model by taking the sum of all the concrete sites that can be built with structural systems of partitions, together with $\mathbb{V}$ for recursion. We then show that all the definable models arise, and hence obtain the definability property, from which we can deduce full abstraction.

2 A categorical setting for recursion

Recursion in a programming language is usually interpreted using Tarski's fixed point theorem (e.g. [17, §12.5]). Although this is usually phrased in terms of partial orders of some flavour, in this section we provide a general abstract categorical treatment (Theorem 2.2). We give a language and its interpretation in §4.

For this section we fix a cartesian closed category $\mathbb{C}$ with a pointed strong monad L. Recall that a cartesian closed category allows us to interpret a terminating typed λ-calculus, and that a strong monad is a triple $(L, \{\eta_X : X \to L(X)\}_X, \{\gg\!=_{X,Y} : L(Y)^X \to L(Y)^{L(X)}\})$ satisfying associativity and identity laws, which allows us to interpret impure computation. A *pointed* monad is one equipped with a natural family of maps $\perp_A : 1 \to L(A)$. We will think of L as a partiality monad, so that morphisms $\Gamma \to L(X)$ are thought of as programs that need not terminate. Our main example is the category vSet with its lifting monad L_{vSet} given in §5, and the category $\mathcal{G}$ with $L_{\mathcal{G}}$ given in §7 is another. In the meantime, it might help the reader to think of the category whose objects are posets and whose morphisms are monotone maps which preserve all suprema of ω-chains that exist, together with the monad that adds a new element to the bottom of a poset. Then Definition 2.1 below would pick out as a full subcategory the category of ω-cpo's and ω-continuous maps.

Many of the ideas in this section and in §3 are well established in synthetic/axiomatic domain theory. We review the literature in §8.2.

2.1 Vertical natural numbers

In this abstract setting, provided certain limits and colimits exist, we can construct objects analogous to the linear orders $(0 \leq 1 \leq 2 \leq \dots)$ and $(0 \leq 1 \leq 2 \leq \dots \leq \infty)$, respectively called the *finite* and *extended vertical natural numbers*. The relationship between these is crucial for Tarski's fixed point theorem.

We assume that the following sequential diagram has a limit $\bar{\omega}$:

$$1 \xleftarrow{!} L1 \xleftarrow{L(!)} LL1 \xleftarrow{LL(!)} \dots \tag{1}$$

We think of this limit as the extended vertical natural numbers. In particular, there is a morphism $\mathsf{succ}_{\bar{\omega}} : \bar{\omega} \to \bar{\omega}$ determined by the cone over diagram (1) with apex $\bar{\omega}$ given by $\bar{\omega} \to L^n 1 \xrightarrow{\eta_{L^n 1}} L^{n+1}1$ and $! : \bar{\omega} \to 1 = L^0 1$. There is another cone with apex 1 given by $1 \xrightarrow{\eta_{L^{n-1}1} \circ \ldots \circ \eta_1} L^n 1$ which defines a morphism $\infty : 1 \to \bar{\omega}$. Note that $\mathsf{succ}_{\bar{\omega}} \circ \infty = \infty$.

We also assume that the following diagram has a colimit ω:

$$1 \xrightarrow{\perp_1} L1 \xrightarrow{L(\perp_1)} LL1 \xrightarrow{LL(\perp_1)} \ldots \tag{2}$$

We think of this colimit as the finite vertical natural numbers. In particular, there is a cocone over diagram (2) with apex ω given by $L^n 1 \xrightarrow{\eta_{L^n 1}} L^{n+1}1 \to \omega$ which defines a morphism $\mathsf{succ}_\omega : \omega \to \omega$. There is a canonical comparison map $i : \omega \to \bar{\omega}$ which comes from maps $L^m 1 \xrightarrow{L^m(\perp_1)} \ldots \xrightarrow{L^{n-1}(\perp_1)} L^n 1$ for $m \leq n$ and $L^m \xrightarrow{L^{m-1}(!)} \ldots \xrightarrow{L^n(!)} L^n 1$ for $m \geq n$.

It is straightforward to check that $i \circ (\mathsf{succ}_\omega : \omega \to \omega) = (\mathsf{succ}_{\bar{\omega}} : \bar{\omega} \to \bar{\omega}) \circ i$.

2.2 Complete objects and fixed points

In the traditional poset-based setting, Tarski's fixed point theorem requires that every chain has a least upper bound. This completeness can be expressed in this abstract categorical setting because a morphism $\omega \to X$ can be thought of as a chain in X.

Recall that an object X is said to be *right-orthogonal* to a morphism $f : A \to B$ if every map $A \to X$ factors uniquely through f. We can then make the following definition:

▶ **Definition 2.1.** *An object $X \in \mathbb{C}$ is L-*complete *if it is right-orthogonal to the morphism* $\mathsf{id}_A \times i : A \times \omega \to A \times \bar{\omega}$ *for every $A \in \mathbb{C}$.*

For example, in the category of ω-cpo's and continuous maps, all objects are complete for the usual lifting monad. From §3 we will work in sheaf categories where one does not expect this.

The present abstract setting admits the following fixed point theorem. The theorem is about L-complete objects that are moreover L-algebras (i.e. objects X equipped with a morphism $L(X) \to X$ satisfying conditions). In the poset setting, L-algebras are just partial orders with a least element.

▶ **Theorem 2.2.** *Let $X \in \mathbb{C}$ be an L-algebra and LX an L-complete object. Then for any map $g : \Gamma \times X \to X$ we can construct a fixed point $\phi_g : \Gamma \to X$ such that $\phi_g(\rho) = g(\rho, \phi_g(\rho))$.*

Given an interpretation for a language in $\mathbb{C}$ such that types are L-complete objects, we can use Theorem 2.2 to interpret fixed points suitable for call-by-value:

▶ **Corollary 2.3.** *Consider objects Γ, A, B in $\mathbb{C}$ such that $L(LB^A)$ is a L-complete object. For a morphism $M : \Gamma \times LB^A \times A \to LB$ we can construct a fixed point $\mathsf{rec}_M : \Gamma \to LB^A$ such that: $\mathsf{rec}_M(\rho)(a) = M(\rho, \mathsf{rec}_M(\rho), a)$.*

Both fixed points ϕ_g and rec_M are constructed in Appendix A.1.

3 Partial maps, semidecidability and recursion in toposes

In this section we keep fixed a Grothendieck topos $\mathcal{E}$. (We will not assume deep familiarity with Grothendieck toposes, but we recall that they are cartesian closed categories with a particularly well behaved notion of subobject and also well-behaved limits/colimits; these toposes turn out to be exactly the categories of sheaves on sites, see §6.1.) We suppose moreover that $\mathcal{E}$ comes with a suitable notion of "semidecidable subset", which is classified by an object Δ of $\mathcal{E}$ as follows.

▶ **Definition 3.1.** *For a fixed object Δ and a fixed monomorphism $\top : 1 \rightarrowtail \Delta$, we say a subobject of A is* semidecidable *if it is a pullback of $\top$ along some map $A \to \Delta$.*

We say that $\top : 1 \rightarrowtail \Delta$ is a generic semidecidable subobject *if:*

- *for every semidecidable subobject $m : A' \rightarrowtail A$ there is precisely one map $\phi : A \to \Delta$ such that m is the pullback of $\top$ along ϕ;*
- *every $0 \rightarrowtail A$ is semidecidable;*
- *semidecidable monomorphisms are closed under composition.*

Our notion is almost exactly what was called a “dominance” in [40] and a “partial truth value object” in [34]. The difference is our requirement that the empty subobjects be semidecidable.

Throughout this section we assume a fixed generic semidecidable subobject $\top : 1 \rightarrowtail \Delta$. It is straightforward to show that semidecidable subobjects are closed under finite meets, including top subobjects, and stable under pullback. Moreover, all coproduct inclusions are semidecidable.

A *partial map* $A \rightharpoonup B$ consists of a semidecidable subobject $A' \rightarrowtail A$ and a map $A' \to B$. Partial maps form a category, which can be given directly or described as the Kleisli category for a certain strong monad L_Δ, the *lifting monad*. The unit of this monad assigns to each object B its *partial map classifier* $B \rightarrowtail L_\Delta B$, which is characterized by the property that maps $A \to L_\Delta B$ correspond to partial maps $A \rightharpoonup B$ (the domain of the partial map is given by pulling back the subobject $B \rightarrowtail L_\Delta B$). It is well-known that this gives a strong monad on $\mathcal{E}$ [34, 5], which is moreover commutative and an “equational lifting monad” in the sense of [3]. The fact that $0 \rightarrowtail 1$ is semidecidable means that L_Δ has a point $\bot_A : 1 \to L_\Delta A$.

3.1 Recipes for complete objects

We now show that a large amount of recursion comes from the assumption of L_Δ-completeness of the generic semidecidable Δ. Since we are working in a Grothendieck topos $\mathcal{E}$, the colimit ω_Δ and limit $\bar{\omega}_\Delta$ arising from the lifting monad L_Δ exist and are preserved by products, as in §2.1. It is useful to consider a slight strengthening of the L_Δ-completeness condition, which roughly says that an object is L_Δ-complete with respect to partial maps.

▶ **Definition 3.2.** *Let $\mathcal{O}_\Delta$ be the class of maps in $\mathcal{E}$ which are pullbacks of maps $i \times \mathsf{id}_A : \omega_\Delta \times A \to \bar{\omega}_\Delta \times A$ along semidecidable subobjects of $\bar{\omega} \times A$. Write $\mathcal{O}_\Delta^\boxslash$ for the class of objects right orthogonal to every map in $\mathcal{O}_\Delta$.*

The following facts are standard and straightforward.

- $\mathcal{O}_\Delta^\boxslash$ is contained in the class of L_Δ-complete objects.
- $\mathcal{O}_\Delta$ is closed under the operations $(-)\times\mathsf{id}_A$, under pullback along semidecidable subobjects, and under colimits in the arrow category of $\mathcal{E}$.
- $\mathcal{O}_\Delta^\boxslash$ is a reflective subcategory of $\mathcal{E}$, closed under limits, and an exponential ideal.

Every Grothendieck topos $\mathcal{E}$ admits a set $\mathcal{S}$ which generates $\mathcal{E}$ under colimits: if $\mathcal{E}$ is a presheaf topos, one may take $\mathcal{S}$ to be the representable presheaves; more generally if $\mathcal{E}$ is a sheaf topos take $\mathcal{S}$ to be the sheafiied representables. Then it follows that the class $\mathcal{O}_\Delta^\boxslash$ is equivalently the class of objects right orthogonal to a certain small subset of $\mathcal{O}_\Delta$, those maps of the form $i \times \mathsf{id}_A$ for $A \in \mathcal{S}$ taken from the generating set.

We summarize the following consequences of the assumption of Δ being L_Δ-complete.

▶ **Proposition 3.3.** *Suppose that Δ is L_Δ-complete.*

- *Δ is in $\mathcal{O}_\Delta^\boxslash$, and for $A \in \mathcal{E}$, $A \in \mathcal{O}_\Delta^\boxslash$ iff $L_\Delta A$ is L_Δ-complete iff $L_\Delta A \in \mathcal{O}_\Delta^\boxslash$.*
- *$\mathcal{O}_\Delta^\boxslash$ is closed under L_Δ and contains 0.*
- *$\mathcal{O}_\Delta^\boxslash$ is closed under I-indexed coproducts iff $\sum_J 1 \in \mathcal{O}_\Delta^\boxslash$ for some set J with $|I| \leq |J|$.*

Proof notes. Δ being L_Δ-complete means that there is a bijection between the semidecidable subobjects of $\omega_\Delta \times A$ and $\bar{\omega}_\Delta \times A$ for any A. From this, and the fact that $\Delta \cong L_\Delta 1$, one deduces the first claim. Closure of $\mathcal{O}^{\boxslash}_\Delta$ under L_Δ can be obtained directly, but also follows from Theorem 3.1 of [8], since L_Δ is a special case of a partial product functor. Finally, note that $\sum_J 1 \in \mathcal{O}^{\boxslash}_\Delta$ implies that a J-indexed join of disjoint semidecidable subobjects of $A \times \bar{\omega}_\Delta$ is semidecidable iff the join of their pullbacks to $A \times \omega_\Delta$ is semidecidable. ◀

4 A higher-order language with recursion

In this section we introduce the call-by-value calculus $\mathsf{PCF_v}$ whose models we will study in the rest of the paper. The calculus is an extension of the simply typed lambda calculus with binary products and sums and a type nat of natural numbers. $\mathsf{PCF_v}$ is given as a fine-grained call-by-value calculus [25], which means there is a syntactic distinction between values and computations. It includes a construct for defining recursive functions $(\mathsf{rec}\, f\, x.\, t)$ which should be thought of as the recursive definition of a function f, $f(x) = t$. There is also a construct for explicitly sequencing computations $\mathsf{let}\, x = t \,\mathsf{in}\, t'$.

$$\begin{aligned}
\text{Types:} \quad & \tau ::= 0 \mid 1 \mid \mathsf{nat} \mid \tau + \tau \mid \tau \times \tau \mid \tau \to \tau \\
\text{Values:} \quad & v, w ::= x \mid \star \mid \mathsf{inl}\, v \mid \mathsf{inr}\, v \mid (v, v) \mid \mathsf{zero} \mid \mathsf{succ}(v) \mid \lambda x.\, t \mid \mathsf{rec}\, f\, x.\, t \\
\text{Computations:} \quad & t ::= \mathsf{return}\, v \mid \mathsf{case}\, v \,\mathsf{of}\, \{\mathsf{inl}\, x \to t,\ \mathsf{inr}\, y \to t'\} \mid \pi_1 v \mid \pi_2 v \mid v\ w \\
& \quad \mid \mathsf{case}\, v\ \mathsf{of}\ \{\mathsf{zero} \to t,\ \mathsf{succ}(x) \to t'\} \mid \mathsf{let}\, x = t \,\mathsf{in}\, t'
\end{aligned}$$

There are two typing relations, one for values, $\vdash^{\mathsf{v}}$, and one for computations, $\vdash^{\mathsf{c}}$, defined as usual. We can define a big-step operational semantics in the usual way, by induction on types, as a relation $\Downarrow_\tau$ between a closed computation and a closed value, both of type τ. The complete definitions appear in Appendix B. For example:

$$\frac{\Gamma,\, x : \tau \vdash^{\mathsf{c}} t : \tau'}{\Gamma \vdash^{\mathsf{v}} \lambda x.\, t} \qquad \frac{\Gamma,\, f : \tau \to \tau',\, x : \tau \vdash^{\mathsf{c}} t : \tau'}{\Gamma \vdash^{\mathsf{v}} \mathsf{rec}\, f\, x.\, t : \tau \to \tau'} \qquad \frac{t[(\mathsf{rec}\, f\, x.\, t)/f,\ v/x] \Downarrow_{\tau'} w}{(\mathsf{rec}\, f\, x.\, t)\ v \Downarrow_{\tau'} w}$$

The operational semantics gives the usual notion of contextual equivalence: two computations t and t' are contextually equivalent iff, for all contexts C such that $C[t]$ and $C[t']$ are closed computations of ground type, $C[t] \Downarrow_\tau v \Leftrightarrow C[t'] \Downarrow_\tau v$, and similarly for values.

4.1 Denotational semantics

We now outline the framework used for our denotational semantics of $\mathsf{PCF_v}$.

▶ **Definition 4.1.** *A* normal model *of* $\mathsf{PCF_v}$ *is a Grothendieck topos* $\mathcal{E}$ *together with a generic semidecidable subobject* $1 \rightarrowtail \Delta$ *such that* $L_\Delta(N_\mathcal{E})$ *is a complete object for* L_Δ*, where* $N_\mathcal{E} = \sum_0^\infty 1$.

The interpretation of $\mathsf{PCF_v}$ types in any normal model $\mathcal{E}$ is given by $[\![0]\!] = 0$, $[\![1]\!] = 1$, $[\![\mathsf{nat}]\!] = \sum_0^\infty 1 = 1 + 1 + \ldots$, $[\![\tau \to \tau']\!] = [\![\tau]\!] \Rightarrow L_\Delta [\![\tau']\!]$, $[\![\tau \times \tau']\!] = [\![\tau]\!] \times [\![\tau']\!]$, and $[\![\tau + \tau']\!] = [\![\tau]\!] + [\![\tau']\!]$. The interpretation for values and computations is standard. A value $\Gamma \vdash^{\mathsf{v}} v : \tau$ is interpreted as a morphism $[\![\Gamma]\!] \to [\![\tau]\!]$ in $\mathcal{E}$. A computation $\Gamma \vdash^{\mathsf{c}} t : \tau$ is a morphism $[\![\Gamma]\!] \to L_\Delta [\![\tau]\!]$. The term $(\mathsf{rec}\, f\, x.\, t)$ can be interpreted with the fixed point constructed in Corollary 2.3.

Since $\Delta \cong L_\Delta 1$ (Definition 3.1) is a retract of $L_\Delta(N_\mathcal{E})$, the object Δ in a normal model is L_Δ-complete. Hence it follows from Proposition 3.3 and its preceding discussion that all $\mathsf{PCF_v}$ types are interpreted as L_Δ-complete objects in a normal model.

5 Presheaves on the vertical natural numbers

This section describes the category vSet, an example of a normal model. An object of vSet, or a *v-set*, is intuitively a set of points equipped with a abstract collection of limiting ω-chains. We ask that the chains be closed under the action of a monoid of reindexings.

Let $\mathbb{V}$ be the monoid of continuous monotone endomorphisms of the extended vertical natural numbers $\{0 \le 1 \le \cdots \le n \le \cdots \le \infty\}$. As such, it is a one-object full subcategory of the category $\omega\mathsf{CPO}$ of ω-cpo's. Recall that the category $[\mathbb{C}^{\mathsf{op}}, \mathsf{Set}]$ of *presheaves* on a small category $\mathbb{C}$ is the category with objects contravariant functors $F : \mathbb{C}^{\mathsf{op}} \to \mathsf{Set}$ and morphisms $F \to G$ natural transformations.

▶ **Definition 5.1.** vSet *is the category* $[\mathbb{V}^{\mathsf{op}}, \mathsf{Set}]$ *of presheaves on* $\mathbb{V}$.

Equivalently, vSet is the category of sets equipped with an action of the monoid $\mathbb{V}$ with equivariant maps. For $X \in \mathsf{vSet}$ we think of $X(\mathrm{V})$ as a set of "abstract chains". We write $|X| = \mathsf{vSet}(1, X)$ for the set of global elements, thought of as "points"; note that we can also describe $|X|$ as the set of $x \in X(\mathrm{V})$ such that $X(e)(x) = x$ for all $e \in \mathbb{V}(\mathrm{V}, \mathrm{V})$. Thus each abstract chain $s \in X(\mathrm{V})$ gives an actual chain of points of X: $X(c_0)(s), X(c_1)(s), \ldots, X(c_\infty)(s)$, where $c_n : \mathrm{V} \to \mathrm{V}$ is the constant map with value n for $n \in \mathbb{N} \sqcup \{\infty\}$.

The category $\omega\mathsf{CPO}$ embeds fully-faithfully into vSet by mapping an ω-cpo D to the set of ω-chains in D each equipped with their supremum. V-sets in the image of $\omega\mathsf{CPO}$ have several special properties; one of them is that the map $X(\mathrm{V}) \to \mathsf{Set}(\mathbb{N} \sqcup \{\infty\}, |X|)$ given by $s \mapsto \lambda n.X(c_n)(s)$ is injective. An $X \in \mathsf{vSet}$ with this property is called a *concrete v-set*, or *concrete presheaf on* $\mathbb{V}$ (we recall a generalization of this later in Definition 6.4). For a concrete v-set X, the abstract chains in $X(\mathrm{V})$ may be identified with a set of functions $|\mathrm{V}| = \mathbb{N} \sqcup \{\infty\} \to |X|$ containing all constant functions and closed under precomposition with endomorphisms of V.

The full embedding $\omega\mathsf{CPO} \hookrightarrow \mathsf{vSet}$ was already observed by Fiore and Rosolini [13, 14], who then considered a category of sheaves on $\mathbb{V}$ as a model of Synthetic Domain Theory. Their sheaf condition is not relevant to our work here. They consider a dominance in their sheaf category, which we treat as a generic semidecidable subobject in vSet. Let $\Delta_{\mathsf{vSet}} \in \mathsf{vSet}$ be the splitting in vSet of the idempotent $r_1 : \mathrm{V} \to \mathrm{V}$ given by $0 \mapsto 0$ and $x \mapsto 1$ for $x \ge 1$. So $\Delta_{\mathsf{vSet}}(\mathrm{V})$ can be identified with the set of monotone sequences $\mathbb{N} \to \{0, 1\}$.

▶ **Lemma 5.2.** Δ_{vSet} *is a generic semidecidable subobject, as in Definition 3.1.*

Proof notes. The most difficult part to check is that semidecidable monomorphisms are closed under composition. Given $\phi : A \to \Delta_{\mathsf{vSet}}$ classifying $m : B \rightarrowtail A$ and given $\psi : B \to \Delta_{\mathsf{vSet}}$, first note that ψ admits an extension map $\psi' : A \to \Delta_{\mathsf{vSet}}$ where, for $x \in A(\mathrm{V})$, $\psi'(x)$ is the greatest element of Δ_{vSet} (in the lexicographic ordering) such that $\phi(\psi'(x)) = (1, 1, \ldots)$ if it exists and $\psi'(x) = (0, 0, \ldots)$ otherwise. Then if ψ is the classifier of $n : C \rightarrowtail B$, the composite $mn : C \rightarrowtail A$ is classified by the map $\xi : A \to \Delta_{\mathsf{vSet}}$ where, for $x \in A(\mathrm{V})$, $\xi(x)_i = \min\{\phi(x)_i, \psi'(x)_i\}$. ◀

Thus vSet admits a strong, pointed lifting monad L_{vSet}, given by partial map classifiers as in the discussion following Definition 3.1. This lifting monad can be explicitly given by $(L_{\mathsf{vSet}}X)(\mathrm{V}) = \{\bot\} + \sum_{n \in \mathbb{N}} (X(\mathrm{V}))_n$ so it has a copy of the set $X(\mathrm{V})$ for each $n \in \mathbb{N}$. The action of an endomorphism e on V is:

$$(L_{\mathsf{vSet}}X)(e)(s \in (X(\mathrm{V}))_n) = \begin{cases} \bot & \text{if } \mathsf{im}(e) \subseteq \{0, \ldots, n-1\} \\ X(e')(s) \in (X(\mathrm{V}))_k & \text{if } e(\{0, \ldots, k-1\}) \subseteq \{0, \ldots, n-1\}, \\ & e(k) > n-1,\ e'(i) = e(k+i) - n \end{cases}$$

and $(L_{\mathsf{vSet}}X)(e)(\bot) = \bot$. There is a ready intuition for $(L_{\mathsf{vSet}}\, X)(e)$ which is precise when X is a concrete v-set: an element of $(X(\mathrm{V}))_n$ is a sequence s of elements from $|X|$, to which we add n $\bot$'s at the beginning. The action $(L_{\mathsf{vSet}}X)(e)$ of an endomorphism e of V is now just the standard reindexing of sequences by function composition $(\bot, \ldots, \bot, s) \circ e$.

We now show that $(\mathsf{vSet}, \Delta_{\mathsf{vSet}})$ satisfies the conditions of a normal model (Definition 4.1) of $\mathsf{PCF_v}$, which means showing that $L_{\mathsf{vSet}}(N_{\mathsf{vSet}}) = L_{\mathsf{vSet}}(\sum_0^\infty 1)$ is L_{vSet}-complete. It is straightforward to give the following explicit description of ω and $\bar{\omega}$: for the L_{vSet} lifting monad on vSet, the limit $\bar{\omega}$ is the representable $y(\mathrm{V})$, and $i : \omega \to \bar{\omega}$ is the subobject of maps with bounded image (in particular, eventually constant).

▶ **Lemma 5.3.** *Δ_{vSet} is L_{vSet}-complete.*

Proof notes. Firstly, one checks that Δ_{vSet} is orthogonal to $i : \omega \to \bar{\omega}$, since the maps into Δ_{vSet} from ω or $\bar{\omega}$ are essentially just the eventually constant binary sequences. Then consider an extension problem $f : \omega \times A \to \Delta_{\mathsf{vSet}}$. Precomposing with the surjection on points $\coprod_{x\in|A|} \omega \times 1_{\{x\}} \to \omega \times A$, there is a unique extension to a map $\coprod_{x\in|A|} \bar{\omega} \times 1_{\{x\}}$. This gives a unique candidate extension of f to $\bar{\omega} \times A$. To see that this is a valid morphism in vSet, one simply checks that it maps $(\bar{\omega} \times A)(\mathrm{V})$ into $\Delta_{\mathsf{vSet}}(\mathrm{V})$. ◀

▶ **Proposition 5.4.** *$L_{\mathsf{vSet}}(N_{\mathsf{vSet}}) = L_{\mathsf{vSet}}(\sum_0^\infty 1)$ is L_{vSet}-complete.*

Proof notes. One observes that any map $\omega \to L_{\mathsf{vSet}}(\sum_0^\infty 1)$ or $\bar{\omega} \to L_{\mathsf{vSet}}(\sum_0^\infty 1)$ factors through one of the subobjects $L_{\mathsf{vSet}}(\iota_i) : \Delta_{\mathsf{vSet}} \cong L_{\mathsf{vSet}}\, 1 \rightarrowtail L_{\mathsf{vSet}}(\sum_0^\infty 1)$, where $\iota_i : 1 \to \sum_0^\infty 1$ is the i-th coproduct inclusion. ◀

Therefore, $(\mathsf{vSet}, \Delta_{\mathsf{vSet}})$ is a normal model for $\mathsf{PCF_v}$. Notice that $[\![0]\!]$ and $[\![1]\!]$ are concrete v-sets. It is a standard fact that concrete presheaves are an exponential ideal, and that products and coproducts preserve concrete presheaves. Moreover, by straightforward inspection the lifting monad L_{vSet} preserves concreteness as well. Therefore, the $\mathsf{PCF_v}$ types are interpreted as *concrete presheaves* in vSet. This observation is useful for the proof of the next theorem (Appendix A.2) because we only need to compare certain morphisms on their underlying points.

▶ **Theorem 5.5.** *The pair* $(\mathsf{vSet}, \Delta_{\mathsf{vSet}})$ *gives a sound and adequate model of* $\mathsf{PCF_v}$.

- *Soundness:* $t \Downarrow_\tau v \implies [\![t]\!] = \eta_{[\![\tau]\!]} \circ [\![v]\!] \in L_{\mathsf{vSet}}[\![\tau]\!]$.
- *Adequacy: if τ is a ground type,* $[\![t]\!] = \eta_{[\![\tau]\!]} \circ [\![v]\!] \implies t \Downarrow_\tau v$.

6 Sheaf conditions for sequentiality

In the previous section we used a simple index category, $\mathbb{V}$, to cut down the interpretation of $\mathsf{PCF_v}$-types in Set to a model with recursion. In this section we discuss the other index categories and their combinations, which we need for a fully abstract model. The motivation for the new index categories is that they each encapsulate a "prediction" of the underlying sets of the interpretations of types and the definable morphisms between them. Roughly speaking, the relations force each prediction to arise as a full subcategory, including what turns out to be the correct prediction.

6.1 Sites and sheaves

As the fully abstract model of §7 is given as the topos of sheaves on a site, we recall here some necessary definitions. The standard reference is [20], but for us all sites will be small.

A *site* is a small category $\mathbb{C}$ equipped with a coverage J, where a *coverage* J on $\mathbb{C}$ is a set of covering families $(a, \{f_i \mid i \in I\})$ where $a \in \mathbb{C}$ and each f_i is a morphism $f_i : a_i \to a$ with codomain a such that, whenever $(a, \{f_i : a_i \to a \mid i \in I\}) \in J$ and $g : b \to a$ is in $\mathbb{C}$, there exists $(b, \{h_i : b_i \to b \mid i \in I'\}) \in J$ such that, for all $i \in I'$, there exists $j \in I$ and $k : b_i \to a_j$ such that $f_j \circ k = g \circ h_i$.

Given a covering family $(a, \{f_i : a_i \to a \mid i \in I\}) \in J$ and a presheaf $F : \mathbb{C}^{\mathsf{op}} \to \mathsf{Set}$, a *matching family* is a collection $(s_i \in F(a_i) \mid i \in I)$ such that for all $i, j \in I$, $b \in \mathbb{C}$, $g : b \to a_i$, and $h : b \to a_j$ we have $F(g)(s_i) = F(h)(s_j)$. A *sheaf* on the site $(\mathbb{C}, J)$ is a presheaf $F : \mathbb{C}^{\mathsf{op}} \to \mathsf{Set}$ such that for every covering family $(a, \{f_i : a_i \to a \mid i \in I\}) \in J$ and matching family $(s_i \in F(a_i) \mid i \in I)$ there is a unique element $s \in F(a)$ such that $F(f_i)(s) = s_i$ for all $i \in I$. The element s is called the *amalgamation* of the matching family (s_i). The category of sheaves is denoted $\mathsf{Sh}(\mathbb{C}, J)$.

The notion of coverage we have given here is a minimal one (see A2.1.9 of [20]). There can be several coverages on one category $\mathbb{C}$ giving rise to the same collection of sheaves. It is common to add saturation conditions to the coverage J to tighten the correspondence between coverages and collections of sheaves, and also to assist calculation. The following two are useful for us.

(M) J contains $(a, \{1_a : a \to a\})$ for all $a \in \mathbb{C}$.

(L) If $(a, \{f_i : a_i \to a \mid i \in I\}) \in J$ and $(b_i, \{g_{ij} : b_{ij} \to a_i \mid j \in J_i\}) \in J$ for $i \in I$ then $(a, \{f_i g_{ij} : b_{ij} \to a \mid i \in I, j \in J_i\}) \in J$.

▶ **Example 6.1.** Every small category $\mathbb{C}$ admits a "trivial" coverage, where $J = \emptyset$ and for which all presheaves on $\mathbb{C}$ are J-sheaves. For us, *the trivial coverage* on $\mathbb{C}$ is given by $J = \{(a, \{1_a : a \to a\}) \mid a \in \mathbb{C}\}$, which has the same sheaves (all presheaves) but also satisfies (M) and (L).

A fundamental fact about $\mathsf{Sh}(\mathbb{C}, J)$ is that it is a reflective subcategory of $[\mathbb{C}^{\mathsf{op}}, \mathsf{Set}]$, i.e. the inclusion functor $\mathsf{Sh}(\mathbb{C}, J) \hookrightarrow [\mathbb{C}^{\mathsf{op}}, \mathsf{Set}]$ is full, faithful and possesses a left adjoint, which is called *sheafification*. A coverage is *subcanonical* if all of the representable presheaves $\mathbb{C}(-, a)$ for $a \in \mathbb{C}$ are sheaves – this means that sheafification leaves representables unchanged as functors $\mathbb{C}^{\mathsf{op}} \to \mathsf{Set}$. The trivial coverage is subcanonical, but many useful coverages are not, and in this latter case the sheafified representables play a role analogous to that of the representable presheaves. Hence we will sometimes find it useful to write y for the composite $\mathbb{C} \to [\mathbb{C}^{\mathsf{op}}, \mathsf{Set}] \to \mathsf{Sh}(\mathbb{C}, J)$ of the Yoneda embedding with sheafification.

6.2 Concrete sites

We restrict our attention to a class of sites that are particularly convenient to work with. Unlike the saturation conditions (M) and (L), these restrictions on $\mathbb{C}$ and J do constrain the possible categories of sheaves. Recall the following from [7].

▶ **Definition 6.2.** *A* concrete site *is a site* $(\mathbb{C}, J)$ *with a terminal object* $\star$ *such that the maps* $\mathbb{C}(a, b) \to \mathsf{Set}(\mathbb{C}(\star, a), \mathbb{C}(\star, b))$ *are all injective, and* $\coprod_{i \in I} \mathbb{C}(\star, a_i) \to \mathbb{C}(\star, a)$ *is surjective for every covering family* $(a, \{f_i : a_i \to a \mid i \in I\}) \in J$.

In a concrete site it is convenient to define $|c| = \mathbb{C}(\star, c)$ for $c \in \mathbb{C}$ and to identify each morphism $c \to d$ with the induced function $|c| \to |d|$. Thus $| - |$ is a faithful (but not necessarily full) functor $\mathbb{C} \to \mathsf{Set}$. For a presheaf $X : \mathbb{C}^{\mathsf{op}} \to \mathsf{Set}$, we also write $|X| = X(\star) \cong \mathsf{Nat}(1, X)$.

A concrete site need not be subcanonical, but we can describe the sheafified representables as follows. For any set A, the presheaf $\mathsf{Set}(| - |, A) : \mathbb{C}^{\mathsf{op}} \to \mathsf{Set}$ is a J-sheaf. Every representable $\mathbb{C}(-, c)$ embeds into the sheaf $\mathsf{Set}(| - |, |c|)$ by concreteness and it follows that the sheafification $y(c)$ is the smallest subfunctor of $\mathsf{Set}(|-|,|c|)$ containing $\mathbb{C}(-,c)$ and closed under amalgamation. When J satisfies (M) and (L), then $y(c)$ is obtained by closing $\mathbb{C}(-,c)$ under amalgamation in just one step.

▶ **Example 6.3.** The category $\mathbb{V}$ as given Definition 5.1 is not quite a concrete site, since it lacks a terminal object. However, as is well-known, the idempotent splitting of any small category has an equivalent presheaf category (see A1.1.19 of [20]). As the idempotent splitting of $\mathbb{V}$ contains a terminal object we are free to add it to $\mathbb{V}$, which we now treat as a concrete site with the trivial coverage.

A concrete site $(\mathbb{C}, J)$ is in particular a site, so it has a category $\mathsf{Sh}(\mathbb{C}, J)$ of sheaves. However, in this setting there is an especially useful subcategory.

▶ **Definition 6.4.** *Let $(\mathbb{C}, J)$ be a concrete site. A* concrete presheaf *is a presheaf $F : \mathbb{C}^{\mathsf{op}} \to \mathsf{Set}$ such that, for every $a \in \mathbb{C}$, the map $(F(x : \star \to a))_{x \in |a|} : F(a) \to \prod_{x \in |a|} |F|$ is injective. A* concrete sheaf *is a concrete presheaf which is also a J-sheaf.*

The advantage of working with concrete presheaves is that if Y is a concrete presheaf, and X is any presheaf, then natural transformations $\alpha : X \to Y$ are determined by the function $\alpha_\star : |X| \to |Y|$. As $Y(a) \subseteq \mathsf{Set}(|a|, |Y|)$, we can think of Y as *being* the set $|Y|$ together with an $\mathsf{ob}(\mathbb{C})$-indexed family of relations.

We remark that concrete sheaves form a reflective subcategory, and so are closed under limits, and an exponential ideal. All representables are concrete presheaves and concrete presheaves are closed under coproducts. Since every concrete presheaf X embeds into the concrete sheaf $\mathsf{Set}(|-|, |X|)$, every concrete presheaf injects into its sheafification and it follows that the concrete sheaves are closed under coproducts in sheaves.

6.3 Defining concrete sites via systems of partitions

To help us define sites that we need for full abstraction, we first recall the category SSP of Marz and Streicher [29, 30, 46].

▶ **Definition 6.5.** *Given a finite set w, a system of partitions S^w is a set containing* sets of disjoint subsets of *w, that is, (partial) partitions of w, and satisfying the following axioms:*

1. $\{w\}, \emptyset \in S^w$.
2. *(Refinement) $P, Q \in S^w$ and $U \in P$ imply that: $(P\setminus\{U\}) \cup (\{U \cap V \mid V \in Q\} \setminus \{\emptyset\}) \in S^w$.*
3. *$U, V \in P \in S^w$ implies that $(P \setminus \{U, V\}) \cup \{U \cup V\} \in S^w$.*

The category SSP has objects pairs (w, S^w) of a finite set w and a system of partitions S^w for it. A morphism $f : (v, S^v) \to (w, S^w)$ is a set function $f : v \to w$ such that if $P = \{w_1, \dots, w_n\} \in S^w$, then $\{f^{-1}(w_1), \dots, f^{-1}(w_n)\} \setminus \{\emptyset\} \in S^v$. Composition is given by composition of functions.

The objects of SSP encode the idea of a finite type w together with a system of computable partial functions $w \rightharpoonup \mathbb{N}$. It may be helpful to think of these as potentially destructive measurements or observations on an unknown value of type w. A partial partition $P \in S^w$ stands for an equivalence class of such functions which are undefined on $w \setminus \bigcup P$, constant on each $U \in P$, and which take distinct values on the members of distinct partition classes $U, V \in P$, where the equivalence is modulo a permutation of $\mathbb{N}$. Axioms 1 and 3 correspond to such functions being closed under post-composition with all partial functions $\mathbb{N} \rightharpoonup \mathbb{N}$, and containing all constant functions (including the totally undefined one). Axiom 2 says that two computable functions $w \rightharpoonup \mathbb{N}$ can themselves be sequenced together, say by checking whether $w \vdash^{\mathsf{c}} t_1 : \mathsf{nat}$ returns 0 and if so returning the outcome of $w \vdash^{\mathsf{c}} t_2 : \mathsf{nat}$.

In light of the above, there is a natural notion of "semidecidable subobject" in SSP. For $P \in S^w$, there is a monomorphism $(\bigcup P, S^w \restriction_P) \to (w, S^w)$, where $S^w\restriction_P = \{Q \in S^w : \bigcup Q \subseteq \bigcup P\}$. We say that any monomorphism isomorphic to one of this form is *semidecidable*. The corresponding notion of partial map admits partial map classifiers and hence a lifting monad. This lifting monad is given by $L_{\mathsf{SSP}}(w, S^w)$ having underlying set $w \sqcup \{\bot\}$ and $S^{L_{\mathsf{SSP}}(w,S^w)} = S^w \sqcup \{\{w \sqcup \{\bot\}\}\}$. We write $\mathsf{SSP}_\bot$ for the Kleisli category of L_{SSP}, or equivalently the category of partial maps in SSP.

We are interested in faithful functors $F : \mathcal{C} \to \mathsf{SSP}_\bot$. The idea is that $\mathcal{C}$ stands for a system of finite types and definable partial functions between them, while F equips each finite type with a system of measurements which is compatible with the partial functions in $\mathcal{C}$. We now construct a topos $\mathcal{E}$ with generic semidecidable subobject such that $\mathcal{E}$ contains $\mathcal{C}$ as a full subcategory, and in which the observations on the SSP-object $F(c)$ are precisely the partial maps $c \rightharpoonup N_{\mathcal{E}} = \sum_0^\infty 1$ in $\mathcal{E}$.

▶ **Definition 6.6.** *For a faithful functor $F : \mathcal{C} \to \mathsf{SSP}_\bot$ the category $\mathcal{I}_{\mathcal{C},F}$ is as follows.*

- *Objects: pairs (c, U) where $c \in \mathcal{C}$ and $U = \bigcup P$ for some $P \in S^{F(c)}$ (equivalently $U = \emptyset$ or $\{U\} \in S^{F(c)}$ by axiom 3 of SSP); and a distinguished terminal object $\star$.*
- *Morphisms $X \to Y$ are certain functions $|X| \to |Y|$, where $|(c, U)| = U$ and $|\star| = \{*\}$. When $X = (c, U)$ and $Y = (d, V)$, we take those functions $f : U \to V$ either constant or for which there is $\phi : c \to d$ in $\mathcal{C}$ such that $F(\phi) : F(c) \to L_{\mathsf{SSP}}(F(d))$ has domain U and $F(\phi)(U) \subseteq V$. When either of X, Y is $\star$, take all functions.*

The category $\mathcal{I}_{\mathcal{C},F}$ serves as "totalization" of $F : \mathcal{C} \to \mathsf{SSP}_\bot$, by adding enough subobjects that every partial map can be represented by a total one. It is not enough to take presheaves on $\mathcal{I}_{\mathcal{C},F}$. We need a coverage in order to force the coproduct $\sum_0^\infty 1$ to have the correct elements. We emphasize that this is not merely an artefact arising from the sum types in $\mathsf{PCF_v}$, it is necessitated by a normal model having nat interpreted as the coproduct $\sum_0^\infty 1$.

▶ **Definition 6.7.** *Given a faithful functor $F : \mathcal{C} \to \mathsf{SSP}_\bot$, the coverage $\mathcal{J}_{\mathcal{C},F}$ has as covers families of partial identity maps $\{(c, U_i) \to (c, U)\}_{1 \le i \le n}$ where $P = \{U_1, \ldots, U_n\} \in S^{F(c)}$ and $\bigcup U_i = U$; and $\star$ is covered only by the identity.*

The following proposition is straightforward. The main point to note is that axiom 2 of SSP is required for the basic coverage axiom. That same axiom is what gives us (L).

▶ **Proposition 6.8.** *$(\mathcal{I}_{\mathcal{C},F}, \mathcal{J}_{\mathcal{C},F})$ is a concrete site, satisfying the (M) and (L) axioms.*

In $\mathsf{Sh}(\mathcal{I}_{\mathcal{C},F}, J_{\mathcal{C},F})$ we define $\Delta_{\mathcal{C},F}$ where $\Delta_{\mathcal{C},F}(c, U)$ is the set of subsets $U' \subseteq U$ where (c, U') is an object of $\mathcal{I}_{\mathcal{C},F}$, and $\Delta_{\mathcal{C},F}(\star) = \{\emptyset, |\star|\}$. The following is straightforward.

▶ **Proposition 6.9.** *$\Delta_{\mathcal{C},F}$ is a concrete sheaf and a generic semidecidable subobject in $\mathsf{Sh}(\mathcal{I}_{\mathcal{C},F}, J_{\mathcal{C},F})$. The lifting monad $L_{\mathcal{C},F}$ preserves concrete sheaves.*

We have the following explicit description of the lifting monad: $(L_{\mathcal{C},F}A)(\star) = A(\star) + \{\bot\}$ and $(L_{\mathcal{C},F}A)(c, U) = \coprod_{U' \subseteq U} A(c, U')$, where the coproduct is taken over all $U' \subseteq U$ such that there exists a partition $P \in S^{F(c)}$ such that $\bigcup P = U'$ (i.e. (c, U') is an object of $\mathcal{I}_{\mathcal{C},F}$).

One should not expect $\Delta_{\mathcal{C},F}$ to be $L_{\mathcal{C},F}$-complete. It is only by summing with $\mathbb{V}$ as in the next section that we obtain a complete generic semidecidable subobject. For this purpose it is still useful to describe the objects $\omega_{\mathcal{C},F}$ and $\bar{\omega}_{\mathcal{C},F}$ explicitly. They are both concrete sheaves, and we can make the identifications $\bar{\omega}_{\mathcal{C},F}(\star) \cong \mathbb{N} \sqcup \{\infty\}$ and $\omega_{\mathcal{C},F}(\star) \cong \mathbb{N}$. More generally, for $(c, U) \in \mathcal{I}_{\mathcal{C},F}$, elements of $\omega_{\mathcal{C},F}(c, U)$ are $\mathbb{N}$-indexed descending sequences of semidecidable subsets of U. The set $\bar{\omega}_{\mathcal{C},F}(c, U)$ consists of the $(\mathbb{N} \sqcup \{\infty\})$-indexed descending sequences of semidecidable subsets of U. Note that there is no continuity condition at infinity, the last subset need only be *contained* in the intersection of the earlier ones.

6.4 Summing concrete sites

The fully abstract model depends on combining many different sites together. Here we describe this process as an elementary construction for "summing" a small collection of concrete sites.

▶ **Definition 6.10.** *Let $\{(\mathbb{C}_i, J_i)\}_{i\in I}$ be a (non-empty) family of concrete sites. Then $\sum_i \mathbb{C}_i$ is the category whose objects are $\coprod \mathsf{ob}(\mathbb{C}_i)/\sim$, where $\sim$ identifies the terminal objects in each category, and whose morphisms $(c \in \mathbb{C}_i) \to (d \in \mathbb{C}_j)$ are those functions $|c| \to |d|$ which are in $\mathbb{C}_i$ if $i = j$ and all constant functions if $i \neq j$. The coverage $\sum_i J_i$ has precisely the covers of J_i for $c \in \mathbb{C}_i$ and $\star$ covered by the identity.*

It is straightforward to see that $(\sum_{i\in I} \mathbb{C}_i, \sum_{i\in I} J_i)$ is a concrete site. It satisfies axioms (M) and (L) if all the $(\mathbb{C}_i, J_i)$ do, but it need not be subcanonical even when the $(\mathbb{C}_i, J_i)$ are. Let us write in_j for the inclusion $\mathbb{C}_j \to \sum_{i\in I} \mathbb{C}_i$. Recall that there is an adjoint triple $(\mathsf{in}_j)_! \dashv (\mathsf{in}_j)^* \dashv (\mathsf{in}_j)_*$ where $(\mathsf{in}_j)^* : [(\sum_i \mathbb{C}_i)^{\mathsf{op}}, \mathsf{Set}] \to [\mathbb{C}_i^{\mathsf{op}}, \mathsf{Set}]$ is given by precomposition with in_j and its adjoints are given by left and right Kan extension.

▶ **Lemma 6.11.** *$(\mathsf{in}_j)^*$ and $(\mathsf{in}_j)_*$ preserve sheaves and the latter is full and faithful; $(\mathsf{in}_j)_!$ preserves finite limits. A presheaf $F \in [(\sum_i \mathbb{C}_i)^{\mathsf{op}}, \mathsf{Set}]$ is a $(\sum_i J_i)$-sheaf iff $(\mathsf{in}_j)^*F \in [\mathbb{C}_j^{\mathsf{op}}, \mathsf{Set}]$ is a J_j-sheaf for all $j \in I$. Similarly F is a concrete presheaf iff every $(\mathsf{in}_j)^*F$ is a concrete presheaf.*

In summary, in_j induces a *local geometric morphism* $\mathsf{Sh}(\sum_{i\in I} \mathbb{C}_i, \sum_{i\in I} J_i) \to \mathsf{Sh}(\mathbb{C}_j, J_j)$ meaning there is an adjoint triple, which we also write as $(\mathsf{in}_j)_! \dashv (\mathsf{in}_j)^* \dashv (\mathsf{in}_j)_*$, where $(\mathsf{in}_j)^*$ is precomposition with in_j and $(\mathsf{in}_j)_!$ is given by left Kan extension along in_j followed by sheafification, such that $(\mathsf{in}_j)_!$ preserves finite limits and both $(\mathsf{in}_j)_!$ and $(\mathsf{in}_j)_*$ are full and faithful. Moreover, each $(\mathsf{in}_j)_!$ preserves the respective sheafified representables, and being a (concrete) sheaf for the summed site can be detected by checking under $(\mathsf{in}_j)^*$ for every $j \in I$. The functors $(\mathsf{in}_j)^*$ are jointly faithful and satisfy $|(\mathsf{in}_j)^*Y| \cong |Y|$. If Y is a concrete presheaf then a function $f : |X| \to |Y|$ gives a natural transformation $X \to Y$ iff it gives a natural transformation $(\mathsf{in}_j)^*X \to (\mathsf{in}_j)^*Y$ for every $j \in I$.

We also make the following straightforward observations about monad-lifting.

▶ **Proposition 6.12.** *Let $\mathbb{T}$ be a (strong) monad on Set, and suppose that, for $i \in I$, $\mathbb{T}_i$ is a strong monad on $\mathsf{Sh}(\mathbb{C}_i, J_i)$ which lifts $\mathbb{T}$ through the global sections function $\mathsf{Sh}(\mathbb{C}_i, J_i) \to \mathsf{Set}$.*

1. *There is a unique strong monad $\widehat{\mathbb{T}}$ on $\mathsf{Sh}(\sum_i \mathbb{C}_i, \sum_i J_i)$ which lifts each of the $\mathbb{T}_j$ through $(\mathsf{in}_j)^* : \mathsf{Sh}(\sum_i \mathbb{C}_i, \sum_i J_i) \to \mathsf{Sh}(\mathbb{C}_j, J_j)$.*
2. *If each $\mathbb{T}_i$ is the partial map classifier for a generic semidecidable subobject Δ_i in $\mathsf{Sh}(\mathbb{C}_i, J_i)$, then there is a generic semidecidable subobject Δ on $\mathsf{Sh}(\sum_i \mathbb{C}_i, J_i)$ whose partial map classifier is $\widehat{\mathbb{T}}$.*
3. *The colimit $\omega_{\mathbb{T}}$ and limit $\bar{\omega}_{\mathbb{T}}$ are sent to $\omega_{\mathbb{T}_j}$ and $\bar{\omega}_{\mathbb{T}_j}$ by $(\mathsf{in}_j)^*$.*

7 A fully abstract model of $\mathsf{PCF_v}$

Let $I_{\mathsf{PCF_v}}$ be the set of all concrete sites of the form $(\mathcal{I}_{\mathcal{C},F}, \mathcal{J}_{\mathcal{C},F})$ where $\mathcal{C}$ has countably many morphisms, together with the site $\mathbb{V}$ (as a concrete site with trivial coverage). For convenience, we continue to write $I_{\mathsf{PCF_v}} = \{(\mathbb{C}_i, J_i) : i \in I_{\mathsf{PCF_v}}\}$, and we write Δ_i for the specified generic semidecidable subobject in $\mathsf{Sh}(\mathbb{C}_i, J_i)$, and L_i for its associated lifting monad.

▶ **Definition 7.1.** *Let* $(\mathcal{I}, \mathcal{J})$ *be the sum of* $I_{\mathsf{PCF_v}}$ *(Definition 6.10) and let* $\mathcal{G} = \mathsf{Sh}(\mathcal{I}, \mathcal{J})$*.*

For $j \in I_{\mathsf{PCF_v}}$, we continue to write $(\mathsf{in}_j)_! \dashv (\mathsf{in}_j)^* \dashv (\mathsf{in}_j)_*$ for the adjoint triple induced by $\mathsf{in}_j : \mathbb{C}_j \hookrightarrow \sum_{i \in I_{\mathsf{PCF_v}}} \mathbb{C}_i$, as in Lemma 6.11. We write y for all sheafified Yoneda embeddings.

We now show that $\mathcal{G}$ is a normal model of $\mathsf{PCF_v}$, and subsequently a fully abstract model. The generic semidecidable subobject $\Delta_{\mathcal{G}}$ is given by $\Delta_{\mathcal{G}}(c) = \Delta_i(c)$ for $c \in \mathbb{C}_i$, as in Proposition 6.12. Thus the lifting monad $L_{\mathcal{G}}$ is determined by $(\mathsf{in}_j)^*(L_{\mathcal{G}}\, A) \cong L_j((\mathsf{in}_j)^* A)$. We can describe $N_{\mathcal{G}} = \sum_0^\infty 1$ explicitly: its set of points is $N_{\mathcal{G}}(\star) = \mathbb{N}$; $N_{\mathcal{G}}(V)$ has only constant sequences in $\mathbb{N}$; and $N_{\mathcal{G}}(c, U)_{\mathcal{C},F} = \{h : U \to \mathbb{N} \mid \{h^{-1}(k) \mid k \in \mathbb{N}\} \in S^w\}$. We have the following (see Appendix A.3 for a proof):

▶ **Proposition 7.2.** $L_{\mathcal{G}}(N_{\mathcal{G}})$ *is* $L_{\mathcal{G}}$*-complete.*

Thus $\mathcal{G}$ satisfies the conditions of Definition 4.1, so $\mathcal{G}$ admits an interpretation of $\mathsf{PCF_v}$. Moreover, it is straightforward to check that $L_{\mathcal{G}}$ preserves concrete sheaves and hence by the discussion in §6.2 the interpretation $[\![\sigma]\!]$ of each $\mathsf{PCF_v}$-type σ is a concrete sheaf. The statement that the interpretation of $\mathsf{PCF_v}$ in $(\mathcal{G}, \Delta_{\mathcal{G}})$ is *adequate* is the same as Theorem 5.5, and the proof is also essentially the same (see also [45]).

7.1 Partial types

As discussed in the introduction, our strategy for obtaining a fully abstract model is to find a model where sufficiently many morphisms are definable. We cannot expect all morphisms to be definable since there are only countably many programs but in a normal model the interpretation of $\mathsf{nat} \to \mathsf{nat}$ always has uncountably many points. Following [31] we show definability only for "partial types" – these are finite approximations to the set of points of each type. As discussed in §6, the site of our sheaf model contains "predictions" of the extent of each partial type and the system of definable functions between them. In the proof of full abstraction we will choose the prediction which is actually realized.

We do not need to consider an intrinsic definition of compactness in a normal model of $\mathsf{PCF_v}$, we simply use definable idempotents to define the partial types. The following was adapted to call-by-value from the call-by-name formulations found in [36, 46].

▶ **Definition 7.3.** *For each type* σ *and* $n \in \mathbb{N}$*, define a computation* $x : \sigma \vdash^{\mathbf{c}} \psi_n^\sigma : \sigma$ *by recursion on* σ *where* ψ_n^{nat} *is "if* $x \leq n$ *then* x *else diverge", and* $\psi_n^{\mathbf{0}} = x$*,* $\psi_n^{\mathbf{1}} = x$*,*

$$\begin{aligned}
\psi_n^{\sigma\to\tau} &= \mathsf{return}\,\lambda u.\,\mathsf{let}\,v = \psi_n^\sigma[u/x]\,\mathsf{in}\,\mathsf{let}\,w = x\ v\,\mathsf{in}\,\psi_n^\tau[w/x],\\
\psi_n^{\sigma+\tau} &= \mathsf{case}\,x\,\mathsf{of}\,\{\mathsf{inl}\,y \to \psi_n^\sigma[y/x],\ \mathsf{inr}\,z \to \psi_n^\tau[z/x]\},\\
\psi_n^{\sigma\times\tau} &= \mathsf{let}\,y = \pi_1 x\,\mathsf{in}\,\mathsf{let}\,z = \pi_2 x\,\mathsf{in}\,\mathsf{let}\,y' = \psi_n^\sigma[y/x]\,\mathsf{in}\,\mathsf{let}\,z' = \psi_n^\tau[z/x]\,\mathsf{in}\,\mathsf{return}\,(y', z').
\end{aligned}$$

We write $h_n^\sigma : [\![\sigma]\!] \to L_{\mathcal{G}}[\![\sigma]\!]$ for the denotation of ψ_n^σ in $\mathcal{G}$. We will say that h_n^σ *fixes* $x \in |[\![\sigma]\!]|$ if $h^\sigma(x) = \eta_{[\![\sigma]\!]}(x)$.

▶ **Proposition 7.4.** *Each* h_n^σ *is an idempotent Kleisli arrow and fixes finitely many points.*

Proof notes. By induction on σ. It is clear h_n^{nat} is idempotent and fixes precisely the subobject $1 + \ldots + 1$ of $[\![\mathsf{nat}]\!]$ given by the first $n+1$ points. For function types, $h_n^{\sigma\to\tau}$ acts on morphisms $f : [\![\sigma]\!] \to L_{\mathcal{G}}[\![\tau]\!]$ by $f \mapsto (h_n^\tau)^\dagger \circ f^\dagger \circ h_n^\sigma$ (where $(-)^\dagger$ is Kleisli extension). The other cases are similar. ◀

From the above it is clear that we can inductively define a system of subobjects $[\![\sigma]\!]_n \rightarrowtail [\![\sigma]\!]$, each with only finitely many points, such that the composite $[\![\sigma]\!]_n \rightarrowtail [\![\sigma]\!] \to L_{\mathcal{G}}[\![\sigma]\!]$ admits a retraction in the Kleisli category, making $[\![\sigma]\!]_n$ a splitting of the idempotent h^σ_n. By construction, these objects satisfy $[\![\sigma \to \tau]\!]_n \cong [\![\sigma]\!]_n \Rightarrow L_{\mathcal{G}}[\![\tau]\!]_n$, $[\![\sigma + \tau]\!]_n \cong [\![\sigma]\!]_n + [\![\tau]\!]_n$, and $[\![\sigma \times \tau]\!]_n \cong [\![\sigma]\!]_n \times [\![\tau]\!]_n$. Treating contexts just as product types in the obvious way, we can think of the partial types $[\![\sigma]\!]_n$ as giving a "truncated" interpretation of $\mathsf{PCF_v}$-types. A computation $\Gamma \vdash^{\mathsf{c}} t : \sigma$ denotes the morphism $[\![\Gamma]\!]_n \to L_{\mathcal{G}}[\![\sigma]\!]_n$ given by sequencing $[\![t]\!] : [\![\Gamma]\!] \to L_{\mathcal{G}}[\![\sigma]\!]$ with the appropriate section and retraction.

The next lemma tells us that every type σ is the "supremum" of a chain of partial types. If we choose a point x of $[\![\sigma]\!]$, $h^\sigma(n,x) = h^\sigma_n(x)$ is its level n approximation. The existence of h^σ means these approximations form a chain, and H^σ witnesses that the supremum is x.

▶ **Lemma 7.5.** *The assignment $h^\sigma(n,x) = h^\sigma_n(x)$ defines a morphism $h^\sigma : \omega_{\mathcal{G}} \times [\![\sigma]\!] \to L_{\mathcal{G}}[\![\sigma]\!]$ in $\mathcal{G}$, whose unique extension $H^\sigma : \bar{\omega}_{\mathcal{G}} \times [\![\sigma]\!] \to L_{\mathcal{G}}[\![\sigma]\!]$ satisfies $H^\sigma(\infty, x) = x$.*

Proof notes. The first claim uses the fact that all types are interpreted as concrete sheaves. The second claim is proved by induction on σ. For example, when $\sigma = \mathsf{nat}$, $H^{\mathsf{nat}}(-,n)$ is eventually constant with value n. For $\sigma \to \tau$, $H^{\sigma\to\tau}(-,f)$ is the sequence with $n \mapsto H^\tau(n,-)^\dagger \circ f^\dagger \circ H^\sigma(n,-)$ (where $(-)^\dagger$ is Kleisli extension). For each $x \in |[\![\sigma]\!]|$, this is the diagonal of the square array $m, n \mapsto H^\tau(m)^\dagger(f^\dagger(H^\sigma(n,x)))$, so one can take the limit separately in the two indices. ◀

7.2 Definability for partial types in $\mathcal{G}$

We show that, for each n, one of the sites used to obtain $\mathcal{G}$ was a correct prediction, and so our summed site already contains the truncated interpretation of $\mathsf{PCF_v}$-types. Let $\mathcal{C}_n$ be the category whose objects are types σ and whose morphisms $\sigma \to \tau$ are morphisms $[\![\sigma]\!]_n \to L[\![\tau]\!]_n$ which arise as the interpretation of a term $x : \sigma \vdash^{\mathsf{c}} t : \tau$. Let $F_n : \mathcal{C}_n \to \mathsf{SSP}_\perp$ map σ to $(|[\![\sigma]\!]_n|, S^{\sigma,n})$ where $P \in S^{\sigma,n}$ iff P is the collection of non-empty fibres of a map $[\![\sigma]\!]_n \to L[\![\mathsf{nat}]\!]$ which arises as the interpretation of a term $x : \sigma \vdash t : \mathsf{nat}$. We treat contexts Γ as objects of $\mathcal{C}_n$ by identifying them with a product type.

Although the global elements of the sheafified representable $y((\sigma, U)_{\mathcal{C}_n,F_n})$ are naturally identified with U, it is not clear that there is a morphism $y((\sigma,U)_{\mathcal{C}_n,F_n}) \to [\![\sigma]\!]_n$ corresponding to the inclusion. Nevertheless, since the latter is a concrete sheaf there is an identification of $[\![\sigma]\!]_n((\Gamma,U)_{\mathcal{C}_n,F_n})$ with a subset of the functions $U \to |[\![\sigma]\!]_n|$. Moreover, $L_{\mathcal{G}}([\![\sigma]\!]_n)((\Gamma,U)_{\mathcal{C}_n,F_n})$ can be identified with a subset of the partial functions $U \rightharpoonup |[\![\sigma]\!]_n|$, whose domain $U' \subseteq U$ is an element of $\Delta_{\mathcal{G}}((\Gamma,U))$, i.e. definable by a computation $\Gamma \vdash^{\mathsf{c}} t : 1$.

For convenience, let us write $\mathsf{in}_n : \mathcal{I}_{\mathcal{C}_n,F_n} \hookrightarrow \mathcal{I}$ for the inclusion of sites. Recall from Lemma 6.11 that in_n induces an adjoint triple $(\mathsf{in}_n)_! \dashv (\mathsf{in}_n)^* \dashv (\mathsf{in}_n)_*$, where $(\mathsf{in}_n)_!$ preserves finite limits and representables, and $(\mathsf{in}_n)_!, (\mathsf{in}_n)_*$ are full and faithful. The next lemma is crucial and is proved in Appendix A.3. Note that, in particular, it implies that every point of $[\![\sigma]\!]_n$ is the interpretation of a closed value.

▶ **Lemma 7.6.** *There is an isomorphism $y(\sigma, |[\![\sigma]\!]_n|) \to (\mathsf{in}_n)^*[\![\sigma]\!]_n$ in $\mathsf{Sh}(\mathcal{I}_{\mathcal{C}_n,F_n}, \mathcal{J}_{\mathcal{C}_n,F_n})$.*

▶ **Theorem 7.7** (Full abstraction)**.** *If two $\mathsf{PCF_v}$ computations $\Gamma \vdash^{\mathsf{c}} t, t' : \sigma$ are contextually equivalent then $[\![t]\!] = [\![t']\!]$, and similarly for values.*

Proof notes. The computations t, t' denote morphisms $[\![\Gamma]\!] \to L_{\mathcal{G}}[\![\sigma]\!]$. By an induction on σ, $[\![t]\!]$ and $[\![t']\!]$ agree on their restrictions to $[\![\Gamma]\!]_n$: for the function type $\sigma \to \tau$ ones uses the fact that every point of $[\![\sigma]\!]_n$ is definable and applies the induction hypothesis for τ. It follows that $[\![t]\!]^\dagger \circ H^\Gamma$ and $[\![t']\!]^\dagger \circ H^\Gamma$ agree on $\omega_{\mathcal{G}} \times [\![\Gamma]\!]$ (where $(-)^\dagger$ is Kleisli extension). But $L_{\mathcal{G}}[\![\sigma]\!]$ is $L_{\mathcal{G}}$-complete, so they also agree on $\bar{\omega}_{\mathcal{G}} \times [\![\Gamma]\!]$. Evaluating at ∞, we get $[\![t]\!] = [\![t']\!]$ by Lemma 7.5. The proof for values is similar. ◀

8 Related work and research directions

8.1 Comparison with the model of Riecke-Sandholm

Our fully abstract model of $\mathsf{PCF_v}$, $\mathcal{G}$, is heavily inspired by the fully abstract model for call-by-value FPC of Riecke and Sandholm [39], itself inspired by [36, 43] (see also subsequent work [29, 30, 46, 22]). Our sites $\mathcal{I}_{\mathcal{C},F}$ (Definition 6.6) are close to the "varying arities" of [39]; their index category $\mathcal{C}$ [39, §3.4] corresponds to our $\mathcal{C}$, and their "path theory" S^w corresponds to our SSP structure $S^{F(w)}$.

The objects of our $\mathcal{G}$ are in particular $\mathbb{V}$-sets, and if we insist that they are moreover ω-cpo's then the Kleisli category of L is almost equivalent to the category $\mathcal{RCPO}$ of [39]. Our sheaf condition corresponds to the structure of a "computational relation" from [39].

There are some technical differences: they use directed cpo's rather than ω-cpo's, and they did not require morphisms $f : v \to w \in \mathcal{C}$ to pull back a partition from S^w to a partition from S^v. But at a higher level, while it is possible that Riecke and Sandholm had sheaves and monads in mind, those concepts which are central to this paper are not explicit in [39].

8.2 Comparison with work on "Synthetic Domain Theory"

The vision of synthetic domain theory (SDT) is that, by working in an intuitionistic set theory, we can interpret types as sets and assume that all functions are suitably continuous. Our work intersects with many of the methods of this theory, even if our motivation is less philosophical and rather to use sheaf categories to build and relate models. We comment on several aspects of SDT.

Partiality. Our treatment of partial maps (§3) is based on [40] and our development of lifting monads on [34, 33]. In recent years the *restriction categories* of [4, 5] have become increasingly popular, although these can be related to earlier methods. Our construction of $\mathcal{I}_{\mathcal{C},F}$ is reminiscent of the "splitting" of a restriction category, and our construction of $\mathsf{Sh}(\mathcal{I}_{\mathcal{C},F}, \mathcal{J}_{\mathcal{C},F})$ is reminiscent of the free cocompletion of [26, 15].

Recursion. Our treatment of recursion (§2) perhaps originates in [9, 10] or [28, §5]; more abstract treatments of the latter were given later [35, 38]. Orthogonality also plays a central role in the representation theorem of [12]. SDT permits an alternative, more refined analysis of recursion, based on "replete objects' [19, 47], which we have not yet pursued.

Sheaf categories. Much work on SDT has focused on realizability categories, but there has been substantial work on sheaf models too, beginning from Scott [41], and running through to the notion of a "Grothendieck model of SDT' [9] which roughly agrees with our notion of normal model (Definition 4.1). The early idea of a "Scott topos" was to take sheaves on a model of the untyped λ-calculus; this is further developed in [40, §7.2] and [47, §5]. Later work considered the monoid $\mathbb{V}$ [14, 13] and a stable version of it [13, 9], which is a step towards sequentiality. Sheaf constructions are also very relevant to definability arguments in terminating, typed calculi [11, 21], so it is perhaps surprising that fully abstract sheaf models of SDT have not been considered previously. Going beyond PCF, one point is that sheaf categories arguably cannot support a small complete category, which is useful for impredicative polymorphism [44, Ax. 3], although there are sheaf models of System F nonetheless [37, Thm. 4.6].

8.3 Summary and outlook

We have given a sheaf theoretic model of a call-by-value PCF (§4) which is fully abstract (§7). Our model uses a categorical framework for partiality (§3) and recursion (§2), and is based on combining sites for sequentiality (§6) with a site for recursion (§5). The way that sites for sheaves can be combined and compared plays a crucial role. Looking beyond this work, we anticipate that in the future it will be informative to use the flexibility of sheaves and sites to compare and combine the methods for recursion here with recent sheaf methods for other aspects of programming (e.g. [16, 18, 27, 42]).

Acknowledgements. One personal starting point was recent work on Kripke logical relations models for full abstraction in languages with effects but without recursion [22, 23]. We are grateful to the authors for discussions, although we have to leave combining recursion with other effects for future work. We thank Marcelo Fiore, Mathieu Huot, Hugo Paquet, Philip Saville, Thomas Streicher, and the anonymous reviewers for helpful feedback.

References

1 Marc Bezem, Thierry Coquand, and Simon Huber. A model of type theory in cubical sets. In *Proc. TYPES 2013*, 2013.
2 P Borthelle, T Hirschowitz, and A Lafont. A cellular Howe theorem. In *Proc. LICS 2020*, 2020.
3 Anna Bucalo, Carsten Führmann, and Alex Simpson. An equational notion of lifting monad. *Theoret. Comput. Sci.*, 294:31–60, 2003.
4 J.R.B. Cockett and Stephen Lack. Restriction categories I: categories of partial maps. *Theoret. Comput. Sci.*, 270(1):223–259, 2002.
5 J.R.B. Cockett and Stephen Lack. Restriction categories II: partial map classification. *Theoret. Comput. Sci.*, 294(1):61–102, 2003.
6 Roy L Crole and Andrew M Pitts. New foundations for fixpoint combinators. In *Proc. LICS 1990*, 1990.
7 Eduardo J. Dubuc. Concrete quasitopoi. In *Applications of Sheaves: Proceedings of the Research Symposium on Applications of Sheaf Theory to Logic, Algebra, and Analysis, Durham, July 9–21, 1977*, pages 239–254. Springer Berlin Heidelberg, 1979.
8 Roy Dyckhoff and Walter Tholen. Exponentiable morphisms, partial products and pullback complements. *J. Pure Appl. Algebra*, 49(1-2):103–116, 1987.
9 M Fiore and G Plotkin. An extension of models of axiomatic domain theory to models of synthetic domain theory. In *Proc. CSL 1996*, 1997.
10 M. Fiore, G. Plotkin, and J. Power. Complete cuboidal sets in axiomatic domain theory. In *Proc. LICS 1997*, pages 268–279, 1997.
11 M. P. Fiore and A. K. Simpson. Lambda definability with sums via Grothendieck logical relations. In *TLCA'99*, 1999.
12 Marcelo Fiore. Enrichment and representation theorems for categories of domains and continuous functions. Available at the author's website, 1996.
13 Marcelo P Fiore and Giuseppe Rosolini. Two models of synthetic domain theory. *J. Pure Appl. Algebra*, 116:151–162, 1997.
14 Marcelo P Fiore and Giuseppe Rosolini. Domains in H. *Theoret. Comput. Sci.*, 264:171–193, 2001.
15 Richard Garner and Daniel Lin. Cocompletion of restriction categories. *Theory Appl. Categ.*, 35(22):809–844, 2020.
16 C Heunen, O Kammar, S Staton, and H Yang. A convenient category for higher-order probability theory. In *Proc. LICS 2017*, 2017.
17 Gérard Huet. *Formal Structures in Computation and Deduction.* Unpublished, 1986.

18 M Huot, S Staton, and M Vákár. Correctness of automatic differentiation via diffeologies and categorical gluing. In *Proc. FOSSACS 2020*, 2020.

19 J M E Hyland. First steps in synthetic domain theory. In *Proc. Category Theory, Como*, Lect. Notes Math., pages 131–156. Springer, 1990.

20 P. T. Johnstone. *Sketches of an elephant: a Topos theory compendium.* Oxford logic guides. Oxford Univ. Press, 2002.

21 Achim Jung and Jerzy Tiuryn. A new characterization of lambda definability. In Marc Bezem and Jan Friso Groote, editors, *Proc. TLCA'93*, volume 664 of *Lecture Notes in Computer Science*, pages 245–257. Springer, 1993. doi:10.1007/BFb0037110.

22 O. Kammar and S. Katsumata. A modern perspective on the O'Hearn-Riecke model. Workshop on Syntax and Semantics of Low-Level Languages (LOLA), 2019.

23 Ohad Kammar, Shinya Katsumata, and Philip Saville. Full abstraction à la O'Hearn & Riecke for call-by-value with sums and effects. Manuscript, January 2021.

24 Paul B Levy. Amb breaks well-pointedness, ground amb doesn't. In *Proc. MFPS 2007*, 2007.

25 Paul B Levy, J Power, and H Thielecke. Modelling environments in call-by-value programming languages. *Inform. Comput.*, 185(2):182–210, 2003.

26 Daniel Lin. Presheaves over a join restriction category. *Applied Categorical Structures*, 27(3):289–310, 2019.

27 B Lindenhovius, M Mislove, and V Zamdzhiev. Mixed linear and non-linear recursive types. In *Proc. ICFP 2019*, 2019.

28 John R Longley and Alex K Simpson. A uniform approach to domain theory in realizability models. *Math. Struct. Comput. Sci.*, 7(5):469–505, 1997.

29 M. Marz. *A Fully Abstract Model for Sequential Computation.* PhD thesis, Technische Universität Darmstadt, 2000.

30 Michael Marz. A fully abstract model for sequential computation. *Electronic Notes in Theoretical Computer Science*, 35:133–152, 2000. Workshop on Domains IV.

31 Robin Milner. Fully abstract models of typed λ-calculi. *Theoret. Comput. Sci.*, 4(1):1–22, 1977.

32 Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93:55–92, 1991.

33 Philip S. Mulry. Monads and algebras in the semantics of partial data types. *Theoretical Computer Science*, 99(1):141–155, 1992.

34 Philip S. Mulry. Partial map classifiers and partial cartesian closed categories. *Theoretical Computer Science*, 136(1):109–123, 1994.

35 Jaap van Oosten and Alex K Simpson. Axioms and (counter)examples in synthetic domain theory. *Annals Pure Appl. Logic*, 104:233–278, 2000.

36 P. W. O'Hearn and J. G. Riecke. Kripke logical relations and PCF. *Inf. Comput.*, 120(1):107–116, 1995.

37 A M Pitts. Polymorphism is set theoretic, constructively. In *Proc. CTCS 1987*, volume 283 of *LNCS*. Springer, 1987.

38 Bernhard Reus and Thomas Streicher. General synthetic domain theory – a logical approach. *Math. Struct. Comput. Sci.*, 9(2):177–223, 1999.

39 J. G. Riecke and A. Sandholm. A relational account of call-by-value sequentiality. *Inf. Comput.*, 179(2):296–331, 2002.

40 Giuseppe Rosolini. *Continuity and effectiveness in topoi.* PhD thesis, University of Oxford, 1986.

41 Dana S Scott. Relating theories of the λ-calculus. In *To H B Curry: essays in combinatory logic, lambda calculus and formalisms*, pages 403–450. Academic Press, 1980.

42 B Sherman, J Michel, and M Carbin. λ_s: Computable semantics for differentiable programming with higher-order functions and datatypes. In *Proc. POPL 2021*, 2021.

43 K. Sieber. Reasoning about sequential functions via logical relations. In *Applications of Categories in Computer Science: Proceedings of the London Mathematical Society Symposium, Durham 1991*, London Mathematical Society Lecture Note Series, page 258–269. Cambridge University Press, 1992.

44 Alex Simpson and Giuseppe Rosolini. Using synthetic domain theory to prove operational properties of a polymorphic programming language based on strictness. Unpublished, 2004.

45 Alex K. Simpson. Computational adequacy in an elementary topos. In Georg Gottlob, Etienne Grandjean, and Katrin Seyr, editors, *Computer Science Logic*, pages 323–342, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

46 T. Streicher. *Domain-theoretic foundations of functional programming.* World Scientific, 2006.

47 Paul Taylor. The fixed point property in synthetic domain theory. In *Proc. LICS 1991*, 1991.

A Proofs of technical results

A.1 Fixed Points

Let $X \in \mathbb{C}$ be such that LX is a L-complete object. Then, for any map $f : \Gamma \times LX \to LX$, we can construct a map $\xi_f : \Gamma \to LX$ with $f(\rho, \xi_f(\rho)) = \xi_f(\rho)$ as follows.

First define a family of maps $\mathsf{ap}_n : \Gamma \times L^n 1 \to LX$ as follows: $\mathsf{ap}_0(\rho, *) = \bot_X$ and ap_{n+1} the following composite:

$$\Gamma \times L^{n+1} 1 \xrightarrow{\sigma_{\Gamma, L^n 1}} L(\Gamma \times L^n 1) \xrightarrow{L(\pi_1, \mathsf{ap}_n)} L(\Gamma \times LX) \xrightarrow{L(f)} LLX \xrightarrow{\mu_X} LX.$$

The sequence (ap_n) defines a map $\mathsf{ap}_\omega : \Gamma \times \omega \to LX$ because it forms a cocone for diagram 2. For this we can show by induction on n that $\mathsf{ap}_{n+1} \circ (\mathsf{id}_\Gamma \times L^n(\bot_1)) = \mathsf{ap_n}$.

Next we show that

$$\mathsf{ap}_\omega \circ (\mathsf{id}_\Gamma \times \mathsf{succ}_\omega) = f \circ (\pi_1, \mathsf{ap}_\omega).$$

The sequence of maps $\mathsf{ap}_{n+1} \circ (\mathsf{id}_\Gamma \times \eta_{L^n 1})$ forms a cocone with apex LX for diagram 2, whose comparison arrow is $\mathsf{ap}_\omega \circ (\mathsf{id}_\Gamma \times \mathsf{succ}_\omega)$. Similarly the sequence $f \circ (\pi_1, \mathsf{ap}_n)$ forms a cocone with comparison arrow $f \circ (\pi_1, \mathsf{ap}_\omega)$. So it suffices to show $\mathsf{ap}_{n+1} \circ (\mathsf{id}_\Gamma \times \eta_{L^n 1}) = f \circ (\pi_1, \mathsf{ap}_n)$ which is not hard.

Let $\mathsf{ap}_{\bar\omega} : \Gamma \times \bar\omega \to LX$ be the unique extension of ap_ω. Observe that $\mathsf{ap}_{\bar\omega} \circ (\mathsf{id}_\Gamma \times \mathsf{succ}_{\bar\omega}) = f \circ (\pi_1, \mathsf{ap}_{\bar\omega})$ as well. Then let $\xi_f(\rho) = \mathsf{ap}_{\bar\omega}(\rho, \infty)$, and now

$$\xi_f(\rho) = \mathsf{ap}_{\bar\omega}(\rho, \infty) = \mathsf{ap}_{\bar\omega}(\rho, \mathsf{succ}_{\bar\omega}(\infty)) = f(\rho, \mathsf{ap}_{\bar\omega}(\rho, \infty)) = f(\rho, \xi_f(\rho))$$

as required.

Assume, as in Theorem 2.2, that $X \in \mathbb{C}$ is an L-algebra and LX an L-complete object. Consider a map $g : \Gamma \times X \to X$. We will construct a fixed point $\phi_g : \Gamma \to X$ for g.

Using the algebra structure of X, (X, α), we can construct a map:

$$\Gamma \times LX \xrightarrow{1 \times \alpha} \Gamma \times X \xrightarrow{g} X \xrightarrow{\eta} LX.$$

Then we can use the result from the previous paragraph to get a fixed point $\xi : \Gamma \to LX$ of this map. So the candidate fixed point for g will be $\phi_g = \alpha \circ \xi$. And indeed:

$$\begin{aligned} g \circ (1, \alpha \circ \xi) &= \alpha \circ \big(\eta \circ g \circ (1 \times \alpha)\big) \circ (1, \xi) && \text{because } \alpha \text{ is an algebra} \\ &= \alpha \circ \xi && \text{because } \xi \text{ is a fixed point.} \end{aligned}$$

Assume, as in Corollary 2.3, that $L(LB^A)$ is an L-complete object and $M : \Gamma \times LB^A \times A \to LB$ is a morphism. To construct a fixed point $\mathsf{rec}_M : \Gamma \to LB^A$ for M, notice that LB^A is an algebra for L because L is strong, so we have:

$$L(LB^A) \times A \xrightarrow{\sigma_{A,LB^A}} L(LB^A \times A) \xrightarrow{L\mathsf{ev}} LLB \xrightarrow{\mu_B} LB.$$

We can curry M to get $\Gamma \times LB^A \to LB^A$ and then construct rec_M as in the previous paragraph.

A.2 Adequacy for vSet

▶ **Theorem 5.5.** *The pair* $(\mathsf{vSet}, \Delta_{\mathsf{vSet}})$ *gives a sound and adequate model of* $\mathsf{PCF_v}$.

- *Soundness:* $t \Downarrow_\tau v \implies [\![t]\!] = \eta_{[\![\tau]\!]} \circ [\![v]\!] \in L_{\mathsf{vSet}}[\![\tau]\!]$.
- *Adequacy: if* τ *is a ground type,* $[\![t]\!] = \eta_{[\![\tau]\!]} \circ [\![v]\!] \implies t \Downarrow_\tau v$.

Proof sketch. Soundness is proved easily by induction on the definition of $\Downarrow_\tau$.

Adequacy is proved using the standard method for cpo's. We define a logical relation by induction on types that says when a term is approximated by an element of the model: $\lhd^{\mathsf{val}}_\tau \subseteq [\![\tau]\!] \times \mathsf{Val}_\tau$ and $\lhd^{\mathsf{comp}}_\tau \subseteq L_{\mathsf{vSet}}[\![\tau]\!] \times \mathsf{Comp}_\tau$. For example:

$$\lhd^{\mathsf{val}}_{\tau\to\tau'} = \{(d,v) \mid \forall a \in [\![\tau]\!],\ w \in \mathsf{Val}_\tau.\ a \lhd^{\mathsf{val}}_\tau w \implies (d\ a) \lhd^{\mathsf{comp}}_{\tau'} (v\ w)\}$$
$$\lhd^{\mathsf{comp}}_\tau = \{(d,t) \mid \text{if } d = \eta_{[\![\tau]\!]} \circ d' \text{ then } \exists w.\ t \Downarrow_\tau w \text{ and } d' \lhd^{\mathsf{val}}_\tau w\}.$$

Then we prove the fundamental property of this logical relation and show it is enough to obtain adequacy.

The fundamental property is proved by induction on terms. For the rec case we prove by induction on types that all subobjects of the form $\{(-) \lhd^{\mathsf{comp}}_{\tau''} t''\}$ are closed under sups of chains. (Here a chain is a map $\omega \to L_{\mathsf{vSet}}[\![\tau'']\!]$, and a chain with a lub is $\bar\omega \to L_{\mathsf{vSet}}[\![\tau'']\!]$.) This replaces the proof from cpo's that the logical relation is an admissible subset. ◀

A.3 A fully abstract model of $\mathsf{PCF_v}$

▶ **Proposition 7.2.** *$L_\mathcal{G}(N_\mathcal{G})$ is $L_\mathcal{G}$-complete.*

Proof. Consider an extension problem $f : (\mathsf{in}_j)_! y(c) \times \omega_\mathcal{G} \to L_\mathcal{G}(N_\mathcal{G})$, where $c \in \mathbb{C}_j$, and consider two cases for j. Firstly, if j is $\mathbb{V}$, then $(\mathsf{in}_j)^* f : y(c) \times \omega_{\mathsf{vSet}} \to L_{\mathsf{vSet}}(N_{\mathsf{vSet}})$ has a unique extension to a map $y(c) \times \bar\omega_{\mathsf{vSet}} \to L_{\mathsf{vSet}}(N_{\mathsf{vSet}})$ in vSet, where the underlying function on points $\phi : |c| \times |\bar\omega_{\mathsf{vSet}}| \to |\mathbb{N} \cup \{\bot\}|$ is given by taking $\phi(x,\infty)$ to be the eventual value of $\phi(x,n)$ as $n \to \infty$. It remains to check that ϕ underlies a natural transformation $(\mathsf{in}_j)_! y(c) \times \bar\omega_\mathcal{G} \to L_\mathcal{G}(N_\mathcal{G})$ in the sheaf category $\mathcal{G}$. This is so since if $d \in \mathbb{C}_k$ with $k \neq j$ then $|d|$ is finite and thus for any pair $(g,h) \in ((\mathsf{in}_j)_!(y(c)) \times \bar\omega_\mathcal{G})(d)$ we have $(\phi \circ (g,h))(y) = \phi(g(y), \min\{N, h(y)\}) = f(g(y), \min\{N, h(y)\}) \in L_{\mathsf{vSet}}(N_{\mathsf{vSet}})(d)$ for some $N \in \mathbb{N}$ not depending on $y \in |d|$. Secondly, if j is of the form $(\mathcal{I}_{\mathcal{C},F}, \mathcal{J}_{\mathcal{C},F})$ for some faithful functor $F : \mathcal{C} \to \mathsf{SSP}_\bot$, then since $|c|$ is finite f factorizes as a retraction $(\mathsf{in}_j)_! y(c) \times \omega_\mathcal{G} \twoheadrightarrow (\mathsf{in}_j)_! y(c) \times {L_\mathcal{G}}^n 1$ for some n followed by a map $(\mathsf{in}_j)_! y(c) \times {L_\mathcal{G}}^n 1 \to \Delta_\mathcal{G}$. This gives one possible extension of f to $(\mathsf{in}_j)_! y(c) \times \bar\omega_\mathcal{G}$. Since it must also be a morphism of the underlying v-sets, it is unique. ◀

We will need the following result on preservation of exponentials, which can be extracted from the proof of Lemma A1.5.8 in [20].

▶ **Proposition A.1** (Frobenius reciprocity). *Let $F : \mathbb{C} \to \mathbb{D}$ be a functor between cartesian closed categories with a left adjoint $L \dashv F$. Then F preserves a given exponential $A \Rightarrow C$ iff, for all $B \in \mathbb{D}$, C is right-orthogonal to the canonical map $L(B \times FA) \to LB \times A$.*

▶ **Lemma 7.6.** *There is an isomorphism* $y(\sigma, |[\![\sigma]\!]_n|) \to (\mathsf{in}_n)^*[\![\sigma]\!]_n$ *in* $\mathsf{Sh}(\mathcal{I}_{\mathcal{C}_n,F_n}, \mathcal{J}_{\mathcal{C}_n,F_n})$.

Proof. First note that $(\mathsf{in}_n)^*$ is faithful on maps into concrete sheaves, and while not in general full, it is bijective on global elements. We proceed by induction on σ. Since $[\![1]\!]_n$ is a terminal object and $[\![0]\!]_n$ is an initial object, both are preserved by $(\mathsf{in}_n)^*$ so the claim there is trivial. Similarly, $(\mathsf{in}_n)^*$ preserves sums and $y : \mathcal{I}_{\mathcal{C}_n,F_n} \to \mathsf{Sh}(\mathcal{I}_{\mathcal{C}_n,F_n}, J_{\mathcal{C}_n,F_n})$ preserves sums of types, hence the base case of $\sigma = \mathsf{nat}$ and the inductive case $\sigma = \sigma_1 + \sigma_2$ both hold. In the case of the product type $\sigma = \sigma_1 \times \sigma_2$, we have first to observe that $(\sigma \times \tau, |[\![\sigma \times \tau]\!]|)$ is actually a product in $\mathcal{I}_{\mathcal{C}_n,F_n}$ since all global elements of $[\![\sigma_1]\!]_n$ and $[\![\sigma_2]\!]_n$ are definable; the claim then follows since $(\mathsf{in}_n)^*$ preserves products.

The interesting case is the function types, since $(\mathsf{in}_n)^*$ does not preserve exponentials in general, but we will show that it does preserve the exponentials $[\![\sigma \to \tau]\!]_n \cong [\![\sigma]\!]_n \Rightarrow L_{\mathcal{G}}[\![\tau]\!]_n$. This will suffice since we now show that $y(\sigma \to \tau, |[\![\sigma \to \tau]\!]_n|)$ is an exponential. Since $(\mathsf{in}_n)^*$ commutes with the lifting monad, the induction hypothesis implies that $(\mathsf{in}_n)^*$ is full and faithful on maps $[\![\sigma]\!]_n \to L_{\mathcal{G}}[\![\tau]\!]_n$, and hence all points of $[\![\sigma \to \tau]\!]_n$ are definable. Then, for any $(\Gamma, U) \in \mathcal{I}_n$, each map $f : y(\Gamma, U) \times y(\sigma, |[\![\sigma]\!]_n|) \to L_{\mathcal{C}_n,F_n} y(\tau, |[\![\tau]\!]_n|)$ has an underlying $f_1 : y(\Gamma \times \sigma, U \times |[\![\sigma]\!]_n|) \to L_{\mathcal{C}_n,F_n} y(\tau, |[\![\tau]\!]_n|)$ given by precomposition with $y(\Gamma \times \sigma, U \times |[\![\sigma]\!]_n|) \to y(\Gamma, U) \times y(\sigma, |[\![\sigma]\!]_n|)$ and thus is definable. Moreover, every definable function does give a natural transformation $y(\Gamma, U) \times y(\sigma, |[\![\sigma]\!]_n|) \to L_{\mathcal{C}_n,F_n} y(\tau, |[\![\tau]\!]_n|)$, whence one may deduce that $y(\sigma, |[\![\sigma]\!]_n|) \Rightarrow L_{\mathcal{C}_n,F_n} y(\tau, |[\![\tau]\!]_n|) \cong y(\sigma \to \tau, |[\![\sigma \to \tau]\!]_n|)$.

Now we use Generalized Frobenius reciprocity to show that $(\mathsf{in}_n)^*([\![\sigma]\!]_n \Rightarrow L_{\mathcal{G}}[\![\tau]\!]_n) \cong (\mathsf{in}_n)^*([\![\sigma]\!]_n) \Rightarrow (\mathsf{in}_n)^*(L_{\mathcal{G}}[\![\tau]\!]_n)$. It clearly suffices to restrict attention to those "B" which are representables $y(\Gamma, U)$. Since $(\mathsf{in}_n)_!(y(\Gamma, U) \times y(\sigma, |[\![\sigma]\!]_n|)) \to (\mathsf{in}_n)_!(y(\Gamma, U)) \times [\![\sigma]\!]_n$ is surjective on points, we have the uniqueness part of orthogonality. Now, given a map $(\mathsf{in}_n)_! y(\Gamma, U) \times (\mathsf{in}_n)_! y(\sigma, |[\![\sigma]\!]_n|) \to L_{\mathcal{G}}[\![\tau]\!]_n$, by precomposition we get a map $(\mathsf{in}_n)_! y(\Gamma \times \sigma, U \times |[\![\sigma]\!]_n|) \to L_{\mathcal{G}}[\![\tau]\!]_n$ whence we deduce that the underlying function is definable. We must show that a definable function is a natural transformation $(\mathsf{in}_n)_!(y(\Gamma, U)) \times [\![\sigma]\!]_n \to L_{\mathcal{G}}[\![\tau]\!]_n$. It suffices to show the same thing with an unsheafified representable: i.e. to consider $\mathcal{I}(-, (\Gamma, U)_{\mathcal{C}_n,F_n}) \times [\![\sigma]\!]_n \to L_{\mathcal{G}}[\![\tau]\!]_n$. On objects of $X \in \mathcal{I}$ not in $\mathcal{I}_{\mathcal{C}_n,F_n}$, the set $\mathcal{I}(X, (\Gamma, U)_{\mathcal{C}_n,F_n}) \times [\![\sigma]\!]_n(X)$ is indeed mapped into $L_{\mathcal{E}}[\![\tau]\!]_n(X)$ since the left factor of the latter contains only constant functions. On objects $(\Gamma', U') \in \mathcal{I}_{\mathcal{C}_n,F_n}$, the same reasoning applies for constant functions $(\Gamma', U') \to (\Gamma, U)$, but for non-constant functions, which are by construction definable, we use the facts that every function in $[\![\sigma]\!]_n(\Gamma', U')$ is definable and that the definable functions are closed under pairing and composition. ◀

B Typing rules and operational semantics for $\mathsf{PCF_v}$

In this section we provide the full type system and operational semantics for the $\mathsf{PCF_v}$ language. Recall the syntax of $\mathsf{PCF_v}$:

$$\begin{aligned}
\text{Types:}\quad & \tau ::= 0 \mid 1 \mid \mathsf{nat} \mid \tau + \tau \mid \tau \times \tau \mid \tau \to \tau \\
\text{Values:}\quad & v, w ::= x \mid \star \mid \mathsf{inl}\, v \mid \mathsf{inr}\, v \mid (v, v) \mid \mathsf{zero} \mid \mathsf{succ}(v) \mid \lambda x.\, t \mid \mathsf{rec}\, f\, x.\, t \\
\text{Computations:}\quad & t ::= \mathsf{return}\, v \mid \mathsf{case}\, v\, \mathsf{of}\, \{\mathsf{inl}\, x \to t,\ \mathsf{inr}\, y \to t'\} \mid \pi_1 v \mid \pi_2 v \mid v\ w \\
& \quad \mid \mathsf{case}\, v\ \mathsf{of}\ \{\mathsf{zero} \to t,\ \mathsf{succ}(x) \to t'\} \mid \mathsf{let}\, x = t\, \mathsf{in}\, t'
\end{aligned}$$

The typing relation is the least relation closed under the following rules:

$$\frac{}{\Gamma, x : \tau \vdash^{\mathbf{v}} x : \tau} \qquad \frac{}{\Gamma \vdash^{\mathbf{v}} \star : 1} \qquad \frac{\Gamma \vdash^{\mathbf{v}} v : \tau}{\Gamma \vdash^{\mathbf{v}} \mathsf{inl}\, v : \tau + \tau'} \qquad \frac{\Gamma \vdash^{\mathbf{v}} v : \tau'}{\Gamma \vdash^{\mathbf{v}} \mathsf{inr}\, v : \tau + \tau'}$$

$$\frac{\Gamma \vdash^{\mathbf{v}} v : \tau \qquad \Gamma \vdash^{\mathbf{v}} v' : \tau'}{\Gamma \vdash^{\mathbf{v}} (v, v') : \tau \times \tau'} \qquad \frac{}{\Gamma \vdash^{\mathbf{v}} \mathsf{zero} : \mathsf{nat}} \qquad \frac{\Gamma \vdash^{\mathbf{v}} v : \mathsf{nat}}{\Gamma \vdash^{\mathbf{v}} \mathsf{succ}(v) : \mathsf{nat}}$$

$$\frac{\Gamma,\, x : \tau \vdash^{\mathbf{c}} t : \tau'}{\Gamma \vdash^{\mathbf{v}} \lambda x.\, t : \tau \to \tau'} \qquad \frac{\Gamma,\, f : \tau \to \tau',\, x : \tau \vdash^{\mathbf{c}} t : \tau'}{\Gamma \vdash^{\mathbf{v}} \mathsf{rec}\, f\, x.\, t : \tau \to \tau'} \qquad \frac{\Gamma \vdash^{\mathbf{v}} v : 0}{\Gamma \vdash^{\mathbf{c}} \mathsf{case}\, v\, \mathsf{of}\ \{\} : \tau}$$

$$\frac{\Gamma \vdash^{\mathbf{v}} v : \tau}{\Gamma \vdash^{\mathbf{c}} \mathsf{return}\, v : \tau} \qquad \frac{\Gamma \vdash^{\mathbf{v}} v : \tau + \tau' \qquad \Gamma, x : \tau \vdash^{\mathbf{c}} t : \sigma \qquad \Gamma, y : \tau' \vdash^{\mathbf{c}} t' : \sigma}{\Gamma \vdash^{\mathbf{c}} \mathsf{case}\, v\, \mathsf{of}\, \{\mathsf{inl}\, x \to t,\ \mathsf{inr}\, y \to t'\} : \sigma}$$

$$\frac{\Gamma \vdash^{\mathbf{v}} v : \tau \times \tau'}{\Gamma \vdash^{\mathbf{c}} \pi_1 v : \tau} \qquad \frac{\Gamma \vdash^{\mathbf{v}} v : \tau \times \tau'}{\Gamma \vdash^{\mathbf{c}} \pi_2 v : \tau'} \qquad \frac{\Gamma \vdash^{\mathbf{v}} v : \tau \to \tau' \qquad \Gamma \vdash^{\mathbf{v}} w : \tau}{\Gamma \vdash^{\mathbf{c}} v\ w : \tau'}$$

$$\frac{\Gamma \vdash^{\mathbf{v}} v : \mathsf{nat} \qquad \Gamma \vdash^{\mathbf{c}} t : \tau \qquad \Gamma, x : \mathsf{nat} \vdash^{\mathbf{c}} t' : \tau}{\Gamma \vdash^{\mathbf{c}} \mathsf{case}\, v\ \mathsf{of}\ \{\mathsf{zero} \to t,\ \mathsf{succ}(x) \to t'\} : \tau} \qquad \frac{\Gamma \vdash^{\mathbf{c}} t : \tau \qquad \Gamma, x : \tau \vdash^{\mathbf{c}} t : \tau'}{\Gamma \vdash^{\mathbf{c}} \mathsf{let}\, x = t\, \mathsf{in}\, t' : \tau'}$$

The big-step operational semantics of $\mathsf{PCF}_{\mathsf{v}}$ is a family of relations, indexed by types, between closed computations and closed values. It is the least relation closed under the rules below:

$$\frac{}{\mathsf{return}\, v \Downarrow_\tau v} \qquad \frac{}{\pi_1(v, v') \Downarrow_\tau v} \qquad \frac{}{\pi_2(v, v') \Downarrow_\tau v'}$$

$$\frac{t[v/x] \Downarrow_\tau w}{\mathsf{case}\, \mathsf{inl}\, v\, \mathsf{of}\, \{\mathsf{inl}\, x \to t,\ \mathsf{inr}\, y \to t'\} \Downarrow_\tau w} \qquad \frac{t'[v/x] \Downarrow_\tau w}{\mathsf{case}\, \mathsf{inr}\, v\, \mathsf{of}\, \{\mathsf{inl}\, x \to t,\ \mathsf{inr}\, y \to t'\} \Downarrow_\tau w}$$

$$\frac{t[(\mathsf{rec}\, f\, x.\, t)/f,\ v/x] \Downarrow_\tau w}{(\mathsf{rec}\, f\, x.\, t)\ v \Downarrow_\tau w} \qquad \frac{t[v/x] \Downarrow_\tau w}{(\lambda x.\, t)\ v \Downarrow_\tau w} \qquad \frac{t \Downarrow_\tau v \qquad t'[v/x] \Downarrow_\tau w}{\mathsf{let}\, x = t\, \mathsf{in}\, t' \Downarrow_\tau w}$$

$$\frac{t \Downarrow_\tau w}{\mathsf{case}\, \mathsf{zero}\ \mathsf{of}\ \{\mathsf{zero} \to t,\ \mathsf{succ}(x) \to t'\} \Downarrow_\tau w}$$

$$\frac{t'[v/x] \Downarrow_\tau w}{\mathsf{case}\, \mathsf{succ}(v)\ \mathsf{of}\ \{\mathsf{zero} \to t,\ \mathsf{succ}(x) \to t'\} \Downarrow_\tau w}$$

Polymorphic Automorphisms and the Picard Group

Pieter Hofstra ✉
Dept. of Mathematics & Statistics, University of Ottawa, Canada

Jason Parker ✉
Department of Mathematics & Computer Science, Brandon University, Canada

Philip J. Scott[1] ✉
Dept. of Mathematics & Statistics, University of Ottawa, Canada

Abstract

We investigate the concept of definable, or inner, automorphism in the logical setting of partial Horn theories. The central technical result extends a syntactical characterization of the group of such automorphisms (called the covariant isotropy group) associated with an algebraic theory to the wider class of quasi-equational theories. We apply this characterization to prove that the isotropy group of a strict monoidal category is precisely its Picard group of invertible objects. Furthermore, we obtain an explicit description of the covariant isotropy group of a presheaf category.

2012 ACM Subject Classification Theory of computation → Categorical semantics; Theory of computation → Algebraic semantics; Theory of computation → Equational logic and rewriting

Keywords and phrases Partial Horn Theories, Monoidal Categories, Definable Automorphisms, Polymorphism, Indeterminates, Normal Forms

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.26

Related Version *Previous Version*: `https://arxiv.org/abs/2102.11081`

Funding *Pieter Hofstra*: Research funded by an NSERC Discovery Grant.
Jason Parker: Postdoctoral research funded by NSERC grant of R. Lucyshyn-Wright (Brandon).
Philip J. Scott: Research funded by an NSERC Discovery Grant.

Acknowledgements Pieter Hofstra would like to acknowledge illuminating discussions with Martti Karvonen and Eugenia Cheng. We would also like to thank the three anonymous referees for their insightful comments, corrections, and suggestions.

1 Introduction

In algebra, model theory, and computer science, one encounters the notion of *definable automorphism* (the nomenclature varies by discipline). In first-order logic for example (see e.g. [13]), an automorphism α of a model M is called *definable* (with parameters in M) when there is a formula $\varphi(x, y)$ in the ambient language (possibly containing constants from M) such that for all $a, b \in M$ we have

$$\alpha(a) = b \iff M \models \varphi(a, b).$$

The case of groups is instructive: for a group M, consider the formula $\varphi(x, y)$ given as

$$\varphi(x, y) : \; y = c^{-1}xc$$

for some $c \in M$. This defines an (inner) automorphism of M. Note that in this case the automorphism is also determined by a term $t(x) := c^{-1}xc$ via $a \mapsto t(a)$.

[1] corresponding author

These definable automorphisms have various interesting aspects: first of all, they are in some sense *polymorphic* or uniform. This means roughly that the same term t, possibly after replacing constants from M, can also define an automorphism of another model N. Secondly, the definable automorphisms can also provide a generalized notion of *inner automorphism*, even for theories where it does not make sense to speak of group-theoretic conjugation. Indeed, Bergman [1, Theorem 1] shows that in the category of groups, the definable group automorphisms, i.e. the inner automorphisms given by conjugation, can be characterized purely *categorically* by the fact that they extend naturally along any homomorphism. That is: an automorphism $\alpha : G \xrightarrow{\sim} G$ is inner precisely when for any homomorphism $m : G \to H$ there is an extension $\alpha_m : H \xrightarrow{\sim} H$ making diagram (a) commute and also making

$$(a)\quad \begin{tikzcd} G \arrow[r, "m"] \arrow[d, "\alpha"'] & H \arrow[d, "\alpha_m"] \\ G \arrow[r, "m"'] & H \end{tikzcd} \qquad\qquad (b)\quad \begin{tikzcd} H \arrow[r, "n"] \arrow[d, "\alpha_m"'] & K \arrow[d, "\alpha_{nm}"] \\ H \arrow[r, "n"'] & K \end{tikzcd}$$

diagram (b) commute for any further homomorphism $n : H \to K$, so that in particular $\alpha = \alpha_{\mathsf{id}_G}$ by diagram (a). If α is conjugation by $g \in G$, then α_m is conjugation by $m(g) \in H$. Conversely, given any system of group automorphisms $\{\alpha_m : H \xrightarrow{\sim} H \mid m : G \to H\}$ with $\alpha = \alpha_{\mathsf{id}_G}$ that makes diagrams (a) and (b) commute, Bergman shows that there is a unique element $s \in G$ such that α is given by conjugation with s. Bergman therefore refers to such a system $\{\alpha_m \mid m : G \to H\}$ as an *extended inner automorphism* of G.[2]

In categorical logic, we have a canonical method for studying this phenomenon. To any category $\mathbb{C}$, we may associate the functor

$$\mathcal{Z}_{\mathbb{C}} : \mathbb{C} \to \mathsf{Grp} \; ; \qquad \mathcal{Z}_{\mathbb{C}}(C) := \mathsf{Aut}(\pi : C/\mathbb{C} \to \mathbb{C}). \tag{1}$$

Let us unpack this. We have the co-slice category $C/\mathbb{C}$ whose objects are maps $C \to D$ and whose arrows are commutative triangles. The projection functor $\pi : C/\mathbb{C} \to \mathbb{C}$ sends $C \to D$ to D. We then consider the group of natural automorphisms of this projection functor, i.e. the group of *invertible* natural transformations $\alpha : \pi \Rightarrow \pi$. To give such an α is equivalent to giving, for each object $m : C \to D$ of $C/\mathbb{C}$, an automorphism $\alpha_m : D \xrightarrow{\sim} D$, subject to the naturality condition that for any composable pair $m : C \to D, n : D \to E$ in $\mathbb{C}$, we have $\alpha_{nm} n = n \alpha_m$ as in diagram (b) above. Thus, in Bergman's terminology, $\mathcal{Z}_{\mathbb{C}}(C)$ is the group of extended inner automorphisms of C. We call $\mathcal{Z}_{\mathbb{C}}$ the *(covariant) isotropy group (functor)* of $\mathbb{C}$. Another useful way of thinking about this group is by noticing that the assignment $C \mapsto \mathsf{Aut}(C)$ is generally not functorial, unless $\mathbb{C}$ is a groupoid. The isotropy group offers a remedy: the assignment $C \mapsto \mathcal{Z}_{\mathbb{C}}(C)$ *is* functorial, as is straightforward to check, and for each C there is a comparison homomorphism

$$\theta_C : \mathcal{Z}_{\mathbb{C}}(C) \to \mathsf{Aut}(C) \; ; \qquad \alpha \mapsto \alpha_{\mathsf{id}_C} \tag{2}$$

that sends an extended inner automorphism α to its component at the identity of C.[3] We can then turn Bergman's aforementioned result for the category Grp into a *definition* for an arbitrary category $\mathbb{C}$, by defining an automorphism $f : C \xrightarrow{\sim} C$ of an object $C \in \mathbb{C}$ to be *inner* just if f is in the image of $\theta_C : \mathcal{Z}_{\mathbb{C}}(C) \to \mathsf{Aut}(C)$. Less precisely, the automorphism $f : C \xrightarrow{\sim} C$ is inner if it can be coherently extended along any arrow out of C.

---

[2] Earlier versions of this result were also proven by Schupp [12] and Pettet [10].

[3] P. Freyd [2] studied a somewhat similar notion while modelling Reynolds' parametricity for parametric polymorphism. As a special case, his work leads to a *monoid* of natural endomorphisms of the projection functor, whereas in our case, we would obtain the subgroup of invertible elements in this monoid.

(For readers familiar with topos theory and/or earlier papers on the subject of isotropy groups, we point out that in [4, 3] we consider instead the *contravariant* isotropy groups $\mathsf{Aut}(\pi : \mathbb{C}/C \to \mathbb{C})$. Now if $\mathbb{T}$ is a suitable logical theory with classifying topos $\mathcal{B}(\mathbb{T})$, then (a restriction of) the contravariant isotropy group of $\mathcal{B}(\mathbb{T})$ coincides with the covariant isotropy group of the category $\mathsf{fp}\mathbb{T}\mathsf{mod}$ of finitely presented $\mathbb{T}$-models. Moreover, calculation of the latter group generally also yields a description of the covariant isotropy group of the larger category $\mathbb{T}\mathsf{mod}$ of *all* $\mathbb{T}$-models, which is our focus in the present paper.)

In [6], the case where $\mathbb{C}$ is the category of models of an equational theory is analysed. Among other things, a complete syntactic characterization of covariant isotropy for such a $\mathbb{C}$ is obtained, recovering not only Bergman's result for $\mathbb{C} = \mathsf{Grp}$ but also characterizing the definable automorphisms of other common algebraic structures such as monoids and rings. In applying the general characterization in specific instances, one typically needs to analyse the result of adjoining one or more indeterminates to a given model, and this in turn leads one to consider the *word problem* for such models.

The present paper, which is based on the PhD research [9] of the second author, is concerned with the analysis of the notion of isotropy or definable automorphism for (strict) monoidal categories and related structures. It hardly needs arguing that monoidal categories play various important roles in mathematics and theoretical computer science, both as objects of study in their own right, as models of logical theories, and as basic tools for studying other phenomena. However, we should point out here an observation by Richard Garner [5, Proposition 3] to the effect that both Cat and Grpd, the categories of small categories and small groupoids respectively, have *trivial* covariant isotropy, in the sense that for any category/groupoid $\mathbb{C}$ we have $\mathcal{Z}(\mathbb{C}) = 1$, the trivial group. The reason for this is roughly as follows: when considering an inner automorphism α of a category $\mathbb{C}$ in Cat, it must in particular extend to the categories obtained from $\mathbb{C}$ by freely adjoining a new object or arrow; but these latter categories are just obtained from $\mathbb{C}$ via disjoint union, which then (as Garner shows) easily entails that α can only be the identity on $\mathbb{C}$ (and an identical argument works for Grpd). As such, it is perhaps surprising that the category of strict monoidal categories has *non-trivial* isotropy. In fact, and this is the central result of the present paper, the isotropy group of a strict monoidal category is precisely its *Picard group* (its group of $\otimes$-invertible objects).

Since the theory of strict monoidal categories is not a purely equational theory, we cannot directly use results from [6]. Instead, we need to work in the setting of *quasi-equational theories*. These are multi-sorted theories in which the operations can be *partial*; equivalently, they are finite-limit theories. These include the theories of categories, groupoids, strict monoidal categories, symmetric/braided/balanced monoidal categories, and crossed modules. They also include what one might call *functor theories*, which are theories describing functors from a small category into a category of models. As a special case, one obtains theories whose categories of models are presheaf categories.[4] Our first main contribution of the paper is then a generalization of the syntactic characterization of isotropy from equational theories to this wider class of quasi-equational theories.

While we have indicated why the non-trivial isotropy of strict monoidal categories is perhaps surprising, there is also a sense in which it is to be expected. Indeed, since strict monoidal categories are monoids internal to Cat, we expect that the isotropy of strict monoidal

[4] Not to be confused with the so-called *theories of presheaf type*, which are theories whose classifying topos happens to be a presheaf topos.

categories is closely related to that of monoids. Since the isotropy of a monoid M is its subgroup of invertible elements, the conjecture that the isotropy of a strict monoidal category is its group of invertible objects is not unreasonable. However, it is not at all immediate that the isotropy of a strict monoidal category should be determined *completely* by its set of objects; the recognition that this *is* the case is the second main contribution of this paper.

A priori, one can try to establish this result in a variety of ways. First of all, it can be approached purely syntactically, by making careful analysis of the word problem for strict monoidal categories. However, several aspects of this analysis can also be cast in more conceptual terms, giving rise to a categorical way of deriving the isotropy of strict monoidal categories from that of monoids. We thus also include a more categorical viewpoint, which applies to several other theories of categorical structures, including crossed modules.

2 Quasi-equational theories

We begin by reviewing the relevant notions from categorical logic. For more details concerning quasi-equational theories and partial Horn logic, we refer to [8]. For a general treatment of categorical logic, see [11].

▶ **Definition 1** (Signatures, Terms, Horn Formulas, Horn Sequents, Quasi-Equational Theories)**.**

- A *signature* Σ is a pair of sets $\Sigma = (\Sigma_{\mathsf{Sort}}, \Sigma_{\mathsf{Fun}})$, where Σ_{Sort} is the set of *sorts* of Σ and Σ_{Fun} is the set of *function/operation symbols* of Σ. Each element $f \in \Sigma_{\mathsf{Fun}}$ comes equipped with a finite tuple of sorts $(A_1, \ldots, A_n, A)$, and we write $f : A_1 \times \ldots \times A_n \to A$.
- Given a signature Σ, we assume that we have a countably infinite set of variables of each sort A. Then one can recursively define the set $\mathsf{Term}(\Sigma)$ of *terms* of Σ in the usual way, so that each term will have a uniquely defined sort. We write $\mathsf{Term}^c(\Sigma)$ for the set of *closed* terms of Σ, i.e. terms containing no variables.
- Given a signature Σ, one can recursively define the set $\mathsf{Horn}(\Sigma)$ of *Horn formulas* of Σ in the usual way, where a Horn formula is a finite conjunction of equations between elements of $\mathsf{Term}(\Sigma)$. We write $\top$ for the empty conjunction.
- A *Horn sequent* over a signature Σ is an expression of the form $\varphi \vdash^{\vec{x}} \psi$, where $\varphi, \psi \in \mathsf{Horn}(\Sigma)$ and have variables among $\vec{x}$.
- A *quasi-equational theory* $\mathbb{T}$ over a signature Σ is a set of Horn sequents over Σ, which we call the *axioms* of $\mathbb{T}$.

One can set up a deduction system of *partial Horn logic* (PHL) for quasi-equational theories, axiomatizing the notion of a *provable sequent* $\varphi \vdash^{\vec{x}} \psi$. Accordingly, for a theory $\mathbb{T}$ we have the notion of a $\mathbb{T}$-provable sequent; moreover, if $\top \vdash^{\vec{x}} \varphi$ is $\mathbb{T}$-provable, then we simply say that $\mathbb{T}$ proves φ, and write $\mathbb{T} \vdash^{\vec{x}} \varphi$.

We refer the reader to [8, Definition 1] for the logical axioms and inference rules of PHL. The distinguishing feature of this deduction system is that equality of terms is *not* assumed to be reflexive, i.e. if $t(\vec{x})$ is a term over a given signature, then $\top \vdash^{\vec{x}} t(\vec{x}) = t(\vec{x})$ is *not* a logical axiom of partial Horn logic, unless t is a variable. In other words, if we abbreviate the equation $t = t$ by $t\downarrow$ (read: *t is defined*), then unless t is a variable, the sequent $\top \vdash^{\vec{x}} t\downarrow$ is *not* a logical axiom of PHL. Furthermore, the logical inference rule of term substitution is then only formulated for *defined* terms.

▶ **Example 2.** We have the following examples of quasi-equational theories:

- Every single-sorted algebraic theory is a quasi-equational theory; this includes the usual algebraic theories of (commutative) monoids, (abelian) groups, (commutative) unital rings, etc.
- The theories of (small) categories, groupoids, categories with a (chosen) terminal object, categories with (chosen) finite products, categories with (chosen) finite limits, locally cartesian closed categories, and elementary toposes, can all be axiomatized as quasi-equational theories over a two-sorted signature (with one sort O for objects and one sort A for arrows). For details see [8, Example 4 and Section 6]. The theory of (small) strict monoidal categories can also be axiomatized as a quasi-equational theory (see Section 4 below).
- If $\mathbb{T}$ is any quasi-equational theory and $\mathcal{J}$ is any small category, then one can axiomatize the functor category $\mathbb{T}\mathsf{mod}^{\mathcal{J}}$ by a quasi-equational theory $\mathbb{T}^{\mathcal{J}}$; see [9, Chapter 5].

In the remainder of the paper, by *theory* we shall mean *quasi-equational theory*, unless explicitly stated otherwise.

We now review the set-theoretic semantics of PHL. This follows the standard pattern of algebraic theories, with the key difference being that function symbols are now only interpreted as *partial* functions. We write $f : A \rightharpoonup B$ for a partial function from A to B, which is by definition a *total* function $f : \mathsf{dom}(f) \to B$ for some subset $\mathsf{dom}(f) \subseteq A$. If Σ is a signature, then a Σ-*structure* M is a family of sets M_C indexed by the sorts C of Σ, together with interpretations of the function symbols $f : A_1 \times \cdots \times A_k \to A$ as partial functions $f^M : M_{A_1} \times \cdots \times M_{A_k} \rightharpoonup M_A$. By induction on the structure of a term t in variable context $x_1 : A_1, \ldots, x_k : A_k$, we obtain its interpretation as a partial function $t^M : M_{A_1} \times \cdots \times M_{A_k} \rightharpoonup M_A$ in a Σ-structure M, while a Horn formula $\varphi(x_1, \ldots, x_k)$ is interpreted as a subset $\varphi(x_1, \ldots, x_k)^M \subseteq M_{A_1} \times \ldots \times M_{A_k}$.

A Σ-structure M *satisfies* a Horn sequent $\varphi \vdash^{\vec{x}} \psi$ if $\varphi(x_1, \ldots, x_k)^M \subseteq \psi(x_1, \ldots, x_k)^M$. When $\mathbb{T}$ is a theory, then a Σ-structure M is a $\mathbb{T}$-*model* when it satisfies all the $\mathbb{T}$-axioms, and hence all the $\mathbb{T}$-provable sequents (by soundness of partial Horn logic).

▶ **Definition 3.** Let Σ be a signature and M, N Σ-structures. A *homomorphism* $h : M \to N$ is a family of total functions $h = (h_A : M_A \to N_A)_{A:\mathsf{Sort}}$ with the property that if $f : A_1 \times \ldots \times A_n \to A$ is any function symbol of Σ and $(a_1, \ldots, a_n) \in \mathsf{dom}\left(f^M\right)$, then $(h_{A_1}(a_1), \ldots, h_{A_n}(a_n)) \in \mathsf{dom}\left(f^N\right)$ and $h_A\left(f^M(a_1, \ldots, a_n)\right) = f^N(h_{A_1}(a_1), \ldots, h_{A_n}(a_n))$. The homomorphism h *reflects definedness* if moreover $(h_{A_1}(a_1), \ldots, h_{A_n}(a_n)) \in \mathsf{dom}\left(f^N\right)$ always implies $(a_1, \ldots, a_n) \in \mathsf{dom}\left(f^M\right)$.

Let us emphasize that the sort components $h_A : M_A \to N_A$ of a homomorphism $h : M \to N$ are *total* functions, rather than partial functions. One could theoretically choose to work with other notions of homomorphism, but for our purposes we have chosen to use the total homomorphisms. When working with homomorphisms we often suppress the sort subscripts. The $\mathbb{T}$-models and their homomorphisms then form a category $\mathbb{T}\mathsf{mod}$, which is complete and cocomplete.

▶ **Definition 4.** A *morphism of theories* $\rho : \mathbb{T} \to \mathbb{S}$ consists of a mapping $A \mapsto \rho(A)$ from the sorts of $\mathbb{T}$ to the sorts of $\mathbb{S}$ and a mapping $f \mapsto \rho(f)$ from the function symbols of $\mathbb{T}$ to the terms of $\mathbb{S}$ that preserves both typing and provability.

When $\rho : \mathbb{T} \to \mathbb{S}$ is a morphism of theories, we have an induced functor $\rho^* : \mathbb{S}\mathsf{mod} \to \mathbb{T}\mathsf{mod}$ by [8, Proposition 28]. This functor ρ^* sends an $\mathbb{S}$-model M to the $\mathbb{T}$-model $\rho^* M$ with $(\rho^* M)_A := M_{\rho(A)}$ for each sort A of $\mathbb{T}$ and $f^{\rho^* M} := \rho(f)^M$ for each function symbol f of $\mathbb{T}$.

In particular, for every sort A of $\mathbb{T}$ there is a forgetful functor $U_A : \mathbb{T}\mathsf{mod} \to \mathsf{Set}$ sending a model M to the carrier set M_A (induced by the theory morphism from the single-sorted empty theory to $\mathbb{T}$ that sends the unique sort of the former theory to the sort A). Each such functor also has a left adjoint F_A (see e.g. [8, Theorem 29]), giving for a set X the free $\mathbb{T}$-model $F_A(X)$ generated by X: $F_A \dashv U_A : \mathsf{Set} \rightleftarrows \mathbb{T}\mathsf{mod}$.

▶ **Definition 5.** For a $\mathbb{T}$-model M, we can form the extension $\mathbb{T}(M)$, the *diagram theory of* M, adapted from ordinary model theory [13]. It is the extension of $\mathbb{T}$ by

- A constant $\overline{a} : A$ and an axiom $\top \vdash \overline{a} \downarrow$ for every element $a \in M_A$ (for every sort A).
- An axiom $\top \vdash \overline{f(a_1, \ldots, a_k)} = f(\overline{a_1}, \ldots, \overline{a_k})$ for every function symbol $f : A_1 \times \cdots \times A_k \to A$ and tuple $(a_1, \ldots, a_k) \in \mathsf{dom}\left(f^M\right)$.

For better readability, we will generally omit the bar notation on constants of M. Clearly M is a model of $\mathbb{T}(M)$, and in fact it is the *initial* model: $\mathbb{T}(M)\mathsf{mod} \simeq M/\mathbb{T}\mathsf{mod}$ (see [9, Lemma 2.2.4] for a proof). The obvious theory morphism $\mathbb{T} \to \mathbb{T}(M)$ corresponds to the forgetful functor $M/\mathbb{T}\mathsf{mod} \to \mathbb{T}\mathsf{mod}$.

One of the central constructions in the present paper is that of *adjoining an indeterminate* to a model. Given a $\mathbb{T}$-model M and a sort A of $\mathbb{T}$, we form a new model $M\langle \mathsf{x}_A \rangle$ which is the result of freely adjoining a new element x_A of sort A to M. Formally, one can define $M\langle \mathsf{x}_A \rangle$ as $M + F_A(1)$, where $F_A(1)$ is the free $\mathbb{T}$-model on one generator of sort A. Consequently, homomorphisms $M\langle \mathsf{x}_A \rangle \to N$ are in natural bijective correspondence with pairs (h, n) consisting of a homomorphism $h : M \to N$ and an element $n \in N_A$. We will write $\mathbb{T}(M, \mathsf{x}_A)$ for the theory extending the diagram theory $\mathbb{T}(M)$ by a new constant $\mathsf{x}_A : A$ and a new axiom $\top \vdash \mathsf{x}_A \downarrow$. One can then equivalently define the $\mathbb{T}$-model $M\langle \mathsf{x}_A \rangle$ as the initial model of $\mathbb{T}(M, \mathsf{x}_A)$. For a sequence of (not necessarily distinct) sorts $A_1, \ldots, A_k$, we will also write $\mathbb{T}(M, \mathsf{x}_1, \ldots, \mathsf{x}_k)$ for the theory extending $\mathbb{T}(M)$ by new, pairwise distinct constants $\mathsf{x}_i : A_i$ and axioms $\top \vdash \mathsf{x}_i \downarrow$ for each $1 \le i \le k$.

Finally, we note that for a $\mathbb{T}$-model M, an indeterminate x_A of sort A, and an arbitrary sort B, we have

$$M\langle \mathsf{x}_A \rangle_B = \{t \in \mathsf{Term}^c\left(\mathbb{T}(M), \mathsf{x}_A\right) \mid t : B \text{ and } \mathbb{T}(M, \mathsf{x}_A) \vdash t \downarrow\} /{=}, \tag{3}$$

i.e. the carrier set $M\langle \mathsf{x}_A \rangle_B$ of the $\mathbb{T}$-model $M\langle \mathsf{x}_A \rangle$ at the sort B is the quotient of the set of provably defined closed terms of sort B, possibly containing x_A and constants from M, modulo the partial congruence relation of $\mathbb{T}(M, \mathsf{x}_A)$-provable equality. For more details, see [9, Remark 2.2.7].

3 Isotropy

We now embark on the syntactic description of the covariant isotropy group of a theory. First, let us briefly review the simpler situation of a single-sorted equational theory $\mathbb{T}$. That is, we describe the isotropy group of a $\mathbb{T}$-model M (details are in [6]). The elements of the model $M\langle \mathsf{x} \rangle$ (for x an indeterminate) can be described explicitly as congruence classes of terms $t(\mathsf{x})$, built from the indeterminate x, constants from M, and the operation symbols of $\mathbb{T}$. Two such terms are congruent if they are $\mathbb{T}(M, \mathsf{x})$-provably equal. For example, if $\mathbb{T}$ is the theory of monoids and M is a monoid with $m_1, m_2, m_3 \in M$, unit e, and $m_1 m_2 = m_3$, then the terms $t = \mathsf{x} m_1 \mathsf{x} m_1 m_2 \mathsf{x}$ and $\mathsf{x} e m_1 \mathsf{x} e m_3 \mathsf{x}$ are congruent.

For a set-theoretic $\mathbb{T}$-model M, each congruence class $[t] \in M\langle \mathsf{x} \rangle$ can be interpreted as a function $t^M : M \to M$, via substitution into the indeterminate x. We thus have a mapping

$$M\langle \mathsf{x} \rangle \to [M, M] \,; \qquad [t] \mapsto t^M$$

where $[M, M]$ is the set of functions from M to itself (well-definedness follows from soundness of the set-theoretic semantics of equational logic). Moreover, this mapping is a homomorphism of monoids, where the monoid structure on $M\langle \mathsf{x} \rangle$ is given by substitution: $[t] \cdot [s] := [t[s/\mathsf{x}]]$, the unit being $[\mathsf{x}]$. We then restrict on both sides to the invertible elements, obtaining a group homomorphism $\mathsf{Inv}(M\langle \mathsf{x} \rangle) \to \mathsf{Perm}(M)$ from the group of substitutionally invertible (congruence classes of) terms to the permutation group of the set M. However, we do not wish to just consider arbitrary permutations of the set M, but rather *automorphisms* of the $\mathbb{T}$-model M; in fact, we want to consider *inner* automorphisms, i.e. automorphisms that extend naturally along any homomorphism $M \to N$. On the level of terms $[t] \in M\langle \mathsf{x} \rangle$, this is achieved by the following definition: $[t]$ is said to *commute generically with* a function symbol $f : A^n \to A$ (A being the unique sort of $\mathbb{T}$) if

$$\mathbb{T}(M, \mathsf{x}_1, \ldots, \mathsf{x}_n) \vdash \; t[f(\mathsf{x}_1, \ldots, \mathsf{x}_n)/\mathsf{x}] = f(t[\mathsf{x}_1/\mathsf{x}], \ldots, t[\mathsf{x}_n/\mathsf{x}]).$$

We then form the subgroup $\mathsf{DefInn}(M)$ of $\mathsf{Inv}(M\langle \mathsf{x} \rangle)$ on those $[t]$ that commute generically with all function symbols of the theory. This ensures that such a $[t]$ induces an *automorphism* of the $\mathbb{T}$-model M and not merely a permutation of its underlying set, thus yielding a mapping $(-)^M : \mathsf{DefInn}(M) \to \mathsf{Aut}(M)$. However, it turns out that such an automorphism induced by an element of $\mathsf{DefInn}(M)$ is also *inner*. Indeed, given $h : M \to N$, we obtain a homomorphism $h\langle \mathsf{x} \rangle : M\langle \mathsf{x} \rangle \to N\langle \mathsf{x} \rangle$ of the substitution monoids, which restricts to a group homomorphism $\mathsf{DefInn}(M) \to \mathsf{DefInn}(N)$. It can then be shown that the subgroup $\mathsf{DefInn}(M)$ is isomorphic to the covariant isotropy group of M, where $\theta_M : \mathcal{Z}(M) \to \mathsf{Aut}(M)$ is the comparison homomorphism (2):

$$\begin{tikzcd}
 & \mathsf{DefInn}(M) \arrow[r, "\subseteq"'] \arrow[d, "(-)^M"] & \mathsf{Inv}(M\langle \mathsf{x} \rangle) \arrow[d, "(-)^M"] \\
\mathcal{Z}(M) \arrow[ur, dashed, "\cong"] \arrow[r, "\theta_M"'] & \mathsf{Aut}(M) \arrow[r, "\subseteq"'] & \mathsf{Perm}(M)
\end{tikzcd}$$

We now explain how to extend this result to a (multi-sorted) *quasi-equational theory* $\mathbb{T}$. The main technical difficulties in this extension involve accommodating multi-sortedness and the possibility of certain terms not being provably defined. To handle multi-sortedness, we need to consider, for a $\mathbb{T}$-model M, the model $M\langle \mathsf{x}_A \rangle$ obtained by adjoining an indeterminate x_A of sort A for any sort A of $\mathbb{T}$. Since substitution corresponds to composition under the interpretation mapping $t \mapsto t^M$, it follows that $M\langle \mathsf{x}_A \rangle_A$ carries a monoid structure (recall (3) for the definition of this set), defined as before in terms of substitution into the indeterminate x_A. We now write

$$M\langle \bar{\mathsf{x}} \rangle := \prod_{A:\mathsf{Sort}} M\langle \mathsf{x}_A \rangle_A$$

for the sort-indexed product monoid of these substitution monoids. An element of $M\langle \bar{\mathsf{x}} \rangle$ is therefore a sort-indexed family of congruence classes of terms $[s_A]_A$, where $s_A \in \mathsf{Term}^c(\mathbb{T}(M), \mathsf{x}_A)$ is of sort A and $\mathbb{T}(M, \mathsf{x}_A) \vdash s_A \downarrow$. Given such a tuple $[s_A]_A$, its interpretation gives us, at each sort A, a *total* function $s_A^M : M_A \to M_A$ (because s_A is provably defined in $\mathbb{T}(M, \mathsf{x}_A)$), defined via substitution into the indeterminate x_A (cf. [9, Remark 2.2.12]). The central definitions towards characterizing those $[s_A]_A \in M\langle \bar{\mathsf{x}} \rangle$ that induce elements of isotropy for M are then as follows:

▶ **Definition 6.** Let M be a $\mathbb{T}$-model and $[s_C]_C \in M\langle \bar{\mathsf{x}} \rangle$.

- If $f : A_1 \times \ldots \times A_n \to A$ is a function symbol of Σ, then we say that $([s_C])_C$ *commutes generically with* f if the Horn sequent

  $$f(\mathsf{x}_1, \ldots, \mathsf{x}_n) \downarrow \vdash \; s_A[f(\mathsf{x}_1, \ldots, \mathsf{x}_n)/\mathsf{x}_A] = f\left(s_{A_1}[\mathsf{x}_1/\mathsf{x}_{A_1}], \ldots, s_{A_n}[\mathsf{x}_n/\mathsf{x}_{A_n}]\right)$$

  is provable in $\mathbb{T}(M, \mathsf{x}_1, \ldots, \mathsf{x}_n)$.

- We say that $([s_C])_C$ is *invertible* if for each sort A there is some $\left[s_A^{-1}\right] \in M\langle \mathsf{x}_A \rangle_A$ with
$$\mathbb{T}(M, \mathsf{x}_A) \vdash s_A \left[s_A^{-1}/\mathsf{x}_A\right] = \mathsf{x}_A = s_A^{-1}[s_A/\mathsf{x}_A].$$
- We say that $([s_C])_C$ *reflects definedness* if for every function symbol $f : A_1 \times \ldots \times A_n \to A$ in Σ with $n \geq 1$, the sequent
$$f\left(s_{A_1}[\mathsf{x}_1/\mathsf{x}_{A_1}], \ldots, s_{A_n}[\mathsf{x}_n/\mathsf{x}_{A_n}]\right) \downarrow \;\vdash f(\mathsf{x}_1, \ldots, \mathsf{x}_n) \downarrow$$
is provable in $\mathbb{T}(M, \mathsf{x}_1, \ldots, \mathsf{x}_n)$.

The condition that $[s_C]_C$ commutes generically with the function symbols of $\mathbb{T}$ then ensures that $[s_C]_C$ induces not just an endofunction of each carrier set M_C but in fact an endo*morphism* of the $\mathbb{T}$-model M. Invertibility of $[s_C]_C$ then ensures that these endomorphisms are bijective. However, due to the fact that function symbols are interpreted as partial maps, a (sortwise) bijective homomorphism is not in general an isomorphism in $\mathbb{T}\mathsf{mod}$: a bijective homomorphism is an isomorphism precisely when it also reflects definedness (cf. [9, Lemma 2.2.33]). Thus, the third condition ensures that the inverses $\left[s_A^{-1}\right]$ also induce endomorphisms.

Let us write $\mathsf{DefInn}(M)$ again for the subgroup of the product monoid $M\langle \bar{\mathsf{x}} \rangle$ consisting of those elements satisfying the three conditions above. We then have the following characterization, of which detailed proofs can be found in [9, Theorems 2.2.41, 2.2.53]:

▶ **Theorem 7.** *Let $\mathbb{T}$ be a quasi-equational theory. Then for any $M \in \mathbb{T}\mathsf{mod}$ we have*
$$\mathcal{Z}(M) \cong \mathsf{DefInn}(M) = \left\{ [s_C]_C \in M\langle \bar{\mathsf{x}} \rangle \;\middle|\; [s_C]_C \begin{array}{l} \textit{is invertible, commutes generically with} \\ \textit{all operations, and reflects definedness.} \end{array} \right\}.$$

4 Monoidal categories and the Picard group

With this description of the isotropy group of an arbitrary quasi-equational theory, we now turn to the specific example of strict monoidal categories. We can axiomatize these using the following signature Σ (where the first two ingredients comprise the signature for *categories*):

- two sorts O and A (for objects and arrows);
- function symbols $\mathsf{dom}, \mathsf{cod} : A \to O$, $\mathsf{id} : O \to A$, and $\circ : A \times A \to A$;
- function symbols $\otimes_O : O \times O \to O$, $\otimes_A : A \times A \to A$;
- constant symbols $I_O : O$ and $I_A : A$.

Whenever reasonable, we omit the subscripts on $\otimes$ and I. As axioms, we take those for categories and add (omitting the hypothesis $\top$):

- $x \otimes y \downarrow, \qquad I \downarrow,$
- $x \otimes (y \otimes z) = (x \otimes y) \otimes z, \qquad x \otimes I = x = I \otimes x,$
- $\mathsf{dom}(f \otimes g) = \mathsf{dom}(f) \otimes \mathsf{dom}(g), \qquad \mathsf{cod}(f \otimes g) = \mathsf{cod}(f) \otimes \mathsf{cod}(g),$
- $f \circ h \downarrow \wedge\, g \circ k \downarrow \vdash (f \otimes g) \circ (h \otimes k) = (f \circ h) \otimes (g \circ k),$
- $\mathsf{id}(x \otimes y) = \mathsf{id}(x) \otimes \mathsf{id}(y), \qquad \mathsf{id}(I_O) = I_A.$

Note that in this fragment of logic, we need to include axioms forcing the tensor products and unit object to be *total* operations. Because of strict associativity, we may omit brackets when dealing with nested expressions involving tensor products. We shall henceforth denote this theory by $\mathbb{T}$, and write $\mathsf{StrMonCat}$ for its category of models, whose objects are small strict monoidal categories and whose morphisms are strict monoidal functors. Our goal is now to prove the following:

▶ **Theorem 8.** *The covariant isotropy group $\mathcal{Z}$: $\mathsf{StrMonCat} \to \mathsf{Grp}$ is naturally isomorphic to the functor $\mathsf{Pic} : \mathsf{StrMonCat} \to \mathsf{Grp}$ that sends a strict monoidal category $\mathbb{C}$ to its Picard group $\mathsf{Pic}(\mathbb{C})$, i.e. the group of $\otimes$-invertible elements in the monoid of objects of $\mathbb{C}$.*

Because a strict monoidal category is a monoid object in Cat, we have two functors

$$\mathrm{Ob}, \mathrm{Arr} : \mathsf{Cat}(\mathsf{Mon}) = \mathsf{StrMonCat} \rightrightarrows \mathsf{Mon}.$$

We shall ultimately prove that the diagram

$$\begin{tikzcd} \mathsf{StrMonCat} \arrow[rr, "\mathrm{Ob}"] \arrow[dr, "\mathcal{Z}"'] & & \mathsf{Mon} \arrow[dl, "\mathcal{Z}_{\mathsf{Mon}}"] \\ & \mathsf{Grp} & \end{tikzcd} \tag{4}$$

commutes up to natural isomorphism, showing that the covariant isotropy group of $\mathsf{StrMonCat}$ is completely determined by the covariant isotropy group of Mon. Since we have proved in [6, Example 4.3] that the latter sends a monoid M to its subgroup of invertible elements, Theorem 8 then follows.[5]

4.1 Monoidal categories and indeterminates

In this section we analyse the process of adjoining an indeterminate to a strict monoidal category. Let us first describe explicitly the result of adjoining an indeterminate to a *monoid*.

▶ **Definition 9.** Let M be a monoid, and X a set of symbols disjoint from M.

- A *word* over $M\langle X\rangle$ is formal string of symbols from the alphabet $M \cup X$.
- A word w is in *(expanded) normal form* when it has the form $w \equiv m_0x_0m_1x_1 \cdots x_{n-1}m_n$ for $m_i \in M$ and $x_j \in X$. In other words, w is in expanded normal form if it contains no two consecutive elements of M, and if every occurrence of some $x \in X$ in w is flanked on both sides by an element of M.

We then have (by taking an arbitrary word, multiplying adjacent elements from M and inserting the unit of M wherever necessary):

▶ **Lemma 10.** *When $M = (M, \cdot, e)$ is a monoid, every element w of the monoid $M\langle \mathsf{x}\rangle$ has a canonical representative $w = m_0\mathsf{x}m_1\mathsf{x}\cdots\mathsf{x}m_n$ in expanded normal form.*

Moreover, the unit of $M\langle \mathsf{x}\rangle$ is represented as the word e and multiplication is given by $(m_0\mathsf{x}m_1\mathsf{x}\cdots\mathsf{x}m_j) \cdot (m'_0\mathsf{x}m'_1\mathsf{x}\cdots\mathsf{x}m'_k) = m_0\mathsf{x}m_1\mathsf{x}\cdots\mathsf{x}(m_j \cdot m'_0)\mathsf{x}m'_1\cdots\mathsf{x}m'_k$.

We now turn to the process of adjoining an indeterminate *object* x_O, i.e. an indeterminate of sort O, to a strict monoidal category $\mathbb{C}$. In order to determine the objects of $\mathbb{C}\langle \mathsf{x}_O\rangle$, we note that the functor $\mathrm{Ob} : \mathsf{StrMonCat} \to \mathsf{Mon}$ has both adjoints:

$$\begin{tikzcd} \mathsf{StrMonCat} \arrow[r, "\mathrm{Ob}" description] & \mathsf{Mon} \arrow[l, bend right, "\Delta"', "\perp"] \arrow[l, bend left, "\nabla", "\perp"'] \end{tikzcd}$$

Here Δ sends a monoid M to the discrete strict monoidal category on M and ∇ sends M to the indiscrete strict monoidal category on M. In fact, if $\mathcal{E}$ is *any* category with finite limits,

[5] For a general functor $F : \mathcal{E} \to \mathcal{F}$ it is *not* the case that $\mathcal{Z}_\mathcal{E} \cong \mathcal{Z}_F \circ F$. In fact, in [3] it is explained that in general the relationship between $\mathcal{Z}_\mathcal{E}$ and $\mathcal{Z}_\mathcal{F} \circ F$ takes the form of a *span*. The commutativity of (4) may thus be expressed by saying that both legs of the span associated with Ob are isomorphisms.

then the forgetful functor $\mathrm{Ob} : \mathsf{Cat}(\mathcal{E}) \to \mathcal{E}$ has both adjoints (the proof is a completely straightforward analogue of the argument for $\mathcal{E} = \mathsf{Set}$). As such, $\mathrm{Ob} : \mathsf{StrMonCat} \to \mathsf{Mon}$ preserves all limits and colimits. Now by definition $\mathbb{C}\langle \mathsf{x}_O \rangle \cong \mathbb{C} + F\mathbf{1}$, where $F\mathbf{1}$ is the free strict monoidal category on a single object; moreover, the latter is easily seen to be isomorphic to $\Delta(F1)$, the discrete strict monoidal category on the free monoid $F1$ on one generator. We thus have

$$\mathrm{Ob}(\mathbb{C}\langle \mathsf{x}_O \rangle) \cong \mathrm{Ob}(\mathbb{C} + F\mathbf{1}) \cong \mathrm{Ob}(\mathbb{C}) + \mathrm{Ob}(F\mathbf{1}) = \mathrm{Ob}(\mathbb{C}) + F1 \cong \mathrm{Ob}(\mathbb{C})\langle \mathsf{x} \rangle.$$

This shows that the object forgetful functor preserves the process of adjoining an indeterminate of sort O.[6]

We now describe the monoid of arrows of $\mathbb{C}\langle \mathsf{x}_O \rangle$. It is not true that $\mathrm{Arr} : \mathsf{StrMonCat} \to \mathsf{Mon}$ preserves arbitrary binary coproducts, but it *does* preserve the specific binary coproduct $\mathbb{C} + F\mathbf{1}$:

▶ **Lemma 11.** *If* $\mathbb{C} \in \mathsf{StrMonCat}$*, then* $\mathrm{Arr}(\mathbb{C}\langle \mathsf{x}_O \rangle) \cong \mathrm{Arr}(\mathbb{C})\langle \mathsf{x} \rangle$*.*

Proof. We sketch a syntactic proof, noting that the result can also be deduced categorically from the fact that the endofunctor $- + F1 : \mathsf{Mon} \to \mathsf{Mon}$ preserves pullbacks.

An element of $\mathrm{Arr}(\mathbb{C}\langle \mathsf{x}_O \rangle)$ is a congruence class of terms t built up from the operations of $\mathbb{T}$, arrows of $\mathbb{C}$, and the term $\mathsf{id}(\mathsf{x}_O)$. One can show by induction that every such term t is congruent to one of the form $t = f_1 \otimes \mathsf{id}(\mathsf{x}_O) \otimes f_2 \otimes \mathsf{id}(\mathsf{x}_O) \otimes \cdots \otimes \mathsf{id}(\mathsf{x}_O) \otimes f_n$ where each f_i is an arrow of $\mathbb{C}$. Thus, the monoid $\mathrm{Arr}(\mathbb{C}\langle \mathsf{x}_O \rangle)$ is isomorphic, by Lemma 10, to $\mathrm{Arr}(\mathbb{C})\langle \mathsf{x} \rangle$. ◀

In fact, we may describe the relationship between the functor $(-) + F\mathbf{1}$ adjoining an indeterminate object to a strict monoidal category and the functor $(-) + F1$ adjoining an indeterminate element to a monoid as follows.

▶ **Proposition 12.** *The functor* $(-) + F\mathbf{1} : \mathsf{Cat}(\mathsf{Mon}) \to \mathsf{Cat}(\mathsf{Mon})$ *is naturally isomorphic to* $\mathsf{Cat}(- + F1)$*.*

We thus obtain the following explicit description of the strict monoidal category $\mathbb{C}\langle \mathsf{x}_O \rangle$:

Objects: Words $a_1 \mathsf{x} a_2 \mathsf{x} \cdots \mathsf{x} a_n$ where each a_i is an object of $\mathbb{C}$.
Morphisms: Words $f_1 \mathsf{x} f_2 \mathsf{x} \cdots \mathsf{x} f_n$ where each f_i is an arrow of $\mathbb{C}$.
Domain: $\mathsf{dom}(f_1 \mathsf{x} \cdots \mathsf{x} f_n) = \mathsf{dom}(f_1) \mathsf{x} \cdots \mathsf{x} \mathsf{dom}(f_n)$.
Codomain: $\mathsf{cod}(f_1 \mathsf{x} \cdots \mathsf{x} f_n) = \mathsf{cod}(f_1) \mathsf{x} \cdots \mathsf{x} \mathsf{cod}(f_n)$.
Identities: $\mathsf{id}(a_1 \mathsf{x} \cdots \mathsf{x} a_n) = \mathsf{id}(a_1) \mathsf{x} \cdots \mathsf{x} \mathsf{id}(a_n)$.
Composition: $(f_1 \mathsf{x} \cdots \mathsf{x} f_n) \circ (g_1 \mathsf{x} \cdots \mathsf{x} g_n) = f_1 g_1 \mathsf{x} \cdots \mathsf{x} f_n g_n$.
Tensors: $(a_1 \mathsf{x} \cdots \mathsf{x} a_n) \otimes (b_1 \mathsf{x} \cdots \mathsf{x} b_m) = a_1 \mathsf{x} \cdots \mathsf{x} (a_n \otimes b_1) \mathsf{x} \cdots \mathsf{x} b_m$.
Tensor units: I_O, I_A (tensor units of $\mathbb{C}$ regarded as one-letter words).

Next, we address the issue of adjoining an indeterminate *arrow* x_A to $\mathbb{C}$. Here we cannot invoke a simple categorical fact about coproducts, because $\mathrm{Arr} : \mathsf{StrMonCat} \to \mathsf{Mon}$ does not preserve coproducts of the relevant kind (which, to be explicit, is coproducts with the free strict monoidal category $F\mathbf{2}$, where $\mathbf{2}$ is the free-living arrow). We are thus forced to carry out a direct *syntactic* analysis of the objects and arrows of $\mathbb{C}\langle \mathsf{x}_A \rangle$. Note that these are generated, under the operations of domain, codomain, identities, composition, and tensor

[6] Note that for a functor $\rho^* : \mathbb{S}\mathsf{mod} \to \mathbb{T}\mathsf{mod}$ induced by a theory morphism $\rho : \mathbb{T} \to \mathbb{S}$ it is *not* in general the case that $\rho^*(M\langle \mathsf{x} \rangle) \cong (\rho^* M)\langle \mathsf{x} \rangle$.

product, from the objects and arrows of $\mathbb{C}$, together with the new arrow x_A. In particular, there will be two new objects $\mathsf{dom}(\mathsf{x}_A)$ and $\mathsf{cod}(\mathsf{x}_A)$, and corresponding identity arrows $\mathsf{id}(\mathsf{dom}(\mathsf{x}_A))$, $\mathsf{id}(\mathsf{cod}(\mathsf{x}_A))$.

▶ **Definition 13.** Let $\mathbb{C} \in \mathsf{StrMonCat}$. A closed term $t \in \mathsf{Term}^c(\mathbb{C}, \mathsf{x}_A)$ of sort O is in *normal form* when it is of the form $t = a_1 \otimes \mathsf{x}_1 \otimes \cdots \otimes \mathsf{x}_{k-1} \otimes a_k$, where each a_i is an object of $\mathbb{C}$ and each $\mathsf{x}_i \in \{\mathsf{dom}(\mathsf{x}_A), \mathsf{cod}(\mathsf{x}_A)\}$. A closed term $t \in \mathsf{Term}^c(\mathbb{C}, \mathsf{x}_A)$ of sort A is in *normal form* when it is of the form $t = f_1 \otimes \mathsf{x}_1 \otimes \cdots \otimes \mathsf{x}_{k-1} \otimes f_k$, where each f_i is an arrow of $\mathbb{C}$ and each $\mathsf{x}_i \in \{\mathsf{x}_A, \mathsf{id}(\mathsf{dom}(\mathsf{x}_A)), \mathsf{id}(\mathsf{cod}(\mathsf{x}_A))\}$.

We may now describe $\mathbb{C}\langle \mathsf{x}_A \rangle$ in terms of normal forms. It is straightforward to prove, by directly verifying the universal property, that the category described below is indeed isomorphic to $\mathbb{C}\langle \mathsf{x}_A \rangle$. Alternatively, one can endow the set $\{t \in \mathsf{Term}^c(\mathbb{C}, \mathsf{x}_A) \mid \mathbb{T}(\mathbb{C}, \mathsf{x}_A) \vdash t\downarrow\}$ with a rewriting system and show that each term has a unique normal form.

Objects: closed terms of sort O in normal form.
Arrows: closed terms of sort A in normal form.
Domain: $\mathsf{dom}(f_1 \otimes \mathsf{x}_1 \otimes \cdots \otimes \mathsf{x}_{k-1} \otimes f_k) = \mathsf{dom}(f_1) \otimes \mathsf{y}_1 \otimes \cdots \otimes \mathsf{y}_{k-1} \otimes \mathsf{dom}(f_k)$ where $\mathsf{y}_i = \mathsf{dom}(\mathsf{x}_A)$ when $\mathsf{x}_i = \mathsf{x}_A$ or $\mathsf{x}_i = \mathsf{id}(\mathsf{dom}(\mathsf{x}_A))$, and $\mathsf{y}_i = \mathsf{cod}(\mathsf{x}_A)$ otherwise.
Codomain: $\mathsf{cod}(f_1 \otimes \mathsf{x}_1 \otimes \cdots \otimes \mathsf{x}_{k-1} \otimes f_k) = \mathsf{cod}(f_1) \otimes \mathsf{y}_1 \otimes \cdots \otimes \mathsf{y}_{k-1} \otimes \mathsf{cod}(f_k)$ where $\mathsf{y}_i = \mathsf{cod}(\mathsf{x}_A)$ when $\mathsf{x}_i = \mathsf{x}_A$ or $\mathsf{x}_i = \mathsf{id}(\mathsf{cod}(\mathsf{x}_A))$, and $\mathsf{y}_i = \mathsf{dom}(\mathsf{x}_A)$ otherwise.
Identities: $\mathsf{id}(a_1 \otimes \mathsf{x}_1 \otimes \cdots \otimes \mathsf{x}_{k-1} \otimes a_k) = \mathsf{id}(a_1) \otimes \mathsf{id}(\mathsf{x}_1) \otimes \cdots \otimes \mathsf{id}(\mathsf{x}_{k-1}) \otimes \mathsf{id}(a_k)$.
Composition: For $t = f_1 \otimes \mathsf{x}_1 \otimes \cdots \otimes \mathsf{x}_{k-1} \otimes f_k$ and $s = g_1 \otimes \mathsf{x}'_1 \otimes \cdots \otimes \mathsf{x}'_{k-1} \otimes g_k$ with $\mathsf{cod}(t) = \mathsf{dom}(s)$, define $s \circ t = (g_1 f_1) \otimes \mathsf{z}_1 \otimes \cdots \otimes \cdots \otimes \mathsf{z}_{k-1} \otimes (g_k f_k)$, where z_i is defined from x_i and x'_i in the evident way.
Tensors: $(a_1 \otimes \mathsf{x}_1 \otimes \cdots \otimes \mathsf{x}_{n-1} \otimes a_n) \otimes (b_1 \otimes \mathsf{y}_1 \otimes \cdots \otimes \mathsf{y}_{m-1} \otimes b_m) = a_1 \otimes \mathsf{x}_1 \otimes \cdots \otimes \mathsf{x}_{n-1} \otimes (a_n \otimes b_1) \otimes \mathsf{y}_1 \otimes \cdots \otimes \mathsf{y}_{m-1} \otimes b_m$.
Tensor units: I_O, I_A (tensor units of $\mathbb{C}$ regarded as one-letter words).

4.2 Isotropy group

We are now in a position to analyse the isotropy group of a strict monoidal category. By the results of the previous section, we know that an element of isotropy of a strict monoidal category $\mathbb{C}$ may be taken to be of the form (s_O, s_A), where s_O and s_A are closed terms in normal form of sort O and A respectively.

The first observation is that elements of isotropy of the monoid $\mathrm{Ob}(\mathbb{C})$ induce elements of isotropy of $\mathbb{C}$ (as we shall see in the next section, this is not specific to strict monoidal categories.) In what follows, we write $\mathcal{Z}(\mathbb{C})$ for the isotropy group of a strict monoidal category $\mathbb{C}$, and $\mathcal{Z}_{\mathsf{Mon}}(M)$ for the isotropy group of a monoid M (which is the group of invertible elements of M by [6, Example 4.3]).

▶ **Lemma 14.** *Let $\mathbb{C} \in \mathsf{StrMonCat}$. When a is an invertible object in the monoid $\mathrm{Ob}(\mathbb{C})$ with inverse b, the pair $(a \otimes \mathsf{x}_O \otimes b, \mathsf{id}(a) \otimes \mathsf{x}_A \otimes \mathsf{id}(b))$ is an element of $\mathcal{Z}(\mathbb{C})$.*

Proof. To show that $(a \otimes \mathsf{x}_O \otimes b, \mathsf{id}(a) \otimes \mathsf{x}_A \otimes \mathsf{id}(b))$ is an element of isotropy, one can straightforwardly verify that it is invertible, commutes generically with all operations of $\mathbb{T}$, and reflects definedness (for details, see [9, Proposition 3.9.35]). However, it is less work to show directly that given a strict monoidal functor $F : \mathbb{C} \to \mathbb{D}$, we obtain an automorphism α_F of $\mathbb{D}$ as follows. On objects we set $\alpha_F(d) = Fa \otimes d \otimes Fb$, while on arrows we set $\alpha_F(f) = \mathsf{id}(Fa) \otimes f \otimes \mathsf{id}(Fb)$. It is routine to check that this defines an automorphism and that the family α_F is natural in F. ◀

The above lemma gives us a mapping $\theta_{\mathbb{C}} : \mathcal{Z}_{\mathsf{Mon}}(\mathrm{Ob}(\mathbb{C})) \to \mathcal{Z}(\mathbb{C})$. It is easily verified that this is in fact a group homomorphism, natural in $\mathbb{C}$.

Next, we define a retraction σ of θ. This is done categorically using the right adjoint ∇ to Ob. Concretely, given an element of isotropy $\alpha \in \mathcal{Z}(\mathbb{C})$, we define an element $\sigma_{\mathbb{C}}(\alpha) \in \mathcal{Z}_{\mathsf{Mon}}(\mathrm{Ob}(\mathbb{C}))$ as follows: consider a monoid homomorphism $h : \mathrm{Ob}(\mathbb{C}) \to N$. This corresponds by the adjunction $\mathrm{Ob} \dashv \nabla$ to a strict monoidal functor $\tilde{h} : \mathbb{C} \to \nabla(N)$; the component of α at $\tilde{h}$ is an automorphism of $\nabla(N)$, whence $\mathrm{Ob}\,(\alpha_{\tilde{h}})$ is an automorphism of N (using the fact that $\mathrm{Ob} \circ \nabla = 1$). This leads to:

▶ **Lemma 15.** *If $\mathbb{C} \in \mathsf{StrMonCat}$, the map $\sigma_{\mathbb{C}} : \mathcal{Z}(\mathbb{C}) \to \mathcal{Z}_{\mathsf{Mon}}(\mathrm{Ob}(\mathbb{C}))$ is a group homomorphism.*

Interpreting this syntactically, we find that if $(s_O, s_A) \in \mathcal{Z}(\mathbb{C})$, then $s_O \in \mathcal{Z}_{\mathsf{Mon}}(\mathrm{Ob}(\mathbb{C}))$, and hence $s_O = a \otimes \mathsf{x}_O \otimes b$ for an invertible object a with inverse b. We also see that $\sigma_{\mathbb{C}}$ is a retraction of $\theta_{\mathbb{C}}$, i.e. that $\sigma_{\mathbb{C}} \circ \theta_{\mathbb{C}} = 1$.

Since $\theta_{\mathbb{C}}$ is a section, it now remains to show that $\theta_{\mathbb{C}}$ is an epimorphism of groups, i.e. is surjective. So we must show for any element of isotropy $(s_O, s_A) = (a \otimes \mathsf{x}_O \otimes b, s_A) \in \mathcal{Z}(\mathbb{C})$ (with invertible object a and inverse b) that we have $s_A = \mathsf{id}(a) \otimes \mathsf{x}_A \otimes \mathsf{id}(b)$. To this end, we first note that since (s_O, s_A) commutes generically with the operations dom and cod we get

$$a \otimes \mathsf{dom}(\mathsf{x}_A) \otimes b = s_O[\mathsf{dom}(\mathsf{x}_A)/\mathsf{x}_O] = \mathsf{dom}(s_A)$$

and likewise

$$a \otimes \mathsf{cod}(\mathsf{x}_A) \otimes b = s_O[\mathsf{cod}(\mathsf{x}_A)/\mathsf{x}_O] = \mathsf{cod}(s_A).$$

Thus, by uniqueness of normal forms, s_A must have the form $f \otimes \mathsf{x}_A \otimes g$ for some morphisms $f : a \to a$ and $g : b \to b$ of $\mathbb{C}$. So we must now show that $f = \mathsf{id}(a)$ and $g = \mathsf{id}(b)$, and for that we use the fact that (s_O, s_A) commutes generically with id, giving

$$f \otimes \mathsf{id}(\mathsf{x}_O) \otimes g = s_A[\mathsf{id}(\mathsf{x}_O)/\mathsf{x}_A] = \mathsf{id}(s_O) = \mathsf{id}(a \otimes \mathsf{x}_O \otimes b) = \mathsf{id}(a) \otimes \mathsf{id}(\mathsf{x}_O) \otimes \mathsf{id}(b).$$

We now get the desired equalities $f = \mathsf{id}(a)$ and $g = \mathsf{id}(b)$ by appealing to the uniqueness of normal forms. This concludes the proof of Theorem 8.

5 Further examples and applications

In this section we briefly explore some further theories of interest, and indicate the extent to which the analysis of the case of strict monoidal categories can be generalized.

5.1 Internal categories

The analysis of strict monoidal categories reveals that it is profitable, at least for the purposes of understanding isotropy, to regard strict monoidal categories as internal categories in the category Mon of monoids. This naturally raises the following question: are there other algebraic theories $\mathbb{T}$ for which the forgetful functor $\mathrm{Ob} : \mathsf{Cat}(\mathbb{T}\mathsf{mod}) \to \mathbb{T}\mathsf{mod}$ induces an isomorphism on the level of isotropy groups?

Let us first state which of the ideas from the case of monoids carry over to a general algebraic theory $\mathbb{T}$. First of all, we still have a string of adjunctions

$$\begin{tikzcd} \mathsf{Cat}(\mathbb{T}\mathsf{mod}) \arrow[r, "\mathrm{Ob}" description] & \mathbb{T}\mathsf{mod} \arrow[l, bend right, "\Delta"', "\perp"] \arrow[l, bend left, "\nabla", "\perp"'] \end{tikzcd}$$

with $\mathrm{Ob} \circ \nabla \cong 1 \cong \mathrm{Ob} \circ \Delta$, since $\mathbb{T}\mathsf{mod}$ has finite limits. This allows us to deduce the existence of a pair of natural comparison homomorphisms

$$\theta_{\mathbb{C}} : \mathcal{Z}_{\mathbb{T}}(\mathrm{Ob}(\mathbb{C})) \to \mathcal{Z}(\mathbb{C}) \ ; \qquad \sigma_{\mathbb{C}} : \mathcal{Z}(\mathbb{C}) \to \mathcal{Z}_{\mathbb{T}}(\mathrm{Ob}(\mathbb{C}))$$

with $\sigma \circ \theta = 1$ (here $\mathcal{Z}$ denotes the isotropy of $\mathsf{Cat}(\mathbb{T}\mathsf{mod})$ and $\mathcal{Z}_{\mathbb{T}}$ that of $\mathbb{T}\mathsf{mod}$). We thus have:

▶ **Lemma 16.** *Let $\mathbb{T}$ be any algebraic theory and $\mathbb{C}$ any internal category in $\mathbb{T}\mathsf{mod}$. Then $\mathcal{Z}_{\mathbb{T}}(\mathrm{Ob}(\mathbb{C}))$ is a retract of $\mathcal{Z}(\mathbb{C})$, naturally in $\mathbb{C}$.*

In the case of strict monoidal categories, we were able to prove syntactically that the embedding-retraction pair (θ, σ) is an isomorphism. The same proof can also be applied in at least two other cases of interest. Recall that a *crossed module* (A, G, δ, a) consists of a pair of groups A, G, a group homomorphism $\delta : A \to G$, and a group homomorphism $a : G \to \mathsf{Aut}(A)$ from G to the automorphism group of A, making certain diagrams commute. If XMod denotes the category of crossed modules and their morphisms, then it is also true that XMod is equivalent to the category $\mathsf{Cat}(\mathsf{Grp})$ of internal categories in Grp (cf. e.g. [7, XII.8]).

▶ **Proposition 17.** *The isotropy group of a crossed module (A, G, δ, a) is isomorphic to G.*

Proof. When composing the functor $\mathrm{Ob} : \mathsf{Cat}(\mathsf{Grp}) \to \mathsf{Grp}$ with the equivalence $\mathsf{XMod} \xrightarrow{\sim} \mathsf{Cat}(\mathsf{Grp})$, one obtains the forgetful functor which sends a crossed module (A, G, δ, a) to G. Moreover, the isotropy group of a group G is G itself by [6, Example 4.1]. ◀

▶ **Proposition 18.** *The covariant isotropy group $\mathcal{Z} : \mathsf{StrSymMonCat} \to \mathsf{Grp}$ of strict **symmetric** monoidal categories is constant, with value the trivial group.*

Proof. The isotropy group of commutative monoids is trivial by [6, Example 4.4]. ◀

We want to emphasize that the preceding proposition is *not* inconsistent with Theorem 8: while Theorem 8 asserts that the covariant isotropy group of a strict symmetric monoidal category $\mathbb{C}$ *in the category* $\mathsf{StrMonCat}$ is isomorphic to its Picard group (which may be non-trivial), the preceding proposition asserts that the covariant isotropy group of $\mathbb{C}$ *in the full subcategory* $\mathsf{StrSymMonCat}$ is trivial. In other words, if $\mathcal{A}$ is a full subcategory of $\mathcal{B}$, with covariant isotropy groups $\mathcal{Z}_{\mathcal{A}} : \mathcal{A} \to \mathsf{Grp}$ and $\mathcal{Z}_{\mathcal{B}} : \mathcal{B} \to \mathsf{Grp}$, then $\mathcal{Z}_{\mathcal{A}}(A)$ may differ from $\mathcal{Z}_{\mathcal{B}}(A)$ for an object A in the full subcategory $\mathcal{A}$.

5.2 Presheaf categories

Using Theorem 7, we can also compute the covariant isotropy of any presheaf category $\mathsf{Set}^{\mathcal{J}}$ for a small category $\mathcal{J}$. We first axiomatize $\mathsf{Set}^{\mathcal{J}}$ as the category of models of a quasi-equational theory.

▶ **Definition 19 (Presheaf Theory).** Let $\mathcal{J}$ be a small category. We define the signature $\Sigma^{\mathcal{J}}$ to have one sort X_i for each $i \in \mathrm{Ob}(\mathcal{J})$ and one function symbol $\alpha_f : X_i \to X_j$ for each arrow $f : i \to j$ in $\mathcal{J}$.

We define the *presheaf theory* $\mathbb{T}^{\mathcal{J}}$ to be the quasi-equational theory over the signature $\Sigma^{\mathcal{J}}$ with the following axioms:

- $\top \vdash^{x:X_i} \alpha_f(x)\downarrow$ *for any* $f : i \to j$ *in* $\mathcal{J}$ (i.e. each α_f is *total*).
- $\top \vdash^{x:X_i} \alpha_{\mathsf{id}_i}(x) = x$ *for every* $i \in \mathrm{Ob}(\mathcal{J})$ (i.e. each α_{id_i} acts as an identity).
- $\top \vdash^{x:X_i} \alpha_g(\alpha_f(x)) = \alpha_{g\circ f}(x)$ for any composable pair $i \xrightarrow{f} j \xrightarrow{g} k$ in $\mathcal{J}$.

We will lighten notation and write i instead of X_i and f instead of α_f. We write x_i for an indeterminate of sort i. It is completely straightforward to verify that we have an isomorphism of categories $\mathbb{T}^{\mathcal{J}}\mathsf{mod} \cong \mathsf{Set}^{\mathcal{J}}$ (for details, see [9, Proposition 5.1.8]). So to compute the covariant isotropy group $\mathcal{Z}_{\mathsf{Set}^{\mathcal{J}}} : \mathsf{Set}^{\mathcal{J}} \to \mathsf{Grp}$ of the category $\mathsf{Set}^{\mathcal{J}}$, it is equivalent to compute the covariant isotropy group $\mathcal{Z}_{\mathbb{T}^{\mathcal{J}}} : \mathbb{T}^{\mathcal{J}}\mathsf{mod} \to \mathsf{Grp}$ of the theory $\mathbb{T}^{\mathcal{J}}$.

According to Theorem 7, we have for a $\mathbb{T}^{\mathcal{J}}$-model (i.e. a functor) $F : \mathcal{J} \to \mathsf{Set}$ that

$$\mathcal{Z}(F) \cong \left\{ [s_i]_i \in \prod_{i\in\mathcal{J}} F\langle \mathsf{x}_i\rangle_i \mid [s_i]_i \text{ is invertible and commutes gen. with all } f : i \to j \right\}.$$

Note that since all terms are provably defined in $\mathbb{T}^{\mathcal{J}}$, we can omit the condition that $[s_i]_i$ reflects definedness. We now require the following preparatory lemma.

▶ **Lemma 20.** *Let* $M \in \mathbb{T}^{\mathcal{J}}\mathsf{mod}$*. If* $f, f' : i \to j$ *are parallel arrows in* $\mathcal{J}$ *and* $\mathbb{T}^{\mathcal{J}}(M, \mathsf{x}_i) \vdash f(\mathsf{x}_i) = f'(\mathsf{x}_i)$*, then* $f = f'$*.*

Proof. The assumption $\mathbb{T}^{\mathcal{J}}(M, \mathsf{x}_i) \vdash f(\mathsf{x}_i) = f'(\mathsf{x}_i)$ implies that for any homomorphism (i.e. natural transformation) $\eta : M \to N$ in $\mathsf{Set}^{\mathcal{J}}$ we have $N(f) = N(f')$, since given any $a \in N_i$ there is a homomorphism $[\eta, a] : M\langle \mathsf{x}_i\rangle \to N$ sending x_i to a (cf. also [9, Lemma 3.1.2]). We now take $N : \mathcal{J} \to \mathsf{Set}$ to be $N := M + \mathcal{J}(i, -)$ and η to be the coproduct inclusion. Then $f = f \circ \mathsf{id}(i) = N(f)(\mathsf{id}(i)) = N(f')(\mathsf{id}(i)) = f' \circ \mathsf{id}(i) = f'$, as required. ◀

As a consequence of this lemma, we find that any term congruence class $[t] \in M\langle \mathsf{x}_i\rangle$ has a *unique* representation as $t \equiv a$ for some $a \in M_j$ or $t \equiv f(\mathsf{x}_i)$ for some f with domain i, depending on whether the indeterminate x_i occurs in t.

Let $\mathsf{Aut}(\mathsf{Id}_{\mathcal{J}})$ be the group of natural automorphisms of the identity functor $\mathsf{Id}_{\mathcal{J}} : \mathcal{J} \to \mathcal{J}$ of a small category $\mathcal{J}$, which is sometimes called the *center* of $\mathcal{J}$. We now have:

▶ **Proposition 21.** *Let* $\mathcal{J}$ *be a small category. For any* $M \in \mathbb{T}^{\mathcal{J}}\mathsf{mod}$ *we have*

$$\mathcal{Z}(M) = \left\{ ([\psi_i(\mathsf{x}_i)])_i \in \prod_{i\in\mathcal{J}} M\langle \mathsf{x}_i\rangle_i : \psi \in \mathsf{Aut}(\mathsf{Id}_{\mathcal{J}}) \right\}.$$

Proof. It is straightforward to prove the right-to-left inclusion using the assumption that ψ is a natural automorphism of $\mathsf{Id}_{\mathcal{J}}$, so let us turn to the less obvious converse inclusion. So suppose that $([s_i])_{i\in\mathcal{J}} \in \mathcal{Z}(M) \subseteq \prod_i M\langle \mathsf{x}_i\rangle_i$. By the lemma, as well as the fact that invertible terms must contain the indeterminate, we may represent $s_i = \psi_i(\mathsf{x}_i)$, where $\psi_i : i \to i$ is a map in $\mathcal{J}$. We show that $\psi := (\psi_i)_{i\in\mathcal{J}}$ is a natural automorphism of $\mathsf{Id}_{\mathcal{J}}$. First, each $\psi_i : i \to i$ is an isomorphism: take the inverse $([t_i])_i$ of $([s_i])_i$, and represent this inverse as $\chi_i(\mathsf{x}_i)$ for $\chi_i : i \to i$. Since $\mathbb{T}^{\mathcal{J}}(M, \mathsf{x}_i)$ then proves the equations $(\psi_i \circ \chi_i)(\mathsf{x}_i) = \psi_i(\chi_i(\mathsf{x}_i)) = \mathsf{x}_i = \mathsf{id}_i(\mathsf{x}_i)$ and $(\chi_i \circ \psi_i)(\mathsf{x}_i) = \mathsf{id}_i(\mathsf{x}_i)$, it follows by Lemma 20 that ψ_i is the inverse of χ_i.

To show that ψ is natural, let $f : j \to k$ be any arrow in $\mathcal{J}$, and let us show that $\psi_k \circ f = f \circ \psi_j$. We know that $([\psi_i(\mathsf{x}_i)])_i = [s_i]_i$ commutes generically with the function symbol $f : X_j \to X_k$ of $\Sigma^{\mathcal{J}}$, which implies that $\mathbb{T}^{\mathcal{J}}(M, \mathsf{x}_j) \vdash (\psi_k \circ f)(\mathsf{x}_j) = (f \circ \psi_j)(\mathsf{x}_j)$, from which we obtain the required $\psi_k \circ f = f \circ \psi_j$ again by Lemma 20. Thus $\psi : \mathsf{Id}_{\mathcal{J}} \xrightarrow{\sim} \mathsf{Id}_{\mathcal{J}}$ is indeed a natural automorphism with $([s_i])_i = ([\psi_i(\mathsf{x}_i)])_i$. ◀

▶ **Corollary 22.** *Let $\mathcal{J}$ be a small category. For any functor $F : \mathcal{J} \to \mathsf{Set}$ we have $\mathcal{Z}(F) \cong \mathsf{Aut}(\mathsf{Id}_{\mathcal{J}})$, and hence the covariant isotropy group functor of $\mathsf{Set}^{\mathcal{J}}$ is constant on the automorphism group of $\mathsf{Id}_{\mathcal{J}}$.*

Proof. Given $([s_i])_{i \in \mathcal{J}} \in \mathcal{Z}(F)$, we know by Proposition 21 that there is some $\psi \in \mathsf{Aut}(\mathsf{Id}_{\mathcal{J}})$ with $[s_i]_i = [\psi_i(\mathsf{x}_i)]_i$. We now show that this assignment $([s_i])_i \mapsto \psi$ is a well-defined group isomorphism $\mathcal{Z}(F) \xrightarrow{\sim} \mathsf{Aut}(\mathsf{Id}_{\mathcal{J}})$. It is well-defined, because if there is also some $\chi \in \mathsf{Aut}(\mathsf{Id}_{\mathcal{J}})$ with $[s_i]_i = [\psi_i(\mathsf{x}_i)]_i = [\chi_i(\mathsf{x}_i)]_i$, then from Lemma 20 we obtain $\psi = \chi$. It is clearly injective, it is surjective by Proposition 21, and it is readily seen to preserve group multiplication, so that it is indeed a group isomorphism. ◀

We can now use Corollary 22 to characterize the covariant isotropy groups of certain presheaf categories of interest.

▶ **Proposition 23.** *If M is a monoid, then the covariant isotropy group $\mathcal{Z} : \mathsf{Set}^M \to \mathsf{Grp}$ of the category of M-sets and M-equivariant maps is constant on $\mathsf{Inv}(Z(M))$, the subgroup of invertible elements of the center of M. In particular, if G is a group, then the covariant isotropy group $\mathcal{Z} : \mathsf{Set}^G \to \mathsf{Grp}$ is constant on $Z(G)$.*

Proof. The result follows immediately from Corollary 22 and the observation that the automorphism group of the identity functor on the monoid M, regarded as a one-object category, is isomorphic to $\mathsf{Inv}(Z(M))$. ◀

▶ **Proposition 24.** *Let $\mathcal{J}$ be a **rigid** category, i.e. a category whose objects have no non-identity automorphisms (e.g. $\mathcal{J}$ could be a preorder or poset). Then the covariant isotropy group $\mathcal{Z} : \mathsf{Set}^{\mathcal{J}} \to \mathsf{Grp}$ is trivial.*

We point out that Corollary 22 illustrates an important difference between covariant isotropy $\mathsf{Set}^{\mathcal{J}} \to \mathsf{Grp}$ and contravariant isotropy $\left(\mathsf{Set}^{\mathcal{J}}\right)^{\mathrm{op}} \to \mathsf{Grp}$. Indeed, the latter is generally *not* constant, but is a representable functor $F \mapsto \mathsf{Set}^{\mathcal{J}}[F, Z]$ for a suitable presheaf of groups Z, that is, an internal group object in $\mathsf{Set}^{\mathcal{J}}$. The connection between covariant and contravariant isotropy is then as follows: the group of global sections of Z is isomorphic to the group $\mathsf{Aut}(\mathsf{Id}_{\mathcal{J}})$:

$$\Gamma(Z) = \mathsf{Set}^{\mathcal{J}}(1, Z) \cong \mathcal{Z}(F) \text{ for } F : \mathcal{J} \to \mathsf{Set}.$$

6 Conclusions and future work

We have shown how a syntactic description of polymorphic automorphisms can be fruitfully applied to characterize the covariant isotropy of several kinds of structures of relevance in logic, algebra, and computer science. Most notably, we have shown that the covariant isotropy group of a strict monoidal category coincides with its Picard group of $\otimes$-invertible objects. We have also shown that the covariant isotropy group of a presheaf category $\mathsf{Set}^{\mathcal{J}}$ behaves quite differently from the contravariant one, in that it is the *constant* group with value $\mathsf{Aut}(\mathsf{Id}_{\mathcal{J}})$.

There are several open questions and possible lines for further inquiry:

1. The generalization from algebraic to quasi-equational theories presented in this paper is the first step on a path upwards through the various fragments of logic. In particular, we hope to generalize some of the techniques to determine the isotropy groups of some *geometric* theories of interest.

2. We have shown how to determine the covariant isotropy groups of presheaf categories, but we have left open the question of how to determine the isotropy of *sheaf* toposes. In particular, it would be of interest to determine the covariant isotropy of the topos of nominal sets (also known as the Schanuel topos).
3. For a theory $\mathbb{T}$ and small category $\mathcal{J}$, there is a theory $\mathbb{S} = \mathbb{S}(\mathbb{T}, \mathcal{J})$ with $\mathbb{S}\mathsf{mod} \cong \mathbb{T}\mathsf{mod}^{\mathcal{J}}$ (in Section 5.2 we considered the special case where $\mathbb{T}$ is the theory of sets). In [9, Chapter 5] the second author has obtained, under mild assumptions on $\mathbb{T}$, a description of the covariant isotropy group of $\mathbb{T}^{\mathcal{J}}\mathsf{mod}$ in terms of $\mathsf{Aut}(\mathsf{Id}_{\mathcal{J}})$ and the isotropy group of $\mathbb{T}$.
4. We have not yet investigated in detail how isotropy behaves with respect to morphisms of theories $\rho : \mathbb{T} \to \mathbb{S}$. We have seen a rather special case in Section 4 with $\mathrm{Ob} : \mathsf{StrMonCat} \to \mathsf{Mon}$, but the general case is more involved.
5. One possible perspective on the theory of strict monoidal categories is that it is a *tensor product* of the theory of categories with that of monoids. This leads to the question of whether, under suitable hypotheses on the theories $\mathbb{T}$ and $\mathbb{S}$, we can describe the isotropy of the tensor product theory $\mathbb{T} \otimes \mathbb{S}$ in terms of that of $\mathbb{T}$ and $\mathbb{S}$.
6. One can define, for a 2-category $\mathcal{E}$ and object $X \in \mathcal{E}$, the 2-group of pseudo-natural auto-equivalences of $X/\mathcal{E} \to \mathcal{E}$. This leads to a 2-dimensional version of isotropy, taking values in 2-groups. It is then possible to show that the 2-isotropy group of a (non-strict) monoidal category (regarded as an object of the 2-category of monoidal categories and strong monoidal functors) is the Picard 2-group. This will be presented in forthcoming work.

References

1 George M. Bergman. An inner automorphism is only an inner automorphism, but an inner endomorphism can be something strange. *Publicacions Matematiques*, 56(1):91–126, 2012.

2 Peter J. Freyd. Core algebra revisited. *Theor. Comput. Sci.*, 375(1-3):193–200, 2007. `doi:10.1016/j.tcs.2006.12.033`.

3 Jonathon Funk, Pieter Hofstra, and Sakif Khan. Higher isotropy. *Theory and Applications of Categories*, 33(20):537–582, 2018.

4 Jonathon Funk, Pieter Hofstra, and Benjamin Steinberg. Isotropy and crossed toposes. *Theory and Applications of Categories*, 26(24):660–709, 2012.

5 Richard Garner. Inner automorphisms of groupoids. Preprint, available at `arXiv:1907.10378`, 2019.

6 Pieter J. W. Hofstra, Jason Parker, and Philip J. Scott. Isotropy of algebraic theories. In Sam Staton, editor, *Proceedings of the Thirty-Fourth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2018, Dalhousie University, Halifax, Canada, June 6-9, 2018*, volume 341 of *Electronic Notes in Theoretical Computer Science*, pages 201–217. Elsevier, 2018. `doi:10.1016/j.entcs.2018.11.010`.

7 Saunders Mac Lane. *Categories for the working mathematician.* Springer, second edition, 1997.

8 Erik Palmgren and Steven J. Vickers. Partial horn logic and cartesian categories. *Ann. Pure Appl. Log.*, 145(3):314–353, 2007. `doi:10.1016/j.apal.2006.10.001`.

9 Jason Parker. *Isotropy groups of quasi-equational theories.* PhD thesis, Université d'Ottawa/University of Ottawa, 2020. URL: `https://ruor.uottawa.ca/handle/10393/41032`.

10 Martin R. Pettet. On inner automorphisms of finite groups. *Proceedings of the American Mathematical Society*, 106(1):87–90, May 1989.

11 Andrew M. Pitts. Categorical logic. In *Handbook of Logic in Computer Science: Volume 5. Algebraic and Logical Structures*, pages 40–128. Oxford University Press, 2000.

12 Paul E. Schupp. A characterization of inner automorphisms. *Proceedings of the American Mathematical Society*, 101(2):226–228, October 1987.

13 Joseph R. Shoenfield. *Mathematical logic*. CRC Press, 2018.

What's Decidable About (Atomic) Polymorphism?

Paolo Pistone
University of Bologna, Italy

Luca Tranchini
Eberhard Karls Universität Tübingen, Germany

Abstract

Due to the undecidability of most type-related properties of System F like type inhabitation or type checking, restricted polymorphic systems have been widely investigated (the most well-known being ML-polymorphism). In this paper we investigate System Fat, or atomic System F, a very weak predicative fragment of System F whose typable terms coincide with the simply typable ones. We show that the type-checking problem for Fat is decidable and we propose an algorithm which sheds some new light on the source of undecidability in full System F. Moreover, we investigate free theorems and contextual equivalence in this fragment, and we show that the latter, unlike in the simply typed lambda-calculus, is undecidable.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Higher order logic

Keywords and phrases Atomic System F, Predicative Polymorphism, ML-Polymorphism, Type-Checking, Contextual Equivalence, Free Theorems

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.27

Related Version *Full Version*: https://arxiv.org/abs/2105.00748

Funding *Luca Tranchini*: DFG TR1112/4-1 "Falsity and Refutation. On the negative side of logic"

1 Introduction

Polymorphism has been a central topic in programming language theory since the late sixties. Today, most general purpose programming languages employ some kind of polymorphism. At the same time, under the Curry-Howard correspondence, quantification over types corresponds to quantification over propositions, that is, to second-order logic. In particular, System F, the archetypical type system for polymorphism, can be seen as a proof-system for (the $\Rightarrow, \forall$-fragment of) second-order intuitionistic logic.

In spite of the numerous applications of polymorphism, practically all interesting type-related properties of (Curry-style) System F (e.g. type checking, type inhabitation, etc.) are undecidable, making this language impractical for any reasonable implementation. This is one of the reasons why a wide literature has investigated more manageable subsystems of System F. Notably, ML-*polymorphism* [41, 42, 40] has found much success due to its decidable type-checking.

Another direction of research was that of investigating *predicative* subsystems of System F [32, 33, 34, 6]. In particular, the so-called *finitely stratified polymorphism* [33] yields a stratification of System F through a sequence of predicative systems $(\mathrm{F}_n)_{n\in\mathbb{N}}$ of growing expressive power (notably, F_0 is the simply typed λ-calculus STλC, and ML-polymorphism coincides with the rank-1 part of F_1). Yet, in spite of such limitations, type checking becomes undecidable already at level 1 of this hierarchy [18].

Could one tell exactly at which point, in the range from the simply typed λ-calculus and ML to full System F, the type-related properties of polymorphism become undecidable?

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi; Article No. 27; pp. 27:1–27:23
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

	$F_0 = ST\lambda C$	F_{at}	ML	F_1	F
TI	decidable [59]	undecidable [52]	open	undecidable [57]	undecidable [37]
TC	decidable [25]	**decidable**	decidable [40]	undecidable [18]	undecidable [64]
T	decidable [25]	**decidable**	decidable [40]	undecidable [18]	undecidable [64]
CE (for numerical functions)	decidable [44]	**decidable**	**undecidable**	**undecidable**	undecidable*
CE (full)	decidable [44]	**undecidable**	**undecidable**	**undecidable**	undecidable**

Figure 1 Decidable and undecidable properties of System F and some predicative fragments (in bold the properties established in the present paper).
*: easy consequence of Rice's theorem and the typability of all primitive recursive functions in F (see also Remark 18).
**: consequence of the undecidability of (CE) for numerical functions.

Atomic Polymorphism. In more recent times Ferreira et al. have undertaken the investigation of what can be seen as the least expressive predicative fragment of F, System F_{at}, or *atomic* System F [12, 11, 13, 15, 16, 10, 9]. The predicative restriction of F_{at} is such that a universally quantified type $\forall X.A$ can be instantiated solely with an atomic type, i.e. a type variable. In this way F_{at} sits in between level 0 (i.e. STλC) and level 1 of the finitely stratified hierarchy. Actually, F_{at} can be seen as a *type refinement* system (in the sense of [39]) of STλC, since all terms typable in F_{at} are simply typable (cf. Lemma 7).

In spite of its very limited expressive power, Ferreira et al. have shown that, thanks to polymorphism, F_{at} enjoys some proof-theoretic properties that STλC lacks. In particular, they defined a predicative variant of the usual encoding of sum and product types inside F, yielding an embedding of intuitionistic propositional logic inside F_{at}. However, while propositional logic is decidable, provability in second-order propositional intuitionistic logic, even with the atomic restriction, is undecidable [56]. This argument (as recently observed in [52]) can be extended to show that the *type inhabitation* property, which is decidable for STλC, is undecidable for F_{at}.

Contributions

In this paper we investigate the following type-related properties of System F_{at}:

Type inhabitation (TI):	given A, is there t such that $\vdash t : A$?
Type-checking (TC):	given Γ, A, t, does $\Gamma \vdash t : A$?
Typability (T):	given Γ, t, is there A such that $\Gamma \vdash t : A$?
Contextual equivalence (CE):	given A, t, u such that $\vdash t, u : A$, do $\mathsf{C}[t]$ and $\mathsf{C}[u]$ reduce to the same Boolean, for all context $\mathsf{C}[\ ] : A \Rightarrow \mathsf{Bool}$?

In Fig. 1 we sum up what is already known and what is established in this paper (in bold) about such properties in predicative fragments of System F. Our main results are that in F_{at} (TC) and (T) are both decidable, and that (CE) is decidable if one restricts oneself to numerical functions, and undecidable in the general case.

Several decidability properties of F_{at} are tight, meaning that they all fail already for F_1. In these cases, our arguments can be used to shed some new insights on the broader question of understanding where the source of undecidability for such properties in full System F lies.

Plan of the paper

After recalling the syntax of F and its fragment in Curry-style and Church-style, we address the properties (TI), (TC), (T) and (CE).

Type Inhabitation. In Section 3 we shortly discuss the undecidability of (TI), by showing how the argument in [57] for System F applies to F_{at} too. This argument yields an encoding inside F_{at} of an undecidable fragment of *first-order* intuitionistic logic. We also observe that F_{at} is actually equivalent to a first-order system, namely to the $\Rightarrow, \forall$-fragment $1\mathrm{Mon}^{\Rightarrow,\forall}$ of first-order *monadic* intuitionistic logic in a language with a unique monadic predicate. To our knowledge, the undecidability of $1\mathrm{Mon}^{\Rightarrow,\forall}$ has not been previously observed (although some slightly more expressive fragments - e.g. including a primitive disjunction [19] or finitely many monadic predicates [54] - have been proven undecidable).

Type-Checking and Typability. In Section 4 we consider the type-checking problem. The undecidability of (TC) for System F was established by Wells in [64], and was later extended to all predicative systems F_n, for $n > 0$ [18]. In all these cases this result was obtained by reducing an undecidable variant of *second-order unification* (SOU) to the type-checking problem. On the other hand, the decidability of (TC) for ML (and $F_0 = \mathrm{ST}\lambda\mathrm{C}$) is based on the famous Hindley-Milner algorithm [40], which reduces this problem to *first-order unification* (FOU), which is decidable.

The fundamental source of undecidability of SOU is the presence of *cyclic* dependences between second order variables, expressed in the simplest case by equations of the form $X(t) = f(v_1, \dots, v_{k-1}, X(u), v_{k+1}, \dots, v_n)$. In fact, *acyclic* SOU is decidable [36]. When type-checking polymorphic programs, such cyclic dependencies are generated by *self-applications*, i.e. terms of the form $\lambda\vec{x}.xt_1 \dots t_{k-1}xt_{k+1} \dots t_n$. In fact, in this case the type $\forall X.A$ assigned to the variable x must satisfy a cyclic equation of the form

$$A[X \mapsto C_1] = B_1 \Rightarrow \dots \Rightarrow B_{k-1} \Rightarrow A[X \mapsto C_2] \Rightarrow B_{k+1} \Rightarrow \dots \Rightarrow B_n$$

(where C_1, C_2 are suitable type instantiations of X). By constrast, no term containing a self-application can be typed in STλC, since cyclic equations cannot be solved by FOU.

Since the terms typable in F_{at} can also be typed in STλC (cf. Lemma 7), it follows that self-applications cannot be typed in F_{at} either. Using this observation, we describe a type-checking algorithm for F_{at} which works in two phases: first, it checks (using FOU) the presence of cyclic dependencies, and returns `failure` if it detects one; then, if phase 1 succeeds, it applies (a suitable variant of) acyclic SOU to decide type-checking. From the decidability of (TC), we deduce the decidability of (T) by a standard argument (see [4]).

Contextual Equivalence. Studying the typable terms of F_{at} might not seem very interesting from a computational viewpoint, as these terms are already typable in STλC. However, due to the presence of some form of polymorphism, investigating programs in F_{at} can be interesting for *equational* reasoning, as we do in Sections 5 and 6. In standard type systems, beyond the standard notions of program equivalence arising from the operational semantics (i.e. $\beta\eta$-equivalence), there may exist several other congruences arising from either denotational models or from some notion of *contextual equivalence*. In STλC, it is well-known that $\simeq_{\beta\eta}$ coincides with the congruence induced by any infinite extensional model [58], as well as with several notions of contextual equivalence (see [5], [7]). In polymorphic type systems the picture is rather different, since $\beta\eta$-equivalence is usually weaker than the congruences arising from extensional models (see [3, 23]), and also weaker than standard notions of contextual equivalence. Moreover, while $\beta\eta$-equivalence is decidable, contextual equivalence is undecidable. Since in many practical situations (see [62, 1]) it is more convenient to reason up to notions of equivalence stronger than $\beta\eta$-equivalence, several techniques to compute (approximations of) contextual equivalence have been investigated, e.g. *free theorems* [63], *parametricity* [53], and *dinaturality* [3].

Our investigation of contextual equivalence starts in Section 5 with an exploration of equational reasoning in $\mathrm{F_{at}}$ using free theorems. We show that the predicative encodings of sum and product types of Ferreira et al. produce products and coproducts in $\mathrm{F_{at}}$ in the categorical sense, provided terms are considered up to (CE) (a fact which is known to hold in F for the usual, impredicative, encodings [23, 61]). We then investigate (CE) for typable numerical functions. Using the fact that the primitive recursive functions are *uniquely* defined in System F up to (CE), we show that (CE) for the representable numerical functions is decidable in $\mathrm{F_{at}}$, and undecidable in ML. Such results rely on the observation that (CE) becomes undecidable as soon as some super-polynomial function (like bounded multiplication) becomes representable. From this it can be deduced that (CE) is undecidable in all fragments F_n, for $n > 0$, of the finitely stratified hierarchy as well.

Finally, in Section 6 we establish that (CE) is undecidable also in $\mathrm{F_{at}}$, by showing that the type inhabitation problem for a suitable extension of $\mathrm{F_{at}}$ can be reduced to it. This result, together with the previous ones, shows that there is no hope to get a decidable contextual equivalence for polymorphic programs through a predicative restriction, and one has rather to look for other kinds of restrictions (see for instance [49]).

2 Predicative Polymorphism and System $\mathrm{F_{at}}$

The systems we consider in this paper are all restrictions of usual Church-style and Curry-style System F. The types are defined in both cases by the grammar

$$A, B ::= X \mid A \Rightarrow B \mid \forall X.A$$

starting from a countable set $\mathtt{Var}^2$ of type variables $X_1, X_2, \dots$. The terms of Church-style System F are defined by the grammar below:

$$t^A, u^A ::= x^A \mid (\lambda x^A.t^B)^{A\Rightarrow B} \mid t^{B\Rightarrow A}u^B \mid (\Lambda X.t^A)^{\forall X.A} \mid (t^{\forall X.A}C)^{A[C/X]}$$

For readability, we will often omit type annotations, when these can be guessed from the context. The terms of Curry-style System F are standard λ-terms, with typing rules defined as in Fig. 2, where Γ indicates a partial function from term variables to types with a finite support, and by $X \notin \mathrm{FV}(\Gamma)$ we indicate that X does not occur free in any type in $\mathrm{Im}(\Gamma)$. We call the type C occurring in $(t^{\forall X.A}C)^{A[C/X]}$ and in the rule $\forall$E in Fig. 2 the *witness* of the type instantiation.

We indicate *term contexts* (i.e. terms with a *hole* [ ]) as $\mathtt{C}[\ ], \mathtt{D}[\ ]$. Moreover, we let $\mathtt{C}[\ ] : A \vdash B$ be a shorthand for $x \mapsto A \vdash \mathtt{C}[x] : B$.

System F is impredicative: *any* type can figure as a witness. In particular, one can construct "circular" instantiations, in which a term of type $\forall X.A$ is instantiated with the same type as witness. A *predicative* fragment of System F is one in which witnesses are restricted in such a way to avoid such circular instantiations.

We will focus on three predicative fragments of System F, both in Church- and Curry-style. The first is System F_1, which is the fragment of F in which witnesses are *quantifier-free*. The second is System $\mathrm{F_{at}}$, which is the fragment of F in which witnesses are *atomic*, that is,

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}\ \mathrm{Var}$$

$$\frac{\Gamma, x \mapsto A \vdash t : B}{\Gamma \vdash \lambda x.t : A \Rightarrow B}\ \mathrm{Abs} \qquad \frac{\Gamma \vdash t : A \Rightarrow B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B}\ \mathrm{Appl}$$

$$\frac{\Gamma \vdash t : B \qquad X \notin \mathrm{FV}(\Gamma)}{\Gamma \vdash t : \forall X.A}\ \forall\mathrm{I} \qquad \frac{\Gamma \vdash t : \forall X.A}{\Gamma \vdash t : A[C/X]}\ \forall\mathrm{E}$$

Figure 2 Typing rules for Curry-style System F.

type variables. The third is system ML [41, 40], which essentially coincides with the *rank 1* fragment of F_1. For any type A, the rank $\mathsf{r}(A)$ is the maximum number of nesting between $\Rightarrow$ and $\forall$, and is defined inductively by $\mathsf{r}(X) = 0$, $\mathsf{r}(A \Rightarrow B) = \max\{\mathsf{r}(A)+1, \mathsf{r}(B)\}$ and $\mathsf{r}(\forall X_1 \dots X_n.A) = \mathsf{r}(A)+1$ (where $n > 0$ and A does not start with a quantifier). To define ML (since type-checking is decidable in ML, we limit ourselves to Curry-style) one first has to enrich the set of λ-terms with the let-constructor, and add a rule

$$\frac{\Gamma, x \mapsto A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{let}\ x\ \mathsf{be}\ u\ \mathsf{in}\ t : B}\ \mathrm{let}$$

ML is the fragment of the resulting system in which typing rules only contain judgements $\Gamma \vdash t : A$, where $\mathsf{r}(A) \leqslant 1$ and for all $B \in \mathrm{Im}(\Gamma)$, $\mathsf{r}(B) \leqslant 1$.

Observe that in F_1 one can encode $\mathsf{let}\ x\ \mathsf{be}\ u\ \mathsf{in}\ t$ by $(\lambda x.t)u$, so that the rule above becomes derivable. This is not possible in ML, due to the rank restriction.

Impredicative and Predicative Encodings. It is well-known that sum and product types can be encoded inside System F by letting

$$A \tilde{+} B = \forall X.(A \Rightarrow X) \Rightarrow (B \Rightarrow X) \Rightarrow X$$
$$A \tilde{\times} B = \forall X.(A \Rightarrow B \Rightarrow X) \Rightarrow X$$

where the type variable X is fresh. The encoding of term constructors $\iota_i(\cdot)$, $\langle \cdot, \cdot \rangle$ and term destructors $\mathrm{Case}_C(\cdot, x^A.\cdot, x^B.\cdot)$ and $\pi_i(\cdot)$ is given (in Church-style) by:

$$\begin{aligned} \iota_1(t) &= \Lambda X.\lambda f^{A\Rightarrow X}.\lambda g^{B\Rightarrow X}.ft & \mathrm{Case}_C(t, x^A.u, x^B.v) &= tC(\lambda x^A.u)(\lambda x^B.v) \\ \iota_2(t) &= \Lambda X.\lambda f^{A\Rightarrow X}.\lambda g^{B\Rightarrow X}.gt & \pi_1(t) &= tA\lambda x^A.\lambda y^B.x \\ \langle t, u \rangle &= \Lambda X.\lambda f^{A\Rightarrow B\Rightarrow X}.ftu & \pi_2(t) &= tB\lambda x^A.\lambda y^B.y \end{aligned}$$

At the level of provability, the encoding is *faithful*: a type is inhabited in the extension of System F with sum and product types iff the encoded type is inhabited in System F. Moreover, the encoding of $\tilde{+}$ satisfies the *disjunction property*: $A \tilde{+} B$ is inhabited iff either A or B are inhabited.

At the level of conversions, the encoding translates β-reduction step for sum and product types into (finite sequences of) β-reduction steps in F. On the other hand, the η-rules for sums and products are not translated by the β- and η- rules of System F. Yet, the equivalence generated by β- and η-rules is preserved by *contextual equivalence* in System F (more on this in Section 5).

The encoding of sum and product types is impredicative: the encoding of term destructors requires witnesses of arbitrary complexity. Notably, given a term t of type $A \tilde{+} B$, the term $\mathrm{Case}_{A \tilde{+} B}(t, x^A.\iota_1(x), x^B.\iota_2(x))$, of type $A \tilde{+} B$, has a circular instantiation of $A \tilde{+} B$.

In [12], and more recently in [9] some alternative, predicative, encodings were defined having System $\mathrm{F_{at}}$ as target. The fundamental observation is that the unrestricted $\forall$E rule is derivable from the restricted one for the types of the form $A\tilde{+}B$ and $A\tilde{\times}B$ (the authors call this phenomenon *instantiation overflow*). In fact, for any type C of System F one can define contexts $\mathsf{IO}^{+}_{C}[\ ] : A\tilde{+}B \vdash (A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C$ and $\mathsf{IO}^{\times}_{C}[\ ] : A\tilde{\times}B \vdash (A \Rightarrow B \Rightarrow C) \Rightarrow C$ by induction on C:

$$\begin{aligned}
\mathsf{IO}^{+}_{X}[\ ] = \mathsf{IO}^{\times}_{X}[\ ] &= [\ ]X \\
\mathsf{IO}^{+}_{C_1\Rightarrow C_2}[\ ] &= \lambda f^{A\Rightarrow C_1\Rightarrow C_2}.\lambda g^{B\Rightarrow C_1\Rightarrow C_2}.\lambda y^{C_1}.\mathsf{IO}^{+}_{C_2}[\ ](\lambda z^{A}.fzy)(\lambda z^{B}.gzy) \\
\mathsf{IO}^{\times}_{C_1\Rightarrow C_2}[\ ] &= \lambda f^{A\Rightarrow B\Rightarrow C_1\Rightarrow C_2}.\lambda y^{C_1}.\mathsf{IO}^{+}_{C_2}[\ ](\lambda z^{A}.\lambda w^{B}.fzwy) \\
\mathsf{IO}^{+}_{\forall Y.C'}[\ ] &= \lambda f^{A\Rightarrow\forall Y.C'}.\lambda g^{A\Rightarrow\forall Y.C'}.\Lambda Y.\mathsf{IO}^{+}_{C'}[\ ](\lambda z^{A}.fzY)(\lambda z^{B}.gzY) \\
\mathsf{IO}^{\times}_{\forall Y.C'}[\ ] &= \lambda f^{A\Rightarrow B\Rightarrow\forall Y.C'}.\Lambda Y.\mathsf{IO}^{+}_{C'}[\ ](\lambda z^{A}.\lambda w^{B}.fzwY)
\end{aligned}$$

One can thus encode the type destructors as for F, but replacing the type application xC in $\mathrm{Case}_C(t, x^A.u, x^B.v)$ with either $\mathsf{IO}^{+}_{C}[x]$ or $\mathsf{IO}^{\times}_{C}[x]$.

At the level of provability, this embedding is faithful when restricted to simple types, i.e. for the intuitionistic propositional calculus (see [13]): a simple type (possibly containing finite sums and products) is inhabited iff its encoding is inhabited in $\mathrm{F_{at}}$. However, faithfulness does not hold for the extension of $\mathrm{F_{at}}$ with sum and product types (see [47]). In particular, one can construct types C, D of F such that $C\tilde{+}D$ is inhabited in $\mathrm{F_{at}}$ while $C + D$ is *not* inhabited in the extension of $\mathrm{F_{at}}$ with sums and products. This also implies that the disjunction property fails for $C\tilde{+}D$ in $\mathrm{F_{at}}$, since neither C nor D are inhabited.

Interestingly, at the level of conversions, this encoding is stronger than the usual one: it translates not only β-reductions, but also the permutative conversions and a restricted form of η-conversion for sums, into sequences of β and η-reductions of $\mathrm{F_{at}}$ (see [11, 14, 9]).

3 Type Inhabitation

In this section we discuss type inhabitation in the systems $\mathrm{F_{at}}$ and $\mathrm{F_1}$. We briefly recall the undecidability argument for (TI) in System F from [57], and observe that this applies to $\mathrm{F_{at}}$ (a more detailed reconstruction can be found in [52]).

The argument in [57] (which was later simplified in [8]) is based on an embedding inside F of an undecidable fragment of first-order logic. We recall the argument in a few more details, so that it will be clear that the same argument shows the undecidability of type inhabitation in both $\mathrm{F_{at}}$ and $\mathrm{F_1}$.

Let $\mathsf{Dyad}_{\Rightarrow,\forall}$ indicate the $\Rightarrow,\forall$-fragment of intuitionistic first-order logic in a language with no function symbol and a finite number of at most *binary* relation symbols. We consider sequents of the form $\Gamma \vdash \bot$ where Γ consists of three type of assumptions:

i. atomic formulas different from $\bot$;
ii. closed formulas of the form $\forall\vec{\alpha}.(\varphi_1 \Rightarrow \ldots \Rightarrow \varphi_n \Rightarrow \psi)$, where $\varphi_1, \ldots, \varphi_n, \psi$ are atomic formulas and each variable in ψ occurs in some the φ_i;
iii. closed formulas of the form $\forall\alpha(\forall\beta(p(\alpha, \beta) \Rightarrow \bot) \Rightarrow \bot)$.

The problem of checking if a sequent $\Gamma \vdash \bot$ as above is deducible in $\mathsf{Dyad}_{\Rightarrow,\forall}$ is undecidable ([57], Theorem 8.8.2).

We fix a finite number of distinguished type variables:

- for each relation symbol p, three variables p_1, p_2, p_3;
- five more variables $\spadesuit, \bullet, \circ_1, \circ_2, \star$.

We let, for any type A, $A^\bullet := A \Rightarrow \bullet$, and we define, for all types A, B:

$$p_{AB} = (A^\bullet \Rightarrow p_1) \Rightarrow (B^\bullet \Rightarrow p_2) \Rightarrow p_3$$
$$p(A,B) = p_{AB} \Rightarrow \star$$

For any type A, we let $\mathcal{U}(A)$ be the set of all types $(A^\bullet \Rightarrow p_i) \Rightarrow \circ_1, A^\bullet \Rightarrow \circ_2$, where $i = 1, 2$. Given a finite list of types $A_1, \dots, A_n$, we let $\mathcal{U}(A_1, \dots, A_n) \Rightarrow B$ be a shorthand for $C_1 \Rightarrow \dots \Rightarrow C_k \Rightarrow B$, where $C_1, \dots, C_k$ are the types in $\bigcup_i \mathcal{U}(A_i)$.

Each formula φ of $\mathsf{Dyad}_{\Rightarrow,\forall}$ is translated into a type $\overline{\varphi}$ as follows:

$$\overline{p(\alpha_i, \alpha_j)} = p(X_i, X_j) \qquad \overline{\bot} = \spadesuit$$
$$\overline{\varphi \Rightarrow \psi} = \overline{\varphi} \Rightarrow \overline{\psi}$$
$$\overline{\forall \alpha_i.\varphi} = \forall \vec{X}_i.(\mathcal{U}(X_i) \Rightarrow \overline{\varphi})$$

One can easily check the following by induction:

▶ **Proposition 1.** *If $\varphi_1, \dots, \varphi_n \vdash \varphi$ is provable in $\mathsf{Dyad}_{\Rightarrow,\forall}$ and $\alpha_{i_1}, \dots, \alpha_{i_k}$ are the variables that occur in $\mathrm{FV}(\varphi)$ but not in $\mathrm{FV}(\varphi_1, \dots, \varphi_n)$, then $x_1 \mapsto \overline{\varphi}_1, \dots, x_n \mapsto \overline{\varphi}_n, \vec{y} \mapsto \mathcal{U}(X_{i_1}, \dots, X_{i_k}) \vdash t : \overline{\varphi}$ holds in $\mathrm{F_{at}}$ for some term t.*

The less trivial part is the following:

▶ **Theorem 2** ([57], Theorem 11.6.14)**.** *For all formulas $\varphi_1, \dots, \varphi_n$ satisfying i-iii, if $x_1 \mapsto \overline{\varphi}_1, \dots, x_n \mapsto \overline{\varphi}_n \vdash t : \spadesuit$ is deducible in System F, then $\varphi_1, \dots, \varphi_n \vdash \bot$ is provable in $\mathsf{Dyad}_{\Rightarrow,\forall}$.*

Since $\mathrm{F_{at}}$ and $\mathrm{F_1}$ are both fragments of F, we can freely substitute them for System F in the statement of Theorem 2. Then, together with Proposition 1 we deduce:

▶ **Corollary 3.** *(TI) is undecidable in both $\mathrm{F_{at}}$ and $\mathrm{F_1}$.*

▶ Remark 4. Although $\mathrm{F_{at}}$ and $\mathrm{F_1}$ are both undecidable, they are not equivalent at the level of provability. For instance, the type $(\forall X.X \Rightarrow Y) \Rightarrow (Z \Rightarrow Z) \Rightarrow Y$ is inhabited in $\mathrm{F_1}$ (by the term $\lambda x^{\forall X.X \Rightarrow Y}.\lambda y^{Z \Rightarrow Z}.x(Z \Rightarrow Z)y$), but not in $\mathrm{F_{at}}$ (as easily seen by a proof-search argument).

▶ Remark 5. The undecidability of the atomic fragment of (full) second-order intuitionistic logic has been known since (at least) [56]. However, from this one cannot deduce the undecidability of $\mathrm{F_{at}}$, due to the fact that disjunction is not faithfully definable in $\mathrm{F_{at}}$ (see also [47]).

▶ Remark 6. It is not difficult to see that System $\mathrm{F_{at}}$ is equivalent to a first-order system, namely to the $\Rightarrow, \forall$-fragment $\mathsf{1Mon}_{\Rightarrow,\forall}$ of *monadic* first-order intuitionistic logic in the language with no function symbol and a *unique* monadic predicate. The equivalence is given by an obvious bijection between formulas and types given by $\widehat{p(\alpha_i)} = X_i$, $\widehat{\varphi \Rightarrow \psi} = \widehat{\varphi} \Rightarrow \widehat{\psi}$ and $\widehat{\forall \alpha_i.\varphi} = \forall X_i.\widehat{\varphi}$. Hence, a consequence of Corollary 3 is that provability in $\mathsf{1Mon}_{\Rightarrow,\forall}$ is undecidable. Provability in extensions of $\mathsf{1Mon}_{\Rightarrow,\forall}$ with either finitely many monadic predicates, or with disjunction, is known to be undecidable [19, 18]. To the best of our knowledge, the undecidability of $\mathsf{1Mon}_{\Rightarrow,\forall}$ has not been observed before.

4 Typability and Type-checking

In usual implementations of polymorphic type systems the Church-style type discipline is generally considered inconvenient, due to the heavy amount of type annotations. Instead, Curry-style languages, for which a compiler can (either completely or partially) reconstruct type annotations, are generally preferred (two standard examples are the languages ML and Haskell). This is the reason why type-checking algorithms for polymorphic type systems in Curry-style (or in some variants of Curry-style with *partial* type annotations [45]) have been extensively investigated [24, 26, 64, 18].

However, while ML admits a decidable type checking in Curry-style (a main reason for its success), type checking has been shown to be undecidable for System F and most of its variants (including the predicative system F_1 [18]), making the Curry-style version of such systems impractical for implementation.

For the simply typed λ-calculus (and crucially also for ML), the type-checking problem can be reduced to *first-order unification* (FOU), that is, to the problem of unifying first-order terms (in a language with a unique binary function symbol corresponding to $\Rightarrow$). Typically, an application $tu : b$ will produce a first-order equation of the form $a_t = a_u \Rightarrow b$, where a_t, a_u are variables indicating the type of t and the type of u, respectively. As FOU is decidable, this suffices to show that type-checking is decidable in this case.

In the case of full polymorphism FOU is not sufficient to solve type-checking. In fact, already in F_1 one can type terms, like e.g. $\lambda x.xx$, which contain *self-applications*. Using FOU, $\lambda x.xx$ yields the unsolvable equation $a_x = a_x \Rightarrow b$, so it is not typable in either STλC or ML. To type-check System F programs one can replace FOU with either *semi-unification* [24, 26] or *second order unification* (SOU) [45, 18]. Here we focus on the latter: in SOU one tries to unify equations involving terms constructed from first-order variables $a, b, c, \dots$ as well as second order variables $\mathsf{F}, \mathsf{G}, \dots$. For instance, the term $\lambda x.xx$ above yields the equations

$$\mathsf{F}a = (\mathsf{F}b) \Rightarrow \mathsf{G} \tag{1}$$

where $\forall X.\mathsf{F}X$ indicates the type of x, and the variables a, b encode the possible witnesses which permit to type xx (in Church-style one could indicate this with $\lambda x^{\forall X.\mathsf{F}X}.((xa)^{\mathsf{F}a}(xb)^{\mathsf{F}b})^{\mathsf{G}}$, so that Eq. (1) is precisely what is needed to make the typing correct). A (non-unique) solution to Eq. (1) is obtained by $\mathsf{F} \mapsto \lambda x.x$, $\mathsf{G} \mapsto Z$, $a \mapsto Y \Rightarrow Z$, $b \mapsto Y$.

Unfortunately, SOU is undecidable [22]. Moreover, one can encode restricted (but still undecidable) variants of SOU in the type checking problem for F_1 [18], showing that (TC) is undecidable for F_1. A fundamental ingredient of these undecidability arguments is the appeal to *variable cycles* (see the discussion in [36]) like the one in Eq. (1), that is, to unification problems from which one can deduce equations of the form $\mathsf{F}a_1 \dots a_n = u[\mathsf{F}]$, that is, equating a second-order variable F with some term containing F itself.

Conversely, *acyclic* SOU, that is, the problem of unifying SOU problems containing no variable cycles, is decidable [36]. These observations can be used to show that type-checking is actually decidable in F_{at}. In fact, a fundamental property of F_{at} (and a reason for its very limited expressive power) is that any term typable in F_{at} is already typable in the simply-typed λ-calculus. Indeed, the following is easily checked by induction:

▶ **Lemma 7.** *If $\Gamma \vdash t : A$ is derivable in the Curry-style F_{at}, then $|\Gamma| \vdash t : |A|$ is derivable in the simply typed λ-calculus, where $|A|$ is defined by $|X| = o$, $|A \Rightarrow B| = |A| \Rightarrow |B|$, $|\forall X.A| = |A|$, and $|\Gamma|(x) = |\Gamma(x)|$.*

An immediate consequence of Lemma 7 is that one cannot type $\lambda x.xx$ in $\mathrm{F_{at}}$ and, more generally, that any λ-term that would give rise to a variable cycle cannot be typed in $\mathrm{F_{at}}$. Observe that the converse does not hold: from the fact that $|\Gamma| \vdash t : |A|$ holds, one cannot deduce $\Gamma \vdash t : A$ (take for instance $t = x$, $\Gamma(x) = X$ and $A = \forall X.X$).

However, these observations suggest that type checking for $\mathrm{F_{at}}$ can be decided by reasoning in two phases: to check if $\Gamma \vdash t : A$ is derivable in $\mathrm{F_{at}}$, first check if $|\Gamma| \vdash t : |A|$ is derivable in STλC using FOU; if this first step fails, then the original problem must fail; if the first step succeeds, then the original type-checking problem for $\mathrm{F_{at}}$ yields an instance of (a suitable variant of) acyclic SOU, which must be decidable. By reasoning in this way, one can thus establish:

▶ **Theorem 8.** *(TC) for Curry-style* $\mathrm{F_{at}}$ *is decidable.*

In App. A (and more in detail in [50]) we describe the decision algorithm for type-checking in $\mathrm{F_{at}}$, which is based on a variant of second-order unification, that we call $\mathrm{F_{at}}$*-unification*. The fundamental idea is to consider SOU problems in a language with first-order *sequence variables* $a, b, \dots$ and two kinds of second-order variables: *projection variables* $\alpha, \beta, \dots$ and *second-order variables* $\mathsf{F}, \mathsf{G}, \dots$. The intuition is that a term of the form $\alpha a_1 \dots a_n$ describes a (skolemized) witness; since the witnesses in $\mathrm{F_{at}}$ are type variables, solving for α means associating it with either a constant function or a projection. Instead, a term of the form $\mathsf{F}\mathfrak{a}_1 \dots \mathfrak{a}_n$ stands for the application of suitable witnesses $\mathfrak{a}_1, \dots, \mathfrak{a}_n$ to some type F, hence solving for F means associating it with some function $\lambda X_1 \dots X_n.A(X_1, \dots, X_n)$, where $A(X_1, \dots, X_n)$ is some type expression parametric on the type variable $X_1, \dots, X_n$. Hence, for example, checking if $\Gamma \vdash xy : \forall Z.Z$ holds in $\mathrm{F_{at}}$, where $\Gamma(x) = \forall X.X \Rightarrow X$ and $\Gamma(y) = \forall Y.Y$, yields the equations

$$\mathsf{F}X = X \Rightarrow X \qquad\qquad \mathsf{G}Y = Y$$
$$\mathsf{F}(\alpha Z) = \mathsf{G}(\beta Z) \Rightarrow \mathsf{H}Z \qquad\qquad \mathsf{H}Z = Z$$

which admit the solution $\mathsf{F} \mapsto \lambda X.X \Rightarrow X$, $\mathsf{G}, \mathsf{H} \mapsto \lambda X.X$ and $\alpha, \beta \mapsto \lambda X.X$. Instead, checking if $\Gamma \vdash xy : \forall Z.Z$, where now $\Gamma(x) = \forall X.X \Rightarrow X$ and $\Gamma(y) = Y$, yields the equations

$$\mathsf{F}X = X \Rightarrow X \qquad\qquad \mathsf{G} = Y$$
$$\mathsf{F}(\alpha Z) = \mathsf{G} \Rightarrow \mathsf{H}Z \qquad\qquad \mathsf{H}Z = Z$$

which have no solution (since one can deduce $Z = \mathsf{H}Z = Y$), showing that (TC) fails in this case (although $|\Gamma| \vdash xy : |\forall Z.Z|$ holds in the simply typed λ-calculus).

From the decidability of (TC) one can deduce the decidability of (T) by a standard argument: we can reduce (T) to (TC) by showing that a type A such that $\Gamma \vdash t : A$ holds exists iff $\Gamma \vdash (\lambda xy.y)t : \forall X.X \Rightarrow X$ holds. In fact, if $\Gamma \vdash t : A$ holds in $\mathrm{F_{at}}$, then from $\Gamma \vdash \lambda xy.y : A \Rightarrow \forall X.(X \Rightarrow X)$ we deduce $\Gamma \vdash (\lambda xy.y)t : \forall X.X \Rightarrow X$. Conversely, from $\Gamma \vdash (\lambda xy.y)t : \forall X.X \Rightarrow X$, we deduce that there exists a type A such that $\Gamma \vdash \lambda xy.y : A \Rightarrow (X \Rightarrow X)$ and $\Gamma \vdash t : A$ holds.

▶ **Corollary 9.** *(T) for Curry-style* $\mathrm{F_{at}}$ *is decidable.*

5 Equational Reasoning in System $\mathrm{F_{at}}$

As a consequence of Lemma 7 from the previous section, all terms which are typable in Curry-style $\mathrm{F_{at}}$ are simply typable. In other words, $\mathrm{F_{at}}$ can be seen as a *type refinement* system for STλC, in the sense of [39]. In particular, as we show below, the numerical functions which can be typed in $\mathrm{F_{at}}$ are precisely the simply typable ones (i.e. the so-called *extended polynomials* [55, 16]).

For this reason, investigating the typable terms of $\mathrm{F_{at}}$ might seem not very interesting from a computational viewpoint. However, in this section we show that studying such terms can be interesting for equational reasoning. In fact, similarly to System F, standard notions of contextual equivalence for $\mathrm{F_{at}}$ are stronger than $\beta\eta$-equivalence, and one can exploit well-known techniques, like the *free theorems* [63], to compute equivalences of $\mathrm{F_{at}}$-typable terms (which do not hold when viewing these terms as typed in STλC).

We first recall two standard notions of contextual equivalence:

▶ **Notation 10.** *We let* $\mathsf{Bool} = \forall X.X \Rightarrow X \Rightarrow X$ *and* $\mathsf{Nat} = \forall X.(X \Rightarrow X) \Rightarrow (X \Rightarrow X)$*. We let* $\mathbf{t} = \lambda xy.x$ *and* $\mathbf{f} = \lambda xy.y$ *indicate the two normal forms of type* Bool*, and for all* $n \in \mathbb{N}$*, we let* $\mathbf{n} = \lambda fx.(f)^n x$ *indicate the n-th Church numeral.*

▶ **Definition 11** (contextual equivalence)**.** *Let* $F^* \in \{\mathrm{F_{at}}, \mathrm{ML}, \mathrm{F_1}, \mathrm{F}\}$*. For all closed terms* t, u *of type* A *in* F^**, we let*

- $t \simeq^{F^*}_{\mathsf{Bool}} u : A$ *iff for any context* $\mathtt{C}[\ ] : A \vdash \mathsf{Bool}$ *in* F^**,* $\mathtt{C}[t] \simeq_{\beta\eta} \mathtt{C}[u]$*;*
- $t \simeq^{F^*}_{\mathsf{Nat}} u : A$ *iff for any context* $\mathtt{C}[\ ] : A \vdash \mathsf{Nat}$ *in* F^**,* $\mathtt{C}[t] \simeq_{\beta\eta} \mathtt{C}[u]$*.*

It is easily seen that $\simeq^{F^*}_{\mathsf{Bool}}$ and $\simeq^{F^*}_{\mathsf{Nat}}$ are congruences of the terms of F^*. Moreover, in System F these two congruences coincide, due to the fact that the identity relation $\mathtt{id} : \mathsf{Nat} \Rightarrow \mathsf{Nat} \Rightarrow \mathsf{Bool}$ is typable. Since this function is also typable in ML, the same holds for ML and $\mathrm{F_1}$. On the other hand, since the identity relation is *not* simply typable, we can deduce (see Lemma 16 below) that it is not typable in $\mathrm{F_{at}}$. For this reason the congruences $\simeq^{\mathrm{F_{at}}}_{\mathsf{Bool}}$ and $\simeq^{\mathrm{F_{at}}}_{\mathsf{Nat}}$ must be treated separately in this case. In what follows we will mostly focus on the latter, since the former identifies distinct normal forms of type Nat, which is not convenient for obvious computational reasons.

▶ Remark 12. The typability of the identity relation $\mathtt{id}$ implies that any extensional model of F must be *infinite*, since for all $n \in \mathbb{N}$, the interpretations of $\mathbf{n}$ and $\mathbf{n+1}$ cannot coincide. Instead, it is not difficult to construct an extensional model of $\mathrm{F_{at}}$ in which any type is interpreted by a *finite* set (to give an idea, let $\mathcal{C}_k$ be a collection of sets of cardinality bounded by a fixed $k \in \mathbb{N}$; one can let then $[\![X]\!] \in \mathcal{C}_k$, $A \Rightarrow B = [\![B]\!]^{[\![A]\!]}$ and $[\![\forall X.A]\!] = \prod_{S \in \mathcal{C}_k} [\![A]\!][X \mapsto S]$).

The so-called free theorems are a class of syntactic equations for typable terms which can be justified by relying on either relational parametricity [53] or dinaturality [3]. We let $t \approx u : A$ indicate that t, u have type A in System F, and that the equivalence $t \simeq u$ can be deduced using β-, η-rules, standard congruence rules (i.e. reflexivity, symmetry, transitivity and context closure), as well as instances of free theorems for System F.

Free theorems can be used to deduce contextual equivalence of $\mathrm{F_{at}}$-terms, thanks to the following:

▶ **Lemma 13** (free theorems in $\mathrm{F_{at}}$)**.** *Let* t, u *be terms of type* A *in* $\mathrm{F_{at}}$*. If* $t \approx u : A$*, where* t, u *are seen as terms of System F, then* $t \simeq^{\mathrm{F_{at}}}_{\mathsf{Nat}} u : A$*.*

Proof. From $t \approx u : A$ it follows $t \simeq^{\mathrm{F}}_{\mathsf{Nat}} u : A$, since $\simeq^{\mathrm{F}}_{\mathsf{Nat}}$ is the coarsest congruence not equating normal forms of type Nat. From $t \simeq^{\mathrm{F}}_{\mathsf{Nat}} u : A$ we deduce $t \simeq^{\mathrm{F_{at}}}_{\mathsf{Nat}} u : A$, since any context in $\mathrm{F_{at}}$ is a context in F. ◀

We discuss below two applications of free theorems to study (CE) in $\mathrm{F_{at}}$.

Categorical Products and Coproducts. As mentioned in Section 2, the usual encoding of products and coproducts in System F preserves β-equivalence but not η-equivalence. For this reason, the encodings of $\times$ and $+$ do not form categorical products and coproducts in System F up to $\beta\eta$-equivalence (more precisely, in the syntactic category in which objects are the types of System F and arrows are the typable terms up to $\simeq_{\beta\eta}$). Instead, it is well-known [51, 23, 61] that η-equivalence of $\times$ and $+$ is preserved in System F up to free theorems: hence $\times$ and $+$ do form categorical products and coproducts in System F up to $\simeq^{\mathrm{F}}_{\mathsf{Nat}}$ (more precisely, in the syntactic category whose arrows are the typable terms up to $\simeq^{\mathrm{F}}_{\mathsf{Nat}}$).

In a similar way, the predicative encodings of $\times$ and $+$ in $\mathrm{F_{at}}$, although preserving some restricted case of η-equivalence, still do not form categorical products and coproducts in $\mathrm{F_{at}}$ up to $\simeq_{\beta\eta}$. We will show that they similarly do form categorical products and coproducts in $\mathrm{F_{at}}$ up to $\simeq^{\mathrm{F_{at}}}_{\mathsf{Nat}}$, as a consequence of the application of free theorems.

For simplicity, we here only consider the case of $+$. However, our argument scales straightforwardly to the encoding of all finite polynomial types, i.e. of all types of the form $\sum_{i=1}^{k}\prod_{j=1}^{k_i} A_{ij}$ (see the [47] for a more detailed discussion).

The fundamental step is showing that the impredicative and predicative encodings are equivalent up to free theorems:

▶ **Lemma 14.** *For all types A, B, C and terms $x \mapsto A \vdash u : C$ and $x \mapsto B \vdash v : C$, the equivalence $\mathsf{IO}^{+}_{C}[y](\lambda x.u)(\lambda x.v) \approx \mathrm{Case}_C(y, x.u, x.v) : C$ holds in System* F*.*

Proof. The free theorem associated with the type $A\tilde{+}B$ is the schematic equation

$$\mathrm{Case}_E(t_1, x.\mathsf{C}[t_2], x.\mathsf{C}[t_3]) \approx \mathsf{C}\Big[\mathrm{Case}_D(t_1, x.t_2, x.t_3)\Big] \tag{2}$$

where $\vdash t_1 : A\tilde{+}B$, $x \mapsto A \vdash t_2 : D$, $x \mapsto B \vdash t_2 : D$ and $\mathsf{C}[\;] : D \vdash E$. In fact, this equation is an instance of the *dinaturality* condition for the type $A\tilde{+}B$ (see [51, 23, 49]).

We argue by induction on C:

- if $C = Y$, then $\mathsf{IO}^{+}_{C}[y](\lambda x.u)(\lambda x.v) = yY(\lambda x.u)(\lambda x.v) = \mathrm{Case}_C(y, x.u, x.v)$;
- if $C = C_1 \Rightarrow C_2$, then

$$\begin{aligned}
\mathsf{IO}^{+}_{C}[y](\lambda x.u)(\lambda x.v) &= \Big(\lambda fgz.\mathsf{IO}^{+}_{C_2}[y](\lambda x.fxz)(\lambda x.gxz)\Big)(\lambda x.u)(\lambda x.v)\\
&\overset{\text{[I.H.]}}{\approx} \Big(\lambda fgz.\mathrm{Case}_{C_2}(y, x.fxz, x.gxz)\Big)(\lambda x.u)(\lambda x.v)\\
&\simeq_\beta \lambda z.\mathrm{Case}_{C_2}(y, x.uz, x.vz)\\
&\approx \lambda z.\Big(\mathrm{Case}_C(y, x.u, x.v)\Big)z\\
&\simeq_\eta \mathrm{Case}_C(y, x.u, x.v)
\end{aligned}$$

  where in the penultimate step we applied Eq. (2) with the context $\mathsf{C}[\;] = [\;]z : C \vdash C_2$.
- if $C = \forall Z.C'$, then

$$\begin{aligned}
\mathsf{IO}^{+}_{C}[y](\lambda x.u)(\lambda x.v) &= \Big(\lambda fg.\Lambda Z.\mathsf{IO}^{+}_{C'}[y](\lambda x.fxZ)(\lambda x.gxZ)\Big)(\lambda x.u)(\lambda x.v)\\
&\overset{\text{[I.H.]}}{\approx} \Big(\lambda fg.\Lambda Z.\mathrm{Case}_{C'}(y, x.fxZ, x.gxZ)\Big)(\lambda x.u)(\lambda x.v)\\
&\simeq_\beta \Lambda Z.\mathrm{Case}_{C'}(y, x.uZ, x.vZ)\\
&\approx \Lambda Z.\Big(\mathrm{Case}_C(y, x.u, x.v)\Big)Z\\
&\simeq_\eta \mathrm{Case}_C(y, x.u, x.v)
\end{aligned}$$

  where in the penultimate step we applied Eq. (2) with the context $\mathsf{C}[\;] = [\;]Z : C \vdash C'$. ◀

▶ **Proposition 15.** *$A\tilde{+}B$ is a categorical coproduct in* $\mathrm{F_{at}}$ *up to* $\simeq^{\mathrm{F_{at}}}_{\mathsf{Nat}}$.

Proof. It suffices to check that the η-rule of the coproduct (see [29]) holds in $\mathrm{F_{at}}$. By translating this rule in F one obtains the equation

$$y \approx \mathrm{Case}_{A\tilde{+}B}(y, x.\iota_1(x), x.\iota_2(x)) : A\tilde{+}B$$

which holds in F up to free theorems (see [51, 23, 61]). Using Lemma 14 we thus deduce that $y \approx \mathsf{IO}^+_{A\tilde{+}B}[y](\lambda x.\iota_1(x))(\lambda x.\iota_2(x)) : A\tilde{+}B$ holds in F, and by Lemma 13 we deduce $y \simeq^{\mathrm{F_{at}}}_{\mathsf{Nat}} \mathsf{IO}^+_{A\tilde{+}B}[y](\lambda x.\iota_1(x))(\lambda x.\iota_2(x)) : A\tilde{+}B$. ◀

Numerical Functions. We now consider the representable numerical functions, that is, the closed typable terms of type $\mathsf{Nat} \Rightarrow \ldots \Rightarrow \mathsf{Nat} \Rightarrow \mathsf{Nat}$. In this case we can strengthen Lemma 7 as follows:

▶ **Lemma 16.** *For any β-normal λ-term t, $\vdash t : \mathsf{Nat} \Rightarrow \ldots \Rightarrow \mathsf{Nat} \Rightarrow \mathsf{Nat}$ holds in Curry-style* $\mathrm{F_{at}}$ *iff $\vdash t : |\mathsf{Nat}| \Rightarrow \ldots \Rightarrow |\mathsf{Nat}| \Rightarrow |\mathsf{Nat}|$ holds in* STλC.

Proof. One direction follows from Lemma 7. For the converse one, let t (which we can suppose w.l.o.g. to be of the form $\lambda x_1 \ldots x_k.u$) be such that $\vdash t : |\mathsf{Nat}| \Rightarrow \ldots \Rightarrow |\mathsf{Nat}| \Rightarrow |\mathsf{Nat}|$. By letting $\mathsf{Nat}[X] = (X \Rightarrow X) \Rightarrow (X \Rightarrow X)$ we deduce that $\{x_i \mapsto \mathsf{Nat}[X]\} \vdash u : \mathsf{Nat}[X]$ holds in $\mathrm{F_{at}}$, and thus that $\{x_i \mapsto \mathsf{Nat}\} \vdash u : \mathsf{Nat}[X]$ holds too, from which we conclude $\vdash u : \mathsf{Nat} \Rightarrow \ldots \Rightarrow \mathsf{Nat} \Rightarrow \mathsf{Nat}$. ◀

A consequence of Lemma 16 is that the representable numerical functions in $\mathrm{F_{at}}$ are precisely the extended polynomials, i.e. the smallest class of functions arising from projections, constant functions, addition, multiplication and the `iszero` function. Instead, it is well-known that the predecessor function (which is not an extended polynomial) is typable in ML [17] and, more generally, the representable functions of ML are included in the class $\mathcal{E}_3$ of the Grzegorczyk hierarchy [33].

Still, in both STλC and $\mathrm{F_{at}}$ the same extended polynomial can be represented by different normal forms. For instance the two normal forms $\lambda xyfz.x(yf)z$ and $\lambda xyfz.y(xf)z$ (encoding the algorithms $n, m \mapsto \underbrace{m + \cdots + m}_{n \text{ times}}$ and $n, m \mapsto \underbrace{n + \cdots + n}_{m \text{ times}}$) both represent the multiplication function.

In System F, one can show that all primitive recursive functions are *uniquely defined* up to free theorems, that is, that for any two terms t, u representing the same primitive recursive function, one can prove $t \approx u$ (see [48], Section 7.5). Using Lemma 13 we deduce then:

▶ **Lemma 17.** *For all $t, u : \mathsf{Nat} \Rightarrow \ldots \Rightarrow \mathsf{Nat} \Rightarrow \mathsf{Nat}$ in $\mathrm{F}^* \in \{\mathrm{F_{at}}, \mathrm{ML}, \mathrm{F_1}, \mathrm{F}\}$, if for all $p_1, \ldots, p_k \in \mathbb{N}$, $t\mathbf{p}_1 \ldots \mathbf{p}_k \simeq_{\beta\eta} u\mathbf{p}_1 \ldots \mathbf{p}_k : \mathsf{Nat}$, then $t \simeq^{\mathrm{F}^*}_{\mathsf{Nat}} u$.*

▶ Remark 18. From Lemma 17 and the fact that all primitive recursive functions are typable in F, one can deduce that $\simeq^{\mathrm{F}}_{\mathsf{Nat}}$ for numerical functions is undecidable in F as a consequence of Rice's theorem.

The problem $\mathrm{Eq}_{\mathcal{C}}$ of deciding $f = g$, where f, g belong to some subclass $\mathcal{C}$ of the primitive recursive functions, is well-investigated. In particular, it is known that:

- if $\mathcal{C}$ is the class of extended polynomials, then $\mathrm{Eq}_{\mathcal{C}}$ is decidable [38];
- if $\mathcal{C}$ contains projections, constants, $+$, $\times$ and bounded multiplication, then $\mathrm{Eq}_{\mathcal{C}}$ is undecidable [31].

From these facts, using Lemma 17, we deduce then:

▶ **Proposition 19.**

(i) *The problem of deciding* $\simeq_{\mathsf{Nat}}^{\mathrm{F_{at}}}$ *over numerical functions in* $\mathrm{F_{at}}$ *is decidable.*

(ii) *The problem of deciding* $\simeq_{\mathsf{Nat}}^{\mathrm{F}^*}$ *over numerical functions in* $\mathrm{F}^* \in \{\mathrm{ML}, \mathrm{F}_1\}$ *is undecidable.*

Proof. (i) is immediate from Lemma 16 and Lemma 17. To prove (ii) it suffices to show that the representable functions in ML are closed under bounded multiplication. We show this fact in detail in [50], App. B. ◀

An immediate corollary is that (CE) is undecidable in both ML and F_1.

6 Contextual Equivalence is Undecidable

In this section we show that the congruences $\simeq_{\mathsf{Nat}}^{\mathrm{F_{at}}}$ and $\simeq_{\mathsf{Bool}}^{\mathrm{F_{at}}}$ are both undecidable. To do this, we will reduce the type inhabitation problem for a suitable extension of $\mathrm{F_{at}}$ to contextual equivalence. We discuss in some detail the undecidability argument for $\simeq_{\mathsf{Bool}}^{\mathrm{F_{at}}}$, while the (very similar) argument for $\simeq_{\mathsf{Nat}}^{\mathrm{F_{at}}}$ can be found in [50], App. C.

Let $\mathrm{F}_{\mathrm{at}}^{\clubsuit}$ be the extension of $\mathrm{F_{at}}$ with new a type constant $\clubsuit$ and a new term constant $\star : \clubsuit$. It is not difficult to see that the undecidability argument for (TI) from Section 3 also applies to $\mathrm{F}_{\mathrm{at}}^{\clubsuit}$.

Let $\widetilde{\top} : \forall X.X \Rightarrow X$ and $\mathsf{Id} := \Lambda X.\lambda x.x$ be the unique closed β-normal term of type $\widetilde{\top}$.

The fundamental idea will be to construct, for each type A of $\mathrm{F}_{\mathrm{at}}^{\clubsuit}$, two terms t_A, u_A of type $(A^* \widetilde{+} \widetilde{\top}) \Rightarrow \mathsf{Bool}$ (where $A^* = Y \Rightarrow A[Y/\clubsuit]$, for some fresh Y), such that $t_A \simeq_{\mathsf{Bool}}^{\mathrm{F_{at}}} u_A$ holds in $\mathrm{F_{at}}$ iff A is inhabited in $\mathrm{F}_{\mathrm{at}}^{\clubsuit}$.

Let us fix a type A of $\mathrm{F}_{\mathrm{at}}^{\clubsuit}$, a variable Y not occurring free in A, and let $A^* = Y \Rightarrow A[Y/\clubsuit]$. We let u_A, v_A be the terms below:

$$u_A = \lambda x.\mathbf{f} \qquad v_A = \lambda x.\mathsf{IO}^{+}_{\mathsf{Bool}}[\ ](\lambda x.\mathbf{t})(\lambda x.\mathbf{f})$$

First observe that if there exists some term t such that $\vdash t : A$ holds in $\mathrm{F}_{\mathrm{at}}^{\clubsuit}$, then we can construct a context $\mathtt{K}[\ ] : (A^* \widetilde{+} \widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash \mathsf{Bool}$ separating u_A and v_A: let $t^* = \lambda y.t[y/\star]$, so that $\vdash t^* : A^*$ and let $\mathtt{K}[\ ] = [\ ](\iota_1(t^*))$. We then have $\mathtt{K}[u_A] \simeq_\beta \mathbf{f}$ and $\mathtt{K}[v_A] \simeq_\beta \mathsf{IO}^{+}_{\mathsf{Bool}}[\iota_1(t^*)](\lambda x.\mathbf{t})(\lambda x.\mathbf{f}) \simeq_{\beta\eta} (\lambda x.\mathbf{t})t^* \simeq_\beta \mathbf{t}$.

The difficult part is to show that if A is not provable in $\mathrm{F}_{\mathrm{at}}^{\clubsuit}$, then no context $\mathtt{K}[\ ] : (A^* \widetilde{+} \widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash \mathsf{Bool}$ can separate u_A and v_A. We will establish this fact by analyzing all possible β-normal term contexts of type $(A^* \widetilde{+} \widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash \mathsf{Bool}$.

In the following, for a term context $\mathtt{K}[\ ]$, we let $\mathtt{K}[\ ] : A \vdash^{\Gamma} B$ be a shorthand for $\Gamma, x \mapsto A \vdash \mathtt{K}[\ ] : B$ (where we suppose that Γ is not defined on x).

We let $\mathbb{G}_1$-$\mathbb{G}_4$ be the families of term contexts defined by mutual recursion as shown in Fig. 3, and typed according to the contexts below

$$\begin{aligned} &\Gamma = \{x_1 \mapsto Z_1, x_1' \mapsto Z_1, \dots, x_p \mapsto Z_p, x_p' \mapsto Z_p\} && \Theta = \{w_1 \mapsto W_1, \dots, w_q \mapsto W_q\} \\ &\Delta = \{y_1 \mapsto A^* \Rightarrow Y_1, \dots, y_r \mapsto A^* \Rightarrow Y_r\} && \Sigma = \{z_1 \mapsto \widetilde{\top} \Rightarrow Y_1, \dots, z_r \mapsto \widetilde{\top} \Rightarrow Y_r\} \end{aligned} \quad (3)$$

for some $p, q, r \in \mathbb{N}$ and variables $Z_1, \dots, Z_p, W_1, \dots, W_q, Y_1, \dots, Y_r$ pairwise distinct and disjoint from A.

It can be checked that none of these contexts can separate u_A and v_A (see [50]):

▶ **Lemma 20.**

1. *For all* $\mathtt{C}[\ ] \in \mathbb{G}_1$, $\mathtt{C}[u_A] \simeq_{\beta\eta} \mathtt{C}[v_A]$.
2. *If* $\mathtt{D}[\ ] \in \mathbb{G}_2$, *then* $\mathtt{D}[u_A] \simeq_{\beta\eta} \mathtt{D}[v_A] \simeq_{\beta\eta} z_i\mathsf{Id}$.
3. *If* $\mathtt{E}[\ ] \in \mathbb{G}_3$, *then* $\mathtt{E}[u_A] \simeq_{\beta\eta} \mathtt{E}[v_A]$.
4. *If* $\mathtt{F}[\ ] \in \mathbb{G}_4$, *then* $\mathtt{F}[u_A] \simeq_{\beta\eta} \mathtt{F}[v_A] \simeq_{\beta\eta} w_i$.

$$
\begin{array}{lll}
\mathbb{G}_1: & \mathtt{C}[\,] ::= x_i \mid x'_i \mid \mathtt{E}[\,]Z_j\mathtt{C}[\,]\mathtt{C}[\,] & : (A^*\widetilde{+}\widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma,\Theta,\Delta,\Sigma} Z_j \\
\mathbb{G}_2: & \mathtt{D}[\,] ::= z_i(\Lambda W.\lambda w.\mathtt{F}[\,]) \mid \mathtt{E}[\,]Y_i\mathtt{D}[\,]\mathtt{D}[\,] & : (A^*\widetilde{+}\widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma,\Theta,\Delta,\Sigma} Y_i \\
\mathbb{G}_3: & \mathtt{E}[\,] ::= \mathbf{t} \mid \mathbf{f} \mid [\,](\Lambda Y.\lambda y.\lambda z.\mathtt{D}[\,]) & : (A^*\widetilde{+}\widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma,\Theta,\Delta,\Sigma} \mathsf{Bool} \\
\mathbb{G}_4: & \mathtt{F}[\,] ::= w \mid \mathtt{E}[\,]W_i\mathtt{F}[\,]\mathtt{F}[\,] & : (A^*\widetilde{+}\widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma,\Theta,\Delta,\Sigma} W_i
\end{array}
$$

Figure 3 Contexts $\mathbb{G}_1$-$\mathbb{G}_4$.

The key ingredient is a lemma stating that, when A is not inhabited in $\mathrm{F}^{\clubsuit}_{\mathrm{at}}$, the families of contexts $\mathbb{G}_1$-$\mathbb{G}_4$ can be used to generate all possible term contexts.

▶ **Lemma 21.** *Let* $\mathtt{K}[\,] : (A^*\widetilde{+}\widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{x_1\mapsto Z, x'_1\mapsto Z} Z$ *be a β-normal term context. If A is not inhabited in $\mathrm{F}^{\clubsuit}_{\mathrm{at}}$, then $\mathtt{K}[\,] \in \mathbb{G}_1$.*

Proof. We will prove the following claim: either there exists contexts $\Gamma, \Theta, \Delta, \Sigma$ as in Eq. (3), for some $p, q, r \in \mathbb{N}$ and variables $Z_1, \dots, Z_p, W_1, \dots, W_q, Y_1, \dots, Y_r$ pairwise distinct and disjoint from A, and a context $\mathtt{H}[\,] : (A^*\widetilde{+}\widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma,\Theta,\Delta,\Sigma} A^*$, or $\mathtt{K}[\,] \in \mathbb{G}_1$. If the main claim is true we can deduce the statement of the lemma as follows: suppose $\mathtt{K}[\,] \notin \mathbb{G}_1$; then let θ be the substitution sending all variables in $\Gamma, \Theta, \Delta, \Sigma$ plus Y onto $\clubsuit$ and being the identity on all other variables. Then $\mathtt{H}\theta[\,] : ((\clubsuit \Rightarrow A)\widetilde{+}\widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma\theta,\Theta\theta,\Delta\theta,\Sigma\theta} : \clubsuit \Rightarrow A$. Then we have $\Gamma\theta, \Theta\theta, \Delta\theta, \Sigma\theta \vdash t : A$, where $t = \mathtt{H}\theta[\lambda x.\mathbf{t}]\star$ and we can conclude that $\vdash t' : A$ holds, where t' is obtained from t by substituting the variables in Γ and Θ by $\star$ and those in Δ and Σ by $\lambda x.\star$.

Let us prove the main claim. Suppose by contradiction that for no $\Gamma, \Theta, \Delta, \Sigma$ there exists a context $\mathtt{H}[\,] : (A^*\widetilde{+}\widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma,\Theta,\Delta,\Sigma} A^*$. We will show by simultaneous induction the following claims:

1. for all $\Gamma, \Theta, \Delta, \Sigma$ as above, if $\mathtt{K}[\,] : (A^*\widetilde{+}\widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma,\Theta,\Delta,\Sigma} Z_i$, then $\mathtt{K}[\,] \in \mathbb{G}_1$;
2. for all $\Gamma, \Theta, \Delta, \Sigma$ as above, if $\mathtt{K}[\,] : (A^*\widetilde{+}\widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma,\Theta,\Delta,\Sigma} Y_i$, then $\mathtt{K}[\,] \in \mathbb{G}_2$;
3. for all $\Gamma, \Theta, \Delta, \Sigma$ as above, if $\mathtt{K}[\,] : (A^*\widetilde{+}\widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma,\Theta,\Delta,\Sigma} \mathsf{Bool}$ and $\mathtt{K}[\,]$ is an elimination context, then $\mathtt{K}[\,] \in \mathbb{G}_3$;
4. for all $\Gamma, \Theta, \Delta, \Sigma$ as above, if $\mathtt{K}[\,] : (A^*\widetilde{+}\widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma,\Theta,\Delta,\Sigma} W_i$, then $\mathtt{K}[\,] \in \mathbb{G}_4$.

The main claim then follows from 1. by taking $\Gamma = \{x \mapsto Z, x' \mapsto Z\}$ and $\Theta = \Delta = \Sigma = \varnothing$.

We argue for each case separately:

1. There exist two possibilities for $\mathtt{K}[\,]$:
   a. $\mathtt{K}[\,] = x_i$ (resp. $= x'_i$), hence $\mathtt{K}[\,] \in \mathbb{G}_1$;
   b. $\mathtt{K}[\,] = \mathtt{K}'[\,]Z\mathtt{K}_1[\,]\mathtt{K}_2[\,]$, where $\mathtt{K}'[\,] : (A^*\widetilde{+}\widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma,\Theta,\Delta,\Sigma} \mathsf{Bool}$ and $\mathtt{K}_i[\,] : (A^*\widetilde{+}\widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma,\Theta,\Delta,\Sigma} Z$, and where $\mathtt{K}'[\,]$ is an elimination context. By the induction hypothesis then $\mathtt{K}'[\,] \in \mathbb{G}_3, \mathtt{K}_i[\,] \in \mathbb{G}_1$, hence $\mathtt{K}[\,] \in \mathbb{G}_1$.
2. There exist three possibilities for $\mathtt{D}[\,]$:
   a. $\mathtt{K}[\,] = y_i\mathtt{K}'[\,]$, where $\mathtt{K}'[\,] : (A^*\widetilde{+}\widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma,\Theta,\Delta,\Sigma} A^*$, but this case is excluded by the hypothesis;
   b. $\mathtt{K}[\,] = z_i(\Lambda W.\lambda w.\mathtt{K}'[\,])$, where $\mathtt{K}'[\,] : (A^*\widetilde{+}\widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma,\Theta\cup\{w\mapsto W\},\Delta,\Sigma} W$ and where W does not occur in $\Gamma, \Theta, \Delta, \Sigma$. By the induction hypothesis then $\mathtt{K}'[\,] \in \mathbb{G}_4$, hence $\mathtt{K}[\,] \in \mathbb{G}_2$;
   c. $\mathtt{K}[\,] = \mathtt{K}'[\,]Y_i\mathtt{K}_1[\,]\mathtt{K}_2[\,]$, where $\mathtt{K}'[\,] : (A^*\widetilde{+}\widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma,\Theta,\Delta,\Sigma} \mathsf{Bool}$, $\mathtt{K}_i[\,] : (A^*\widetilde{+}\widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma,\Theta,\Delta,\Sigma} Y_i$, and $\mathtt{K}'[\,]$ is an elimination context. By the induction hypothesis this implies $\mathtt{K}'[\,] \in \mathbb{G}_3$ and $\mathtt{K}_i \in \mathbb{G}_2$, so we can conclude $\mathtt{K}[\,] \in \mathbb{G}_2$.

3. If $\mathtt{K}[\ ]$ is an elimination context, then it must be $\mathtt{K}[\ ] = x\mathtt{K}'[\ ]$, where $\mathtt{K}'[\ ] : (A^* \widetilde{+} \widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma \cup \{x_1 \mapsto Z', x_2 \mapsto Z''\}, \Theta, \Delta, \Sigma} A^* \widetilde{+} \widetilde{\top}$. Moreover, $\mathtt{K}'$ must be of the form $\Lambda Y.\lambda y.\lambda z.\mathtt{K}''[\ ]$, where $\mathtt{K}''[\ ] : (A^* \widetilde{+} \widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma \cup \{x_1 \mapsto Z', x_2 \mapsto Z''\}, \Theta, \Delta \cup \{y \mapsto A^* \Rightarrow Y\}, \Sigma \cup \{z \mapsto \widetilde{\top} \Rightarrow Y\}} Y$, and where Y is distinct from all variables in $\Gamma \cup \{x_1 \mapsto Z', x_2 \mapsto Z''\}, \Theta, \Delta, \Sigma$; then by the induction hypothesis we deduce $\mathtt{K}''[\ ] \in \mathbb{G}_2$, and thus $\mathtt{K}[\ ] \in \mathbb{G}_3$.
4. There are two possible cases:
   a. $\mathtt{K}[\ ] = w_i$, hence $\mathtt{K}[\ ] \in \mathbb{G}_4$;
   b. $\mathtt{K}[\ ] = \mathtt{K}'[\ ]W_i\mathtt{K}_1\mathtt{K}_2$, where $\mathtt{K}'[\ ] : (A^* \widetilde{+} \widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma, \Theta, \Delta, \Sigma} \mathsf{Bool}$, $\mathtt{K}_i[\ ] : (A^* \widetilde{+} \widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{\Gamma, \Theta, \Delta, \Sigma} W_i$ and $\mathtt{K}'[\ ]$ is an elimination context. By the induction hypothesis this implies $\mathtt{K}'[\ ] \in \mathbb{G}_3$ and $\mathtt{K}_i \in \mathbb{G}_4$, whence $\mathtt{K}[\ ] \in \mathbb{G}_4$. ◀

▶ **Proposition 22.** *$u_A \not\simeq^{\mathrm{F_{at}}}_{\mathsf{Bool}} v_A$ iff A is inhabited in $\mathrm{F}^{\clubsuit}_{\mathrm{at}}$.*

Proof. We only need to show one side of the statement: suppose A is not inhabited in $\mathrm{F}^{\clubsuit}_{\mathrm{at}}$. Any context $\mathtt{K}[\ ] : (A^* \widetilde{+} \widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash \mathsf{Bool}$ can be written, up to η-equivalence, as $\mathtt{K}[\ ] = \Lambda Z.\lambda x_1 x_2.\mathtt{K}'[\ ]$, with $\mathtt{K}'[\ ] : (A^* \widetilde{+} \widetilde{\top}) \Rightarrow \mathsf{Bool} \vdash^{x_1 \mapsto Z, x_2 \mapsto Z} \mathsf{Bool}$. As we can suppose $\mathtt{K}[\ ]$ to be β-normal, by Lemma 21, it must be $\mathtt{K}'[\ ] \in \mathbb{G}_1$. Hence, by Lemma 20 we deduce that $\mathtt{K}[u_A] \simeq_{\beta\eta} \mathtt{K}[v_A]$. ◀

▶ **Theorem 23.** *The congruences $\simeq^{\mathrm{F_{at}}}_{\mathsf{Bool}}$ and $\simeq^{\mathrm{F_{at}}}_{\mathsf{Nat}}$ are both undecidable.*

7 Conclusion

Related works. The literature on ML-polymorphism, both at theoretical and applicative level, is vast. Several extensions of ML to account for first-class polymorphism while retaining a decidable type-checking have been investigated, mostly following two directions: first, that of considering type systems with explicit type annotations (as the system PolyML [20]); second, that of encoding first-class polymorphism in a ML-style system by means of *coercions* (as in System Fc [60] or in ML^{F} [30]). In the last case, coherently with our discussion of FOU and SOU, the price to pay to remain decidable is that self-applications of λ-abstracted variables must come with explicit type annotations. This approach is currently followed in the design of the Haskell compiler, which supports first-class polymorphism.

Predicative restrictions of System F and their expressive power have been also largely investigated [32, 33, 6]. For example, the numerical functions representable in Leivant's finitely stratified polymorphism are precisely those at the third level of Grzegorczyk's hierarchy [33], and transfinitely stratified systems have been shown to represent all primitive recursive functions [6]. In [34] a system with expressive power comparable to System $\mathrm{F_{at}}$ is shown to characterize the polytime functions.

Research by Ferreira and her collaborators on System $\mathrm{F_{at}}$ has mostly focused on predicative translations of intuitionistic logic and their reduction properties [12, 11, 10]. As mentioned before, these translations rely on the observation that for certain types the unrestricted $\forall$E-rule is admissible in $\mathrm{F_{at}}$. The characterization of the class of types satisfying this property is an open problem (a partial characterization is described in [46]).

Another way to obtain interesting subsystems of System F is by restricting the class of types which can be universally quantified (instead of the admissible witnesses). For instance, the system in [2] forbids quantifier nestings, while the system in [35] only allows quantification $\forall X.A$ when X occurs at depth at most 2 in A (i.e. when X occurs at most twice to the left of an implication). Interestingly, both systems have the expressive power of Gödel's System T (which is not a first-order system).

Another kind of restrictions on the shape of types have been investigated by the authors in [49], motivated by ideas from the categorical semantics of polymorphism [3]. The two resulting fragments $\Lambda2^{\kappa\leqslant 0}, \Lambda2^{\kappa\leqslant 1}$ are equivalent, respectively, to the simply typed λ-calculus with finite sums and products, and to its extension with least and greatest fixpoints (in particular, (CE) is decidable in $\Lambda2^{\kappa\leqslant 0}$).

Finally, polymorphism in *linear* type systems has been investigated too. Interestingly, (TI) [28, 27] and (CE) [43] remain undecidable even in this case.

Future work. The main interest we found in investigating F_{at} was to shed some new light on the source of undecidability of type-related properties for full System F. Yet, one might well ask whether the decidability of type-checking makes F_{at} a reasonable candidate for implementations. Admittedly, our decision algorithm, which was only oriented to prove decidability, is not very practical: checking failure is coNP with respect to the number of type symbols. Yet, it does not seems unlikely that more optimized algorithms can be developed.

By the way, given that the terms typable in F_{at} are simply typable, would an implementation of atomic polymorphism be interesting at all? In contrast with ML, type-checking atomically polymorphic programs is decidable at any rank. One could thus investigate extensions of ML with first class atomic polymorphism (realistically, in presence of other type constructors like e.g. some restricted version of dependent types, see [65]).

A more interesting direction, suggested by our decision algorithm, would be to investigate systems with full, impredicative, polymorphism, but obeying some condition ensuring *acyclicity*, so that TC (based on SOU) remains decidable. One would thus retain some advantages of first-class polymorphism (e.g. the modularity/genericity of programs) while admitting self-applications only in "ML-style" (or with explicit type annotations, as in ML^{F} [30]). For instance, a way to ensure acycliclity might be to require that a polymorphic λ-abstracted variable be used in an *affine* way, i.e. at most once.

References

1 Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for free for free: parametricity, with and without types. In *Proceedings of the ACM on Programming Languages*, volume 1 of *ICFP*, page Article No. 39, New York, 2017.

2 Thorsten Altenkirch and Thierry Coquand. A finitary subsystem of the polymorphic λ-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications*, pages 22–28, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

3 E.S. Bainbridge, Peter J. Freyd, Andre Scedrov, and Philip J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70:35–64, 1990.

4 Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

5 Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda calculus with types.* Perspectives in Logic. Cambridge University Press, 2013.

6 Norman Danner and Daniel Leivant. Stratified polymorphism and primitive recursion. *Mathematical Structures in Computer Science*, 9(4):507–522, 1999.

7 Kosta Došen and Zoran Petrić. The typed Böhm theorem. *Electronic Notes in Theoretical Computer Science*, 50(2):117–129, 2001.

8 Andrej Dudenhefner and Jakob Rehof. A simpler undecidability proof for system F inhabitation. In *TYPES 2018*, pages 2:1–2:11, 2018.

9 José Espírito Santo and Gilda Ferreira. A refined interpretation of intuitionistic logic by means of atomic polymorphism. *Studia Logica*, 108(3):477–507, 2020. doi:10.1007/s11225-019-09858-1.

10 José Espírito Santo and Gilda Ferreira. The Russell-Prawitz embedding and the atomization of universal instantiation. *Logic Journal of the IGPL*, July 2020. jzaa025.

11 Fernando Ferreira and Gilda Ferreira. Commuting conversions vs. the standard conversions of the "good" connectives. *Studia Logica*, 92(1):63–84, 2009.

12 Fernando Ferreira and Gilda Ferreira. Atomic polymorphism. *Journal of Symbolic Logic*, 78(1):260–274, 2013.

13 Fernando Ferreira and Gilda Ferreira. The faithfulness of Fat: a proof-theoretic proof. *Studia Logica*, 103(6):1303–1311, 2015.

14 Gilda Ferreira. η-conversions of IPC implemented in atomic F. *Logic Journal of the IGPL*, 25(2):115–130, June 2016.

15 Gilda Ferreira and Bruno Dinis. Instantiation overflow. *Reports on Mathematical Logic*, 51:15–33, 2016.

16 Gilda Ferreira and Vasco T Vasconcelos. The computational content of atomic polymorphism. *Logic Journal of the IGPL*, 27(5):625–638, December 2018.

17 Steven Fortune, Daniel Leivant, and Michael O'Donnell. The expressiveness of simple and second-order type structures. *Journal of the ACM*, 30(1):151–185, 1983.

18 Ken-etsu Fujita and Aleksy Schubert. The undecidability of type related problems in the type-free style system F with finitely stratified polymorphic types. *Information and Computation*, 218:69–87, 2012.

19 Dov M. Gabbay. *Semantical Investigations in Heyting's Intuitionistic Logic*, volume 148. Springer Science + Business, Dordrecht, 1981.

20 Jacques Garrigue and Didier Rémy. Extending ml with semi-explicit higher-order polymorphism. In Martín Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software*, pages 20–46, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

21 Paola Giannini and Simona Ronchi Della Rocca. Characterization of typings in polymorphic type discipline. In *Proceedings of the 3-th Annual IEEE Symposium on Logic in Computer Science*, pages 61–70, Edinburgh, 1988.

22 Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13(2):225–230, 1981.

23 Ryu Hasegawa. Categorical data types in parametric polymorphism. *Mathematical Structures in Computer Science*, 4(1):71–109, 2009.

24 Fritz Henglein. *Polymorphic type inference and semi-unification*. PhD thesis, The State University of New Jersey, 1989.

25 Roger J. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1069.

26 Assaf J. Kfoury, Jerzy Tiuryn, and Paweł Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102(1):83–101, 1993.

27 Yves Lafont. The undecidability of second order linear logic without exponentials. *The Journal of Symbolic Logic*, 61(02):541–548, 1996. `doi:10.2307/2275674`.

28 Yves Lafont and Andre Scedrov. The Undecidability of Second Order Multiplicative Linear Logic. *Information and Computation*, 125(1):46–51, 1996. `doi:10.1006/inco.1996.0019`.

29 Joachim Lambek and Philip J. Scott. *Introduction to higher order categorical logic*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1988.

30 Didier Le Botlan and Didier Rémy. Mlf: Raising ml to the power of system f. In *Proc. of the International Conference on Functional Programming (ICFP '03)*, pages 27–38, 2003.

31 R. D. Lee. *Decidable classes of recursive equations*. PhD thesis, University of Leicester, 1969.

32 Daniel Leivant. Stratified polymorphism. In *LICS '89. Proceedings of the 4th Annual Symposium on Logic in Computer Science*. IEEE, 1989.

33 Daniel Leivant. Finitely stratified polymorphism. *Information and Computation*, 93(1):93–113, 1991.

34 Daniel Leivant. A foundational delineation of Poly-time. *Information and Computation*, 110(2):391–420, 1994.

35 Daniel Leivant. Peano's lambda calculus: the functional abstraction implicit in arithmetic. In *Logic, meaning and computation, Essays in Memory of Alonzo Church*, volume 305 of *Synthese Library, Studies in Epistemology, Logic, Methodology and Philosophy of Science*, pages 313–329. Springer Netherlands, 2001.

36 Jordi Levy. Decidable and undecidable second-order unification problems. In Tobias Nipkow, editor, *Rewriting Techniques and Applications*, pages 47–60, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

37 M. H. Löb. Embedding first order predicate logic in fragments of intuitionistic logic. *Journal of Symbolic Logic*, 41:705–718, 1976.

38 Jan Małolepszy, Małgorzata Moczurad, and Marek Zaionc. Schwichtenberg-style lambda definability is undecidable. In Philippe de Groote and J. Roger Hindley, editors, *Typed Lambda Calculi and Applications*, pages 267–283, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

39 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 '15, pages 3–16, New York, NY, USA, 2015. Association for Computing Machinery.

40 R. Milner and L. Damas. The principal type schemes for functional programs. In *Symposium on Principles of Programming Languages, ACM*, 1982.

41 Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17(3):248–375, 1978.

42 Robin Milner. The standard ml core language. *Polymorphism*, 2(2), 1985.

43 Le Than Dung Nguyen, Paolo Pistone, Thomas Seiller, and Lorenzo Tortora de Falco. Finite semantics of polymorphism, complexity and the expressive power of type fixpoints, 2019. URL: `https://hal.archives-ouvertes.fr/hal-01979009`.

44 Vincent Padovani. *Filtrage d'ordre supérieur*. PhD thesis, Université Paris 7, 1996.

45 Frank Pfenning. On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae*, 19(1-2):185–199, 1993.

46 Paolo Pistone. Proof nets and the instantiation overflow property, 2018. `arXiv:1803.09297`.

47 Paolo Pistone and Luca Tranchini. The naturality of natural deduction II. some remarks on atomic polymorphism, 2020. `arXiv:1908.11353`.

48 Paolo Pistone and Luca Tranchini. The Yoneda Reduction of Polymorphic Types (extended version), 2020. `arXiv:1907.03481`.

49 Paolo Pistone and Luca Tranchini. The Yoneda Reduction of Polymorphic Types. In Christel Baier and Jean Goubault-Larrecq, editors, *29th EACSL Annual Conference on Computer Science Logic (CSL 2021)*, volume 183 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:22, Dagstuhl, Germany, 2021. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

50 Paolo Pistone and Luca Tranchini. What's decidable about (atomic) polymorphism?, 2021. Available at `arXiv:2105.00748`.

51 Gordon Plotkin and Martin Abadi. A logic for parametric polymorphism. In *TLCA '93, International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375. Springer Berlin Heidelberg, 1993.

52 M Clarence Protin. Type inhabitation of atomic polymorphism is undecidable. *Journal of Logic and Computation*, January 2021. exaa090.

53 John C. Reynolds. Types, abstraction and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing '83*, pages 513–523. North-Holland, 1983.

54 Aleksy Schubert, Paweł Urzyczyn, and Konrad Zdanowski. On the Mints hierarchy in first-order intuitionistic logic. *Logical Methods in Computer Science*, 12(4:11):1–25, 2016.

55 Helmut Schwichtenberg. Definierbare funktionen im λ-kalkül mit typen. *Archiv für mathematische Logik und Grundlagenforschung*, 17(3):113–114, 1975.

56 S. K. Sobolev. The intuitionistic propositional calculus with quantifiers. *Mathematical Notes of the Academy of Sciences of the USSR*, 22(528-532), 1977.

57 Morten Heine Sorensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*, volume 149 of *Studies in logic and the foundations of mathematics*. Elsevier Science, 2006.

58 R. Statman. Completeness, invariance and λ-definability. *The Journal of Symbolic Logic*, 47(1):17–26, 1982.

59 Richard Statman. intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science*, 9(1):67–72, 1979.

60 Marin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In *TLDI '07 Proceedings of the 2007 ACM SIGPLAN Internatinal workshop in Types in languages design and implementation*, pages 53–66. ACM New York, 2007.

61 Luca Tranchini, Paolo Pistone, and Mattia Petrolo. The naturality of natural deduction. *Studia Logica*, `https://doi.org/10.1007/s11225-017-9772-6`, 2017.

62 J. Voigtländer. Proving correctness via free theorems: the caser of the destroy/build-rule. In *Proceedings of the ACM SIGPLAN symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 13–20, New York, 2008. ACM press.

63 Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on functinoal programming languages and computer architecture - FPCA '89*, 1989.

64 J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98:111–156, 1998.

65 Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 214–227, New York, NY, USA, 1999. Association for Computing Machinery.

A $\mathrm{F_{at}}$-unification

In this section we describe a decidable unification problem, that we call $\mathrm{F_{at}}$-unification, and we show that this problem captures type-checking for $\mathrm{F_{at}}$.

A decidable second-order unification problem. We consider a second-order language composed of three different sorts of variables: *sequence variables* $a, b, c, \dots$, *projection variables* $\alpha^n, \beta^n, \gamma^n, \dots$ and *second-order variables* $\mathsf{F}^n, \mathsf{G}^n, \dots$ (where in the last two cases n indicates the arity of the variable). The language includes expressions of three sorts, noted $\langle * \rangle$, $*$ and $T(*)$; the expressions of each type are defined by the grammars below:

$$\mathfrak{a}, \mathfrak{b}, \mathfrak{c} ::= \langle X_1 \dots X_n \rangle \mid a \mid \alpha^n a_1 \dots a_n \qquad (\text{sort } \langle * \rangle)$$

$$\phi, \psi ::= X \mid \pi^l(\mathfrak{a}) \mid \mathsf{F}^n \mathfrak{a}_1 \dots \mathfrak{a}_n \mid \Phi \Rightarrow \Psi \qquad (\text{sort } *)$$

$$\Phi, \Psi ::= \forall a.\phi \qquad (\text{sort } T(*))$$

A $\mathrm{F_{at}}$*-unification problem* is a pair (U, E), where U is a set of equations of the form $\phi = \psi$ between expressions of type $*$, and E is a set of *constraints* of the form $(\alpha : a)$ or $(a : k)$, where $k \in \mathbb{N}$.

Given a $\mathrm{F_{at}}$-unification problem (U, E), for all projection variable α^n occurring in U, let $\deg(\alpha)$ indicate the maximum l such that $\pi^l(\alpha^n a_1 \dots a_n)$ occurs in U.

A *substitution* for a $\mathrm{F_{at}}$-unification problem (U, E) is given by the following data:

- for each sequence variable a, a natural number $k_a^S \in \mathbb{N}$;
- for each projection variable α^n, a pair $(k_\alpha^S, S(\alpha))$ made of a natural number $k_\alpha^S \geqslant \deg(\alpha)$ and a sequence $S(\alpha) = \langle S(\alpha)_1, \dots, S(\alpha)_{k_\alpha^S} \rangle$, where $S(\alpha)_i$ is either of the form $\lambda x_1 . \dots . x_n . X$ or of the form $\lambda x_1 . \dots . x_n . \pi^l(x_j)$, where l is such that, whenever $\alpha^n a_1 \dots a_n$ occurs in U, $l \leqslant k_{a_j}^S$;

- for each second-order variable F^n, a function $S(\mathsf{F})$ of the form $\lambda\rho_1.\dots.\rho_n.A(\rho_1,\dots,\rho_n)$, where $A(\rho_1,\dots,\rho_n)$ is given by the grammar

$$A, B ::= X \mid \pi^l(\rho_i) \mid A \Rightarrow B \mid \forall X.A$$

with $i \in \{1,\dots,n\}$ and l being such that, if $\mathsf{F}^n\mathfrak{a}_1\dots\mathfrak{a}_n$ occurs in U, then $l \leqslant k^S_{\mathfrak{a}_i}$ (where $k^S_{\mathfrak{a}}$ is k if $\mathfrak{a} = \langle X_1,\dots,X_k\rangle$, is k^S_a if $\mathfrak{a} = a$, and is k^S_α if $\mathfrak{a} = \alpha^r a_1\dots a_r$).

Given a substitution S, we define (1) for any expression $\mathfrak{a}$ of sort $\langle *\rangle$, a sequence $S(\mathfrak{a})$ of type variables, (2) for any expression ϕ of sort $*$, a type $S(\phi)$, and (3) for any expression Φ of sort $T(*)$, a type $S(\Phi)$ as follows:

- if $\mathfrak{a} = a$, $S(a)$ is an arbitrary sequence of pairwise distinct variables $\langle S(a)_1,\dots,S(a)_{k_a}\rangle$ (chosen in such a way that if $a \neq b$, $S(a)$ and $S(b)$ are disjoint);
- if $\mathfrak{a} = \langle X_1,\dots,X_r\rangle$, then $S(\mathfrak{a}) = \langle X_1,\dots,X_r\rangle$;
- if $\mathfrak{a} = \alpha^n a_1\dots a_n$, then $S(\mathfrak{a}) = \langle U_1,\dots,U_{k^S_\alpha}\rangle$ where for all $i \leqslant k^S_\alpha$:
  - if $S(\alpha)_i = \lambda\vec{x}.X$, then $U_i = X$;
  - if $S(\alpha)_i = \lambda\vec{x}.\pi^l(x_j)$, then $U_i = S(a_j)_l$;
- if $\phi = X$, then $S(\phi) = X$;
- if $\phi = \pi^l(\mathfrak{a})$, then $S(\phi) = S(\mathfrak{a})_l$;
- if $\phi = \mathsf{F}\mathfrak{a}_1\dots\mathfrak{a}_n$, and $S(\mathsf{F}) = \lambda\vec{\rho}.A$, then $S(\phi) = A[\pi^l(\rho_i) \mapsto S(\mathfrak{a}_i)_l]$;
- if $\phi = \Phi \Rightarrow \Psi$, then $S(\phi) = S(\Phi) \Rightarrow S(\Psi)$;
- if $\Phi = \forall a.\phi$, then $S(\Phi) = \forall S(a).S(\phi)$.

A substitution S for (U, E) is a *unifier* of (U, E) if the following hold:

1. for any equation $\phi = \psi \in U$, $S(\phi) = S(\psi)$ holds;
2. for any constraint of the form $\alpha : a \in E$, $k^S_a = k^S_\alpha$;
3. for any constraint of the form $a : k \in E$, $k^S_a = k$.

We let `Fat-unification` indicate the problem of finding a unifier for a $\mathrm{F_{at}}$-unification problem. The rest of this subsection is devoted to establish the following:

▶ **Theorem 24.** `Fat-unification` *is decidable.*

A $\mathrm{F_{at}}$-unification problem (U, E) is in *normal form* if if contains no equation of the form $\Phi_1 \Rightarrow \Psi_1 = \Phi_2 \Rightarrow \Psi_2$. Any unification problem can be put in normal form by repeatedly applying the following simplification rule:

$$\frac{U + \{(\forall a_1.\phi_1) \to (\forall b_1.\psi_1) = (\forall a_2.\phi_2) \to (\forall b_2.\psi_2)\}}{\big(U + \{\phi_1 = \phi_2, \psi_1 = \psi_2\}\big)\big[a_2 \mapsto a_1, b_2 \mapsto b_1\big]}$$

Given a $\mathrm{F_{at}}$-unification problem in normal form (U, E), we say that an equation $\phi = \psi$ *can be deduced from* U if $\phi = \psi$ can be deduced from a finite set of equations in U by applying standard first-order equality rules. We say that two second-order variables F, G are *equivalent* (noted $\mathsf{F} \simeq \mathsf{G}$) if an equation of the form $\mathsf{F}\mathfrak{a}_1\dots\mathfrak{a}_n = \mathsf{G}\mathfrak{b}_1\dots\mathfrak{b}_n$ can be deduced from U; we say that F is *connected with* G (noted $\mathsf{F} \rightsquigarrow \mathsf{G}$) if an equation of the form $\mathsf{F}\mathfrak{a}_1\dots\mathfrak{a}_n = \Phi \Rightarrow \Psi$, where U occurs in $\Phi \Rightarrow \Psi$, can be deduced from U. We say that (U, E) has a *variable cycle* if there exist variables $\mathsf{F}_1,\dots,\mathsf{F}_k$ such that $\mathsf{F}_1 \overset{\simeq}{\rightsquigarrow} \mathsf{F}_2 \overset{\simeq}{\rightsquigarrow} \dots \overset{\simeq}{\rightsquigarrow} \mathsf{F}_n \overset{\simeq}{\rightsquigarrow} \mathsf{F}_1$ (where $\mathsf{F} \overset{\simeq}{\rightsquigarrow} \mathsf{G}$ means that F is connected with some variable equivalent to G).

▶ **Lemma 25.** *Let (U, E) be a unification problem in normal form. If (U, E) has a variable cycle, then it has no solution.*

Proof. To prove the lemma we show that any unification problem (U, E) yields a *first-order* unification problem U^* and that any unifier of (U, E) yields a unifier of U^*. For the translation, we fix a constant c, and we associate any second-order variable F with a first-order variable x_{F}; any expression is translated into a first order expression by:

$$\begin{aligned} \mathfrak{a}^* &= c \\ \mathsf{F}^n \mathfrak{a}_1 \dots \mathfrak{a}_n &= x_{\mathsf{F}} \\ (\Phi \Rightarrow \Psi)^* &= \Phi^* \Rightarrow \Psi^* \\ (\forall a.\phi)^* &= \phi^* \end{aligned}$$

We finally let $U^* = \{\phi^* = \psi^* \mid \phi = \psi \in U\}$. Observe that if $\mathsf{F} \simeq \mathsf{G}$ in U, then $x_{\mathsf{F}} = x_{\mathsf{G}}$ in U^*, and if $\mathsf{F} \rightsquigarrow \mathsf{G}$ in U, then U^* contains an equation of the form $x_{\mathsf{F}} = t \Rightarrow u$, where x_{G} occurs in $t \Rightarrow u$. Hence a variable cycle in (U, E) induces a variable cycle in U^*.

For any substitution S for (U, E), we define a first-order substitution S^* as follows: given $\lambda\vec{\rho}.A$ we define A^* by $X^* = c$, $(\pi^l(\rho_i))^* = c$, $(A \Rightarrow B)^* = A^* \Rightarrow B^*$ and $(\forall X.A)^* = A^*$. We let then $S^*(x_{\mathsf{F}}) = S(\mathsf{F})^*$.

One can easily check that if S is a unifier for (U, E), then S^* is a unifier of U^*. As a consequence, if (U, E) has a variable cycle, so does U^*, and by well-known facts about first-order unification, U^* has no unifier, and so neither (U, E) does. ◀

Let us call a unification problem (U, E) *simple* if it contains no expression of the form $\Phi \Rightarrow \Psi$. If (U, E) has no variable cycle, then it can be reduced to a simple unification problem by applying the following rules:

$$\frac{U + \{X = \Phi \Rightarrow \Psi\}}{\{X = Y\}} \qquad\qquad \frac{U + \{\pi^l(\mathfrak{a}) = \Phi \Rightarrow \Psi\}}{\{X = Y\}}$$

$$\frac{U + \{\mathsf{F}^n \mathfrak{a}_1^1 \dots \mathfrak{a}_n^1 = (\forall c_1.\phi_1) \Rightarrow (\forall d_1.\psi_1), \dots, \mathsf{F}^n \mathfrak{a}_1^r \dots \mathfrak{a}_n^r = (\forall c_r.\phi_r) \Rightarrow (\forall d_r.\psi_r)\}}{U\Big[\mathsf{F}^n \vec{\mathfrak{a}} \mapsto (\mathsf{F}_1^{n+1} \vec{\mathfrak{a}} c \Rightarrow \mathsf{F}_2^{n+1} \vec{\mathfrak{a}} d)\Big] + \left\{ \begin{matrix} \mathsf{F}_1^{n+1} \mathfrak{a}_1^1 \dots \mathfrak{a}_n^1 c_1 = \phi_1, \dots, \mathsf{F}_1^{n+1} \mathfrak{a}_1^r \dots \mathfrak{a}_n^r c_r = \phi_r \\ \mathsf{F}_2^{n+1} \mathfrak{a}_1^1 \dots \mathfrak{a}_n^1 d_1 = \psi_1, \dots, \mathsf{F}_2^{n+1} \mathfrak{a}_1^r \dots \mathfrak{a}_n^r d_r = \psi_r \end{matrix} \right\}}$$

Where in the first two rules Y is any type variable distinct from X, and in the last rule we suppose that U contains no equation of the form $\mathsf{F}^n \mathfrak{a}_1 \dots \mathfrak{a}_n = \Phi \Rightarrow \Psi$. Observe that, by acycliclity, F cannot occur in either ϕ_i or ψ_i. One can argue by induction on the well-founded preorder $\overset{\simeq}{\rightsquigarrow}$ that all terms of the form $\Phi \Rightarrow \Psi$ can be eliminated by applying a finite number of instances of the rules above.

The last step to ensure decidability is showing (1) that all solutions to a $\mathrm{F_{at}}$-unification problem (U, E) can be generated algorithmically and (2) that one can suppose that, if a solution exists at all, this can be found within a *finite* search-space (that is, one in which only projections $\pi^l(\mathfrak{a})$, with l less than some fixed value K depending on the size of (U, E), occur). Step (2) ensures that, if a solution is not found after a finite search, one can conclude that no solution exists at all. These are the two ingredients of the proof of the proposition below, which is shown in detail in [50].

▶ **Proposition 26.** *There is an algorithm that generates all unifiers of a simple unification problem, if there exists any, and returns* `failure` *otherwise.*

Type-checking $\mathrm{F_{at}}$ by second-order unification. A *type-checking problem* is a triple (Γ, t, A) where Γ is a term context, t is a λ-term with $FV(t) \subseteq \Gamma$ and A is a type. A $\mathrm{F_{at}}$-*solution* of a type-checking problem is a type derivation in $\mathrm{F_{at}}$ of $\Gamma \vdash t : A$. We wish to prove the following:

$$\frac{\Gamma(x) = A \qquad A \preceq B}{\Gamma \vdash x : \forall \vec{X} : B}\ \vec{X} \notin FV(\Gamma) \qquad \frac{\Gamma, x \mapsto A \vdash t : B}{\Gamma \vdash \lambda x.t : \forall \vec{X}.A \Rightarrow B}\ \vec{X} \notin FV(\Gamma)$$

$$\frac{\Gamma \vdash t : A \Rightarrow B \qquad \Gamma \vdash u : A \qquad B \preceq C}{\Gamma \vdash tu : \forall \vec{X}.C}\ \vec{X} \notin FV(\Gamma)$$

Figure 4 Synthetic typing rules for Curry-style $\mathrm{F_{at}}$.

▶ **Theorem 27.** *For any type-checking problem (Γ, t, A), there exists a $\mathrm{F_{at}}$-unification problem $\mathbf{V}(\Gamma, t, A)$ such that (Γ, t, A) has a solution in $\mathrm{F_{at}}$ iff $\mathbf{V}(\Gamma, t, A)$ has a unifier.*

The first step is to associate with each term t finite sets of sequence variables, projection variables and second-order variables as follows (we suppose that no variable occurs both free and bound in t, and that any bound variable is bound exactly once):

- with each variable x in t, we associate two sequence variables a_x, b_x, a projection variable α^1_x, and two second-order variables $\mathsf{F}^1_x, \mathsf{G}^1_x$;
- with each subterm of t of the form uv, we similarly associate two sequence variables a_{uv}, b_{uv}, a projection variable α^1_{uv} and two second-order variables $\mathsf{F}^2_{uv}, \mathsf{G}^1_{uv}$;
- with each subterm of t of the form $\lambda x.u$, we associate a sequence variable $b_{\lambda x.u}$, and a second order variable $\mathsf{G}^1_{\lambda x.t}$.

Given a set of equations U and a sequence variable a not occurring in U, we let Ua be the set of equations obtained by replacing all terms $\alpha^n a_1 \dots a_n$ by $\alpha^{n+1} a_1 \dots a_n a$ and all term $\mathsf{F}^n \mathfrak{a}_1 \dots \mathfrak{a}_n$ by $\mathsf{F}^{n+1} \mathfrak{a}_1 \dots \mathfrak{a}_n a$.

We define a set of equations $\mathbf{U}(t)$, by induction on t as follows:

- $\mathbf{U}(x)$ is formed by the equation

$$\mathsf{F}_x(\alpha_x b_x) = \mathsf{G}_x b_x$$

- $\mathbf{U}(\lambda x.t)$ is formed by $\mathbf{U}(t) b_{\lambda x.t}$ plus the equations

$$\mathsf{G}_{\lambda x.t} b_{\lambda x.t} = (\forall a_x.\mathsf{F}_x a_x \vec{b} b_{\lambda x.t}) \Rightarrow \forall b_t.\mathsf{G}_t b_t b_{\lambda x.t}$$

- $\mathbf{U}(tu)$ is formed by $\mathbf{U}(t) b_{tu}, \mathbf{U}(u) b_{tu}$ plus the equations:

$$\mathsf{G}_t b_t b_{tu} = (\forall b_u.\mathsf{G}_u b_u b_{tu}) \Rightarrow (\forall a_{tu}.\mathsf{F}_{tu} a_{tu} b_{tu})$$
$$\mathsf{F}_{tu}(\alpha_{tu} b_{tu}) b_{tu} = \mathsf{G}_{tu} b_{tu}$$

We let $\mathbf{V}(\Gamma, t, A) = (\mathbf{U}(\Gamma, t, A), \mathbf{E}(\Gamma, t, A))$, where $\mathbf{U}(\Gamma, t, A)$ is the union of $\mathbf{U}(t)$ and all equations $\forall a_x.\mathsf{F}_x a_x = \Gamma(x)$ and $\forall b_t.\mathsf{G}_t b_t = A$. $\mathbf{E}(\Gamma, t, A)$ is formed by all constraints of the form $(\alpha_x : a_x)$ and $(\alpha_{tu} : b_t)$, as well as all constraints of the form $(a_x : k)$, where $\Gamma(x) = \forall X_1 \dots X_k.C$, all constraints of the form $(b_u : 0)$ where t contains a subterm of the form uv, and the constraint (b_t, h), where $A = \forall X_1 \dots X_h.A'$.

To show that solving $\mathbf{V}(\Gamma, t, A)$ is equivalent to checking if $\Gamma \vdash t : A$, as in [21], we first define synthetic typing rules for Curry-style $\mathrm{F_{at}}$ as shown in Fig. 4, where $A \preceq B$ holds when $A = \forall X_1 \dots X_n.A$ and $B = A[X_1 \mapsto Y_1, \dots, X_n \mapsto Y_n]$.

One can check by induction on t that a synthetic type derivation of $\Gamma \vdash t : A$ yields a unifier of $\mathbf{V}(\Gamma, t, A)$. Conversely, we show that from a unifier S for $\mathbf{V}(\Gamma, t, A)$ we can construct a synthetic typing derivation of $\Gamma \vdash t : A$. We argue by induction on t:

- if $t = x$, then we have $\Gamma(x) = \forall X_1 \dots X_N.S(\mathsf{F}_x)\vec{X}$, where $N = k^S_{a_x}$, $A = \forall Y_1 \dots Y_P.S(\mathsf{G}_x)\vec{Y}$, where $P = k^S_{b_x}$, and moreover, $S(\mathsf{F}_x)(S(\alpha_x)_1\vec{Y})\dots(S(\alpha_x)_N\vec{Y}) = S(\mathsf{G}_x)\vec{Y}$ (using the fact that $k^S_{\alpha_x} = k^S_{a_x} = N$). Observe that $(S(\alpha_x)_j\vec{Y})$ is a variable, and we deduce then that $\Gamma(x) \preceq S(\mathsf{G}_x)\vec{Y}$; since we can suppose that $\vec{Y}$ does not occur in Γ, we deduce then that

$$\dfrac{\Gamma(x) = \forall\vec{X}.S(\mathsf{F}_x)\vec{X} \qquad \forall\vec{X}.S(\mathsf{F}_x)\vec{X} \preceq S(\mathsf{G}_x)\vec{Y}}{\Gamma \vdash x : A}\ \vec{Y} \notin FV(\Gamma)$$

- if $t = \lambda x.u$, then we have that $A = \forall X_1 \dots X_N.A_1 \Rightarrow A_2$, where $A_1 = \forall Y_1 \dots Y_P.S(\mathsf{F}_x)\vec{Y}\vec{X}$ and $A_2 = \forall Z_1 \dots Z_Q.S(\mathsf{G}_u)\vec{Z}\vec{X}$, $N = k^S_{b_{\lambda x.u}}$, $P = k^S_{a_x}$, $Q = k^S_{b_u}$ and where we can suppose that the X_i do not occur free in Γ; since $\mathsf{U}(t) = \mathsf{U}(u)b_{\lambda x.t}$ we deduce that S unifies $\mathbf{V}(\Gamma \cup \{x : A_1\}, u, A_2))$. By I.H. we deduce then the existence of a type derivation of $\Gamma, x : A_1 \vdash u : A_2$, and since the X_i do not occur in Γ we finally have

$$\dfrac{\overset{[\text{I.H.}]}{\Gamma, x : A_1 \vdash u : A_2}}{\Gamma \vdash t : A}\ \vec{X} \notin FV(\Gamma)$$

- if $t = uv$, then we have that $A = \forall X_1 \dots X_N.S(\mathsf{G}_{uv})\vec{X}$, $S(\mathsf{G}_u)\vec{X} = (\forall Y_1 \dots Y_P.S(\mathsf{G}_v)\vec{Y}\vec{X}) \Rightarrow (\forall Z_1 \dots Z_Q.S(\mathsf{F}_{uv})\vec{Z}\vec{X})$ and that $S(\mathsf{F}_{uv})(S(\alpha_{uv})_1\vec{X})\dots(S(\alpha_{uv})_N\vec{X})\vec{X} = S(\mathsf{G}_{uv})\vec{X}$, where $N = k^S_{b_{uv}}$, $P = k^S_{b_u}$ and $Q = k^S_{a_{uv}}$, and where we use the fact that $k^S_{b_u} = 0$. Moreover, for any choice of the variables $\vec{X}$, we have that S unifies $\mathbf{V}(\Gamma, u, (\forall Y_1 \dots Y_P.S(\mathsf{G}_v)\vec{Y}\vec{X}) \Rightarrow \forall Z_1 \dots Z_Q.S(\mathsf{F}_{uv})\vec{Z}\vec{X})$ and $\mathbf{V}(\Gamma, v, \forall Y_1 \dots Y_P.S(\mathsf{G}_u)\vec{Y}\vec{X})$; by choosing the $\vec{X}$ so that they do not occur free in Γ, using the I.H. and the fact that $k^S_{\alpha_{uv}} = k^S_{a_{uv}} = Q$, we deduce then

$$\dfrac{\overset{[\text{I.H.}]}{\Gamma \vdash u : (\forall Y_1 \dots Y_P.S(\mathsf{G})_v\vec{Y}\vec{X}) \Rightarrow (\forall Z_1 \dots Z_Q.S(\mathsf{F}_{uv})\vec{Z}\vec{X})} \qquad \overset{[\text{I.H.}]}{\Gamma \vdash v : \forall Y_1 \dots Y_P.S(\mathsf{G}_v)\vec{Y}\vec{X}} \qquad \forall Z_1 \dots Z_Q.S(\mathsf{F}_{uv})\vec{Z}\vec{X} \preceq S(\mathsf{G}_{uv})\vec{X}}{\Gamma \vdash t : A}\ \vec{X} \notin FV(\Gamma)$$

Coalgebra Encoding for Efficient Minimization

Hans-Peter Deifel
Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

Stefan Milius
Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

Thorsten Wißmann
Radboud University Nijmegen, The Netherlands

Abstract

Recently, we have developed an efficient generic partition refinement algorithm, which computes behavioural equivalence on a state-based system given as an encoded coalgebra, and implemented it in the tool CoPaR. Here we extend this to a fully fledged minimization algorithm and tool by integrating two new aspects: (1) the computation of the transition structure on the minimized state set, and (2) the computation of the reachable part of the given system. In our generic coalgebraic setting these two aspects turn out to be surprisingly non-trivial requiring us to extend the previous theory. In particular, we identify a sufficient condition on encodings of coalgebras, and we show how to augment the existing interface, which encapsulates computations that are specific for the coalgebraic type functor, to make the above extensions possible. Both extensions have linear run time.

2012 ACM Subject Classification Theory of computation → Models of computation; Theory of computation → Logic and verification

Keywords and phrases Coalgebra, Partition refinement, Transition systems, Minimization

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.28

Related Version *Full Version*: `https://arxiv.org/abs/2102.12842`

Funding *Hans-Peter Deifel*: Supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Research and Training Group 2475 “Cybercrime and Forensic Computing” (393541319/GRK2475/1-2019).
Stefan Milius: Supported by Deutsche Forschungsgemeinschaft (DFG) under project MI 717/5-2.
Thorsten Wißmann: Supported by NWO TOP project 612.001.852.

Acknowledgements We would like to thank the anonymous referees for their comments, which helped us to improve the presentation.

1 Introduction

The task of minimizing a given finite state-based system has arisen in different contexts throughout computer science and for various types of systems, such as standard deterministic automata, tree automata, transition systems, Markov chains, probabilistic or other weighted systems. In addition to the obvious goal of reducing the mere memory consumption of the state space, minimization often appears as a subtask of a more complex problem. For instance, probabilistic model checkers benefit from minimizing the input system before performing the actual model checking algorithm, as e.g. demonstrated in benchmarking by Katoen et al. [32].

Another example is the graph isomorphism problem. A considerable portion of input instances can already be decided correctly by performing a step called colour refinement [9], which amounts to the minimization of a weighted transition system wrt. weighted bisimilarity.

Minimization algorithms typically perform two steps: first a reachable subset of the state set of the given system is computed by a standard graph search, and second, in the resulting reachable system all behaviourally equivalent states are identified. For the latter step one uses *partition refinement* or *lumping* algorithms that start by identifying all states and then iteratively refine the resulting partition of the state set by looking one step into the transition structure of the given system. There has been a lot of research on efficient partition refinement procedures, and the most efficient algorithms for various concrete system types have a run time in $\mathcal{O}(m \log n)$, for a system with n states and m transitions, e.g. Hopcroft's algorithm for deterministic automata [30] and the algorithm by Paige and Tarjan [36] for transition systems, even if the number of action labels is not fixed [43]. Partition refinement of probabilistic systems also underwent a dynamic development [18, 52], and the best algorithms for Markov chain lumping now match the complexity of the relational Paige-Tarjan algorithm [22, 31, 44]. For the minimization of more complex system types such as Segala systems [6, 26] (combining probabilities and non-determinism) or weighted tree automata [29], partition refinement algorithms with a similar quasilinear run time have been designed over the years.

Recently, we have developed a generic partition refinement algorithm [23, 48] and implemented it in the tool CoPaR [19, 51]. This generic algorithm computes the partition of the state set modulo behavioural equivalence for a wide variety of stated-based system types, including all the above. This genericity in the system type is achieved by working with *coalgebras* for a functor which encapsulates the specific types of transitions of the input system. More precisely, the algorithm takes as input a syntactic description of a set functor and an *encoding* of a coalgebra for that functor and then computes the simple quotient, i.e. the quotient of the state set modulo behavioural equivalence. The algorithm works correctly for every zippable set functor (Definition 2.8). It matches, and in some cases even improves on, the run-time complexity of the best known partition refinement algorithms for many concrete system types [51, Table 1].

The reasons why this run-time complexity can be stated and proven generically are: first, the encoding allows us to talk about the number of states and, in particular, the number of transitions of an input coalgebra. But more importantly, every iterative step of partition refinement requires only very few system-type specific computations. These computations are encapsulated in the *refinement interface* [48], which is then used by the generic algorithm.

An important feature of our coalgebraic algorithm is its modularity: in the tool the user can freely combine functors with already implemented refinement interfaces by products, coproducts and functor composition. A refinement interface for the combined functor is then automatically derived. In this way more structured systems types such as (simple and general) Segala systems and weighted tree automata can be handled.

In the present paper, we extend our algorithm to a fully fledged minimizer. In previous work [3] it has been shown that for set functors preserving intersections, every coalgebra equipped with a point, modelling initial states, has a minimization called the *well-pointed modification*. Well-pointedness means that the coalgebra does not have any proper quotients (i.e. it is *simple*) nor proper pointed subcoalgebras (i.e. it is *reachable*), in analogy to minimal deterministic automata being reachable and observable (see e.g. [5, p. 256]). The well-pointed modification is obtained by taking the reachable part of the simple quotient of a given pointed coalgebra [3] (and the more usual reversed order, simple quotient of the reachable part, is correct for functors preserving inverse images [50, Sec. 7.2]). Our previous work on coalgebraic minimization algorithms has focused on computing the simple quotient. Here we extend our algorithm by two missing aspects of minimization and provide their correctness proofs: the computation of (1) the transition structure of the minimized system, and (2) the reachable states of an input coalgebra.

One may wonder why (1) is a step worth mentioning at all because for many concrete system types this is trivial, e.g. for deterministic automata where the transitions between equivalence classes are simply defined by choosing representatives and copying their transitions from the input automaton. However, for other system types this step is not that obvious, e.g. for weighted automata where transition weights need to be summed up and transitions might actually disappear in the minimized system because weights cancel out. We found that in the generic coalgebraic setting enabling the computation of the (encoding of) the transition structure of the minimized coalgebra is surprisingly non-trivial, requiring us to extend the theory behind our algorithm.

In order to be able to perform this computation generically we work with *uniform encodings*, which are encodings that satisfy a coherence property (Definition 3.10). We prove that all encodings used in our previous work are uniform, and that the constructions enabling modularity of our algorithm preserve uniformity (Prop. 3.12). We also prove that uniform encodings are subnatural transformations, but the converse does not hold in general. In addition, we introduce the *minimization interface* containing the new function `merge` (to be implemented together with the refinement interface for each new system type) which takes care of transitions that change as a result of minimization. We provide `merge` operations for all functors with explicitly implemented refinement interfaces (Example 4.4), and show that for combined system types minimization interfaces can be automatically derived (Prop. 4.11); similarly as for refinement interfaces. Our main result is that the (encoded) transition structure of the minimized coalgebra can be correctly computed in linear time (Thm. 4.9).

Concerning extension (2), the computation of reachable states, it is well-known that every pointed coalgebra has a reachable part (being the smallest subcoalgebra) [3, 49]. Moreover, for a set functor preserving intersections it coincides with the reachable part of the canonical graph of the coalgebra [3, Lem. 3.16]. Recently, it was shown that the reachable part of a pointed coalgebra can be constructed iteratively [49, Thm. 5.20] and that this corresponds to performing a standard breadth-first search on the canonical graph. The missing ingredient to turn our previous partition refinement algorithm into a minimizer is to relate the canonical graph with the encoding of the input coalgebra. We prove that for a functor with a subnatural encoding, the encoding (considered as a graph) of every coalgebra coincides with its canonical graph (Theorem 5.6).

Putting everything together, we obtain an algorithm that computes the well-pointed modification of a given pointed coalgebra. Both additions can be implemented with linear run time in the size of the input coalgebra and hence do not add to the run-time complexity of the previous partition refinement algorithm. We have provided such an implementation with the new version of our tool CoPaR.

All proofs and additional details can be found in the full version [21].

Reachability in Coalgebraic Minimization. There are several works on coalgebraic minimization, ranging from abstract constructions to concrete and implemented algorithms [1, 34, 35, 48, 51], that compute the simple quotient [27] of a given coalgebra. These are not concerned with reachability since coalgebras are not equipped with initial states in general.

In Brzozowski's automata minimization algorithm [16], reachability is one of the main ingredients. This is due to the duality of reachability and observability described by Arbib and Manes [4], and this duality is used twice in the algorithm. Consequently, reachability also appears as a subtask in the categorical generalizations of Brzozowski's algorithm [10, 14, 15, 35, 38]. These generalizations concern automata processing input words and so do not cover minimization of (weighted) tree automata. Segala systems are not treated either. Due to the dualization, Brzozowki's classical algorithm for deterministic automata has doubly exponential time complexity in the worst case (although it performs well on certain types of non-deterministic automata, compared to determinization followed by minimization [41]).

2 Background

Our algorithmic framework [48] is defined on the level of coalgebras for set functors, following the paradigm of *universal coalgebra* [39]. Coalgebras can model a wide variety of systems.

In the following we recall standard notation for sets and functions as well as basic notions from the theory of coalgebras. We fix a singleton set $1 = \{*\}$; for each set X, we have a unique map $!\colon X \to 1$. We denote the disjoint union (coproduct) of sets A, B by $A + B$ and use $\mathsf{inl}, \mathsf{inr}$ for the canonical injections into the coproduct, as well as $\mathsf{pr}_1, \mathsf{pr}_2$ for the projections out of the product. We use the notation $\langle\cdots\rangle$, respectively $[\cdots]$, for the unique map induced by the universal property of a product, respectively coproduct. We also fix two sets $2 = \{0, 1\}$ and $3 = \{0, 1, 2\}$ and use the former as a set of boolean values with 0 and 1 denoting *false* and *true*, respectively. For each subset S of a set X, the *characteristic function* $\chi_S\colon X \to 2$ assigns 1 to elements of S and 0 to elements of $X \setminus S$. We denote by Set the category of all sets and maps. We shall indicate injective and surjective maps by $\rightarrowtail$ and $\twoheadrightarrow$, respectively.

Recall that an endofunctor $F\colon \mathsf{Set} \to \mathsf{Set}$ assigns to each set X a set FX, and to each map $f\colon X \to Y$ a map $Ff\colon FX \to FY$, preserving identities and composition, that is we have $F\,\mathsf{id}_X = \mathsf{id}_{FX}$ and $F(g \cdot f) = Fg \cdot Ff$. We denote the composition of maps by $\cdot$ written infix, as usual. An *F-coalgebra* is a pair (X, c) that consists of a set X of *states* and a map $c\colon X \to FX$ called *(transition) structure*. A *morphism* $h\colon (X, c) \to (Y, d)$ of F-coalgebras is a map $h\colon X \to Y$ preserving the transition structure, i.e. $Fh \cdot c = d \cdot h$. Two states $x, y \in X$ of a coalgebra (X, c) are *behaviourally equivalent* if there exists a coalgebra morphism h with $h(x) = h(y)$.

▶ **Example 2.1.** Coalgebras and the generic notion for behavioural equivalence instantiate to a variety of well-known system types and their equivalences:

1. The *finite powerset functor* $\mathcal{P}_{\mathsf{f}}$ maps a set to the set of all its finite subsets and functions $f\colon X \to Y$ to $\mathcal{P}_{\mathsf{f}}f = f[-]\colon \mathcal{P}_{\mathsf{f}}X \to \mathcal{P}_{\mathsf{f}}Y$ taking direct images. Its coalgebras are finitely branching (unlabelled) transition systems and coalgebraic behavioural equivalence coincides with Milner and Park's (strong) bisimilarity.
2. Given a commutative monoid $(M, +, 0)$, the *monoid-valued functor* $M^{(-)}$ maps a set X to the set of finitely supported functions from X to M. These are the maps $f\colon X \to M$, such that $f(x) = 0$ for all except finitely many $x \in X$. Given a map $h\colon X \to Y$ and a finitely supported function $f\colon X \to M$, $M^{(h)}(f)\colon M^{(X)} \to M^{(Y)}$ is defined as $M^{(h)}(f)(y) = \sum_{x \in X, h(x) = y} f(x)$. Coalgebras for $M^{(-)}$ correspond to finitely branching weighted transition systems with weights from M. If a coalgebra morphism $h\colon (X, c) \to (Y, d)$ merges two states s_1, s_2, then for all transitions $x \xrightarrow{m_1} s_1$, $x \xrightarrow{m_2} s_2$ in (X, c) there must be a transition $h(x) \xrightarrow{m_1 + m_2} h(s_1) = h(s_2)$ in (Y, d) and similarly if more than two states are merged. Coalgebraic behavioural equivalence captures weighted bisimilarity [33, Prop. 2].
   Note that the monoid may have inverses: if $s_2 = -s_1$, then the transitions in the above example cancel each other out, leading to a transition $h(x) \xrightarrow{0} h(s_1)$ with weight 0, which in fact represents the absence of a transition. This happens for example for the monoid $(\mathbb{R}, +, 0)$ of real numbers. A simple minimization algorithm for real weighted transition

(i.e. $\mathbb{R}^{(-)}$-coalgebras) systems is given by Valmari and Franceschinis [44]. These systems subsume Markov chains which are precisely the coalgebras for the finite probability distribution functor $\mathcal{D}$, a subfunctor of $\mathbb{R}^{(-)}$.

3. Given a signature Σ consisting of operation symbols σ, each with a prescribed natural number, its *arity* $\mathsf{ar}(\sigma)$, the *polynomial functor* F_Σ sends each set X to the set of (shallow) terms over X, specifically to the set

$$\{\sigma(x_1,\dots,x_n) \mid \sigma \in \Sigma, \mathsf{ar}(\sigma) = n, (x_1,\dots,x_n) \in X^n\}.$$

The action of F on a function $f\colon X \to Y$ is given by

$$F_\Sigma f(\sigma(x_1,\dots,x_n)) = \sigma(f(x_1),\dots,f(x_n)).$$

A coalgebra structure $c\colon X \to F_\Sigma X$ assigns to a state $x \in X$ an expression $\sigma(x_1,\dots,x_n)$, where σ is an output symbol and x_1 to x_n are the successor states. Two states are behaviourally equivalent if their tree-unfoldings, obtained by repeatedly applying the coalgebra structure c, yields the same (infinite) Σ-tree.

4. For a fixed alphabet A, the functor given by $FX = 2 \times X^A$ is a special case of a polynomial functor over a signature with two symbols of arity $|A|$. An F-coalgebra $c\colon X \to 2 \times X^A$ is the same as a deterministic automaton without an initial state: the structure c assigns a pair (b,t) to each $x \in X$, where the boolean value $b \in 2$ determines its finality, and the function $t\colon A \to X$ assigns to each input letter from $a \in A$ the successor state of x under a. Here, behavioural equivalence coincides with language equivalence in the usual automata theoretic sense.

5. The *bag functor* $\mathcal{B}$ sends a set X to the set of finite multisets over X and functions $f\colon X \to Y$ to $\mathcal{B}f\colon \mathcal{B}X \to \mathcal{B}Y$ given by $\mathcal{B}f(\{\!\!\{ x_1,\dots,x_2 \}\!\!\}) = \{\!\!\{ f(x_1),\dots,f(x_2) \}\!\!\}$, where we use the multiset braces $\{\!\!\{$ and $\}\!\!\}$ to differentiate from standard set notation; in particular $\{\!\!\{ x,x \}\!\!\} \neq \{\!\!\{ x \}\!\!\}$. Coalgebras for $\mathcal{B}$ are finitely branching transition systems where multiple transitions between any two states are allowed, or equivalently, weighted transition systems with positive integers as weights. This follows from the fact that the bag functor is (naturally isomorphic to) the monoid-valued functor for the monoid $(\mathbb{N},+,0)$. Hence, behavioural equivalence coincides with weighted bisimilarity again.
Note that every undirected graph may be considered as a $\mathcal{B}$-coalgebra by turning every edge into two directed edges with weight 1. Then two states are behaviourally equivalent iff they are identified by *colour refinement*, also called the 1-dimensional Weisfeiler-Lehman algorithm (see e.g. [9, 17, 46]).

▶ **Example 2.2** (Modularity). New system types can be constructed from existing ones by functor composition. For example, labelled transition systems (LTSs) are coalgebras for the functor $FX = \mathcal{P}_\mathsf{f}(A \times X)$, which is the composite of $\mathcal{P}_\mathsf{f}$ and $A \times -$ for a label alphabet A, and precisely the bisimilar states in an F-coalgebra are behaviourally equivalent. Composing further, *Segala systems* (or *probabilistic LTSs* [26]) are coalgebras for $FX = \mathcal{P}_\mathsf{f}(A \times \mathcal{D}X)$, for which coalgebraic behavioural equivalence instantiates to probabilistic bisimilarity [7]. Another example are *weighted tree automata* [29] with weights in a commutative monoid M and input signature Σ; they are coalgebras for the composed functor $FX = M^{(\Sigma X)}$, for which behavioural equivalence coincides with *backwards bisimilarity* [20].

Simple, Reachable, and Well-Pointed Coalgebras. Minimizing a given pointed coalgebra means to compute its well-pointed modification. We now briefly recall the corresponding coalgebraic concepts. For a more detailed and well-motivated discussion with examples, see e.g. [2, Sec. 9].

First, a *quotient coalgebra* of an F-coalgebra (X, c) is represented by a surjective F-coalgebra morphism, for which we write $q\colon (X, c) \twoheadrightarrow (Y, d)$, and a *subcoalgebra* of (X, c) is represented by an injective F-coalgebra morphism $m\colon (S, s) \rightarrowtail (X, c)$.

A coalgebra (X, c) is called *simple* if it does not have any proper quotient coalgebras [27]. That is, every quotient $q\colon (X, c) \twoheadrightarrow (Y, d)$ is an isomorphism. Equivalently, distinct states $x, y \in X$ are never behaviourally equivalent. Every coalgebra has an (up to isomorphism) unique simple quotient (see e.g. [2, Prop. 9.1.5]).

▶ **Example 2.3.**
1. A deterministic automaton regarded as a coalgebra for $FX = 2 \times X^A$ is simple iff it is observable [5, p. 256], that is, no distinct states accept the same formal language.
2. A finitely branching transition system considered as a $\mathcal{P}_\mathsf{f}$-coalgebra is simple, if it has no pairs of strongly bisimilar but distinct states; in other words if two states x, y are strongly bisimilar, then $x = y$.
3. A similar characterization holds for monoid-valued functors (such as the bag functor) wrt. weighted bisimilarity.

A *pointed* coalgebra is a coalgebra (X, c) equipped with a point $i\colon 1 \to X$, equivalently a distinguished element $i \in X$, modelling an initial state. Morphisms of pointed coalgebras are the point-preserving coalgebra morphisms, i.e. morphisms $h\colon (X, c, i) \to (Y, d, j)$ satisfying $h \cdot i = j$. Quotients and subcoalgebras of pointed coalgebras are defined wrt. these morphisms. A pointed coalgebra (X, c, i) is called *reachable* if it has no proper subcoalgebra, that is, every subcoalgebra $m\colon (S, s, j) \rightarrowtail (X, c, i)$ is an isomorphism. Every pointed coalgebra has a unique reachable subcoalgebra (see e.g. [2, Prop. 9.2.6]). The notion of reachable coalgebras corresponds well with graph theoretic reachability in concrete examples. We elaborate on this a bit more in Section 5.

▶ **Example 2.4.**
1. A deterministic automaton considered as a pointed coalgebra for $FX = 2 \times X^A$ (with the point given by the initial state) is reachable if all of its states are reachable from the initial state.
2. A pointed $\mathcal{P}_\mathsf{f}$-coalgebra is a finitely branching directed graph with a root node. It is reachable precisely when every node is reachable from the root node.
3. Similarly, for monoid-valued functors such as the bag functor, reachability is precisely graph theoretic reachability, where a transition weight of 0 means "no edge".

Finally, a pointed coalgebra (X, c, i) is *well-pointed* if it is reachable and simple. Every pointed coalgebra has a *well-pointed modification*, which is obtained by taking the reachable part of its simple quotient (see [2, Not. 9.3.4]).

▶ Remark 2.5. For a functor preserving inverse images, one may reverse the two constructions: the well-pointed modification is the simple quotient of the reachable part of a given pointed coalgebra [50, Sec. 7.2]. This is the usual order in which minimization of systems is performed algorithmically. However, for a functor that does not preserve inverse images, quotients of reachable coalgebras need not be reachable again [50, Ex. 5.3.27], possibly rendering the usual order incorrect.

Our present paper is concerned with the *minimization problem* for coalgebras, i.e. the problem to compute the well-pointed modification of a given pointed coalgebra in terms of its encoding.

▶ Remark 2.6. Recall that a (sub)natural transformation σ from a functor F to a functor G is a set-indexed family of maps $\sigma_X\colon FX \to GX$ such that for every (injective) function $m\colon X \to Y$ the square below commutes; we also say that σ is *(sub)natural in X*.

From previous results (see [48, Prop. 2.13] and [49, Thm. 4.6]) one obtains the following sufficient condition for reductions of reachability and simplicity. Given a family of maps $\sigma_X : FX \to GX$, then every F-coalgebra (X, c) yields a G-coalgebra $(X, \sigma_X \cdot c)$ and we can reduce minimization tasks from F-coalgebras to G-coalgebras as follows:

1. Suppose that $\sigma : F \to G$ is *sub-cartesian*, that is the squares below are pullbacks for every injective map $m : X \rightarrowtail Y$. Then the reachable part of a pointed F-coalgebra (X, c, i) is obtained from the reachable part of the G-coalgebra $(X, \sigma_X \cdot c, i)$.

$$\begin{tikzcd} FX \arrow[r, "\sigma_X"] \arrow[d, "Fm"'] & GX \arrow[d, "Gm"] \\ FY \arrow[r, "\sigma_Y"] & GY \end{tikzcd}$$

2. Suppose that F is a subfunctor of G, i.e. we have a natural transformation σ with injective components $\sigma_X : FX \rightarrowtail GX$. Then the problem of computing the simple quotient for F-coalgebras reduces to that for G-coalgebras: the simple quotient of $(X, \sigma_X \cdot c)$ yields that of (X, c).

Consequently, if F is a subfunctor of G via a subcartesian σ, the minimization problem for F-coalgebras reduces to that for G-coalgebras. For example, the distribution functor $\mathcal{D}$ is a subcartesian subfunctor of $\mathbb{R}^{(-)}$. (For details see the full version [21].)

Preliminaries on Bags. The bag functor defined in Example 2.1 plays an important role in our minimization algorithm, not only as one of many possible system types, but bags are also used as a data structure. To this end, we use a couple of additional properties of this functor.

▶ Remark 2.7.

1. Since $\mathcal{B}$ can also be regarded as a monoid-valued functor for $(\mathbb{N}, +, 0)$, every bag $b = \{\!\{x_1, \dots, x_n\}\!\} \in \mathcal{B}X$ may be identified with a finitely supported function $X \to \mathbb{N}$, assigning to each $x \in X$ its multiplicity in b. We shall often make use of this fact and represent bags as functions.
2. The set $\mathcal{B}X$ itself is a commutative monoid with bag-union as the operation and the empty bag $\{\!\{\}\!\}$ as the identity element. In fact, this is the free commutative monoid over X. It therefore makes sense to consider the monoid-valued functor $(\mathcal{B}X)^{(-)}$ for a monoid of bags. Note that for every pair of sets A, X, the set $(\mathcal{B}A)^{(X)}$ of finitely supported functions from X to $\mathcal{B}A$ is isomorphic to $\mathcal{B}(A \times X)$ as witnessed by the following isomorphism (where swap, curry and $\mathsf{uncurry}$ are the evident canonical bijections):

$$\mathsf{group} = \big(\mathcal{B}(A \times X) \xrightarrow{\mathcal{B}(\mathsf{swap})} \mathcal{B}(X \times A) \xrightarrow{\mathsf{curry}} (\mathcal{B}A)^{(X)}\big), \text{ and}$$
$$\mathsf{ungroup} = \big((\mathcal{B}A)^{(X)} \xrightarrow{\mathsf{uncurry}} \mathcal{B}(X \times A) \xrightarrow{\mathcal{B}(\mathsf{swap})} \mathcal{B}(A \times X)\big).$$

Note that since swap is self-inverse and curry, $\mathsf{uncurry}$ are mutually inverse, group and $\mathsf{ungroup}$ are mutually inverse, too. In symbols:

$$\mathsf{group} \cdot \mathsf{ungroup} = \mathsf{id}_{(\mathcal{B}A)^{(X)}}, \quad \mathsf{ungroup} \cdot \mathsf{group} = \mathsf{id}_{\mathcal{B}(A \times X)}. \tag{1}$$

We often need to filter a bag of tuples $\mathcal{B}(A \times X)$ by a subset $S \subseteq X$. To this end we define the maps $\mathsf{fil}_S : \mathcal{B}(A \times X) \to \mathcal{B}(A)$ for sets $S \subseteq X$ and A by

$$\mathsf{fil}_S(f) = \big(a \mapsto \sum_{x \in S} f(a, x)\big) = \{\!\{ a \mid (a, x) \in f, x \in S \}\!\},$$

where the multiset comprehension is given for intuition.

Zippable Functors. One crucial ingredient for the efficiency of the generic partition refinement algorithm [48] is that the coalgebraic type functor is zippable:

▶ **Definition 2.8** [48, Def. 5.1]**.** A set functor F is called *zippable* if the following maps are injective for every pair A, B of sets:

$$F(A+B) \xrightarrow{\langle F(A+!), F(!+B)\rangle} F(A+1) \times F(1+B).$$

Zippability of a functor allows that partitions are refined incrementally by the algorithm [48, Prop. 5.18], which in turn is the key for allowing a low run time complexity of the implementation. For additional visual explanations of zippability, see [48, Fig. 2]. We shall need this notion in the proof of Proposition 3.3, and later proofs use this result.

It was shown in [48] that all functors in Example 2.1 are zippable. In addition, zippable functors are closed under products, coproducts and subfunctors. However, they are not closed under functor composition, e.g. $\mathcal{P}_\mathsf{f}\mathcal{P}_\mathsf{f}$ is not zippable [48, Ex. 5.10].

The Trnková Hull. For purposes of universal coalgebra, we may assume without loss of generality that set functors preserve injections. Indeed, every set functor preserves nonempty injections (being the split monomorphisms in Set). As shown by Trnková [42, Prop. II.4 and III.5], for every set functor F there exists an essentially unique set functor $\bar{F}$ which coincides with F on nonempty sets and functions, and preserves finite intersections (whence injections). The functor $\bar{F}$ is called the *Trnková hull* of F. Since F and $\bar{F}$ coincide on nonempty sets and maps, the categories of coalgebras for F and $\bar{F}$ are isomorphic.

3 Coalgebra Encodings

In order to make abstract coalgebras tractable for computers and to have a notion of the size of a coalgebra structure in terms of nodes and edges as for standard transition systems, our algorithmic framework encodes coalgebras using a graph-like data structure. To this end, we require functors to be equipped with an encoding as follows.

▶ **Definition 3.1.** An *encoding* of a set functor F consists of a set A of *labels* and a family of maps $\flat_X \colon FX \to \mathcal{B}(A \times X)$, one for every set X, such that the following map is injective:

$$FX \xrightarrow{\langle F!, \flat_X\rangle} F1 \times \mathcal{B}(A \times X).$$

An *encoding* of a coalgebra $c\colon X \to FX$ is given by $\langle F!, \flat_X\rangle \cdot c\colon X \to F1 \times \mathcal{B}(A \times X)$.

Intuitively, the encoding $\flat_X$ of a functor F specifies how an F-coalgebra should be represented as a directed graph, and the required injectivity models that different coalgebras have different representations.

▶ Remark 3.2. Previously [48, Def. 6.1], the map $\langle F!, \flat_X\rangle$ was not explicitly required to be injective. Instead, a family of maps $\flat_X \colon FX \to \mathcal{B}(A \times X)$ and a *refinement interface* for F was assumed. The definition of a refinement interface for F is tailored towards the computation of behaviourally equivalent states and its details are therefore not relevant for the present work. All we need here is that the existence of a refinement interface implies the injectivity condition of Definition 3.1 and consequently, we inherit all examples of encodings from the previous work:

▶ **Proposition 3.3.** *For every zippable set functor F with a family of maps $\flat_X \colon FX \to \mathcal{B}(A \times X)$ and a refinement interface, the family $\flat_X$ is an encoding for F.*

▶ **Example 3.4.** We recall a number of encodings from [48]; the injectivity is clear, and in fact implied by Proposition 3.3:

1. Our encoding for the finite powerset functor $\mathcal{P}_\mathsf{f}$ resembles unlabelled transition systems by taking the singleton set $A = 1$ as labels. The map $\flat_X : \mathcal{P}_\mathsf{f}(X) \to \mathcal{B}(1 \times X) \cong \mathcal{B}(X)$ is the obvious inclusion, i.e. $\flat_X(t)(*, x) = 1$ if $x \in t$ and 0 otherwise.
2. The monoid-valued functor $M^{(-)}$ has labels from $A = M$ and $\flat_X : M^{(X)} \to \mathcal{B}(M \times X)$ is given by $\flat_X(t)(m, x) = 1$ if $t(x) = m \neq 0$ and 0 otherwise.
3. For a polynomial functor F_Σ, we use $A = \mathbb{N}$ as the label set and define the maps $\flat_X : F_\Sigma X \to \mathcal{B}(\mathbb{N} \times X)$ by $\flat_X(\sigma(x_1, \dots, x_n)) = \{\!\!\{ (1, x_1), \dots, (n, x_n) \}\!\!\}$.
   Note that $\flat_X$ itself is not injective if Σ has at least two operation symbols with the same arity. E.g. for DFAs ($F_\Sigma X = 2 \times X^A$), $\flat_X$ only retrieves information about successor states but disregards the "finality" of states. However, pairing $\flat_X$ with $F! : FX \to F1$ yields an injective map.
4. The bag functor $\mathcal{B}$ itself also has $A = \mathbb{N}$ as labels and $\flat_X(t)(n, x) = 1$ if $t(x) = n$ and 0 otherwise. This is just the special case of the encoding for a monoid-valued functor for the monoid $(\mathbb{N}, +, 0)$.

The encoding does by no means imply a reduction of the problem of minimizing F-coalgebras to that of coalgebras for $\mathcal{B}(A \times -)$ (cf. Remark 2.6). In fact, the notions of behavioural equivalence for F-coalgebras and coalgebras for $\mathcal{B}(A \times -)$, can be radically different. If $\flat_X$ is natural in X, then behavioural equivalence wrt. F implies that for $\mathcal{B}(A \times -)$, but not necessarily conversely. However, we do not assume naturality of $\flat_X$, and in fact it fails in all of our examples except one:

▶ **Proposition 3.5.** *The encoding $\flat_X : F_\Sigma X \to \mathcal{B}(A \times X)$ for the polynomial functor F_Σ is a natural transformation.*

▶ **Example 3.6.** The encoding $\flat_X : \mathcal{P}_\mathsf{f}(X) \to \mathcal{B}(1 \times X) \cong \mathcal{B}(X)$ in Example 3.4 item 1 is not natural. Indeed, consider the map $! : 2 \to 1$, for which we have

$$\mathcal{B}(!) \cdot \flat_2(\{0,1\}) = \mathcal{B}(!)\{\!\!\{ 0, 1 \}\!\!\} = \{\!\!\{ *, * \}\!\!\} \neq \{\!\!\{ * \}\!\!\} = \flat_1(\{*\}) = \flat_1 \cdot \mathcal{P}_\mathsf{f}(!)(\{0,1\}).$$

Similar examples show that the encodings in Example 3.4 item 2 (for all non-trivial monoids) and item 4 are not natural.

An important feature of our algorithm and tool is that all implemented functors can be combined by products, coproducts and functor composition. That is, the functors from Example 3.4 are implemented directly, but the algorithm also automatically handles coalgebras for more complicated combined functors, like those in Example 2.2, e.g. $\mathcal{P}_\mathsf{f}(A \times -)$. The mechanism that underpins this feature is detailed in previous work [20, 48] and depends crucially on the ability to form coproducts and products of encodings:

▶ **Construction 3.7** [20, 48]**.** Given a family of functors $(F_i)_{i \in I}$ with encodings $(\flat_{X,i})_{i \in I}$ and $(A_i)_{i \in I}$, we obtain the following encodings with labels $A = \coprod_{i \in I} A_i$:

1. for the coproduct functor $F = \coprod_{i \in I} F_i$ we take

$$\flat_X : \coprod_{i \in I} F_i X \xrightarrow{\coprod_{i \in I} \flat_{X,i}} \coprod_{i \in I} \mathcal{B}(A_i \times X) \xrightarrow{[\mathcal{B}(\mathsf{in}_i \times X)]_{i \in I}} \mathcal{B}\Big(\coprod_{i \in I} A_i \times X\Big).$$

2. for the product functor $F = \prod_{i \in I} F_i$ we take

$$\flat_X : \prod_{i \in I} F_i X \to \mathcal{B}\Big(\coprod_{i \in I} A_i \times X\Big) \qquad\qquad \flat_X(t)(\mathsf{in}_i(a), x) = \flat_i(\mathsf{pr}_i(t))(a, x),$$

   where $\mathsf{in}_i : A_i \to \coprod_j A_j$ and $\mathsf{pr}_i : \prod_j F_j X \to F_i X$ denote the canonical coproduct injections and product projections, respectively.

▶ **Proposition 3.8.** *The families $\flat_X$ defined in Construction 3.7 yield encodings for the functors $\prod_{i\in I} F_i$ and $\coprod_{i\in I} F_i$, respectively.*

▶ Remark 3.9. Since zippable functors are not closed under composition, modularity cannot be achieved by simply providing a construction of an encoding for a composed functor (at least not without giving up on the efficient run-time complexity). Functor composition is reduced to coproducts making a detour via many-sorted sets. Here is a rough explanation of how this works. Suppose that F is a *finitary* set functor, which means that for every $x \in FX$ there exists a finite subset $Y \subseteq X$ and $x' \in FY$ such that $x = Fm(x')$ for the inclusion map $m\colon Y \hookrightarrow X$. Given a finite coalgebra $c\colon X \to FGX$, it can be turned into a 2-sorted coalgebra $(c', d')\colon (X, Y) \to (FY, GX)$ as follows: since F is finitary one picks a finite subset Y of GX such that there exists a map $c'\colon X \to FY$ with $c = Fd' \cdot c'$, where $d'\colon Y \hookrightarrow GX$ is the inclusion map. Then c' and d' are combined into one coalgebra on the disjoint union $X + Y$ as shown below:

$$X + Y \xrightarrow{c'+d'} FY + GX \xrightarrow{[F\,\mathsf{inr}, G\,\mathsf{inl}]} (F+G)(X+Y)$$

for the coproduct of the functors F and G, where $\mathsf{inl}\colon X \to X+Y$ and $\mathsf{inr}\colon Y \to X+Y$ are the two coproduct injections. Full details may be found in [48, Sec. 8].

For the sake of computing the coalgebra structure of the minimized coalgebra, we require that, intuitively, the labels used for encoding FX are independent of the cardinality of X:

▶ **Definition 3.10.** An encoding $\flat_X$ for a set functor F is called *uniform* if it fulfils the following property for every $x \in X$:

$$\begin{tikzcd}
FX \arrow[r, "\flat_X"] \arrow[d, "F\chi_{\{x\}}"'] & \mathcal{B}(A\times X) \arrow[dr, "\mathsf{fil}_{\{x\}}"] & \\
& & \mathcal{B}(A) \\
F2 \arrow[r, "\flat_2"] & \mathcal{B}(A\times 2) \arrow[ur, "\mathsf{fil}_{\{1\}}"'] &
\end{tikzcd} \tag{2}$$

Intuitively, the condition in Definition 3.10 expresses that in an encoded coalgebra, the edges (and their labels) to a state x do not change if other states $y, z \in X \setminus \{x\}$ are identified by a possible partition on the state space. Diagram (2) expresses the extreme case of such a partition, particularly the one where *all* elements of X except for x are identified in a block, with x being in a separate singleton block.

Fortunately, requiring uniformity does not exclude any of the existing encodings that we recalled above.

▶ **Proposition 3.11.** *All encodings from Example 3.4 are uniform.*

Uniform encodings interact nicely with the modularity constructions:

▶ **Proposition 3.12.** *Uniform encodings are closed under product and coproduct.*

That is, given functors $(F_i)_{i\in I}$ with uniform encodings $(\flat_i)_{i\in I}$, then the encodings for the functors $\coprod_{i\in I} F_i$ and $\prod_{i\in I} F_i$, as defined in Construction 3.7, are uniform.

Admittedly, the condition in Definition 3.10 is slightly technical. However, we will now prove that it sits strictly between two standard properties, naturality and subnaturality.

▶ **Proposition 3.13.**

1. *Every natural encoding is uniform.*
2. *Every uniform encoding is a subnatural transformation.*

The converses of both of the above implications fail in general. For the converse of 1 we saw a counterexample in Example 3.6, and for the converse of 2 we have the following counterexample.

▶ **Example 3.14.** Consider the following encoding for the functor $FX = X \times X \times X$ given by $A = 3 + 3$ and

$$\flat_X : FX \to \mathcal{B}(A \times X)$$
$$\flat_X(x, y, z) = \begin{cases} \{(\mathsf{inl}\,0, x), (\mathsf{inl}\,1, y), (\mathsf{inl}\,2, z)\} & \text{if } y = z, \\ \{(\mathsf{inr}\,0, x), (\mathsf{inr}\,1, y), (\mathsf{inr}\,2, z)\} & \text{if } y \neq z. \end{cases}$$

This encoding is subnatural, since the value of $y = z$ is preserved by injections under F. But it is not uniform, for if $x \neq y \neq z$, then we have

$$\mathsf{fil}_{\{1\}}(\flat(F\chi_{\{x\}}(x, y, z))) = \mathsf{fil}_{\{1\}}(\flat(1, 0, 0)) = \{\mathsf{inl}\,0\} \neq \{\mathsf{inr}\,0\} = \mathsf{fil}_{\{x\}}(\flat(x, y, z)).$$

4 Computing the Simple Quotient

The previous coalgebraic partition refinement algorithm and its tool implementation in CoPaR compute for a given encoding of a coalgebra (X, c) the state set of its simple quotient $q\colon (X, c) \twoheadrightarrow (Y, d)$, that is the partition Y of the set X corresponding to behavioural equivalence. But the algorithm does not compute the coalgebra structure d of the simple quotient (and note that it is not given the structure c explicitly, to begin with). Here we will fill this gap. We are interested in computing the encoding $Y \xrightarrow{d} FY \xrightarrow{\flat_Y} \mathcal{B}(A \times Y)$ given the encoding $X \xrightarrow{c} FX \xrightarrow{\flat_X} \mathcal{B}(A \times X)$ of the input coalgebra and the quotient map $q\colon X \twoheadrightarrow Y$.

The edge labels in the encoding of the quotient coalgebra relate to the labels in the encoded input coalgebra in a functor specific way. For example, for weighted transition systems, the labels are the transition weights, which are added whenever states are identified. In contrast, for deterministic automata (or when F is a polynomial functor), the labels (i.e. input symbols) on the transitions remain the same even when states are identified.

Thus, when computing the encoding of the simple quotient, the modification of edge labels is functor specific. Algorithmically, this is reflected by specifying a new interface containing one function `merge`, which is intended to be implemented together with the refinement interface (Section 3) for every functor of interest. The abstract function `merge` is then used in the generic Construction 4.8 in order to compute the encoding of the simple quotient.

▶ **Definition 4.1.** A *minimization interface* for a set functor F equipped with a functor encoding $\flat_X : FX \to \mathcal{B}(A \times X)$ is a function $\mathtt{merge}\colon \mathcal{B}(A) \to \mathcal{B}(A)$ such that the following diagram commutes for all $S \subseteq X$:

$$\begin{tikzcd} FX \arrow[r, "\flat_X"] \arrow[d, "F\chi_S"'] & \mathcal{B}(A \times X) \arrow[r, "\mathsf{fil}_S"] & \mathcal{B}(A) \arrow[d, "\mathtt{merge}"] \\ F2 \arrow[r, "\flat_2"] & \mathcal{B}(A \times 2) \arrow[r, "\mathsf{fil}_{\{1\}}"] & \mathcal{B}(A) \end{tikzcd} \tag{3}$$

Intuitively, `merge` expresses what happens on the labels of edges from one state to one block. It receives the bag of all labels of edges from a particular source state x to a *set of* states S that the minimization procedure identified as equivalent. It then computes the edge labels from x to the merged state S of the minimized coalgebra in a functor specific

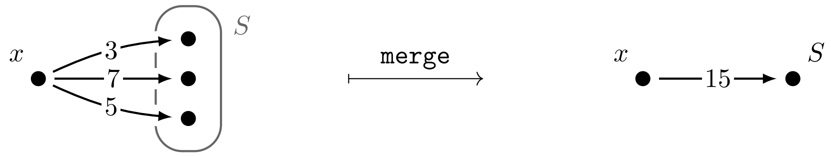


Figure 1 Example application of `merge` for the monoid-valued functor.

way. Figure 1 depicts this process for a monoid-valued functor (cf. Example 2.1, item 2). In this example, `merge` sums up the labels (which are monoid elements), resulting in a correct transition label to the new merged state.

Before we give formal definitions of `merge` for the functors of interest, let us show that there is a close connection between properties of `merge` and the encoding; this will simplify the definition of `merge` later (Example 4.4).

First, if `merge` receives the bag of labels from a source state to a *single* target state, then there is nothing to be merged and thus `merge` should simply return its input bag. Moreover, we can even characterize uniform encodings by this property:

▶ **Lemma 4.2.** *Given a minimization interface, the following are equivalent:*
1. $\mathtt{merge}(\mathsf{fil}_{\{x\}}(\flat_X(t))) = \mathsf{fil}_{\{x\}}(\flat_X(t))$ *for all* $t \in FX$.
2. $\flat_X$ *is uniform.*

Similarly, the property that `merge` is *always* the identity characterizes *natural* encodings:

▶ **Lemma 4.3.** *For every encoding* $\flat_X \colon FX \to \mathcal{B}(A \times X)$*, the following are equivalent:*
1. *The identity on* $\mathcal{B}A$ *is a minimization interface.*
2. $\flat_X$ *is a natural transformation.*

▶ **Example 4.4.**
1. For the finite powerset functor $\mathcal{P}_\mathsf{f}(-)$, with labels $A = 1$, we define $\mathtt{merge}\colon \mathcal{B}1 \to \mathcal{B}1$ by $\mathtt{merge}(\ell)(*) = \min(1, \ell(*))$.
2. For monoid-valued functors $M^{(-)}$ with $A = M$, `merge` is defined as
$$\mathtt{merge}(\ell) = \begin{cases} \{\!\!\{\, \Sigma\ell \,\}\!\!\} & \Sigma\ell \neq 0 \\ \{\!\!\{\}\!\!\} & \text{otherwise,} \end{cases}$$
where $\Sigma\colon \mathcal{B}(M) \to M$ is defined by $\Sigma\{\!\!\{\, m_1, \dots, m_n \,\}\!\!\} = m_1 + \cdots + m_n$.
3. The encoding for the polynomial functor F_Σ for a signature Σ is a natural transformation and hence its minimization interface is given by $\mathtt{merge} = \mathsf{id}$ (see Lemma 4.3).

▶ **Proposition 4.5.** *All* `merge` *maps in Example 4.4 are minimization interfaces and run in linear time in the size of their input bag.*

Having `merge` defined for the functors of interest, we can now use it to compute the encoding of the simple quotient.

▶ **Assumption 4.6.** For the remainder of this section we assume that $F1 \neq \emptyset$.

This is w.l.o.g. since $F1 = \emptyset$ if and only if $FX = \emptyset$ for all sets X, for which there is only one coalgebra (which is therefore its own simple quotient already).

▶ **Proposition 4.7.** *Suppose that the set functor F is equipped with a uniform encoding* $\flat_X\colon FX \to \mathcal{B}(A \times X)$ *and a minimization interface* `merge`*. Then the diagram below commutes for every map* $q\colon X \to Y$*,*

$$\begin{tikzcd} FX \arrow[r, "\flat_X"] \arrow[d, "Fq"'] & \mathcal{B}(A \times X) \arrow[r, "\mathcal{B}(A\times q)"] & \mathcal{B}(A \times Y) \arrow[r, "\mathsf{group}"] \arrow[d, dashed] & \mathcal{B}(A)^{(Y)} \arrow[d, "\mathtt{merge}^{(Y)}"] \\ FY \arrow[rr, "\flat_Y"] & & \mathcal{B}(A \times Y) & \mathcal{B}(A)^{(Y)} \arrow[l, "\mathsf{ungroup}"'] \end{tikzcd} \tag{4}$$

Note that the dashed arrow is not simply the identity map because $\flat_X$ fails to be natural for most functors of interest (Example 3.6).

Proof (Sketch). One first proves that $\mathtt{merge}$ preserves empty bags: $\mathtt{merge}(\{\!\!\{\}\!\!\}) = \{\!\!\{\}\!\!\}$. The commutativity of the desired diagram (4) is proven by extending it by every evaluation map $\mathsf{ev}(y)\colon \mathcal{B}(A)^{(Y)} \to \mathcal{B}(A)$, $y \in Y$, which form a jointly injective family. The extended diagram for $y \in Y$ is then proven commutative using (2) for y, (3) for $S = q^{-1}[y]$, which is also used in the form $\chi_{\{y\}} \cdot q = \chi_S$ in addition to two easy properties of ev and fil: $\mathsf{fil}_{\{y\}} = \mathsf{ev}(y) \cdot \mathsf{group}$ and $\mathsf{fil}_{\{y\}} \cdot \mathcal{B}(A \times q) = \mathsf{fil}_S$. ◀

▶ **Construction 4.8.** Given the encoded F-coalgebra $(X, \flat_X \cdot c)$, the quotient $q\colon X \twoheadrightarrow Y$, and a minimization interface for F, we define the map $e\colon Y \to \mathcal{B}(A \times Y)$ as follows: given an element $y \in Y$, choose any $x \in X$ with $q(x) = y$ and put

$$e(y) := (\mathsf{ungroup} \cdot \mathtt{merge}^{(Y)} \cdot \mathsf{group} \cdot \mathcal{B}(A \times q) \cdot \flat_X \cdot c)(x),$$

where the involved types are as follows:

$$\begin{tikzcd}
X \arrow[r, "c"] \arrow[d, two heads, "q"'] & FX \arrow[r, "\flat_X"] & \mathcal{B}(A \times X) \arrow[r, "\mathcal{B}(A\times q)"] & \mathcal{B}(A \times Y) \arrow[r, "\mathsf{group}"] & \mathcal{B}(A)^{(Y)} \arrow[d, "\mathtt{merge}^{(Y)}"] \\
Y \arrow[rr, "e"] & & \mathcal{B}(A \times Y) & & \mathcal{B}(A)^{(Y)} \arrow[ll, "\mathsf{ungroup}"']
\end{tikzcd} \tag{5}$$

For the well-definedness and the correctness of Construction 4.8, we need to prove that (5) commutes. Moreover, observe that c is not directly given as input, and that the structure $d\colon Y \to FY$ of the simple quotient is not computed; only their encodings $\flat_X \cdot c$ and $e = \flat_Y \cdot d$ are.

▶ **Theorem 4.9.** *Suppose that $q\colon (X, c) \twoheadrightarrow (Y, d)$ represents a quotient coalgebra. Then Construction 4.8 correctly yields the encoding $e = \flat_Y \cdot d$ given the encoding $\flat_X \cdot c$ and the partition of X associated to q.*

If $\mathtt{merge}$ *runs in linear time (in its parameter), then Construction 4.8 can be implemented with linear run time (in the size of the input coalgebra $\flat_X \cdot c$).*

In the run time analysis, a bit of care is needed so that the implementation of group has linear run time; see the full version [21] for details. From Proposition 4.5 we see that for every functor from Example 2.1, Construction 4.8 can be implemented with linear run time.

4.1 Modularity of Minimization Interfaces

Modularity in the system type is gained by reducing functor composition to products and coproducts (Remark 3.9). Since we want the construction of the minimized coalgebra structure to benefit from the same modularity, we need to verify closure under product and coproduct for the notions required in Proposition 4.7. We have already done so for uniform encodings (Proposition 3.12); hence it remains to show that minimization interfaces can also be combined by product and coproduct:

▶ **Construction 4.10.** Given a family of functors $(F_i)_{i \in I}$ together with uniform encodings $\flat_i\colon F_i X \to \mathcal{B}(A_i \times X)$ and minimization interfaces $\mathtt{merge}_i\colon \mathcal{B}(A_i) \to \mathcal{B}(A_i)$, we define $\mathtt{merge}$ for the (co)product functors $\prod_{i \in I} F_i$ and $\coprod_{i \in I} F_i$ as follows:

$$\mathtt{merge}\colon \mathcal{B}(\textstyle\coprod_{i \in I} A_i) \to \mathcal{B}(\coprod_{i \in I} A_i) \qquad \mathtt{merge}(t)(\mathsf{in}_i\, a) = \mathtt{merge}_i(\mathsf{filter}_i(t))(a),$$

where $\mathsf{filter}_i\colon \mathcal{B}(\coprod_{j \in I} A_j) \to \mathcal{B}(A_i)$ is given by $\mathsf{filter}_i(f)(a) = f(\mathsf{in}_i(a))$.

Curiously, the definition of `merge` is the same for products and coproducts, e.g. because the label sets are the same (see Construction 3.7). However, the correctness proofs turns out to be quite different. Note that for coproducts, all labels in the image of $\mathsf{fil}_S \cdot \flat_X$ are in the same coproduct component. Thus, filter_i never removes elements and acts as a mere type-cast when the above `merge` is used in accordance with its specification.

▶ **Proposition 4.11.** *The* `merge` *function defined in Construction 4.10 yields a minimization interface for the functors $\prod_{i\in I} F_i$ and $\coprod_{i\in I} F_i$. It can be implemented with linear run-time if each* $\mathtt{merge}_i$ *is linear in its input.*

▶ **Corollary 4.12.** *The class of set functors having a minimization interface contains all polynomial and all monoid-valued functors and is closed under product and coproduct.*

Consequently, Construction 4.8 correctly yields encoded quotient coalgebras for those functors. Note that all functors from Example 4.4 are contained in this class. Furthermore, functor composition can be dealt with by using coproducts as explained in Remark 3.9.

5 Reachability

Having quotiented an encoded coalgebra by behavioural equivalence, the remaining task is to restrict the coalgebra to the states that are actually reachable from a distinguished initial state. For an intersection preserving set functor, the reachable part of a pointed coalgebra can be constructed iteratively, and this reduces to standard graph search on the canonical graph of the coalgebra [49, Cor. 5.26f], which we now recall. Throughout, $\mathcal{P}$ denotes the (full) powerset functor. The following is inspired by Gumm [28, Def. 7.2]:

▶ **Definition 5.1.** Given a functor $F\colon \mathsf{Set} \to \mathsf{Set}$, we define a family of maps $\tau_X^F\colon FX \to \mathcal{P}X$ by $\tau_X^F(t) = \{x \in X \mid 1 \xrightarrow{t} FX$ does not factorize through $F(X\setminus\{x\}) \xrightarrow{Fi} FX\}$, where $i\colon X\setminus\{x\} \hookrightarrow X$ denotes the inclusion map.

The *canonical graph* of a coalgebra $c\colon X \to FX$ is the directed graph $X \xrightarrow{c} FX \xrightarrow{\tau_X^F} \mathcal{P}X$. The nodes are the states of (X, c) and one has an edge from x to y whenever $y \in \tau_X^F(c(x))$.

Note that for a pointed coalgebra (X, c, i) its canonical graph is equipped with the same point $i\colon 1 \to X$, that is, the canonical graph is equipped with a root node $i(*) \in X$. As we pointed out in Section 2, reachability of the pointed $\mathcal{P}$-coalgebra $(X, \tau_X^F \cdot c, i)$ precisely means that every $x \in X$ is reachable from the root node in the canonical graph.

▶ **Example 5.2.**

1. For a deterministic automaton considered as a coalgebra for $FX = 2 \times X^A$ the canonical graph is precisely its usual underlying state transition graph.
2. For the finite powerset functor $\mathcal{P}_\mathsf{f}$, it is easy to see that $\tau_X^{\mathcal{P}_\mathsf{f}}\colon \mathcal{P}_\mathsf{f}X \hookrightarrow \mathcal{P}X$ is the inclusion map. Thus, the canonical graph of a $\mathcal{P}_\mathsf{f}$-coalgebra (a finitely branching graph) is itself.
3. For the functor $\mathcal{B}(A \times -)$ the maps $\tau_X^{\mathcal{B}(A\times -)}\colon \mathcal{B}(A \times X) \to \mathcal{P}X$ act as follows

$$\{\!\!\{ (a_1, x_1), \dots, (a_n, x_n) \}\!\!\} \mapsto \{x_1, \dots, x_n\}.$$

   Hence, if we view a coalgebra $X \to \mathcal{B}(A \times X)$ as a finitely-branching graph whose edges are labelled by pairs of elements of A and $\mathbb{N}$, then the canonical graph is that same graph but without the edge labels. This holds similarly also for other monoid-valued functors.

To perform reachability analysis on encoded coalgebras, we would like that the canonical graph of a coalgebra and its encoding coincide. This clearly follows when, given a set functor F with encoding $\flat_X : FX \to \mathcal{B}(A \times X)$, the following equation holds for every set X:

$$\tau_X^F = \big(FX \xrightarrow{\flat_X} \mathcal{B}(A \times X) \xrightarrow{\tau_X^{\mathcal{B}(A\times -)}} \mathcal{P}X\big). \tag{6}$$

▶ **Assumption 5.3.** For the rest of this section we assume that F is an intersection preserving set functor equipped with a subnatural encoding $\flat_X : FX \to \mathcal{B}(A \times X)$.

▶ Remark 5.4. That F preserves intersections is an extremely mild condition for set functors. All the functors in Example 3.4 preserve intersections. Furthermore, the collection of intersection preserving set functors is closed under products, coproducts, and functor composition. A subfunctor $\sigma : F \rightarrowtail G$ of an intersection preserving functor G preserves intersections if σ is a cartesian natural transformation, that is all naturality squares are pullbacks (cf. Remark 2.6).

Let us note that for every finitary set functor (cf. Remark 3.9) the Trnková hull $\bar{F}$ (see p. 8) preserves intersections [2, Cor. 8.1.17].

We are now ready to show the desired equality (6) by point-wise inclusion in either direction. Under the running Assumption 5.3 it follows that the encoding of a coalgebra can only mention states that are in the coalgebra's canonical graph:

▶ **Proposition 5.5.** *For every $t \in FX$ we have that $\tau_X^{\mathcal{B}(A\times -)}(\flat_X(t)) \subseteq \tau_X^F(t)$.*

Proof (Sketch). This is shown by contraposition. If x is not in $\tau_X^F(t)$, then we know that the map $t : 1 \to FX$ factorizes through $F(X \setminus \{x\}) \xrightarrow{Fi} FX$ (cf. Definition 5.1). Using the subnaturality square of $\flat$ for the map i then yields $x \notin \tau_X^{\mathcal{B}(A\times -)}(\flat_X(t))$. ◀

For the converse inclusion, we additionally require that F meets the assumptions of the partition refinement algorithm:

▶ **Theorem 5.6.** *The canonical graph of a finite coalgebra coincides with that of its encoding.*

For every finite set X one proves the equation (6): $\tau_X^F = \tau_X^{\mathcal{B}(A\times -)} \cdot \flat_X$. It suffices to prove the reverse of the inclusion in Proposition 5.5 – again by contraposition. This time the argument is more involved using that the map $\langle F!, \flat_X \rangle$ is injective (Definition 3.1), and that F preserves intersections. (For details see the full version [21].)

As a consequence of Theorem 5.6, the states in the reachable part of a pointed coalgebra (X, c, i) are precisely the states reachable from the node $i(*) \in X$ in the (underlying graph of the) encoding $\flat_X \cdot c : X \to \mathcal{B}(A \times X)$, cf. Example 5.23. Thus, given (the encoding of) a pointed coalgebra (X, c, i), its reachable part can be computed in linear time by a standard breadth-first search on the encoding viewed as a graph (ignoring the labels).

This holds for all the functors in Example 3.4 and every functor obtained from them by forming products, coproducts and functor composition.

6 Conclusions and Future Work

We have shown how to extend a generic coalgebraic partition refinement algorithm to a fully fledged minimization algorithm. Conceptually, this is the step from computing the simple quotient of a coalgebra to computing the well-pointed modification of a pointed coalgebra. To achieve this, our extension includes two new aspects: (1) the computation of the transition structure of the simple quotient given an encoding of the input coalgebra and the partition of its state space modulo behavioural equivalence, and (2) the computation of the encoding of the reachable part from the encoding of a given pointed coalgebra. Both of these new steps have also been implemented in the Coalgebraic Partition Refiner CoPaR, together with a new pretty-printing module that prints out the resulting encoded coalgebra in a functor-specific human-readable syntax.

There are a number of questions for further work. This mainly concerns broadening the scope of generic coalgebraic partition refinement algorithms. First, we will further broaden the range of system types that our algorithm and tool can accommodate, and provide support for base categories beside the sets as studied in the present work, e.g. nominal sets, which underlie nominal automata [13, 40].

Concerning genericity, there is an orthogonal approach by Ranzato and Tapparo [37], which is variable in the choice of the *notion of process equivalence* – however within the realm of standard labelled transition systems (see also [25]). Similarly, Blom and Orzan [11, 12] use a technique called *signature refinement*, which handles strong and branching bisimulation as well as Markov chain lumping (see also [45]).

To overcome the bottleneck on memory consumption that is inherent in partition refinement [43, 44], symbolic and distributed methods have been employed for many concrete system types [8, 11, 12, 24, 45, 47]. We will explore in future work whether these methods, possibly generic in the equivalence notion, can be extended to the coalgebraic generality.

References

1 Jiří Adámek, Filippo Bonchi, Barbara König, Mathias Hülsbusch, Stefan Milius, and Alexandra Silva. A coalgebraic perspective on minimization and determinization. In Lars Birkedal, editor, *Proc. Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 7213 of *Lecture Notes Comput. Sci.*, pages 58–73. Springer, 2012.

2 Jiří Adámek, Stefan Milius, and Lawrence S. Moss. Initial algebras, terminal coalgebras, and the theory of fixed points of functors. draft book, July 2020. URL: https://www8.cs.fau.de/ext/milius/publications/files/CoalgebraBook.pdf.

3 JiříAdámek, Stefan Milius, Lawrence S. Moss, and Lurdes Sousa. Well-pointed coalgebras. *Log. Methods Comput. Sci.*, 9(2):1–51, 2014.

4 Michael A. Arbib and Ernest G. Manes. Adjoint machines, state-behaviour machines, and duality. *J. Pure Appl. Algebra*, 6:313–344, 1975.

5 Michael A. Arbib and Ernest G. Manes. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer, 1986.

6 Christel Baier, Bettina Engelen, and Mila Majster-Cederbaum. Deciding bisimilarity and similarity for probabilistic processes. *J. Comput. Syst. Sci.*, 60:187–231, 2000. doi:10.1006/jcss.1999.1683.

7 Falk Bartels, Ana Sokolova, and Erik de Vink. A hierarchy of probabilistic system types. *Theoretical Computer Science*, 327:3–22, 2004.

8 Damien Bergamini, Nicolas Descoubes, Christophe Joubert, and Radu Mateescu. BISIMULATOR: A modular tool for on-the-fly equivalence checking. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2005*, volume 3440 of *Lecture Notes in Comput. Sci.*, pages 581–585. Springer, 2005. doi:10.1007/b107194.

9 Christoph Berkholz, Paul S. Bonsma, and Martin Grohe. Tight lower and upper bounds for the complexity of canonical colour refinement. *Theory Comput. Syst.*, 60(4):581–614, 2017. doi:10.1007/s00224-016-9686-0.

10 Nick Bezhanishvili, Marcello Bonsangue, Helle Hvid Hansen, Dexter Kozen, Clemens Kupke, Prakash Panangaden, and Alexandra Silva. Minimisation in logical form. Technical report, Cornell University, May 2020. available at arXiv:2005.11551.

11 Stefan Blom and Simona Orzan. Distributed branching bisimulation reduction of state spaces. In *Parallel and Distributed Model Checking, PDMC 2003*, volume 89 of *Electron. Notes Theor. Comput. Sci.*, pages 99–113. Elsevier, 2003.

12 Stefan Blom and Simona Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. *STTT*, 7(1):74–86, 2005. `doi:10.1007/s10009-004-0159-4`.

13 Mikołaj Bojańczyk, Bartek Klin, and Slawomir Lasota. Automata theory in nominal sets. *Log. Methods Comput. Sci.*, 10(3), 2014. `doi:10.2168/LMCS-10(3:4)2014`.

14 Filippo Bonchi, Marcello Bonsangue, Helle Hvid Hansen, Prakash Panangaden, Jan Rutten, and Alexandra Silva. Algebra-coalgebra duality in Brzozowski's minimization algorithm. *ACM Trans. Comput. Log.*, 15(1):3:1–3:29, 2014.

15 Filippo Bonchi, Marcello Bonsangue, Jan Rutten, and Alexandra Silva. Brzozowski's algorithm (co)algebraically. In Robert L. Constable and Alexandra Silva, editors, *Logic and Program Semantics, Kozen Festschrift*, volume 7230 of *Lecture Notes in Comput. Sci.*, pages 12–23. Springer, 2012.

16 Janusz A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In J. Fox, editor, *Mathematical Theory of Automata*, volume 12 of *MRI Symposia Series*, pages 529–561. Polytechnic Institute of Brooklyn, Polytechnic Press, 1962.

17 Jin-Yi Cai, Martin Fürer, and Neil Immerman. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12(4):389–410, December 1992. `doi:10.1007/bf01305232`.

18 Stefano Cattani and Roberto Segala. Decision algorithms for probabilistic bisimulation. In *Concurrency Theory, CONCUR 2002*, volume 2421 of *Lecture Notes in Comput. Sci.*, pages 371–385. Springer, 2002. `doi:10.1007/3-540-45694-5`.

19 CoPaR: The Coalgebraic Partion Refiner, February 2021. Available at `https://git8.cs.fau.de/software/copar`.

20 Hans-Peter Deifel, Stefan Milius, Lutz Schröder, and Thorsten Wißmann. Generic partition refinement and weighted tree automata. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods – The Next 30 Years*, pages 280–297, Cham, October 2019. Springer International Publishing. `doi:10.1007/978-3-030-30942-8_18`.

21 Hans-Peter Deifel, Stefan Milius, and Thorsten Wißmann. Coalgebra encoding for efficient minimization. full version with appendix. `arXiv:2102.12842`.

22 Salem Derisavi, Holger Hermanns, and William Sanders. Optimal state-space lumping in markov chains. *Inf. Process. Lett.*, 87(6):309–315, 2003. `doi:10.1016/S0020-0190(03)00343-0`.

23 Ulrich Dorsch, Stefan Milius, Lutz Schröder, and Thorsten Wißmann. Efficient coalgebraic partition refinement. In Roland Meyer and Uwe Nestmann, editors, *28th International Conference on Concurrency Theory (CONCUR 2017)*, volume 85 of *LIPIcs*, pages 28:1–28:16. Schloss Dagstuhl, 2017.

24 Hubert Garavel and Holger Hermanns. On combining functional verification and performance evaluation using CADP. In *Formal Methods Europe, FME 2002*, volume 2391 of *Lecture Notes in Comput. Sci.*, pages 410–429. Springer, 2002. `doi:10.1007/3-540-45614-7`.

25 Jan Groote, David Jansen, Jeroen Keiren, and Anton Wijs. An $O(m \log n)$ algorithm for computing stuttering equivalence and branching bisimulation. *ACM Trans. Comput. Log.*, 18(2):13:1–13:34, 2017. `doi:10.1145/3060140`.

26 Jan Friso Groote, Jao Rivera Verduzco, and Erik P. de Vink. An efficient algorithm to determine probabilistic bisimulation. *Algorithms*, 11(9):131, 2018. `doi:10.3390/a11090131`.

27 H. Peter Gumm. *Thomas Ihringer: Algemeine Algebra. Mit einem Anhang über Universelle Coalgebra von H. P. Gumm*, volume 10 of *Berliner Studienreihe zur Mathematik*. Heldermann Verlag, 2003.

28 H. Peter Gumm. From T-coalgebras to filter structures and transition systems. In José Luiz Fiadeiro, Neil Harman, Markus Roggenbach, and Jan Rutten, editors, *Algebra and Coalgebra in Computer Science*, volume 3629 of *Lecture Notes in Comput. Sci.*, pages 194–212. Springer Berlin Heidelberg, 2005. `doi:10.1007/11548133_13`.

29 Johanna Högberg, Andreas Maletti, and Jonathan May. Backward and forward bisimulation minimization of tree automata. *Theoret. Comput. Sci.*, 410:3539–3552, 2009.

30 John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

31 Dung Huynh and Lu Tian. On some equivalence relations for probabilistic processes. *Fund. Inform.*, 17:211–234, 1992.

32 Joost-Pieter Katoen, Tim Kemna, Ivan Zapreev, and David Jansen. Bisimulation minimisation mostly speeds up probabilistic model checking. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2007*, volume 4424 of *Lecture Notes in Comput. Sci.*, pages 87–101. Springer, 2007. doi:10.1007/978-3-540-71209-1.

33 Bartek Klin. Structural operational semantics for weighted transition systems. In Jens Palsberg, editor, *Semantics and Algebraic Specification: Essays Dedicated to Peter D. Mosses on the Occasion of His 60th Birthday*, volume 5700 of *Lecture Notes in Comput. Sci.*, pages 121–139. Springer, 2009.

34 Barbara König and Sebastian Küppers. A generalized partition refinement algorithm, instantiated to language equivalence checking for weighted automata. *Soft Comput.*, 22:1103–1120, 2018.

35 Nick Nick Bezhanishvili, Clemens Kupke, and Prakash Panangaden. Minimization via duality. In Luke Ong and R. de Queiroz, editors, *Proc. WoLLIC*, volume 7456 of *Lecture Notes in Comput. Sci.* Springer, 2012.

36 Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.

37 Francesco Ranzato and Francesco Tapparo. Generalizing the Paige-Tarjan algorithm by abstract interpretation. *Inf. Comput.*, 206:620–651, 2008. doi:10.1016/j.ic.2008.01.001.

38 Jurriaan Rot. Coalgebraic minimization of automata by initiality and finality. In Lars Birkedal, editor, *Proc. MFPS*, volume 325 of *Electron. Notes Theor. Comput. Sci.*, pages 253–276. Elsevier, 2016.

39 J.J.M.M. Rutten. Universal coalgebra: a theory of systems. *Theoret. Comput. Sci.*, 249(1):3–80, 2000. doi:10.1016/S0304-3975(00)00056-6.

40 Lutz Schröder, Dexter Kozen, Stefan Milius, and Thorsten Wißmann. Nominal automata with name binding. In *Foundations of Software Science and Computation Structures, FOSSACS 2017*, volume 10203 of *Lecture Notes in Comput. Sci.*, pages 124–142, 2017. doi:10.1007/978-3-662-54458-7.

41 Deian Tabakov and Moshe Vardi. Experimental evaluation of classical automata constructions. In G. Sutcliffe and A. Voronkov, editors, *Proc. LPAR*, volume 3835 of *Lecture Notes in Artificial Intelligence*, pages 396–411. Springer, 2005.

42 Věra Trnková. On a descriptive classification of set functors I. *Comment. Math. Univ. Carolin.*, 12:143–174, 1971.

43 Antti Valmari. Bisimilarity minimization in $\mathcal{O}(m \log n)$ time. In *Applications and Theory of Petri Nets, PETRI NETS 2009*, volume 5606 of *Lecture Notes in Comput. Sci.*, pages 123–142. Springer, 2009. doi:10.1007/978-3-642-02424-5.

44 Antti Valmari and Giuliana Franceschinis. Simple $\mathcal{O}(m \log n)$ time Markov chain lumping. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2010*, volume 6015 of *Lecture Notes in Comput. Sci.*, pages 38–52. Springer, 2010.

45 Tom van Dijk and Jaco van de Pol. Multi-core symbolic bisimulation minimization. *J. Softw. Tools Technol. Transfer*, 20(2):157–177, 2018.

46 Boris Weisfeiler. *On Construction and Identification of Graphs*. Springer, 1976. doi:10.1007/bfb0089374.

47 Anton Wijs. Gpu accelerated strong and branching bisimilarity checking. In Christel Baier and Cesare Tinelli, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9035 of *Lecture Notes in Comput. Sci.*, pages 368–383. Springer, 2015.

48 Thorsten Wißmann, Ulrich Dorsch, Stefan Milius, and Lutz Schröder. Efficient and modular coalgebraic partition refinement. *Log. Methods. Comput. Sci.*, 16(1):8:1–8:63, 2020.

49 Thorsten Wißmann, Stefan Milius, Jérémy Dubut, and Shin-ya Katsumata. A coalgebraic view on reachability. *Comment. Math. Univ. Carolin.*, 60(4), 2019.

50 Thorsten Wißmann. *Coalgebraic Semantics and Minimization in Sets and Beyond*. Phd thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2020. URL: `https://opus4.kobv.de/opus4-fau/frontdoor/index/index/docId/14222`.

51 Thorsten Wißmann, Hans-Peter Deifel, Stefan Milius, and Lutz Schröder. From generic partition refinement to weighted tree automata minimization, 2020. accepted for publication in *Formal Aspects of Computing*; available online at `arXiv:2004.01250`. `arXiv:2004.01250`.

52 Lijun Zhang, Holger Hermanns, Friedrich Eisenbrand, and David Jansen. Flow Faster: Efficient decision algorithms for probabilistic simulations. *Log. Meth. Comput. Sci.*, 4(4), 2008. `doi:10.2168/LMCS-4(4:6)2008`.

On the Logical Strength of Confluence and Normalisation for Cyclic Proofs

Anupam Das
University of Birmingham, UK

Abstract

In this work we address the logical strength of confluence and normalisation for non-wellfounded typing derivations, in the tradition of "cyclic proof theory" . We present a circular version CT of Gödel' s system T, with the aim of comparing the relative expressivity of the theories CT and T. We approach this problem by formalising rewriting-theoretic results such as confluence and normalisation for the underlying "coterm" rewriting system of CT within fragments of second-order arithmetic.

We establish confluence of CT within the theory RCA_0, a conservative extension of primitive recursive arithmetic and $I\Sigma_1$. This allows us to recast type structures of hereditarily recursive operations as "coterm" models of T. We show that these also form models of CT, by formalising a totality argument for circular typing derivations within suitable fragments of second-order arithmetic. Relying on well-known proof mining results, we thus obtain an interpretation of CT into T; in fact, more precisely, we interpret level-n-CT into level-$(n+1)$-T, an optimum result in terms of abstraction complexity.

A direct consequence of these model-theoretic results is weak normalisation for CT. As further results, we also show strong normalisation for CT and continuity of functionals computed by its type 2 coterms.

2012 ACM Subject Classification Theory of computation → Equational logic and rewriting; Theory of computation → Proof theory; Theory of computation → Higher order logic; Theory of computation → Lambda calculus

Keywords and phrases confluence, normalisation, system T, circular proofs, reverse mathematics, type structures

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.29

Related Version This work is based on part of the following preprint, where related results, proofs and examples may be found.
Extended Version: `https://arxiv.org/abs/2012.14421` [12]

Funding This work was supported by a UKRI Future Leaders Fellowship, *Structure vs. Invariants in Proofs*, project reference MR/S035540/1.

Acknowledgements I would like to thank Denis Kuperberg, Laureline Pinault and Damien Pous for several interesting discussions on this and related topics. I am also grateful to the anonymous reviewers for their helpful feedback and suggestions.

1 Introduction

Cyclic (or *circular*) proofs have attracted increasing attention in recent years, in settings including modal fixed point logics [28, 16, 35, 1, 18], predicate logic [8, 9, 7, 6], algebras [31, 14, 15, 13], arithmetic [33, 5, 11] and type systems [19, 4, 3]. In short, cyclic proofs are possibly non-wellfounded derivations ("coderivations") that have only finitely many distinct subderivations (and so are finitely presentable). That they are meaningful (i.e., sound, total, terminating, etc.) is usually guaranteed by some ω-regular correctness condition at the level of their infinite branches.

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi; Article No. 29; pp. 29:1–29:23
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this work we investigate the interface between theories of arithmetic and type systems. These two settings are fundamentally related by means of well-known *proof interpretations*, such as the functional and realisability interpretations (see, e.g., [2, 24]). In particular Gödel's system T, a simply typed classical quantifier-free theory with recursion and induction, is capable of interpreting all of Peano Arithmetic, effectively trading off quantifier complexity for abstraction complexity (i.e. type level).

Inspired by the aforementioned previous work on circular type systems, we present a circular version, CT, of T, and compare the relative expressivity of (fragments of) the two theories. More precisely, we show that the restriction of CT to level n (CT_n) is interpreted in the restriction of T to level $n+1$ (T_{n+1}). This result is optimal due to a converse result in parallel work [12] (that is beyond the scope of the present paper).[1]

Since non-wellfounded derivations do not directly admit inductive arguments and their correctness relies on nontrivial infinitary combinatorics, we employ a "proof mining" approach towards establishing this interpretation. More precisely, we formalise models of CT_n within fragments of (second-order) arithmetic, and rely on the aforementioned proof interpretations to extract corresponding terms of T_{n+1}. This builds on analogous aforementioned work in the arithmetic setting, namely [33, 11], also taking advantage of second-order theories.

Our formalisation requires us to establish a form of confluence for the underlying rewrite system of CT, which we show holds in one of the weakest second-order theories RCA_0, essentially a form of primitive recursive arithmetic with quantification over sets. Showing that these structures indeed constitute models of CT requires a formalisation of the totality argument for circular derivations, with quantifiers relativised to this structure.

A direct consequence of these model-theoretic results is weak normalisation for coterms of CT. As further results, we also show strong normalisation for CT and continuity of functionals computed by its type 2 coterms.

Relation to other work. In [26] the authors present a circular version of the underlying type system of T, using a slightly different type language including a Kleene $*$. In particular, they show that circular derivations compute, in the standard model, just the primitive recursive functionals at type 1, i.e. the natural number functions computed by terms of T, also using a formalisation within second-order theories of arithmetic. We generalise that result in several ways: (a) we optimise the result with respect to abstraction complexity; (b) we give a *logical* correspondence, at the level of theories, not just the standard model; (c) we give bona fide confluence and normalisation results for the underlying rewrite system on coterms.

This work is based on part of the (unpublished) preprint [12], where related results, proofs and examples may be found.

Preliminaries. We shall assume some basic familiarity with the underlying technical disciplines of this work, which are now well-established and form the subjects of multiple monographs. In particular, these include rewriting theory [37], subsystems of second-order arithmetic [34, 22], and Gödel's system T and program extraction [2, 24]. Some familiarity with higher-order computability [27] and metamathematics [20, 23, 38] is also helpful.

[1] It is easy, however, to see that T_n is interpreted in CT_n, as we will see in Example 2.5.

$$
\mathsf{ex}\ \dfrac{\vec{\rho}, \sigma, \rho, \vec{\sigma} \Rightarrow \tau}{\vec{\rho}, \rho, \sigma, \vec{\sigma} \Rightarrow \tau}
\qquad
\mathsf{wk}\ \dfrac{\vec{\sigma} \Rightarrow \tau}{\vec{\sigma}, \sigma \Rightarrow \tau}
\qquad
\mathsf{cntr}\ \dfrac{\vec{\sigma}, \sigma, \sigma \Rightarrow \tau}{\vec{\sigma}, \sigma \Rightarrow \tau}
\qquad
\mathsf{cut}\ \dfrac{\vec{\sigma} \Rightarrow \sigma \quad \vec{\sigma}, \sigma \Rightarrow \tau}{\vec{\sigma} \Rightarrow \tau}
$$

$$
\mathsf{id}\ \dfrac{}{\sigma \Rightarrow \sigma}
\qquad
\mathsf{L}\ \dfrac{\vec{\sigma} \Rightarrow \rho \quad \vec{\sigma}, \sigma \Rightarrow \tau}{\vec{\sigma}, \rho \to \sigma \Rightarrow \tau}
\qquad
\mathsf{R}\ \dfrac{\vec{\sigma}, \sigma \Rightarrow \tau}{\vec{\sigma} \Rightarrow \sigma \to \tau}
$$

$$
0\ \dfrac{}{\Rightarrow N}
\qquad
\mathsf{s}\ \dfrac{}{N \Rightarrow N}
\qquad
\mathsf{cond}\ \dfrac{\vec{\sigma} \Rightarrow \tau \quad \vec{\sigma}, N \Rightarrow \tau}{\vec{\sigma}, N \Rightarrow \tau}
\qquad
\mathsf{rec}_\tau\ \dfrac{\vec{\sigma} \Rightarrow \tau \quad \vec{\sigma}, N, \sigma \Rightarrow \tau}{\vec{\sigma}, N \Rightarrow \tau}
$$

Figure 1 Sequent style typing rules for T.

2 A circular version of Gödel's T

Throughout this work we shall work with theories that are *simply* or *finitely* typed. Namely **types**, written σ, τ etc., are generated by the following grammar:

$$\sigma, \tau \quad ::= \quad N \quad | \quad (\sigma \to \tau)$$

A simply typed **theory** is a multi-sorted (classical) first-order theory, whose sorts are just the simple types, equipped with **application** operators $\circ_{\sigma,\sigma\to\tau}$ for each pair $\sigma, \sigma \to \tau$ of types, as usual. (Typed) **terms**, written s, t etc., are formed from constants of a simply typed language under typed application. We simply write $t\,s$ for the application of a term t of type $\sigma \to \tau$ to a term s of type σ. As usual we may sometimes omit parentheses, e.g. writing $r\,s\,t$ instead of $((r\,s)\,t)$.

In this work, we always assume **intensional equality** for simply typed theories. Namely we have binary relation symbols $=_\sigma$ for each type σ, axiomatised by reflexivity, $t =_\sigma t$, and the Leibniz property, $(s =_\sigma t \wedge \varphi(s)) \supset \varphi(t)$, for each formula φ and terms s, t of type σ.

2.1 Sequent calculus presentation of T terms

Sequent calculi give us a way to write typed terms that are more succinct with respect to type level, and also enjoy elegant proof theoretic properties, e.g. cut-elimination. Importantly, the induced relations between type occurrences makes it easier to define our correctness criterion for non-wellfounded derivations later.

▶ **Definition 2.1** (Sequent calculus). *Sequents are expressions $\vec{\sigma} \Rightarrow \tau$, where $\vec{\sigma}$ is a list of types and τ is a type. The typing rules for T are given in Figure 1.*

Here, and throughout this subsection, colours of each type occurrence in typing rules may be ignored for now and will become relevant later in Section 2.2.

Each rule instance (or **step**) determines a constant of the appropriate type. E.g., a step $\dfrac{\vec{\rho} \Rightarrow \rho \quad \vec{\sigma} \Rightarrow \sigma}{\vec{\tau} \Rightarrow \tau}$ is a constant of type $(\vec{\rho} \to \rho) \to (\vec{\sigma} \to \sigma) \to \vec{\tau} \to \tau$.[2] In this way, we may identify each derivation with a term obtained by just repeatedly applying rule instances, starting from the conclusion, to its subderivations. Note that this "combinatory" approach, treating rule instances as constants rather than, say, meta-level operations on λ-terms, ensures that this association of a term to a derivation is *continuous*. This is important for our later association of "coterm" to a "coderivation" .

[2] Here and elsewhere we freely write, say, $\vec{\rho} \to \rho$ for $\rho_1 \to \cdots \to \rho_n \to \rho$ when $\vec{\rho}$ is a list $(\rho_1, \ldots, \rho_n)$.

$$\begin{array}{rcl} \mathsf{id}\ x & = & x \\ \mathsf{ex}\ t\ \vec{x}\ x\ y\ \vec{y} & = & t\ \vec{x}\ y\ x\ \vec{y} \\ \mathsf{wk}\ t\ \vec{x}\ x & = & t\ \vec{x} \\ \mathsf{cntr}\ t\ \vec{x}\ x & = & t\ \vec{x}\ x\ x \end{array} \qquad \begin{array}{rcl} \mathsf{cut}\ s\ t\ \vec{x} & = & t\ \vec{x}\ (s\ \vec{x}) \\ \mathsf{L}\ s\ t\ \vec{x}\ y & = & t\ \vec{x}\ (y\ (s\ \vec{x})) \\ \mathsf{R}\ t\ \vec{x}\ x & = & t\ \vec{x}\ x \end{array}$$

$$\begin{array}{rcl} \mathsf{rec}\ s\ t\ \vec{x}\ 0 & = & s\ \vec{x} \\ \mathsf{rec}\ s\ t\ \vec{x}\ \mathsf{s}z & = & t\ \vec{x}\ z\ (\mathsf{rec}\ s\ t\ \vec{x}\ z) \end{array} \qquad \begin{array}{rcl} \mathsf{cond}\ s\ t\ \vec{x}\ 0 & = & s\ \vec{x} \\ \mathsf{cond}\ s\ t\ \vec{x}\ \mathsf{s}z & = & t\ \vec{x}\ z \end{array}$$

Figure 2 Equational axiomatisation of T, where z is a variable of type N.

1. $\neg\, \mathsf{s}x = 0$
2. $\mathsf{s}x = \mathsf{s}y \supset x = y$
(Ind) If $\vdash \varphi(0)$ and $\vdash \varphi(x) \supset \varphi(\mathsf{s}x)$ then $\vdash \varphi(t)$, for φ quantifier-free.

Figure 3 Number-theoretic axioms for T, where x, y and t are variables/a term of type N.

A term of the form $\overbrace{\mathsf{s}\cdots\mathsf{s}}^{n}0$ is called a **numeral**, and is more succinctly written just $\underline{n}$.

▶ **Definition 2.2** (System T). *T is the simple type theory over the language given by Figure 1, axiomatised by the formulas and rules from Figure 2 and Figure 3.*

▶ Remark 2.3 (Standard model). We may consider usual Henkin structures for simply typed theories, called **type structures**. One particular structure, the "standard" or "full set-theoretic" model $\mathfrak{N}$, is given by the following interpretation:

- $N^{\mathfrak{N}}$ is $\mathbb{N}$ and $(\sigma \to \tau)^{\mathfrak{N}}$ is the set of functions $\sigma^{\mathfrak{N}} \to \tau^{\mathfrak{N}}$.
- $0^{\mathfrak{N}} := 0 \in \mathbb{N}$ and $\mathsf{s}^{\mathfrak{N}}(n) := n + 1$.
- The other constants of T are interpreted by (higher-order) functionals by taking the equations from Figure 2 as definitions, left-to-right.
- Given $f \in \sigma^{\mathfrak{N}}$ and $g \in (\sigma \to \tau)^{\mathfrak{N}}$, $g \circ^{\mathfrak{N}} f \in \tau^{\mathfrak{N}}$ is defined as $g(f)$.
- For each type σ, we have an *extensional* equality relation $=^{\mathfrak{N}}_{\sigma}$:
  - $=^{\mathfrak{N}}_{N}$ is just equality of natural numbers;
  - for $f, g \in (\sigma \to \tau)^{\mathfrak{N}}$, we have $f =^{\mathfrak{N}}_{\sigma\to\tau} g$ just if $\forall x \in \sigma^{\mathfrak{N}}. f(x) =^{\mathfrak{N}}_{\tau} g(x)$.

It is clear, by reduction to induction at the meta-level, that the interpretations of the constants above are well-defined, and that the axioms of Figure 3 (as well as Figure 2) are satisfied in $\mathfrak{N}$. Thus $\mathfrak{N}$ constitutes a bona fide model of T.

2.2 "Coderivations" and a correctness condition

Coterms are generated *coinductively* from constants and variables under typed application. Formally, we may construe a coterm as a possibly infinite binary tree (of height $\leq \omega$) where each leaf (if any) is labelled by a typed variable or constant and each interior node is labelled by a typed application operation, having type consistent with the types of its children. I.e., an interior node with children of types σ and $\sigma \to \tau$, respectively, must have type τ.

Similarly, a **coderivation**, is a possibly non-wellfounded tree built from the derivation rules of Figure 1. As for (well-founded) derivations and terms, we treat coderivations as coterms in the natural way. We say that a coderivation or coterm is **regular** (or **circular**) if it has finitely many distinct sub-coderivations or sub-coterms, respectively. Note that a regular coderivation or coterm is indeed finitely presentable, e.g. as a finite directed graph, possibly with cycles, or a finite binary tree with "backpointers".

Note that the equational theory induced by Figure 2 forms a Kleene-Herbrand-Gödel style equational specification for regular coterms (cf., e.g., [23]). This allows us to view coterms as *partial recursive functionals* in the standard model $\mathfrak{N}$ of the appropriate type, though a full exposition is beyond the scope of this paper. Instead we will give a more formal (and, indeed, formal*ised*) treatment of "regular" coterms and their computational interpretations in Section 3. We point the reader to the excellent book [27] for further details on models of (partial) (recursive) function(al)s.

Nonetheless, let us temporarily adopt the notation $t^{\mathfrak{N}}$ for the partial functional "computed" by a coterm t in $\mathfrak{N}$, and present some examples, at the same time establishing some foundational results. As before, the reader may safely ignore the colouring of type occurrences in what follows. That will become meaningful later in the section.

▶ **Example 2.4** (Extensional completeness at type 1). For *any* $f : \mathbb{N}^k \to \mathbb{N}$, there is a coderivation $t : N^k \Rightarrow N$ s.t. $t^{\mathfrak{N}} = f$. To demonstrate this, we proceed by induction on k.[3] If $k = 0$ then the numerals clearly suffice. Otherwise, suppose $f : \mathbb{N} \times \mathbb{N}^k \to \mathbb{N}$ and write f_n for the projection $\mathbb{N}^k \to \mathbb{N}$ by $f_n(\vec{x}) = f(n, \vec{x})$. We define the coderivation for f as follows:

$$
\mathsf{cond}\ \dfrac{\dfrac{f_0}{\vec{N} \Rightarrow N} \qquad \mathsf{cond}\ \dfrac{\dfrac{f_1}{\vec{N} \Rightarrow N} \qquad \mathsf{cond}\ \dfrac{\dfrac{f_2}{\vec{N} \Rightarrow N} \qquad \mathsf{cond}\ \dfrac{\vdots}{N, \vec{N} \Rightarrow N}}{N, \vec{N} \Rightarrow N}}{N, \vec{N} \Rightarrow N}}{N, \vec{N} \Rightarrow N} \tag{1}
$$

where the derivations for each f_n are obtained by the inductive hypothesis. It is not difficult to see that the interpretation of this coderivation in the standard model indeed coincides with f.

Notice that, while we have extensional completeness at type 1, we cannot possibly have such a result for higher types by a cardinality argument: there are only continuum many coderivations.

▶ **Example 2.5** (Naïve simulation of primitive recursion). Terms of T may be interpreted as coterms without the rec combinators in a straightforward manner, by the following translation:

$$
\mathsf{rec}\ \dfrac{\vec{\sigma} \Rightarrow \sigma \qquad \vec{\sigma}, N, \sigma \Rightarrow \sigma}{\vec{\sigma}, N \Rightarrow \sigma} \quad \rightsquigarrow \quad \mathsf{cond}\ \dfrac{\vec{\sigma} \Rightarrow \sigma \qquad \mathsf{cut}\ \dfrac{\mathsf{cond}\ \dfrac{\vdots}{\vec{\sigma}, N \Rightarrow \sigma}\,\bullet \qquad \vec{\sigma}, N, \sigma \Rightarrow \sigma}{\vec{\sigma}, N \Rightarrow \sigma}}{\vec{\sigma}, N \Rightarrow \sigma}\,\bullet \tag{2}
$$

where the occurrences of $\bullet$ indicate roots of identical coderivations.

[3] While we may assume $k = 1$ WLoG by the availability of sequence (de)coding, the current argument is both more direct and avoids the use of cuts (on non-numerals).

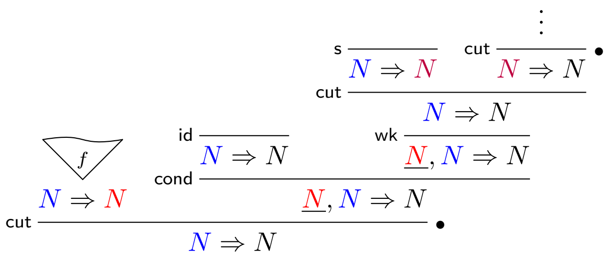
Denoting the RHS of (2) above as rec', we can check that the two sides of (2) are equivalent under Figures 2 and 3. Formally, we show $\mathsf{rec}'\,s\,t\,\vec{x}\,y = \mathsf{rec}\,s\,t\,\vec{x}\,y$ by induction on y:

$$\begin{array}{rcll}
\mathsf{rec}'s\,t\,\vec{x}\,0 & = & \mathsf{cond}\,s\,(\mathsf{cut}\,(\mathsf{rec}'s\,t)\,t)\,\vec{x}\,0 & \text{by definition of } \mathsf{rec}' \text{ above} \\
& = & s\,\vec{x} & \text{by } \mathsf{cond} \text{ axioms} \\
& = & \mathsf{rec}\,s\,t\,\vec{x}\,0 & \text{by } \mathsf{rec} \text{ axioms} \\
\mathsf{rec}'s\,t\,\vec{x}\,\mathsf{s}y & = & \mathsf{cond}\,s\,(\mathsf{cut}\,(\mathsf{rec}'s\,t)\,t)\,\vec{x}\,\mathsf{s}y & \text{by definition of } \mathsf{rec}' \text{ above} \\
& = & \mathsf{cut}\,(\mathsf{rec}'s\,t)\,t\,\vec{x}\,y & \text{by } \mathsf{cond} \text{ axioms} \\
& = & t\,\vec{x}\,y\,(\mathsf{rec}'\,s\,t\,\vec{x}\,y) & \text{by } \mathsf{cut} \text{ axiom} \\
& = & t\,\vec{x}\,y\,(\mathsf{rec}\,s\,t\,\vec{x}\,y) & \text{by inductive hypothesis} \\
& = & \mathsf{rec}\,s\,t\,\vec{x}\,\mathsf{s}y & \text{by } \mathsf{rec} \text{ axioms}
\end{array}$$

▶ **Example 2.6** (Turing completeness). The set of regular coderivations is Turing-complete,[4] i.e. $\{t^{\mathfrak{N}} \mid t : N^k \Rightarrow N \text{ regular}\}$ includes all partial recursive functions on $\mathbb{N}$. We have already seen in Example 2.5 that we can encode the primitive recursive functions, so it remains to simulate minimisation, i.e. the operation $\mu x(fx = 0)$, for a given function f, returning the least natural number x s.t. $fx = 0$ (if it exists). For this, we observe that $\mu x(fx = 0)$ is equivalent to $H\,0$ where:

$$H\,x \;=\; \mathsf{cond}\ (f\,x)\ x\ (H\,\mathsf{s}x) \tag{3}$$

Note that H is computed by the following coderivation:



(4)

It is intuitive here to think of the blue N standing for x, the red N standing for $f(x)$, and the purple N standing for $\mathsf{s}x$. Again, the reader may verify that this coderivation indeed satisfies Equation (3) in the standard model $\mathfrak{N}$. Note that we only used the type N above, and no higher-order types, so Turing-completeness holds already for N-only regular coderivations.

▶ **Definition 2.7** (Immediate ancestry). *Let t be a (co)derivation. A type occurrence σ^1 is an **immediate ancestor**[5] of a type occurrence σ^2 in t if σ^1 and σ^2 appear in the LHSs of a premiss and conclusion, respectively, of a rule instance and have the same colour in the corresponding rule typeset in Figure 1. If σ^1 and σ^2 are elements of an indicated list, say $\vec{\sigma}$, we also require that they are at the same position of the list in the premiss and the conclusion. Note that, if σ^1 is an immediate ancestor of σ^2, they are necessarily occurrences of the same type.*

[4] For a model of program execution, we may simply take the aforementioned Kleene-Herbrand-Gödel model with *equational derivability*, cf. [23]. Note that this coincides with derivability by the axioms thus far presented.

[5] This terminology is standard in proof theory, e.g. as in [10].

The notion of immediate ancestor thus defined, being a binary relation, induces a directed graph whose paths will form the basis of our termination criterion.

▶ **Definition 2.8** (Threads and progress). *A **thread** is a maximal path in the graph of immediate ancestry. A **σ-thread** is a thread whose elements are occurrences of the type σ. We say that a N-thread **progresses** when it is principal for a* cond *step (i.e. it is the indicated blue N in the* cond *rule typeset in Figure 1). A (infinitely) **progressing** thread is a N-thread that progresses infinitely often (i.e. it is infinitely often the indicated blue N in the* cond *rule typeset in Figure 1.)*

*A coderivation is **progressing** if every infinite branch has a progressing thread.*

Note that progressing threads do not necessarily begin at the root of a coderivation, they may begin arbitrarily far into a branch. In this way, the progressing coderivations are closed under all typing rules. Note also that arbitrary coderivations may be progressing, not only the regular ones.

▶ **Example 2.9** (Extensional completeness at type 1, revisited). Recalling Example 2.4, note that the infinite branch marked $\cdots$ in (1) has a progressing thread along the red Ns. Other infinite branches, say through f_0, f_1, etc., will have progressing threads along their infinite branches by an appropriate inductive hypothesis, though these may progress for the first time arbitrarily far from the root of (1).

As previously mentioned, we shall focus our attention in this work on the regular coderivations. Let us take a moment to appreciate some previous (non-)examples of regular coderivations with respect to the progressing criterion.

▶ **Example 2.10** (Primitive recursion and Turing-completeness, revisited). Recalling Example 2.5, notice that the RHS of (2) is a progressing coderivation: there is precisely one infinite branch (that loops on $\bullet$) and it has a progressing thread on the blue N indicated there.

Now recalling Example 2.6, notice that the coderivation given for H in (4) is *not* progressing: the only infinite branch loops on $\bullet$ and immediate ancestry, as indicated by the colouring, admits no thread along the $\bullet$-loop.

One of the most appealing features of the progressing criterion is that it is decidable (for regular coderivations) by a well-known reduction to unviersality of Büchi automata (see, e.g., [17] for an exposition for a similar circular system). On the semantic side, we duly have:

▶ **Proposition 2.11.** *If $t : \vec\sigma \Rightarrow \tau$ is a progressing coderivation, then $t^{\mathfrak{N}}$ is a well-defined total functional in $(\vec\sigma \to \tau)^{\mathfrak{N}}$.*

Proof sketch. First, observe that each constant (i.e. rule instance) computes a total functional of corresponding type. Thus, contrapositively, if $t^{\mathfrak{N}}$ is non-total then so is one of its immediate sub-coderivations. Continuing this reasoning yields an infinite branch $(t_i : \vec\sigma_i \Rightarrow \tau_i)_i$ of non-total coderivations. Now, by the progressing criterion, there must be a progressing thread $(N_i)_{i\geq k}$ along this branch. Assigning to each occurrence N_i the least natural number n_i on which t_i is non-total yields a monotone non-increasing sequence $(n_i)_{i\geq k}$ that does not converge (by definition of progressing thread), giving the required contradiction. ◀

2.3 Some fragments and program extraction

Let us write T^- for the restriction of T to the language without the rec constants from Figure 1, and so also without the rec axioms from Figure 2.

▶ **Definition 2.12** (Circular version of T). *The language of CT contains every regular progressing coderivation of T^- as a symbol. We identify "terms" of this language (i.e. finite applications of regular progressing coderivations, constants and variables) with coterms in the obvious way, and call them* ***regular progressing coterms****. CT itself is axiomatised by the schemata from Figures 2 and 3, now interpreting the metavariables s, t etc. there as ranging over (regular progressing) coterms.*

The aim of this work is to compare fragments of CT and fragments of T delineated by type level. Recall that the **level** of a type σ, written $\mathsf{lev}(\sigma)$ is given by: $\mathsf{lev}(N) := 0$ and $\mathsf{lev}(\sigma \to \tau) := \max(1 + \mathsf{lev}(\sigma), \mathsf{lev}(\tau))$.

▶ **Definition 2.13** (Type level restricted fragments of T and CT). *T_n is the restriction of T to the language containing only recursors rec_σ where $\mathsf{lev}(\sigma) \leq n$.*

CT_n is the restriction of CT to the language containing only coderivations where all types occurring have level $\leq n$. CT_n still has symbols for each constant of T^-.

Note that this definition of CT_n is quite natural, since it is known that T_n derivations (of level $n+1$ functionals) can be put into an analogous form (see, e.g., [12]). For instance, the coderivation in Equation (4) has level 0 (though it is not an element of CT_0 since it is not progressing). Note that CT itself is just the union of all CT_n, since regular coderivations have only finitely many type occurrences and so exhibit a maximum type level.

The significance of the fragments T_n, in terms of quantifier-restricted fragments of arithmetic, was investigated in the seminal work of Parsons [29]. Let us first recall such fragments in a two-sorted framework.

RCA_0 is a second-order[6] theory in the language of arithmetic (i.e. with symbols $0, \mathsf{s}, +, \times, <$). It is axiomatised by an appropriate extension of Robinson's Q to the second-order setting, along with comprehension for (provably) Δ^0_1 predicates and induction for Σ^0_1 formulas. A comprehensive presentation of RCA_0 and related theories can be found in, e.g., [34, 22].

Writing $I\Sigma^0_n$ for the induction scheme for Σ^0_n formulas we have:

▶ **Proposition 2.14** ([29]). *If $\mathsf{RCA}_0 + I\Sigma^0_{n+1} \vdash \forall \vec{x} \exists y A(\vec{x}, y)$, where A is Δ^0_0, then there is a T_n term t with $T_n \vdash A(\vec{x}, t\,\vec{x})$.*[7]

Since we use it later, let us note that $I\Sigma^0_n$ is equivalent, over a weak base theory (certainly RCA_0), to induction on Boolean combinations of Σ^0_n formulas, cf., e.g., [20]. The theory ACA_0 is obtained from RCA_0 by adding comprehension for arithmetical predicates, and is equivalent, over arithmetical theorems, to the extension of RCA_0 by arithmetical induction.

Let us also mention a nontrivial result from previous work that we shall make use of:

▶ **Proposition 2.15** ([11]). *For any regular progressing coderivation t, RCA_0 proves that t is progressing.*

Since progressiveness is, a priori, a Π^1_2 property, the above result is not at all immediate and relies on a formalisation of Büchi automaton theory that is implicit in [25]. Note that this result is "non-uniform" , in that the quantification over coderivations t takes place at the meta-level. As noted in [11], the above result cannot be strengthened to a uniform one unless RCA_0 (and so PRA) is inconsistent, by a reduction to Gödel-incompleteness.

---

[6] As for simple type theories, all references to "second" or "higher" order are purely due to convention. Strictly speaking, these are multi-sorted first-order theories.

[7] We assume here some standard encoding of Δ_0 formulas into quantifier-free formulas of T_0. Alternatively we could admit bounded quantifiers into the language of T, on which induction is allowed, without affecting expressivity. We shall gloss over this technicality here.

3 Confluence and models of T

We cannot formalise the standard model $\mathfrak{N}$ in arithmetic for cardinality reasons, however there are natural models of partial recursive functionals that can be formalised, namely the *hereditarily recursive* operations of finite type (see, e.g., [27]). We shall recast this type structure using regular coterms, in light of Example 2.6 and Example 2.10.

3.1 Reduction sequences and their logical complexity

▶ **Definition 3.1.** *The* ***reduction*** *relation $\rightsquigarrow$ on coterms is defined by orienting all the equations in Figure 2 left-to-right and taking closure under substitution and contexts. We write $\approx$ for the reflexive, symmetric, transitive closure of $\rightsquigarrow$, and freely use standard rewriting theoretic terminology and notations for these relations.*

Since coterms are potentially infinite, equality for them is a Π^0_1 predicate. Thus, for the sake of simplicity, we shall henceforth deal with only regular coterms, which are finite so may be coded by natural numbers. Representing regular coterms as finite directed graphs, note that equality now reduces to checking bisimilarity, which is provably recursive in RCA_0.

In fact, throughout this section, we will only deal with coterms that are finite applications of regular coderivations, variables and constants ("FARs" for short). We better show that these are at least closed under reduction. To this end, let us write, for $v \in \{0,1\}^*$, t_v for the sub-coterm of t rooted at position v. We have:

▶ **Proposition 3.2** (RCA_0). *If $s \rightsquigarrow t$ then t is finitely composed of sub-coterms of s:*

$$\exists \text{ a finite term } r(x_1, \dots, x_n).\, \exists \langle v_1, \dots, v_n \rangle.\, t = r(s_{v_1}, \dots, s_{v_n}) \tag{5}$$

We can take $s_{v_1}, \dots, s_{v_n}$ to include the coderivations indicated in the contractum of a reduction, as well as the "comb" of the redex of the reduction in s, i.e. the siblings of all the nodes in the path leading to the redex. $r(\vec{x})$ is now the finite term induced by the contracta and this comb.

Naturally, this property also holds for $\rightsquigarrow^*$ and $\approx$, by Σ^0_1-induction. As a consequence:

▶ **Corollary 3.3** (RCA_0). *If s is a FAR and $s \rightsquigarrow t$ or $s \rightsquigarrow^* t$ or $s \approx t$, then t is a FAR.*

Note, in particular, that $\rightsquigarrow$, $\rightsquigarrow^*$ and $\approx$, restricted to FARs, are Σ^0_1-relations.

3.2 Confluence of reduction

In order to obtain basic metamathematical properties of the coterm models we later consider, we need to know that our model of computation is *deterministic*, so that coterms have unique interpretations. There are various ways to prove this in arithmetic, but we will approach it in terms of *confluence* in rewriting theory.

Throughout this subsection we continue to deal only with FARs, i.e. coterms that are finite applications of regular coderivations, variables and constants. The main goal of this subsection is to prove the following:

▶ **Theorem 3.4** (Church-Rosser, RCA_0). *Let $t : \sigma$ be a FAR. If $t_0 \mathrel{{}^*{\leftsquigarrow}} t \rightsquigarrow^* t_1$ then there is $t' : \sigma$ such that $t_0 \rightsquigarrow^* t' \mathrel{{}^*{\leftsquigarrow}} t_1$.*

To some extent, we follow a standard approach to proving this result. However, since coterms are infinite (and, moreover, non-wellfounded), we must carry out our argument without appeal to induction on *term structure*, as is usual in presentations of arguments due to Tait and Martin-Löf (cf., e.g., [21]). Instead, we perform an argument by induction on reduction *length*, as in, e.g., [30].

▶ **Definition 3.5** (Parallel reduction). *We define the relation $\rhd$ on FARs as follows:*
1. $t \rhd t$ *for any FAR* t.[8]
2. *For a reduction step* $\mathsf{r}\,\vec{t} \rightsquigarrow r(\vec{t})$, *if each* $t_i \rhd t'_i$ *then we have* $\mathsf{r}\,\vec{t} \rhd r(\vec{t'})$.
3. *For a reduction step* $\mathsf{r}\,\vec{t}\mathsf{s}s \rightsquigarrow r(\vec{t}, s)$ *(i.e. a* rec *or* cond *successor step), if each* $t_i \rhd t'_i$ *and* $s \rhd s'$ *then we have* $\mathsf{r}\,\vec{t}\mathsf{s}s \rhd r(\vec{t'}, s')$.
4. *If* $s \rhd s'$ *and* $t \rhd t'$ *then* $s\,t \rhd s'\,t'$.

▶ **Proposition 3.6** (RCA_0). $s \rightsquigarrow t \implies s \rhd t$ *and* $s \rhd t \implies s \rightsquigarrow^* t$.

The proof of this result is not difficult, but before giving an argument let us point out a particular consequence that we will need, obtained by Σ^0_1-induction on the length of reduction sequences:

▶ **Corollary 3.7** (RCA_0). $s \rightsquigarrow^* t \iff s \rhd^* t$

Even though it is not necessary to prove the proposition above, we shall first prove the following useful lemma since we will use it later:

▶ **Lemma 3.8** (Substitution, RCA_0). *Suppose* $t \rhd t'$. *If* $s \rhd s'$ *then* $s[t/x] \rhd s'[t'/x]$, *for a variable* x *of the same type as* t *and* t'.

Writing, say, $d : s \rightsquigarrow^* t$ for the (provably) Δ^0_1 predicate "d is a $\rightsquigarrow$-derivation from s to t", the above result is shown by proving

$$d : s \rhd s' \implies s[t/x] \rhd s'[t'/x]$$

by Σ^0_1-induction on the structure of the derivation $d : s \rhd s'$. We crucially use the fact that we are dealing with FARs for the base case when $s' = s$, using a subinduction on the maximum depth of an x-occurrence in s.

Notice that Proposition 3.6 now follows immediately, by simply instantiating the Lemma above with $s = s'$ to deduce context-closure of $\rhd$.

▶ **Lemma 3.9** (Diamond property of $\rhd$, RCA_0). *Suppose* $t_0 \lhd s \rhd t_1$. *Then there is some* u *with* $t_0 \rhd u \lhd t_1$.

Before giving the proof, it will be useful to have the following intermediate result, which follows by Σ^0_1-induction:

▶ **Proposition 3.10** (RCA_0). *Suppose* $d : \mathsf{r}\,\vec{s} \rhd t$, *and there is no redex in* $\mathsf{r}\,\vec{s}$ *involving* r. *There are some* $\vec{t}$ *s.t.* $t = \mathsf{r}\,\vec{t}$ *and, for each* i, *some* $d_i : s_i \rhd t_i$ *for some* $d_i < d$.

The diamond property, Lemma 3.9, now follows by proving

$$\exists s'. ((d_0 : s \rhd t_0 \ \text{ and } \ d_1 : s \rhd t_1) \implies (t_0 \rhd s' \text{ and } t_1 \rhd s'))$$

by Σ^0_1-induction on $\min(|d_0|, |d_1|)$. We use Lemma 3.8 for the case when both d_0 and d_1 end by clause (2), and we use Proposition 3.10 when d_0 ends by clause (2) and d_1 ends by clause (4) or vice-versa.

[8] Note that we really do seem to require $t \rhd t$ for arbitrary FARs t, not just variables and constants, since we cannot finitely derive the former from the latter.

▶ **Proposition 3.11** (Weighted CR for $\rhd$, RCA_0). *If $t_0 \lhd^m t \rhd^n t_1$ then there is some t' with $t_0 \rhd^n t' \lhd^m t_1$.*

The argument for this follows by proving

$$(d_0 : t \rhd^m t_0 \text{ and } d_1 : t \rhd^n t_1) \implies \exists t'(t_0 \rhd^n t' \text{ and } d_1' : t_1 \rhd^m t')$$

by Σ^0_1-induction on $m = |d_0|$. The following corollary is immediate:

▶ **Corollary 3.12** (CR for $\rhd$, RCA_0). *If $t_0 \lhd^* t \rhd^* t_1$ then there is t' s.t. $t_0 \rhd^* t' \lhd^* t_1$.*

We may finally conclude the main result of this subsection:

Proof of Theorem 3.4. Suppose $t_0 \leftsquigarrow^* s \rightsquigarrow^* t_1$. Then, by Corollary 3.7 we have $t_0 \lhd^* s \rhd^* t_1$. By Corollary 3.12 above, we have some s' with $t_0 \rhd^* s' \lhd^* t_1$, whence $t_0 \rightsquigarrow^* s' \leftsquigarrow t_1$ by Corollary 3.7 again. ◀

3.3 Hereditarily total coterms under conversion

We are now ready to present a type structure that will allow us to obtain an interpretation of CT_n within T_{n+1}. The structure that we present in this subsection is essentially the *hereditrarily recursive operations* of finite type, but where we adopt FARs under conversion as the underlying model of computation, cf. Example 2.6 and Example 2.10.

▶ **Definition 3.13.** *We define the following sets of FARs:*
- $\mathsf{HR}_N := \{t : N \mid \exists n \in \mathbb{N}.\ t \approx \underline{n}\}$
- $\mathsf{HR}_{\sigma\to\tau} := \{t : \sigma \to \tau \mid \forall s \in \mathsf{HR}_\sigma.\ t\,s \in \mathsf{HR}_\tau\}$

We write HR_n for the union of all HR_σ with $\mathsf{lev}(\sigma) \le n$.

Note that it is immediate from the definition that each HR_σ contains only closed FARs of type σ. Notice that, by the confluence result of the previous subsection, Theorem 3.4, if $t \approx \underline{n}$ then $n \in \mathbb{N}$ is unique and in fact $t \rightsquigarrow^* \underline{n}$ (provably in RCA_0). In this way we can view every element of HR_N as computing a unique natural number by means of reduction.

▶ **Fact 3.14.** *HR_N is Σ^0_1, and if $\mathsf{lev}(\sigma) = n > 0$ then HR_σ is Π^0_{n+1}.*

This is obtained by a (meta-level) induction on the type σ. The same induction also yields:

▶ **Proposition 3.15** (Closure properties of HR). *Fix types σ and τ. RCA_0 proves the following:*
1. *If $s \in \mathsf{HR}_\sigma$ and $t \in \mathsf{HR}_{\sigma\to\tau}$ then $ts \in \mathsf{HR}_\tau$. (HR closed under application)*
2. *If $t \in \mathsf{HR}_\tau$ and $t \approx t'$ then $t' \in \mathsf{HR}_\tau$. (HR closed under conversion)*

Note that provability within RCA_0 above is *non-uniform* in σ and τ, i.e. RCA_0 proves the statements for each particular σ and τ. These properties justify defining the following type structure:

▶ **Definition 3.16** (HR structure). *We write HR for the type structure defined as follows:*
- σ^{HR} *is* HR_σ.
- r^{HR} *is just* r *for each constant* r.
- $t \circ^{\mathsf{HR}} s$ *is just* ts.
- $=^{\mathsf{HR}}_\sigma$ *is* $\approx_\sigma$.

Ultimately we will show that this structure constitutes a model of CT. For this the following lemma will be key:

▶ **Lemma 3.17** (Induction for HR, RCA_0). *Suppose $r(x)$ and $s(x)$ are FARs. If $r(0) \approx s(0)$ and $\forall t \in \mathsf{HR}_N.(r(t) \approx s(t) \implies r(\mathsf{s}t) \approx s(\mathsf{s}t))$, then $\forall t \in \mathsf{HR}_N.r(t) \approx s(t)$.*

This result is essentially “forced” by the definition of HR_N, reducing induction in HR to induction in RCA_0. We also rely on the Leibniz property of equality in the structure (i.e. if $s \approx t$ and $\varphi(s)$ then $\varphi(t)$), which is facilitated by the symmetry and transitivity of $\approx$.

Note that the axioms governing the constants are immediate given that our reduction relation is obtained from them. The remaining number-theoretic axioms follow from confluence (for $\neg \mathsf{s}0 \approx 0$, by uniqueness of normal forms) and the fact that no reduction rule has s at the head (for $\mathsf{s}s \approx \mathsf{s}t$ implies $s \approx t$, requiring a Σ^0_1-induction).

Thus to conclude that HR actually constitutes a model of T (or CT) it remains to show that it interprets each term t of T (or coterm of CT), i.e. that indeed $t \in \mathsf{HR}$. For T, this follows from Tait's seminal normalisation result [36]:

▶ **Proposition 3.18.** HR *is a model of* T*.*

In fact this result can be formalised non-uniformly in the following sense: for each term t of type τ with $\mathsf{lev}(\sigma) \leq n$, we have $\mathsf{RCA}_0 + I\Sigma^0_{n+1} \vdash \mathsf{HR}_\tau(t)$. We will see a similar situation for membership of CT_n coderivations in HR_{n+1} later, but with the quantifier complexity of induction increased by 1.

4 Interpretation of CT into T

In this section we show that the type structure HR introduced in the previous section indeed constitutes a model of CT. In fact, we will formalise the membership of CT_n coderivations in HR_{n+1} within the theory $\mathsf{RCA}_0 + I\Sigma^0_{n+2}$ (non-uniformly), whence we obtain explicit equivalent terms of T_{n+1} by program extraction. Throughout this section we continue to work only with regular coterms that are finite applications of coderivations, variables and constants (i.e. FARs).

4.1 Canonical branches of non-total coterms

In this section we give a formalised proof of the totality of CT-coterms. Our approach will be to import a suitable version of the proof of Proposition 2.11 but relativise all the quantifiers, there in the standard model, to their respective domains in HR.

First let us note that HR is closed under the typing rules of CT:

▶ **Observation 4.1.** *Consider a rule instance* $\mathsf{r}\,\dfrac{\vec\sigma_0 \Rightarrow \tau_0 \quad \cdots \quad \vec\sigma_k \Rightarrow \tau_k}{\vec\sigma \Rightarrow \tau}$ *for some* $k < 2$*. If* $t_i \in \mathsf{HR}_{\vec\sigma_i \to \tau_i}$ *for* $i < k$ *then* $\mathsf{r}\ t_0 \ \cdots \ t_k \in \mathsf{HR}_{\vec\sigma \to \tau}$*.*

This follows by simple inspection of the rules of CT. By contraposition, any coderivation $\notin \mathsf{HR}$ must induce an infinite branch of coderivations $\notin \mathsf{HR}$, similarly to the proof of Proposition 2.11. The next definition formalises a canonical such branch, as induced by an input on which a coderivation is non-hereditarily-total. We shall present just the definition of the branch first, and then argue that it is well-defined, for each explicit CT_n coderivation, in $\mathsf{RCA}_0 + I\Sigma^0_{n+2}$.

▶ **Definition 4.2** (Branch generated by a non-total input)**.** *Let* $t_0 : \vec\sigma_0 \Rightarrow \tau_0$ *be a coderivation and let* $\vec s_0 \in \mathsf{HR}_{\vec\sigma_0}$ *s.t.* $t_0\,\vec s_0 \notin \mathsf{HR}_\tau$*. We define the branch* $(t_i : \vec\sigma_i \Rightarrow \tau_i)_{i \geq 0}$ *and inputs* $\vec s_i \in \mathsf{HR}_{\vec\sigma_i}$, generated *by* t_0 *and* $\vec s_0$ *below. Each rule instance is as typeset in Figure 1, with immediate sub-coderivations* t *and* t' *respectively. Furthermore, we preserve the invariant* $t_i\,\vec s_i \notin \mathsf{HR}_{\tau_i}$ *throughout the definition.*

1. *(t_i cannot be an initial sequent).*
2. *Suppose t_i ends with* wk *and $\vec{s}_i = (\vec{s}, s)$. Then $t_{i+1} := t$ and $\vec{s}_{i+1} := \vec{s}$.*
3. *Suppose t_i ends with* ex *and $\vec{s}_i = (\vec{r}, r, s, \vec{s})$. Then $t_{i+1} := t$ and $\vec{s}_{i+1} := (\vec{r}, s, r, \vec{s})$.*
4. *Suppose t_i ends with* cntr *and $\vec{s}_i = (\vec{s}, s)$. Then $t_{i+1} := t$ and $\vec{s}_{i+1} := (\vec{s}, s, s)$.*
5. *Suppose t_i ends with* cut *and $\vec{s}_i = \vec{s}$. Then if $t\,\vec{s} \in \mathsf{HR}_\sigma$ then $t_{i+1} := t'$ and $\vec{s}_{i+1} := (\vec{s}, t\,\vec{s})$. Otherwise, $t_{i+1} := t$ and $\vec{s}_{i+1} := \vec{s}$.*
6. *Suppose t_i ends with* L *and $\vec{s}_i = (\vec{s}, s)$. If $t\,\vec{s} \in \mathsf{HR}_\rho$ then $t_{i+1} := t'$ and $\vec{s}_{i+1} := (\vec{s}, s\,(t\,\vec{s}))$. Otherwise $t_{i+1} := t$ and $\vec{s}_{i+1} := \vec{s}$.*
7. *Suppose t_i ends with* R *and $\vec{s}_i = \vec{s}$. Let s be the least[9] element of HR_σ such that $t\,\vec{s}\,s \notin \mathsf{HR}_\tau$. We set $t_{i+1} := t$ and $\vec{s}_{i+1} := (\vec{s}, s)$.*
8. *Suppose t_i ends with* cond *and $\vec{s}_i = (\vec{s}, r)$. If $r \approx 0$ then $t_{i+1} := t$ and $\vec{s}_{i+1} := \vec{s}$. Otherwise, if $r \approx \mathsf{s}\underline{n}$, then $t_{i+1} := t'$ and $\vec{s}_{i+1} := (\vec{s}, \underline{n})$.*

The main result of this subsection is:

▶ **Proposition 4.3.** *Let $t_0 : \vec{\sigma}_0 \Rightarrow \tau_0$ be a fixed coderivation in which all types occurring have level $\leq n$. $\mathsf{RCA}_0 + I\Sigma^0_{n+2}$ proves the following: if $\vec{s}_0 \in \mathsf{HR}_{\vec{\sigma}_0}$ s.t. $t_0\,\vec{s}_0 \notin \mathsf{HR}_{\tau_0}$ then the branch $(t_i)_i$ and inputs $(\vec{s}_i)_i$ generated by t_0 and s_0 are Δ^0_{n+2}-well-defined.*

Most of the cases follow by the inductive hypothesis and the closure of HR under $\approx$. Crucially, for the R case, we must use the Σ^0_{n+1}-minimisation principle, a consequence of $I\Sigma^0_{n+1}$ cf. [20], to find the "least" FAR s satisfying a Σ^0_{n+1} property. We also use confluence to ensure that the cond-case is well-defined.

4.2 Progressing coterms are hereditarily total

We are now ready to show that CT-coterms are hereditarily total, i.e. that they belong to HR. Now that we have formalised the infinite "non-total" branches of the proof of Proposition 2.11, relativised to the type structure HR, we continue to formalise the remainder of the argument. First, again by confluence, we have:

▶ **Lemma 4.4** (RCA_0)**.** *Let $t_0 : \vec{\sigma}_0 \Rightarrow \tau_0$ and $\vec{s}_0 \in \mathsf{HR}_{\vec{\sigma}_0}$ be a coderivation and inputs s.t. $t_0\,\vec{s}_0 \notin \mathsf{HR}_{\tau_0}$. Furthermore let $(t_i : \vec{\sigma}_i \Rightarrow \tau_i)_i$ and $\vec{s}_i \in \mathsf{HR}_{\vec{\sigma}_i}$ be a branch and inputs generated by t_0 and $\vec{s}_0$, satisfying Definition 4.2.*

Suppose some N-occurrence $N^{i+1} \in \vec{\sigma}_{i+1}$ is an immediate ancestor of some N-occurrence $N^i \in \vec{\sigma}_i$. Write $s_i \in \vec{s}_i$ for the coterm in HR_N corresponding to N^i, and similarly $s_{i+1} \in \vec{s}_{i+1}$ for the coterm $s_{i+1} \in \mathsf{HR}_N$ corresponding to N^{i+1}. If $s_i \approx \underline{n}_i$ and $s_{i+1} \approx \underline{n}_{i+1}$, for $n_i, n_{i+1} \in \mathbb{N}$, then:

1. *$n_i \geq n_{i+1}$.*
2. *If N^i is principal for a* cond *step, then $n_i > n_{i+1}$.*

In order to complete our formalisation of the totality argument, we actually have to use an "arithmetical approximation" of thread progression that nonetheless suffices for our purposes, similarly to [11]. The reason for this is that, even though non-total branches are well-defined by Proposition 4.3, we do not a priori have access to them as *sets* in extensions of RCA_0 by induction principles, and so the lack of progressing threads along them does not directly contradict the fact that a coderivation is progressing.[10]

---

[9] Recall that, strictly speaking, we assume all our objects are coded by natural numbers in the ambient theory (here fragments of second-order arithmetic). Thus we may always find a "least" object satisfying a property when one exists. Naturally this will correspond to a form of induction in the proof of well-definedness.

[10] Notice that this is not an issue in the presence of arithmetical comprehension, i.e. in ACA_0, but in that case logical complexity of defined sets is not a stable notion: all of arithmetical comprehension reduces to Π^0_1-comprehension.

▶ **Proposition 4.5** (RCA_0)**.** *Suppose t_i and $\vec{s}_i$ are as in Lemma 4.4. Any N-thread along $(t_i)_i$ is not progressing. Moreover, $\forall k.\exists m.$ any N-thread from t_k progresses $\leq m$ times.*

The main result of this subsection is:

▶ **Theorem 4.6.** *Let $t : \vec{\sigma} \Rightarrow \tau$ be a CT_n-coderivation. Then $\mathsf{RCA}_0 + I\Sigma^0_{n+2} \vdash t \in \mathsf{HR}_{\vec{\sigma}\to\tau}$.*

As well as using Proposition 4.5, this result relies crucially on the fact that we prove that CT-coderivations progress in RCA_0, Proposition 2.15 (itself from [11], allowing us to "substitute" the Δ^0_{n+2}-definition of a non-hereditarily-total branch from Definition 4.2 to obtain an argument using $I\Sigma^0_{n+2}$ overall.

▶ **Corollary 4.7.** HR *is a model of CT.*

4.3 Interpretation of CT_n into T_{n+1}

We may now realise our model-theoretic results as bona fide interpretations of fragments of CT into fragments of T. As a word of warning, coterms of CT in this section, when operating inside T, should formally be understood by their Gödel codes, i.e. in this section T is "one meta-level higher" than CT. Until now we have been formalising the metatheory of CT within second-order arithmetic, and so arithmetising its syntax as natural numbers. Since we will here invoke program extraction from these fragments of arithmetic to fragments of T to interpret CT, the same coding carries over. At the risk of confusion, we shall suppress this formality henceforth.

▶ **Theorem 4.8.** *If $CT_n \vdash s = t$ then $T_{n+1} \vdash s \approx t$.*

The main idea here is that our formalisation of the HR model within arithmetic allows us to prove the following *reflection principle* in $\mathsf{RCA}_0 + I\Sigma^0_2$:

$$\forall P \,(\text{if "}P \text{ is a } CT_n \text{ proof of } s = t\text{" then } \exists d : s \approx t)$$

Since this statement is Π^0_2, we may apply program extraction, Proposition 2.14, to indeed witness the required derivation d within T_{n+1}, as required.

▶ **Corollary 4.9.** *If $t : \vec{N} \Rightarrow N$ is a progressing coterm of CT_n, then there is a T_{n+1}-term $t : \vec{N} \to N$ such that $t'^{\mathfrak{N}} = t^{\mathfrak{N}}$.*

5 Further results

In this section we shall give some further rewriting-theoretic results related to the system CT we have presented.

5.1 Continuity at type 2

It is well-known that the type 2 functionals of T are *continuous*, in the sense that any type 1 function input is only queried a finite number of times, e.g. [38, 32, 39]. For the case of CT, we may actually formalise a variation of the classical argument of [38] within second-order arithmetic, extending the simulation of CT coterms within T to type 2 functionals. For the sake of brevity, we shall not refine our exposition by type level in this subsection.

Let us fix a CT coderivation $t : \vec{\sigma} \Rightarrow N$ s.t. each $\sigma_i = N_1 \to \cdots \to N_{k_i} \to N$, and let us henceforth work in ACA_0, distinguishing second-order variables $f_i : \mathbb{N}^{k_i} \to \mathbb{N}$, intuitively representing the inputs for t. Within CT, introduce new (uninterpreted) constant symbols $\underline{f}_i : N_1 \to \cdots \to N_{k_i} \to N$ for each σ_i, and new reduction steps:

$$\underline{f}_i \, \underline{n}_1 \ldots \underline{n}_{k_i} \rightsquigarrow \underline{f_i(n_1, \ldots, n_{k_i})} \tag{6}$$

Notice that reduction is now still semi-recursive in the oracles $\vec{f}$, i.e. $\rightsquigarrow, \rightsquigarrow^*, \approx$ are now $\Sigma^0_1(\vec{f})$. To save the effort of reproving our confluence results from Section 3 with these new oracle symbols, we shall simply henceforth assume a suitable consistency principle:

$$\mathrm{UNF}_N \quad : \quad \forall m, n. (\underline{m} \approx \underline{n} \supset m = n)$$

Note that, since this is a true Π^0_1 statement (by meta-level reasoning), it carries no computational content and adding it to ACA_0 still admits extraction into T (see, e.g., [24]).[11] From here, we define $\mathsf{HR}^{\vec{f}}_\sigma$ just as HR_σ, but allowing coterms to include the symbols $\underline{\vec{f}}$. Since each HR_σ is arithmetical in $\rightsquigarrow$, we have that each $\mathsf{HR}^{\vec{f}}_\sigma$ is arithmetical in our extended reduction relation, so with free second-order variables $\vec{f}$. Note in particular that we have that each $\underline{f}_i \in \mathsf{HR}^{\vec{f}}_{\sigma_i}$, thanks to (6) above. By adapting our approach from Section 4, we may show:

▶ **Theorem 5.1** ($\mathsf{ACA}_0 + \mathrm{UNF}_N$). $\forall \vec{f}. t\, \underline{\vec{f}} \in \mathsf{HR}^{\vec{f}}_N$

Expanding out this result we have that $\mathsf{ACA}_0 + \mathrm{UNF}_N \vdash \forall \vec{f}. \exists n. t\, \vec{f} \approx \underline{n}$. Note that this yields the required syntactic continuity property: since any $\approx$-sequence is finite, we may compute $t(\vec{f})$ by querying each f_i only finitely many times. From here, by applying a relativised version of program extraction (see, e.g., [24]), we obtain a strengthening of our simulation of CT-coterms by T terms to type 2 (stated without refinement to type level):

▶ **Corollary 5.2.** *If t is a level 2 coterm of CT, then there is a T term t' s.t. $t'^{\mathfrak{N}} = t^{\mathfrak{N}}$.*

5.2 A "term model" à la Tait and strong normalisation

It is an immediate consequence of our results that CT-coterms are *weakly normalising*. Namely, by an induction on type (using confluence for the base case, at type N), we may show that each $t \in \mathsf{HR}$ is weakly normalising. Thus, by Theorem 4.6, we have:

▶ **Proposition 5.3.** *Each closed CT coterm is weakly normalising. Moreover, any CT_n coterm is provably weakly normalising inside* $\mathsf{RCA}_0 + I\Sigma^0_{n+2}$.

In this section we will go further and show that CT-coterms are actually *strongly normalising*, just like T-terms. For the sake of brevity, we will not formalise our exposition within arithmetic. We will define a minimal "coterm model" in a similar way to Tait's term models of sytem T [36]. This is complementary to our development of HR: while that structure was an "over-approximation" of the language of CT, the structure we are about to define is an "under-approximation" , by virtue of its definition. Naturally, the point is to show that the approximation is, in fact, tight.

▶ **Definition 5.4** (Convertibility). *We define the following sets of closed CT-coterms:*

- $\mathcal{C}_N := \{t : N \mid t \text{ is strongly normalising}\}$.
- $\mathcal{C}_{\sigma\to\tau} := \{t : \sigma \to \tau \mid \forall s \in \mathcal{C}_\sigma . ts \in \mathcal{C}_\tau\}$.

By an induction on type, we establish suitable versions of Proposition 3.15 and the normalisation property for $\mathcal{C}$:

▶ **Proposition 5.5.** *We have the following:*

1. *If $t \in \mathcal{C}_{\sigma\to\tau}$ and $s \in \mathcal{C}_\sigma$ then $ts \in \mathcal{C}_\tau$. ($\mathcal{C}$ closed under application)*
2. *If $t \in \mathcal{C}_\tau$ and $t \rightsquigarrow t'$ then $t' \in \mathcal{C}_\tau$. ($\mathcal{C}$ closed under reduction)*
3. *If $t \in \mathcal{C}_\tau$ then t is strongly normalising. ($\mathcal{C} \subseteq \mathrm{SN}$)*

[11] The drawback of this approach is that it does not yield any bona fide interpretation of CT into T, which is why we chose to formalise a confluence argument for our main interpretation result.

Closure of $\rightsquigarrow$ under contexts is required for 2 and 3. Note that the strong normalisation condition for $\mathcal{C}_N$ is crucial to justify closure under reduction, (2), at base type N. In contrast, for HR_N we only asked for *conversion* to a numeral, and so the analogous property of closure under conversion was a consequence of symmetry.

Let us call a coterm t **neutral** if, for any s, any redex of ts is either entirely in t or entirely in s. We also have the following expected characterisation of convertibility by induction on type:

▶ **Lemma 5.6** (Convertibility lemma). *Let t be neutral. If $\forall t' \leftsquigarrow t.\ t' \in \mathcal{C}_\tau$, then $t \in \mathcal{C}_\tau$.*

As for classical proofs of strong normalisation for T, we must also make use of a sub-induction on the size of the complete reduction trees of elements of $\mathcal{C}$; recall that they are strongly normalising, by Proposition 5.5, and so have finite reduction trees by König's lemma,[12] since there are always only finitely many redexes.

Now we can go on to define a non-converting branch, just like we did for the standard model $\mathfrak{N}$ in Proposition 2.11 (non-total branch), and for HR in Definition 4.2 (non-hereditarily-total branch). As in the latter case, we need to prove well-definedness of such a branch, cf. Observation 4.1 and Proposition 4.3.

▶ **Proposition 5.7** (Preservation of convertibility). *Let $\vec{r} \in \mathcal{C}_{\vec{\rho}}$ and $\vec{s} \in \mathcal{C}_{\vec{\sigma}}$. We have:*[13]

- *If $s \in \mathcal{C}_\sigma$ then $\mathsf{id}\, s \in \mathcal{C}_\sigma$.*
- *If $r \in \mathcal{C}_\rho, s \in \mathcal{C}_\sigma$ and $t\,\vec{r}\,s\,r\,\vec{s} \in \mathcal{C}_\tau$ then $\mathsf{ex}\,t\,\vec{r}\,r\,s\,\vec{s} \in \mathcal{C}_\tau$.*
- *If $s \in \mathcal{C}_\sigma$ and $t\,\vec{s} \in \mathcal{C}_\tau$ then $\mathsf{wk}\,t\,\vec{s}\,s \in \mathcal{C}_\tau$.*
- *If $s \in \mathcal{C}_\sigma$ and $t\,\vec{s}\,s\,s \in \mathcal{C}_\tau$ then $\mathsf{cntr}\,t\,\vec{s}\,s \in \mathcal{C}_\tau$.*
- *If $t_0\,\vec{s} \in \mathcal{C}_\sigma$ and $\forall s \in \mathcal{C}_\sigma.\ t_1\,\vec{s}\,s \in \mathcal{C}_\tau$ then $\mathsf{cut}\,t_0\,t_1\,\vec{s} \in \mathcal{C}_\tau$.*
- *If $r \in \mathcal{C}_{\rho\to\sigma}$ and $t_0\,\vec{s} \in \mathcal{C}_\rho$ and $\forall s \in \mathcal{C}_\sigma.\ t_1\,\vec{s}\,s \in \mathcal{C}_\tau$ then $\mathsf{L}\,t_0\,t_1\,\vec{s}\,r \in \mathcal{C}_\tau$.*
- *If $\forall s \in \mathcal{C}_\sigma.\ t\,\vec{s}\,s \in \mathcal{C}_\tau$ then $\mathsf{R}\,t\,\vec{s} \in \mathcal{C}_{\sigma\to\tau}$.*
- *$0 \in \mathcal{C}_N$.*
- *If $s \in \mathcal{C}_N$ then $\mathsf{s}s \in \mathcal{C}_N$.*
- *If $s \in \mathcal{C}_N$ and $t_0\,\vec{s} \in \mathcal{C}_\tau$ then $\mathsf{cond}\,t_0\,t_1\,\vec{s}\,0 \in \mathcal{C}_\tau$.*
- *If $s \in \mathcal{C}_N$ and $t_1\,\vec{s}\,s \in \mathcal{C}_\tau$ then $\mathsf{cond}\,t_0\,t_1\,\vec{s}\,\mathsf{s}s \in \mathcal{C}_\tau$.*

This is proved by an induction on the reduction trees of $\vec{s}, s, \vec{r}, r$ (which, again, are strongly normalising), in most cases appealing directly to the convertibility lemma above. For the L case we rely on closure of $\mathcal{C}$ under application, cf. Proposition 5.5, and for the R case we must employ a sub-induction on the reduction tree of an input $s \in \mathcal{C}_\sigma$.

As a consequence of our results in Sections 3 and 4, observe that any $s \in \mathcal{C}_N$ reduces to a unique numeral. This is because $\mathcal{C}_N$ contains *only* CT-coterms, by definition, which are weakly normalising and confluent. From here we may establish the main result of this subsection:

▶ **Theorem 5.8** (Convertibility for CT). *Any CT-coderivation $t : \vec{\sigma} \Rightarrow \tau$ is in $\mathcal{C}_{\vec{\sigma}\to\tau}$.*

The proof constructs a "non-converting" branch similarly to Definition 4.2 (or the proof of Proposition 2.11). There is one subtlety, however, in the treatment of the cond case, requiring the uniqueness of normal forms for elements of $\mathcal{C}_N$. We obtain the required inputs for the premiss occurrences of N by an induction on the reduction tree of an input of the conclusion occurrence.

---

[12] Note that König's lemma is *equivalent* to arithmetical comprehension, i.e. ACA_0, already over RCA_0 (cf., e.g., [34]).

[13] All rules have type as presented in Figure 1.

Since $\mathcal{C}$ is closed under application, Proposition 5.5, we inherit $\mathcal{C}$ membership for all CT-coterms. Since elements of $\mathcal{C}$ are strongly normalising, again Proposition 5.5, and since reduction is confluent, Theorem 3.4, we finally have:

▶ **Corollary 5.9** (Strong normalisation for CT)**.** *Any closed CT coterm strongly normalises to a unique normal form.*

6 Conclusions

In this work we gave an interpretation of a theory of level n circular derivations (CT_n) into level $n+1$ T (T_{n+1}), by formalising models of CT within fragments of arithmetic and applying program extraction. This result is optimal by a converse result from parallel work [12]. In particular, CT_n and T_{n+1} are *equi-consistent*. We also showed confluence, strong normalisation, and continuity at type 2 for CT-coterms.

In future work it would be interesting to establish results on Curry-Howard aspects of our underlying type systems, establishing forms of cut-elimination and relationships with infinitary lambda-calculi. Ideas from [4, 15, 3] may prove useful to this effect.

References

1. Bahareh Afshari and Graham E. Leigh. Cut-free completeness for modal mu-calculus. 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.2017.8005088`.
2. Jeremy Avigad and Solomon Feferman. Gödel's functional ("dialectica") interpretation. *Handbook of Proof Theory*, 137:337–405, 1998.
3. David Baelde, Amina Doumane, Denis Kuperberg, and Alexis Saurin. Bouncing threads for infinitary and circular proofs. *CoRR*, abs/2005.08257, 2020. `arXiv:2005.08257`.
4. David Baelde, Amina Doumane, and Alexis Saurin. Infinitary proof theory: the multiplicative additive case. In *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29–September 1, 2016, Marseille, France*, pages 42:1–42:17, 2016. `doi:10.4230/LIPIcs.CSL.2016.42`.
5. Stefano Berardi and Makoto Tatsuta. Equivalence of inductive definitions and cyclic proofs under arithmetic. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20–23, 2017*, pages 1–12, 2017. `doi:10.1109/LICS.2017.8005114`.
6. James Brotherston, Dino Distefano, and Rasmus Lerchedahl Petersen. Automated cyclic entailment proofs in separation logic. In *CADE-23 – 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31–August 5, 2011. Proceedings*, pages 131–146, 2011. `doi:10.1007/978-3-642-22438-6_12`.
7. James Brotherston, Nikos Gorogiannis, and Rasmus L. Petersen. A generic cyclic theorem prover. In *Programming Languages and Systems – 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11–13, 2012. Proceedings*, pages 350–367, 2012. `doi:10.1007/978-3-642-35182-2_25`.
8. James Brotherston and Alex Simpson. Complete sequent calculi for induction and infinite descent. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10–12 July 2007, Wroclaw, Poland, Proceedings*, pages 51–62, 2007. `doi:10.1109/LICS.2007.16`.
9. James Brotherston and Alex Simpson. Sequent calculi for induction and infinite descent. *J. Log. Comput.*, 21(6):1177–1216, 2011. `doi:10.1093/logcom/exq052`.
10. Samuel R. Buss, editor. *Handbook of Proof Theory*. Studies in Logic and the Foundations of Mathematics 137. Elsevier, 1998.

11 Anupam Das. On the logical complexity of cyclic arithmetic. *Log. Methods Comput. Sci.*, 16(1), 2020. doi:10.23638/LMCS-16(1:1)2020.

12 Anupam Das. A circular version of Gödel's T and its abstraction complexity, 2021. arXiv:2012.14421.

13 Anupam Das, Amina Doumane, and Damien Pous. Left-handed completeness for kleene algebra, via cyclic proofs. In Gilles Barthe, Geoff Sutcliffe, and Margus Veanes, editors, *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16–21 November 2018*, volume 57 of *EPiC Series in Computing*, pages 271–289. EasyChair, 2018. URL: https://easychair.org/publications/paper/SDqf.

14 Anupam Das and Damien Pous. A cut-free cyclic proof system for Kleene algebra. In *Automated Reasoning with Analytic Tableaux and Related Methods – 26th International Conference, TABLEAUX 2017, Brasília, Brazil, September 25–28, 2017, Proceedings*, pages 261–277, 2017. doi:10.1007/978-3-319-66902-1_16.

15 Anupam Das and Damien Pous. Non-wellfounded proof theory for (kleene+action)(algebras+lattices). In Dan R. Ghica and Achim Jung, editors, *27th EACSL Annual Conference on Computer Science Logic, CSL 2018, September 4–7, 2018, Birmingham, UK*, volume 119 of *LIPIcs*, pages 19:1–19:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.CSL.2018.19.

16 Christian Dax, Martin Hofmann, and Martin Lange. A proof system for the linear time μ-calculus. In S. Arun-Kumar and Naveen Garg, editors, *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13–15, 2006, Proceedings*, volume 4337 of *Lecture Notes in Computer Science*, pages 273–284. Springer, 2006. doi:10.1007/11944836_26.

17 Christian Dax, Martin Hofmann, and Martin Lange. A proof system for the linear time μ-calculus. In S. Arun-Kumar and Naveen Garg, editors, *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13–15, 2006, Proceedings*, volume 4337 of *Lecture Notes in Computer Science*, pages 273–284. Springer, 2006. doi:10.1007/11944836_26.

18 Amina Doumane. Constructive completeness for the linear-time μ-calculus. 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.2017.8005075.

19 Jérôme Fortier and Luigi Santocanale. Cuts for circular proofs: semantics and cut-elimination. In *Computer Science Logic 2013 (CSL 2013), September 2–5, 2013, Torino, Italy*, pages 248–262, 2013. doi:10.4230/LIPIcs.CSL.2013.248.

20 Petr Hájek and Pavel Pudlák. *Metamathematics of First-Order Arithmetic*. Perspectives in mathematical logic. Springer, 1993. URL: http://www.springer.com/mathematics/book/978-3-540-63648-9.

21 J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, USA, 1986.

22 Denis R. Hirschfeldt. *Slicing the truth: On the computable and reverse mathematics of combinatorial principles*. World Scientific, 2014.

23 S.C. Kleene. *Introduction to Metamathematics*. Bibliotheca Mathematica. North Holland, 7 edition, 1980.

24 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.

25 Leszek Aleksander Kolodziejczyk, Henryk Michalewski, Pierre Pradic, and Michal Skrzypczak. The logical strength of büchi's decidability theorem. *Log. Methods Comput. Sci.*, 15(2), 2019. doi:10.23638/LMCS-15(2:16)2019.

26 Denis Kuperberg, Laureline Pinault, and Damien Pous. Cyclic Proofs, System T, and the Power of Contraction. *Proceedings of the ACM on Programming Languages*, 2021. doi:10.1145/3434282.

27 John Longley and Dag Normann. *Higher-Order Computability*. Theory and Applications of Computability. Springer, 2015. doi:10.1007/978-3-662-47992-6.

28 Damian Niwinski and Igor Walukiewicz. Games for the mu-calculus. *Theor. Comput. Sci.*, 163(1&2):99–116, 1996. doi:10.1016/0304-3975(95)00136-0.

29 Charles Parsons. On n-quantifier induction. *The Journal of Symbolic Logic*, 37(3):466–482, 1972.

30 Frank Pfenning. A proof of the church-rosser theorem and its representation in a logical framework. Technical report, Carnegie-Mellon University, Pittsburgh. Department of Computer Science., 1992.

31 Luigi Santocanale. A calculus of circular proofs and its categorical semantics. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8–12, 2002, Proceedings*, volume 2303 of *Lecture Notes in Computer Science*, pages 357–371. Springer, 2002. doi:10.1007/3-540-45931-6_25.

32 B. Scarpellini. A model for barrecursion of higher types. *Compositio Mathematica*, 23(1):123–153, 1971. URL: http://eudml.org/doc/89072.

33 Alex Simpson. Cyclic arithmetic is equivalent to Peano arithmetic. In *Foundations of Software Science and Computation Structures – 20th International Conference, FOSSACS 2017, Proceedings*, pages 283–300, 2017. doi:10.1007/978-3-662-54458-7_17.

34 Stephen G. Simpson. *Subsystems of second order arithmetic*, volume 1. Cambridge University Press, 2009.

35 Thomas Studer. On the proof theory of the modal mu-calculus. *Stud Logica*, 89(3):343–363, 2008. doi:10.1007/s11225-008-9133-6.

36 William W. Tait. Intensional interpretations of functionals of finite type I. *J. Symb. Log.*, 32(2):198–212, 1967. doi:10.2307/2271658.

37 Terese. *Term rewriting systems*, volume 55 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 2003.

38 Anne S. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*. Lecture Notes in Mathematics. Springer, 1 edition, 1973. URL: http://gen.lib.rus.ec/book/index.php?md5=5E0718442C0178C4B23065E54AC7889C.

39 Chuangjie Xu. A syntactic approach to continuity of T-definable functionals. *Logical Methods in Computer Science*, Volume 16, Issue 1, 2020. doi:10.23638/LMCS-16(1:22)2020.

A Further material for Section 4

Proof of Proposition 4.3. Let us write $\mathrm{Gen}(i,(t_0,\vec{s}_0),(t_i,\vec{s}_i))$ for "t_i and $\vec{s}_i$ are the i^{th} sequent and input tuple generated by t_0 and $\vec{s}_0$". Notice that the construction of t_i and $\vec{s}_i$ itself is recursive in HR_n, t_0 and $\vec{s}_0$, and so Gen is certainly recursion-theoretically $\Delta^0_{n+2}(t_0,\vec{s}_0)$, by appealing to Fact 3.14. To formally prove that Gen is Δ^0_{n+2} *inside* our theory, it suffices to show determinism:

$$\forall i.\forall (t_i,\vec{s}_i),(t'_i,\vec{s}'_i).\left(\begin{array}{rl} & \mathrm{Gen}(i,(t_0,\vec{s}_0),(t_i,\vec{s}_i)) \wedge \mathrm{Gen}(i,(t_0,\vec{s}_0),(t'_i,\vec{s}'_i)) \\ \implies & t_i = t'_i \wedge \vec{s}_i = \vec{s}'_i \end{array}\right)$$

Writing Gen syntactically as a Σ^0_{n+2} formula, the above may be directly proved by Π^0_{n+2}-induction on i, appealing to the cases of Definition 4.2 above.

It remains to show that the construction is total, i.e. that each $(t_i,\vec{s}_i)$ actually exists. In fact we will simultaneously prove this and the inductive invariant of the construction, so the formula,

$$\exists (t_i,\vec{s}_i).(\mathrm{Gen}(i,(t_0,\vec{s}_0),(t_i,\vec{s}_i)) \wedge t_i\,\vec{s}_i \notin \mathsf{HR}_{\tau_i}) \tag{7}$$

by induction on i. Note that, since $\mathsf{lev}(\tau_i) \leq n$ we have that HR_{τ_i} is Π^0_{n+1} by Fact 3.14, and so $t_i \vec{s}_i \notin \mathsf{HR}_{\tau_i}$ is Σ^0_{n+1}, whereas $\mathrm{Gen}(i, (t_0, \vec{s}_0), (t_i, \vec{s}_i))$ is Δ^0_{n+2} as already mentioned. Thus the inductive invariant in (7) is indeed Σ^0_{n+2}.

First, to justify (1), let us consider the possible initial sequents:

- For the 0 rule: we have $0 \in \mathsf{HR}_N$ by definition;
- For the s rule: if $t \in \mathsf{HR}_N$, then $t \approx \underline{n}$ for some $n \in \mathbb{N}$, by definition of HR_N, and so also $\mathsf{s}t \approx \mathsf{s}\underline{n}$, by closure of $\approx$ under contexts. Hence $\mathsf{s}t \in \mathsf{HR}_N$.
- For an id_σ rule: if $s \in \mathsf{HR}_\sigma$ then $\mathsf{id}\, s \approx s$ by id reduction. Hence $\mathsf{id}\, s \in \mathsf{HR}_\sigma$.

Now, the base case, for $i = 0$, follows by the assumption on t_0 and $\vec{s}_0$, so let us assume that $\mathrm{Gen}(i, (t_0, \vec{s}_0), (t_i, \vec{s}_i))$ and $t_i\, \vec{s}_i \notin \mathsf{HR}_{\tau_i}$. We will witness the existential of the inductive invariant with the coderivation t_{i+1} and inputs $\vec{s}_{i+1}$ as given in Definition 4.2 above (justifying their existence when necessary), showing $t_{i+1}\, \vec{s}_{i+1} \notin \mathsf{HR}_{\tau_{i+1}}$. We shall also adopt the same notation for inputs and types as in Definition 4.2.

For (2), the wk case, we have:

	$t_i\, \vec{s}_i \notin \mathsf{HR}_\tau$	by inductive hypothesis
$\therefore$	$\mathsf{wk}\, t\, \vec{s}\, s \notin \mathsf{HR}_{\tau_i}$	by definitions
$\therefore$	$t\, \vec{s} \notin \mathsf{HR}_\tau$	by $\rightsquigarrow_{\mathsf{wk}}$ and closure of HR_τ under $\approx$
$\therefore$	$t_{i+1}\, \vec{s}_{i+1} \notin \mathsf{HR}_{\tau_{i+1}}$	by definitions

For (3), the ex case, we have:

	$t_i\, \vec{s}_i \notin \mathsf{HR}_{\tau_i}$	by inductive hypothesis
$\therefore$	$\mathsf{ex}\, t\, \vec{r}\, r\, s\, \vec{s} \notin \mathsf{HR}_\tau$	by definitions
$\therefore$	$t\, \vec{r}\, s\, r\, \vec{s} \notin \mathsf{HR}_\tau$	by $\rightsquigarrow_{\mathsf{ex}}$ and $\because$ HR_τ closed under $\approx$
$\therefore$	$t_{i+1}\, \vec{s}_{i+1} \notin \mathsf{HR}_{\tau_{i+1}}$	by definitions

For (4), the cntr case, we have:

	$t_i\, \vec{s}_i \notin \mathsf{HR}_{\tau_i}$	by inductive hypothesis
$\therefore$	$\mathsf{cntr}\, t\, \vec{s}\, s \notin \mathsf{HR}_\tau$	by definitions
$\therefore$	$t\, \vec{s}\, s\, s \notin \mathsf{HR}_\tau$	by $\rightsquigarrow_{\mathsf{cntr}}$ and $\because$ HR_τ closed under $\approx$
$\therefore$	$t_{i+1}\, \vec{s}_{i+1} \notin \mathsf{HR}_{\tau_{i+1}}$	by definitions

For (5), the cut case, assume without loss of generality that $t\, \vec{s} \in \mathsf{HR}_\tau$. We have:

	$t_i\, \vec{s}_i \notin \mathsf{HR}_{\tau_i}$	by inductive hypothesis
$\therefore$	$\mathsf{cut}\, t\, t' \vec{s} \notin \mathsf{HR}_\tau$	by definitions
$\therefore$	$t' \vec{s}\, (t\, \vec{s}) \notin \mathsf{HR}_\tau$	by $\rightsquigarrow_{\mathsf{cut}}$ and $\because$ HR_τ closed under $\approx$
$\therefore$	$t_{i+1}\, \vec{s}_{i+1} \notin \mathsf{HR}_{\tau_{i+1}}$	by definitions

For (6), the L case, assume without loss of generality that $t\, \vec{s} \in \mathsf{HR}_\tau$, and so also $s\, (t\, \vec{s}) \in \mathsf{HR}_\sigma$ by Proposition 3.15. We have:

	$t_i\, \vec{s}_i \notin \mathsf{HR}_{\tau_i}$	by inductive hypothesis
$\therefore$	$\mathsf{L}\, t\, t' \vec{s}\, s \notin \mathsf{HR}_\tau$	by definitions
$\therefore$	$t' \vec{s}\, (s\, (t\, \vec{s})) \notin \mathsf{HR}_\tau$	by $\rightsquigarrow_{\mathsf{L}}$ and $\because$ HR_τ closed under $\approx$
$\therefore$	$t_{i+1}\, \vec{s}_{i+1} \notin \mathsf{HR}_{\tau_{i+1}}$	by definitions

For (7), the R case, we have:

	$t_i\,\vec{s}_i \notin \mathsf{HR}_{\tau_i}$	by inductive hypothesis
$\therefore$	$\mathsf{R}\,t\,\vec{s} \notin \mathsf{HR}_{\sigma\to\tau}$	by definitions
$\therefore$	$\exists s' \in \mathsf{HR}_\sigma.\ \mathsf{R}\,t\,\vec{s}\,s' \notin \mathsf{HR}_\tau$	by definition of $\mathsf{HR}_{\sigma\to\tau}$
$\therefore$	$\exists s' \in \mathsf{HR}_\sigma.\ t\,\vec{s}\,s' \notin \mathsf{HR}_\tau$	by $\rightsquigarrow_{\mathsf{R}}$ and $\because$ HR_τ closed under $\approx$
$\therefore$	$t\,\vec{s}\,s \notin \mathsf{HR}_\tau$	$\because$ s is well-defined by Σ^0_{n+1}-minimisation
$\therefore$	$t_{i+1}\,\vec{s}_{i+1} \notin \mathsf{HR}_{\tau_{i+1}}$	by definitions

In the penultimate step, note that we have from the inductive hypothesis $\exists s(s \in \mathsf{HR}_\sigma \wedge t\,\vec{s}\,s \notin \mathsf{HR}_\tau)$, where $\mathsf{lev}(\sigma) < n$ and $\mathsf{lev}(\tau) \leq n$. Thus $(s \in \mathsf{HR}_\sigma \wedge t\,\vec{s}\,s \notin \mathsf{HR}_\tau)$ is indeed Σ^0_{n+1}, by Fact 3.14, and so Σ^0_{n+1}-minimisation applies.

For (8), the cond case, note by the inductive hypothesis we have $r \in \mathsf{HR}_N$ so by definition of HR_N and confluence, we have that r converts to a unique numeral. Thus the two cases considered by the definition of t_{i+1} and $\vec{s}_{i+1}$ are exhaustive and exclusive, and we consider each separately.

If $r \approx 0$ then we have:

	$t_i\,\vec{s}_i \notin \mathsf{HR}_{\tau_i}$	by inductive hypothesis
$\therefore$	$\mathsf{cond}\,t\,t'\,\vec{s}\,r \notin \mathsf{HR}_\tau$	by definitions
$\therefore$	$\mathsf{cond}\,t\,t'\,\vec{s}\,0 \notin \mathsf{HR}_\tau$	by assumption and $\because$ HR_τ closed under $\approx$
$\therefore$	$t\,\vec{s} \notin \mathsf{HR}_\tau$	by $\rightsquigarrow_{\mathsf{cond}}$ and $\because$ HR_τ closed under $\approx$
$\therefore$	$t_{i+1}\,\vec{s}_{i+1} \notin \mathsf{HR}_{\tau_{i+1}}$	by definitions

If $r \approx \mathsf{s}\underline{n}$ then we have:

	$t_i\,\vec{s}_i \notin \mathsf{HR}_{\tau_i}$	by inductive hypothesis
$\therefore$	$\mathsf{cond}\,t\,t'\,\vec{s}\,r \notin \mathsf{HR}_\tau$	by definitions
$\therefore$	$\mathsf{cond}\,t\,t'\,\vec{s}\,\mathsf{s}\underline{n} \notin \mathsf{HR}_\tau$	by assumption and $\because$ HR_τ closed under $\approx$
$\therefore$	$t'\,\vec{s}\,\underline{n} \notin \mathsf{HR}_\tau$	by $\rightsquigarrow_{\mathsf{cond}}$ and $\because$ HR_τ closed under $\approx$
$\therefore$	$t_{i+1}\,\vec{s}_{i+1} \notin \mathsf{HR}_{\tau_{i+1}}$	by definitions

This concludes the proof. ◀

Proof of Proposition 4.5. We shall prove only the "moreover" clause, the former following a fortiori. First, suppose we have a (finite) N-thread $(N^i)^l_{i=k}$ beginning at t_k. Let $s_i \in \vec{s}_i$ be the corresponding input of N^i for $1 \leq i \leq l$, and let each $r_i \approx \underline{n_i}$, for unique $n_i \in \mathbb{N}$, by definition of HR_N and confluence. Letting m be the number of times that $(N^i)^l_{i=1}$ progresses, we may show by induction on l that $n_l \leq n_k - m$, using Lemma 4.4 for the inductive steps.

Now, to prove the "moreover" statement, fix some k and let $\vec{N}^k \subseteq \vec{\sigma}_k$ exhaust the N occurrences in $\vec{\sigma}_k$. Let $\vec{r}_k \subseteq \vec{s}_k$ be the corresponding inputs, and write $\vec{n}_k$ for the unique natural numbers such that each $r_{ki} \approx \underline{n_{ki}}$, by definition of HR_N and confluence. We may now simply set $m := \max \vec{n}_k$, whence no thread from t_k may progress more than m times by the preceding paragraph. ◀

Proof of Theorem 4.6. First, by Proposition 2.15 (from [11]), we have that RCA_0 proves that t is progressing. Consequently RCA_0 proves that, for any branch $(t_i)_i$, there is some k s.t. there are arbitrarily often progressing finite threads beginning from t_k:[14]

$$\exists k.\forall m.\ \text{there is a (finite) } N\text{-thread from } t_k \text{ progressing} > m \text{ times} \tag{8}$$

[14] The argument for this is similar to that of Proposition 6.2 from [11].

Note that this statement is purely arithmetical in $(t_i)_i$ and so, if $(t_i)_i$ is Δ^0_{n+2}-well-defined, then in fact $\mathsf{RCA}_0 + I\Sigma^0_{n+2}$ proves (8), by conservativity over $I\Sigma_{n+2}((t_i)_i)$ and then substitution of the Δ_{n+2}-definition of $(t_i)_i$.

Now, working inside $\mathsf{RCA}_0 + I\Sigma^0_{n+2}$, suppose for contradiction that $\vec{s} \in \mathsf{HR}_{\vec{\sigma}}$ s.t. $t\,\vec{s} \notin \mathsf{HR}_\tau$. By Proposition 4.3, we can Δ^0_{n+2}-well-define the branch $(t_i)_i$ generated by t and $\vec{s}$. Thus we indeed have (8), contradicting Proposition 4.5. ◀

Proof sketch of Theorem 4.8. Let us work inside $\mathsf{RCA}_0 + I\Sigma^0_{n+2}$. By Theorem 4.6 we have that $s, t \in \mathsf{HR}_\sigma$, so suppose that $CT_n \vdash s = t$ (which is a Σ^0_1 relation). Now, invoking Lemma 3.17 and by verifying the other axioms for FARs in general, we indeed have that $s \approx t$, by Σ^0_1-induction on the CT_n proof of $s = t$.

Now, invoking the extraction theorem, Proposition 2.14, for the above paragraph, we can extract a T_{n+1}-term $d(\cdot)$ witnessing the following "reflection" principle:

$$T_{n+1} \vdash \text{“}P \text{ is a } CT_n \text{ proof of } s = t\text{”} \supset d(P) : s \approx t$$

We may duly substitute a concrete CT_n proof P of $s = t$ into the above principle to conclude that $T_{n+1} \vdash s \approx t$, as required. ◀

B Further material for Section 5

Proof sketch of Theorem 5.1. The argument is essentially the same as that for Theorem 4.6. Assuming otherwise, for contradiction, we may generate a non-hereditarily-total branch just as in Definition 4.2, and its well-definedness is shown just as in Proposition 4.3. Note that all induction/minimisation used is in fact arithmetical in $\rightsquigarrow$ and $\mathsf{HR}^{\vec{f}}_\sigma$, so the branch is indeed $\Delta^0_{n+2}(\vec{f})$-well-defined (for n the maximal type level in t).

Since we no longer concern ourselves with the refinement of type levels, the remainder of the argument is actually simpler than that of Section 4. Instead of dealing with the arithmetical approximation of progressiveness, we may immediately access the generated non-total branch *as a set*, thanks to the availability of arithmetical comprehension in ACA_0. We also have a suitable version of Lemma 4.4 for $\mathsf{HR}^{\vec{f}}_N$, this time using UNF_N instead of confluence, and so the appropriate contradiction of the well-ordering property of $\mathbb{N}$ is readily obtained. ◀

▶ **Observation B.1.** *If $s \in \mathcal{C}_N$ then s reduces to a unique numeral.*

Proof. Since $\mathcal{C}_N$ contains *only* CT-coterms, we have as a special case of Theorem 4.6 that $s \approx \underline{n}$ for some $n \in \mathbb{N}$. By confluence, we have that n is unique and furthermore $s \rightsquigarrow^* \underline{n}$. ◀

Proof of Theorem 5.8. Suppose for contradiction we have $\vec{s} \in \mathcal{C}_{\vec{\sigma}}$ such that $t\,\vec{s} \notin \mathcal{C}_\tau$. We define a branch $(t_i : \vec{\sigma}_i \Rightarrow \tau_i)_i$ of t and inputs $\vec{s}_i \in \mathcal{C}_{\vec{\sigma}_i}$ s.t. $t_i\,\vec{s}_i \notin \mathcal{C}_{\tau_i}$ by induction on i just like in Definition 4.2 (or the proof of Proposition 2.11). The only difference is that we use Proposition 5.7 above for preservation in $\mathcal{C}$ rather than the analogous closure properties for HR (or $\mathfrak{N}$).

There is one subtlety, which is the treatment of the cond case. Suppose we have a regular progressing coderivation,

$$\mathsf{cond}\ \dfrac{\begin{matrix} t \\ \vec{\sigma} \Rightarrow \tau \end{matrix} \qquad \begin{matrix} t' \\ \vec{\sigma}, N \Rightarrow \tau \end{matrix}}{\vec{\sigma}, N \Rightarrow \tau}$$

and $\vec{s}_i = (\vec{s}, s)$ with $\vec{s} \in \mathcal{C}_{\vec{\sigma}}$, $s \in \mathcal{C}_N$ and $\mathsf{cond}\,t\,t'\vec{s}\,s \notin \mathcal{C}_\tau$. Since $s \in \mathcal{C}_N$ we have from Observation B.1 that s reduces to a unique numeral $\underline{n}$. We will show that,

- if $n = 0$ then $t\,\vec{s} \notin \mathcal{C}_\tau$; and,
- if $n = m + 1$ then there is some $r \in \mathcal{C}_N$ reducing to $\underline{m}$ with $t'\vec{s}\,r \notin \mathcal{C}_\tau$;

by induction on $\mathrm{RedTree}(\vec{s}) + \mathrm{RedTree}(s)$. By the conversion lemma, Lemma 5.6, there must be a reduction from $\mathsf{cond}\,t\,t'\,\vec{s}\,s$ not reaching $\mathcal{C}_\tau$. Let us consider the possible cases:

- If $s = 0$ and $\mathsf{cond}\,t\,t'\,\vec{s}\,s \rightsquigarrow t\,\vec{s} \notin \mathcal{C}_\tau$ then we are done.
- If $s = \mathsf{s}r$ and $\mathsf{cond}\,t\,t'\,\vec{s}\,s \rightsquigarrow t'\vec{s}\,r \notin \mathcal{C}_\tau$ then we are done. (Note that such r must strongly normalise to $\underline{m}$, and so in particular $r \in \mathcal{C}_N$).
- If $\mathsf{cond}\,t\,t'\,\vec{s}\,s \rightsquigarrow \mathsf{cond} t\,t'\vec{s}'s' \notin \mathcal{C}_\tau$, then by the inductive hypothesis either,
  - $n = 0$ and $t\,\vec{s}' \notin \mathcal{C}_\tau$, so $t\,\vec{s} \notin \mathcal{C}_\tau$ by Proposition 5.5.(2); or,
  - $n = m + 1$ and there is some $r \in \mathcal{C}_N$ reducing to $\underline{m}$ s.t. $t'\,\vec{s}'\,r \notin \mathcal{C}_\tau$, so $t'\vec{s}\,r \notin \mathcal{C}_\tau$ by Proposition 5.5.(2).

From here, any progressing thread $(N^i)_{i \geq k}$ along $(t_i)_i$ yields a sequence of coterms $(r_i \in \mathcal{C}_N)_{i \geq k}$ that, under normalisation, induces an infinitely often descending sequence of natural numbers, yielding the required contradiction. ◀

Abstract Clones for Abstract Syntax

Nathanael Arkor
University of Cambridge, UK

Dylan McDermott
Reykjavik University, Iceland

Abstract

We give a formal treatment of simple type theories, such as the simply-typed λ-calculus, using the framework of abstract clones. Abstract clones traditionally describe first-order structures, but by equipping them with additional algebraic structure, one can further axiomatize second-order, variable-binding operators. This provides a syntax-independent representation of simple type theories. We describe multisorted second-order presentations, such as the presentation of the simply-typed λ-calculus, and their clone-theoretic algebras; free algebras on clones abstractly describe the syntax of simple type theories quotiented by equations such as β- and η-equality. We give a construction of free algebras and derive a corresponding induction principle, which facilitates syntax-independent proofs of properties such as adequacy and normalization for simple type theories. Working only with clones avoids some of the complexities inherent in presheaf-based frameworks for abstract syntax.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Equational logic and rewriting; Theory of computation → Higher order logic; Theory of computation → Proof theory

Keywords and phrases simple type theories, abstract clones, second-order abstract syntax, substitution, variable binding, presentations, free algebras, induction, logical relations

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.30

Funding *Dylan McDermott*: Icelandic Research Fund project grant № 196323-053.

1 Introduction

The abstract concept of type theory is crucial in the study of programming languages. However, while it is generally appreciated that the concrete syntax associated to a type theory is peripheral to its fundamental structure, conventional techniques for working with type theories and proving properties thereof are predominantly syntactic. The primary reason for this incongruity is that, though abstract frameworks for defining and reasoning about general classes of type theories have been developed (e.g. [14, 13, 5, 12, 21, 2, 3, 19], there called *second-order abstract syntax*), the mathematical prerequisites are significant and often appear unapproachable to those without a firm category theoretic background. This is regrettable, because these general techniques alleviate much of the rote associated to syntactic proofs, such as those for adequacy, normalization, and the admissibility of substitution.

It so happens that there exists in the mathematical folklore an approach that is particularly well-suited to capturing the essential structure of simple type theories and yet requires essentially no experience with category theory to employ fruitfully: this is the formalism of *abstract clones* (often simply called *clones*) with algebraic structure. The structure of an abstract clone captures the notion of a context-indexed family of terms, closed under variable projection and substitution; equipping clones with algebraic structure permits the expression of variable-binding operators, like the λ-abstraction operator familiar from λ-calculi. It is known amongst cognoscenti that abstract clones might be employed for this purpose: for instance, Fiore, Plotkin, and Turi [16] proved that abstract clones are equivalent to their notion of *substitution monoids*, which represent families of (unityped) terms with an associated capture-avoiding substitution operation; later, Fiore and Mahmoud [32, 15] proved that abstract clones with algebraic structure are equivalent to the Σ-*monoids* of Fiore et al., which extend substitution monoids with second-order (i.e. variable binding) algebraic structure. In a separate line of inquiry, Hyland [24] uses abstract clones with algebraic structure to give a modern treatment of the unityped λ-calculus. However, it does not appear that abstract clones have previously been expressly proposed for the study of simple type theories (in fact, the definition of a typed abstract clone with algebraic structure is absent from the literature).

Here, we give an exposition of the use of abstract clones with algebraic structure in defining simple type theories and proving various of their properties. After setting up the relevant definitions (Section 2), we describe how simple type theories can be modelled by algebras of second-order presentations (Section 3). We then show that free algebras exist, giving an abstract description of the syntax of the type theory (Section 4). We derive an induction principle [30] that enables abstract reasoning about the syntax (Section 5), and show that this is powerful enough to prove non-trivial properties of type theories, in particular using logical relations (Section 6). We also compare the clone-theoretic framework to other approaches (Section 7). Though we do not expect our treatment to be surprising to experts familiar with prior categorical developments, it is an important perspective in the understanding of simple type theories and deserves explication.

Though we occasionally make reference to category theory throughout the paper, knowledge of category theory is not necessary to understand the content.

2 Abstract clones and first-order presentations

A typed (or *multisorted*) *abstract clone* [36], henceforth simply *clone*, encapsulates the structure of terms in simple contexts, closed under variables and substitution. Informally, for each context $x_1 : A_1, \ldots, x_n : A_n$ and type B, where A_1 to A_n are types (or *sorts*), a clone $\mathbf{X}$ specifies a set of terms $X(A_1, \ldots, A_n; B)$, each element of which is considered a term of type B in the context $x_1 : A_1, \ldots, x_n : A_n$. It also specifies terms var_i representing the projection of the variable x_i from the context, and functions $\mathsf{subst}_{\Gamma;A_n,\ldots,A_n;B} : X(A_1, \ldots, A_n; B) \times X(\Gamma; A_1) \times \cdots \times X(\Gamma; A_n) \to X(\Gamma; B)$ representing simultaneous substitution:

$$\begin{array}{rll} t \in X(A_1,\ldots,A_n;B) & \textit{represents} & x_1 : A_1,\ldots,x_n : A_n \vdash t : B \\ \mathsf{var}_i^{(A_1,\ldots,A_n)} \in X(A_1,\ldots,A_n;A_i) & \textit{represents} & x_1 : A_1,\ldots,x_n : A_n \vdash x_i : A_i \\ \mathsf{subst}_{\Gamma;A_1,\ldots,A_n;B}(t,u_1,\ldots,u_n) & \textit{represents} & \Gamma \vdash t\{x_1 \mapsto u_1,\ldots,x_n \mapsto u_n\} : B \end{array}$$

The clone $\mathbf{X}$ is required to satisfy laws expressing that (1) substituting variables for themselves does nothing; (2) applying a substitution to a variable results in the term corresponding to that variable in the substitution; and (3) substitution is associative.

▶ **Notation 1.** *We fix a set S of types (sorts). We denote by S^* the free monoid on S, i.e. lists of elements of S. Conceptually, contexts $x_1 : A_1, \ldots, x_n : A_n$ are given by elements $[A_1, \ldots, A_n] \in S^*$, since variable names carry no information. We write $\diamond \in S^*$ for the empty context, and Γ, Ξ for the concatenation of $\Gamma \in S^*$ and $\Xi \in S^*$. For contexts $\Gamma, \Delta \in S^*$, where $\Delta = [A_1, \ldots, A_n]$, we define $X(\Gamma; \Delta) = \prod_{i \leq n} X(\Gamma; A_i)$. We call the elements $\boldsymbol{\sigma} \in X(\Gamma; \Delta)$* substitutions*; a substitution is therefore a tuple $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_n)$ of terms $\sigma_i \in X(\Gamma; A_i)$.*

▶ **Definition 2.** *An S-sorted* clone *$\mathbf{X} = (X, \mathsf{var}, \mathsf{subst})$ consists of*

- *for each context $\Gamma \in S^*$ and sort $A \in S$, a set $X(\Gamma; A)$ of* terms*;*
- *for each context $\Gamma \in S^*$, a tuple $\mathsf{var}^{(\Gamma)} \in X(\Gamma; \Gamma)$ of* variables*;*

- *for each pair of contexts* $\Gamma, \Delta \in S^*$ *and sort* $A \in S$*, a* substitution function $\mathsf{subst}_{\Gamma;\Delta;A} : X(\Delta; A) \times X(\Gamma; \Delta) \to X(\Gamma; A)$*, which we write as* $t[\boldsymbol{\sigma}] = \mathsf{subst}_{\Gamma;\Delta;A}(t, \boldsymbol{\sigma})$*;*

such that

$$\mathsf{var}_i^{(A_1,\ldots,A_n)}[\boldsymbol{\sigma}] = \sigma_i \qquad \text{for each } \boldsymbol{\sigma} \in X(\Gamma; A_1, \ldots, A_n) \text{ and } i \leq n \tag{1}$$

$$t[\mathsf{var}^{(\Gamma)}] = t \qquad \text{for each } t \in X(\Gamma; A) \tag{2}$$

$$t[\sigma'_1[\boldsymbol{\sigma}], \ldots, \sigma'_m[\boldsymbol{\sigma}]] = (t[\boldsymbol{\sigma}'])[\boldsymbol{\sigma}] \qquad \text{for each } t \in X(\Xi; A),\ \boldsymbol{\sigma}' \in X(\Delta; \Xi),\ \boldsymbol{\sigma} \in X(\Gamma; \Delta) \tag{3}$$

A clone homomorphism $f : \mathbf{X} \to \mathbf{X}'$ *consists of a function* $f_{\Gamma;B} : X(\Gamma; B) \to X'(\Gamma; B)$ *for each context* $\Gamma \in S^*$ *and sort* $B \in S$*, such that the following hold, where* $\Delta = [A_1, \ldots, A_n] \in S^*$*:*

$$f_{\Delta;A_i}(\mathsf{var}_i^{(\Delta)}) = \mathsf{var}_i^{(\Delta)\prime} \qquad \text{for each } i \leq n$$

$$f_{\Gamma;B}(t[\boldsymbol{\sigma}]) = (f_{\Delta;B}(t))[f_{\Gamma;A_1}(\sigma_1), \ldots, f_{\Gamma;A_n}(\sigma_n)]' \quad \text{for each } t \in X(\Delta; B),\ \boldsymbol{\sigma} \in X(\Gamma; \Delta)$$

We write $\mathbf{Clone}(S)$ *for the category of* S*-sorted clones and homomorphisms.*

We extend every clone homomorphism $f : \mathbf{X} \to \mathbf{X}'$ to act on substitutions as follows, where $\Delta = [A_1, \ldots, A_n] \in S^*$:

$$f_{\Gamma;\Delta} : X(\Gamma; \Delta) \to X'(\Gamma; \Delta) \qquad\qquad f_{\Gamma;\Delta}(\boldsymbol{\sigma}) = (f_{\Gamma;A_1}(\sigma_1), \ldots, f_{\Gamma;A_n}(\sigma_n))$$

▶ **Example 3.** We denote by $\mathbf{Var}_S$ the S-sorted *clone of variables*, whose family of terms is given by $\mathsf{Var}_S(A_1, \ldots, A_n; B) = \{i \mid A_i = B\}$; whose variables are given by $\mathsf{var}_i^{(\Gamma)} = i$; and whose substitution is given by $i[\boldsymbol{\sigma}] = \sigma_i$. $\mathbf{Var}_S$ is the initial object in $\mathbf{Clone}(S)$: for any S-sorted clone $\mathbf{X}$, there is a unique homomorphism $\triangleright : \mathbf{Var}_S \to \mathbf{X}$ given by $\triangleright_{\Gamma;B}(i) = \mathsf{var}_i^{(\Gamma)}$.

▶ **Example 4.** The terms of any universal algebra [8] form a *monosorted* clone (i.e. an S-sorted clone for which S is a singleton $\{*\}$). The sets of terms, along with the variables and substitution function, exactly match the classical notions. For instance, monoids form a clone $\mathbf{Mon}$, where $\mathrm{Mon}(\underbrace{*, \ldots, *}_{n}; *)$ is the free monoid on n elements.

▶ **Example 5.** Let Ty be the set of sorts freely generated by a base type $\mathsf{b} \in \mathsf{Ty}$ and function types $(A \Rightarrow B) \in \mathsf{Ty}$ for $A, B \in \mathsf{Ty}$ (precisely, Ty is the free magma on $\{\mathsf{b}\}$). The terms of the simply typed λ-calculus (STLC) form a Ty-sorted clone $\mathbf{\Lambda}$. Consider terms generated by the following rules:

$$\frac{}{\Gamma, x : A, \Delta \vdash x : A} \qquad \frac{\Gamma \vdash f : A \Rightarrow B \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{app}\, f\, a : B} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.\, t : A \Rightarrow B}$$

(We write app to distinguish application of λ-terms from application of mathematical functions. We also use named variables for readability, identifying α-equivalent terms.) Capture-avoiding simultaneous substitution $t\{x_i \mapsto u_i\}_i$ of terms is defined in the usual way by recursion on t:

$$x_j\{x_i \mapsto u_i\}_i = u_j \qquad (\mathsf{app}\, f\, a)\{x_i \mapsto u_i\}_i = \mathsf{app}\,(f\{x_i \mapsto u_i\}_i)\,(a\{x_i \mapsto u_i\}_i)$$

$$(\lambda x : A.\, t)\{x_i \mapsto u_i\}_i = \lambda y : A.\, (t\{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n, x \mapsto y\})$$

The clone $\mathbf{\Lambda}$ has sets of terms $\Lambda(A_1, \ldots, A_n; B) = \{x_1 : A_1, \ldots, x_n : A_n \vdash t : B\}$, variables $\mathsf{var}_i^{(\Gamma)} = x_i$, and substitution $t[\boldsymbol{\sigma}] = t\{x_i \mapsto \sigma_i\}_i$.

There is a related Ty-sorted clone $\mathbf{\Lambda}_{\beta\eta}$ of STLC terms up to $\beta\eta$-equality, defined by quotienting the sets of terms associated to $\mathbf{\Lambda}$ by the equivalence relation $\approx_{\beta\eta}$, where $\Gamma \vdash t \approx_{\beta\eta} t' : A$ is the congruence relation generated by the following rules:

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{app}\,(\lambda x : A.\, t)\, u \approx_{\beta\eta} t\{x \mapsto u\} : B}\,(\beta) \qquad \frac{\Gamma \vdash t : A \Rightarrow B}{\Gamma \vdash t \approx_{\beta\eta} \lambda x : A.\, \mathsf{app}\, t\, x : A \Rightarrow B}\,(\eta)$$

▶ Remark 6. We shall only consider abstract clones with *sets* of types. However, as illustrated by the previous example, the types in a simple type theory often have algebraic structure themselves. By considering only the underlying set of types, the algebraic structure is forgotten. This simplifies the development, at the cost of some loss of expressivity. By specifying a (monosorted) clone of types, rather than a set, one recovers exactly the *simple type theories* of Arkor and Fiore [5].

$\mathbf{Clone}(S)$ is a cartesian category, permitting us to combine clones pointwise. The terminal object $\mathbf{1}$ is the unique clone in which every set of terms is a singleton. The binary product $\mathbf{X}_1 \times \mathbf{X}_2$ has sets of terms given by products of sets $(X_1 \times X_2)(\Gamma; A) = X_1(\Gamma; A) \times X_2(\Gamma; A)$, variables $\mathsf{var}_i^{(\Gamma)} = (\mathsf{var}_i^{(\Gamma)}, \mathsf{var}_i^{(\Gamma)})$, and substitution $(t_1, t_2)[(\sigma_{11}, \sigma_{21}), \ldots, (\sigma_{1n}, \sigma_{2n})] = (t_1[\boldsymbol{\sigma}_1], t_2[\boldsymbol{\sigma}_2])$.

▶ Remark 7. S-sorted abstract clones form a *variety* in the sense of universal algebra; this means that $\mathbf{Clone}(S)$ is the category of models for a (multisorted) algebraic theory. Such categories are well-behaved, and several of the properties we mention throughout the paper (such as being cartesian) follow abstractly from this observation. We often choose to be more explicit for ease of comprehension, but make note where this abstract perspective is helpful.

2.1 Substitution and context extension

We briefly consider the structure of substitutions $\boldsymbol{\sigma}$ in S-sorted clones $\mathbf{X}$, in particular to define various substitutions that we use below, and to characterize context extension in clones. If $\boldsymbol{\sigma} \in X(\Gamma; \Delta)$ and $\boldsymbol{\sigma}' \in X(\Delta; \Xi)$ are substitutions, then their *composition* $(\boldsymbol{\sigma}' \circ \boldsymbol{\sigma}) \in X(\Gamma; \Xi)$ is the substitution $(\sigma'_1[\boldsymbol{\sigma}], \ldots, \sigma'_m[\boldsymbol{\sigma}])$, where m is the length of Ξ. The three equations in the definition of a clone (Definition 2) equivalently state (1 & 2) that var is the (left- and right-) unit for composition $(\mathsf{var}^{(\Delta)} \circ \boldsymbol{\sigma} \;=\; \boldsymbol{\sigma} \;=\; \boldsymbol{\sigma} \circ \mathsf{var}^{(\Gamma)})$; and (3) that composition is associative $(\boldsymbol{\sigma}'' \circ (\boldsymbol{\sigma}' \circ \boldsymbol{\sigma}) \;=\; (\boldsymbol{\sigma}'' \circ \boldsymbol{\sigma}') \circ \boldsymbol{\sigma})$. In fact, this perspective underlies the connection between abstract clones and cartesian multicategories (which may be considered categories whose morphisms have multiple inputs, corresponding to each of the variables in a context): we elaborate on this connection in Section 7.

We call the substitutions $\boldsymbol{\rho} \in \mathsf{Var}_S(\Gamma; \Delta)$ *variable renamings.* This is justified by observing that $\boldsymbol{\rho}$ selects a variable in the context Δ for each variable in Γ. If $t \in X(\Delta; A)$ is a term in some clone $\mathbf{X}$, then $t[\triangleright\boldsymbol{\rho}] \in X(\Gamma; A)$ corresponds to the term in which the variables in t have been renamed according to $\boldsymbol{\rho}$. A special case of renaming is *weakening* $\mathsf{wk}_\Xi^{(\Gamma)} = (1, \ldots, n) \in \mathsf{Var}_S(\Gamma, \Xi; \Gamma)$. Using weakening and composition, we may define the *lifting* of a substitution $\boldsymbol{\sigma} \in X(\Gamma; \Delta)$ to a larger context:

$$\mathsf{lift}_\Xi(\boldsymbol{\sigma}) \;=\; (\boldsymbol{\sigma} \circ (\triangleright\mathsf{wk}_\Xi^{(\Gamma)}), \triangleright(n+1, \ldots, n+m)) \;\in\; X(\Gamma, \Xi; \Delta, \Xi)$$

where n is the length of Γ and m is the length of Ξ.

Context extension induces the following operation on clones. Given an S-sorted clone $\mathbf{X}$ and context $\Xi \in S^*$, we let $\Uparrow^\Xi\mathbf{X}$ be the S-sorted clone with terms $(\Uparrow^\Xi X)(\Gamma; A) = X(\Gamma, \Xi; A)$, variables $(\mathsf{var}_i^{(\Gamma,\Xi)})_{i \leq n} \in X(\Gamma, \Xi; \Gamma)$, and substitution $t[\boldsymbol{\sigma}, \triangleright(n+1, \ldots, n+m)] \in X(\Gamma, \Xi; A)$ for $t \in X(\Delta, \Xi; A)$ and $\boldsymbol{\sigma} \in X(\Gamma, \Xi; \Delta)$, where n is the length of Γ and m is the length of Ξ. This satisfies a universal property as follows. Weakening forms a homomorphism $\mathsf{weaken}_\mathbf{X}^{(\Xi)} : \mathbf{X} \to \Uparrow^\Xi\mathbf{X}$ that sends $t \in X(\Gamma; A)$ to $t[\triangleright\mathsf{wk}_\Xi^{(\Gamma)}] \in X(\Gamma, \Xi; A)$. Then, for every homomorphism $g : \Uparrow^\Xi\mathbf{X} \to \mathbf{Y}$, we obtain a homomorphism $g \circ \mathsf{weaken}_\mathbf{X}^{(\Xi)} : \mathbf{X} \to \mathbf{Y}$ and a substitution $g_{\diamond;\Xi}(\mathsf{var}^{(\Xi)}) \in Y(\diamond; \Xi)$. Together, these uniquely determine g: to give a homomorphism g is just to give a homomorphism $\mathbf{X} \to \mathbf{Y}$ and a closed term σ_i for each extra variable from Ξ. (From the perspective of algebraic theories, context extension $\Uparrow^\Xi\mathbf{X}$ corresponds to the construction of the *polynomial* [28] or *simple slice category* [26] over Ξ.)

▶ **Lemma 8.** *For each clone homomorphism* $f : \mathbf{X} \to \mathbf{Y}$ *and substitution* $\boldsymbol{\sigma} \in Y(\diamond; \Xi)$*, there is a unique homomorphism* $g : \Uparrow^{\Xi}\mathbf{X} \to \mathbf{Y}$ *such that* $g \circ \mathsf{weaken}^{(\Xi)}_{\mathbf{X}} = f$ *and* $g_{\diamond;\Xi}(\mathsf{var}^{(\Xi)}) = \boldsymbol{\sigma}$*.*

Proof. Suppose g is such a homomorphism. Then, for each $t \in X(\Gamma, \Xi; A)$, we have $g_{\Gamma;A}(t) = (g_{\Gamma,\Xi;A}(\mathsf{weaken}^{(\Xi)}_{\mathbf{X}}(t)))[\mathsf{var}^{(\Gamma)}, (g_{\diamond;\Xi}(\mathsf{var}^{(\Xi)})) \circ \triangleright \mathsf{wk}^{(\diamond)}_{\Gamma}] = (f_{\Gamma,\Xi;A}(t))[\mathsf{var}^{(\Gamma)}, \boldsymbol{\sigma} \circ \triangleright \mathsf{wk}^{(\diamond)}_{\Gamma}]$, where the first equality uses preservation of variables and substitution by g, and the second uses the assumptions on g. Hence, g is unique when it exists. For existence, define $g_{\Gamma;A}(t) = (f_{\Gamma,\Xi;A}(t))[\mathsf{var}^{(\Gamma)}, \boldsymbol{\sigma} \circ \triangleright \mathsf{wk}^{(\diamond)}_{\Gamma}]$. ◀

Substitutions $\boldsymbol{\sigma} \in Y(\diamond; \Xi)$ are in natural bijection with homomorphisms $\Uparrow^{\Xi}\mathsf{Var}_S \to \mathbf{Y}$, and so Lemma 8 equivalently states that $\Uparrow^{\Xi}\mathbf{X}$ is the coproduct of $\mathbf{X}$ and $\Uparrow^{\Xi}\mathsf{Var}_S$. (This contrasts with presheaf-based frameworks [16, 22], in which context extension is exponentiation.)

2.2 First-order presentations

Clones describe collections of terms closed under variable projection and substitution. We will frequently be interested in clones equipped with extra structure, so as, for example, to interpret the operations of a given type theory. *Presentations* permit the axiomatization of clones that interpret various operations, subject to sets of axioms; while the *algebras* for a given presentation are exactly those clones that satisfy the axiomatization. Later, we will see how clones may be freely generated from presentations, allowing one to define a clone simply by specifying its generating operators and axioms.

Our treatment of first-order presentations is the classical notion of presentation for multisorted universal algebra [9, 17].

▶ **Definition 9.** *An S-sorted* first-order signature Σ *consists of a set* $\Sigma(\Gamma; B)$ *for each* $(\Gamma; B) \in S^* \times S$*. We call the elements* $\mathsf{o} \in \Sigma(\Gamma; B)$ *the* $(\Gamma; B)$-ary operators*.* Terms over Σ *are generated by the following rules:*

$$\frac{}{\Gamma, x : A, \Delta \vdash x : A} \qquad \frac{\Gamma \vdash t_1 : A_1 \quad \cdots \quad \Gamma \vdash t_n : A_n}{\Gamma \vdash \mathsf{o}(t_1, \ldots, t_n) : B}\ (\mathsf{o} \in \Sigma(A_1, \ldots, A_n; B))$$

An $(A_1, \ldots, A_n; B)$-ary term t *over* Σ *is a term* $x_1 : A_1, \ldots, x_n : A_n \vdash t : B$*, and an* $(\Gamma; B)$-ary equation *over* Σ *is a pair* (t, u) *of* $(\Gamma; B)$*-ary terms. An S-sorted* first-order presentation $\boldsymbol{\Sigma} = (\Sigma, E)$ *consists of an S-sorted first-order signature* Σ *and, for each* $(\Gamma; B) \in S^* \times S$*, a set* $E(\Gamma; B)$ *of* $(\Gamma; B)$*-ary equations.*

▶ Remark 10. Observe that the operators of a signature correspond to terms in the logic specified below (namely, first-order equational logic). In particular, a $(\Gamma; B)$-ary operator o, where $\Gamma = [A_1, \ldots, A_n] \in S^*$, may be thought of either as a function $\mathsf{o} : A_1, \ldots, A_n \to B$, or as a term $x_1 : A_1, \ldots, x_n : A_n \vdash \mathsf{o} : B$. These perspectives are complementary, and mirror the practice in categorical logic of representing terms by morphisms.

▶ **Definition 11.** *If* $\Gamma \vdash u_i : A_i$ *for* $i \le n$ *and* $x_1 : A_1, \ldots x_n : A_n \vdash t : B$ *are terms over an S-sorted first-order signature* Σ*, their substitution* $\Gamma \vdash t\{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\} : B$ *is defined by recursion on t in the usual way. The* equational logic *over an S-sorted first-order presentation* $\boldsymbol{\Sigma} = (\Sigma, E)$ *consists of the following rules for the congruence of* $\approx$ *under operations and substitution, together with reflexivity, symmetry and transitivity of* $\approx$*:*

$$\frac{\Gamma \vdash t_1 \approx u_1 : A_1 \quad \cdots \quad \Gamma \vdash t_n \approx u_n : A_n}{\Gamma \vdash \mathsf{o}(t_1, \ldots, t_n) \approx \mathsf{o}(u_1, \ldots, u_n) : B}\ (\mathsf{o} \in \Sigma(A_1, \ldots, A_n; B))$$

$$\frac{\Gamma \vdash t'_1 \approx u'_1 : A_1 \quad \cdots \quad \Gamma \vdash t'_n \approx u'_n : A_n}{\Gamma \vdash t\{x_i \mapsto t'_i\}_i \approx u\{x_i \mapsto u'_i\}_i : B}\ ((t, u) \in E(A_1, \ldots, A_n; B))$$

The terms over Σ *form a clone* $\mathbf{Term}^{\mathbf{\Sigma}} = (\mathsf{Term}^{\mathbf{\Sigma}}, \mathsf{var}, \mathsf{subst})$*, where* $\mathsf{Term}^{\mathbf{\Sigma}}(\Gamma; A)$ *is the set of* $\approx$*-equivalence classes of* $(\Gamma; A)$*-ary terms; the variables are* $\mathsf{var}_i^{(\Gamma)} = x_i$*; and substitution is* $t[\boldsymbol{\sigma}] = t\{x_i \mapsto \sigma_i\}_i$*. A clone* $\mathbf{X}$ *is* presented by $\mathbf{\Sigma}$ *when* $\mathbf{Term}^{\mathbf{\Sigma}}$ *is isomorphic to* $\mathbf{X}$ *in* $\mathbf{Clone}(S)$ *(that is, when there are homomorphisms* $\mathbf{Term}^{\mathbf{\Sigma}} \rightleftarrows \mathbf{X}$ *that are mutually inverse).*

▶ Remark 12. A clone may have many different presentations: for instance, the clone $\mathbf{Mon}$ of monoids (Example 4) may be presented by a unit and a binary multiplication operation, or by an n-ary multiplication operation for each $n \in \mathbb{N}$ (subject to suitable axioms).

▶ **Example 13.** Fix a finite set $V = \{v_1, \ldots, v_k\}$ of *values*. The Ty-sorted presentation $\mathbf{\Sigma}_V^{\mathbf{GS}}$ of *global V-valued state* has a $(\underbrace{\mathsf{b}, \ldots, \mathsf{b}}_{k}; \mathsf{b})$-ary operator get, a $(\mathsf{b}; \mathsf{b})$-ary operator put_{v_i} for each $i \leq k$, and equations

$$
\begin{aligned}
x : \mathsf{b} &\vdash \mathsf{get}(\mathsf{put}_{v_1}(x), \ldots, \mathsf{put}_{v_k}(x)) \approx x : \mathsf{b} & \\
x_1 : \mathsf{b}, \ldots, x_k : \mathsf{b} &\vdash \mathsf{put}_{v_i}(\mathsf{get}(x_1, \ldots, x_k)) \approx \mathsf{put}_{v_i}(x_i) : \mathsf{b} & \text{for each } i \leq k \\
x : \mathsf{b} &\vdash \mathsf{put}_{v_i}(\mathsf{put}_{v_j}(x)) \approx \mathsf{put}_{v_j}(x) : \mathsf{b} & \text{for each } i, j \leq k
\end{aligned}
$$

Informally, the term $\mathsf{get}(t_1, \ldots, t_n)$ gets the current value v_i of the state and then continues as t_i, while the term $\mathsf{put}_{v_i}(t)$ sets the state to v_i and then continues as t. (In Example 23 below, we combine this presentation with the STLC to obtain a call-by-name calculus with global state. In call-by-name calculi, effects occur at base types, so it is only necessary to axiomatize get and put_{v_i} operators for $\mathsf{b} \in \mathsf{Ty}$, rather than for all types.) We denote by $\mathbf{GS}_V$ the clone $\mathbf{Term}^{\mathbf{\Sigma}_V^{\mathbf{GS}}}$ arising from the presentation $\mathbf{\Sigma}_V^{\mathbf{GS}}$.

3 Second-order presentations

Just as first-order presentations describe algebraic structure, second-order presentations describe binding algebraic structure [16]. Variable-binding operators are prevalent in type theory: for instance, the λ-abstraction operator of the STLC, let-in expressions in functional programming languages, and case-splitting in calculi with sum types. Second-order presentations are similar to first-order presentations, except that each operator must describe its binding structure, i.e. how many variables (and of what types) it binds in each operand. Hence, while first-order arities have the form $(A_1, \ldots, A_n; B) \in S^* \times S$, second-order arities have the form $((\Delta_1; A_1), \ldots, (\Delta_n; A_n); B) \in (S^* \times S)^* \times S$. Operators of such an arity take n arguments of types $A_1, \ldots A_n$ and produce terms of type B: the length of the context $\Delta_i \in S^*$ is the number of variables bound by the i^{th} argument; and the argument types are given by the list Δ_i. First-order operators may be expressed as second-order operators that bind no variables.

▶ **Definition 14.** *An S-sorted* second-order signature *[16, 13] consists of a set* $\Sigma(\Psi; B)$ *for each* $(\Psi; B) \in (S^* \times S)^* \times S$*. We call the elements* $\mathsf{o} \in \Sigma(\Psi; B)$ *the* $(\Psi; B)$-ary operators.

▶ **Example 15.** The Ty-sorted second-order signature Σ^Λ of the STLC consists of an $((\diamond; A{\Rightarrow}B), (\diamond; A); B)$-ary operator app and an $((A; B); (A{\Rightarrow}B))$-ary operator abs for each $A, B \in \mathsf{Ty}$. Thus each application operator app has two arguments, neither of which bind variables; and each λ-abstraction operator abs has one argument, which binds one variable.

Just as the axioms of first-order presentations are expressed in first-order equational logic, the axioms of second-order presentations are expressed in the second-order equational logic of Fiore and Hur [13]. Second-order equational logic extends the first-order setting with *metavariables* [1, 18, 11], which conceptually stand for parameterized placeholders for terms.

Each variable $x : A$ in first-order logic has an associated type $A \in S$; correspondingly, each metavariable $\textsc{m} : (A_1, \ldots, A_n; A)$ has an associated context and type (called *second-order arities* in [5]). $\textsc{m}$ may be thought of as a variable parameterized by n terms of types A_1 through A_n; a nullary ($n = 0$) metavariable behaves like an ordinary variable. There are several alternative ways to describe second-order equational logic [6], but we follow Fiore and Hur [13] in associating to each term both a variable context and a metavariable context: a metavariable context Ψ is a list of context–sort pairs $(\Delta; A) \in S^* \times S$. The judgment $\Psi \mid \Delta \vdash t : A$ expresses that the term t has sort A in variable context Δ and metavariable context Ψ. Below, we write $\vec{x}$ for a list $x_1, \ldots, x_n$ of variables, $\vec{x}.\, t$ to indicate binding of the variables $\vec{x}$ in t, and write $\vec{x} : \Delta$ as an abbreviation of $x_1 : A_1, \ldots, x_n : A_n$ for $\Delta = [A_1, \ldots, A_n]$.

▶ **Definition 16.** *Suppose S is a set and Σ is an S-sorted second-order signature.* Terms over Σ *are generated by the following rules for variables, metavariables, and operators:*

$$\frac{}{\Psi \mid \Gamma, x : A, \Delta \vdash x : A}$$

$$\frac{\Psi, \textsc{m} : (A_1, \ldots, A_n; B), \Phi \mid \Gamma \vdash t_1 : A_1 \quad \cdots \quad \Psi, \textsc{m} : (A_1, \ldots, A_n; B), \Phi \mid \Gamma \vdash t_n : A_n}{\Psi, \textsc{m} : (A_1, \ldots, A_n; B), \Phi \mid \Gamma \vdash \textsc{m}(t_1, \ldots, t_n) : B}$$

$$\frac{\Psi \mid \Gamma, \vec{x}_1 : \Delta_1 \vdash t_1 : A_1 \quad \cdots \quad \Psi \mid \Gamma, \vec{x}_n : \Delta_n \vdash t_n : A_n}{\Psi \mid \Gamma \vdash \mathsf{o}((\vec{x}_1.\, t_1), \ldots, (\vec{x}_n.\, t_n)) : B} \; (\mathsf{o} \in \Sigma((\Delta_1; A_1), \ldots, (\Delta_n; A_n); B))$$

A $((\Delta_1; A_1), \ldots, (\Delta_n; A_n); B)$-ary term *over Σ is a term $\textsc{m}_1 : (\Delta_1, A_1), \ldots, \textsc{m}_n : (\Delta_n; A_n) \mid \diamond \vdash t : B$, and a* $(\Psi; B)$-ary equation *is a pair (t, u) of $(\Psi; B)$-ary terms. An* S-sorted second-order presentation $\mathbf{\Sigma} = (\Sigma, E)$ *consists of an S-sorted second-order signature Σ and, for each $(\Psi; B) \in (S^* \times S)^* \times S$, a set $E(\Psi; B)$ of $(\Psi; B)$-ary equations over Σ.*

Multisorted second-order presentations may essentially be taken as a definition of *simple type theory* (modulo the subtlety regarding type operators described in Remark 6): just as the informal notion of *algebra* was formalized through the framework of universal algebra [8], so second-order presentations facilitate a precise, formal definition of simple type theory [5].

▶ **Example 17.** The operators of the signature Σ^Λ of the STLC present the following rules:

$$\frac{\Psi \mid \Gamma \vdash f : A \Rightarrow B \qquad \Psi \mid \Gamma \vdash a : A}{\Psi \mid \Gamma \vdash \mathsf{app}(f, a) : B} \qquad \frac{\Psi \mid \Gamma, x : A \vdash t : B}{\Psi \mid \Gamma \vdash \mathsf{abs}(x.\, t) : A \Rightarrow B}$$

We can then give, for each $A, B \in \mathsf{Ty}$, an $((A; B), (\diamond; A); B)$-ary equation for β-equality, and an $((\diamond; A{\Rightarrow}B); (A{\Rightarrow}B))$-ary equation for η-equality:

$$\textsc{m}_1 : (A; B), \textsc{m}_2 : (\diamond; A) \mid \diamond \vdash \quad \mathsf{app}(\mathsf{abs}(x.\, \textsc{m}_1(x)), \textsc{m}_2()) \approx \textsc{m}_1(\textsc{m}_2()) \quad : B \qquad (\beta)$$

$$\textsc{m} : (\diamond; A{\Rightarrow}B) \mid \diamond \vdash \quad \mathsf{abs}(x.\, \mathsf{app}(\textsc{m}(), x)) \approx \textsc{m}() \quad : A \Rightarrow B \qquad (\eta)$$

The signature Σ^Λ together with these equations forms the Ty-sorted second-order presentation $\mathbf{\Sigma}^{\Lambda_{\beta\eta}}$ of the STLC with $\beta\eta$-equality. Note that second-order equations permit the expression of *axiom schemata*, as axioms containing metavariables (in both the traditional and precise sense of the term "metavariable") [12, 5]. Without second-order equations, one would have to add β and η equations for each instantiation of the metavariables in the rules above.

▶ **Definition 18.** *If $(\Psi \mid \Gamma \vdash u_i : A_i)_i$ and $\Psi \mid x_1 : A_1, \ldots, x_n : A_n \vdash t : B$ are terms over an S-sorted second-order signature Σ, then their* substitution $\Psi \mid \Gamma \vdash t\{x_i \mapsto u_i\}_i : B$ *is defined by recursion on t:*

$$x_j\{x_i \mapsto u_i\}_i = u_j \qquad \textsc{m}(t_1, \ldots t_m)\{x_i \mapsto u_i\}_i = \textsc{m}(t_1\{x_i \mapsto u_i\}_i, \ldots, t_m\{x_i \mapsto u_i\}_i)$$

$$\mathsf{o}((\vec{y_1}.\, t_1), \ldots, \mathsf{o}(\vec{y_k}.\, t_k))\{x_i \mapsto u_i\}_i = \mathsf{o}((\vec{y_1}.\, t_1\{x_i \mapsto u_i\}_i), \ldots, (\vec{y_k}.\, t_k\{x_i \mapsto u_i\}_i))$$

(On the right-hand side of the definition on operators, the terms t_i are weakened, and we omit from the substitution variables that are mapped to themselves.) If instead we have terms $(\Psi \mid \Gamma, \vec{x_i} : \Delta_i \vdash u_i : A_i)_i$ *and* $\mathrm{M}_1 : (\Delta_1; A_1), \ldots, \mathrm{M}_n : (\Delta_n; A_n) \mid \Gamma' \vdash t : B$ *then their* metasubstitution $\Psi \mid \Gamma, \Gamma' \vdash t\{\mathrm{M}_i \mapsto (\vec{x_i}.\, u_i)\}_i : B$ *is defined using ordinary substitution by recursion on* t*:*

$$x\{\mathrm{M}_i \mapsto (\vec{x_i}.\, u_i)\}_i = x \qquad \mathrm{M}_j(t_1, \ldots, t_m)\{\mathrm{M}_i \mapsto (\vec{x_i}.\, u_i)\}_i = u_j\{x_{jk} \mapsto t_k\{\mathrm{M}_i \mapsto (\vec{x_i}.\, u_i)\}_i\}_k$$

$$\begin{aligned} &\mathsf{o}((\vec{y_1}.\, t_1), \ldots, (\vec{y_k}.\, t_k))\{\mathrm{M}_i \mapsto (\vec{x_i}.\, u_i)\}_i \\ &\quad = \mathsf{o}((\vec{y_1}.\, t_1\{\mathrm{M}_i \mapsto (\vec{x_i}.\, u_i)\}_i), \ldots, (\vec{y_k}.\, t_k\{\mathrm{M}_i \mapsto (\vec{x_i}.\, u_i)\}_i)) \end{aligned}$$

3.1 Algebras

The algebras for a presentation are the abstract clones interpreting each of the operations of the signature, subject to the axioms of the presentation. In other words, a presentation is a specification of structure, while the algebras are the realizations, or models, of that structure. For instance, in the first-order setting, the algebras for the presentation of monoids form (set-theoretic) monoids.

► **Definition 19.** *An* algebra $(\mathbf{X}, [\![-]\!])$ *for an S-sorted second-order signature Σ (called "presentation clones" in [32]) consists of an S-sorted clone $\mathbf{X}$ and, for each context Γ and $((\Delta_1; A_1), \ldots, (\Delta_n; A_n); B)$-ary operator o, a function $[\![\mathsf{o}]\!]_\Gamma : \prod_i X(\Gamma, \Delta_i; A_i) \to X(\Gamma; B)$ such that, for all substitutions $\boldsymbol{\sigma} \in X(\Xi; \Gamma)$ and tuples of terms $(t_i \in X(\Gamma, \Delta_i; A_i))_i$,*

$$([\![\mathsf{o}]\!]_\Gamma(t_1, \ldots, t_n))[\boldsymbol{\sigma}] = [\![\mathsf{o}]\!]_\Xi(t_1[\mathsf{lift}_{\Delta_1}\boldsymbol{\sigma}], \ldots, t_n[\mathsf{lift}_{\Delta_n}\boldsymbol{\sigma}])$$

A homomorphism $f : (\mathbf{X}, [\![-]\!]) \to (\mathbf{X}', [\![-]\!]')$ *of Σ-algebras is a homomorphism $f : \mathbf{X} \to \mathbf{X}'$ of clones such that, for all $\mathsf{o} \in \Sigma((\Delta_1; A_1), \ldots, (\Delta_n; A_n); B)$ and $(t_i \in X(\Gamma, \Delta_i; A_i))_i$,*

$$f_{\Gamma;B}([\![\mathsf{o}]\!]_\Gamma(t_1, \ldots, t_n)) \;=\; [\![\mathsf{o}]\!]'_\Gamma(f_{\Gamma,\Delta_1;A_1}t_1, \ldots, f_{\Gamma,\Delta_n;A_n}t_n)$$

The interpretation of operators in a Σ-algebra $(\mathbf{X}, [\![-]\!])$ extends to an interpretation $[\![t]\!]_\Gamma : \prod_i X(\Gamma, \Delta_i; A_i) \to X(\Gamma, \Xi; B)$ of each term $\mathrm{M}_1 : (\Delta_1; A_1), \ldots, \mathrm{M}_n : (\Delta_n; A_n) \mid \vec{x} : \Xi \vdash t : B$ as follows (where n is the length of Γ):

$$\begin{aligned} [\![x_i]\!]_\Gamma(\boldsymbol{\sigma}) &= \mathsf{var}^{(\Gamma,\Xi)}_{n+i} \\ [\![\mathrm{M}_i(t_1, \ldots, t_m)]\!]_\Gamma(\boldsymbol{\sigma}) &= \sigma_i[\mathsf{var}^{(\Gamma,\Xi)}_1, \ldots, \mathsf{var}^{(\Gamma,\Xi)}_n, [\![t_1]\!]_\Gamma(\boldsymbol{\sigma}), \ldots, [\![t_m]\!]_\Gamma(\boldsymbol{\sigma})] \\ [\![\mathsf{o}((\vec{x}_1.\, t_1), \ldots, (\vec{x}_m.\, t_m))]\!]_\Gamma(\boldsymbol{\sigma}) &= [\![\mathsf{o}]\!]_{\Gamma,\Xi}([\![t_1]\!]_\Gamma(\boldsymbol{\sigma}), \ldots, [\![t_m]\!]_\Gamma(\boldsymbol{\sigma})) \end{aligned}$$

► **Definition 20.** *An algebra $(\mathbf{X}, [\![-]\!])$ for a second-order presentation $\mathbf{\Sigma} = (\Sigma, E)$ is a Σ-algebra such that, for all equations $(t, u) \in E(\Psi; A)$ and contexts Γ, we have $[\![t]\!]_\Gamma = [\![u]\!]_\Gamma$. We let $\mathbf{\Sigma}$-$\mathbf{Alg}$ be the category of $\mathbf{\Sigma}$-algebras and all Σ-algebra homomorphisms between them.*

► **Example 21.** An algebra for the presentation $\mathbf{\Sigma}^{\Lambda_{\beta\eta}}$ of the STLC with $\beta\eta$-equality consists of a Ty-sorted clone $\mathbf{X}$ and functions

$$[\![\mathsf{app}]\!]_\Gamma : X(\Gamma; A \Rightarrow B) \times X(\Gamma; A) \to X(\Gamma; B) \qquad [\![\mathsf{abs}]\!]_\Gamma : X(\Gamma, A; B) \to X(\Gamma; A \Rightarrow B)$$

that commute with substitution and satisfy

$$[\![\mathsf{app}]\!]_\Gamma([\![\mathsf{abs}]\!]_\Gamma(t),\, t') = t[\mathsf{var}^{(\Gamma)}, t'] \quad \text{for } t \in X(\Gamma, A; B), t' \in X(\Gamma; A) \qquad (\beta)$$

$$[\![\mathsf{abs}]\!]_\Gamma([\![\mathsf{app}]\!]_{\Gamma,A}(t[\triangleright\mathsf{wk}^{(\Gamma)}_A], \mathsf{var}^{(\Gamma,A)}_{n+1})) = t \quad \text{for } t \in X(\Gamma; A{\Rightarrow}B) \qquad (\eta)$$

For each set Z we have a set-theoretic interpretation of the STLC, which forms a $\mathbf{\Sigma}^{\Lambda_{\beta\eta}}$-algebra $(\mathcal{M}_Z, \mathcal{M}_Z[\![-]\!])$ as follows. Define interpretations $\mathcal{M}_Z[\![A]\!] \in \mathbf{Set}$ of each sort $A \in \mathsf{Ty}$ recursively by setting $\mathcal{M}_Z[\![\mathsf{b}]\!] = Z$ and $\mathcal{M}_Z[\![A \Rightarrow B]\!] = \mathbf{Set}(\mathcal{M}_Z[\![A]\!], \mathcal{M}_Z[\![B]\!])$ (where $\mathbf{Set}(Y, Y')$ is the set of functions $Y \to Y'$). We then have a Ty-sorted clone $\mathcal{M}_Z$, where the sets of terms are given by $\mathcal{M}_Z(A_1, \ldots, A_n; B) = \mathbf{Set}(\prod_i \mathcal{M}_Z[\![A_i]\!], \mathcal{M}_Z[\![B]\!])$, the variables by projections $\mathsf{var}_i^{(\Gamma)} = \pi_i$, and substitution by $f[\boldsymbol{\sigma}] = (\xi \mapsto f(\sigma_1(\xi), \ldots, \sigma_n(\xi)))$. This forms a $\mathbf{\Sigma}^{\Lambda_{\beta\eta}}$-algebra, with interpretations of the operators given by function application and currying. More generally, the interpretation of the STLC in any cartesian-closed category $\mathbf{C}$ with a specified object $Z \in \mathbf{C}$ forms a $\mathbf{\Sigma}^{\Lambda_{\beta\eta}}$-algebra taking $\mathcal{M}_Z(A_1, \ldots, A_n; B) = \mathbf{C}(\prod_i \mathcal{M}_Z[\![A_i]\!], \mathcal{M}_Z[\![B]\!])$ to be the sets of terms, where $\mathcal{M}_Z[\![\mathsf{b}]\!] = Z$ and $\mathcal{M}_Z[\![A \Rightarrow B]\!] = \mathcal{M}_Z[\![B]\!]^{\mathcal{M}_Z[\![A]\!]}$.

The cartesian structure of $\mathbf{Clone}(S)$ lifts to $\mathbf{\Sigma}\text{-}\mathbf{Alg}$ for every presentation $\mathbf{\Sigma}$: the clone $\mathbf{1}$ uniquely forms a $\mathbf{\Sigma}$-algebra, and the product $(\mathbf{X}_1, [\![-]\!]_1) \times (\mathbf{X}_2, [\![-]\!]_2)$ is the clone $\mathbf{X}_1 \times \mathbf{X}_2$ equipped with interpretations $[\![\mathsf{o}]\!]_\Gamma((\sigma_{11}, \sigma_{21}), \ldots, (\sigma_{1n}, \sigma_{2n})) = ([\![\mathsf{o}]\!]_{1,\Gamma}(\boldsymbol{\sigma}_1), [\![\mathsf{o}]\!]_{2,\Gamma}(\boldsymbol{\sigma}_2))$.

4 Free algebras

Second-order S-sorted presentations $\mathbf{\Sigma}$ can be viewed as descriptions of simple type theories for which S is the set of types. In particular, the operators specify the term formers of the type theory (such as λ-abstraction, or application). From this perspective, the syntax of the type theory described by $\mathbf{\Sigma}$ is the *initial* $\mathbf{\Sigma}$-algebra: there is a unique Σ-algebra homomorphism from the algebra formed by the syntax to any other algebra, given by induction on terms. More generally, given the syntax of an existing theory in the form of a clone $\mathbf{X}$, the *free* $\mathbf{\Sigma}$-algebra on $\mathbf{X}$ is given by augmenting $\mathbf{X}$ by the operators and equations of $\mathbf{\Sigma}$; or, from another perspective, augmenting the type theory described by $\mathbf{\Sigma}$ with the operations specified by $\mathbf{X}$. For example, the free $\mathbf{\Sigma}^{\Lambda_{\beta\eta}}$-algebra on $\mathbf{GS}_V$ (Example 13) may be seen as the STLC extended by additional term formers (get and $\mathsf{put}_{v_1}, \ldots, \mathsf{put}_{v_k}$) representing the side-effects of global state.

▶ **Definition 22.** *Suppose $\mathbf{\Sigma} = (\Sigma, E)$ is an S-sorted second-order presentation and $\mathbf{X}$ is an S-sorted clone. A $\mathbf{\Sigma}$-algebra $F_{\mathbf{\Sigma}}\mathbf{X}$ equipped with a clone homomorphism $\eta_{\mathbf{X}} : \mathbf{X} \to F_{\mathbf{\Sigma}}\mathbf{X}$ is the* free $\mathbf{\Sigma}$-algebra *on $\mathbf{X}$ if, for any other $\mathbf{\Sigma}$-algebra $(\mathbf{Y}, [\![-]\!])$ and clone homomorphism $f : \mathbf{X} \to \mathbf{Y}$, there is a unique Σ-algebra homomorphism $f^\dagger : F_{\mathbf{\Sigma}}\mathbf{X} \to (\mathbf{Y}, [\![-]\!])$ such that $f^\dagger \circ \eta_{\mathbf{X}} = f$. The* initial $\mathbf{\Sigma}$-algebra *is the free $\mathbf{\Sigma}$-algebra on Var_S.*

▶ **Example 23.** Recall the presentation $\mathbf{\Sigma}^{\Lambda_{\beta\eta}}$ of the STLC with $\beta\eta$-equality from Example 17. The initial $\mathbf{\Sigma}^{\Lambda_{\beta\eta}}$-algebra is the clone $\mathbf{\Lambda}_{\beta\eta}$ of STLC terms up to $\approx_{\beta\eta}$ (Example 5), with the operators app and abs interpreted as

$$((f, a) \mapsto \mathsf{app}\ f\ a) : \Lambda_{\beta\eta}(\Gamma; A \Rightarrow B) \times \Lambda_{\beta\eta}(\Gamma; A) \to \Lambda_{\beta\eta}(\Gamma; B)$$
$$(t \mapsto \lambda x : A.\, t) : \Lambda_{\beta\eta}(\Gamma, A; B) \to \Lambda_{\beta\eta}(\Gamma; A \Rightarrow B)$$

The free $\mathbf{\Sigma}^{\Lambda_{\beta\eta}}$-algebra on the clone $\mathbf{GS}_V$ of global V-valued state (Example 13) can be described as follows for $V = \{v_1, \ldots, v_k\}$. The underlying Ty-sorted clone is defined in the same way as $\mathbf{\Lambda}_{\beta\eta}$, but with the following additional term formers and equations (omitting the typing constraints on equations).

$$\frac{\Gamma \vdash t_1 : \mathsf{b} \quad \cdots \quad \Gamma \vdash t_k : \mathsf{b}}{\Gamma \vdash \mathsf{get}(t_1, \ldots, t_k) : \mathsf{b}} \qquad \frac{\Gamma \vdash t : \mathsf{b}}{\Gamma \vdash \mathsf{put}_{v_i}(t) : \mathsf{b}}\ (i \leq k)$$

$$\begin{aligned} \mathsf{get}(\mathsf{put}_{v_1}(t), \ldots, \mathsf{put}_{v_k}(t)) &\approx_{\beta\eta} t \\ \mathsf{put}_{v_i}(\mathsf{get}(t_1, \ldots, t_k)) &\approx_{\beta\eta} \mathsf{put}_{v_i}(t_i) && (i \leq k) \\ \mathsf{put}_{v_i}(\mathsf{put}_{v_j}(t)) &\approx_{\beta\eta} \mathsf{put}_{v_j}(t) && (i, k \leq k) \end{aligned}$$

Terms

$$\frac{}{\Gamma, x : A, \Delta \vdash_{\mathbf{X}} x : A} \qquad \frac{\Gamma \vdash_{\mathbf{X}} t_1 : A_1 \quad \cdots \quad \Gamma \vdash_{\mathbf{X}} t_n : A_n}{\Gamma \vdash_{\mathbf{X}} f(t_1, \ldots, t_n) : B} \; (f \in X(A_1, \ldots, A_n; B))$$

$$\frac{\Gamma, \vec{x}_1 : \Delta_1 \vdash_{\mathbf{X}} t_1 : A_1 \quad \cdots \quad \Gamma, \vec{x}_n : \Delta_n \vdash_{\mathbf{X}} t_n : A_n}{\Gamma \vdash_{\mathbf{X}} \mathsf{o}((\vec{x}_1. t_1), \ldots, (\vec{x}_n. t_n)) : B} \; (\mathsf{o} \in \Sigma((\Delta_1; A_1), \ldots, (\Delta_n; A_n); B))$$

Equations (reflexivity, symmetry, transitivity omitted)

$$\frac{\Gamma \vdash_{\mathbf{X}} t_1 \approx u_1 : A_1 \quad \cdots \quad \Gamma \vdash_{\mathbf{X}} t_n \approx u_n : A_n}{\Gamma \vdash_{\mathbf{X}} f(t_1, \ldots, t_n) \approx f(u_1, \ldots, u_n) : B} \; (f \in X(A_1, \ldots, A_n; B))$$

$$\frac{\Gamma, \vec{x}_1 : \Delta_1 \vdash_{\mathbf{X}} t_1 \approx u_1 : A_1 \quad \cdots \quad \Gamma, \vec{x}_n : \Delta_n \vdash_{\mathbf{X}} t_n \approx u_n : A_n}{\Gamma \vdash_{\mathbf{X}} \mathsf{o}((\vec{x}_1. t_1), \ldots, (\vec{x}_n. t_n)) \approx \mathsf{o}((\vec{x}_1. u_1), \ldots, (\vec{x}_n. u_n)) : B} \; (\mathsf{o} \in \Sigma((\Delta_1; A_1), \ldots; B))$$

$$\frac{\Gamma, \vec{x}_1{:}\Delta_1 \vdash_{\mathbf{X}} t_1 \approx u_1 : A_1 \quad \cdots \quad \Gamma, \vec{x}_n{:}\Delta_n \vdash_{\mathbf{X}} t_n \approx u_n : A_n}{\Gamma \vdash_{\mathbf{X}} t'\{M_i \mapsto (\vec{x}_i. t_i)\}_i \approx u'\{M_i \mapsto (\vec{x}_i. u_i)\}_i : B} \; ((t', u') \in E((\Delta_1; A_1), \ldots; B))$$

$$\frac{\Gamma \vdash_{\mathbf{X}} t_1 : A_1 \quad \cdots \quad \Gamma \vdash_{\mathbf{X}} t_n : A_n}{\Gamma \vdash_{\mathbf{X}} t_i \approx \mathsf{var}_i^{(A_1, \ldots, A_n)}(t_1, \ldots, t_n) : B} \; (i \leq n)$$

$$\frac{\Gamma \vdash_{\mathbf{X}} t_1 : A_1 \quad \cdots \quad \Gamma \vdash_{\mathbf{X}} t_n : A_n}{\Gamma \vdash_{\mathbf{X}} f(\sigma_1(t_1, \ldots, t_n), \ldots, \sigma_k(t_1, \ldots, t_n)) \approx (f[\boldsymbol{\sigma}])(t_1, \ldots, t_n) : B}$$
$$(f \in X(A'_1, \ldots, A'_k; B), \boldsymbol{\sigma} \in X(A_1, \ldots, A_n; A'_1, \ldots, A'_k))$$

Figure 1 Construction of the free (Σ, E)-algebra on a clone $\mathbf{X} = (X, \mathsf{var}, \mathsf{subst})$.

This forms a $\mathbf{\Sigma}^{\Lambda_{\beta\eta}}$-algebra in the same way as $\mathbf{\Lambda}_{\beta\eta}$ above. The morphism $\eta_{\mathbf{GS}_V}$ is given by $\eta_{\mathbf{GS}_V}(\mathsf{get}(t_1, \ldots, t_k)) = \mathsf{get}(\eta_{\mathbf{GS}_V}(t_1), \ldots, \eta_{\mathbf{GS}_V}(t_k))$ and $\eta_{\mathbf{GS}_V}(\mathsf{put}_{v_i}(t)) = \mathsf{put}_{v_i}(\eta_{\mathbf{GS}_V}(t))$.

If $\mathbf{\Sigma}'$ is a first-order presentation, the free $\mathbf{\Sigma}$-algebra on $\mathbf{Term}^{\mathbf{\Sigma}'}$ is closed under the operators of $\mathbf{\Sigma}'$: each $\mathsf{o} \in \mathbf{\Sigma}'(A_1, \ldots, A_n; B)$ induces a term $\eta(\mathsf{o}(x_1, \ldots, x_n)) \in F_{\mathbf{\Sigma}}\mathbf{X}(A_1, \ldots, A_n; B)$ and hence functions $(\boldsymbol{\sigma} \mapsto \eta(\mathsf{o}(\vec{x}))[\boldsymbol{\sigma}]) : F_{\mathbf{\Sigma}}\mathbf{X}(\Gamma; A_1, \ldots, A_n) \to F_{\mathbf{\Sigma}}\mathbf{X}(\Gamma; B)$.

We show that free algebras for any signature, and on any clone, exist, by constructing them explicitly. Existence of these free algebras facilitates the developments in the next sections. However, note that we do not rely on the explicit description: after this section, we reason about free algebras solely using the universal property in Definition 22. This is important, as we wish to reason about type theories independently of their syntax, which leads to greatly simplified proofs. (It is also possible to prove the existence of free algebras entirely abstractly using a monadicity theorem and Remark 7, avoiding concrete syntax.)

In universal algebra, free algebras of first-order presentations are constructed in two steps: by first closing a sort-indexed set X of constants under the operators of the presentation; and then quotienting the terms by the equations of the presentation. Figure 1 gives the analogous construction in the second-order setting. First, we construct terms $\Gamma \vdash_{\mathbf{X}} t : B$ from variables, the terms of the clone $f \in X(A_1, \ldots, A_n; B)$ (viewed as function symbols), and the operators of the presentation $\mathbf{\Sigma}$. Second, we quotient by the equivalence relation $\approx$ generated by congruence, the equations of $\mathbf{\Sigma}$ (using metasubstitution), and rules imposing compatibility with the clone structure of $\mathbf{X}$. The clone $F_{\mathbf{\Sigma}}\mathbf{X}$ has terms $F_{\mathbf{\Sigma}}\mathbf{X}(\Gamma; B) = \{\Gamma \vdash_{\mathbf{X}} t : B\}/\approx$, with variables and substitution defined in the evident way; the homomorphism $\eta_{\mathbf{X}} : \mathbf{X} \to F_{\mathbf{\Sigma}}\mathbf{X}$ sends $t \in X(\Gamma; B)$ to $x_1 : A_1, \ldots, x_n : A_n \vdash_{\mathbf{X}} t(x_1, \ldots, x_n) : B$, where $\Gamma = [A_1, \ldots, A_n]$.

▶ **Proposition 24.** *For every S-sorted second-order presentation $\mathbf{\Sigma}$ and S-sorted clone $\mathbf{X}$, the free $\mathbf{\Sigma}$-algebra $F_{\mathbf{\Sigma}}\mathbf{X}$ exists.*

The forgetful functor $\mathbf{\Sigma}\text{-}\mathbf{Alg} \to \mathbf{Clone}(S)$ therefore has a left adjoint (in fact, it is monadic).

5 Induction over second-order syntax

We now describe how the formalism of abstract clones may be used to prove properties of simple type theories. To begin, we consider predicates over abstract clones, which are predicates over the terms of the type theory induced by the clone, closed under the structural operations of variable projection and substitution. Below, we extend each family of subsets $P(\Gamma; A) \subseteq Y(\Gamma; A)$ to contexts by defining $P(\Gamma; A_1, \ldots, A_n)$ to be the set of all substitutions $\boldsymbol{\sigma} \in Y(\Gamma; A_1, \ldots, A_n)$ such that $\sigma_i \in P(\Gamma; A_i)$ for all $i \leq n$.

▶ **Definition 25.** *A* predicate P *over an S-sorted clone $\mathbf{X}$ consists of a subset $P(\Gamma; A) \subseteq X(\Gamma; A)$ for each $(\Gamma; A) \in S^* \times S$ such that, for all contexts $\Gamma = [A_1, \ldots, A_n]$ and $i \leq n$, we have $\mathsf{var}_i^{(\Gamma)} \in P(\Gamma; A_i)$, and, for all $t \in P(\Delta; B)$ and $\boldsymbol{\sigma} \in P(\Gamma; \Delta)$, we have $t[\boldsymbol{\sigma}] \in P(\Gamma; B)$.*

Closure under variables and under substitution imply that P forms a clone $\mathbf{P}$ whose inclusion $\mathbf{P} \hookrightarrow \mathbf{X}$ into $\mathbf{X}$ is a clone homomorphism. Predicates over S-sorted clones are equivalently the subobjects in $\mathbf{Clone}(S)$, and are hence closed under arbitrary conjunction, existential quantification, and quotients of equivalence relations. (This follows from Remark 7, since varieties are exact categories [7, Theorem 5.11], and all exact categories enjoy these properties.) They are also closed under context extension: if P is a predicate over $\mathbf{X}$ and Ξ is a context, then $\Uparrow^{\Xi} P$ is a predicate over $\Uparrow^{\Xi} \mathbf{X}$.

We present a meta-theorem for establishing properties of simple type theories.

▶ **Theorem 26** (Induction principle for second-order syntax)**.** *Suppose that $(\mathbf{Y}, [\![-]\!])$ is an algebra for an S-sorted second-order presentation $\mathbf{\Sigma}$, that $f : \mathbf{X} \to \mathbf{Y}$ is a clone homomorphism from an S-sorted clone $\mathbf{X}$, and that P is a predicate over $\mathbf{Y}$. If*

- *for all operators $\mathsf{o} \in \Sigma((\Delta_1; A_1), \ldots, (\Delta_n; A_n); B)$, contexts $\Gamma \in S^*$, and tuples of terms $(t_i \in P(\Gamma, \Delta_i; A_i))_i$ we have $[\![\mathsf{o}]\!]_\Gamma(t_1, \ldots, t_n) \in P(\Gamma; B)$;*
- *for all terms $t \in X(\Gamma; A)$ we have $f_{\Gamma;A}(t) \in P(\Gamma; A)$,*

then, for all free terms $t \in (F_{\mathbf{\Sigma}}\mathbf{X})(\Gamma; A)$, we have $f^\dagger_{\Gamma;A}(t) \in P(\Gamma; A)$.

Proof. The predicate P is closed under operators, so the interpretations of operators in $\mathbf{Y}$ make $\mathbf{P}$ into a $\mathbf{\Sigma}$-algebra. The image of f is contained in P, so f forms a clone homomorphism $\mathbf{X} \to \mathbf{P}$. By the universal property of the free algebra $F_{\mathbf{\Sigma}}\mathbf{X}$, we therefore have an algebra homomorphism $F_{\mathbf{\Sigma}}\mathbf{X} \to \mathbf{P}$. This necessarily sends $t \in (F_{\mathbf{\Sigma}}\mathbf{X})(\Gamma; A)$ to $f^\dagger_{\Gamma;A}(t) \in P(\Gamma; A)$. ◀

We give two corollaries of this induction principle. The first is for proving properties of closed terms, which take the form of families of subsets $P(A) \subseteq Y(\diamond; A)$. Given such a family P, let $P(A_1, \ldots, A_n)$ be the set of all $\boldsymbol{\sigma} \in Y(\diamond; A_1, \ldots, A_n)$ such that $\sigma_i \in P(A_i)$ for all $i \leq n$, and define a predicate $P^\sharp$ over $\mathbf{Y}$ by $P^\sharp(\Gamma; A) = \{t \in Y(\Gamma; A) \mid \forall \boldsymbol{\sigma} \in P(\Gamma).\, t[\boldsymbol{\sigma}] \in P(A)\}$. Applying the induction principle above to $P^\sharp$ gives us the following.

▶ **Corollary 27.** *Suppose that $\mathbf{\Sigma}$ is an S-sorted second-order presentation, that $(\mathbf{Y}, [\![-]\!])$ is a $\mathbf{\Sigma}$-algebra, and that $(P(A) \subseteq Y(\diamond; A))_{A \in S}$ is a family of subsets. For every S-sorted clone $\mathbf{X}$ and clone homomorphism $f : \mathbf{X} \to \mathbf{Y}$, if*

- *for every operator $\mathsf{o} \in \Sigma((\Delta_1; A_1), \ldots, (\Delta_n; A_n); B)$ and tuple $(t_i \in P^\sharp(\Delta_i; A_i))_{i \leq n}$ of terms, we have $[\![\mathsf{o}]\!]_\diamond(t_1, \ldots, t_n) \in P(B)$;*
- *for every term $t \in X(\Gamma; B)$, we have $f_{\Gamma;B}(t) \in P^\sharp(\Gamma; B)$,*

then, for every type $A \in S$ and free term $t \in (F_{\mathbf{\Sigma}}\mathbf{X})(\diamond; A)$, we have $f^\dagger_{\diamond;A}(t) \in P(A)$.

Proof. $P^\sharp(\diamond; A) = P(A)$, so it suffices to apply Theorem 26 to the predicate $P^\sharp$. We therefore check the two assumptions of that theorem. Closure of $P^\sharp$ under f is immediate; and $P^\sharp$ is closed under operators because, if $(t_i \in P^\sharp(\Gamma, \Delta_i; A_i))_i$ and $\boldsymbol{\sigma} \in P(\Gamma)$, then $t_i[\mathsf{lift}_{\Delta_i}\boldsymbol{\sigma}] \in P^\sharp(\Delta_i; A_i)$ for all $i \leq n$, so that $[\![\mathsf{o}]\!]_\Gamma(t_1, \ldots, t_n)[\boldsymbol{\sigma}] = [\![\mathsf{o}]\!]_\diamond(t_1[\mathsf{lift}_{\Delta_1}\boldsymbol{\sigma}], \ldots, t_n[\mathsf{lift}_{\Delta_n}\boldsymbol{\sigma}]) \in P(B)$. ◀

Families of subsets $P(A) \subseteq X(\diamond; A)$ are closed under arbitrary conjunction and disjunction, complements, and universal and existential quantification. They form a *tripos* [25, 34], and hence a model of higher-order logic over $\mathbf{Clone}(S)$; the tripos-theoretic methods of Hofmann [22] carry over in this way to the setting of abstract clones.

The second corollary is for families of subsets $P(\Gamma; A) \subseteq Y(\Gamma; A)$ that are not known to be closed under substitution. (In some cases proving closure under substitution requires an induction over terms, but induction over terms is what this section is meant to enable.) Analogously to the construction $P^\sharp$ for predicates over closed terms, we define a predicate $P^\flat$ over $\mathbf{Y}$ by $P^\flat(\Gamma; A) = \{t \in Y(\Gamma; A) \mid \forall \Delta, \boldsymbol{\sigma} \in P(\Delta; \Gamma).\, t[\boldsymbol{\sigma}] \in P(\Delta; A)\}$.

▶ **Corollary 28.** *Suppose that $\boldsymbol{\Sigma}$ is an S-sorted second-order presentation, that $(\mathbf{Y}, [\![-]\!])$ is a $\boldsymbol{\Sigma}$-algebra, and that $(P(\Gamma; A) \subseteq Y(\Gamma; A))_{(\Gamma;A) \in S^* \times S}$ is a family of subsets. For every S-sorted clone $\mathbf{X}$ and homomorphism $f : \mathbf{X} \to \mathbf{Y}$, if*

- *for every context Γ we have $\mathsf{var}^{(\Gamma)} \in P(\Gamma; \Gamma)$;*
- *for every context Γ, operator $\mathsf{o} \in \Sigma((\Delta_1; A_1), \ldots, (\Delta_n; A_n); B)$, and tuple of terms $(t_i \in P^\flat(\Gamma, \Delta_i; A_i))_i$ we have $[\![\mathsf{o}]\!]_\Gamma(t_1, \ldots, t_n) \in P^\flat(\Gamma; B)$;*
- *for every term $t \in X(\Gamma; B)$ we have $f_{\Gamma;B}(t) \in P^\flat(\Gamma; B)$,*

then, for every free term $t \in (F_{\boldsymbol{\Sigma}}\mathbf{X})(\Gamma; A)$, we have $f^\dagger_{\Gamma;A}(t) \in P(\Gamma; A)$.

Proof. We can apply Theorem 26 to $P^\flat$ because it is closed under operators and under f. Hence $f^\dagger_{\Gamma;A}(t) \in P^\flat(\Gamma; A)$ for each $t \in (F_{\boldsymbol{\Sigma}}\mathbf{X})(\Gamma; A)$, and so $\mathsf{var}^{(\Gamma)} \in P(\Gamma; \Gamma)$ implies that $f^\dagger_{\Gamma;A}(t) = (f^\dagger_{\Gamma;A}(t))[\mathsf{var}^{(\Gamma)}] \in P(\Gamma; A)$. ◀

The above corollaries are designed to enable logical relations arguments, in which the fundamental lemma is proven using an induction hypothesis that quantifies over substitutions. In particular, in Corollary 28 we require $P^\flat$ to be closed under the operators, rather than P. There is a third corollary that instead requires closure of P under operators (this would essentially be the principle of induction on $\Gamma \vdash_{\mathbf{X}} t : A$), but this is less useful for our purposes.

6 Logical relations

We provide two extended examples of proofs using the induction principles of the previous section, both involving the presentation $\boldsymbol{\Sigma}^{\Lambda_{\beta\eta}}$ of the STLC with $\beta\eta$-equality. The first is a proof of the adequacy of the set-theoretic model of the STLC, which uses induction on closed terms; the second is a proof that every STLC term is $\beta\eta$-equal to one in normal form, using induction on open terms. Both examples are logical relations proofs, the former using ordinary logical relations and the latter using Kripke relations [27]. Though both properties are known to hold, these proofs in particular illustrate that our induction principles are powerful enough to justify logical relations arguments. We include a proof of normalization for the STLC with global state in Appendix A, as a further motivating example.

6.1 Closed terms and adequacy

We say that a model $\mathcal{M}$ of the STLC is *adequate* when, for all closed terms t and u of the base type b, if $\mathcal{M}[\![t]\!] = \mathcal{M}[\![u]\!]$, then t and u are equal up to $\beta\eta$-equality. (In adequate models, equality of denotations implies observational equivalence for terms of arbitrary types.)

We first show that we can perform logical relations arguments for the STLC using our induction principle: specifically Corollary 27. Fix a $\mathbf{\Sigma}^{\Lambda_{\beta\eta}}$-algebra $(\mathbf{Y}, [\![-]\!])$, homomorphism $f : \mathbf{X} \to \mathbf{Y}$ from some clone $\mathbf{X}$, and a subset $P(\mathsf{b}) \subseteq Y(\diamond; \mathsf{b})$ of closed terms of base type. We extend P to a family of subsets $P(A) \subseteq Y(\diamond; A)$ in the standard way for logical relations:

$$P(A \Rightarrow B) = \{t \in Y(\diamond; A{\Rightarrow}B) \mid \forall a \in P(A).\, [\![\mathsf{app}]\!]_\diamond(t, a) \in P(B)\}$$

Applying Corollary 27 to P gives us the following:

▶ **Lemma 29.** *If, for every context Γ and term $t \in X(\Gamma; B)$, we have $f_{\Gamma;B}(t) \in P^\sharp(\Gamma; B)$, then, for every free term $t \in (F_{\mathbf{\Sigma}^{\Lambda_{\beta\eta}}}\mathbf{X})(\diamond; A)$, we have $f^\dagger_{\diamond;A}(t) \in P(A)$.*

Proof. The only non-trivial assumption of Corollary 27 is closure under operators. Closure under app is immediate from the definition of the logical relation. Closure under abs holds because, if $t \in P^\sharp(A; B)$, then, for all $a \in P(A)$, we have $[\![\mathsf{app}]\!]_\diamond([\![\mathsf{abs}]\!]_\diamond(t), a) = t[a] \in P(B)$ using the β law, so that $[\![\mathsf{abs}]\!]_\diamond(t) \in P(A \Rightarrow B)$. ◀

Note that if terms are generated only by λ-abstraction and application then there are no closed terms of base type. For a more interesting example, we therefore consider the STLC with booleans (where the base type b is the type of booleans). Consider the Ty-sorted first-order presentation $\mathbf{\Sigma}_{\mathbf{Bool}}$ with two $(\diamond; \mathsf{b})$-ary operators true, false, and, for each $A \in \mathsf{Ty}$, a $(\mathsf{b}, A, A; A)$-ary operator ite ("if-then-else"), along with two equations:

$$y : A, z : A \vdash \mathsf{ite}(\mathsf{true}(), y, z) \approx y : A \qquad y : A, z : A \vdash \mathsf{ite}(\mathsf{false}(), y, z) \approx z : A$$

Let $\mathbf{Bool}$ be the Ty-sorted clone that is presented by $\mathbf{\Sigma}_{\mathbf{Bool}}$.

Consider the free $\mathbf{\Sigma}^{\Lambda_{\beta\eta}}$-algebra $F_{\mathbf{\Sigma}^{\Lambda_{\beta\eta}}}\mathbf{Bool}$, and the $\mathbf{\Sigma}^{\Lambda_{\beta\eta}}$-algebra $\mathcal{M}_{\mathbb{B}}$ (as defined in Example 21) with $\mathbb{B} = \{\mathrm{tt}, \mathrm{ff}\}$. The former should be thought of as containing the terms of the STLC with booleans (we make this precise below); the latter is the usual model in $\mathbf{Set}$. Both have clone homomorphisms from $\mathbf{Bool}$: the free algebra has $\eta_{\mathbf{Bool}} : \mathbf{Bool} \to F_{\mathbf{\Sigma}^{\Lambda_{\beta\eta}}}\mathbf{Bool}$; the model $\mathcal{M}_{\mathbb{B}}$ has the unique $g : \mathbf{Bool} \to \mathcal{M}_{\mathbb{B}}$ such that

$$g_{\Gamma;\mathsf{b}}(\mathsf{true}()) = \zeta \mapsto \mathrm{tt} \qquad g_{\Gamma;\mathsf{b}}(\mathsf{false}()) = \zeta \mapsto \mathrm{ff}$$

$$g_{\Gamma;A}(\mathsf{ite}(t_1, t_2, t_3)) = \zeta \mapsto \begin{cases} g_{\Gamma;A}(t_2)(\zeta) & \text{if } g_{\Gamma;\mathsf{b}}(t_1)(\zeta) = \mathrm{tt} \\ g_{\Gamma;A}(t_3)(\zeta) & \text{if } g_{\Gamma;\mathsf{b}}(t_1)(\zeta) = \mathrm{ff} \end{cases}$$

The algebra homomorphism $g^\dagger : F_{\mathbf{\Sigma}^{\Lambda_{\beta\eta}}}\mathbf{Bool} \to \mathcal{M}_{\mathbb{B}}$ gives the interpretation of STLC terms in the model. Define a subset $P(\mathsf{b}) \subseteq (F_{\mathbf{\Sigma}^{\Lambda_{\beta\eta}}}\mathbf{Bool} \times \mathcal{M}_{\mathbb{B}})(\diamond; \mathsf{b}) = (F_{\mathbf{\Sigma}^{\Lambda_{\beta\eta}}}\mathbf{Bool})(\diamond; \mathsf{b}) \times \mathbb{B}$ by

$$P(\mathsf{b}) = \{(\eta_{\mathbf{Bool}}(\mathsf{true}()), \mathrm{tt}), (\eta_{\mathbf{Bool}}(\mathsf{false}()), \mathrm{ff})\}$$

This extends to a logical relation P by the definition on function types above and, by a simple proof, satisfies the precondition of Lemma 29, where the clone homomorphism f is $\langle \eta_{\mathbf{Bool}}, g\rangle : \mathbf{Bool} \to F_{\mathbf{\Sigma}^{\Lambda_{\beta\eta}}}\mathbf{Bool} \times \mathcal{M}_{\mathbb{B}}$. Hence, for all $t \in (F_{\mathbf{\Sigma}^{\Lambda_{\beta\eta}}}\mathbf{Bool})(\diamond; A)$, we have $(t, g^\dagger_{\diamond;A}(t)) = \langle \eta_{\mathbf{Bool}}, g\rangle^\dagger_{\diamond;A}(t) \in P(A)$. When $A = \mathsf{b}$ this immediately implies, for all $t, t' \in (F_{\mathbf{\Sigma}^{\Lambda_{\beta\eta}}}\mathbf{Bool})(\diamond; \mathsf{b})$, that if $g^\dagger_{\diamond;\mathsf{b}}(t) = g^\dagger_{\diamond;\mathsf{b}}(t')$ then $t = t'$.

This last property is seen to be adequacy of the set-theoretic model $\mathcal{M}_{\mathbb{B}}$ as follows. Let $\mathbf{\Lambda}_{\beta\eta\mathbf{Bool}}$ be the Ty-sorted clone that is defined in the same way as $\mathbf{\Lambda}_{\beta\eta}$ (Example 5) but with additional term formers and equations (omitting the typing constraints on equations):

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{b}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{b}} \qquad \frac{\Gamma \vdash t_1 : \mathsf{b} \quad \Gamma \vdash t_2 : A \quad \Gamma \vdash t_3 : A}{\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : A} \qquad \begin{array}{l} \mathsf{if\ true\ then}\ t_2\ \mathsf{else}\ t_3 \approx_{\beta\eta} t_2 \\ \mathsf{if\ false\ then}\ t_2\ \mathsf{else}\ t_3 \approx_{\beta\eta} t_3 \end{array}$$

$\mathbf{\Lambda}_{\beta\eta\mathbf{Bool}}$ forms an $\mathbf{\Sigma}^{\Lambda_{\beta\eta}}$-algebra, and there is a clone homomorphism $\eta : \mathbf{Bool} \to \mathbf{\Lambda}_{\beta\eta}$ making it into the free $\mathbf{\Sigma}^{\Lambda_{\beta\eta}}$-algebra on $\mathbf{Bool}$. Hence we can apply the method above with $F_{\mathbf{\Sigma}^{\Lambda_{\beta\eta}}}\mathbf{Bool} = \mathbf{\Lambda}_{\beta\eta\mathbf{Bool}}$. The algebra homomorphism $g^\dagger : \mathbf{\Lambda}_{\beta\eta\mathbf{Bool}} \to \mathcal{M}_{\mathbb{B}}$ sends each term $\Gamma \vdash t : A$ to its interpretation as a function $\prod_i \mathcal{M}_{\mathbb{B}}[\![\Gamma_i]\!] \to \mathcal{M}_{\mathbb{B}}[\![A]\!]$. Adequacy is therefore exactly the property that $g^\dagger_{\diamond;\mathsf{b}}(t) = g^\dagger_{\diamond;\mathsf{b}}(t')$ implies $t = t'$.

6.2 Open terms and normalization

As a second example, we show that every term of the STLC is equal (up to $\beta\eta$-equality) to one in *η-long β-normal form* (we define these normal forms below). The proof mostly follows Fiore [10], except that we reason abstractly using the universal property of free algebras via our induction principle. It makes use of *Kripke logical relations* (with varying arity), which were introduced by Jung and Tiuryn [27] to study λ-definability.

We first show that our induction principle enables arguments using Kripke logical relations over the STLC. Fix a $\mathbf{\Sigma}^{\Lambda_{\beta\eta}}$-algebra $(\mathbf{Y}, [\![-]\!])$, homomorphism $f : \mathbf{X} \to \mathbf{Y}$ from a clone $\mathbf{X}$, and a subset $P(\Gamma; \mathsf{b}) \subseteq Y(\Gamma; \mathsf{b})$ for each Γ. We extend P from the base type b to all types by

$$P(\Gamma; A{\Rightarrow}B) = \{t{\in}Y(\Gamma; A{\Rightarrow}B) \mid \forall\Delta, \boldsymbol{\rho}{\in}\mathsf{Var}_S(\Delta;\Gamma), a{\in}P(\Delta;A).\ [\![\mathsf{app}]\!]_\Delta(t[\rhd\boldsymbol{\rho}], a) \in P(\Delta;B)\}$$

This is the standard definition of a Kripke logical relation on function types (other than using all renamings $\boldsymbol{\rho}$ rather than just weakenings, which is inessential). We therefore have a family of subsets $P(\Gamma; A) \subseteq Y(\Gamma; A)$, to which we apply Corollary 28 and obtain the following.

▶ **Lemma 30.** *If the family of subsets P satisfies*

- *for every context Γ we have $\mathsf{var}^{(\Gamma)} \in P(\Gamma;\Gamma)$;*
- *for every variable renaming $\boldsymbol{\rho} \in \mathsf{Var}_S(\Delta;\Gamma)$ and term $t \in P(\Gamma;\mathsf{b})$ we have $t[\rhd\boldsymbol{\rho}] \in P(\Delta;\mathsf{b})$;*
- *for every term $t \in X(\Gamma; B)$ and substitution $\boldsymbol{\sigma} \in P(\Delta;\Gamma)$ we have $(f_{\Gamma;B}t)[\boldsymbol{\sigma}] \in P(\Delta; B)$,*

then, for every free term $t \in (F_{\mathbf{\Sigma}^{\Lambda_{\beta\eta}}}\mathbf{X})(\Gamma; A)$, we have $f^\dagger_{\Gamma;A}(t) \in P(\Gamma; A)$.

Proof. The only non-trivial assumption of Corollary 28 is closure under operators. For closure under app, if $t \in P^\flat(\Gamma; A \Rightarrow B)$ and $u \in P^\flat(\Gamma; A)$, then, for all $\boldsymbol{\sigma} \in P(\Delta;\Gamma)$, we have $([\![\mathsf{app}]\!]_\Gamma(t,u))[\boldsymbol{\sigma}] = [\![\mathsf{app}]\!]_\Delta(t[\boldsymbol{\sigma}], u[\boldsymbol{\sigma}])$, because interpretations of operators commute with substitution; this is an element of $P(\Delta; B)$ using $t[\boldsymbol{\sigma}] \in P(\Delta; A \Rightarrow B)$ on the identity variable-renaming. For closure under abs, suppose that $t \in P^\flat(\Gamma, A; B)$. The assumption of the present lemma that P is closed under variable renamings at the base type b extends to all types A by an easy induction on A. For every $\boldsymbol{\sigma} \in P(\Delta;\Gamma)$, $\boldsymbol{\rho} \in \mathsf{Var}_S(\Xi;\Delta)$, and $a \in P(\Xi; A)$, we then have that $t[(\boldsymbol{\sigma} \circ \rhd\boldsymbol{\rho}), a] \in P(\Xi; B)$. Preservation of substitution by $[\![\mathsf{abs}]\!]$, and the β law, together imply that $[\![\mathsf{app}]\!]_\Xi(([\![\mathsf{abs}]\!]_\Gamma(t))[\boldsymbol{\sigma}][\rhd\boldsymbol{\rho}]) = t[(\boldsymbol{\sigma} \circ \rhd\boldsymbol{\rho}), a] \in P(\Xi; B)$. Hence $[\![\mathsf{abs}]\!]_\Gamma(t) \in P^\flat(\Gamma; A \Rightarrow B)$ as required. ◀

We use this to show normalization as follows. *Normal forms* $\Gamma \vdash_n t : A$ are defined mutually inductively with the *neutral forms* $\Gamma \vdash_m t : A$ by the following rules:

$$\frac{}{\Gamma, x : A, \Delta \vdash_m x : A} \qquad \frac{\Gamma \vdash_m f : A \Rightarrow B \quad \Gamma \vdash_n a : A}{\Gamma \vdash_m \mathsf{app}\,f\,a : B} \qquad \frac{\Gamma \vdash_m t : \mathsf{b}}{\Gamma \vdash_n t : \mathsf{b}} \qquad \frac{\Gamma, x : A \vdash_n t : B}{\Gamma \vdash_n \lambda x : A.\,t : A \Rightarrow B}$$

Consider the initial $\mathbf{\Sigma}^{\Lambda_{\beta\eta}}$-algebra, which is the clone $\mathbf{\Lambda}_{\beta\eta}$ of STLC terms up to $\approx_{\beta\eta}$. We write $\mathsf{Nf}(\Gamma; A)$ for the subset of STLC terms that are equivalent to a term in normal form under $\approx_{\beta\eta}$; and likewise write $\mathsf{Ne}(\Gamma; A)$ for neutral forms. We consider both as subsets of $\Lambda_{\beta\eta}(\Gamma; A)$; both are closed under variable renaming. The family of subsets we consider, $P(\Gamma; A) \subseteq \Lambda_{\beta\eta}(\Gamma; A)$, is defined on the base type as $P(\Gamma; \mathsf{b}) = \mathsf{Nf}(\Gamma; \mathsf{b})$, and on other types by the logical relations definition above. By a simple induction on the sort A, one can show that $\mathsf{Ne}(\Gamma; A) \subseteq P(\Gamma; A) \subseteq \mathsf{Nf}(\Gamma; A)$ (e.g. as in [10]). Since variables are neutral, this tells us in particular that $\mathsf{var}^{(\Gamma)} \in P(\Gamma; \Gamma)$ for all Γ. It then follows from Lemma 30 that $t = (\triangleright)^{\dagger}(t) \in P(\Gamma; A) \subseteq \mathsf{Nf}(\Gamma; A)$ for all $t \in \Lambda_{\beta\eta}(\Gamma; A)$, and so that every term of the STLC is $\beta\eta$-equivalent to one in normal form.

7 Comparison to other approaches

While we promote abstract clones as an elementary approach to simple type theories (qua multisorted second-order abstract syntax), there are several equivalent concepts that have been used to similar effect. We give a brief overview of the existing literature on the subject and a comparison with our work; we give references where possible, but unfortunately some of the relationships here exist only in the mathematical folklore.

Presheaves and substitution monoids

The study of second-order abstract syntax was initiated by Fiore et al. [16, 10], who represent term structure using presheaf categories. In their setting, one considers functors $T : \mathbb{L}(S)^{\mathrm{op}} \to \mathbf{Set}^S$, where $\mathbb{L}(S)$ is the category in which objects are contexts Γ, and morphisms $\boldsymbol{\rho} : \Delta \to \Gamma$ are variable renamings $\boldsymbol{\rho} \in \mathsf{Var}_S(\Gamma; \Delta)$ (recall Section 2.1). The S-indexed sets $T(\Gamma)$ consist of the sorted terms in context Γ; while the functions $T(\boldsymbol{\rho})$ rename the variables inside the terms to change their context. Substitution is accounted for by considering the monoidal structure $(\bullet, V)$ on $[\mathbb{L}(S)^{\mathrm{op}}, \mathbf{Set}^S]$, in which $T \bullet T'$ represents (for each context Γ) the simultaneous substitution of each variable in T with a term from T', and V represents the variables in each context. Monoids with respect to this structure are equipped with variables and substitution operations; they are equivalently abstract clones [16, Proposition 3.4]. Fiore and Hur [13] define $\mathbf{\Sigma}$-algebras as monoids in $[\mathbb{L}(S)^{\mathrm{op}}, \mathbf{Set}^S]$ equipped with interpretations of the operators of a presentation $\mathbf{\Sigma}$ satisfying its equations; they are equivalent to our $\mathbf{\Sigma}$-algebras. Our setting is therefore equivalent to that of Fiore et al. The advantage of our approach is that abstract clones require less categorical machinery; for those comfortable with category theory, this will be less of a concern.

There are some technical differences with previous work. Fiore and Hur [13] show the existence of the free $\mathbf{\Sigma}$-algebras on each presheaf T; in light of our free algebra result, the construction of the free algebra on T can be factored into two steps: constructing the free clone $\mathbf{X}$ on T by freely adding variables and substitution, and then taking the free $\mathbf{\Sigma}$-algebra on the clone $\mathbf{X}$. In our examples above, we begin with a clone that admits substitution, and hence do not freely add substitution. In a separate treatment, Hofmann [22] gives an induction principle for the λ-calculus using presheaves, but only considers predicates over closed terms; we obtain induction for closed terms as a corollary of induction over open terms.

Cartesian multicategories

Each abstract clone $\mathbf{X}$ has an identity operation for every sort B, given by the unique variable projection $\mathsf{var}_1^{([B])} \in X(B; B)$, along with admissible operations of exchange, weakening, and contraction. In this way, the sets of terms $X(\Gamma; A)$ form the structure of a *cartesian* *multicategory* with object set S (intuitively a category whose morphisms may have multiple inputs, subject to the structural properties of first-order equational logic). Conversely, every cartesian multicategory gives rise to an abstract clone. Thus, one could carry out the development of this paper in the context of cartesian multicategories (cf. [5, Section 9]). Clones are our preferred choice, because the definition of clone (in which projections are the primary operation) provides a more minimal axiomatisation than that of cartesian multicategory (in which the structural operations are primary). Note that one-object cartesian multicategories are usually called *cartesian operads*, which correspond to monosorted abstract clones.

Algebraic theories

The traditional approach to describing first-order algebraic structure in categorical logic is through *algebraic theories* [29]. An algebraic theory is represented by a category with cartesian products, which permit the multimorphisms of a cartesian multicategory to be represented by morphisms from a product: for a context $[A_1, \ldots, A_n]$, the terms $x_1 : A_1, \ldots, x_n : A_n \vdash t : B$ are represented by a hom-set $X(A_1 \times \cdots \times A_n, B)$. The relationship between cartesian multicategories and algebraic theories is the notion of *representability* for cartesian multicategories [33]. Second-order structure in the context of algebraic theories is captured by *second-order algebraic theories* [14, 32, 6], which generalize the first-order setting by introducing exponential objects that represent function types. Every second-order presentation $\mathbf{\Sigma}$ induces a second-order algebraic theory, the algebras for which are given by taking coslices over $\mathbf{\Sigma}$ [6].

Monads and relative monads

There is a classical correspondence in category theory between algebraic theories and certain monads on the category of sets [31], which in turn are equivalent to J-relative monads, for J the inclusion of finite sets into sets [4]. This has led to a line of investigation in which monads are used directly for second-order abstract syntax [20, 21, 2, 3, 19]. There are strong connections between this approach and that of presheaves and substitution monoids: for a detailed comparison, see the thesis of Zsidó [38]. In particular, the distinction between abstract clones and J-relative monads is slight, and the results of our development could equivalently be rephrased as statements about relative monads (cf. [6]).

8 Conclusion

We have shown that the abstract syntax of simple type theories has an elementary treatment using abstract clones. The framework we describe allows the specification of the terms and equations of type theories via second-order presentations [13, 14]. Free algebras then give the syntax along with an accompanying induction principle, which we show enables abstract proofs of non-trivial properties such as adequacy. We emphasize that abstract clones axiomatize the syntax only of *simple* type theories: clones cannot express linear types, dependent types, or type theories in which variables stand only for certain classes of term (e.g. polarized type theories [37], and the call-by-value λ-calculus). In some cases, analogous structures are already known (for instance, symmetric multicategories for linear type theories [35, 23]); for others, such as dependent type theories, this remains an open problem.

References

1 Peter Aczel. A general Church–Rosser theorem. Unpublished manuscript, 1978.

2 Benedikt Ahrens. Modules over relative monads for syntax and semantics. *Mathematical Structures in Computer Science*, 26(1):3–37, 2016. doi:10.1017/S0960129514000103.

3 Benedikt Ahrens, André Hirschowitz, Ambroise Lafont, and Marco Maggesi. Modular specification of monads through higher-order presentations. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, volume 131, pages 1–16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.FSCD.2019.6.

4 Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Monads need not be endofunctors. In *Foundations of Software Science and Computational Structures*, pages 297–311. Springer, 2010. doi:10.1007/978-3-642-12032-9_21.

5 Nathanael Arkor and Marcelo Fiore. Algebraic models of simple type theories: a polynomial approach. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, page 88–101. Association for Computing Machinery, 2020. doi:10.1145/3373718.3394771.

6 Nathanael Arkor and Dylan McDermott. Higher-order algebraic theories. *Preprint*, 2020. URL: https://www.cl.cam.ac.uk/~na412/Higher-order%20algebraic%20theories.pdf.

7 Michael Barr. Exact categories. In *Exact categories and categories of sheaves*, pages 1–120. Springer, 1971.

8 Garrett Birkhoff. On the structure of abstract algebras. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 31, pages 433–454. Cambridge University Press, 1935. doi:10.1017/S0305004100013463.

9 Garrett Birkhoff and John D Lipson. Heterogeneous algebras. *Journal of Combinatorial Theory*, 8(1):115–133, 1970. doi:10.1016/S0021-9800(70)80014-X.

10 Marcelo Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 26–37. Association for Computing Machinery, 2002. doi:10.1145/571157.571161.

11 Marcelo Fiore. Second-order and dependently-sorted abstract syntax. In *Proceedings of the 23rd Annual IEEE Symposium on Logic in Computer Science*, pages 57–68. IEEE, 2008. doi:10.1109/LICS.2008.38.

12 Marcelo Fiore and Makoto Hamana. Multiversal polymorphic algebraic theories: syntax, semantics, translations, and equational logic. In *Proceedings of the 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 520–529. IEEE, 2013. doi:10.1109/LICS.2013.59.

13 Marcelo Fiore and Chung-Kil Hur. Second-order equational logic. In *Computer Science Logic*, pages 320–335. Springer, 2010. doi:10.1007/978-3-642-15205-4_26.

14 Marcelo Fiore and Ola Mahmoud. Second-order algebraic theories. In *Mathematical Foundations of Computer Science*, pages 368–380. Springer, 2010. doi:10.1007/978-3-642-15155-2_33.

15 Marcelo Fiore and Ola Mahmoud. Functorial semantics of second-order algebraic theories, 2014. arXiv:1401.4697.

16 Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*, pages 193–202. IEEE, 1999. doi:10.1109/LICS.1999.782615.

17 Joseph Goguen and José Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics*, 11(3):307–334, 1985.

18 Makoto Hamana. Free σ-monoids: A higher-order syntax with metavariables. In *Asian Symposium on Programming Languages and Systems*, pages 348–363. Springer, 2004. doi:10.1007/978-3-540-30477-7_23.

19 André Hirschowitz, Tom Hirschowitz, and Ambroise Lafont. Modules over monads and operational semantics. In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, volume 167, pages 12–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.FSCD.2020.12.

20 André Hirschowitz and Marco Maggesi. Modules over monads and linearity. In *Logic, Language, Information, and Computation*, pages 218–237. Springer, 2007. doi:10.1007/978-3-540-73445-1_16.

21 André Hirschowitz and Marco Maggesi. Modules over monads and initial semantics. *Information and Computation*, 208(5):545–564, 2010. doi:10.1016/j.ic.2009.07.003.

22 Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*, pages 204–213. IEEE, 1999. doi:10.1109/LICS.1999.782616.

23 Mathieu Huot. Operads with algebraic structure. *MPRI Internship Report*, 2016. URL: http://users.ox.ac.uk/~scro3639/M1_Report.pdf.

24 J.M.E. Hyland. Classical lambda calculus in modern dress. *Mathematical Structures in Computer Science*, 27(5):762–781, 2017. doi:10.1017/S0960129515000377.

25 J.M.E. Hyland, P.T. Johnstone, and A.M. Pitts. Tripos theory. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 88, pages 205–232. Cambridge University Press, 1980. doi:10.1017/S0305004100057534.

26 Bart Jacobs. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, 1999.

27 Achim Jung and Jerzy Tiuryn. A new characterization of lambda definability. In *Typed Lambda Calculi and Applications*, pages 245–257. Springer, 1993. doi:10.1007/BFb0037110.

28 J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1988.

29 F. William Lawvere. Functorial semantics of algebraic theories. *Proceedings of the National Academy of Sciences of the United States of America*, 50(5):869–872, 1963. doi:10.1073/pnas.50.5.869.

30 Daniel J. Lehmann and Michael B. Smyth. Algebraic specification of data types: A synthetic approach. *Mathematical systems theory*, 14(1):97–139, 1981. doi:10.1007/BF01752392.

31 F.E.J. Linton. An outline of functorial semantics. In *Seminar on Triples and Categorical Homology Theory*, pages 7–52. Springer, 1969. doi:10.1007/BFb0083080.

32 Ola Mahmoud. Second-order algebraic theories. Technical Report UCAM-CL-TR-807, University of Cambridge, Computer Laboratory, 2011. doi:10.48456/tr-807.

33 Claudio Pisani. Sequential multicategories. *Theory and Applications of Categories*, 29(19):496–541, 2014. URL: http://www.tac.mta.ca/tac/volumes/29/19/29-19abs.html.

34 Andrew M. Pitts. Tripos theory in retrospect. *Mathematical structures in computer science*, 12(3):265–279, 2002. doi:10.1017/S096012950200364X.

35 Miki Tanaka. Abstract syntax and variable binding for linear binders. In *Mathematical Foundations of Computer Science*, pages 670–679. Springer, 2000. doi:10.1007/3-540-44612-5_62.

36 Walter Taylor. Abstract clone theory. In *Algebras and orders*, volume 389 of *NATO ASI Series C*, pages 507–530. Springer, 1993. doi:10.1007/978-94-017-0697-1_11.

37 Noam Zeilberger. *The Logical Basis of Evaluation Order and Pattern-matching*. PhD thesis, Carnegie Mellon University, 2009.

38 Julianna Zsidó. *Typed abstract syntax*. PhD thesis, Université Nice Sophia Antipolis, 2010.

A Normalization with global state

As a further example of the application of abstract clones to problems motivated by simple type theories, we prove a normalization result for the STLC with V-valued global state: concretely, this calculus is given by the free algebra of the second-order presentation $\mathbf{\Sigma}^{\Lambda_{\beta\eta}}$

of the STLC with $\beta\eta$-equality on the clone $\mathbf{GS}_V$ of V-valued global state, whose syntax is described in Example 23. The proof is similar to normalization of the STLC without global state (Section 6.2); in particular, we reuse Lemma 30.

Recall that for $V = \{v_1, \ldots, v_k\}$, the free algebra consists of the syntax of the STLC extended by the additional term formers get and put_{v_i}. Normal and neutral forms are defined as in Section 6.2, except with

$$\text{the rule } \frac{\Gamma \vdash_m t : \mathsf{b}}{\Gamma \vdash_n t : \mathsf{b}} \text{ replaced by } \frac{\Gamma \vdash_m t_1 : \mathsf{b} \quad \cdots \quad \Gamma \vdash_m t_k : \mathsf{b}}{\Gamma \vdash_n \mathsf{get}(\mathsf{put}_{w_1}(t_1), \ldots, \mathsf{put}_{w_k}(t_k)) : \mathsf{b}} \ (w_1, \ldots, w_k \in V).$$

Again we write $\mathsf{Nf}(\Gamma; A)$ (respectively $\mathsf{Ne}(\Gamma; A)$) for the subsets of terms equal to a normal (respectively neutral) form, and define the logical relation $P(\Gamma; A)$ on the base type as $P(\Gamma; \mathsf{b}) = \mathsf{Nf}(\Gamma; \mathsf{b})$, and on other types by the logical relations definition in Section 6.2. Again we have $\mathsf{Ne}(\Gamma; A) \subseteq P(\Gamma; A) \subseteq \mathsf{Nf}(\Gamma; A)$ by induction on A; the only difference with the previous proof is that on base types one has $\mathsf{Ne}(\Gamma; \mathsf{b}) \subseteq \mathsf{Nf}(\Gamma; \mathsf{b})$, because for $t \in \mathsf{Ne}(\Gamma; \mathsf{b})$ we have $t \approx_{\beta\eta} \mathsf{get}(\mathsf{put}_{v_1}(t), \ldots, \mathsf{put}_{v_k}(t)) \in \mathsf{Nf}(\Gamma; \mathsf{b})$. To prove that every term is equal to one in normal form up to $\approx_{\beta\eta}$, it suffices to apply Lemma 30 with f the clone homomorphism $\eta_{\mathbf{GS}_V} : \mathbf{GS}_V \to F_{\mathbf{\Sigma}^{\Lambda_{\beta\eta}}} \mathbf{GS}_V$. The first two assumptions of the lemma have the same proof as before. For the third, since $\mathbf{GS}_V$ is presented by $\mathbf{\Sigma}_V^{\mathbf{GS}}$ and clone homomorphisms preserve variables and substitution, it suffices to show that

- for each $t_1, \ldots, t_k \in P(\Gamma; \mathsf{b})$, we have $\mathsf{get}(t_1, \ldots, t_k) \in P(\Gamma; \mathsf{b})$;
- for each $t \in P(\Gamma; \mathsf{b})$ and $i \leq k$, we have $\mathsf{put}_{v_i}(t) \in P(\Gamma; \mathsf{b})$.

The first statement holds because if $t_i = \mathsf{get}(\mathsf{put}_{w_{i1}}(t'_{i1}), \ldots, \mathsf{put}_{w_{ik}}(t'_{ik}))$ then

$$\mathsf{get}(t_1, \ldots, t_k) \approx_{\beta\eta} \mathsf{get}(\mathsf{put}_{w_{11}}(t'_{11}), \ldots, \mathsf{put}_{w_{kk}}(t'_{kk})) \in \mathsf{Nf}(\Gamma; \mathsf{b}) = P(\Gamma; \mathsf{b})$$

The second statement holds because if $t = \mathsf{get}(\mathsf{put}_{w_1}(t'_1), \ldots, \mathsf{put}_{w_k}(t'_k))$ then

$$\mathsf{put}_{v_i}(t) \ \approx_{\beta\eta} \ \mathsf{put}_{w_i}(t'_i) \ \approx_{\beta\eta} \ \mathsf{get}(\mathsf{put}_{w_i}(t'_i), \ldots, \mathsf{put}_{w_i}(t'_i)) \in \mathsf{Nf}(\Gamma; \mathsf{b}) = P(\Gamma; \mathsf{b})$$

Tuple Interpretations for Higher-Order Complexity

Cynthia Kop
Department of Software Science, Radboud University Nijmegen, The Netherlands

Deivid Vale
Department of Software Science, Radboud University Nijmegen, The Netherlands

Abstract

We develop a class of algebraic interpretations for many-sorted and higher-order term rewriting systems that takes type information into account. Specifically, base-type terms are mapped to *tuples* of natural numbers and higher-order terms to functions between those tuples. Tuples may carry information relevant to the type; for instance, a term of type nat may be associated to a pair $\langle \mathsf{cost}, \mathsf{size} \rangle$ representing its evaluation cost and size. This class of interpretations results in a more fine-grained notion of complexity than runtime or derivational complexity, which makes it particularly useful to obtain complexity bounds for higher-order rewriting systems.

We show that rewriting systems compatible with tuple interpretations admit finite bounds on derivation height. Furthermore, we demonstrate how to mechanically construct tuple interpretations and how to orient β and η reductions within our technique. Finally, we relate our method to runtime complexity and prove that specific interpretation shapes imply certain runtime complexity bounds.

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

Keywords and phrases Complexity, higher-order term rewriting, many-sorted term rewriting, polynomial interpretations, weakly monotonic algebras

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.31

Related Version An extended appendix with full proofs and additional examples is available at [32].
Extended Version: `https://arxiv.org/abs/2105.01112`

Funding The authors are supported by the NWO TOP project "ICHOR", NWO 612.001.803/7571 and the NWO VIDI project "CHORPE", NWO VI.Vidi.193.075.

1 Introduction

Term rewriting systems (TRSs) are a conceptually simple but powerful computational model. It is simple because computation is modelled straightforwardly by step-by-step applications of transformation rules. It is powerful in the sense that any algorithm can be expressed in it (Turing Completeness). These characteristics make TRSs a formalism well-suited as an abstract analysis language, for instance to study properties of functional programs. We can then define specific analysis techniques for each property of interest.

One such property is *complexity*. The study of complexity has long been a topic of interest in term rewriting [11, 27, 25, 7, 24, 35], as it both holds relations to computational complexity [3, 11, 12] and resource analysis [6, 13] and is highly challenging. Most commonly studied are the notions of runtime and derivational complexity, which capture the number of steps that may be taken when starting with terms of a given size and shape. In essence, this is a form of resource analysis which abstracts away from the true machine cost of reduction in a rewriting engine but still has a close relation to it [8, 18, 1, 12].

These notions do not obviously extend to the *higher-order* setting, however. In higher-order term rewriting, a term may represent a function; yet, the size of a function does not tell us much about its behaviour. Rather, properties such as "the function is size-increasing" may be more relevant. Clearly a more sophisticated complexity notion is needed.

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi; Article No. 31; pp. 31:1–31:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper we will propose a new method to analyse many-sorted and higher-order term rewriting systems, which can be used as a foundation to obtain a variety of complexity results. This method is based on *interpretations* in a monotonic algebra as also used for termination analysis [39, 22], where a term of function type is mapped to a monotonic function. Unlike [39, 22], we map a term of base type not to an integer, but rather to a vector of integers describing different values of interest in the term. This will allow us to reason separately about – for instance – the length of a list and the size of its greatest element, and to describe the behaviour of a term of function type in a fine-grained way.

This method is also relevant for termination analysis, since we essentially generalise and extend *matrix interpretations* [35] to higher-order rewriting. In addition, the technique may add some power to the arsenal of a complexity or termination analysis tool for first-order term rewriting; in particular *many-sorted* term rewriting due to the way we use type information.

A note on terminology. We use the word "complexity" as it is commonly used in term rewriting: a worst-case measure of the number of steps in a reduction. In this paper we do not address the question of true resource use or connections to computational complexity. In particular, we do not address the true cost of beta-reduction. This is left to future work.

Outline of the paper. We will start by recalling the definition of and fixing notation for many-sorted and higher-order term rewriting (§2). Then, we will define tuple interpretations for many-sorted first-order rewriting to explore the idea (§3), discuss our primary objective of *higher-order* tuple interpretations (§4), and relate our method to runtime complexity (§5). Finally, we will discuss related work (§6) and end with conclusions and future work (§7).

2 Preliminaries

We assume the reader is familiar with first-order term rewriting and λ-calculus. In this section, we fix notation and discuss the higher-order rewriting format used in the paper.

2.1 First-Order Many-Sorted Rewriting

Many-sorted term rewriting [38] is in principle the same as first-order term rewriting. The only difference is that we impose a sort system and limit interest to well-sorted terms.

Formally, we assume given a non-empty set of *sorts* $\mathcal{S}$. A *many-sorted signature* consists of a set $\mathcal{F}$ of function symbols together with two functions that map each symbol to a finite sequence of *input sorts* and an *output sort*. Fixing a many-sorted signature, we will denote $\mathsf{f} :: [\iota_1 \times \cdots \times \iota_k] \Rightarrow \kappa$ if $\mathsf{f} \in \mathcal{F}$ and f has input sorts $\iota_1, \dots, \iota_k$ and output sort κ. We also assume given a set $\mathcal{X} = \bigcup_{\iota \in \mathcal{S}} \mathcal{X}_\iota$ of variables disjoint from $\mathcal{F}$, such that all $\mathcal{X}_\iota$ are pairwise disjoint. The set $T_{fo}(\mathcal{F}, \mathcal{X})$ of *many-sorted terms* is inductively defined as the set of expressions s such that $s :: \kappa$ can be derived for some sort κ using the clauses:

$$x :: \kappa \text{ if } x \in \mathcal{X}_\kappa \qquad \mathsf{f}(s_1, \dots, s_k) :: \kappa \text{ if } \mathsf{f} :: [\iota_1 \times \cdots \times \iota_k] \Rightarrow \kappa \text{ and each } s_i :: \iota_i$$

If $s :: \kappa$, we call κ the sort of s. Substitutions, rewrite rules and reduction are defined as usual in first-order term rewriting, except that substitutions are sort-preserving (each variable is mapped to a term of the same sort) and both sides of a rule have the same sort. We omit these definitions, since they are a special case of the higher-order definitions in Section 2.2.

▶ **Example 1.** We fix nat and list for the sorts of natural numbers and lists of natural numbers, respectively; and a signature with the symbols: $\mathsf{0} :: \mathsf{nat}$ (this is shorthand notation for $[] \Rightarrow \mathsf{nat}$), $\mathsf{s} :: [\mathsf{nat}] \Rightarrow \mathsf{nat}$, $\mathsf{nil} :: \mathsf{list}$, $\mathsf{cons} :: [\mathsf{nat} \times \mathsf{list}] \Rightarrow \mathsf{list}$, $\mathsf{rev} :: [\mathsf{list}] \Rightarrow \mathsf{list}$,

$\mathsf{sum} :: [\mathsf{list}] \Rightarrow \mathsf{nat}$, $\mathsf{append} :: [\mathsf{list} \times \mathsf{list}] \Rightarrow \mathsf{list}$, and $\oplus :: [\mathsf{nat} \times \mathsf{nat}] \Rightarrow \mathsf{nat}$. The rules below compute well-known functions over lists and numbers. We follow the convention of using infix notation for cons and $\oplus$, i.e., $\mathsf{cons}(x, xs)$ is written $x : xs$ and $\oplus(x, y)$ is written $x \oplus y$.

$$\begin{array}{rlrl} x \oplus \mathsf{0} \to x & & \mathsf{sum}(\mathsf{nil}) \to \mathsf{0} \\ x \oplus \mathsf{s}(y) \to \mathsf{s}(x \oplus y) & & \mathsf{sum}(x : xs) \to \mathsf{sum}(xs) \oplus x \\ \mathsf{append}(\mathsf{nil}, xs) \to xs & & \mathsf{rev}(\mathsf{nil}) \to \mathsf{nil} \\ \mathsf{append}(x : xs, ys) \to x : \mathsf{append}(xs, ys) & & \mathsf{rev}(x : xs) \to \mathsf{append}(\mathsf{rev}(xs), x : \mathsf{nil}) \end{array}$$

2.2 Higher-Order Rewriting

For higher-order rewriting, we will use *algebraic functional systems* (AFS), a slightly simplified form of a higher-order language introduced by Jouannaud and Okada [29]. This choice gives an easy presentation, as it combines algebraic definitions in a first-order style with a function mechanism using λ-abstractions and term applications.

Given a non-empty set of *sorts* $\mathcal{S}$, the set $\mathcal{ST}$ of *simple types* (or just *types*) is given by: (a) $\mathcal{S} \subseteq \mathcal{ST}$; (b) if $\sigma, \tau \in \mathcal{ST}$ then $\sigma \Rightarrow \tau \in \mathcal{ST}$. Types are denoted by σ, τ and sorts by ι, κ. A *higher-order signature* consists of a set $\mathcal{F}$ of function symbols together with two functions that map each symbol to a finite sequence of *input types* and an *output type*; fixing a signature, we denote this type information $\mathsf{f} :: [\sigma_1 \times \cdots \times \sigma_k] \Rightarrow \tau$. A function symbol is said to be higher-order if at least one of its input types or its output type is an arrow type.

We also assume given a set $\mathcal{X} = \bigcup_{\sigma \in \mathcal{ST}} \mathcal{X}_\sigma$ of variables disjoint from $\mathcal{F}$ (and pairwise disjoint) so that each $\mathcal{X}_\sigma$ is countably infinite. The set $T(\mathcal{F}, \mathcal{X})$ of terms is inductively defined as the set of expressions whose type can be derived using the following clauses:

$$\begin{array}{llll} x :: \sigma & \text{if } x \in \mathcal{X}_\sigma & (\lambda x.s) :: \sigma \Rightarrow \tau & \text{if } x \in \mathcal{X}_\sigma \text{ and } s :: \tau \\ (s\,t) :: \tau & \text{if } s :: \sigma \Rightarrow \tau \text{ and } \ :: \sigma & \mathsf{f}(s_1, \dots, s_k) :: \tau & \text{if } \mathsf{f} :: [\sigma_1 \times \cdots \times \sigma_k] \Rightarrow \tau \\ & & & \text{and each } s_i :: \sigma_i \end{array}$$

If $s :: \sigma$, we say that σ is the type of s. It is easy to see that each term has a unique type.

As in the λ-calculus, a variable x is *bound* in a term if it occurs in the scope of an abstractor $\lambda x.$; it is *free* otherwise. A term is called *closed* if it has no free variables and *ground* if it also has no bound variables. Term equality is modulo α-conversion and bound variables are renamed if necessary. Application is left-associative and has precedence over abstractions; for example, $\lambda x.s\,t\,u$ reads $\lambda x.((s\,t)\,u)$. A substitution is a finite, type-preserving mapping $\gamma : \mathcal{X} \to T(\mathcal{F}, \mathcal{X})$, typically denoted $[x_1 := s_1, \dots, x_n := t_n]$. Its *domain* $\{x_1, \dots, x_n\}$ is denoted $\mathtt{dom}(\gamma)$. A substitution γ is applied to a term s, notation $s\gamma$, by renaming all bound variables in s to fresh variables and then replacing each $x \in \mathtt{dom}(\gamma)$ by $\gamma(x)$. Formally:

$$\begin{array}{rclllrcll} x\gamma & = & \gamma(x) & \text{if } x \in \mathtt{dom}(\gamma) & & (s\,t)\gamma & = & (s\gamma)\,(t\gamma) & \\ x\gamma & = & x & \text{if } x \notin \mathtt{dom}(\gamma) & & \mathsf{f}(s_1, \dots, s_k)\gamma & = & \mathsf{f}(s_1\gamma, \dots, s_k\gamma) & \\ & & & & & (\lambda x.s)\gamma & = & \lambda y.(s([x := y]\gamma)) & \text{for } y \text{ fresh} \end{array}$$

Here, $[x := y]\gamma$ is the substitution that maps x to y and all variables in $\mathtt{dom}(\gamma)$ other than x to $\gamma(x)$. The result of $s\gamma$ is unique modulo α-renaming.

A *rewriting rule* is a pair of terms $\ell \to r$ of the same type such that all free variables of r also occur in ℓ. Given a set of rewriting rules $\mathcal{R}$, the rewrite relation induced by $\mathcal{R}$ on the set $T(\mathcal{F}, \mathcal{X})$ is the smallest monotonic relation that is stable under substitution and contains both all elements of $\mathcal{R}$ and β-reduction. That is, it is inductively generated by:

$$\begin{array}{rclll} (\lambda x.s)\ t & \rightarrow_{\mathcal{R}} & s[x := t] & & \\ \ell\gamma & \rightarrow_{\mathcal{R}} & r\gamma & \text{if } \ell \rightarrow r \in \mathcal{R} \\ \mathsf{f}(\dots, s, \dots) & \rightarrow_{\mathcal{R}} & \mathsf{f}(\dots, t, \dots) & \text{if } s \rightarrow_{\mathcal{R}} t \end{array} \qquad \begin{array}{rcll} \lambda x.s & \rightarrow_{\mathcal{R}} & \lambda x.t & \text{if } s \rightarrow_{\mathcal{R}} t \\ s\ u & \rightarrow_{\mathcal{R}} & t\ u & \text{if } s \rightarrow_{\mathcal{R}} t \\ u\ s & \rightarrow_{\mathcal{R}} & u\ t & \text{if } s \rightarrow_{\mathcal{R}} t \end{array}$$

Note that we do not, by default, include the common η-reduction rule scheme ("$\lambda x.s\ x \rightarrow_{\mathcal{R}} s$ if x is not a free variable in s"). We avoid this because not all sources consider it, and it is easy to add by including, for all types σ, τ, a rule $\lambda x.F\ x \rightarrow F$ with $F \in \mathcal{X}_{\sigma \Rightarrow \tau}$ in $\mathcal{R}$.

An *algebraic functional system* (AFS) is the combination of a set of terms $T(\mathcal{F}, \mathcal{X})$ and a rewrite relation $\rightarrow_{\mathcal{R}}$ over $T(\mathcal{F}, \mathcal{X})$. An AFS is typically given by supplying $\mathcal{F}$ and $\mathcal{R}$.

A *many-sorted term rewriting system* (TRS), as discussed in Section 2.1, is a pair $(T_{fo}(\mathcal{F}, \mathcal{X}), \rightarrow_{\mathcal{R}})$ where $\mathcal{F}$ is a many-sorted signature and $\rightarrow_{\mathcal{R}}$ a rewrite relation over $T_{fo}(\mathcal{F}, \mathcal{X})$. That is, it is essentially an AFS where we only consider first-order terms.

► **Example 2.** Following common examples in higher-order rewriting, we will use (as a running example) the AFS $(\mathcal{F}, \mathcal{R})_{\mathtt{fold}}$, with symbols $\mathsf{nil} :: \mathsf{list}$, $\mathsf{cons} :: [\mathsf{nat} \times \mathsf{list}] \Rightarrow \mathsf{list}$, $\mathsf{map} :: [(\mathsf{nat} \Rightarrow \mathsf{nat}) \times \mathsf{list}] \Rightarrow \mathsf{list}$, $\mathsf{foldl} :: [(\mathsf{nat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat}) \times \mathsf{nat} \times \mathsf{list}] \Rightarrow \mathsf{nat}$, and rules:

$$\begin{array}{rcl} \mathsf{foldl}(F, z, \mathsf{nil}) & \rightarrow & z \\ \mathsf{foldl}(F, z, x : xs) & \rightarrow & \mathsf{foldl}(F, (F\ z\ x), xs) \end{array} \qquad \begin{array}{rcl} \mathsf{map}(F, \mathsf{nil}) & \rightarrow & \mathsf{nil} \\ \mathsf{map}(F, x : xs) & \rightarrow & (F\ x) : \mathsf{map}(F, xs) \end{array}$$

2.3 Functions and orderings

An extended well-founded set is a tuple $(A, >, \geq)$ such that $>$ is a well-founded ordering on A; $\geq$ is a quasi-ordering on A; $x > y$ implies $x \geq y$; and $x > y \geq z$ implies $x > z$. Hence, it is permitted, but not required, that $\geq$ is the reflexive closure of $>$.

For sets A, B, the notation $A \Longrightarrow B$ denotes the set of functions from A to B. Function equality is extensional: for $f, g \in A \Longrightarrow B$ we say $f = g$ iff $f(x) = g(x)$ for all $x \in A$.

If $(A, >, \geq)$ and $(B, \succ, \succeq)$ are extended well-founded sets, we say that $f \in A \Longrightarrow B$ is *weakly monotonic* if $x \geq y$ implies $f(x) \succeq f(y)$. In addition, if $(A_1, >_1, \geq_1), \dots, (A_n, >_n, \geq_n)$ are all well-founded sets, we say that $f \in A_1 \times \dots \times A_n \Longrightarrow B$ is weakly monotonic if we have $f(x_1, \dots, x_n) \succeq f(y_1, \dots, y_n)$ whenever $x_i \geq_i y_i$ for all $1 \leq i \leq n$. We say that f is *strict* in argument j if $x_j >_j y_j$ (and also $x_i \geq_i y_i$ for all i) implies $f(x_1, \dots, x_n) \succ f(y_1, \dots, y_n)$.

We say that $f \in A_1 \times \dots \times A_n \Longrightarrow B$ is *strongly monotonic* if f is weakly monotonic and strict in all its arguments (and similar for $f \in A \Longrightarrow B$).

3 First-Order tuple interpretation

In this section, we will introduce the concept of tuple interpretations for many-sorted term rewriting. This is the core methodology which the higher-order theory is built on top of. This theory also has value by itself as a first-order termination and complexity technique.

It is common in the rewriting literature to use termination proofs to assess the *difficulty* of rewriting a term to normal form [7, 27]. The intuition comes from the idea that by ordering rewriting rules in descending order we gauge the order of magnitude of reduction. The same principle applies for syntactic [24, 25, 34] and semantic [27, 26, 35] termination proofs.

On the semantic side there is a natural strategy: given an extended well-founded set $\mathcal{A} = (A, >, \geq)$ find an interpretation from terms to elements of A so that $[\![s]\!] > [\![t]\!]$ whenever $s \rightarrow_{\mathcal{R}} t$. (This can typically be done by showing that $[\![\ell]\!] > [\![r]\!]$ for all rules $\ell \rightarrow r$). This interpretation holds information about the complexity of $(\mathcal{F}, \mathcal{R})$ since the maximum length of a reduction starting in a term s is bounded by number of $>$ steps that may be done starting in $[\![s]\!]$. If $[\![s]\!]$ is a natural number, this gives a bound immediately.

In the setting of many-sorted term rewriting, we may formally define this as follows.

▶ **Definition 3.** *Let $\mathcal{S}$ be a set of sorts and $\mathcal{F}$ an $\mathcal{S}$-signature. A many-sorted monotonic algebra $\mathcal{A}$ consists of a family of extended well-founded sets $(A_\iota, >_\iota, \geq_\iota)_{\iota \in \mathcal{S}}$ together with an interpretation $\mathcal{J}$ which associates to each $\mathsf{f} :: [\iota_1 \times \cdots \times \iota_k] \Rightarrow \kappa$ in $\mathcal{F}$ a strongly monotonic function $\mathcal{J}_\mathsf{f} \in A_{\iota_1} \times \cdots \times A_{\iota_k} \Longrightarrow A_\kappa$. Let α be a function that maps variables of sort ι to elements of A_ι. We extend $\mathcal{J}$ to a function $[\![\cdot]\!]_\alpha$ that maps terms of sort ι to elements of A_ι, by letting $[\![x]\!]_\alpha = \alpha(x)$ if x is a variable of sort ι, and $[\![\mathsf{f}(s_1, \dots, s_k)]\!]_\alpha = \mathcal{J}_\mathsf{f}([\![s_1]\!]_\alpha, \dots, [\![s_k]\!]_\alpha)$. We say that a TRS $(\mathcal{F}, \mathcal{R})$ is* compatible *with $\mathcal{A}$ if $[\![\ell]\!]_\alpha > [\![r]\!]_\alpha$ for all α and all $\ell \to r \in \mathcal{R}$.*

We will generally omit the subscript α when it is clear from context, writing $[\![s]\!]$ instead of $[\![s]\!]_\alpha$. In examples, we may write something like $[\![s]\!] = x + y$ to mean $[\![s]\!]_\alpha = \alpha(x) + \alpha(y)$.

▶ **Theorem 4.** *If $(\mathcal{F}, \mathcal{R})$ is compatible with $\mathcal{A}$ then for all α: $[\![s]\!]_\alpha > [\![t]\!]_\alpha$ whenever $s \to_\mathcal{R} t$.*

Proof Sketch. By induction on the size of s using strong monotonicity of each $\mathcal{J}_\mathsf{f}$. ◀

A common notion in the literature on complexity of term rewriting is *derivation height*:

$$\mathtt{dh}_\mathcal{R}(t) := \max\{n \in \mathbb{N} \mid \exists s.\, t \to^n s\}.$$

Intuitively, $\mathtt{dh}_\mathcal{R}(t)$ describes the worst-case number of steps for all possible reductions starting in t. If $(\mathcal{F}, \mathcal{R})$ is terminating, then $\mathtt{dh}_\mathcal{R}(\cdot)$ is a total function. If $(A_\iota, >_\iota) = (\mathbb{N}, >)$ then we easily see that $\mathtt{dh}_\mathcal{R}(t) \leq [\![t]\!]$ for any term $t : \iota$. Hence, $[\![\cdot]\!]$ can be used to bound the derivation height function. However, this may give a severe overestimation, as demonstrated below.

▶ **Example 5.** Let $(\mathcal{F}, \mathcal{R})_{\mathsf{ab}}$ be the TRS with only a rule $\mathsf{a}(\mathsf{b}(x)) \to \mathsf{b}(\mathsf{a}(x))$ and signature $\mathsf{a}, \mathsf{b} : [\mathsf{string}] \Rightarrow \mathsf{string}$ and $\epsilon : \mathsf{string}$. We can prove termination by the following interpretation:

$$[\![\mathsf{a}(x)]\!] = 2 * x \qquad [\![\mathsf{b}(x)]\!] = x + 1 \qquad [\![\epsilon]\!] = 0$$

Indeed, we have $[\![\ell]\!] > [\![r]\!]$ for the only rule as $[\![\mathsf{a}(\mathsf{b}(x))]\!] = 2 * x + 2 > 2 * x + 1 = [\![\mathsf{b}(\mathsf{a}(x))]\!]$. Now consider a term $t = \mathsf{a}^n(\mathsf{b}^m(\epsilon))$. Then $\mathtt{dh}_\mathcal{R}(t) = n * m$ whereas $[\![t]\!] = 2^n m$; an exponential difference! Such an overestimation is problematic if we want to use $[\![\cdot]\!]$ to bound $\mathtt{dh}_\mathcal{R}(\cdot)$.

We could find a tight bound for the system of Example 5 by a reasoning like the following: for every term s, let $\#bs(s)$ be the number of b occurrences in s. For a term t, let $cost(t)$ denote $\sum\{\{\#bs(s) \mid \mathsf{a}(s) \text{ is a subterm of } t\}\}$. Then, the cost of a term decreases exactly by 1 in each step. As the normal form has cost 0, we find the tight bound $cost(\mathsf{a}^n(\mathsf{b}^m(\epsilon))) = n * m$.

This reasoning relies on tracking more than one value. We can formalise this reasoning using an algebra interpretation (and will do so in Example 8), by choosing the right $\mathcal{A}$:

▶ **Definition 6.** *A* tuple algebra *is an algebra $\mathcal{A} = (A, \mathcal{J})$ with $A = (A_\iota, >_\iota, \geq_\iota)_{\iota \in \mathcal{S}}$ such that each A_ι has the form $\mathbb{N}^{K[\iota]}$ (for an integer $K[\iota] \geq 1$) and we let $\langle n_1, \dots, n_{K[\iota]} \rangle \geq_\iota \langle n'_1, \dots, n'_{K[\iota]} \rangle$ if each $n_i \geq n'_i$, and $\langle n_1, \dots, n_{K[\iota]} \rangle >_\iota \langle n'_1, \dots, n'_{K[\iota]} \rangle$ if additionally $n_1 > n'_1$.*

Intuitively, the first component always indicates "cost": the number of steps needed to reduce a term to normal form. This is the component that needs to decrease in each rewrite step to have $[\![s]\!] > [\![t]\!]$ whenever $s \to_\mathcal{R} t$. The remaining components represent some value of interest for the sort. This could for example be the size of the term (or its normal form), the length of a list, or following Example 5, the number of occurrences of a specific symbol. For these components, we only require that they do not increase in a reduction step.

By the definition of $>_\iota$, and using Theorem 4, we can conclude:

▶ **Corollary 7.** *If a TRS $(\mathcal{F}, \mathcal{R})$ is compatible with a tuple algebra then it is terminating and $\mathtt{dh}_{\mathcal{R}}(t) \leq [\![t]\!]_1$, for all terms t. (Here, $[\![t]\!]_1$ indicates the first component of the tuple $[\![t]\!]$.)*

Using this, we obtain a tight bound on the derivation height of $\mathsf{a}^n(\mathsf{b}^m(\epsilon))$ in Example 5:

▶ **Example 8.** The TRS $(\mathcal{F}, \mathcal{R})_{\mathsf{ab}}$ is compatible with the tuple algebra with $A_{\mathsf{string}} = \mathbb{N}^2$ and

$$[\![\mathsf{a}(x)]\!] = \langle x_1 + x_2, x_2 \rangle \qquad [\![\mathsf{b}(x)]\!] = \langle x_1, x_2 + 1 \rangle \qquad [\![\epsilon]\!] = \langle 0, 0 \rangle$$

Here, again, subscripts indicate tuple indexing; i.e., $\langle n, m \rangle_1 = n$ and $\langle n, m \rangle_2 = m$. Note that for every ground term s we have $[\![s]\!]_2 = \#bs(s)$. The first component exactly sums $\#bs(t)$ for every subterm t of s which has the form $\mathsf{a}(t')$. We have: $[\![\mathsf{a}(\mathsf{b}(x))]\!] = \langle x_1 + x_2 + 1, x_2 + 1 \rangle >_{\mathsf{nat}} \langle x_1 + x_2, x_2 + 1 \rangle = [\![\mathsf{b}(\mathsf{a}(x))]\!]$. The interpretation functions $\mathcal{J}_{\mathsf{a}}$ and $\mathcal{J}_{\mathsf{b}}$ are indeed monotonic. For example, for $\mathcal{J}_{\mathsf{a}}$: if $x >_{\mathsf{nat}} y$ then $x_1 + x_2 > y_1 + y_2$ (since $x_1 > y_1$ and $x_2 \geq y_2$) and $x_2 \geq y_2$; and if $x \geq_{\mathsf{nat}} y$ then $x_1 + x_2 \geq y_1 + y_2$ and $x_2 \geq y_2$. We have $[\![\mathsf{a}^n(\mathsf{b}^m(\epsilon))]\!] = (n * m, m)$.

To build strongly monotonic functions we can for instance use the following observation:

▶ **Lemma 9.** *A function $F : \mathbb{N}^{K[\iota_1]} \times \cdots \times \mathbb{N}^{K[\iota_k]} \Longrightarrow \mathbb{N}^{K[\kappa]}$ is strongly monotonic if we can write $F(x^1, \ldots, x^k) = \langle x_1^1 + \cdots + x_1^k + S_1(x^1, \ldots, x^k),\ S_2(x^1, \ldots, x^k), \ldots,\ S_{K[\kappa]}(x^1, \ldots, x^k) \rangle$, where each S_i is a weakly monotonic function in $\mathbb{N}^{K[\iota_1]} \times \cdots \times \mathbb{N}^{K[\iota_k]} \Longrightarrow \mathbb{N}$.*

Moreover, a function $S : \mathbb{N}^{K[\iota_1]} \times \cdots \times \mathbb{N}^{K[\iota_k]} \Longrightarrow \mathbb{N}$ is weakly monotonic if it is built from constants in $\mathbb{N}$, variable components x_j^n and weakly monotonic functions in $\mathbb{N}^n \Longrightarrow \mathbb{N}$.

For the "weakly monotonic functions in $\mathbb{N}^n \Longrightarrow \mathbb{N}$" we could for instance use $+$, $*$ or max.

To determine the length $K[\iota]$ of the tuple for a sort ι, we use a semantic approach, similar to one used in [19] in the context of functional languages: the elements of the tuple are values of interest for the sort. The two prominent examples in this paper are the sort nat of natural numbers – which is constructed from the symbols $\mathsf{0} :: \mathsf{nat}$ and $\mathsf{s} :: [\mathsf{nat}] \Rightarrow \mathsf{nat}$ – and the sort list of lists of natural numbers – which is constructed using $\mathsf{nil} :: \mathsf{list}$ and $\mathsf{cons} :: [\mathsf{nat} \times \mathsf{list}] \Rightarrow \mathsf{list}$. For natural numbers, we consider their size, so the number of ss. For lists, we consider both their length and an upper bound on the size of their elements. This gives $K[\mathsf{nat}] = 2$ (cost of reducing the term, size of its normal form) and $K[\mathsf{list}] = 3$ (cost of reducing, length of normal form, maximum element size). In the remainder of this paper, we will use x_{c} as syntactic sugar for x_1 (the cost component of x), x_{s} and x_{l} as x_2 and x_{m} as x_3.

▶ **Example 10.** Consider the TRS defined in Example 1. We start by giving an interpretation for the type constructors: the symbols $\mathsf{0}, \mathsf{nil}, \mathsf{s}$ and cons which are used to construct natural numbers and lists. To be in line with the semantics for the type interpretation, we let:

$$[\![\mathsf{0}]\!] = \langle 0, 0 \rangle \qquad [\![\mathsf{s}(x)]\!] = \langle x_{\mathsf{c}}, x_{\mathsf{s}} + 1 \rangle$$
$$[\![\mathsf{nil}]\!] = \langle 0, 0, 0 \rangle \qquad [\![x : xs]\!] = \langle x_{\mathsf{c}} + xs_{\mathsf{c}}, xs_{\mathsf{l}} + 1, \max(x_{\mathsf{s}}, xs_{\mathsf{m}}) \rangle$$

This expresses that $\mathsf{0}$ has no evaluation cost and size 0; analogously, nil has no evaluation cost and 0 as length and maximum element. The cost of evaluating a term $\mathsf{s}(t)$ depends entirely on the cost of the term's argument t; the size component counts the number of ss. The cost component for cons similarly sums the costs of its arguments, while the length is increased by 1, and the maximum element is the maximum between its head and tail.

For the remaining symbols we choose the following interpretations:

$$\begin{aligned}
[\![x \oplus y]\!] &= \langle x_{\mathsf{c}} + y_{\mathsf{c}} + y_{\mathsf{s}} + 1, x_{\mathsf{s}} + y_{\mathsf{s}} \rangle \\
[\![\mathsf{sum}(xs)]\!] &= \langle xs_{\mathsf{c}} + 2 * xs_{\mathsf{l}} + xs_{\mathsf{l}} * xs_{\mathsf{m}} + 1, xs_{\mathsf{l}} * xs_{\mathsf{m}} \rangle \\
[\![\mathsf{rev}(xs)]\!] &= \langle xs_{\mathsf{c}} + xs_{\mathsf{l}} + \tfrac{xs_{\mathsf{l}} * (xs_{\mathsf{l}} + 1)}{2} + 1, xs_{\mathsf{l}}, xs_{\mathsf{m}} \rangle \\
[\![\mathsf{append}(xs, ys)]\!] &= \langle xs_{\mathsf{c}} + ys_{\mathsf{c}} + xs_{\mathsf{l}} + 1, xs_{\mathsf{l}} + ys_{\mathsf{l}}, \max(xs_{\mathsf{m}}, ys_{\mathsf{m}}) \rangle
\end{aligned}$$

Checking compatibility is easily done for the interpretation above, and strong monotonicity follows by Lemma 9 (as $n \mapsto \frac{n*(n+1)}{2} \in \mathbb{N} \Longrightarrow \mathbb{N}$ is weakly monotonic). We see that the cost of evaluating append is linear in the first list length and independent of the size of the list elements, while evaluating sum gives a quadratic dependency on length and size combined.

Our tuple interpretations have some similarities with matrix interpretations [21], where also each term is associated to an n-tuple. In essence, matrix interpretations *are* tuple interpretations, for systems with only one sort. However, the shape of the interpretation functions $\mathcal{J}_{\mathsf{f}}$ in matrix interpretations is limited to functions following Lemma 9 where each S is a linear multivariate polynomial. Hence, our interpretations are a strict generalisation, which also admits interpretations such as those used for sum, rev and append in Example 10.

For the purpose of termination, tuple interpretations strictly extend the power of both polynomial interpretations and matrix interpretations already in the first-order case.

▶ **Example 11.** A TRS that implements division in [4] shows a limitation of polynomial interpretations: it contains a rule $\mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y)))$ which cannot be oriented by any polynomial interpretation, because $[\![\mathsf{minus}(x, \mathsf{s}(x))]\!] > [\![\mathsf{s}(x)]\!]$ for any strongly monotonic polynomial $\mathcal{J}_{\mathsf{minus}}$. Due to the duplication of y, this rule also cannot be handled by a matrix interpretation. However, we do have a compatible tuple interpretation:

$$\begin{array}{rclrcl} [\![\mathsf{0}]\!] & = & \langle 0, 0\rangle & [\![\mathsf{minus}(x, y)]\!] & = & \langle x_{\mathsf{c}} + y_{\mathsf{c}} + y_{\mathsf{s}} + 1, x_{\mathsf{s}}\rangle \\ [\![\mathsf{s}(x)]\!] & = & \langle x_{\mathsf{c}}, x_{\mathsf{s}} + 1\rangle & [\![\mathsf{quot}(x, y)]\!] & = & \langle x_{\mathsf{c}} + x_{\mathsf{s}} + y_{\mathsf{c}} + x_{\mathsf{s}} * y_{\mathsf{c}} + x_{\mathsf{s}} * y_{\mathsf{s}} + 1, x_{\mathsf{s}}\rangle \end{array}$$

In practice, in first-order termination or complexity analysis one would not exclusively use interpretations, but rather a combination of different techniques. In that context, tuple interpretations may be used as one part of a large toolbox. They are likely to offer a simple complexity proof in many cases, but they are unlikely to be an essential technique since so many other methods have already been developed. Indeed, all examples in this section can be handled with previously established theory. For instance, Example 5 can be handled with matrix interpretations, while sum and rev may be analysed using ideas from [24] and [35].

However, developing a new technique for first-order termination and traditional complexity analysis is not our goal. Our method *does* provide a more fine-grained notion of complexity, which may consider information such as the length of a list. Moreover, the first-order case is an important stepping stone towards higher-order analysis, where far fewer methods exist.

4 Higher-order tuple interpretations

In this section, we will extend the ideas from Section 3 to the higher-order setting, and hence define the core notion of this paper: higher-order tuple interpretations. To do this, we will build on the notion of *strongly monotonic algebras* originating in [39].

4.1 Strongly monotonic algebras

In first-order term rewriting, the complexity of a TRS is often measured as *runtime* or *derivational* complexity. Both measures consider initial terms s of a certain shape, and supply a bound on $\mathtt{dh}_{\mathcal{R}}(s)$ given the size of s. However, this is not a good approach for higher-order terms: the behaviour of a term of higher type generally cannot be captured in an integer.

▶ **Example 12.** Consider the AFS obtained by combining Examples 1 and 2. The evaluation cost of a term $\mathsf{foldl}(F, n, q)$ depends almost completely on the behaviour of the functional subterm F, and not only on its evaluation cost. To see this, consider two cases: $F_1 :=$

$\lambda x.\lambda y.y \oplus x$, and $F_2 := \lambda x.\lambda y.x \oplus x$. For natural numbers n, m, the evaluation cost of both $F_1(n, m)$ and $F_2(n, m)$ is the same: $n + 1$. However, the *size* of the result is different. Hence, the number of steps needed to compute $\mathsf{foldl}(F_1, n, q)$ for a number n and list q is quadratic in the size of n and q, while the number of steps needed for $\mathsf{foldl}(F_2, n, q)$ is exponential.

As Example 12 shows, higher-order rewriting is a natural place to separate cost and size. But more than that, we need to know what a function does with its arguments: whether it is size-increasing, how long it takes to evaluate them, and more.

This is naturally captured by the notion of (weakly or strongly) monotonic algebras for higher-order rewriting introduced by v.d. Pol [39]: here, a term of arrow type is interpreted as a function, which allows the interpretation to retain all relevant information.

Monotonic interpretations were originally defined for a different higher-order rewriting formalism, which does make some difference in the way abstraction and application is handled. *Weakly* monotonic algebras were transposed to AFSs in [22]; however, here we extend the more natural notion of *hereditarily* monotonic algebras which v.d. Pol only briefly considered.[1]

▶ **Definition 13.** *Let $\mathcal{S}$ be a set of sorts and $\mathcal{F}$ a higher-order signature. We assume given for every sort ι an extended well-founded set $(A_\iota, >_\iota, \geq_\iota)$. From this, we define the set of* strongly monotonic functionals*, as follows:*

- *For all sorts ι: $\mathcal{M}_\iota := A_\iota$ and $\sqsupset_\iota := >_\iota$ and $\sqsupseteq_\iota := \geq_\iota$.*
- *For an arrow type $\sigma \Rightarrow \tau$:*
  - *$\mathcal{M}_{\sigma \Rightarrow \tau} := \{F \in \mathcal{M}_\sigma \Longrightarrow \mathcal{M}_\tau \mid F$ is strongly monotoic$\}$*
  - *$F \sqsupset_{\sigma \Rightarrow \tau} G$ iff $\mathcal{M}_\sigma$ is non-empty and $\forall x \in \mathcal{M}_\sigma . F(x) \sqsupset_\tau G(x)$, and*
    $F \sqsupseteq_{\sigma \Rightarrow \tau} G$ iff $\forall x \in \mathcal{M}_\sigma . F(x) \sqsupseteq_\tau G(x)$.

That is, $\mathcal{M}_{\sigma \Rightarrow \tau}$ contains strongly monotonic functions from $\mathcal{M}_\sigma$ to $\mathcal{M}_\tau$ and both $\sqsupset_{\sigma \Rightarrow \tau}$ and $\sqsupseteq_{\sigma \Rightarrow \tau}$ do a point-wise comparison. By a straightforward induction on types we have:

▶ **Lemma 14.** *For all types σ, $(\mathcal{M}_\sigma, \sqsupset_\sigma, \sqsupseteq_\sigma)$ is an extended well-founded set; that is:*

- *$\sqsupset_\sigma$ is well-founded and $\sqsupseteq_\sigma$ is reflexive;*
- *both $\sqsupset_\sigma$ and $\sqsupseteq_\sigma$ are transitive;*
- *for all $x, y, z \in \mathcal{M}_\sigma$, $x \sqsupset_\sigma y$ implies $x \sqsupseteq_\sigma y$ and $x \sqsupset_\sigma y \sqsupseteq_\sigma z$ implies $x \sqsupset_\sigma z$.*

We will define *higher-order strongly monotonic algebras* as an extension of Definition 3, mapping a term of type σ to an element of $\mathcal{M}_\sigma$. Functional terms $\mathsf{f}(s_1, \ldots, s_k)$ and variables can be handled as before, but we now also have to deal with application and abstraction. Application is straightforward: since terms of higher type are mapped to functions, we can interpret application as function application, i.e., $[\![s \cdot t]\!]_\alpha := [\![s]\!]_\alpha([\![t]\!]_\alpha)$. However, abstraction is more difficult. The natural choice would be to view abstraction as defining a function; i.e., let $[\![\lambda x.s]\!]_\alpha$ be the function $d \mapsto [\![s]\!]_{\alpha[x:=d]}$. Unfortunately, this is not necessarily monotonic: $d \mapsto [\![s]\!]_{\alpha[x:=d]}$ is strongly monotonic only if x occurs freely in s. For example $\lambda x.\mathsf{0}$ would be mapped to a constant function, which is not in $\mathcal{M}_{\mathsf{nat} \Rightarrow \mathsf{nat}}$. Moreover, this definition would give $[\![(\lambda x.s) \cdot t]\!]_\alpha = [\![s[x := t]]\!]_\alpha$, so β-steps would not be counted toward the evaluation cost.

We handle both problems by using a choosable function $\mathit{MakeSM}_{\sigma,\tau}$, which takes a function that may be strongly monotonic or constant, and turns it strongly monotonic.

[1] In [39], v.d. Pol rejects hereditarily (or: strongly) monotonic algebras because they are not so well-suited for analysing the HRS format [36] where reasoning is modulo $\rightarrow_\beta$: it is impossible to both interpret all terms of function type to strongly monotonic functions and have $[\![(\lambda x.s)\ t]\!] = [\![s[x := t]]\!]$. In the AFS format, we do not have the latter requirement. In [22], where the authors considered the AFS format like we do here (but for interpretations to $\mathbb{N}$ rather than to tuples), weakly monotonic algebras were used because they are a more natural choice in the context of dependency pairs.

▶ **Definition 15.** *A (σ,τ)-monotonicity function $\mathit{MakeSM}_{\sigma,\tau}$ is a strongly monotonic function in $C_{\sigma,\tau} \Longrightarrow \mathcal{M}_{\sigma\Longrightarrow\tau}$, where the set $C_{\sigma,\tau}$ is defined as $\mathcal{M}_{\sigma\Rightarrow\tau} \cup \{F \in \mathcal{M}_\sigma \Longrightarrow \mathcal{M}_\tau \mid F(x) = F(y)$ for all $x, y \in \mathcal{M}_\sigma\}$. (Here, the set $C_{\sigma,\tau}$ is ordered by point-wise comparison.)*

With this definition, we are ready to define strongly monotonic algebras.

▶ **Definition 16.** *A strongly monotonic algebra $\mathcal{A}_\mathcal{M}$ consists of a family $(\mathcal{M}_\sigma, \sqsupset_\sigma, \sqsupseteq_\sigma)_{\sigma\in\mathcal{ST}}$, an interpretation function $\mathcal{J}$ which associates to each $\mathsf{f} :: [\sigma_1 \times \cdots \times \sigma_k] \Rightarrow \tau$ in $\mathcal{F}$ an element of $\mathcal{M}_{\sigma_1\Rightarrow\ldots\Rightarrow\sigma_k\Rightarrow\tau}$, and a (σ,τ)-monotonicity function $\mathit{MakeSM}_{\sigma,\tau}$, for each $\sigma, \tau \in \mathcal{ST}$.*

Let α be a function that maps variables of type σ to elements of $\mathcal{M}_\sigma$. We extend $\mathcal{J}$ to a function $[\![\cdot]\!]_\alpha$ that maps terms of type σ to elements of $\mathcal{M}_\sigma$, as follows:

$$
\begin{array}{ll}
[\![x]\!]_\alpha = \alpha(x) \text{ for variables } x & [\![\mathsf{f}(s_1,\ldots,s_k)]\!]_\alpha = \mathcal{J}_\mathsf{f}([\![s_1]\!]_\alpha,\ldots,[\![s_k]\!]_\alpha) \\
[\![s \cdot t]\!]_\alpha = [\![s]\!]_\alpha([\![t]\!]_\alpha) & [\![\lambda x.s]\!]_\alpha = \mathit{MakeSM}_{\sigma,\tau}(d \mapsto [\![s]\!]_{\alpha[x:=d]}) \text{ if } x :: \sigma \text{ and } s :: \tau
\end{array}
$$

We can see by induction on s that for $s :: \sigma$ indeed $[\![s]\!]_\alpha \in \mathcal{M}_\sigma$. We say that an AFS $(\mathcal{F},\mathcal{R})$ is *compatible* with $\mathcal{A}_\mathcal{M}$ if for all valuations α both (1) $[\![\ell]\!]_\alpha \sqsupset [\![r]\!]_\alpha$, for all $\ell \to r \in \mathcal{R}$; and (2) $[\![(\lambda x.s)\,t]\!]_\alpha \sqsupset [\![s[x := t]]\!]_\alpha$, for any $s :: \sigma$, $t :: \tau$ and $x \in \mathcal{X}_\tau$.

As before, we will typically omit the α subscript and use notation like $[\![s]\!] = F(x+3)$ to denote $[\![s]\!]_\alpha = \alpha(F)(\alpha(x)+3)$. When types are not relevant, we will denote $\sqsupset$ instead of specifying $\sqsupset_\sigma$, and we may write $f \in \mathcal{M}$ to mean $f \in \mathcal{M}_\sigma$ for some $\sigma \in \mathcal{ST}$.

We extend Theorem 4 into the following compatibility result.

▶ **Theorem 17.** *If $(\mathcal{F},\mathcal{R})$ is compatible with $\mathcal{A}_\mathcal{M}$, then for all α, $[\![s]\!]_\alpha \sqsupset [\![t]\!]_\alpha$ when $s \to_\mathcal{R} t$.*

For Definition 13 and Theorem 17, we can choose the well-founded sets $(A_\iota, >_\iota, \geq_\iota)$ for each sort, and the functions $\mathit{MakeSM}_{\sigma,\tau}$ for each pair of types, as we desire. A *higher-order tuple algebra* is a strongly monotonic algebra where each $(A_\iota, >_\iota, \geq_\iota)$ follows Definition 6.

▶ **Example 18.** Let $A_\mathsf{nat} = \mathbb{N}^2$ and $A_\mathsf{list} = \mathbb{N}^3$ as before, and assume cons and nil are interpreted as in Example 10. Consider the rules for map in Example 2. We let:

$$[\![\mathsf{map}(F, xs)]\!] = \langle (xs_\mathsf{l}+1) * (F(\langle xs_\mathsf{c}, xs_\mathsf{m}\rangle)_\mathsf{c}+1),\ xs_\mathsf{l},\ F(xs_\mathsf{c}, xs_\mathsf{m})_\mathsf{s}\rangle$$

This expresses that map does not increase the list length (as the length component is just xs_l), the greatest element of the result is bounded by the value of F on the greatest element of xs, and the evaluation cost is mostly expressed by a number of F steps that is linear in the length of xs. We will see in Lemma 23 that $\mathcal{J}_\mathsf{map}$ is indeed strongly monotonic.

To prove compatibility of the AFS with $\mathcal{A}_\mathcal{M}$, we must first see that $[\![\ell]\!] \sqsupset [\![r]\!]$ for all rules $\ell \to_\mathcal{R} r$. For the first map rule this is easy: $[\![\mathsf{map}(F,\mathsf{nil})]\!] = \langle F(\langle 0,0\rangle)_\mathsf{c}+1, 0, F(\langle 0,0\rangle)_\mathsf{s}\rangle \sqsupset_\mathsf{list} \langle 0,0,0\rangle = [\![\mathsf{nil}]\!]$. For the second map rule, we must check that $\langle \text{cost-}\ell, \text{len-}\ell, \text{max-}\ell\rangle \sqsupset_\mathsf{list} \langle \text{cost-}r, \text{len-}r, \text{max-}r\rangle$; that is, cost-$\ell >$ cost-r and len-$\ell \geq$ len-r and max-$\ell \geq$ max-r, where:

$$
\begin{array}{lclcl}
\text{cost-}\ell & = & [\![\mathsf{map}(F, x : xs)]\!]_\mathsf{c} & = & (xs_\mathsf{l}+2) * (F(\langle x_\mathsf{c}+xs_\mathsf{c}, \max(x_\mathsf{s}, xs_\mathsf{m})\rangle)_\mathsf{c}+1) \\
\text{cost-}r & = & [\![F(x) : \mathsf{map}(F, xs)]\!]_\mathsf{c} & = & F(\langle x_\mathsf{c}, x_\mathsf{s}\rangle)_\mathsf{c} + (xs_\mathsf{l}+1) * (F(\langle xs_\mathsf{c}, xs_\mathsf{m}\rangle)_\mathsf{c}+1) \\
\text{len-}\ell & = & [\![\mathsf{map}(F, x : xs)]\!]_\mathsf{l} & = & xs_\mathsf{l}+1 \quad = \quad [\![F(x) : \mathsf{map}(F, xs)]\!]_\mathsf{l} \quad = \quad \text{len-}r \\
\text{max-}\ell & = & [\![\mathsf{map}(F, x : xs)]\!]_\mathsf{m} & = & F(\langle x_\mathsf{c}+xs_\mathsf{c}, \max(x_\mathsf{s}, xs_\mathsf{m})\rangle)_\mathsf{s} \\
\text{max-}r & = & [\![F(x) : \mathsf{map}(F, xs)]\!]_\mathsf{m} & = & \max(F(\langle x_\mathsf{c}, x_\mathsf{s}\rangle)_\mathsf{s}, F(\langle xs_\mathsf{c}, xs_\mathsf{m}\rangle)_\mathsf{s})
\end{array}
$$

To see why cost-$\ell >$ cost-r, we observe that for all x, xs: $\langle x_\mathsf{c}+xs_\mathsf{c}, \max(x_\mathsf{s}+xs_\mathsf{m})\rangle \sqsupseteq_\mathsf{nat}$ both $\langle x_\mathsf{c}, x_\mathsf{s}\rangle$ and $\langle xs_\mathsf{c}, xs_\mathsf{m}\rangle$. Since $F \in \mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}}$ therefore $F(\langle x_\mathsf{c}+xs_\mathsf{c}, \max(x_\mathsf{s}+xs_\mathsf{m})\rangle) \sqsupseteq_\mathsf{nat}$ both $F(\langle x_\mathsf{c}, x_\mathsf{s}\rangle)$ and $F(\langle xs_\mathsf{c}, xs_\mathsf{m}\rangle)$. We find max-$\ell \geq$ max-r by a similar reasoning.

4.2 Interpreting abstractions

Example 18 is not complete: we have not yet defined the functions $\mathit{MakeSM}_{\sigma,\tau}$, and we have not shown that $[\![(\lambda x.s)\ t]\!] \sqsupset [\![s[x := t]]\!]$ always holds. To achieve this, we will define some standard functions to build elements of $\mathcal{M}$. This allows us to easily construct strongly monotonic functionals, both to build $\mathit{MakeSM}_{\sigma,\tau}$ and to create interpretation functions $\mathcal{J}_{\mathsf{f}}$.

▶ **Definition 19.** *For every type σ, we define: $\mathtt{0}_\sigma \in \mathcal{M}_\sigma$; $\mathtt{costof}_\sigma \in \mathcal{M}_\sigma \Longrightarrow \mathbb{N}$; and $\mathtt{addc}_\sigma \in \mathbb{N} \times \mathcal{M}_\sigma \Longrightarrow \mathcal{M}_\sigma$ by mutual recursion on σ as follows.*

$$\begin{array}{ll} \mathtt{0}_\iota = \langle 0, \dots, 0\rangle & \mathtt{0}_{\sigma\Rightarrow\tau} = d \mapsto \mathtt{addc}_\tau(\mathtt{costof}_\sigma(d), \mathtt{0}_\tau) \\ \mathtt{costof}_\iota(\langle n_1, \dots, n_{K[\iota]}\rangle) = n_1 & \mathtt{costof}_{\sigma\Rightarrow\tau}(F) = \mathtt{costof}_\tau(F(\mathtt{0}_\sigma)) \\ \mathtt{addc}_\iota(c, \langle n_1, \dots, n_{K[\iota]}\rangle) = \langle c + n_1, n_2, \dots, n_{K[\iota]}\rangle & \mathtt{addc}_{\sigma\Rightarrow\tau}(c, F) = d \mapsto \mathtt{addc}_\tau(c, F(d)) \end{array}$$

Here, $\mathtt{0}_\sigma$ defines the *minimal element* of $\mathcal{M}_\sigma$. The function $\mathtt{costof}_\sigma$ maps every F to the cost component of $F(\mathtt{0}_{\sigma_1}, \dots, \mathtt{0}_{\sigma_m})$; hence, if $F \sqsupset_\sigma G$ we have $\mathtt{costof}_\sigma(F) > \mathtt{costof}_\sigma(G)$. The function $\mathtt{addc}_\sigma$ pointwise increases an element of $\mathcal{M}_\sigma$ by adding to the cost component: if $F(x_1, \dots, x_m) = \langle n_1, \dots, n_k\rangle$, then $\mathtt{addc}(c, F)(x_1, \dots, x_m) = \langle c + n_1, n_2, \dots, n_k\rangle$.

It is easy to see that $\mathtt{0}_\sigma$ and $\mathtt{addc}_\sigma(n, X)$ are in $\mathcal{M}$ for all σ (by induction on σ), and that $\mathtt{costof}_\sigma$ and $\mathtt{addc}_\sigma$ are strict in all their arguments. Various properties of these functions are detailed in the appendix (Lemmas B.4–B.8). We will particularly use that always $F(\mathtt{addc}(n, x)) \sqsupseteq \mathtt{addc}(n, F(x))$ (Lemma B.7) and $\mathtt{costof}(F(x)) \geq \mathtt{costof}(x)$ (Lemma B.8).

We can use these functions to for instance create candidates for $\mathit{MakeSM}_{\sigma,\tau}$. While many suitable definitions are possible, we will particularly consider the following:

▶ **Definition 20.** *For types σ, τ, and F a weakly monotonic function in $\mathcal{M}_\sigma \Longrightarrow \mathcal{M}_\tau$, let:*

$$\Phi_{\sigma,\tau}(F) = \begin{cases} d \mapsto \mathtt{addc}_{\sigma\Rightarrow\tau}(1, F(d)) & \text{if } F \text{ is in } \mathcal{M}_{\sigma\Rightarrow\tau} \\ d \mapsto \mathtt{addc}_{\sigma\Rightarrow\tau}(\mathtt{costof}_\sigma(d) + 1, F(d)) & \text{otherwise} \end{cases}$$

Then $\Phi_{\sigma,\tau}$ is a (σ,τ)-monotonicity function. To see this, the most challenging part is proving that $\Phi_{\sigma,\tau}(F) \sqsupset \Phi_{\sigma,\tau}(G)$ if $F \sqsupset G$ and $F \in \mathcal{M}_{\sigma\Rightarrow\tau}$ while G is a constant function. We can prove this using the result that $x \sqsupset y$ implies $\mathtt{addc}(1, x) \sqsupseteq y$ for all x, y. We have:

▶ **Lemma 21.** *If $\mathit{MakeSM}_{\sigma,\tau} = \Phi_{\sigma,\tau}$ then $[\![(\lambda x.s)\, t]\!] \sqsupset_\tau [\![s[x := t]]\!]$, for $s :: \tau$, $t :: \sigma$, $x \in \mathcal{X}_\sigma$.*

Proof Sketch. We expand $\mathit{MakeSM}_{\sigma,\tau}$ to achieve $[\![(\lambda x.s)\, t]\!]_\alpha = \mathtt{addc}_\tau(\mathtt{costof}_\sigma([\![t]\!]_\alpha) + 1, [\![s]\!]_{\alpha[x:=[\![t]\!]]})$ or $[\![(\lambda x.s)\, t]\!]_\alpha = \mathtt{addc}_\tau(1, [\![s]\!]_{\alpha[x:=[\![t]\!]]})$. By induction on τ we prove that $\mathtt{addc}_\tau(n, F) \sqsupset_\tau F$ for all $n \geq 1$. So either way, $[\![(\lambda x.s)\, t]\!]_\alpha \sqsupset_\tau [\![s]\!]_{\alpha[x:=[\![t]\!]]}$. Finally, we prove a substitution lemma, $[\![s]\!]_{\alpha[x:=[\![t]\!]_\alpha]} = [\![s[x := t]]\!]_\alpha$, by induction on s. ◀

In examples in the remainder of this paper, we will assume that $\mathit{MakeSM}_{\sigma,\tau} = \Phi_{\sigma,\tau}$. With these choices we do not only orient the β-rule (and thus satisfy item (2) of the compatibility conditions), but also the η-reduction rules mentioned in Section 2.2.

▶ **Lemma 22.** *If $\mathit{MakeSM}_{\sigma,\tau} = \Phi_{\sigma,\tau}$ then for any $F \in \mathcal{X}_{\sigma\Rightarrow\tau}$ we have: $[\![\lambda x.F\, x]\!] \sqsupset_{\sigma\Rightarrow\tau} [\![F]\!]$.*

Proof Sketch. Since $F \neq x$, we have $[\![F]\!]_{\alpha[x:=d]} = \alpha(F)$ for all α and d. Consequently, $[\![\lambda x.F\, x]\!] \sqsupseteq_{\sigma\Rightarrow\tau} d \mapsto \mathtt{addc}_\tau(1, F(d))$ either way. We are done as: $\mathtt{addc}_\tau(1, F(d)) \sqsupset_\tau F(d)$. ◀

4.3 Creating strongly monotonic interpretation functions

We can use Theorem 17 to obtain bounds on the derivation heights of given terms. However, to achieve this, we must find an interpretation function $\mathcal{J}$, and prove that each $\mathcal{J}_{\mathsf{f}}$ is in $\mathcal{M}$. We will now explore ways to construct such strongly monotonic functions. It turns out to be useful to also consider *weakly* monotonic functions. In the following, we will write "f is $\mathrm{wm}(A_1,\dots,A_k;B)$" to mean that f is a weakly monotonic function in $A_1 \times \cdots \times A_k \Longrightarrow B$.

▶ **Lemma 23.** *Let $x^1,\dots,x^k$ be variables ranging over $\mathcal{M}_{\sigma_1},\dots,\mathcal{M}_{\sigma_k}$ respectively; we shortly denote this sequence $\vec{x}$. We let $\overrightarrow{\mathcal{M}_\sigma}$ denote the sequence $\mathcal{M}_{\sigma_1},\dots,\mathcal{M}_{\sigma_k}$. Then:*

1. *if $F(\vec{x}) = x^i$ then F is wm$(\overrightarrow{\mathcal{M}_\sigma};\mathcal{M}_{\sigma_i})$, and F is strict in argument i;*
2. *if $F(\vec{x}) = x^i(F_1(\vec{x}),\dots,F_n(\vec{x}))$, $\sigma_i = \tau_1 \Rightarrow \dots \Rightarrow \tau_n \Rightarrow \rho$, and each F_j is wm$(\overrightarrow{\mathcal{M}_\sigma};\mathcal{M}_{\tau_j})$ then F is wm$(\overrightarrow{\mathcal{M}_\sigma};\mathcal{M}_\rho)$ and for all $p \in \{1,\dots,k\}$: F is strict in argument p if $p = i$ or some F_j is strict in argument p;*
3. *if $F(\vec{x}) = \langle G_1(\vec{x}),\dots,G_{K[\iota]}(\vec{x})\rangle$ and each G_j is wm$(\overrightarrow{\mathcal{M}_\sigma};\mathbb{N})$ then F is wm$(\overrightarrow{\mathcal{M}_\sigma};\mathcal{M}_\iota)$, and for all $p \in \{1,\dots,k\}$: F is strict in argument p if G_1 is.*

The last result uses functions mapping to $\mathbb{N}$; these can be constructed using the observations:

4. *if $G(\vec{x}) = n$ for some $n \in \mathbb{N}$ then G is wm$(\overrightarrow{\mathcal{M}_\sigma};\mathbb{N})$;*
5. *if $G(\vec{x}) = x^i_j$ and $\sigma_i = \iota \in \mathcal{S}$ and $1 \le j \le K[\iota]$, then G is wm$(\overrightarrow{\mathcal{M}_\sigma};\mathbb{N})$, and G is strict in argument i if $j = 1$;*
6. *if $G(\vec{x}) = f(G_1(\vec{x}),\dots,G_n(\vec{x}))$ and all G_j are wm$(\overrightarrow{\mathcal{M}_\sigma};\mathbb{N})$ and f is wm$(\mathbb{N},\dots,\mathbb{N};\mathbb{N})$, then G is wm$(\overrightarrow{\mathcal{M}_\sigma};\mathbb{N})$, and for all $p \in \{1,\dots,k\}$: G is strict in argument p if, for some $j \in \{1,\dots,n\}$: G_j is strict in argument p and f is strict in argument j;*
7. *if $G(\vec{x}) = F(\vec{x})_j$ and F is wm$(\overrightarrow{\mathcal{M}_\sigma};\mathcal{M}_\iota)$ and $1 \le j \le K[\iota]$ then G is wm$(\overrightarrow{\mathcal{M}_\sigma};\mathbb{N})$ and if $j = 1$ then for all $p \in \{1,\dots,k\}$: G is strict in argument p if F is.*

Proof Sketch. We easily see that in each case, F or G is in the given function space. To show weak monotonicity, assume given both $\vec{x}$ and $\vec{y}$ such that each $x^i \sqsupseteq y^i$; we then check for all cases that $F(\vec{x}) \sqsupseteq F(\vec{y})$, or $G(\vec{x}) \ge G(\vec{y})$. For the strictness conditions, we assume that $x^p \sqsupset y^p$ and similarly check all cases. ◀

The reader may recognise items (4–6): these largely correspond to the sufficient conditions for a weakly monotonic function S in Lemma 9. For the function f in item (6), we could for instance choose $+$, $*$ or max, where $+$ is strict in all arguments. However, we can get beyond Lemma 9 by using the other items; for example, applying variables to each other.

Now, if a function f is wm$(\overrightarrow{\mathcal{M}_\sigma};\mathcal{M}_\tau)$ and f is strict in all its arguments, then we easily see that the function $d_1 \mapsto \cdots \mapsto d_k \mapsto f(d_1,\dots,d_k)$ is an element of $\mathcal{M}_{\sigma_1 \Rightarrow \dots \Rightarrow \sigma_k \Rightarrow \tau}$. To illustrate how this can be used in practice, we show monotonicity of $\mathcal{J}_{\mathsf{map}}$ of Example 18:

▶ **Example 24.** Suppose $\mathcal{J}_{\mathsf{map}}(F,q) = (\ F(\langle q_{\mathsf{c}},q_{\mathsf{m}}\rangle)_{\mathsf{c}} + q_{\mathsf{l}} * F(\langle q_{\mathsf{c}},q_{\mathsf{m}}\rangle)_{\mathsf{c}} + q_{\mathsf{l}} + 1\ ,\ q_{\mathsf{l}}\ ,\ F(\langle q_{\mathsf{c}},q_{\mathsf{m}}\rangle)_{\mathsf{l}}\ )$. By (5), the functions $(F,q) \mapsto q_i$ are wm$(\mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}},\mathcal{M}_{\mathsf{list}};\mathbb{N})$ for $i \in \{\mathsf{c},\mathsf{l},\mathsf{m}\}$ and moreover, $(F,q) \mapsto q_{\mathsf{c}}$ is strict in argument 2. Hence, by (3), $(F,q) \mapsto \langle q_{\mathsf{c}},q_{\mathsf{m}}\rangle$ is wm$(\mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}},\mathcal{M}_{\mathsf{list}};\mathcal{M}_{\mathsf{nat}})$ and strict in argument 2. Therefore, by (2), $(F,q) \mapsto F(\langle q_{\mathsf{c}},q_{\mathsf{m}}\rangle)$ is wm$(\mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}},\mathcal{M}_{\mathsf{list}};\mathcal{M}_{\mathsf{nat}})$ and strict in both arguments. Hence, by (7), $(F,q) \mapsto F(\langle q_{\mathsf{c}},q_{\mathsf{m}}\rangle)_{\mathsf{c}}$ and $(F,q) \mapsto F(\langle q_{\mathsf{c}},q_{\mathsf{m}}\rangle)_{\mathsf{l}}$ are wm$(\mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}},\mathcal{M}_{\mathsf{list}};\mathbb{N})$ and the former is strict in both arguments.

Continuing like this, it is not hard to see how we can iteratively prove that $(F,q) \mapsto (\ F(\langle q_{\mathsf{c}},q_{\mathsf{m}}\rangle)_{\mathsf{c}} + q_{\mathsf{l}} * F(\langle q_{\mathsf{c}},q_{\mathsf{m}}\rangle)_{\mathsf{c}} + q_{\mathsf{l}} + 1\ ,\ q_{\mathsf{l}}\ ,\ F(\langle q_{\mathsf{c}},q_{\mathsf{m}}\rangle)_{\mathsf{l}}\ )$ is wm$(\mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}},\mathcal{M}_{\mathsf{list}};\mathcal{M}_{\mathsf{list}})$ and strict in both arguments, which immediately gives $\mathcal{J}_{\mathsf{map}} \in \mathcal{M}_{(\mathsf{nat}\Rightarrow\mathsf{nat})\Rightarrow\mathsf{list}\Rightarrow\mathsf{list}}$.

In practice, it is usually not needed to write such an elaborate proof: Lemma 23 essentially tells us that if a function is built exclusively using variables and variable applications, projections $F(\vec{x})_j$, constants, and weakly monotonic operators over the natural numbers, then that function is weakly monotonic; we only need to check that the cost component indeed increases if one of the variables x^i is increased.

Unfortunately, while Lemma 23 is useful for rules like the ones for map, it is not enough to handle functions like foldl, where the same function is repeatedly applied on a term. As foldl-like functions occur more often in higher-order rewriting, we should also address this.

To handle iteration, we define: for a function $Q \in A \Longrightarrow A$ and natural number n, let $Q^n(a)$ indicate repeated function application; that is, $Q^0(a) = a$ and $Q^{n+1}(a) = Q^n(Q(a))$.

▶ **Lemma 25.** *Suppose F is $wm(\overrightarrow{\mathcal{M}_\sigma}, \mathcal{M}_{\tau\Rightarrow\tau})$ and G is $wm(\overrightarrow{\mathcal{M}_\sigma}; \mathbb{N})$. Suppose that for all $u^1 \in \mathcal{M}_{\sigma_1}, \dots, u^k \in \mathcal{M}_{\sigma_k}$ and $v \in \mathcal{M}_\tau$ we have: $F(u^1, \dots, u^k, v) \sqsupseteq_\tau v$. Then the function $(x^1, \dots, x^k) \mapsto F(x^1, \dots, x^k)^{G(x^1,\dots,x^k)}$ is $wm(\overrightarrow{\mathcal{M}_\sigma}, \mathcal{M}_{\tau\Rightarrow\tau})$.*

With this in hand, we can orient the foldl rules of Example 2.

▶ **Example 26.** For $F \in \mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}\Rightarrow\mathsf{nat}}$ and $x, y \in \mathcal{M}_{\mathsf{nat}}$, let *Helper* be defined by:

$$\mathit{Helper}(F, y, x) = \langle F(x, y)_{\mathsf{c}},\ \max(x_{\mathsf{s}}, F(x, y)_{\mathsf{s}})\rangle.$$

Then *Helper* is $\mathrm{wm}(\mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}\Rightarrow\mathsf{nat}}, \mathcal{M}_{\mathsf{nat}}, \mathcal{M}_{\mathsf{nat}}; \mathcal{M}_{\mathsf{nat}})$ and strict in its third argument by Lemma 23(1,2,3,6,7), Hence, *Helper* is $\mathrm{wm}(\mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}\Rightarrow\mathsf{nat}}, \mathcal{M}_{\mathsf{nat}}; \mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}})$. Since, in general, $\mathtt{costof}_{\mathsf{nat}}(F(x, y)) \geq \mathtt{costof}_{\mathsf{nat}}(x)$, we have $\mathit{Helper}(F, y, x) \sqsupseteq_{\mathsf{nat}} x$. Using Lemma 25, we therefore see that the function $(F, z, xs) \mapsto \mathit{Helper}(F, \langle xs_{\mathsf{c}}, xs_{\mathsf{m}}\rangle)^{xs_{\mathsf{l}}}(z)$ is weakly monotonic, and strict in its second argument. This ensures that the following function is in $\mathcal{M}$.

$$[\![\mathsf{foldl}(F, z, xs)]\!] = \mathit{Helper}(F, \langle xs_{\mathsf{c}}, xs_{\mathsf{m}}\rangle)^{xs_{\mathsf{l}}}(\langle 1 + xs_{\mathsf{c}} + xs_{\mathsf{l}} + F(0_{\mathsf{nat}}, 0_{\mathsf{nat}})_{\mathsf{c}} + z_{\mathsf{c}}, z_{\mathsf{s}}\rangle)$$

This interpretation function is compatible with the rules for foldl in Example 2. First, we have $[\![\mathsf{foldl}(F, z, \mathsf{nil})]\!] = \langle\ 1 + F(0_{\mathsf{nat}}, 0_{\mathsf{nat}})_{\mathsf{c}} + z_{\mathsf{c}},\ z_{\mathsf{s}}\ \rangle \sqsupset_{\mathsf{nat}} \langle z_{\mathsf{c}}, z_{\mathsf{s}}\rangle = z$, which orients the first rule. For the second, we will use the general property that (**) $F(\mathtt{addc}(n, x), y) \sqsupseteq \mathtt{addc}(n, F(x, y))$ (Lemma B.6). We denote $A := \langle x_{\mathsf{c}} + xs_{\mathsf{c}}, \max(x_{\mathsf{s}}, xs_{\mathsf{m}})\rangle$ and $B := 1 + xs_{\mathsf{c}} + xs_{\mathsf{l}} + F(0_{\mathsf{nat}}, 0_{\mathsf{nat}})_{\mathsf{c}} + z_{\mathsf{c}}$. Then we have $[\![\mathsf{foldl}(F, z, x : xs)]\!] = \mathit{Helper}(F, A)^{xs_{\mathsf{l}}+1}(\langle B + x_{\mathsf{c}} + 1, z_{\mathsf{s}}\rangle)$, which:

$$\begin{array}{ll}
\sqsupset_{\mathsf{nat}} & \mathit{Helper}(F, A)^{xs_{\mathsf{l}}}(\mathit{Helper}(F, A, \langle B, z_{\mathsf{s}}\rangle)) \text{ because } \langle B + x_{\mathsf{c}} + 1, z_{\mathsf{s}}\rangle \sqsupset_{\mathsf{nat}} \langle B, z_{\mathsf{s}}\rangle \\
\sqsupseteq_{\mathsf{nat}} & \mathit{Helper}(F, A)^{xs_{\mathsf{l}}}(F(\langle B, z_{\mathsf{s}}\rangle, A)) \text{ because } \mathit{Helper}(F, n, m) \sqsupseteq_{\mathsf{nat}} F(m, n) \\
\sqsupseteq_{\mathsf{nat}} & \mathit{Helper}(F, \langle xs_{\mathsf{c}}, xs_{\mathsf{m}}\rangle)^{xs_{\mathsf{l}}}(F(\langle B, z_{\mathsf{s}}\rangle, x)) \text{ because } A \sqsupseteq_{\mathsf{nat}} \langle xs_{\mathsf{c}}, xs_{\mathsf{m}}\rangle \text{ and } A \sqsupseteq_{\mathsf{nat}} x \\
\sqsupseteq_{\mathsf{nat}} & \mathit{Helper}(F, \langle xs_{\mathsf{c}}, xs_{\mathsf{m}}\rangle)^{xs_{\mathsf{l}}}(\mathtt{addc}_{\mathsf{nat}}(1 + xs_{\mathsf{c}} + xs_{\mathsf{l}} + F(0_{\mathsf{nat}}, 0_{\mathsf{nat}})_{\mathsf{c}}, F(z, x))) \text{ by (**)} \\
= & [\![\mathsf{foldl}(F, (F\ z\ x), xs)]\!].
\end{array}$$

The interpretation in Example 26 may *seem* too convoluted for practical use: it does not obviously tell us something like "F is applied a linear number of times on terms whose size is bounded by n". However, its value becomes clear when we plug in specific bounds for F.

▶ **Example 27.** The function sum, defined in Example 1, could alternatively be defined in terms of foldl: let $\mathsf{sum}(xs) \to \mathsf{foldl}(\lambda xy.(x \oplus y), 0, xs)$. To find an interpretation for this function, we use the interpretation functions for $\mathsf{0}$, s, nil, cons and $\oplus$ from Example 10. Then $[\![\lambda xy.(x \oplus y)]\!] = d, e \mapsto (d_{\mathsf{c}} + e_{\mathsf{c}} + e_{\mathsf{s}} + 3, d_{\mathsf{s}} + e_{\mathsf{s}})$. We easily see that $\mathit{Helper}([\![\lambda xy.(x \oplus y)]\!], \langle xs_{\mathsf{c}}, xs_{\mathsf{m}}\rangle, z) = \langle z_{\mathsf{c}} + xs_{\mathsf{c}} + xs_{\mathsf{m}} + 3, z_{\mathsf{s}} + xs_{\mathsf{m}}\rangle$. Importantly, the iteration variable z is used in a very innocent way: although its size is increased, this increase is by the same number (xs_{m}) in every iteration step. Moreover, the length of z does not affect the evaluation cost. Hence, we can choose $[\![\mathsf{sum}(xs)]\!] = \langle 5 + xs_\mathsf{c} + xs_\mathsf{l} + xs_\mathsf{l} * (xs_\mathsf{c} + xs_\mathsf{m} + 3), xs_\mathsf{l} * xs_\mathsf{m}\rangle$. This is close to the interpretation from Example 10 but differs both in a small overhead for the β-reductions, and because our interpretation of foldl slightly overestimates the true cost.

This approach can be used to obtain bounds for any function that may be defined in terms of foldl, which includes many first-order functions. For example, with a small change to the signature of foldl, we could let $\mathsf{rev}(xs) = \mathsf{foldl}(\lambda xy.(y : x), \mathsf{nil}, xs)$; however, this would necessitate corresponding changes in the interpretation of foldl.

5 Finding complexity bounds

A key notion in complexity analysis of first-order rewriting is *runtime complexity*. In this section, we will define a conservative notion of runtime complexity for higher-order term rewriting, and show how our interpretations can be used to find runtime complexity bounds.

In first-order (and many-sorted) term rewriting, a *defined symbol* is any function symbol f such that there is a rule $\mathsf{f}(\ell_1, \dots, \ell_k) \to r$ in the system; all other symbols are called *constructors*. A *ground constructor term* is a ground term without defined symbols. A *basic term* has the form $\mathsf{f}(s_1, \dots, s_k)$ with f a defined symbol and $s_1, \dots, s_k$ all ground constructor terms. The *runtime complexity* of a TRS is then a function φ in $(\mathbb{N} \setminus \{0\}) \Longrightarrow \mathbb{N}$ that maps each n to a number $\varphi(n)$ so that for every basic term s of size at most n: $\mathtt{dh}_\mathcal{R}(s) \le \varphi(n)$.

The comparable notion of *derivational complexity* considers the derivation height for arbitrary ground terms of size n, but we will not use that here, since it can often give very high bounds that are not necessarily representative for realistic use of the system. In practice, a computation with a TRS would typically start with a main function, which takes *data* (e.g., natural numbers, lists) as input. This is exactly a basic term. Hence, the notion of runtime complexity roughly captures the worst-case number of steps for a realistic computation.

It is not obvious how this notion translates to the higher-order setting. It may be tempting to literally apply the definition to an AFS, but a "ground constructor term" (or perhaps "closed constructor term") is not a natural concept in higher-order rewriting; it does not intuitively capture data. Moreover, we would like to create a *robust* notion which can be extended to simple functional programming languages, so which is not subject to minor language difference like whether partial application of function symbols is allowed.

Instead, there are two obvious ways to capture the idea of input in higher-order rewriting:

- *closed irreducible terms*; this includes all ground constructor terms, but also for instance $\lambda x.\mathsf{0} \oplus x$ (but not $\lambda x.x \oplus \mathsf{0}$, since this can be rewritten following the rules in Example 1);
- *data*: this includes only ground constructor terms with no higher-order subterms.

As we observed in Example 12, the size of a higher-order term does not capture its behaviour. Hence, a notion of runtime complexity using closed irreducible terms is not obviously meaningful – and might be closer to *derivational* complexity due to defined symbols inside abstractions. Therefore, we here take the conservative choice and consider *data*.

▶ **Definition 28.** *In an AFS $(\mathcal{F}, \mathcal{R})$, a* data constructor *is a function symbol $\mathsf{c} :: [\iota_1 \times \dots \times \iota_k] \Rightarrow \iota_0$ with each $\iota_i \in \mathcal{S}$, such that there is no rule of the form $\mathsf{c}(\ell_1, \dots, \ell_k) \to r$. A* data term *is a term $\mathsf{c}(d_1, \dots, d_k)$ such that c is a constructor and all d_i are also data terms.*

In practice, a sort is defined by its data constructors. For example, nat is defined by 0 and s, and list by nil and cons. In typical examples of first- and higher-order term rewriting systems, rules are defined to exhaustively pattern match on all constructors for a sort.

With this definition, we can conservatively extend the original notion of runtime complexity to be applicable to both many-sorted and higher-order term rewriting.

▶ **Definition 29.** *A* basic term *is a term of the form* $\mathsf{f}(d_1, \dots, d_k)$ *with all* d_i *data terms and* f *not a data constructor. We let* $|d|$ *denote the total number of symbols in a basic term* d.

The runtime complexity *of an AFS is a function* $\varphi \in (\mathbb{N} \setminus \{0\}) \Longrightarrow \mathbb{N}$ *so that for all* n *and basic terms* d*, with* $|d| \leq n$*:* $\mathtt{dh}_{\mathcal{R}}(d) \leq \varphi(n)$*.*

Note that if $\mathsf{f}(d_1, \dots, d_k)$ is a basic term, then $\mathsf{f} :: [\iota_1 \times \cdots \times \iota_k] \Rightarrow \tau$ with all ι_i sorts. Hence, higher-order runtime complexity considers the same (first-order) notion of basic terms as the first-order case; terms such as $\mathsf{map}(F, s)$ or even $\mathsf{map}(\lambda x.\mathsf{s}(x), \mathsf{nil})$ are not basic. One might reasonably question whether such a first-order notion is useful when studying the complexity of *higher-order* term rewriting. However, we argue that it is: runtime complexity aims to address the length of computations that begin at a typical starting point. When performing a *full program* analysis of an AFS, the computation will still typically start in a basic term, for instance; the entry-point symbol main applied to some user input $d_1, \dots, d_k$.

▶ **Example 30.** We consider an AFS from the Termination Problem Database, v11.0 [16].

$$\begin{array}{rclrcl}
x \oplus \mathsf{0} & \to_{\mathcal{R}} & x & \mathsf{rec}(\mathsf{0}, y, F) & \to_{\mathcal{R}} & y \\
x \oplus \mathsf{s}(y) & \to_{\mathcal{R}} & \mathsf{s}(x \oplus y) & \mathsf{rec}(\mathsf{s}(x), y, F) & \to_{\mathcal{R}} & F \cdot x \cdot \mathsf{rec}(x, y, F) \\
 & & & x \otimes y & \to_{\mathcal{R}} & \mathsf{rec}(y, \mathsf{0}, \lambda n.\lambda m.x \oplus m)
\end{array}$$

Here, $\mathsf{rec} :: [\mathsf{nat} \times \mathsf{nat} \times (\mathsf{nat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat})] \Rightarrow \mathsf{nat}$. The only basic terms have the form $\mathsf{s}^n(\mathsf{0}) \oplus \mathsf{s}^m(\mathsf{0})$ or $\mathsf{s}^n(\mathsf{0}) \otimes \mathsf{s}^m(\mathsf{0})$. Using our method, we obtain cubic runtime complexity; to be precise: $\mathcal{O}(m^2 * n)$. The interpretation functions are found in Appendix A.

To derive runtime complexity for both first- and higher-order rewriting, our approach is to consider bounds for the functions $\mathcal{J}_{\mathsf{f}}$; we only need to consider the first-order symbols f.

▶ **Definition 31.** *Let* $P \in \mathcal{M}_{\iota_1 \Rightarrow \dots \Rightarrow \iota_m \Rightarrow \kappa}$ *be of the form* $P(x^1, \dots, x^m) = \langle P_1(x^1, \dots, x^m), \dots, P_{K[\kappa]}(x^1, \dots, x^m)\rangle$*. Then* P *is* linearly bounded *if each component function* P_l *of* P *is upper-bounded by a positive linear polynomial, i.e., there is a constant* $a \in \mathbb{N}$ *such that* $P_l(x^1, \dots, x^m) \leq a*(1+\sum_{i=1}^{m} \sum_{j=1}^{K[\iota_i]} x_j^i)$*. We say that* P *is* additive *if there exists a constant* $a \in \mathbb{N}$ *such that* $\sum_{l=1}^{K[\kappa]} P_l(x^1, \dots, x^m) \leq a + \sum_{i=1}^{m} \sum_{j=1}^{K[\iota_i]} x_j^i$*.*

By this definition, P_l is not required to be a linear function, only to be bounded by one. This means that for instance $\min(x_j^i, 2 * x_b^a)$ can be used, but $x_j^i * x_b^a$ cannot. It is easily checked that all the data constructors in this paper have an additive interpretation. For example, for $\mathcal{J}_{\mathsf{cons}}$: $(x_{\mathsf{c}} + xs_{\mathsf{c}}) + (x_{\mathsf{l}} + 1) + \max(x_{\mathsf{s}}, xs_{\mathsf{m}}) \leq 1 + x_{\mathsf{c}} + x_{\mathsf{s}} + xs_{\mathsf{c}} + xs_{\mathsf{l}} + xs_{\mathsf{s}}$.

▶ **Lemma 32.** *Let* $(\mathcal{F}, \mathcal{R})$ *be an AFS or TRS that is compatible with a strongly monotonic algebra with interpretation function* $\mathcal{J}$*. Then:*

1. *if* $\mathcal{J}_{\mathsf{c}}$ *is additive for all data constructors* c*, then there exists a constant* $b > 0$ *in* $\mathbb{N}$ *so that for all data terms* s*: if* $|s| \leq n$ *then* $[\![s]\!]_l \leq b * n$*, for each component* $[\![s]\!]_l$ *of* $[\![s]\!]$*;*
2. *if* $\mathcal{J}_{\mathsf{c}}$ *is linearly bounded for all data constructors* c*, then there exists a constant* $b > 0$ *in* $\mathbb{N}$ *so that for all data terms* s*: if* $|s| \leq n$ *then* $[\![s]\!]_l \leq 2^{b*n}$*, for each component* $[\![s]\!]_l$ *of* $[\![s]\!]$*.*

By using Lemma 32, we quickly obtain some ways to bound runtime complexity:

▶ **Corollary 33.** *Let* $(\mathcal{F}, \mathcal{R})$ *be an AFS or TRS that is compatible with a strongly monotonic algebra with interpretation function* $\mathcal{J}$*, and let* $\mathcal{F}_C$ *denote its set of data constructors, and* $\mathcal{F}_B$ *the set of all other symbols* f *with a signature* $\mathsf{f} :: [\iota_1 \times \cdots \times \iota_m] \Rightarrow \tau$*. Then:*

- *if* $\mathcal{J}_{\mathsf{f}}$ *is additive for all* $\mathsf{f} \in \mathcal{F}_C \cup \mathcal{F}_B$*, then* $(\mathcal{F}, \mathcal{R})$ *has linear runtime complexity;*
- *if* $\mathcal{J}_{\mathsf{c}}$ *is additive for all* $\mathsf{c} \in \mathcal{F}_C$ *and for all* $\mathsf{f} \in \mathcal{F}_B$*,* $\mathcal{J}_{\mathsf{f}}(\vec{x}) = (P_1(\vec{x}), \dots, P_k(\vec{x}))$ *where* P_1 *is bounded by a polynomial, then* $(\mathcal{F}, \mathcal{R})$ *has polynomial runtime complexity;*
- *if* $\mathcal{J}_{\mathsf{f}}$ *is linearly bounded for all* $\mathsf{f} \in \mathcal{F}_C \cup \mathcal{F}_B$*, then* $(\mathcal{F}, \mathcal{R})$ *has exponential runtime complexity.*

We could easily use these results as part of an automatic complexity tool – and indeed, combine them with other methods for complexity analysis. However, this is not truly our goal: runtime complexity is only a part of the picture, especially in higher-order term rewriting where we may want to analyse modules that get much more hairy input. Our technique aims to give more fine-grained information, where we consider the impact of input with certain properties – like the length of a list or the depth of a tree. For this, the person interested in the analysis should be the one to decide on the interpretations of the constructors.

With this information given, though, it should be possible to automatically find interpretations for the other functions. The search for the best strategy requires dedicated research, which we leave to future work; however, we expect Lemmas 23 and 25 to play a large role. We also note that while the cost component may depend on the other components, the other components (which represent a kind of size property) typically do not depend on the cost.

6 On Related Work

Rewriting. There are several first-order complexity techniques based on interpretations. For example, in [11], the consequences of using additive, linear, and polynomial interpretations to the natural numbers are investigated; and in [26], context-dependent interpretations are introduced, which map terms to real numbers to obtain tighter bounds. Most closely related to our approach are *matrix interpretations* [21, 35], and a technique by the first author for complexity analysis of conditional term rewriting [31]. In both cases, terms are mapped to tuples as they are in our approach, although neither considers sort information, and matrix interpretations use linear interpretation functions. Our technique is a generalisation of both.

Higher-order Rewriting. In *higher-order* term rewriting (but a formalism without λ-abstraction), Baillot and Dal Lago [10] develop a version of higher-order polynomial interpretations which, like the present work, is based on v.d. Pol's higher-order interpretations [39]. In similar ways to our Section 5, the authors enforce polynomial bounds on derivational complexity by imposing restrictions on the shape of interpretations. Their method differs from ours in various ways, most importantly by mapping terms to $\mathbb{N}$ rather than tuples. In addition, the interpretations are limited to higher-order polynomials. This yields an ordering with the subterm property (i.e., $\mathsf{f}(\dots, s, \dots) \sqsupset s$), which means that TRSs like Example 11 cannot be handled. Moreover, it is not possible to find a general interpretation for functions like foldl or rec; the method can only handle instances of foldl with a linear function.

Beyond this, it unfortunately seems that relatively little work has thus far been done on complexity analysis of higher-order term rewriting. However, complexity of *functional programs* is an active field of research with a close relation to higher-order term rewriting.

Functional Programming. There are various techniques to statically analyse resource use of functional programs. These may be fully automated [5, 9, 42], semi-automated designed to reason about programmer specified-bounds [45, 15, 23], or even manual techniques, integrated with type system or program logic semantics [14, 17]. We discuss the most pertinent ones.

An approach using rewriting for full-program analysis is to translate functional programs to TRSs [6], which can be analysed using first-order complexity techniques. This takes advantage of the large body of work on first-order complexity, but loses information; the transformation often yields a system that is harder to analyse than the original.

The research methodology in most studies in functional programming differs significantly from rewriting techniques. Nevertheless, there are some studies with clear connections to our approach; in particular our separation of cost and size (and other structural properties). Most relevant, in [19] the authors use a similar approach by giving semantics to a complexity-aware intermediate language allowing arbitrary user-defined notions for size – such as list length or maximum element size; recurrence relations are then extracted to represent the complexity.

Additionally, most modern complexity analysis is done via enhancements at the type system level [2, 5, 28, 40, 23, 20]. For example, types may be annotated with a counter, the heap size or a data type's size measure. Notably, a line of work on Resource-Aware ML [28, 37, 30] studies resource use of OCaml programs with methods based on Tarjan's amortized analysis [43]. Types are annotated with *potentials* (a cost measure), and type inference generates a set of linear constraints which is sent over to an external solver. For Haskell, Liquid Haskell [41, 44] provides a language to annotate types, which can be used to prove properties of the program; this was recently extended to include complexity [23]. Unlike RAML, this approach is not fully automatic: type annotations are checked, not derived.

These works in functional programming have a different purpose from ours: they study the resource use in a specific language, typically with a fixed evaluation strategy. Our method, in contrast, allows for arbitrary evaluation, which could be specified to various strategies in future work. Moreover, most of these works limit interest to full-program analysis. We do this for runtime complexity, but our method offers more, by providing general interpretations for individual functions like map or foldl. Similarly, most of these works impose additive type annotations for the constructors; we do not restrict the constructor interpretations outside Lemma 32. On the other hand, many do consider (shallow) polymorphism, which we do not.

While in functional programming one considers resource usage [40, 28], rewriting is concerned with the number of steps, which can be translated to a form of resource measure if the true cost of each step is kept low. This is achieved by imposing restrictions on reduction strategy and term representation [1, 18]. Our approach carries the blessing of being general and machine independent and the curse of not necessarily being a reasonable cost model.

7 Conclusion and Future Work

In this paper, we have introduced tuple interpretations for many-sorted and higher-order term rewriting. This includes providing a new definition of strongly monotonic algebras, a compatibility theorem, a function *MakeSM* that orients β- and η-reductions, and several lemmas to prove monotonicity of interpretation functions. We also show that for certain restrictions on interpretation functions, we find linear, polynomial or exponential bounds on runtime complexity (for a simple but natural definition of higher-order runtime complexity).

Our type-based, semantical approach allows us to relate various "size" notions (e.g., list length, tree depth, term size. etc.) to reduction cost, and thus offers a more fine-grained analysis than traditional notions like runtime complexity. Most importantly, we can express the complexity of a higher-order function in terms of the behaviour of its (function) arguments. In the future, we hope that this could be used towards a truly higher-order complexity notion.

Some further examples and weaknesses. Aside from the three higher-order examples in this paper, we have successfully applied our method to a variety of higher-order benchmarks in the Termination Problem Database [16], all with additive interpretations for the constructors. Two additional examples (filter and deriv) are included in Appendix A.

A clear weakness we discovered was that our method can only handle "plain function-passing" systems [33]. That is, we typically do not succeed on systems where a variable of function type occurs inside a subterm of base type, and occurs outside this subterm in the right-hand side. Examples of such systems are ordrec, which has a rule $\mathsf{ordrec}(\mathsf{lim}(F), x, G, H) \to_{\mathcal{R}} H \cdot F \cdot (\lambda n.\mathsf{ordrec}(F \cdot n, x, G, H))$ with $\mathsf{lim} :: [\mathsf{nat} \Rightarrow \mathsf{ord}] \Rightarrow \mathsf{ord}$, and apply, which has a rule $\mathsf{lapply}(x, \mathsf{fcons}(F, xs)) \to_{\mathcal{R}} F \cdot \mathsf{lapply}(x, xs)$ with $\mathsf{fcons} :: [(\mathsf{a} \Rightarrow \mathsf{a}) \times \mathsf{listf}] \Rightarrow \mathsf{listf}$.

Future work. We intend to consider the effect of different evaluation strategies, such as innermost evaluation, weak-innermost evaluation (where rewriting below an abstraction is not allowed, as is commonly the case in functional programming) or outermost evaluation. This extension is likely to be an important step towards another goal: to more closely relate our complexity notion to a reasonable measure of resource consumption in a rewriting engine.

In addition, we plan to extend first-order complexity techniques like dependency tuples [24], which may allow us to overcome the weakness described above. Another goal is to enrich our type system to support a notion of polymorphism and add polymorphic interpretations into the play. We also aim to develop a tool to automatically find suitable tuple interpretations.

References

1. B. Accatoli and U. Dal Lago. (leftmost-outermost) beta reduction is invariant, indeed. *LMCS*, 2016. `doi:10.2168/LMCS-12(1:4)2016`.
2. S. Alves, D. Kesner, and D. Ventura. A quantitative understanding of pattern matching. In *Proc. TYPES*, LIPIcs, 2020. `doi:10.4230/LIPIcs.TYPES.2019.3`.
3. T. Arai and G. Moser. Proofs of termination of rewrite systems for polytime functions. In *Proc. FSTTCS*, 2005. `doi:10.1007/11590156_4`.
4. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *TCS*, 2000. `doi:10.1016/S0304-3975(99)00207-8`.
5. M. Avanzini and U. Dal Lago. Automating sized-type inference for complexity analysis. In *Proc. ICFP*, 2017. `doi:10.1145/3110287`.
6. M. Avanzini, U. Dal Lago, and G. Moser. Analysing the complexity of functional programs: Higher-order meets first-order. In *Proc. ICFP*, 2015. `doi:10.1145/2784731.2784753`.
7. M. Avanzini and G. Moser. Complexity analysis by rewriting. In *Proc. FLOPS*, 2008. `doi:10.1007/978-3-540-78969-7_11`.
8. M. Avanzini and G. Moser. Closing the gap between runtime complexity and polytime computability. In *Proc. RTA*, 2010. `doi:10.4230/LIPIcs.RTA.2010.33`.
9. Ralph B. Automated higher-order complexity analysis. *TCS*, 2004. `doi:10.1016/j.tcs.2003.10.022`.
10. P. Baillot and U. Dal Lago. Higher-order interpretations and program complexity. *IC*, 2016. `doi:10.1016/j.ic.2015.12.008`.
11. G. Bonfante, A. Cichon, J. Marion, and H. Touzet. Complexity classes and rewrite systems with polynomial interpretation. In *Proc. CSL*, 1998. `doi:10.1007/10703163_25`.
12. G. Bonfante, J. Marion, and J. Moyen. On lexicographic termination ordering with space bound certifications. In *Proc. PSI*, 2001. `doi:10.1007/3-540-45575-2_46`.
13. M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating runtime and size complexity analysis of integer programs. In *Proc. TACAS*, 2014. `doi:10.1007/978-3-642-54862-8_10`.
14. Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-end verification of stack-space bounds for C programs. *SIGPLAN Not.*, 2014. `doi:10.1145/2666356.2594301`.
15. E. Çiçek, D. Garg, and U. Acar. Refinement types for incremental computational complexity. In *Proc. ESOP*, 2015. `doi:10.1007/978-3-662-46669-8_17`.
16. Community. Termination problem database, version 11.0. Directory Higher_Order_Rewriting_Union_Beta/Mixed_HO_10/, 2019. URL: `http://termination-portal.org/wiki/TPDB`.
17. U. Dal Lago and M. Gaboardi. Linear dependent types and relative completeness. In *Proc. LICS*, 2011. `doi:10.1109/LICS.2011.22`.
18. U. Dal Lago and S. Martini. Derivational complexity is an invariant cost model. In *Proc. FOPARA*, 2010. `doi:10.1007/978-3-642-15331-0_7`.
19. N. Danner, D.R. Licata, and R. Ramyaa. Denotational cost semantics for functional languages with inductive types. In *Proc. ICFP*, 2015. `doi:10.1145/2784731.2784749`.

20 A. Das, S. Balzer, J. Hoffman, F. Pfenning, and I. Santurkar. Resource-aware session types for digital contracts, 2019. arXiv:1902.06056.

21 J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *JAR*, 2008. doi:10.1007/11814771_47.

22 C. Fuhs and C. Kop. Polynomial interpretations for higher-order rewriting. In *Proc. RTA*, 2012. doi:10.4230/LIPIcs.RTA.2012.176.

23 M. A. T. Handley, N. Vazou, and G. Hutton. Liquidate your assets: Reasoning about resource usage in liquid haskell. *ACM POPL*, 2019. doi:10.1145/3371092.

24 N. Hirokawa and G. Moser. Automated complexity analysis based on the dependency pair method. In *Proc. IJCAR*, 2008. doi:10.1007/978-3-540-71070-7_32.

25 D. Hofbauer. Termination proofs by multiset path orderings imply primitive recursive derivation lengths. *TCS*, 1992. doi:10.1007/3-540-53162-9_50.

26 D. Hofbauer. Termination proofs by context-dependent interpretations. In *Proc. RTA*, 2001. doi:10.1007/3-540-45127-7_10.

27 D. Hofbauer and C. Lautemann. Termination proofs and the length of derivations. In *Proc. RTA*, 1989. doi:10.1007/3-540-51081-8_107.

28 J. Hoffmann, K. Aehlig, and M. Hofmann. Resource aware ml. In *Proc. CAV*, 2012. doi:10.1007/978-3-642-31424-7_64.

29 J. Jouannaud and M. Okada. A computation model for executable higher-order algebraic specification languages. In *Proc. LICS*, 1991. doi:10.1109/LICS.1991.151659.

30 D. M. Kahn and J. Hoffmann. Exponential automatic amortized resource analysis. In *Proc. FoSSaCS*, 2020. doi:10.1007/978-3-030-45231-5_19.

31 C. Kop, A. Middeldorp, and T. Sternagel. Complexity of conditional term rewriting. *LMCS*, 2017. doi:10.23638/LMCS-13(1:6)2017.

32 C. Kop and D. Vale. Tuple interpretations for higher-order complexity (extended), 2021. arXiv:2105.01112.

33 K. Kusakari and M. Sakai. Enhancing dependency pair method using strong computability in simply-typed term rewriting. *AAECC*, 2007. doi:10.1007/s00200-007-0046-9.

34 G. Moser. Derivational complexity of knuth-bendix orders revisited. In *Proc. LPAR*, 2006. doi:10.1007/11916277_6.

35 G. Moser, A. Schnabl, and J. Waldmann. Complexity analysis of term rewriting based on matrix and context dependent interpretations. In *Proc. FSTTCS*, 2008. doi:10.4230/LIPIcs.FSTTCS.2008.1762.

36 T. Nipkow. Higher-order critical pairs. In *Proc. LICS*, 1991. doi:10.1109/LICS.1991.151658.

37 Y. Niu and J. Hoffmann. Automatic space bound analysis for functional programs with garbage collection. In *Proc. LPAR*, 2018. doi:10.29007/xkwx.

38 E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002. doi:10.1007/978-1-4757-3661-8.

39 J.C. van de Pol. *Termination of Higher-order Rewrite Systems*. PhD thesis, University of Utrecht, 1996. URL: https://www.cs.au.dk/~jaco/papers/thesis.pdf.

40 V. Rajani, M. Gaboardi, D. Garg, and J. Hoffmann. A unifying type-theory for higher-order (amortized) cost analysis. *ACM POPL*, 2021. doi:10.1145/3434308.

41 P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. *SIGPLAN Not.*, 2008. doi:10.1145/1379022.1375602.

42 M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *Proc. CAV*, 2014. doi:10.1007/978-3-319-08867-9_50.

43 R. E. Tarjan. Amortized computational complexity. *ADM*, 1985. doi:10.1137/0606031.

44 N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In *Proc. ESOP*, 2013. doi:10.1007/978-3-642-37036-6_13.

45 P. Wang, D. Wang, and A. Chlipala. Timl: A functional language for practical complexity analysis with invariants. *ACM POPL*, 2017. doi:10.1145/3133903.

A Extended examples

Extrec. The system in Example 30 has the following interpretation:

$$\begin{aligned}
[\![0]\!] &= \langle 0,0\rangle \\
[\![\mathsf{s}(x)]\!] &= \langle x_\mathsf{c}, x_\mathsf{s}+1\rangle \\
[\![x \oplus y]\!] &= \langle x_\mathsf{c}+y_\mathsf{c}+y_\mathsf{s}+1, x_\mathsf{s}+y_\mathsf{s}\rangle \\
[\![x \otimes y]\!] &= \langle 1+y_\mathsf{s}*(x_\mathsf{c}+y_\mathsf{c}+x_\mathsf{s}*(y_\mathsf{s}+1)/2+3), x_\mathsf{s}*y_\mathsf{s}\rangle \\
[\![\mathsf{rec}(x,y,F)]\!] &= \mathit{Helper}(x,F)^{x_\mathsf{s}}(\ \langle 1+x_\mathsf{c}+y_\mathsf{c}+x_\mathsf{s}+F(0_\mathsf{nat},0_\mathsf{nat})_\mathsf{c}, y_\mathsf{s}\rangle\ ) \\
\mathit{Helper}(x,F) &= z \mapsto \langle F(x,z)_\mathsf{c}, \max(z_\mathsf{s}, F(x,z)_\mathsf{s})\rangle
\end{aligned}$$

Then we always have (*A) $\mathit{Helper}(x,F)(z) \sqsupseteq_\mathsf{nat} z$ because $F(x,z)_\mathsf{c} \geq z_\mathsf{c}$ which we will see in Lemma B.8, and clearly $\max(z_\mathsf{s}, F(x,z)_\mathsf{s}) \geq z_\mathsf{s}$. Hence, the monotonicity requirements are satisfied. We also clearly have (*B) $\mathit{Helper}(x,F)(z) \sqsupseteq_\mathsf{nat} F(x,z)$, since clearly $\max(z_\mathsf{s}, F(x,z)_\mathsf{s}) \geq F(x,z)_\mathsf{s}$. For most rules, it is easy to see that $[\![\ell]\!] \sqsupset [\![r]\!]$. We only show:

- $[\![\mathsf{rec}(\mathsf{s}(x),y,F)]\!] \sqsupset_\mathsf{nat} [\![F \cdot x \cdot \mathsf{rec}(x,y,F)]\!]$:
  $[\![\mathsf{rec}(\mathsf{s}(x),y,F)]\!] = \mathit{Helper}(\langle x_\mathsf{c}, x_\mathsf{s}+1\rangle, F)^{x_\mathsf{s}+1}(\ \langle 1+x_\mathsf{c}+y_\mathsf{c}+(x_\mathsf{s}+1)+F(0_\mathsf{nat},0_\mathsf{nat})_\mathsf{c}, y_\mathsf{s}\rangle\ ) =$
  $\mathit{Helper}(\langle x_\mathsf{c}, x_\mathsf{s}+1\rangle, F)(\mathit{Helper}(\langle x_\mathsf{c}, x_\mathsf{s}+1\rangle, F)^{x_\mathsf{s}}(\ \langle 2+x_\mathsf{c}+y_\mathsf{c}+x_\mathsf{s}+F(0_\mathsf{nat},0_\mathsf{nat})_\mathsf{c}, y_\mathsf{s}\rangle\ )) \sqsupseteq_\mathsf{nat}$
  $F(\langle x_\mathsf{c}, x_\mathsf{s}+1\rangle, \mathit{Helper}(\langle x_\mathsf{c}, x_\mathsf{s}+1\rangle, F)^{x_\mathsf{s}}(\ \langle 2+x_\mathsf{c}+y_\mathsf{c}+x_\mathsf{s}+F(0_\mathsf{nat},0_\mathsf{nat})_\mathsf{c}, y_\mathsf{s}\rangle\ ))$ by (*B),
  $\sqsupset_\mathsf{nat} F(\langle x_\mathsf{c}, x_\mathsf{s}\rangle, \mathit{Helper}(\langle x_\mathsf{c}, x_\mathsf{s}\rangle, F)^{x_\mathsf{s}}(\ \langle 1+x_\mathsf{c}+y_\mathsf{c}+x_\mathsf{s}+F(0_\mathsf{nat},0_\mathsf{nat})_\mathsf{c}, y_\mathsf{s}\rangle\ ))$ by monotonicity,
  $= F(x, \mathit{Helper}(x,F)^{x_\mathsf{s}}(\ \langle 1+x_\mathsf{c}+y_\mathsf{c}+x_\mathsf{s}+F(0_\mathsf{nat},0_\mathsf{nat})_\mathsf{c}, y_\mathsf{s}\rangle\ )) = [\![F \cdot x \cdot \mathsf{rec}(x,y,F)]\!]$.
- $[\![x \otimes y]\!] \sqsupset_\mathsf{nat} [\![\mathsf{rec}(y, 0, \lambda n.\lambda m.x \oplus m)]\!]$:
  - $[\![\lambda n.\lambda m.x \oplus m]\!] = n \mapsto m \mapsto \langle x_\mathsf{c}+n_\mathsf{c}+m_\mathsf{c}+m_\mathsf{s}+3, x_\mathsf{s}+m_\mathsf{s}\rangle$
  - $\mathit{Helper}(y, [\![\lambda n.\lambda m.x \oplus m]\!]) = m \mapsto \langle x_\mathsf{c}+y_\mathsf{c}+m_\mathsf{c}+m_\mathsf{s}+3, x_\mathsf{s}+m_\mathsf{s}\rangle$
  - For given i, $\mathit{Helper}(y, [\![\lambda n.\lambda m.x \oplus m]\!])^i(m)_\mathsf{s} = (\sum_{j=0}^{i} x_\mathsf{s}) + m_\mathsf{s} = x_\mathsf{s} * i + m_\mathsf{s}$
  - $\mathit{Helper}(y, [\![\lambda n.\lambda m.x \oplus m]\!])^{y_\mathsf{s}} = m \mapsto \langle \sum_{i=1}^{y_\mathsf{s}}(x_\mathsf{c}+y_\mathsf{c}+(x_\mathsf{s}*i+m_\mathsf{s})+3)+m_\mathsf{c}, y_\mathsf{s}*x_\mathsf{s}+m_\mathsf{s}\rangle =$ $\langle y_\mathsf{s}*(x_\mathsf{c}+y_\mathsf{c}+m_\mathsf{s}+3)+x_\mathsf{s}*\sum_{i=1}^{y_\mathsf{s}}(i)+m_\mathsf{c}, y_\mathsf{s}*x_\mathsf{s}+m_\mathsf{s}\rangle = \langle y_\mathsf{s}*(x_\mathsf{c}+y_\mathsf{c}+m_\mathsf{s}+3)+x_\mathsf{s}*(y_\mathsf{s}*(y_\mathsf{s}+1)/2)+m_\mathsf{c}, y_\mathsf{s}*x_\mathsf{s}+m_\mathsf{s}\rangle = \langle y_\mathsf{s}*(x_\mathsf{c}+y_\mathsf{c}+m_\mathsf{s}+x_\mathsf{s}*(y_\mathsf{s}+1)/2+3)+m_\mathsf{c}, y_\mathsf{s}*x_\mathsf{s}+m_\mathsf{s}\rangle$

  Hence, $[\![x \otimes y]\!] = \langle 1+y_\mathsf{s}*(x_\mathsf{c}+y_\mathsf{c}+x_\mathsf{s}*(y_\mathsf{s}+1)/2+3), x_\mathsf{s}*y_\mathsf{s}\rangle \sqsupset_\mathsf{nat} \langle y_\mathsf{s}*(x_\mathsf{c}+y_\mathsf{c}+x_\mathsf{s}*(y_\mathsf{s}+1)/2+3)+0, x_\mathsf{s}*y_\mathsf{s}+0\rangle = [\![\mathsf{rec}(y,0,\lambda n.\lambda m.x \oplus m)]\!]$

Filter. We show an example from the Termination Problem Database, v11.0.

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{rand}(x) & \to_\mathcal{R} & x & \mathsf{filter}(F,\mathsf{nil}) & \to_\mathcal{R} & \mathsf{nil} \\
\mathsf{rand}(\mathsf{s}(x)) & \to_\mathcal{R} & \mathsf{rand}(x) & \mathsf{filter}(F, x : xs) & \to_\mathcal{R} & \mathsf{consif}(F \cdot x, x, \mathsf{filter}(F, xs)) \\
\mathsf{bool}(0) & \to_\mathcal{R} & \mathsf{false} & \mathsf{consif}(\mathsf{true}, x, xs) & \to_\mathcal{R} & x : xs \\
\mathsf{bool}(\mathsf{s}(0)) & \to_\mathcal{R} & \mathsf{true} & \mathsf{consif}(\mathsf{false}, x, xs) & \to_\mathcal{R} & xs
\end{array}$$

We will use the notation q instead of xs to avoid clutter in the proof. We let $\mathcal{M}_\mathsf{nat} = \mathbb{N}^2$ and $\mathcal{M}_\mathsf{list} = \mathbb{N}^3$ as before, and additionally let $\mathcal{M}_\mathsf{boolean} = \mathbb{N}$ (so no size components). We let:

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
[\![\mathsf{true}]\!] & = & \langle 0\rangle & [\![\mathsf{s}(x)]\!] & = & \langle x_\mathsf{c}, x_\mathsf{s}+1\rangle & [\![\mathsf{bool}(x)]\!] & = & \langle x_\mathsf{c}+1\rangle \\
[\![\mathsf{false}]\!] & = & \langle 0\rangle & [\![\mathsf{nil}]\!] & = & \langle 0,0,0\rangle & [\![\mathsf{rand}(x)]\!] & = & \langle 1+x_\mathsf{c}+x_\mathsf{s}, x_\mathsf{s}\rangle \\
[\![0]\!] & = & \langle 0,0\rangle & [\![x : q]\!] & = & \langle x_\mathsf{c}+q_\mathsf{c}, q_\mathsf{l}+1, \max(x_\mathsf{s}, q_\mathsf{m})\rangle
\end{array}$$

$$\begin{aligned}
[\![\mathsf{consif}(z,x,q)]\!] &= \langle z_\mathsf{c}+x_\mathsf{c}+q_\mathsf{c}+1, q_\mathsf{l}+1, \max(x_\mathsf{s}, q_\mathsf{m})\rangle \\
[\![\mathsf{filter}(F,q)]\!] &= \langle 1+(q_\mathsf{l}+1)*(2+q_\mathsf{c}+F(\langle q_\mathsf{c}, q_\mathsf{m}\rangle)_\mathsf{c}), q_\mathsf{l}, q_\mathsf{m}\rangle
\end{aligned}$$

It is easy to see that monotonicity requirements are satisfied. As for orienting the rules, we show only the second filter rule.

- $[\![\mathsf{filter}(F, x : q)]\!] \sqsupset_{\mathsf{list}} [\![\mathsf{consif}(F \cdot x, x, \mathsf{filter}(F, q))]\!]$
  $[\![\mathsf{filter}(F, x : q)]\!] = \langle 1{+}(q_\mathsf{l}{+}2){*}(2{+}x_\mathsf{c}{+}q_\mathsf{c}{+}F(\langle x_\mathsf{c}{+}q_\mathsf{c}, \max(x_\mathsf{s}, q_\mathsf{m})\rangle)_\mathsf{c}), q_\mathsf{l}{+}1, \max(x_\mathsf{s}, q_\mathsf{m})\rangle = \langle 3{+}x_\mathsf{c}{+}q_\mathsf{c}{+}F(\langle x_\mathsf{c}{+}q_\mathsf{c}, \max(x_\mathsf{s}, q_\mathsf{m})\rangle)_\mathsf{c}{+}(q_\mathsf{l}{+}1){*}(2{+}x_\mathsf{c}{+}q_\mathsf{c}{+}F(\langle x_\mathsf{c}{+}q_\mathsf{c}, \max(x_\mathsf{s}, q_\mathsf{m})\rangle)_\mathsf{c}), q_\mathsf{l}{+}1, \max(x_\mathsf{s}, q_\mathsf{m})\rangle \sqsupset_{\mathsf{list}} \langle 2{+}x_\mathsf{c}{+}F(\langle x_\mathsf{c}, x_\mathsf{s}\rangle)_\mathsf{c}{+}(q_\mathsf{l}{+}1){*}(2{+}q_\mathsf{c}{+}F(\langle q_\mathsf{c}, q_\mathsf{m}\rangle)_\mathsf{c}), q_\mathsf{l}{+}1, \max(x_\mathsf{s}, q_\mathsf{m})\rangle = \langle F(x)_\mathsf{c}{+}x_\mathsf{c}{+}(1{+}(q_\mathsf{l}{+}1){*}(2{+}q_\mathsf{c}{+}F(\langle q_\mathsf{c}, q_\mathsf{m}\rangle)_\mathsf{c})){+}1, q_\mathsf{l}{+}1, \max(x_\mathsf{s}, q_\mathsf{m})\rangle = \langle F(x)_\mathsf{c}{+}x_\mathsf{c}{+}[\![\mathsf{filter}(F, q)]\!]_\mathsf{c} + 1, [\![\mathsf{filter}(F, q)]\!]_\mathsf{l} + 1, \max(x_\mathsf{s}, [\![\mathsf{filter}(F, q)]\!]_\mathsf{m})\rangle = [\![\mathsf{consif}(F \cdot x, x, \mathsf{filter}(F, q))]\!]$.

Deriv. Our final example also comes from the termination problem database.

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{der}(\lambda x.y) & \to_{\mathcal{R}} & \lambda z.0 & \mathsf{der}(\lambda x.\mathsf{sin}(x)) & \to_{\mathcal{R}} & \lambda z.\mathsf{cos}(z) \\
\mathsf{der}(\lambda x.x) & \to_{\mathcal{R}} & \lambda z.1 & \mathsf{der}(\lambda x.\mathsf{cos}(x)) & \to_{\mathcal{R}} & \lambda z.\mathsf{min}(\mathsf{cos}(z)) \\
\mathsf{der}(\lambda x.\mathsf{plus}(F \cdot x, G \cdot x)) & \to_{\mathcal{R}} & \multicolumn{4}{l}{\lambda z.\mathsf{plus}(\mathsf{der}(F) \cdot z, \mathsf{der}(G) \cdot z)} \\
\mathsf{der}(\lambda x.\mathsf{times}(F \cdot x, G \cdot x)) & \to_{\mathcal{R}} & \multicolumn{4}{l}{\lambda z.\mathsf{plus}(\mathsf{times}(\mathsf{der}(F) \cdot z, G \cdot z), \mathsf{times}(F \cdot z, \mathsf{der}(G) \cdot z))} \\
\mathsf{der}(\lambda x.\mathsf{ln}(F \cdot x)) & \to_{\mathcal{R}} & \multicolumn{4}{l}{\lambda z.\mathsf{div}(\mathsf{der}(F) \cdot z, F \cdot z)}
\end{array}$$

With $\mathsf{der} :: [\mathsf{real} \Rightarrow \mathsf{real}] \Rightarrow \mathsf{real} \Rightarrow \mathsf{real}$. We let $\mathcal{M}_{\mathsf{real}} = \mathbb{N}^3$ where the first component indicates cost, and the second and third component roughly indicate the number of plus/times/ln occurrences and the number of times/ln occurrences respectively. We will denote x_s for x_2, and $x_\star$ for x_3. We use the following interpretation:

$$\begin{array}{rcl@{\qquad}rcl}
[\![0]\!] & = & \langle 0,0,0\rangle & [\![\mathsf{plus}(x,y)]\!] & = & \langle x_\mathsf{c}+y_\mathsf{c}, x_\mathsf{s}+y_\mathsf{s}+1, x_\star+y_\star\rangle \\
[\![1]\!] & = & \langle 0,0,0\rangle & [\![\mathsf{times}(x,y)]\!] & = & \langle x_\mathsf{c}+y_\mathsf{c}, x_\mathsf{s}+y_\mathsf{s}+1, x_\star+y_\star+1\rangle \\
[\![\mathsf{cos}(x)]\!] & = & x & [\![\mathsf{ln}(x)]\!] & = & \langle x_\mathsf{c}, x_\mathsf{s}+1, x_\star+1\rangle \\
[\![\mathsf{sin}(x)]\!] & = & x & [\![\mathsf{der}(F)]\!] & = & z \mapsto \langle \\
[\![\mathsf{min}(x)]\!] & = & \langle x_\mathsf{c},0,0\rangle & & & \quad 1+F(z)_\mathsf{c}+2*F(z)_\mathsf{s}+F(z)_\star*F(z)_\mathsf{c}, \\
[\![\mathsf{div}(x,y)]\!] & = & \langle x_\mathsf{c}+y_\mathsf{c},0,0\rangle & & & \quad F(z)_\mathsf{s}*(F(z)_\star+1), \\
 & & & & & \quad F(z)_\star*(F(z)_\star+1)\ \rangle
\end{array}$$

It is easy to see that monotonicity requirements are satisfied. In addition, all the rules are oriented by this interpretation. We only show the one for times.

- $[\![\mathsf{der}(\lambda x.\mathsf{times}(F \cdot x, G \cdot x))]\!] \sqsupset_{\mathsf{real}} [\![\lambda z.\mathsf{plus}(\mathsf{times}(\mathsf{der}(F) \cdot z, G \cdot z), \mathsf{times}(F \cdot z, \mathsf{der}(G) \cdot z))]\!]$
  - $[\![\lambda x.\mathsf{times}(F \cdot x, G \cdot x))]\!] = x \mapsto \langle 1{+}F(x)_\mathsf{c}{+}G(x)_\mathsf{c}, F(x)_\mathsf{s}{+}G(x)_\mathsf{s}{+}1, F(x)_\star{+}G(x)_\star{+}1\rangle$
  - $[\![\mathsf{times}(\mathsf{der}(F) \cdot z, G \cdot z)]\!] = \langle 1 + F(z)_\mathsf{c} + 2 * F(z)_\mathsf{s} + F(z)_\star * F(z)_\mathsf{c} + G(z)_\mathsf{c}, F(z)_\mathsf{s} * (F(z)_\star + 1) + G(z)_\mathsf{s} + 1, F(z)_\star * (F(z)_\star + 1) + G(z)_\star + 1\rangle$
  - $[\![\mathsf{times}(F \cdot z, \mathsf{der}(G) \cdot z))]\!] = \langle 1 + G(z)_\mathsf{c} + 2 * G(z)_\mathsf{s} + G(z)_\star * G(z)_\mathsf{c} + F(z)_\mathsf{c}, G(z)_\mathsf{s} * (G(z)_\star + 1) + F(z)_\mathsf{s} + 1, G(z)_\star * (G(z)_\star + 1) + F(z)_\star + 1\rangle$

  $[\![\mathsf{der}(\lambda x.\mathsf{times}(F \cdot x, G \cdot x))]\!] = z \mapsto \langle 1 + \mathrm{cost}, \mathrm{size}, \mathrm{star}\rangle$, where:
  - $\mathrm{cost} = (1{+}F(z)_\mathsf{c}{+}G(z)_\mathsf{c}){+}2{*}(F(z)_\mathsf{s}{+}G(z)_\mathsf{s}{+}1){+}(F(z)_\star{+}G(z)_\star{+}1){*}(1{+}F(z)_\mathsf{c}{+}G(z)_\mathsf{c})$;
  - $\mathrm{size} = (F(z)_\mathsf{s} + G(z)_\mathsf{s} + 1) * (F(z)_\star + G(z)_\star + 2)$;
  - $\mathrm{star} = (F(z)_\star + G(z)_\star + 1) * (F(z)_\star + G(z)_\star + 2)$.

  We have $\mathrm{size} = F(z)_\mathsf{s} + G(z)_\mathsf{s} + 1 + (F(z)_\mathsf{s} + G(z)_\mathsf{s} + 1) * (F(z)_\star + G(z)_\star + 1) \geq F(z)_\mathsf{s}{+}G(z)_\mathsf{s}{+}1{+}F(z)_\mathsf{s}{*}(F(z)_\star{+}1){+}G(z)_\mathsf{s}{*}(G(z)_\star{+}1){+}1{*}1 = (F(z)_\mathsf{s}{*}(F(z)_\star{+}1){+}G(z)_\mathsf{s}{+}1){+}(G(z)_\mathsf{s}{*}(G(z)_\star{+}1){+}F(z)_\mathsf{s}{+}1) = [\![\mathsf{plus}(\mathsf{times}(\mathsf{der}(F) \cdot z, G \cdot z), \mathsf{times}(F \cdot z, \mathsf{der}(G) \cdot z))]\!]_\mathsf{s}$
  The proof that $\mathrm{star} \geq [\![\mathsf{plus}(\mathsf{times}(\mathsf{der}(F) \cdot z, G \cdot z), \mathsf{times}(F \cdot z, \mathsf{der}(G) \cdot z))]\!]_\star$ is the same, just with $\cdot_\mathsf{s}$ replaced by $\cdot_\star$.
  Finally, $\mathrm{cost} > F(z)_\mathsf{c} + G(z)_\mathsf{c} + 2 * F(z)_\mathsf{s} + 2 * G(z)_\mathsf{s} + 2 + 1 + F(z)_\mathsf{c} + G(z)_\mathsf{c} + (F(z)_\star + G(z)_\star) * (F(z)_\mathsf{c} + G(z)_\mathsf{c}) = 1 + 1 + F(z)_\mathsf{c} + 2 * F(z)_\mathsf{s} + F(z)_\star * F(z)_\mathsf{c} + G(z)_\mathsf{c} + 1 + G(z)_\mathsf{c} + 2 * G(z)_\mathsf{s} + G(z)_\star * G(z)_\mathsf{c} + F(z)_\mathsf{c} = 1 + [\![\mathsf{plus}(\mathsf{times}(\mathsf{der}(F) \cdot z, G \cdot z), \mathsf{times}(F \cdot z, \mathsf{der}(G) \cdot z))]\!]_\mathsf{c}$

B Proof sketches and unstated lemmas

We here present proof sketches for lemmas in the text where they were omitted, as well as unstated lemmas that for instance support the correctness of our definition. Complete proofs can be found in the extended appendix [32].

Proof Sketch of Lemma 14. Each individual statement follows by induction on σ. ◀

In the text, we quietly asserted that Definition 16 is well-defined. This follows from:

▶ **Lemma B.1.** *For all terms $s :: \sigma$ and suitable α as described in Definition 16 we have: $[\![s]\!]_\alpha \in \mathcal{M}_s$, and for all variables x occurring in the domain of α: $d \mapsto [\![s]\!]_{[x:=d]}$ is either a strongly monotonic function, or a constant function.*

Proof Sketch. By induction on the form of s. The second part of the induction hypothesis is used to prove that $[\![\lambda x.s]\!] \in \mathcal{M}$, as *MakeSM* must be applied on either a strongly monotonic or a constant function. ◀

To prove Theorem 17 we need an AFS version of the so called *Substitution Lemma*. We begin by giving a systematic way of extending a substitution (seen as a morphism between terms) to a valuation, seen as morphism from terms to elements of $\mathcal{A}_\mathcal{M}$.

▶ **Definition B.2.** *Given a substitution $\gamma = [x_1 := s_1, \ldots, x_n := s_n]$ and a valuation α, we define α^γ as the valuation such that $\alpha^\gamma(x) = \alpha(x)$, if $x \notin \mathtt{dom}(\gamma)$; and $\alpha^\gamma(x) = [\![x\gamma]\!]_\alpha$, otherwise.*

▶ **Lemma B.3** (Substitution Lemma)**.** *For any substitution γ and valuation α, $[\![s\gamma]\!]_\alpha = [\![s]\!]_{\alpha^\gamma}$. Additionally, if $[\![s]\!] \sqsupset_\sigma [\![t]\!]$ ($[\![s]\!] \sqsupseteq_\sigma [\![t]\!]$), then $[\![s\gamma]\!] \sqsupset_\sigma [\![t\gamma]\!]$ ($[\![s\gamma]\!] \sqsupseteq_\sigma [\![t\gamma]\!]$).*

Proof.
By inspection of Definition B.2 it can be easily shown by induction on s that the diagram to the right commutes. As a consequence, if $[\![s]\!]_\alpha \sqsupset_\sigma [\![t]\!]$ for any valuation α, then $[\![s]\!]_{\alpha^\gamma} \sqsupset_\sigma [\![t]\!]_{\alpha^\gamma}$ in particular. So $[\![s\gamma]\!]_\alpha \sqsupset_\sigma [\![t]\!]_\alpha$.
The case for $\sqsupseteq_\sigma$ is analogous. ◀

$$\begin{tikzcd}
 & T(\mathcal{F},\mathcal{X}) \arrow[d, "{[\![\cdot]\!]_\alpha}"] \\
T(\mathcal{F},\mathcal{X}) \arrow[ur, "\gamma"] \arrow[r, "{[\![\cdot]\!]_{\alpha^\gamma}}"'] & \mathcal{A}_\mathcal{M}
\end{tikzcd}$$

Proof Sketch of Theorem 17. This follows easily by induction on the definition of $s \to_\mathcal{R} t$, using the substitution lemma. ◀

We posit some results regarding the functions $\mathtt{0}_\sigma$, $\mathtt{addc}_\sigma$ and $\mathtt{costof}_\sigma$.

▶ **Lemma B.4.** *For all types σ: (1) $\mathtt{0}_\sigma \in \mathcal{M}_\sigma$; (2) for all $n \in \mathbb{N}$ and $x \in \mathcal{M}_\sigma$: $\mathtt{addc}_\sigma(n, x) \in \mathcal{M}_\sigma$; (3) $\mathtt{costof}_\sigma$ is weakly monotonic and strict in its first argument; (4) $\mathtt{addc}_\sigma$ is weakly monotonic and strict in both its arguments.*

Proof Sketch. All claims follow easily by a mutual induction on σ. ◀

▶ **Lemma B.5.** *For all types σ, for all $x \in \mathcal{M}_\sigma$: (1) $\mathtt{addc}_\sigma(0, x) = x$; (2) for all $n, m \in \mathbb{N}$: $\mathtt{addc}_\sigma(n, \mathtt{addc}_\sigma(m, x)) = \mathtt{addc}_\sigma(n+m, x)$; (3) if $n > 0$ then $\mathtt{addc}_\sigma(n, x) \sqsupset_\sigma x$; (4) if $y \in \mathcal{M}_\sigma$ is such that $x \sqsupset_\sigma y$ then $x \sqsupseteq_\sigma \mathtt{addc}_(1, y)$; (5) for all $n \in \mathbb{N}$: $\mathtt{costof}_\sigma(\mathtt{addc}_\sigma(n, x)) = n + \mathtt{costof}_\sigma(x)$.*

Proof Sketch. All claims follow easily by induction on σ. ◀

▶ **Lemma B.6.** *For all σ, τ, $F \in \mathcal{M}_{\sigma\Rightarrow\tau}$, $x \in \mathcal{M}_\sigma$, $n \in \mathbb{N}$: $F(\mathtt{addc}_\sigma(n, x)) \sqsupseteq_\sigma \mathtt{addc}_\tau(n, F(x))$.*

Proof Sketch. By induction on n, using the various claims in Lemma B.5. ◀

▶ **Lemma B.7.** *For all types σ and all $x \in \mathcal{M}_\sigma$: $x \sqsupseteq_\sigma \mathtt{addc}_\sigma(\mathtt{costof}_\sigma(x), \mathtt{0}_\sigma)$.*

Proof Sketch. By induction on σ, using Lemmas B.4–B.6. ◀

▶ **Lemma B.8.** *For $F \in \mathcal{M}_{\sigma\Rightarrow\tau}$ and $x \in \mathcal{M}_\sigma$ we have: $\mathtt{costof}_\tau(F(x)) \geq \mathtt{costof}_\sigma(x)$.*

Proof. Let $n := \mathtt{costof}_\sigma(x)$. By Lemma B.7, $x \sqsupseteq_\sigma \mathtt{addc}_\sigma(\mathtt{costof}_\sigma(x), \mathtt{0}_\sigma) = \mathtt{addc}_\sigma(n, \mathtt{0}_\sigma)$. Hence, by monotonicity of F, $F(x) \sqsupseteq_\tau F(\mathtt{addc}_\sigma(n, \mathtt{0}_\sigma))$. By Lemma B.6, this implies that $F(x) \sqsupseteq_\tau \mathtt{addc}_\tau(n, F(\mathtt{0}_\sigma))$. Since $\mathtt{costof}_\tau$ is strict in its first argument by Lemma B.4(3), we thus have $\mathtt{costof}_\tau(F(x)) \geq \mathtt{costof}_\sigma(\mathtt{addc}_\tau(n, F(\mathtt{0}_\sigma)))$, which $\geq n$ by Lemma B.5(5). ◀

We can now prove that Definition 20 indeed defines a (σ, τ)-monotonicity function.

▶ **Lemma B.9.** *Let σ, τ be simple types. Then $\Phi_{\sigma,\tau}$ is a (σ, τ)-monotonicity function.*

Proof Sketch. By case analysis and Lemma B.4 we see that $\Phi_{\sigma,\tau}$ maps $C_{\sigma,\tau}$ to $\mathcal{M}_{\sigma,\tau}$. To see that $\Phi_{\sigma,\tau}$ is strongly monotonic we also use a case analysis. If F and G are both constant functions or both strongly monotonic, the result follows easily; F is constant and G not cannot occur because eventually $\mathtt{costof}(G)(x) > \mathtt{costof}(F)(x)$; and if F is strongly monotonic and G is constant then $F(x) \sqsupset \mathtt{addc}(\mathtt{costof}(x), G(x))$ because $G(x) = G(\mathtt{0})$ and $F(x) \sqsupseteq F(\mathtt{addc}(\mathtt{costof}(x), \mathtt{0})) \sqsupseteq \mathtt{addc}(\mathtt{costof}(x), F(\mathtt{0}))$ by Lemmas B.4–B.8. ◀

Proof of Lemma 21. We have either $[\![(\lambda x.s) \cdot t]\!]_\alpha = \mathtt{addc}_\tau(\mathtt{costof}_\sigma([\![t]\!]_\alpha) + 1, [\![s]\!]_{\alpha[x:=[\![t]\!]]})$ or $[\![(\lambda x.s) \cdot t]\!]_\alpha = \mathtt{addc}_\tau(1, [\![s]\!]_{\alpha[x:=[\![t]\!]]})$. By Lemma B.5(3) we have $[\![(\lambda x.s) \cdot t]\!]_\alpha \sqsupset_\tau [\![s]\!]_{\alpha[x:=[\![t]\!]]}$ in both cases. By Lemma B.3, $[\![s]\!]_{\alpha[x:=[\![t]\!]]} = [\![s[x := b]]\!]_\alpha$. This completes the proof. ◀

Proof of Lemma 22. Since $F \neq x$, we have $d \mapsto [\![F \cdot x]\!]_{\alpha[x:=d]} = d \mapsto \alpha(F)(d)$, which by extensionality is $\alpha(F)$. Since $\alpha(F)$ is monotonic we have $[\![\lambda x.F\,x]\!]_\alpha = \Phi_{\sigma,\tau}(d \mapsto [\![F \cdot x]\!]_{\alpha x:=d}) = \Phi_{\sigma,\tau}(\alpha(F)) = \mathtt{addc}_{\sigma,\tau}(1, \alpha(F))$. By Lemma B.5(3) this $\sqsupset_{\sigma\Rightarrow\tau} \alpha(F) = [\![F]\!]$. ◀

Proof Sketch of Lemma 25. Let $Q(x_1, \dots, x_k) := y \mapsto F(x_1, \dots, x_k)^{G(x_1,\dots,x_k)}(y)$. To see that Q maps to $\mathcal{M}_{\tau\Rightarrow\tau}$, so that Q is strongly monotonic. We show that $F(\vec{u})^n(x) \sqsupset F(\vec{u})^n(y)$ whenever $x \sqsupset y$ by a straighforward induction on n, and similar for $x \sqsupseteq y$. To see that Q is weakly monotonic in its first k arguments, we show by induction on n that for all n, m with $n \geq m$ we have $F(u_1, \dots, u_k)^n \sqsupseteq_{\tau\Rightarrow\tau} F(u'_1, \dots, u'_k)^m$ if each $u_i \sqsupseteq u'_i$. The result then follows because $G(u_1, \dots, u_k) \geq G(u'_1, \dots, u'_k)$ by weak monotonicity of G. ◀

Proof Sketch of Lemma 32. For claim (1), let b be the largest of the constants used for each constructor; i.e., we have $\sum_{l=1}^{K[\kappa]} P_l(x^1, \dots, x^m) \leq b + \sum_{i=1}^m \sum_{j=1}^{K[\iota_i]} x^i_j$ whenever $\mathcal{J}_\mathsf{c}(\vec{x}) = \langle P_1(\vec{x}), \dots, P_{K_\kappa}(\vec{x})\rangle$. We prove by induction on the size of a data term $s :: \kappa$ that $\sum_{l=1}^{K[\kappa]} [\![s]\!]_l \leq a * |s|$. Then certainly $[\![s]\!]_l \leq b * |s|$ holds for any component $[\![s]\!]_l$.

For claim (2), let a be the largest of the constants used for each constructor c and component P_l, and let k be the largest value $K[\iota]$ for any sort in the program; let $b := \max(2, a * k)$. We prove by induction on the size of a data term s that $[\![s]\!]_l \leq 2^{b*|s|}$, In the proof, we use that $n + m \leq n * m$ whenever $n, m \geq 2$ and $2 * n \leq 2^n$ if $n \geq 2$, and hence: $2 * a * k * \sum_{i=1}^m 2^{a*k*|s_i|} \leq (2 * a * k) * \prod_{i=1}^m 2^{a*k*|s_i|} \leq 2^b * 2^{b*\sum_{i=1}^m |s_i|}$. ◀

Output Without Delay: A π-Calculus Compatible with Categorical Semantics

Ken Sakayori
The University of Tokyo, Japan

Takeshi Tsukada
Chiba University, Japan

Abstract

The quest for logical or categorical foundations of the π-calculus (not limited to session-typed variants) remains an important challenge. A categorical type theory correspondence for a variant of the **i**/**o**-typed π-calculus was recently revealed by Sakayori and Tsukada, but, at the same time, they exposed that this categorical semantics contradicts with most of the behavioural equivalences. This paper diagnoses the nature of this problem and attempts to fill the gap between categorical and operational semantics. We first identify the source of the problem to be the mismatch between the operational and categorical interpretation of a process called the forwarder. From the operational viewpoint, a forwarder may add an arbitrary delay when forwarding a message, whereas, from the categorical viewpoint, a forwarder must not add any delay when forwarding a message. Led by this observation, we introduce a calculus that can express forwarders that do not introduce delay. More specifically, the calculus we introduce is a variant of the π-calculus with a new operational semantics in which output actions are forced to happen as soon as they get unguarded. We show that this calculus (i) is compatible with the categorical semantics and (ii) can encode the standard π-calculus.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Process calculi; Theory of computation → Denotational semantics

Keywords and phrases π-calculus, categorical semantics, linear approximation

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.32

Funding This work was supported by JSPS KAKENHI Grant Numbers JP20J13473 and JP19K20211.

Acknowledgements We would like to thank anonymous referees for useful comments.

1 Introduction

The connection between the π-calculus and logic or categorical type theory has been studied since the early stages of the development of the π-calculus [1, 3, 2]. Among others, a close correspondence between a session typed π-calculus and intuitionistic linear logic [6] (and hence also the relationship to categorical models of linear logic) is well-understood. The session-typed calculi corresponding linear logic, however, are not quite expressive since they are race-free and deadlock-free. So it is natural to question whether a similar categorical foundation can be given to processes not limited to deadlock-free and race-free processes.

A fundamental difficulty in developing a categorical type theory for process calculi in the presence of race condition has been recently pointed out by Sakayori and Tsukada [19]. They showed that asynchronous π-calculus processes modulo observational equivalence (weak barbed congruence) do not form a category, under some mild assumptions [19, Theorem 1].[1] Hence, the observational equivalence cannot be an instance of an equational theory characterised by a certain categorical structure; this is in contrast to the case of λ-calculus, where observational equivalence is a $\beta\eta$-theory.

[1] The choice of the behavioural equivalence does not matter since their argument also applies to many other behavioural equivalences, such as must-testing equivalence.

Hence, if a process calculus based on the (asynchronous) π-calculus were to have some categorical foundation, its operational behaviour must be distant from conventional behaviour.

This paper introduces a variant of the π-calculus whose observational equivalence harmonises with categorical semantics. We introduce a novel reduction semantics to the π-calculus and show that processes modulo weak barbed congruence, defined on top of the new reduction semantics, form a category; more precisely, they form a compact closed Freyd category [19] (described below).

Before introducing the operational semantics that we propose, let us explain the problem of conventional behavioural equivalences in a little more detail.

The problem is about the behaviour of a special process $!a(x).\bar{b}\langle x\rangle$, which is often called a *forwarder* or a *link*. Intuitively this process transfers a message from channel a to channel $\bar{b}$. This intuition justifies the following equation

$$(\boldsymbol{\nu} a)(P \mid\, !a(x).\bar{b}\langle x\rangle) \quad = \quad P\{\bar{b}/\bar{a}\}, \qquad a, \bar{b} \notin \mathbf{fn}(P), \tag{1}$$

which indeed holds for the weak barbed congruence in an asynchronous setting. If we adopt "parallel composition + hiding" as the notion of composition, i.e. if we regard $(\boldsymbol{\nu} a)(P \mid\, !a(x).\bar{b}\langle x\rangle)$ as a composition of $!a(x).\bar{b}\langle x\rangle$ and P, (1) says that the forwarder is a right-identity. If processes modulo weak barbed congruence formed a category, a right-identity would be the identity and in particular a left-identity, as in any other categories. The left-identity law is

$$(\boldsymbol{\nu} b)(!a(x).\bar{b}\langle x\rangle \mid P) \quad = \quad P\{a/b\}, \qquad a, \bar{b} \notin \mathbf{fn}(P), \tag{2}$$

but this is invalid with respect to weak barbed congruence.

To see why (2) fails for weak barbed congruence, let us review the conventional behavioural interpretation of a forwarder $!a(x).\bar{b}\langle x\rangle$: it receives a message from a, *possibly waits as long as it wants or needs*, and then sends the message to the receiver. Hence the process $(\boldsymbol{\nu} b)(!a(x).\bar{b}\langle x\rangle \mid P)$ can immediately receive a message from a and keep it until P actually requires a message from b. On the other hand, $P\{a/b\}$ do not receive a message from a unless $P\{a/b\}$ actually requires it. This difference is significant in the presence of race condition, and thus (2) fails for weak barbed congruence.

A similar observation on a problem caused by delays introduced by forwarders and a solution against that problem has been made in the context of game semantics. When giving a game semantics of a synchronous session typed π-calculus, Castellan and Yoshida [7] observed that the (traditional) copycat strategy – the game semantic counterpart of the forwarder process – does not behave as identity due to the delay it introduces. To avoid this problem, they introduced a copycat strategy that does not introduce any delay and proved that this "delayless copycat strategy" works as the identity.

Whereas [7] added delayless forwarders as semantic elements that processes cannot represent, this paper discusses a new operational semantics on processes with respect to which forwarders are delayless. The main result shows that behavioural equivalences under the delayless interpretation is in harmony with categorical semantics.

The new operational semantics introduced in this paper is a reduction semantics that forces output actions to happen as soon as they get unguarded. Under the new operational semantics, when a forwarder $!a(x).\bar{b}\langle x\rangle$ receives a message m from a, it *must immediately send* m to a receiver b. In other words, the following two transitions are atomic

$$!a(x).\bar{b}\langle x\rangle \quad \xrightarrow{a(m)} \quad \underline{!a(x).\bar{b}\langle x\rangle \mid \bar{b}\langle m\rangle} \quad \xrightarrow{\bar{b}\langle m\rangle} \quad !a(x).\bar{b}\langle x\rangle,$$

and the process cannot stop at the underlined intermediate step since it has an unguarded output action. So one-step reduction in our calculus corresponds to multi-step reduction in the conventional calculus. We may consider that the new behaviour expresses a synchronous communication since a message m now cannot be kept in a communication medium $\bar{a}\langle m\rangle$.

In our proposed calculus, processes modulo observational equivalence form a compact closed Freyd category. This means that not only equations (1) and (2) but also some equational laws studied for the asynchronous π-calculus are valid under this new operational behaviour. This is because compact closed Freyd category is a categorical structure that corresponds to a theory of processes, i.e. a congruence over asynchronous π-processes satisfying certain equational laws [19]. One of the laws is (2), and the others are laws that frequently appear in the study of asynchronous π-calculus, such as the replication theorem [17].

We also show that a π-calculus with the standard reduction semantics, can be embedded into the proposed calculus by using a special constant τ for delay. The translation replaces each output action $\bar{a}\langle m\rangle$ with $\tau.\bar{a}\langle m\rangle$, making explicit the delay of the output action in the conventional π-calculus. For instance, the conventional behaviour of a forwarder $!a(x).\bar{b}\langle x\rangle \xrightarrow{a(m)} !a(x).\bar{b}\langle x\rangle \mid \bar{b}\langle m\rangle$ is mimicked by $!a(x).\tau.\bar{b}\langle x\rangle \xrightarrow{a(m)} !a(x).\bar{b}\langle x\rangle \mid \tau.\bar{b}\langle m\rangle$ in the new operational semantics.

Technically the new operational semantics is quite complicated since its one-step reduction is a multi-step reduction with a certain condition in the conventional calculus. To overcome the difficulty in reasoning about such a complicated calculus, we develop an intersection type system, or equivalently a system of linear approximations [21, 14], that captures the behaviour of a process. We think that the system would be of independent technical interest.

Organisation of the paper

Section 2 introduces our calculus and states the main result; the following sections are devoted to its proof. After reviewing the idea of linear approximations and its correspondence to reduction sequences in Section 3, we formalise this idea in Section 4. Section 5 defines an LTS based on linear approximations, and Section 6 shows that barbed congruence has a categorical model. Section 7 discusses related work and Section 8 concludes the paper.

2 A process calculus with undelayed output

This section (i) introduces a variant of the π-calculus whose barbed congruence can be captured categorically and (ii) claims the main result of this paper. The syntax of the calculus is the same as that of the π_F-calculus introduced by Sakayori and Tsukada [19], but the calculus is equipped with a non-standard reduction semantics; *we also call this calculus the π_F-calculus.*[2] The proof of the main result will be given in the following sections.

2.1 Syntax

The π_F-calculus is a variant of the polyadic asynchronous π-calculus with **i**/**o**-types,[3] which this paper calls *sorts* in order to avoid confusion with intersection types introduced later.

▶ **Definition 1** (Sorts). *The set of* sorts*, ranged over by S and T, is given by*

$$S, T ::= \mathbf{ch}^o[T_1, \dots, T_n] \mid \mathbf{ch}^i[T_1, \dots, T_n] \qquad (n \geq 0).$$

The sort $\mathbf{ch}^o[T_1, \dots, T_n]$ (resp. $\mathbf{ch}^i[T_1, \dots, T_n]$) is for channels for sending (resp. receiving) n arguments of types $T_1, \dots, T_n$. We often write $\vec{T}$ for a sequence of sorts $T_1, \dots, T_n$. The dual *$T^\perp$ of sort T is defined by $\mathbf{ch}^o[\vec{T}]^\perp \stackrel{\text{def}}{=} \mathbf{ch}^i[\vec{T}]$ and $\mathbf{ch}^i[\vec{T}]^\perp \stackrel{\text{def}}{=} \mathbf{ch}^o[\vec{T}]$.*

---

[2] Although the π_F-calculus introduced in this paper and the original π_F-calculus [19] have different reduction semantics it is not that odd to call them with the same name. This is because the reduction semantics is not essential to establish the correspondence to compact closed Freyd categories; we only need the "algebraic semantics" to establish the correspondence (cf. Appendix A).

[3] Unlike the original **i**/**o**-types [17], no names have both input and output capabilities. Names are used to represent the input/output endpoints of a channel.

$$\frac{\Delta \vdash P \qquad \Delta \vdash Q}{\Delta \vdash P \mid Q} \qquad \frac{\Delta, x : T, y : T^{\perp} \vdash P}{\Delta \vdash (\boldsymbol{\nu}_T xy)P} \qquad \frac{(x : \mathbf{ch}^i[\vec{T}]) \in \Delta \quad \Delta, \vec{y} : \vec{T} \vdash P}{\Delta \vdash !x(\vec{y}).P}$$

$$\frac{(x : \mathbf{ch}^o[\vec{T}]) \in \Delta \quad \vec{y} : \vec{T} \subseteq \Delta}{\Delta \vdash x\langle\vec{y}\rangle} \qquad \frac{(\tau : \mathbf{ch}^i[]) \in \Delta \quad \Delta \vdash P}{\Delta \vdash \tau.P} \qquad \frac{}{\Delta \vdash \mathbf{0}}$$

Figure 1 Sort assignment rules for processes.

▶ **Definition 2** (Processes). *The set of* processes *is defined by*

$$P, Q, R ::= \mathbf{0} \mid (P|Q) \mid (\boldsymbol{\nu}_T xy)P \mid x\langle\vec{y}\rangle \mid !x(\vec{y}).P \mid \tau.P,$$

where x and y range over a set of names and $\vec{y}$ represents a (possibly empty) sequence of names. We often elide sort annotations and write $(\boldsymbol{\nu} xy)$ for $(\boldsymbol{\nu}_T xy)$. The set of free names of P, written $\mathbf{fn}(P)$, and bound names of P written $\mathbf{bn}(P)$ are defined as usual.

All the constructs, except for the name restriction, are standard so their meaning should be clear.[4] The name restriction $(\boldsymbol{\nu}_T xy)P$ hides the names x and y of type T and $T^{\perp}$ and, at the same time, establishes a connection between x and y. The input-output connection is *not a priori* and communications only happen over bound names connected by $\boldsymbol{\nu}$; this is different from the standard π-calculi where $\bar{a}$ is considered as an output to a.

For a technical reason, we introduce not only *structural congruence*, but also a notion called *structural precongruence* $\Rrightarrow$ (cf. Remark 10). A precongruence is like a congruence, but it is just reflexive and transitive, not necessarily symmetric. We define $\Rrightarrow$ as the smallest precongruence relation on processes that satisfies the following rules:

$$\begin{array}{lll} P \mid \mathbf{0} \Lleftarrow\!\!\Rrightarrow P & P \mid Q \Lleftarrow\!\!\Rrightarrow Q \mid P & (P \mid Q) \mid R \Lleftarrow\!\!\Rrightarrow P \mid (Q \mid R) \\ (\boldsymbol{\nu} wx)(\boldsymbol{\nu} yz)P \Lleftarrow\!\!\Rrightarrow (\boldsymbol{\nu} yz)(\boldsymbol{\nu} wx)P & & ((\boldsymbol{\nu} xy)P) \mid Q \Rrightarrow (\boldsymbol{\nu} xy)(P \mid Q) \end{array}$$

where $P \Lleftarrow\!\!\Rrightarrow Q$ means $P \Rrightarrow Q$ and $Q \Rrightarrow P$, w, x, y, z are distinct in the fourth rule and $x, y \notin \mathbf{fn}(Q)$ in the fifth rule. Unlike the structural congruence, the restriction of the scope of $(\boldsymbol{\nu} xy)$ is not allowed. The *structural congruence* $\equiv$ is the symmetric closure of $\Rrightarrow$.

The typing rules are rather straightforward. A *sort environment*, written Δ, is a finite set of bindings of the form $t\colon T$, where t is either a name x or τ, such that the names in Δ are pairwise distinct. The *sort assignment relation* $\Delta \vdash P$ is the least relation closed under the rules listed in Figure 1.

2.2 Reduction semantics

As mentioned in Section 1, a one-step reduction in our calculus corresponds to a multi-step reduction in the conventional calculus. So we first introduce the conventional reduction relation $\to$ and then define a new reduction relation $\Rightarrow$ using the conventional reduction.

The *standard reduction relation* $\xrightarrow{\ell}$ ($\ell = \tau$ or 0) is defined by the base rules

$$\begin{array}{rcl} (\boldsymbol{\nu}\vec{w}\vec{z})(\boldsymbol{\nu}\bar{a}a)(!a(\vec{x}).P \mid \bar{a}\langle\vec{y}\rangle \mid Q) & \xrightarrow{0} & (\boldsymbol{\nu}\vec{w}\vec{z})(\boldsymbol{\nu}\bar{a}a)(!a(\vec{x}).P \mid P\{\vec{y}/\vec{x}\} \mid Q) \\ (\boldsymbol{\nu}\vec{w}\vec{z})(\tau.P \mid Q) & \xrightarrow{\tau} & (\boldsymbol{\nu}\vec{w}\vec{z})(P \mid Q) \end{array}$$

[4] Another notable characteristic of the π_F-calculus is that it does not have non-replicated inputs $a(\vec{x}).P$.

together with the structural rule which concludes $P \xrightarrow{\ell} Q$ from $P \Rrightarrow P' \xrightarrow{\ell} Q' \Rrightarrow Q$ for some P' and Q'. We write $P \to Q$ if the label is not important. The following is an example of a (multi-step) reduction:

$$\begin{aligned}
& (\boldsymbol{\nu}\bar{a}a)(\boldsymbol{\nu}\bar{b}b)(\tau.\bar{a}\langle m\rangle \mid !a(x).\bar{b}\langle x\rangle \mid !b(y).!c(z).P) \\
\xrightarrow{\tau}\ & (\boldsymbol{\nu}\bar{a}a)(\boldsymbol{\nu}\bar{b}b)(\bar{a}\langle m\rangle \mid !a(x).\bar{b}\langle x\rangle \mid !b(y).!c(z).P) \\
\xrightarrow{0}\ & (\boldsymbol{\nu}\bar{a}a)(\boldsymbol{\nu}\bar{b}b)(!a(x).\bar{b}\langle x\rangle \mid \bar{b}\langle m\rangle \mid !b(y).!c(z).P) \\
\xrightarrow{0}\ & (\boldsymbol{\nu}\bar{a}a)(\boldsymbol{\nu}\bar{b}b)(!a(x).\bar{b}\langle x\rangle \mid !c(z).P\{m/y\} \mid !b(y).!c(z).P).
\end{aligned}$$

In our calculus, the output action $\bar{b}\langle x\rangle$ in $!a(x).\bar{b}\langle x\rangle$ (resp. $\bar{a}\langle m\rangle$ in $\tau.\bar{a}\langle m\rangle$) must be performed at the same time as the input action $a(x)$ (resp. τ). Therefore, the above multi-step reduction should be regarded as a one-step reduction:

$$\begin{aligned}
& (\boldsymbol{\nu}\bar{a}a)(\boldsymbol{\nu}\bar{b}b)(\tau.\bar{a}\langle m\rangle \mid !a(x).\bar{b}\langle x\rangle \mid !b(y).!c(z).P) \\
& \Rightarrow (\boldsymbol{\nu}\bar{a}a)(\boldsymbol{\nu}\bar{b}b)(!a(x).\bar{b}\langle x\rangle \mid !c(z).P\{m/y\} \mid !b(y).!c(z).P).
\end{aligned}$$

We formally define $\Rightarrow$. A process P has an *unguarded output action* if $P \equiv (\boldsymbol{\nu}\vec{w}\vec{z})(\bar{a}\langle\vec{x}\rangle \mid Q)$ for some Q. A process with an unguarded output action is regarded as an incomplete, intermediate state that needs to perform further actions to complete an "atomic operation". We say that P is *settled* if P has no unguarded output action. We write $P \Rightarrow Q$ if $P \xrightarrow{\tau}(\xrightarrow{0})^* Q$ and Q is settled.

The notion of *barbed congruence* can be easily adapted to this setting.

▶ **Definition 3** (Barbed bisimulation and barbed congruence). *Let $\mathcal{R}$ be a binary relation on settled processes. We say that $\mathcal{R}$ is a* barbed bisimulation *if whenever $P \mathrel{\mathcal{R}} Q$,*

1. *$P{\downarrow}^{\tau}_{\bar{a}}$ if and only if $Q{\downarrow}^{\tau}_{\bar{a}}$*
2. *$P \Rightarrow P'$ implies $Q \Rightarrow Q'$ and $P' \mathrel{\mathcal{R}} Q'$ for some process Q'*
3. *$Q \Rightarrow Q'$ implies $P \Rightarrow P'$ and $P' \mathrel{\mathcal{R}} Q'$ for some process P',*

where $P{\downarrow}^{\tau}_{\bar{a}}$ means that $P \xrightarrow{\tau} (\xrightarrow{0})^* \equiv (\boldsymbol{\nu}\vec{x}\vec{y})(\bar{a}\langle\vec{z}\rangle \mid P')$ and $\bar{a}$ is a free name of P.

The barbed bisimilarity *$\stackrel{\bullet}{\sim}_\tau$ is the largest barbed bisimulation. Processes P and Q are* barbed congruent*, written $P \simeq^c_\tau Q$, if $\tau.C[P] \stackrel{\bullet}{\sim} \tau.C[Q]$ for all context C. (The additional τ-prefixing is to ensure that the processes are settled.)*

The main result of this paper is that there exists a categorical model that is fully abstract with respect to $\simeq^c_\tau$. We use the categorical structure named *compact closed Freyd category* [19] to interpret π_F-calculus processes. The proof is given in the subsequent sections.

▶ **Theorem 4.** *π_F-processes modulo $\simeq^c_\tau$ forms a compact closed Freyd category. Hence there exists a compact closed Freyd category that is fully abstract with respect to $\simeq^c_\tau$.*

The proof can be easily adapted to prove a similar claim for any other congruence that subsumes $\simeq^c_\tau$, such as weak barbed congruence (for $\Rightarrow$).

2.3 Relationship to the standard semantics

We have introduced two reduction relations to the π_F-calculus, namely $\to$ and $\Rightarrow$. There exists an embedding of the π_F-calculus with $\to$ to that with $\Rightarrow$.

The translation is quite simple: it replaces each output action $\bar{a}\langle\vec{x}\rangle$ with $\tau.\bar{a}\langle\vec{x}\rangle$, reflecting the fact that an output action in the standard semantics can be delayed. Let us write $(-)^\dagger$ for this translation. It preserves the semantics in the following sense.

▶ **Proposition 5.** *Suppose* $\Delta \vdash P$. *Then (i)* $P \to Q$ *implies* $(P)^\dagger \Rightarrow (Q)^\dagger$, *(ii)* $(P)^\dagger \Rightarrow Q'$ *implies* $Q' = (Q)^\dagger$ *and* $P \to Q$ *for some* Q, *and (iii)* $P{\downarrow_{\bar a}}$ *iff* $(P)^\dagger{\downarrow^\tau_{\bar a}}$.

From this proposition and the compositionality of $(-)^\dagger$, we obtain the following result. Let $\simeq^c$ for the conventional (strong) barbed congruence for π_F-processes, defined by replacing $\Rightarrow$ with $\longrightarrow$ and $\downarrow^\tau_{\bar a}$ with $\downarrow_{\bar a}$ (i.e. existence of a free unguarded output $\bar a$) in Definition 3.

▶ **Theorem 6.** *If* $\Delta \vdash P$, $\Delta \vdash Q$ *and* $(P)^\dagger \simeq^c_\tau (Q)^\dagger$, *then* $P \simeq^c Q$.

This translation, however, is *not* fully abstract with respect to barbed congruence. Contexts that are not in the image of the translation $(-)^\dagger$ give additional observational power.

3 Overview

To prove Theorem 4, we appeal to an axiomatic characterisation of compact closed Freyd category, proved in [19]: π_F-processes modulo an equivalence relation $\mathcal{R}$ forms a compact closed Freyd category if and only if $\mathcal{R}$ is a congruence satisfying six axioms[5], such as (2) and

$$(\boldsymbol{\nu}\bar a a)(!a(\vec x).P \mid C[\bar a\langle \vec y\rangle]) = (\boldsymbol{\nu}\bar a a)(!a(\vec x).P \mid C[P\{\vec y/\vec x\}]),\ a \notin \mathbf{fn}(P, C),\ \bar a \notin \mathbf{bn}(C), \qquad (3)$$

where C is a context. Since barbed congruence is a congruence by definition, it suffices to check that barbed congruence satisfies the axioms.

However, checking the required axioms directly using the definition of $\Rightarrow$ in Section 2 does not seem tractable. Recall that $P \Rightarrow Q$ is indeed a reduction sequence $P \xrightarrow{\tau} P_1 \xrightarrow{0} \dots \xrightarrow{0} P_n \xrightarrow{0} Q$. The problem is that $P_i \xrightarrow{0} P_{i+1}$ is defined in terms of the structure of P_i, which may be quite different from that of P. A representation of reduction sequence defined by structural induction on P, without directly referring to P_i, would be desirable.

We thus utilise the correspondence of (i) reduction sequences, (ii) derivations in an intersection type system, and (iii) linear approximations [21, 14].

An example of a linear approximation is $(a_1.\tau_1.\bar a_2 \parallel a_2.\bot) \sqsubset\, !a.\tau.\bar a$ where the green part is the linear approximation of the right-hand side. A linear approximation is *linear* in the sense that each name is used exactly once and all inputs are non-replicated; it is an *approximation* in the sense that some part is discarded (e.g. $\bot \sqsubset \tau.\bar a$ or $\bot \sqsubset !a.\tau.\bar a$) and replicated inputs are replaced by a finite number of its copies (e.g. $(a_1.\tau_1.\bar a_2 \parallel a_2.\bot) \sqsubset\, !a.\tau.\bar a$).[6]

To see how a linear approximation corresponds to a reduction sequence, let us consider the following linear approximation:

$$\begin{aligned}&(\boldsymbol{\nu}[\langle \bar a_1, a_1\rangle\langle \bar a_2, a_2\rangle\langle \bar a_3, a_3\rangle])((a_1.\tau_1.(\bar a_2 \mid \bar a_3) \parallel a_2.\bot) \;\mid\; \tau_2.\bar a_1 \;\mid\; a_3.\bot)\\ &\qquad\qquad\qquad\qquad \sqsubset \quad (\boldsymbol{\nu}\bar a a)(!a.\tau.(\bar a \mid \bar a) \;\mid\; \tau.\bar a \;\mid\; !a.\tau.\bar b).\end{aligned}$$

Because of linearity, a linear approximation is race-free; hence it induces an essentially unique reduction sequence. For example,

$$\begin{aligned}&(\boldsymbol{\nu}[\langle \bar a_1, a_1\rangle\langle \bar a_2, a_2\rangle\langle \bar a_3, a_3\rangle])((a_1.\tau_1.(\bar a_2 \mid \bar a_3) \parallel a_2.\bot) \mid \tau_2.\bar a_1 \mid a_3.\bot) \qquad (4)\\ \xrightarrow{\tau_2}\xrightarrow{0}\ & (\boldsymbol{\nu}[\langle \bar a_2, a_2\rangle\langle \bar a_3, a_3\rangle])(a_2.\bot \mid \tau_1.(\bar a_2 \mid \bar a_3) \mid a_3.\bot)\\ \xrightarrow{\tau_1}\xrightarrow{0}\ & (\boldsymbol{\nu}[\langle \bar a_2, a_2\rangle])(a_2.\bot \mid (\bar a_2 \mid \bot) \mid \bot) \quad \xrightarrow{0} \quad (\boldsymbol{\nu}[])(\bot \mid (\bot \mid \bot) \mid \bot).\end{aligned}$$

---

[5] The axioms can be found in the Appendix A; please refer to [19] for a more detailed description.

[6] In the approximation, $\mid$ represents parallel-composition coming from the original process, whereas $p \parallel q$ means that p and q originate from the same replicated (sub)process.

Importantly a reduction sequence of an approximation canonically induces that of the approximated process: the reduction sequence corresponding to (4) is

$$\begin{aligned}(\boldsymbol{\nu}\bar{a}a)(!a.\tau.(\bar{a} \mid \bar{a}) \mid \tau.\bar{a} \mid !a.\tau.\bar{b}) &\xrightarrow{\tau}\xrightarrow{0} (\boldsymbol{\nu}\bar{a}a)(!a.\tau.(\bar{a} \mid \bar{a}) \mid \tau.(\bar{a} \mid \bar{a}) \mid !a.\tau.\bar{b}) \\ &\xrightarrow{\tau}\xrightarrow{0} (\boldsymbol{\nu}\bar{a}a)(!a.\tau.(\bar{a} \mid \bar{a}) \mid (\bar{a} \mid \tau.\bar{b}) \mid !a.\tau.\bar{b}) \\ &\xrightarrow{0} (\boldsymbol{\nu}\bar{a}a)(!a.\tau.(\bar{a} \mid \bar{a}) \mid (\tau.(\bar{a} \mid \bar{a}) \mid \tau.\bar{b}) \mid !a.\tau.\bar{b}).\end{aligned} \tag{5}$$

Via the three-way correspondence mentioned above, this phenomenon can be understood as Subject Reduction of the intersection type system.

Conversely, given a reduction sequence, we can construct a linear approximation that represents the reduction sequence. This is a consequence of Subject Expansion, namely, $p \sqsubset P$ and $Q \to P$ imply $q \sqsubset Q$ and $q \to p$ for some q. The approximation for $P_1 \to P_2 \to \ldots \to P_n$ is obtained by iteratively applying this lemma to $p_n \sqsubset P_n$, where p_n is the approximation that discards everything.

So far, we have discussed a relationship between $\{ Q \mid P \to^* Q \}$ and $\{ p \mid p \sqsubset P \}$. This relation can be seen as a bisimulation, by appropriately introducing a relation to $\{ p \mid p \sqsubset P \}$. Note that such a relation is not the reduction, since $p \to q$ changes the subject, i.e. $q \sqsubset Q$ for some Q with $P \xrightarrow{0} Q$ but not $q \sqsubset P$. Instead, we introduce an "ordering" over approximations of P. The idea is that a longer reduction sequence corresponds to a larger approximation. We write $p_1 \trianglelefteq p_2$ if p_1 is obtained by discarding some (sub)processes of p_2. For example, the second step of (5) corresponds to

$$(\boldsymbol{\nu}[\langle \bar{a}_1, a_1\rangle])((a_1.\bot) \mid \tau_2.\bar{a}_1 \mid \bot) \trianglelefteq (\boldsymbol{\nu}[\langle \bar{a}_1, a_1\rangle\langle \bar{a}_3, a_3\rangle])(a_1.\tau_1.(\bot \mid \bar{a}_3) \ \mid\ \tau_2.\bar{a}_1 \ \mid\ a_3.\bot),$$

representing that the third process in (5) is obtained by performing the actions corresponding to τ_1 and $\bar{a}_3$.

The bisimilarity gives us a characterisation of the behaviour of a process P in terms of linear approximations (or intersection type derivations) for P. Then Theorem 4 can be proved by "proof manipulation". For example, the proof of the above mentioned axiom $(\boldsymbol{\nu}\bar{a}a)(!a(\vec{x}).P \mid C[\bar{a}\langle\vec{y}\rangle]) = (\boldsymbol{\nu}\bar{a}a)(!a(\vec{x}).P \mid C[P\{\vec{y}/\vec{x}\}])$ resembles to the proof of the substitution lemma in a typical type system.

4 Linear approximation and execution sequence

We introduce *linear processes* by which executions of processes can be described.

4.1 Linear processes and intersection types

We start by defining linear processes. Although the definition of linear processes depends on the definition of intersection types because processes are annotated by types, we defer defining types for the sake of presentation.

▶ **Definition 7** (Linear processes). *A* linear name *is an object of the form x_i where x is an ordinary name and i is a natural number. Similarly, a* linear term*, denoted by t, is either a linear name or a constant of the form τ_i.*

Linear processes *are defined by the following grammar:*

$$\begin{aligned} p, q &::= \mathbf{0} \mid x_i\langle\lambda_1, \ldots, \lambda_n\rangle \mid x_i(\mu_1, \ldots, \mu_n).p \mid \tau_i.p \\ &\quad\mid (p \mid q) \mid (p_1 \parallel \cdots \parallel p_n) \mid (\boldsymbol{\nu}[\langle x_{i_1}, y_{i_1}\rangle_{\rho_1}, \ldots, \langle x_{i_n}, y_{i_n}\rangle_{\rho_n}])p \\ \mu &::= \langle x_{i_1}, \ldots, x_{i_n}\rangle \qquad \lambda ::= \langle \varphi_1 \cdot x_{i_1}, \ldots, \varphi_n \cdot x_{i_n}\rangle \end{aligned}$$

Here name restriction is annotated with types ρ_i, and the argument of an output action is annotated with witnesses of type isomorphisms φ_i. (The notion of types and type isomorphisms are introduced below and thus can be ignored for the moment.) In the above definition n may be 0*; for example, $(\boldsymbol{\nu}[])p$ and $\langle\rangle$ are valid process and list, respectively. We require that each linear term of a linear process appears* exactly once.

The informal meanings of the constructs are almost the same as that of the ordinary processes. The linear processes $\mathbf{0}$, $x_i\langle\lambda_1,\dots,\lambda_n\rangle$ and $x_i(\mu_1,\dots,\mu_n).p$ are nil process, output action and input prefixing, respectively. An important difference from the ordinary process is that, in linear processes, the output and input take lists of variables as arguments. When a list of linear names is received each element of a list must be used exactly once. There are two types of parallel composition $p \mid q$ and $p \parallel q$. The former is the conventional parallel composition and the latter is used when a replicated process is approximated by finite parallel compositions.[7] We use $\Pi_i p_i$ as a shorthand notation of $p_1 \parallel \cdots \parallel p_n$ and write the nullary composition of $\parallel$ as $\perp$. The approximation relation defined later (Section 4.2) may also help the readers to understand the intuitive meaning of linear processes.

We also identify processes with "similar structure". The *strong structural congruence*, written $p \equiv_0 q$ over linear processes is the smallest congruence relation that satisfies:

$$p \parallel q \equiv_0 q \parallel p \qquad (p \parallel q) \parallel r \equiv_0 p \parallel (q \parallel r)$$
$$(\boldsymbol{\nu}[\langle x_1, y_1\rangle,\dots,\langle x_n, y_n\rangle])p \equiv_0 (\boldsymbol{\nu}[\langle x_{\sigma(1)}, y_{\sigma(1)}\rangle,\dots,\langle x_{\sigma(n)}, y_{\sigma(n)}\rangle])p,$$

where σ is a permutation over $\{1,\dots,n\}$. Given $I=\{i_1,\dots,i_n\}$, we write $\boldsymbol{\nu}[\langle x_i, y_i\rangle]_{i\in I}$ for $\boldsymbol{\nu}[\langle x_{i_1}, y_{i_1}\rangle,\dots,\langle x_{i_n}, y_{i_n}\rangle]$ because how the pairs $\langle x_{i_j}, y_{i_j}\rangle$ are ordered is inessential.

We now define the intersection types. The syntax of *raw types* and *raw (indexed) intersection types* are given by the following grammar:

$$\begin{array}{rl}\text{(Raw types)} & \rho ::= \mathbf{ch}^o_\alpha[\theta_1,\dots,\theta_m] \mid \mathbf{ch}^i_\alpha[\theta_1,\dots,\theta_n] \\ \text{(Raw intersection types)} & \theta ::= \bigwedge_{i\in I}(i,\rho_i)\end{array}$$

where $I \subseteq_{\text{fin}} \mathbf{Nat}$ and α ranges over the set of *levels* $(\mathcal{A},\leq)$, a universal poset in which any finite poset can be embedded into. In the above grammar, an intersection $\bigwedge_{i\in I}(i,\rho_i)$ is a map $i\mapsto\rho_i$ from I to types. The intuitive meaning of $\bigwedge_{i\in I}(i,\rho_i)$ is the intersection $\rho_{i_1}\wedge\rho_{i_2}\wedge\cdots\wedge\rho_{i_n}$ provided that $I=\{i_1<i_2<\cdots<i_n\}$.

Levels express timing information, and types are defined as raw types with "appropriate levels". Let us write $\overline{\mathbf{lv}}(\rho)$ and $\overline{\mathbf{lv}}(\theta)$ for the set of levels that appear in ρ and θ, respectively. Then *types* and *intersection types* are inductively defined as follows: $\mathbf{ch}^m_\alpha[\theta_1,\dots,\theta_n]$ ($m\in\{i,o\}$) is a type if θ_i is an intersection type for all $i\in\{1,\dots,n\}$ and $\alpha\leq\gamma$ for all $\gamma\in\overline{\mathbf{lv}}(\theta_1,\dots,\theta_n)$ and $\bigwedge_{i\in I}(i,\rho_i)$ is an intersection type if ρ_i is a type for all $i\in I$. *Hereafter, we use the metavariables ρ and θ to range over types and intersection types, respectively.*

Notations. We define $[n] \stackrel{\text{def}}{=} \{1,\dots,n\}$ for a natural number n. A special symbol $\bullet$ is introduced to mean undefined type of sort T; now an intersection type θ can be also be represented by a (total) function from $\mathbf{Nat}$ to the union of the set of types and $\{\bullet\}$. We write $(i_1,\rho_{i_1})\wedge\cdots\wedge(i_n,\rho_{i_n})$ for the intersection type θ such that $\mathrm{dom}(\theta)=\{i_1<\cdots<i_n\}$

[7] (For readers familiar with resource calculi) Although the intuitive meaning of $p \parallel q$ is the parallel composition of p and q, this process should be thought of as an analogous to the bag in the resource λ-calculi [5, 10].

and $\theta(i_j) = \rho_{i_j}$ for every $j \in [n]$. We also write $\top$ for the empty intersection, i.e. θ such that $\theta(i) = \bullet$ for all $i \in \mathbf{Nat}$. The *dual* $\rho^{\perp}$ of type ρ is defined by $\mathbf{ch}^o_\alpha[\vec{\theta}]^{\perp} \stackrel{\text{def}}{=} \mathbf{ch}^i_\alpha[\vec{\theta}]$ and $\mathbf{ch}^i_\alpha[\vec{\theta}]^{\perp}_\alpha \stackrel{\text{def}}{=} \mathbf{ch}^o_\alpha[\vec{\theta}]$. We also define $\theta^{\perp}$ by $\theta^{\perp}(i) \stackrel{\text{def}}{=} (\theta(i))^{\perp}$. In what follows, we may often omit the annotations φ on the outputs because they are not needed as long as we are dealing with simple examples.

The type $\mathbf{ch}^i_\alpha[\theta_1, \dots, \theta_n]$ is for a channel that is used to receive n lists, where the i-th list has type θ_i and the type $\mathbf{ch}^o_\alpha[\vec{\theta}]$ is for output channels. If the i-th list has type $(i_1, \rho_1) \wedge \cdots \wedge (i_m, \rho_m)$, it means that the j-the element of the list has type ρ_j. For example, $a_i(\langle x_1, x_2\rangle, \langle \bar{y}_1\rangle).x_1().x_2().\bar{y}_1\langle\langle\rangle\rangle$ is well-typed if a_i has type $\mathbf{ch}^i_\alpha[(1, \mathbf{ch}^i_\beta[]) \wedge (2, \mathbf{ch}^i_\gamma[]), (1, \mathbf{ch}^o_\gamma[\top])]$ with $\alpha \leq \beta \leq \gamma$. As mentioned, the levels are used to describe the timing of actions. In the above example, the level γ tells us that the second element of the first argument, namely x_2, and the first element of the second argument, namely $\bar{y}_1$, must be used at the same timing. Levels also describe the fact that x_1 must be used before x_2 and $\bar{y}_1$ are used.

Although the intersection types are non-commutative in the sense that $(0, \rho) \wedge (1, \rho') \neq (0, \rho') \wedge (1, \rho)$, we consider that they are isomorphic. Intuitively, this means that we do not mind much about the order of elements in a list. For example, we consider that $\bar{a}_0\langle\langle x_0, x_1\rangle\rangle$ and $\bar{a}_0\langle\langle x_1, x_0\rangle\rangle$ are almost identical. Without this identification, we face a technical problem: an approximation of a forwarder $a_0(\langle y_0, y_1\rangle).\bar{b}_0\langle\langle y_1, y_0\rangle\rangle$ cannot be seen as an "identity" because $(\boldsymbol{\nu}[\langle \bar{a}_0, a_0\rangle])(a_0(\langle y_0, y_1\rangle).\bar{b}_0\langle\langle y_1, y_0\rangle\rangle \mid \bar{a}_0\langle\langle x_0, x_1\rangle\rangle)$ "reduces to" $\bar{b}_0\langle\langle x_1, x_0\rangle\rangle$. Another possible way to avoid this problem is to use fully *commutative* intersection types. We did not use this approach because, in a commutative type system, the relationship between linear processes and execution sequences becomes less precise.

▶ **Definition 8** (Type isomorphism). *We write $\varphi\colon \rho \cong \rho'$ (resp. $\varphi\colon \theta \cong \theta'$) to mean that ρ and ρ' (resp. θ and θ') are* isomorphic *and that φ is the* witness *of this isomorphism. This relation is defined by the rules below:*[8]

$$\frac{}{\mathrm{id}_\bullet \colon \bullet \cong \bullet}$$

$$\frac{\varphi_i \colon \theta_i \cong \theta'_i \qquad (\textit{for } i \in [n])}{\mathbf{ch}^o_\alpha[\varphi_1, \dots, \varphi_n] \colon \mathbf{ch}^o_\alpha[\theta'_1, \dots, \theta'_n] \cong \mathbf{ch}^o_\alpha[\theta_1, \dots, \theta_n]}$$

$$\frac{\varphi_i \colon \theta_i \cong \theta'_i \qquad (\textit{for } i \in [n])}{\mathbf{ch}^i_\alpha[\varphi_1, \dots, \varphi_n] \colon \mathbf{ch}^i_\alpha[\theta_1, \dots, \theta_n] \cong \mathbf{ch}^i_\alpha[\theta'_1, \dots, \theta'_n]}$$

$$\frac{\sigma \colon \mathbf{Nat} \xrightarrow{\cong} \mathbf{Nat} \qquad \varphi_i \colon \rho_i \cong \rho'_{\sigma(i)} \qquad (\textit{for } i \in \mathbf{Nat})}{(\sigma, (\varphi_i)_{i \in \mathbf{Nat}}) \colon \bigwedge_{i \in \mathbf{Nat}} (i, \rho_i) \cong \bigwedge_{i \in \mathbf{Nat}} (i, \rho'_i)}$$

▶ Remark 9. The reason for annotating arguments of free outputs with φ is quite technical. The notion of type isomorphism was taken from the rigid intersection type system given by Tsukada et al. [21], but in their calculus, witnesses do not appear in the syntax. This is so because all the (raw) terms in their resource calculus are assumed to be in η-long form. (See [21] for details.)

Similarly, we may remove witnesses of type isomorphisms from our linear calculus if there is a way to convert a linear process p to an "equivalent" process p' that does not contain any free outputs. A possible way to do this is to transform a free output to a "bound

[8] Here, $\bigwedge_{i \in I}(i, \rho_i)$ is considered as a total map $\bigwedge_{i \in \mathbf{Nat}}(i, \rho_i)$ in which $\rho_i \stackrel{\text{def}}{=} \bullet$ if $i \notin I$.

$$\frac{\varphi_i : \theta_i \cong \theta'_i \qquad \varphi_i \bullet x^i = \lambda_i \quad (\text{for } i \in [n]) \quad \alpha \leq \overline{\mathbf{lv}}(\theta_1, \ldots, \theta_n)}{x^1 : \theta_1 \sqcap \cdots \sqcap x^n : \theta_n, \bar{a} : (i, \mathbf{ch}^o_\alpha[\theta'_1, \ldots, \theta'_n]) \vdash_\alpha \bar{a}_i\langle \lambda_1, \ldots, \lambda_n \rangle} \ (\text{TOut})$$

$$\frac{\Gamma, x^1 : \theta_1, \ldots, x^n : \theta_n \vdash_\beta p \quad \mathrm{id}_{\theta_i} \bullet x^i = \mu_i \quad \alpha \leq \beta}{\Gamma \sqcap a : (i, \mathbf{ch}^i_\beta[\theta_1, \ldots, \theta_n]) \vdash_\alpha a_i(\mu_1, \ldots, \mu_n).p} \ (\text{TIn}) \qquad \frac{\Gamma \vdash_\beta p \quad \alpha \leq \beta}{\Gamma \sqcap \tau : (i, \mathbf{ch}^i_\beta[]) \vdash_\alpha \tau_i.p} \ (\text{TTau})$$

$$\frac{\Gamma_1 \vdash_\alpha p_1 \qquad \Gamma_2 \vdash_\alpha p_2}{\Gamma_1 \sqcap \Gamma_2 \vdash_\alpha p_1 \mid p_2} \ (\text{TPar}) \qquad \frac{\Gamma_i \vdash_\alpha p_i \quad (1 \leq i \leq n)}{\Gamma_1 \sqcap \cdots \sqcap \Gamma_n \vdash_\alpha p_1 \parallel \cdots \parallel p_n} \ (\text{TRep})$$

$$\frac{}{\emptyset \vdash_\alpha \mathbf{0}} \ (\text{TNil}) \qquad \frac{\Gamma, \bar{a} : \theta, a : \theta^\perp \vdash_\alpha p}{\Gamma \vdash_\alpha (\boldsymbol{\nu}[\langle \bar{a}_i, a_i \rangle]_{i \in \mathrm{dom}(\theta)})p} \ (\text{TNu})$$

Figure 2 Typing rules for the intersection type system.

output + forwarder" (cf. [4]). That is to (recursively) transform a free output $\bar{a}_0\langle\langle \bar{b}_0 \rangle\rangle$ into $(\boldsymbol{\nu}[\langle \bar{c}_0, c_0 \rangle])(\bar{a}_0\langle\langle \bar{c}_0 \rangle\rangle \mid b(\mu).\bar{c}_0\langle \mu \rangle)$. We chose to keep free outputs in the syntax of the linear process because by doing so, it is easier to see the correspondence between a (non-linear) process P, which may contain free outputs, and its linear approximation p. Note that we cannot assume that (non-linear) processes do not contain free outputs because the validity of the transformation $\bar{a}\langle b \rangle = (\boldsymbol{\nu}\bar{c}c)(\bar{a}\langle \bar{c} \rangle \mid c \hookrightarrow \bar{b})$ is not something that is taken for granted (even if the forwarder does not introduce any delay). Actually, the above translation is an instance of the rule (2) that we aim to validate in this work.

We define yet another operator $\theta_1 \sqcap \theta_2$ for intersection types which "coalesces" the two intersection. It is defined by $(\theta_1 \sqcap \theta_2)(i) \stackrel{\text{def}}{=} \theta_1(i)$ if $i \in \mathrm{dom}(\theta_1)$, $(\theta_1 \sqcap \theta_2)(i) \stackrel{\text{def}}{=} \theta_2(i)$ if $i \in \mathrm{dom}(\theta_2)$ and $(\theta_1 \sqcap \theta_2)(i) \stackrel{\text{def}}{=} \bullet$ otherwise, provided that $\mathrm{dom}(\theta_1) \cap \mathrm{dom}(\theta_2) = \emptyset$.

A *type environment*, often denoted by Γ, is a finite set of pairs of the form $t : \theta$ with $\theta \neq \top$ such that $(t^1 : \theta_1), (t^2 : \theta_2) \in \Gamma$ implies $t^1 \neq t^2$. For notational convenience, we may write $\Gamma, x : \top$ to express Γ, i.e. allow $\top$ to appear in a type environment. We define $\mathrm{dom}(\Gamma)$ as $\{t \mid \exists\theta.\ (t : \theta) \in \Gamma\}$ and $\Gamma(t)$ by $\Gamma(t) \stackrel{\text{def}}{=} \theta$ if $(t : \theta) \in \Gamma$ and $\Gamma(t) \stackrel{\text{def}}{=} \top$ otherwise. For type environments Γ_1 and Γ_2, $\Gamma_1 \sqcap \Gamma_2$ is defined by pointwise extension of $\sqcap$, that is $\Gamma_1 \sqcap \Gamma_2 \stackrel{\text{def}}{=} \{(t : \theta) \mid t \in \mathrm{dom}(\Gamma_1) \cup \mathrm{dom}(\Gamma_2), \theta = \Gamma_1(t) \sqcap \Gamma_2(t)\}$ provided that $\Gamma_1(t) \sqcap \Gamma_2(t)$ is defined for $t \in \mathrm{dom}(\Gamma_1) \cup \mathrm{dom}(\Gamma_2)$; otherwise $\Gamma_1 \sqcap \Gamma_2$ is undefined.

We consider judgments of the form $\Gamma \vdash_\alpha p$ and the typing rules are given in Figure 2. We stipulate that the deduction is allowed only if the result of the $\sqcap$ operation in the conclusion is defined. The operation $\varphi \bullet x$ used in the above definition is defined by

$$(\sigma, (\varphi_i)_{i \in \mathbf{Nat}}) \bullet x \stackrel{\text{def}}{=} \langle \varphi_{\sigma^{-1}(i_1)} \bullet x_{\sigma^{-1}(i_1)}, \ldots, \varphi_{\sigma^{-1}(i_n)} \bullet x_{\sigma^{-1}(i_n)} \rangle,$$

where $(\sigma, (\varphi_i)_{i \in \mathbf{Nat}}) : \theta \cong \theta'$ and $\mathrm{dom}(\theta') = \{i_1 < \cdots < i_n\}$; similarly $\mathrm{id}_\theta \bullet x$ is also used to express $\langle x_{i_1}, \ldots, x_{i_n} \rangle$ when $\mathrm{dom}(\theta) = \{i_1 < \cdots < i_n\}$.

Let us explain how the subscript α of $\vdash_\alpha$ is used; the other parts of the typing rule should be easy to understand. The intuitive meaning of the subscript α of $\vdash_\alpha$ is the "current time". The typing rule for output actions ensures that the "level of $\bar{a}_i$" is the "current time", that is the rule ensures that the output cannot be delayed. On the other hand, we may delay an input or a τ action. For example, in the rule (TIn), the "level of a_i" can be greater than α meaning that we can delay the use of a_i. The rule (TIn) also says that the "level of a_i" must be equal to the level assigned to $\Gamma, x^1 : \theta_1, \ldots, x^n : \theta_n \vdash_\beta p$. This expresses the fact that the unguarded outputs in p must be used as soon as a_i is used, i.e. there cannot be any delay between an input and an output.

$$\frac{}{x : \{i_1, \dots, i_n\} \vdash \langle \varphi_1 \bullet x_{i_1}, \dots, \varphi_n \bullet x_{i_n} \rangle \sqsubset x} \qquad \frac{}{\vdash \mathbf{0} \sqsubset \mathbf{0}} \qquad \frac{}{\vdash \bot \sqsubset \tau.P}$$

$$\frac{X_j \vdash \lambda_j \sqsubset x^j \quad (\text{for } j \in [n])}{X_1 \sqcap \dots \sqcap X_n \sqcap \bar{a} : \{i\} \vdash \bar{a}_i \langle \lambda_1, \dots, \lambda_n \rangle \sqsubset \bar{a}\langle x^1, \dots, x^n \rangle} \qquad \frac{X \vdash p \sqsubset P}{X \sqcap \tau : \{i\} \vdash \tau_i.p \sqsubset \tau.P}$$

$$\frac{X, x^1 : S_1, \dots, x^n : S_n \vdash p \sqsubset P \qquad x^j : S_j \vdash \mu_j \sqsubset x^j \quad (\text{for } j \in [n])}{X \sqcap a : \{i\} \vdash a_i(\mu_1, \dots, \mu_n).p \sqsubset a(x^1, \dots, x^n).P}$$

$$\frac{X_1 \vdash p \sqsubset P \qquad X_1 \vdash q \sqsubset Q}{X_1 \sqcap X_2 \vdash p \mid q \sqsubset P \mid Q} \qquad \frac{X_i \vdash p_i \sqsubset P \quad (\text{for } i \in [n])}{X_1 \sqcap \dots \sqcap X_n \vdash p_1 \parallel \dots \parallel p_n \sqsubset !P}$$

$$\frac{X, x : S, y : S \vdash p \sqsubset P \qquad S = \{i_1, \dots, i_n\} \qquad \rho_i \sqsubset T \quad (\text{for } i \in S)}{X \vdash (\boldsymbol{\nu}[\langle x_{i_1}, y_{i_1} \rangle_{\rho_{i_1}}, \dots, \langle x_{i_n}, y_{i_n} \rangle_{\rho_{i_n}}])p \sqsubset (\boldsymbol{\nu}_T xy)P}$$

Figure 3 Rules for approximation relation. We stipulate that the deduction is allowed only if the result of the $\sqcap$ operation in the conclusion is defined.

4.2 Approximation

In this subsection we show how sorts are refined by intersection types and processes are approximated by linear processes.

Given a sort T, the *refinement relation* $\rho \sqsubset T$ (resp. $\theta \sqsubset T$), meaning that the type ρ (resp. the intersection type θ) refines the sort T, is defined by the following rules:

$$\frac{\theta_i \sqsubset T_i \quad (i \in [n]) \qquad m \in \{i, o\}}{\mathbf{ch}^m_\alpha[\theta_1, \dots, \theta_n] \sqsubset \mathbf{ch}^m[T_1, \dots, T_n]} \qquad \frac{\rho_i \sqsubset T \quad (i \in I \subseteq_{\text{fin}} \mathbf{Nat})}{\bigwedge_{i \in I}(i, \rho_i) \sqsubset T}.$$

We write $\Gamma \sqsubset \Delta$ if $(x : \theta) \in \Gamma$ implies that $(x : T) \in \Delta$ for some T and $\theta \sqsubset T$.

Next we show how processes are approximated by linear processes.

A *term refinement* X is a finite set of the form $t^1 : S_1, \dots, t^n : S_n$ such that $S_i \subseteq_{\text{fin}} \mathbf{Nat}$ and $i \neq j$ implies $t^i \neq t^j$, where each t^i is a (non-linear) channel name or the constant τ. The set S_i expresses how many times t^i is used in the approximation. Notations $X(t)$ and $X_1 \sqcap X_2$ are defined analogous to $\Gamma(t)$ and $\Gamma_1 \sqcap \Gamma_2$. There is a canonical way to obtain a term refinement from a type environment: given a type environment Γ, we define $\Gamma^\natural$ as $\{(t : \mathrm{dom}(\Gamma(t))) \mid t \in \mathrm{dom}(\Gamma)\}$.

An *approximation judgement* is of the form $X \vdash p \sqsubset P$ and inference rules for judgments are given in Figure 3. It should be emphasized that we do not allow $\bot \sqsubset \bar{a}\langle \vec{x} \rangle$, that is we ensure that all the output actions are used. Note that we can discard an output action that is guarded by τ, i.e. $\bot \sqsubset \tau.\bar{a}\langle \vec{x} \rangle$, and this is why the translation $(-)^\dagger$ defined in Section 2 allows us to relate the reduction $\longrightarrow$ with $\Rightarrow$.

4.3 Reduction

This subsection defines the reduction relation for linear processes. We also show that every reduction sequence from P has a representation by a linear process that approximates P.

The reduction relation of linear processes is almost the same as that of processes except for the fact that we take actions of type isomorphisms to linear processes into account. The action of φ to linear processes is defined by the rules in Figure 4. It is defined via the action of type isomorphisms on subject names and operation $\{\varphi \bullet y/x\}$, which substitutes

$$\langle \varphi_1 \bullet x_{i_1}, \ldots, \varphi_k \bullet (\varphi' \bullet x_{i_k}), \ldots, \varphi_n \bullet x_{i_n} \rangle \stackrel{\text{def}}{=} \langle \varphi_1 \bullet x_{i_1}, \ldots, (\varphi_k \circ \varphi') \bullet x_{i_k}, \ldots, \varphi_n \bullet x_{i_n} \rangle$$
$$(\mathbf{ch}^o[\varphi] \bullet \bar{a}_i)\langle\langle \varphi'_{i_1} \bullet x_{i_1}, \ldots, \varphi'_{i_n} \bullet x_{i_n} \rangle\rangle \stackrel{\text{def}}{=} \bar{a}_i \langle\langle (\varphi_{i_1} \circ \varphi'_{\sigma(i_1)}) \bullet x_{\sigma(i_1)}, \ldots, (\varphi_{i_n} \circ \varphi'_{\sigma(i_n)}) \bullet x_{\sigma(i_n)} \rangle\rangle$$
$$(\mathbf{ch}^i[\varphi] \bullet a_i)(\langle x_{i_1}, \ldots, x_{i_n} \rangle).p \stackrel{\text{def}}{=} a_i(\langle x_{i_1}, \ldots, x_{i_n} \rangle).p\{\varphi_{\sigma^{-1}(j)} \bullet x_{\sigma^{-1}(j)}/x_j\}_{j \in \{i_1, \ldots, i_n\}}$$

Figure 4 Action of isomorphisms on linear (monadic) processes where $\varphi = (\sigma, (\varphi_i))$ in the last two equations; the action on polyadic processes is defined similarly.

$\varphi \bullet y$ to x. The substitution $\{\varphi \bullet y/x\}$ works as the standard substitution, except for the fact the action of φ is performed after the substitution. The witness $\varphi_2 \circ \varphi_1 \colon \rho_1 \cong \rho_3$ is the *composition* of $\varphi_1 \colon \rho_1 \cong \rho_2$ and $\varphi_2 \colon \rho_2 \cong \rho_3$, which is defined as in the case of rigid resource calculus [21]. (The definition of $\varphi_2 \circ \varphi_1$ is not necessary to understand the following content; see Appendix B.1 for the definition.) For readability, given $\lambda \stackrel{\text{def}}{=} \langle \varphi_1 \bullet y_1, \ldots, \varphi_n \bullet y_n \rangle$ and $\mu \stackrel{\text{def}}{=} \langle x_1, \ldots, x_n \rangle$, we write $\{\lambda/\mu\}$ to denote $\{\varphi_1 \bullet y_1/x_1, \ldots, \varphi_n \bullet y_n/x_n\}$.

The *structural precongruence* $\Rrightarrow$ over linear process is the smallest *precongruence* relation that contains $\equiv_0$, contains α-equivalence and satisfies:

$$\mathbf{0} \mid p \Lleftarrow\!\!\Rrightarrow p \qquad p \mid q \Lleftarrow\!\!\Rrightarrow q \mid p \qquad (p \mid q) \mid r \Lleftarrow\!\!\Rrightarrow p \mid (q \mid r)$$
$$(\boldsymbol{\nu}[\langle \vec{w}, \vec{z} \rangle])(\boldsymbol{\nu}[\langle \vec{y}, \vec{z} \rangle])p \Lleftarrow\!\!\Rrightarrow (\boldsymbol{\nu}[\langle \vec{y}, \vec{z} \rangle])(\boldsymbol{\nu}[\langle \vec{w}, \vec{x} \rangle])p \qquad (\mathbf{fn}(\vec{w}, \vec{x}) \cap \mathbf{fn}(\vec{y}, \vec{z}) = \emptyset)$$
$$(\boldsymbol{\nu}[\langle x_1, y_1 \rangle, \ldots, \langle x_n, y_n \rangle])p \mid q \Rrightarrow (\boldsymbol{\nu}[\langle x_1, y_1 \rangle, \ldots, \langle x_n, y_n \rangle])(p \mid q) \quad (\vec{x}, \vec{y} \notin \mathbf{fn}(q))$$

where $p \Lleftarrow\!\!\Rrightarrow q$ means $p \Rrightarrow q$ and $q \Rrightarrow p$. The *structural congruence* $\equiv$ for linear processes is defined as symmetric closure of $\Rrightarrow$.

We define the *one-step reduction relation* over well-typed linear processes by the base rule

$$(\boldsymbol{\nu}\vec{\xi})(\boldsymbol{\nu}[\langle \bar{a}_j, a_j \rangle]_{j \in J})(\Pi_{i \in I} a_i(\mu_{i1}, \ldots, \mu_{in_i}).p_i \mid \bar{a}_m \langle \lambda_1, \ldots, \lambda_n \rangle \mid q) \xrightarrow{0}$$
$$(\boldsymbol{\nu}\vec{\xi})(\boldsymbol{\nu}[\langle \bar{a}_j, a_j \rangle]_{j \in J'})(\Pi_{i \in I'} a_i(\mu_{i1}, \ldots, \mu_{in_i}).p_i \mid p_m\{\lambda_1/\mu_{m1}, \ldots, \lambda_n/\mu_{mn}\} \mid q)$$

where $(\boldsymbol{\nu}\vec{\xi})$ is a sequence of name restrictions, $m \in I \subseteq J$, $J' = J \setminus \{m\}$ and $I' = I \setminus \{m\}$, and the structural rule which concludes $p \xrightarrow{0} q$ from $p \Rrightarrow p'$ and $p' \xrightarrow{0} q$. The relation $\xrightarrow{\tau}$ is obtained by replacing the base rule of the $\xrightarrow{0}$ with $(\boldsymbol{\nu}\vec{\xi})(\tau_i.p \mid q) \xrightarrow{\tau} (\boldsymbol{\nu}\vec{\xi})(p \mid q)$.

▶ Remark 10. We use $\Rrightarrow$ instead of $\equiv$ in the definition of reduction because $X \vdash p \sqsubset P$ and $p \equiv q$ does not ensure the existence of Q such that $X \vdash q \sqsubset Q$ and $P \equiv Q$. For instance, if $P \stackrel{\text{def}}{=} (\boldsymbol{\nu}\bar{a}a)(!a(x).R \mid \tau.\bar{a}\langle y \rangle)$ then $(\boldsymbol{\nu}[])(\bot \mid \bot)$ approximates P and this linear process is structurally congruent to $(\boldsymbol{\nu}[])\bot \mid \bot$, but there is no Q such that $(\boldsymbol{\nu}[])\bot \mid \bot \sqsubset Q$ and $P \equiv Q$.

We now show the relationship between execution sequences and linear approximations. Let us write $P \xrightarrow{\pi} Q$ if there exists a sequence $P = P_0 \xrightarrow{l_1} P_1 \xrightarrow{l_2} \cdots \xrightarrow{l_n} P_n = Q$, where each l_i is either 0 or τ, and $\pi = l_1 l_2 \ldots l_n$; $p \xrightarrow{\pi} q$ is defined similarly. We write $(p \xrightarrow{\pi} q) \sqsubset (P \xrightarrow{\pi} Q)$ if there exists $p = p_0 \xrightarrow{l_1} \cdots \xrightarrow{l_n} p_n = q$ and $P = P_0 \xrightarrow{l_1} \cdots \xrightarrow{l_n} P_n = Q$ such that $X_i \vdash p_i \sqsubset P_i$ for some X_i for each $i \in \{0, \ldots, n\}$ and $\pi = l_1 \cdots l_n$.

▶ **Proposition 11.** *Let $\tau : \mathbf{ch}^i[] \vdash P$, i.e. let P be a process without any free names.*

1. *Suppose that $\Gamma \vdash_\alpha p$ and $\Gamma^\natural \vdash p \sqsubset P$. If $p \xrightarrow{\pi} q$ then we have $(p \xrightarrow{\pi} q) \sqsubset (P \xrightarrow{\pi} Q)$ for some Q.*
2. *Assume $P \xrightarrow{\pi} Q$, $\Gamma \vdash_\alpha q$ and $\Gamma^\natural \vdash q \sqsubset Q$. Then we have $(p \xrightarrow{\pi} q) \sqsubset (P \xrightarrow{\pi} Q)$ for some p.*

5 LTS based on linear approximations

Using the notion of linear processes, we introduce a labelled transition system (LTS) for processes in the form of a presheaf to describe the behaviour of processes in which outputs cannot be delayed. Intuitively, the LTS that describes the behaviour of P is given as an LTS whose states are linear approximations of P and transition relation is the extension relation $\trianglelefteq$, which we briefly explained in Section 3. This LTS will be presented as a presheaf following the view that presheaves can be regarded as transition systems [9, 23].

5.1 Extension relation

We now define an ordering $p' \trianglelefteq p$ over linear processes, which may be read as "p extends p'". Giving a larger linear approximation corresponds to extending an execution sequence.

Before we define the extension relation on linear processes, we define the extension relation over types.

▶ **Definition 12.** *Let A be a set of levels.* Restriction of types and intersection types *are inductively defined by:*

$$\mathbf{ch}^{o}_{\alpha}[\theta_1,\dots,\theta_n]\restriction_A \stackrel{\mathrm{def}}{=} \begin{cases}\mathbf{ch}^{o}_{\alpha}[\theta_1\restriction_A,\dots,\theta_n\restriction_A] & (\text{if } \alpha\in A)\\ \bullet & (\text{otherwise})\end{cases} \quad \text{and } (\theta\restriction_A)(i) \stackrel{\mathrm{def}}{=} \theta(i)\restriction_A.$$

where restrictions over input types are defined similar to that of output types. The restriction of type isomorphisms $\varphi\restriction_A$ is defined in a similar manner. (See the appendix for details.) We write $\rho' <: \rho$ and if $\rho' = \rho\restriction_A$ for some $A \subseteq \mathcal{A}$ and $\varphi' <: \varphi$ if $\varphi' = \varphi\restriction_A$ for some $A \subseteq \mathcal{A}$.

The *extension relation* on linear processes, written $p' \trianglelefteq p$, is inductively defined by the rules in Figure 5. For example, $a_1(\langle\rangle).\bot \trianglelefteq a_1(\langle x_1\rangle).\tau_1.x_1\langle\rangle$ holds and this intuitively means that $!a(x).x\langle\rangle \xrightarrow{a(x)} !a(x).x\langle\rangle \mid x\langle\rangle$ can be extended to $!a(x).x\langle\rangle \xrightarrow{a(x)} !a(x).x\langle\rangle \mid x\langle\rangle \xrightarrow{x\langle\rangle} !a(x).x\langle\rangle \mid \mathbf{0}$ (under the assumption that both of the linear processes approximate $!a(x).x\langle\rangle$).

Extending a linear process does not precisely correspond to extending an execution sequence: there are cases where an execution sequence cannot be extended even if the corresponding linear process can be extended. This problem is due to the existence of deadlocks. For instance, we have $(\boldsymbol{\nu}[])(\boldsymbol{\nu}[])(\bot \mid \bot) \trianglelefteq (\boldsymbol{\nu}[\langle\bar{a}_1, a_1\rangle])(\boldsymbol{\nu}[\langle\bar{b}_1, b_1\rangle])(a_1.\tau_1.\bar{b}_1 \mid b_1.\tau_2.\bar{a}_1)$, but both of the linear processes are not reducible. To exclude linear processes that may create a deadlock, we introduce the notion of terminable processes:

▶ **Definition 13.** *A linear process p is* idle *if it has no action (input, output nor τ), i.e. consisting of 0, $\bot$, $\mid$ and $\boldsymbol{\nu}[]$. A linear process p is* terminable *if $(\boldsymbol{\nu}\vec{\xi}).(p \mid q) \xrightarrow{0}{}^{*} r$ for some $\vec{\xi}$, q and idle r.*

Only terminable processes will be used as the states of the LTS.

In case p and p' correspond to executions that only consists of $\xrightarrow{0}$ and $\xrightarrow{\tau}$ the intuition that $p \trianglelefteq p'$ corresponds to "extending execution sequences" can be formalised as follows:

▶ **Proposition 14.** *Let $\tau : \mathbf{ch}^{i}[] \vdash P$ and let $\mathcal{R}$ be a relation between execution sequences starting from P and well-typed terminable linear approximations of P such that $(P \xrightarrow{\pi} Q)\ \mathcal{R}\ p$ if and only if $(p \xrightarrow{\pi} q) \sqsubset (P \xrightarrow{\pi} Q)$ for a process q that is typed under the empty environment. Then if $(P \xrightarrow{\pi} Q)\ \mathcal{R}\ p$*

1. *$Q \xrightarrow{\pi'} Q'$ implies that $(P \xrightarrow{\pi} Q \xrightarrow{\pi'} Q')\ \mathcal{R}\ p'$ and $p \trianglelefteq p'$ for some p'.*
2. *if $p \trianglelefteq p'$ for some terminable well-typed linear process p' that approximates P, then there is an execution $Q \xrightarrow{\pi'} Q'$ such that $(P \xrightarrow{\pi} Q \xrightarrow{\pi'} Q')\ \mathcal{R}\ p'$.*

$$\frac{I = \{i_1 < \cdots < i_m\} \qquad J = \{j_1 < \cdots < j_n\} \qquad J \subseteq I \qquad \varphi'_i <: \varphi_i \quad (\text{for } i \in J)}{\langle \varphi'_{j_1} \bullet x_{j_1}, \ldots, \varphi'_{j_n} \bullet x_{j_n} \rangle \trianglelefteq \langle \varphi_{i_1} \bullet x_{i_1}, \ldots, \varphi_{i_m} \bullet x_{i_m} \rangle}$$

$$\frac{}{\mathbf{0} \trianglelefteq \mathbf{0}} \qquad \frac{}{\bot \trianglelefteq \tau_i.p} \qquad \frac{p \trianglelefteq q}{\tau_i.p \trianglelefteq \tau_i.q} \qquad \frac{J \subseteq I \qquad \rho'_i <: \rho_i \quad (\text{for } i \in J) \qquad p \trianglelefteq q}{(\boldsymbol{\nu}[\langle \bar{a}_i, a_i \rangle_{\rho'_i}]_{i \in J})q \trianglelefteq (\boldsymbol{\nu}[\langle \bar{a}_i, a_i \rangle_{\rho_i}]_{i \in I})p}$$

$$\frac{\lambda'_j \trianglelefteq \lambda_j \quad (\text{for } j \in [n])}{\bar{a}_i \langle \lambda'_1, \ldots, \lambda'_n \rangle \trianglelefteq \bar{a}_i \langle \lambda_1, \ldots, \lambda_n \rangle} \qquad \frac{p \trianglelefteq q \qquad \mu'_j \trianglelefteq \mu_j \quad (\text{for } j \in [n])}{a_i(\mu'_1, \ldots, \mu'_n).p \trianglelefteq a_i(\mu_1, \ldots, \mu_n).q}$$

$$\frac{}{\bot \trianglelefteq a_i(\vec{\mu}).q} \qquad \frac{p' \trianglelefteq p \qquad q' \trianglelefteq q}{p' \mid q' \trianglelefteq p \mid q} \qquad \frac{m \le n \qquad p'_i \trianglelefteq p_i \quad p'_i \ne \bot \quad (\text{for } i \in [m])}{p'_1 \parallel \cdots \parallel p'_m \trianglelefteq p_1 \parallel \cdots \parallel p_n}$$

Figure 5 Rules for extension relation. Here we identify processes up to $\equiv_0$.

5.2 Presheaf semantics

We define the LTS of $\Delta \vdash P$ as a presheaf $[\![P]\!] \colon \mathcal{E}_\Delta \to \mathbf{Sets}$. Roughly speaking, the path category $\mathcal{E}_\Delta$ is a category of type environments that refines Δ and $[\![P]\!]$ maps a type environment Γ to the set of approximations of P that is typed under Γ.

Actually, the objects of the path category are not only type environments, but the pair of type environments and the "current time".

▶ **Definition 15.** *We say that* (Γ, α) extends (Γ', α) *and write* $(\Gamma', \alpha) <: (\Gamma, \alpha)$ *if there exists a* witness $A \subseteq \overline{\mathbf{lv}}(\Gamma) \cup \{\alpha\}$ *that satisfies (i)* $\mathrm{dom}(\Gamma') \subseteq \mathrm{dom}(\Gamma)$ *and* $\Gamma'(t) = \Gamma(t){\restriction_A}$*, for* $t \in \mathrm{dom}(\Gamma)$*, (ii)* $\alpha \in A$ *and (iii)* A *is downward-closed: for every* β, γ *appearing in* Γ*,* $\beta \le \gamma$ *and* $\gamma \in A$ *implies* $\beta \in A$*.*
We define the *category of type environments* $\mathcal{E}_\Delta$ to be a category whose objects are (Γ, α) such that $\Gamma \sqsubset \Delta$ and whose morphisms are given by the relation $(\Gamma', \alpha) <: (\Gamma, \alpha)$.

We now define the presheaf $[\![P]\!]$. Given $\Delta \vdash P$ and $\Gamma \sqsubset \Delta$, the set $[\![P]\!](\Gamma, \alpha)$ is defined by $[\![P]\!](\Gamma, \alpha) \stackrel{\text{def}}{=} \{p \mid \Gamma^\natural \vdash p \sqsubset P, \Gamma \vdash_\alpha p \text{ and } p \text{ is terminable}\}$. (Here we are identifying linear processes up to $\equiv_0$.)

▶ **Proposition 16.** *Assume that* $\Gamma \vdash_\alpha p$*,* p *is terminable and* $(\Gamma', \alpha) <: (\Gamma, \alpha)$*. Then there is a unique (up to* $\equiv_0$*) linear process that satisfy* $q \trianglelefteq p$ *and* $\Gamma' \vdash_\alpha q$*.*

By Proposition 16 there is a map $[\![P]\!](-,-)$ that maps an extension relation $(\Gamma', \alpha) <: (\Gamma, \alpha)$ to a function from $[\![P]\!](\Gamma, \alpha)$ to $[\![P]\!](\Gamma', \alpha)$ that maps $p \in [\![P]\!](\Gamma, \alpha)$ to q such that $q \trianglelefteq p$ and $\Gamma' \vdash_\alpha q$. Given $\Gamma \vdash_\alpha p$, we will write $p{\restriction_{\Gamma', \alpha}}$ for the process that is uniquely determined by the above proposition, provided that $(\Gamma', \alpha) <: (\Gamma, \alpha)$.

▶ **Theorem 17.** *Let* $\Delta \vdash P$*. Then* $[\![P]\!](-,-)$ *is a functor from* $\mathcal{E}_\Delta$ *to* **Sets***.*

▶ **Example 18.** Consider a process $P \stackrel{\text{def}}{=} (\boldsymbol{\nu}\bar{a}a)(!a(x).\tau.\bar{z}\langle\rangle \mid !a(x).\tau.x\langle\bar{y}\rangle \mid \bar{a}\langle\bar{w}\rangle)$ such that $\Delta \vdash P$, where $\Delta \stackrel{\text{def}}{=} \tau : \mathbf{ch}^i[], \bar{w} : \mathbf{ch}^o[\mathbf{ch}^o[]], \bar{y} : \mathbf{ch}^o[], \bar{z} : \mathbf{ch}^o[]$. Then we have

$$[\![P]\!](\Gamma_1, \alpha) = \{(\boldsymbol{\nu}[\langle \bar{a}_1, a_1 \rangle])(a_1(\langle\rangle).\bot \mid \bot \mid \bar{a}_1\langle\langle\rangle\rangle), (\boldsymbol{\nu}[\langle \bar{a}_1, a_1 \rangle])(\bot \mid a_1(\langle\rangle).\bot \mid \bar{a}_1\langle\langle\rangle\rangle)\}$$
$$[\![P]\!](\Gamma_2, \alpha) = \{(\boldsymbol{\nu}[\langle \bar{a}_1, a_1 \rangle])(\bot \mid a_1(\langle x_1 \rangle).\tau_1.x_1\langle\langle\rangle\rangle \mid \bar{a}_1\langle\langle \bar{w}_1 \rangle\rangle)\}$$
$$[\![P]\!](\Gamma_3, \alpha) = \{(\boldsymbol{\nu}[\langle \bar{a}_1, a_1 \rangle])(\bot \mid a_1(\langle x_1 \rangle).\tau_1.x_1\langle\langle \bar{y}_1 \rangle\rangle \mid \bar{a}_1\langle\langle \bar{w}_1 \rangle\rangle)\}$$

for $\Gamma_1 \stackrel{\text{def}}{=} \emptyset$, $\Gamma_2 \stackrel{\text{def}}{=} \tau\colon(1, \mathbf{ch}^{i}_{\beta}[]), \bar{w} : (1, \mathbf{ch}^{o}_{\beta}[\top])$ and $\Gamma_3 \stackrel{\text{def}}{=} \tau\colon(1, \mathbf{ch}^{i}_{\beta}[]), \bar{w} : (1, \mathbf{ch}^{o}_{\beta}[(1, \mathbf{ch}^{o}_{\gamma}[])])$, $\bar{y} : (1, \mathbf{ch}^{o}_{\gamma}[])$ with $\alpha < \beta < \gamma$. Note that $(\Gamma_2, \alpha) <: (\Gamma_3, \alpha)$ because we can take $\{\alpha, \beta\}$ as the witness. We also have $(\Gamma_1, \alpha) <: (\Gamma_2, \alpha)$ since $\{\alpha\}$ is a witness. The function $[\![P]\!]((\Gamma_1, \alpha) <: (\Gamma_2, \alpha))$ maps the only linear process of $[\![P]\!](\Gamma_2, \alpha)$ to the linear process $(\boldsymbol{\nu}[\langle a_1, \bar{a}_1\rangle])(\bot \mid a_1(\langle\rangle).\bot \mid \bar{a}\langle\langle\rangle\rangle)$.

6 $\simeq^{c}_{\tau}$ is a π_F-theory

As explained in Section 3, to prove Theorem 4, it suffices to show that (i) $\simeq^{c}_{\tau}$ satisfies the axioms such as (2) and (3), and (ii) that barbed congruence is a congruence relation, which trivially holds. Instead of directly proving (i), we define a yet another equivalence $\sim$ and show that $\sim$ is a congruence relation that satisfies the axioms and $\sim \subseteq \simeq^{c}_{\tau}$. These are relatively easier to show than to directly prove (i).

The equivalence $\sim$ is defined using the notion of *open map bisimulation* [12].[9] We write $P \sim Q$ if and only if $[\![P]\!]$ and $[\![Q]\!]$ are open map bisimilar, i.e. if there is a span $[\![P]\!] \xleftarrow{f} X \xrightarrow{g} [\![Q]\!]$, where f and g are open maps. A map $f\colon [\![P]\!] \to [\![Q]\!]$ is called an *open map* if for every $m\colon y(\Gamma_1, \alpha_1) \to y(\Gamma, \alpha_2)$, making the square below commute

$$\begin{tikzcd} y(\Gamma_1,\alpha) \arrow[r, "p"] \arrow[d, "m"] & {[\![P]\!]} \arrow[d, "f"] \\ y(\Gamma_2,\alpha) \arrow[r, "q"] & {[\![Q]\!]} \end{tikzcd} \qquad \text{there is a diagonal map } d \qquad \begin{tikzcd} y(\Gamma_1,\alpha) \arrow[r, "p"] \arrow[d, "m"] & {[\![P]\!]} \arrow[d, "f"] \\ y(\Gamma_2,\alpha) \arrow[r, "q"] \arrow[ur, "d"] & {[\![Q]\!]} \end{tikzcd}$$

making the two triangles commute.

Showing that there is an open map $f\colon [\![P]\!] \to [\![Q]\!]$ is analogous to giving a functional bisimulation (indexed by (Γ, α)) between $[\![P]\!](\Gamma, \alpha)$ and $[\![Q]\!](\Gamma, \alpha)$. The naturality of f means that f is a simulation because the naturality says $f(p)\restriction_{\Gamma_1,\alpha} = f(p\restriction_{\Gamma_1,\alpha})$. The morphism f being open ensures that it is not only a simulation, but a bisimulation. The existence of a diagonal map ensures that if (i) $\Gamma_1 \vdash_\alpha p$ and $f_{\Gamma_1,\alpha}(p) = q$ and (ii) $\Gamma_2 \vdash_\alpha q'$ with $(\Gamma_1, \alpha) <: (\Gamma_2, \alpha)$ and $q = q'\restriction_{\Gamma_1,\alpha}$ then there is p' such that $f_{\Gamma_2,\alpha}(p') = q'$. In simpler words, the existence of a diagonal map says that if $r(p) = q$ and "q can be extended as $q \trianglelefteq q'$" then "p can be extended accordingly".

The fact that $\sim$ satisfies the rules such as (3) (given in Section 3), can be proved by "proof manipulation". As explained, to show that $P \sim Q$, it suffices to give a functional bisimulation between $[\![P]\!](\Gamma, \alpha)$ and $[\![Q]\!](\Gamma, \alpha)$. As a special case, let us consider the case where $P = (\boldsymbol{\nu}\bar{a}a)(!a(x).P \mid \bar{a}\langle y\rangle)$ and $Q = (\boldsymbol{\nu}\bar{a}a)(!a(x).P \mid P\{y/x\})$. In this case, a functional bisimulation f can be defined by $f(p) \stackrel{\text{def}}{=} q$, where $p \xrightarrow{0} q$. The proof that this f is a bisimulation is similar to that of subject reduction/expansion. For example, if $f(p) = q$ and $q \trianglelefteq q'$ then it suffices to construct a linear process p' such that $p' \xrightarrow{0} q'$ (subject to the condition that p' is a suitable extension of p) as in the proof of subject expansion. Checking that $\sim$ satisfies the other axioms can be done similarly.

We can also show that $\sim$ is a congruence relation. Checking that $\sim$ is a congruence is not that difficult, thanks to the fact that $\trianglelefteq$ is defined according to the structure of a process. Also note that, unlike in the traditional π-calculus, we do not have any problem with input prefixing since communication only occurs between names that are explicitly bound by $\boldsymbol{\nu}$ in the π_F-calculus. That is, placing a process P into a context $C \stackrel{\text{def}}{=} !a(x).[]$ does not add new possibilities for interactions among names in P.

[9] To be more specific, we define the open-map bisimulation in the setting where $y\mathcal{E}_\Delta$ (the Yoneda embedding of $\mathcal{E}_\Delta$) is the path category and $[\mathcal{E}_\Delta^{\mathrm{op}}, \mathbf{Sets}]$ is the category of models.

The fact that $\sim$ is a congruence relation implies the following theorem.

▶ **Theorem 19.** *π_F-processes modulo $\sim$ form a compact closed Freyd category.*

The main theorem (Theorem 4), which states the existence of a compact closed Freyd model that is fully abstract with respect to $\simeq^c_\tau$, is a consequence of the above theorem and the following lemma:

▶ **Lemma 20.** *If $P \sim Q$ then $P \simeq^c_\tau Q$.*

This lemma is proved by showing that $\sim$ implies $\overset{\bullet}{\sim}_\tau$ (and using the fact that $\sim$ is a congruence), which basically follows from the relation between linear approximations and $\Rightarrow$ (Proposition 14). Roughly speaking, to show that $P \sim Q$ implies $P \overset{\bullet}{\sim}_\tau Q$ it suffices to show that the following relation is a barbed bisimulation, where $\mathcal{R}$ is the relation used in Proposition 14.

$$\left\{ (P_k, Q_k) \;\middle|\; \begin{array}{l} (P = P_0 \Rightarrow P_1 \Rightarrow \cdots \Rightarrow P_k)\ \mathcal{R}\ p_k \\ (Q = Q_0 \Rightarrow Q_1 \Rightarrow \cdots \Rightarrow Q_k)\ \mathcal{R}\ q_k \\ p_k \sim q_k \\ \text{for some sequences } P = P_0 \Rightarrow P_1 \Rightarrow \cdots \Rightarrow P_k \text{ and} \\ Q = Q_0 \Rightarrow Q_1 \Rightarrow \cdots \Rightarrow Q_k \text{ and} \\ \text{some linear approximations } p_k \text{ and } q_k \end{array} \right\}$$

Here $p \sim q$ means that there exists a span of open maps $[\![P]\!] \xleftarrow{f} X \xrightarrow{g} [\![Q]\!]$ and an element x of $X(\Gamma, \alpha)$ such that $f_{\Gamma,\alpha}(x) = p$ and $g_{\Gamma,\alpha}(x) = q$, i.e. p and q are "bisimilar states". Strictly speaking, we cannot directly use Proposition 14 because P and Q have free outputs. However, the same argument can be applied to the current situation to show the correspondence between $\trianglelefteq$ and extending a reduction sequence, which concludes the above lemma.

7 Related Work

The notion of presheaf plays a central role in our work, but we are not the first one to use presheaf to model the π-calculus. Cattani et al. [8] gave a denotational semantics of the π-calculus within an indexed category of profunctors. The model is fully abstract in the sense that bisimulation in the model, obtained from open maps, coincides with strong late bisimulation. Although their work and our work both use presheaf (or profunctor), they are conceptually different. Our work is motivated by categorical type theory correspondence, whereas the work by Cattani et al. [8] is motivated by a desire to obtain a systematic and algebraic understanding of bisimulation. From a technical point of view, the definition of the path category is significantly different as well. Their path category is indexed by the category of finite name sets and injective maps so that it can treat fresh names as in the domain theoretic models of the π-calculus [20, 11]. On the other hand, our path category is simply the category of type environments of an intersection type system.

A non-idempotent intersection type system for a variant of the π-calculus has also been introduced by Dal Lago et al. [13]. This intersection type system is also inspired by the notion of linear approximation. The connection between linear approximations and intersection types [14] was applied to the encoding of π-calculus to proof-nets to derive the basis of an intersection type system for a fragment of the local π-calculus [24, 16] called *hyperlocalised*

π*-calculus*. They showed that the type system obtained this way characterises some "good behaviour", such as deadlock-freedom, of hyperlocalised processes. In contrast to our work, they use intersection types to guarantee that typable processes are "well-behaved", rather than to define the "operational semantics" of the calculus.

As briefly explained in the introduction, the delays that forwarders add has also been an issue in the field of game semantics. In game semantics, forwarders correspond to copycat strategies and the delay copycat strategies introduce was an obstacle to model synchronous computations using game semantics. Game models in which a "copycat strategy that does not introduce any delay" can be expressed were recently introduced by Castellan and Yoshida to give a fully abstract game semantics of the *synchronous* session π-calculus [7] and by Melliès in a framework called template games [15]. Although these work are apparently different from ours, we believe that they are relevant to our work given that there is a tight relationship between game semantics and linear approximations [22]; detailed comparisons are left for future work.

8 Conclusion

We proposed a variant of the π-calculus whose barbed congruence $\simeq^c_\tau$ can be captured categorically in return for having a non-standard reduction relation $\Rrightarrow$. Technically, to handle $\Rrightarrow$, we developed a system of linear approximations that captures the behaviour of a process and developed an LTS based on linear approximations. The standard reduction relation $\rightarrow$ and $\Rrightarrow$ have been related by the translation $(-)^\dagger$, and we showed that $(P)^\dagger \simeq^c_\tau (Q)^\dagger$ implies $P \simeq^c Q$ ($\simeq^c$ is the conventional barbed congruence). Although we fail to achieve full abstraction, this result is important because it suggests the possibility of using compact closed Freyd models to reason about conventional π-calculus via the translation, which is the future direction we aim to pursue.

References

1 Samson Abramsky. Proofs as processes. *Theor. Comput. Sci.*, 135(1):5–9, 1994. `doi:10.1016/0304-3975(94)00103-0`.

2 Samson Abramsky, Simon J. Gay, and Rajagopal Nagarajan. Interaction categories and the foundations of typed concurrent programming. In *Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany*, pages 35–113, 1996.

3 Gianluigi Bellin and Philip J. Scott. On the π-calculus and linear logic. *Theor. Comput. Sci.*, 135(1):11–65, 1994. `doi:10.1016/0304-3975(94)00104-9`.

4 Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theor. Comput. Sci.*, 195(2):205–226, 1998. `doi:10.1016/S0304-3975(97)00220-X`.

5 Gérard Boudol. The lambda-calculus with multiplicities (abstract). In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 1–6. Springer, 1993. `doi:10.1007/3-540-57208-2_1`.

6 Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016. `doi:10.1017/S0960129514000218`.

7 Simon Castellan and Nobuko Yoshida. Two sides of the same coin: session types and game semantics: a synchronous side and an asynchronous side. *PACMPL*, 3(POPL):27:1–27:29, 2019. `doi:10.1145/3290340`.

8 Gian Luca Cattani, Ian Stark, and Glynn Winskel. Presheaf models for the pi-calculus. In *Category Theory and Computer Science, 7th International Conference, CTCS '97, Santa Margherita Ligure, Italy, September 4-6, 1997, Proceedings*, pages 106–126, 1997. doi:10.1007/BFb0026984.

9 Gian Luca Cattani and Glynn Winskel. Presheaf models for concurrency. In *Computer Science Logic, 10th International Workshop, CSL '96, Annual Conference of the EACSL, Utrecht, The Netherlands, September 21-27, 1996, Selected Papers*, pages 58–75, 1996. doi:10.1007/3-540-63172-0_32.

10 Thomas Ehrhard and Laurent Regnier. Uniformity and the taylor expansion of ordinary lambda-terms. *Theor. Comput. Sci.*, 403(2-3):347–372, 2008. doi:10.1016/j.tcs.2008.06.001.

11 Marcelo P. Fiore, Eugenio Moggi, and Davide Sangiorgi. A fully abstract model for the π-calculus. *Inf. Comput.*, 179(1):76–117, 2002. doi:10.1006/inco.2002.2968.

12 André Joyal, Mogens Nielsen, and Glynn Winskel. Bisimulation from open maps. *Inf. Comput.*, 127(2):164–185, 1996. doi:10.1006/inco.1996.0057.

13 Ugo Dal Lago, Marc de Visme, Damiano Mazza, and Akira Yoshimizu. Intersection types and runtime errors in the pi-calculus. *PACMPL*, 3(POPL):7:1–7:29, 2019. doi:10.1145/3290320.

14 Damiano Mazza, Luc Pellissier, and Pierre Vial. Polyadic approximations, fibrations and intersection types. *PACMPL*, 2(POPL):6:1–6:28, 2018. doi:10.1145/3158094.

15 Paul-André Melliès. Categorical combinatorics of scheduling and synchronization in game semantics. *PACMPL*, 3(POPL):23:1–23:30, 2019. doi:10.1145/3290336.

16 Massimo Merro. *Locality in the π-calculus and applications to distributed objects*. PhD thesis, École Nationale Supérieure des Mines de Paris, 2000.

17 Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996. doi:10.1017/S096012950007002X.

18 John Power and Edmund Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 7(5):453–468, 1997. doi:10.1017/S0960129597002375.

19 Ken Sakayori and Takeshi Tsukada. A categorical model of an $\mathbf{i}/\mathbf{o}$-typed π-calculus. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, pages 640–667, 2019. doi:10.1007/978-3-030-17184-1_23.

20 Ian Stark. A fully abstract domain model for the π-calculus. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 36–42, 1996. doi:10.1109/LICS.1996.561301.

21 Takeshi Tsukada, Kazuyuki Asada, and C.-H. Luke Ong. Generalised species of rigid resource terms. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12, 2017. doi:10.1109/LICS.2017.8005093.

22 Takeshi Tsukada and C.-H. Luke Ong. Plays as resource terms via non-idempotent intersection types. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 237–246, 2016. doi:10.1145/2933575.2934553.

23 Glynn Winskel and Mogens Nielsen. Presheaves as transition systems. In *Partial Order Methods in Verification, Proceedings of a DIMACS Workshop, Princeton, New Jersey, USA, July 24-26, 1996*, pages 129–140, 1996. doi:10.1090/dimacs/029/08.

24 Nobuko Yoshida. Minimality and separation results on asynchronous mobile processes – representability theorems by concurrent combinators. *Theor. Comput. Sci.*, 274(1-2):231–276, 2002. doi:10.1016/S0304-3975(00)00310-8.

A Compact closed Freyd category and π_F-theory

We briefly review the correspondence between the π_F-calculus and compact closed Freyd category originally proposed in [19] to make this paper self-contained.

▶ **Definition 21** (Compact closed Freyd category [19]). *A* compact closed Freyd category *is a Freyd category [18] $J\colon \mathcal{C} \to \mathcal{K}$ such that (1) $\mathcal{K}$ is compact closed, and (2) J has the (chosen) right adjoint $I \Rightarrow (-)\colon \mathcal{K} \to \mathcal{C}$.*

An equivalence $\mathcal{E}$ is a π_F*-theory* if it is closed under the following rules. Each rule has implicit assumptions that the both sides of the equation are well-sorted processes.

$$\frac{a \notin \mathbf{fn}(P, C) \qquad \bar{a} \notin \mathbf{bn}(C)}{\Gamma \vdash (\boldsymbol{\nu}\bar{a}a)(!a(\vec{x}).P \mid C[\bar{a}\langle\vec{y}\rangle]) = (\boldsymbol{\nu}\bar{a}a)(!a(\vec{x}).P \mid C[P\{\vec{y}/\vec{x}\}])}\ (\textsc{E-Beta})$$

$$\frac{a, \bar{a} \notin \mathbf{fn}(P)}{\Gamma \vdash (\boldsymbol{\nu}\bar{a}a)!a(\vec{y}).P = \mathbf{0}}\ (\textsc{E-GC}) \qquad \frac{\bar{a}, a \notin \mathbf{fn}(\bar{c}\langle\vec{x}\rangle)}{\Gamma \vdash \bar{c}\langle\vec{x}\rangle = (\boldsymbol{\nu}\bar{a}a)(a \hookrightarrow \bar{b} \mid \bar{c}\langle\vec{x}\{\bar{a}/\bar{b}\}\rangle)}\ (\textsc{E-FOut})$$

$$\frac{b, \bar{a} \notin \mathbf{fn}(P)}{\Gamma \vdash (\boldsymbol{\nu}\bar{a}a)(b \hookrightarrow \bar{a} \mid P) = P\{b/a\}}\ (\textsc{E-Eta})$$

$$\frac{P \equiv Q}{\Gamma \vdash P = Q}\ (\textsc{E-SCong}) \qquad \frac{\Delta \vdash P = Q \qquad C\colon \Gamma/\Delta\text{-context}}{\Gamma \vdash C[P] = C[Q]}\ (\textsc{E-Ctx})$$

Here a (Γ/Δ)-context is a context C such that $\Gamma \vdash C[P]$ for every $\Delta \vdash P$.

Any set Ax of equations-in-context has the minimum theory $Th(Ax)$ that contains Ax. We write $Ax \rhd \Delta \vdash P = Q$ if $(\Delta \vdash P = Q) \in Th(Ax)$. It should be noted that the original paper [19] only considers theory over the empty signature and that the π_F-calculus over the empty signature does not have the constant τ. The calculus defined in this paper is a π_F-calculus defined over the signature with a single constant $\tau : \mathbf{ch}^i[]$.

The important property that has been used in the body of this paper is that the term model $Cl(Ax)$ is a compact closed Freyd category for every set of non-logical axioms Ax [19, Theorem 3]. Given a set Ax of non-logical axioms, the term model $Cl(Ax)$ is defined as processes modulo $Ax \rhd \Delta \vdash P = Q$. Objects are list of types and a morphism (of the compact closed category) from $\vec{T}$ to $\vec{S}$ is an equivalence class $[\vec{x} : \vec{T}, \vec{y} : \vec{S}^\perp \vdash P]$. The composition of morphisms is defined by "parallel composition + hiding". For morphisms $P : \vec{T} \to \vec{S}$ and $Q : \vec{S} \to \vec{U}$, i.e. processes such that $\vec{x} : \vec{T}, \vec{y} : \vec{S}^\perp \vdash P$ and $\vec{z} : \vec{S}, \vec{w} : \vec{U}^\perp \vdash Q$, their composite is $\vec{x} : \vec{T}, \vec{w} : \vec{U}^\perp \vdash (\boldsymbol{\nu}\vec{y}\vec{z})(P|Q)$. (See [19] for the full definition.)

B Supplementary materials for Section 4

B.1 Groupoid structure of types and type isomorphisms

As expected, the witness of type isomorphisms can be *composed* so that $\varphi_1\colon \rho_1 \cong \rho_2$ and $\varphi_2\colon \rho_2 \cong \rho_3$ implies $(\varphi_2 \circ \varphi_1)\colon \rho_1 \cong \rho_3$. Composition of witnesses are defined by:

$$\begin{aligned}
\mathbf{ch}^o_\alpha[\varphi'_1, \dots, \varphi'_n] \circ \mathbf{ch}^o_\alpha[\varphi_1, \dots, \varphi_n] &\stackrel{\text{def}}{=} \mathbf{ch}^o_\alpha[\varphi_1 \circ \varphi'_1, \dots, \varphi_n \circ \varphi'_n] \\
\mathbf{ch}^i_\alpha[\varphi'_1, \dots, \varphi'_n] \circ \mathbf{ch}^i_\alpha[\varphi_1, \dots, \varphi_n] &\stackrel{\text{def}}{=} \mathbf{ch}^i_\alpha[\varphi'_1 \circ \varphi_1, \dots, \varphi'_n \circ \varphi_n] \\
(\sigma_2, (\varphi'_i)_{i \in \mathbf{Nat}}) \circ (\sigma_1, (\varphi_i)_{i \in \mathbf{Nat}}) &\stackrel{\text{def}}{=} (\sigma_2\sigma_1, (\varphi'_{\sigma_1(i)} \circ \varphi_i)_{i \in \mathbf{Nat}})
\end{aligned}$$

Types and type isomorphisms forms a groupoid. That is, we can define the inverse operator $(-)^{-1}$ for witnesses of type isomorphisms and show that there is an identity $\mathrm{id}_\rho \colon \rho \cong \rho$ for every type ρ. The inverse operator $(-)^{-1}$ is defined by $(\mathbf{ch}^m_\alpha[\varphi_1, \dots, \varphi_n])^{-1} \stackrel{\text{def}}{=} \mathbf{ch}^m_\alpha[\varphi_1^{-1}, \dots, \varphi_n^{-1}]$ for $m \in \{i, o\}$ and $(\sigma, (\varphi_i)_{i \in \mathbf{Nat}})^{-1} \stackrel{\text{def}}{=} (\sigma^{-1}, (\varphi^{-1}_{\sigma^{-1}(i)})_{i \in \mathbf{Nat}})$.

B.2 Subject reduction/expansion and Proposition 11

This section outlines the proof of Proposition 11, which states the correspondence between execution sequences and linear approximations. The proposition is a consequence of the subject reduction/expansion lemma. The proof for Proposition 14 (which we omit) is similar; the only difference is that we also need to take the order $\trianglelefteq$ into account.

As usual, to prove the subject reduction we use a substitution lemma:

▶ **Lemma 22** (Substitution Lemma). *Suppose that* $\Gamma \sqcap x : (i, \rho) \vdash_\alpha p$, $\varphi \colon \rho' \cong \rho$ *and* $\Gamma \sqcap y : (j, \rho')$ *is defined. Then* $\Gamma \sqcap y : (j, \rho') \vdash_\alpha p\{\varphi \bullet y_j / x_i\}$.

The proof of this lemma is similar to that of the conventional substitution lemma, except for the fact that we need to take group actions into account. Similarly, the following lemma can be proved by induction on the structure of p.

▶ **Lemma 23.** *Let* $\Gamma \sqcap x : (i, \rho) \vdash_\alpha p$, $\varphi \colon \rho' \cong \rho$ *and assume that* $y_j \notin \mathbf{fn}(p)$. *Then* $p\{\varphi \bullet y_j / x_i\}\{\varphi^{-1} \bullet x_i / y_j\} = p$

Now the subject reduction/expansion lemmas, and similar lemmas for the τ-reduction can be stated as follows. We omit the proofs as they can be shown by standard arguments with the help of Lemma 22 and 23.

▶ **Lemma 24** (Subject reduction). *Assume that* $\Gamma \vdash_\alpha p$ *and* $p \xrightarrow{0} q$. *Then we have* $\Gamma \vdash_\alpha q$. *Moreover, if* $\Gamma^\natural \vdash p \sqsubset P$ *then there exists* Q *such that* $\Gamma^\natural \vdash q \sqsubset Q$ *and* $P \xrightarrow{0} Q$.

▶ **Lemma 25.** *Suppose that* $\Gamma \sqcap \tau : (i, \mathbf{ch}^i_\beta[]) \vdash_\alpha p$, $\beta \leq \gamma$ *for all* $\gamma \in \overline{\mathbf{lv}}(\Gamma)$ *and* $p \xrightarrow{\tau_i} q$. *Then we have* $\Gamma \vdash_\beta q$. *Moreover, if* $({}^\natural\Gamma) \sqcap \tau : \{i\} \vdash p \sqsubset P$, *then there exists* Q *such that* $\Gamma^\natural \vdash q \sqsubset Q$ *and* $P \xrightarrow{\tau} Q$.

▶ **Lemma 26** (Subject expansion). *Suppose that* $P \xrightarrow{0} P'$, $\Gamma \vdash_\alpha p'$ *and* $\Gamma^\natural \vdash p' \sqsubset P'$. *Then there exists* p *such that* $\Gamma \vdash_\alpha p$, $\Gamma^\natural \vdash p \sqsubset P$ *and* $p \xrightarrow{0} p'$.

▶ **Lemma 27.** *Suppose that* $P \xrightarrow{\tau} Q$, $\Gamma \vdash_\alpha q$ *and* $\Gamma^\natural \vdash q \sqsubset Q$. *For all* $i \notin \mathrm{dom}(\Gamma(\tau))$, *there exists* p *and* β *such that* $\Gamma \sqcap \tau : (i, \mathbf{ch}^i_\beta[]) \vdash_\alpha p$, $\Gamma^\natural \sqcap \tau : \{i\} \vdash p \sqsubset P$ *and* $p \xrightarrow{\tau_i} q$.

Now we are ready to prove Proposition 11.

▶ **Proposition 11.** *Let* $\tau : \mathbf{ch}^i[] \vdash P$, *i.e. let* P *be a process without any free names.*

1. *Suppose that* $\Gamma \vdash_\alpha p$ *and* $\Gamma^\natural \vdash p \sqsubset P$. *If* $p \xrightarrow{\pi} q$ *then we have* $(p \xrightarrow{\pi} q) \sqsubset (P \xrightarrow{\pi} Q)$ *for some* Q.
2. *Assume* $P \xrightarrow{\pi} Q$, $\Gamma \vdash_\alpha q$ *and* $\Gamma^\natural \vdash q \sqsubset Q$. *Then we have* $(p \xrightarrow{\pi} q) \sqsubset (P \xrightarrow{\pi} Q)$ *for some* p.

Proof. (Proof of 1.) Let us write $\mathbf{ch}^i_{\alpha_i}[]$ for $\Gamma(\tau)(i)$ when $i \in \mathrm{dom}(\Gamma(\tau))$. By the assumption that $p \xrightarrow{\pi} q$, there exists a sequence $p = p_0 \xrightarrow{l_1} \cdots \xrightarrow{l_n} p_n = q$. Let $\tau_{i_1} \dots \tau_{i_k}$ be the subword of π that is obtained by deleting 0 from π. Without loss of generality, we may assume that $\alpha_{i_1} < \cdots < \alpha_{i_k}$ and $\alpha_{i_k} < \alpha$ for all $\alpha \in \{\alpha_i \mid i \in \mathrm{dom}(\Gamma(\tau))\} \setminus \{\alpha_{i_1}, \dots, \alpha_{i_k}\}$;

if not we can always reannotate the levels appearing in Γ and use that type environment instead of Γ. Now suppose that $l_1 = \tau_{i_1}$. Then we can apply Lemma 25 to obtain P_1 such that $P_0 \xrightarrow{\tau} P_1$ and $\Gamma_1^{\natural} \vdash p_1 \sqsubset P_1$, where Γ_1 is the type environment that satisfy $\Gamma_1 \vdash_{\alpha_{i_1}} p_1$. If $l_1 = 0$ we can use Lemma 24 instead. By repeating this argument we obtain a sequence $P = P_0 \xrightarrow{l_1} \cdots \xrightarrow{l_n} P_n = Q$ that can be used to show $(p \xrightarrow{\pi} q) \sqsubset (P \xrightarrow{\pi} Q)$.

(Proof of 2.) Since $P \xrightarrow{\pi} Q$, we have $P = P_0 \xrightarrow{l_1} \cdots \xrightarrow{l_n} P_n = Q$, where $\pi = l_1 \dots l_n$. Let us consider the case where $l_n = \tau$. In this case we can appeal to Lemma 27 (if $l_n = 0$ we use Lemma 26). By Lemma 27, we have p_{n-1} such that (1) $p_{n-1} \xrightarrow{\tau} q$, (2) $\Gamma \sqcap \tau : (i, \mathbf{ch}^i_\beta[]) \vdash_\beta p_{n-1}$ and (3) $\Gamma^\natural \sqcap \tau : \{i\} \vdash p_{n-1} \sqsubset P_{n-1}$, for some index i such that $i \notin \mathrm{dom}(\Gamma(\tau))$ and some level β. By repeating the argument we obtain $p \xrightarrow{\pi} q$ with the desired property. ◀

C Supplementary materials for Section 5

C.1 Restriction of types and type isomorphisms

▶ **Definition 28** (Complete version of Definition 12). *Let A be a set of levels.* Restriction of types and intersection types *are inductively defined by:*

$$\mathbf{ch}^m_\alpha[\theta_1, \dots, \theta_n]{\restriction_A} \stackrel{\text{def}}{=} \begin{cases} \mathbf{ch}^m_\alpha[\theta_1{\restriction_A}, \dots, \theta_n{\restriction_A}] & (\textit{if } \alpha \in A) \\ \bullet & (\textit{otherwise}) \end{cases}$$

$$(\theta{\restriction_A})(i) \stackrel{\text{def}}{=} \theta(i){\restriction_A},$$

where $m \in \{i, o\}$.

Similarly, restriction of type isomorphisms *is defined by:*

$$\mathbf{ch}^m_\alpha[\varphi_1, \dots, \varphi_n] \stackrel{\text{def}}{=} \begin{cases} \mathbf{ch}^m_\alpha[\varphi_1{\restriction_A}, \dots, \varphi_n{\restriction_A}] & (\textit{if } \alpha \in A) \\ \mathrm{id}_\bullet & (\textit{otherwise}) \end{cases}$$

$$(\sigma, (\varphi_i)_{i \in \mathbf{Nat}}){\restriction_A} \stackrel{\text{def}}{=} (\sigma, (\varphi_i{\restriction_A})_{i \in \mathbf{Nat}})$$

where $m \in \{i, o\}$. We write $\rho' <: \rho$ (resp. $\theta' <: \theta$) if $\rho' = \rho{\restriction_A}$ (resp. $\theta' = \theta{\restriction_A}$) for some $A \subseteq \mathcal{A}$ and $\varphi' <: \varphi$ if $\varphi' = \varphi{\restriction_A}$ for some $A \subseteq \mathcal{A}$.

C.2 Overview for the proof of Theorem 17

Here we briefly explain how to show that $[\![P]\!](-,-)$ is a presheaf (Theorem 17). Since Theorem 17, which says that $[\![P]\!]$ is a presheaf, is an immediate consequence of Proposition 16, the main goal of this section is to sketch the proof of Proposition 16:

▶ **Proposition 16.** *Assume that $\Gamma \vdash_\alpha p$, p is terminable and $(\Gamma', \alpha) <: (\Gamma, \alpha)$. Then there is a unique (up to $\equiv_0$) linear process that satisfy $q \trianglelefteq p$ and $\Gamma' \vdash_\alpha q$.*

The proof of Proposition 16 proceeds by induction on the structure of the derivation of $\Gamma \vdash_\alpha p$. The non-trivial case is the case of ν-restriction because it is not clear how the type annotated to the $\boldsymbol{\nu}$ binder should be restricted. To handle this case, we use the following lemmas, which says that "how the annotated type should be restricted is determined by how the type environment is restricted".

▶ **Lemma 29.** *Suppose that $\Gamma \vdash_\alpha (\boldsymbol{\nu}[\langle \bar{a}_i, a_i \rangle]_{i \in \mathrm{dom}(\theta)})p$ and that $(\boldsymbol{\nu}[\langle \bar{a}_i, a_i \rangle]_{i \in \mathrm{dom}(\theta)})p$ is terminable. Then $\overline{\mathbf{lv}}(\theta) \subseteq \overline{\mathbf{lv}}(\Gamma) \cup \{\alpha\}$.*

▶ **Lemma 30.** *Let $(\boldsymbol{\nu}[\langle\bar{a}_i, a_i\rangle]_{i\in\mathrm{dom}(\theta)})p$ be a terminable process typed under Γ, i.e. $\Gamma \vdash_\alpha (\boldsymbol{\nu}[\langle\bar{a}_i, a_i\rangle]_{i\in\mathrm{dom}(\theta)})p$. Suppose that $(\boldsymbol{\nu}[\langle\bar{a}_i, a_i\rangle]_{i\in\mathrm{dom}(\theta')})q \trianglelefteq (\boldsymbol{\nu}[\langle\bar{a}_i, a_i\rangle]_{i\in\mathrm{dom}(\theta)})p$ and $\Gamma' \vdash_\alpha (\boldsymbol{\nu}[\langle\bar{a}_i, a_i\rangle]_{i\in\mathrm{dom}(\theta')})q$, where Γ' satisfies $\Gamma'(t)(i) <: \Gamma(t)(i)$ for all term t and index i.If there is a level β such that $\beta \in \overline{\mathbf{lv}}(\theta)$ but $\beta \notin \overline{\mathbf{lv}}(\theta')$, then there is a term t and an index i such that $\beta \in \Gamma(t)(i)$ and $\beta \notin \Gamma'(t)(i)$.*

Instead of giving a detailed proof of these lemmas, we look at an example.

▶ **Example 31.** Let us consider a well typed linear process

$$\tau : (0, \mathbf{ch}^i_\beta[]) \vdash_\gamma (\boldsymbol{\nu}[\langle\bar{b}_0, b_0\rangle_{\rho_b}])(\boldsymbol{\nu}[\langle\bar{a}_0, a_0\rangle_{\rho_a}])(\tau_0.\bar{a}_0\langle\langle b_0\rangle\rangle \mid a_0(\langle\bar{x}_0\rangle).\bar{x}_0\langle\langle\rangle\rangle \mid b_0(\langle\rangle))$$

where $\rho_b = \mathbf{ch}^o_\beta[]$ and $\rho_a = \mathbf{ch}^o_\alpha[(0, \rho_\beta)]$. The following figure shows the way to point a free name (or a constant τ_i) whose type contains the level $\beta \in \overline{\mathbf{lv}}(\rho_b)$. (In this case we can tell that the type for τ_0 contains β.)

$$(\boldsymbol{\nu}[\langle\bar{b}_0, b_0\rangle])(\boldsymbol{\nu}[\langle\bar{a}_0, a_0\rangle])(\tau_0.\bar{a}_0\,\langle\langle\bar{b}_0\rangle\rangle \mid a_0(\langle\bar{x}_0\rangle).\bar{x}_0\,\langle\langle\rangle\rangle \mid b_0(\langle\rangle))$$

Let us explain what the pointers mean. A pointer points to a name that must be "executed at the same time" with the name placed at the source of the pointer. We start from $\bar{b}_0$ because that is the name with type ρ_b. Since $\bar{b}_0$ is in an object position of an output via the name $\bar{a}_0$ and $\bar{a}_0$ is bound, we first look for the name that communicates with $\bar{a}_0$, which is a_0 in this case. Because $\bar{x}_0$ is the argument that corresponds to $\bar{b}_0$, the type for $\bar{x}_0$ must have the level β for its "outermost level". So now we have another name $\bar{x}_0$ whose type has β as the "outermost level", and the link from $\bar{b}_0$ to $\bar{x}_0$ is used to expresses this fact. Now we look for the place where $\bar{x}_0$ is actually used, this is expressed by the second link. Since $\bar{x}_0$ is guarded by a_0 we now know that a_0 must be executed at the same time as $\bar{x}_0$. Because $\bar{a}_0$ communicates with a_0, we know that $\bar{a}_0$ and a_0 must be executed simultaneously and thus we have a pointer from a_0 to $\bar{a}_0$. The output $\bar{a}_0$ is guarded by τ_0, so we know that τ_0 also happens at the same time. Because τ_0 is a constant we conclude that β appears in the type environment.

Lemma 29 can be proved by formalising the notion of pointer and generalising the above procedure.

Lemma 30 can be proved by showing that $\trianglelefteq$ does not create any "dangling pointer". That is if $q \trianglelefteq p$ and linear terms t_j, t_i appearing in p are linked by a pointer, then either t_j and t_i both appears in q or t_j and t_i do not appear in q. This follows from the definition of $\trianglelefteq$ and the way we add pointers. For example, let us consider the case where p is the linear process depicted above. The only process q such that $\bar{b}_0$ does not appear in q and $q \trianglelefteq p$ is $(\boldsymbol{\nu}[])(\boldsymbol{\nu}[])(\bot \mid \bot \mid \bot)$.

With the above lemmas it is straightforward to prove Proposition 16 by induction on the structure of the derivation of $\Gamma \vdash_\alpha p$ and Theorem 17 follows as a corollary of this proposition.